Review: Properties of Transactions

- A transaction has the following “ACID” properties:
  - **Atomicity**: either all of its changes take effect or none do
  - **Consistency preservation**: its operations take the database from one consistent state to another
  - **Isolation**: it is not affected by other concurrent transactions
  - **Durability**: once it completes, its changes survive failures

- To guarantee isolation, a DBMS has to prevent problematic interleavings like this one:

```
txn 1
  read balance1
  write(balance1 - 500)
  read balance2
  if (balance1+balance2 < min)
    write(balance1 - fee)

txn 2
  read balance1
  read balance2
  if (balance1+balance2 < min)
    write(balance1 - fee)
```

- We’ll now look at mechanisms that are used to do so.
Concurrency Control Goals

- The mechanisms used to prevent problematic interleavings are known as concurrency control mechanisms.
  - they control the actions of concurrent transactions

- Goals: ensure that the schedule of actions that results from a set of concurrent txns is:
  - serializable: equivalent to some serial schedule
  - recoverable: ordered so that the system can safely recover from a crash or undo an aborted transaction
    - need to ensure that a txn does not commit before a txn whose write it has read
  - cascadeless: ensure that an abort of one transaction does not produce a series of cascading rollbacks
    - need to prevent dirty reads

A Simple Lock-Based Scheme

- Use locks to control access to database elements.
  - lock pages, records, or possibly even entire tables

- We’ll start with a simple scheme:
  - one lock per data element
  - before accessing a data element A, a transaction T1 must first request and acquire the lock for A
    - we say that T1 “locks A”
  - if T2 already holds the lock for A, T1 must wait until T2 releases the lock
    - we say that T2 “unlocks A”
A Simple Lock-Based Scheme (cont.)

- We expand our notation for schedules:
  \( r_t(A) \) = transaction \( T_t \) reads \( A \)
  \( w_t(A) \) = transaction \( T_t \) writes \( A \)
  \( c_t \) = transaction \( T_t \) commits
  \( l_t(A) \) = transaction \( T_t \) requests a lock for \( A \)
  \( u_t(A) \) = transaction \( T_t \) unlocks \( A \)

- Example:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l(B); r(B) )</td>
<td>( l(A); r(A) )</td>
</tr>
<tr>
<td>( l(A); w(A) )</td>
<td>( r(A); w(A) )</td>
</tr>
<tr>
<td>( u(A); u(B) )</td>
<td>( u(A); u(B) )</td>
</tr>
</tbody>
</table>

A Simple Lock-Based Scheme (cont.)

- As necessary, the DBMS *denies* lock requests for data elements that are currently locked.
  - make the txn wait until the other txn releases the lock, and then grant the lock
  - we'll show a second request for the lock, even though the txn may not actually need to make one

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l(X) )</td>
<td>( l(X) )</td>
</tr>
<tr>
<td>( r(X) )</td>
<td>( denied; wait for T1 )</td>
</tr>
<tr>
<td>( w(X) )</td>
<td>( granted )</td>
</tr>
<tr>
<td>( u(X) )</td>
<td>( r(X) )</td>
</tr>
<tr>
<td></td>
<td>( u(X) )</td>
</tr>
</tbody>
</table>
Locking and Serializability

- Just having locks isn't enough to guarantee serializability.
- Example: our problematic schedule can still be carried out.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read balance1</td>
<td>read balance1</td>
</tr>
<tr>
<td>write(balance1 - 500)</td>
<td>read balance2</td>
</tr>
<tr>
<td>read balance2</td>
<td>if (balance1+balance2 &lt; min)</td>
</tr>
<tr>
<td>write(balance2 + 500)</td>
<td>write(balance1 - fee)</td>
</tr>
</tbody>
</table>

T1          | T2
---|---
\(l(bal1) ; r(bal1)\) | \(l(bal1) ; r(bal1)\)
\(w(bal1) ; u(bal1)\) | \(l(bal2) ; u(bal1)\)
\(l(bal2) ; r(bal2)\) | \(l(bal1) ; r(bal2)\)
\(w(bal2) ; u(bal2)\) | \(w(bal1)\)

Two-Phase Locking (2PL)

- To ensure serializability, systems use *two-phase locking (2PL)*.
- 2PL requires that *all* of a txn’s lock actions come before *all* its unlock actions.
- Two phases:
  - lock-acquisition phase (aka the "growing" phase):
    locks are acquired, but no locks are released
  - lock-release phase (aka the "shrinking" phase):
    once a lock is released, no new locks can be acquired
- Reads and writes can occur in both phases.
Two-Phase Locking (2PL) (cont.)

• In our earlier example, T1 does not follow the 2PL rule.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(bal1); r(bal1)</td>
<td>l(bal1); r(bal1)</td>
</tr>
<tr>
<td>w(bal1); u(bal1)</td>
<td>l(bal2); r(bal2)</td>
</tr>
<tr>
<td>l(bal2); r(bal2)</td>
<td>w(bal1)</td>
</tr>
<tr>
<td>w(bal2); u(bal2)</td>
<td>u(bal1); u(bal2)</td>
</tr>
</tbody>
</table>

2PL would prevent this interleaving. Why?

An Informal Argument for 2PL's Correctness

• 2PL always produces conflict serializable schedules.

• Consider schedules involving only two transactions. To get one that is not conflict serializable, we need:
  1) at least one conflict that requires T1 → T2
     • T1 operates first on the data item in this conflict
     • T1 must unlock it before T2 can lock it: u₁(A) .. l₂(A)
  2) at least one conflict that requires T2 → T1
     • T2 operates first on the data item in this conflict
     • T2 must unlock it before T1 can lock it: u₂(B) .. l₁(B)

• To get both 1 and 2, at least one of the txns would have to acquire a lock after it has already released a lock.
  • example: .. u₁(A) .. l₂(A) .. u₂(B) .. l₁(B) ..
  • this isn’t possible under 2PL
Deadlock

- Consider the following schedule:

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(B); r(B)</td>
<td>i(A); w(A)</td>
</tr>
<tr>
<td>l(A)</td>
<td>(B)</td>
</tr>
<tr>
<td>denied; wait for T2</td>
<td>denied; wait for T1</td>
</tr>
</tbody>
</table>

- This schedule produces **deadlock**.
  - T₁ is waiting for T₂ to unlock A
  - T₂ is waiting for T₁ to unlock B
  - neither can make progress

- We'll see later how to deal with deadlocks.

Shared vs. Exclusive Locks

- With only one type of lock, overlapping transactions can't read the same data item, even though two reads don't conflict.

- To get around this, use more than one **mode** of lock:
  - **shared locks** (also called **read locks**):
    - allow a txn to read a data item
    - multiple txns can hold a shared lock for the same data item at the same time
    - to acquire a shared lock for a data item D, no other txn may hold an exclusive lock for D
  - **exclusive locks** (also called **write locks**):
    - allow a txn to read or write a data item
    - only one txn can hold an exclusive lock for a given data item
    - to acquire an exclusive lock for a data item D, no other txn may hold *any* lock for D
Shared vs. Exclusive Locks (cont.)

- New operations for our schedules:
  \( sli(A) \) = transaction \( T_i \) requests a shared lock for \( A \)
  \( xli(A) \) = transaction \( T_i \) requests an exclusive lock for \( A \)

- Examples:

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( sli(B); r(B) )</td>
<td>( sli(B); r(B) )</td>
</tr>
<tr>
<td>( xl(C); r(C) )</td>
<td>( u(A); u(B) )</td>
</tr>
<tr>
<td>( w(C) )</td>
<td>( u(A); u(C) )</td>
</tr>
</tbody>
</table>

Without shared locks, \( T_2 \) would need to wait until \( T_1 \) unlocked \( B \).

Note: \( T_1 \) acquires an exclusive lock before reading \( C \). Why?

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>granted or denied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( xl(A); sl(B) )</td>
<td>( sl(A) )</td>
<td>granted or denied?</td>
</tr>
<tr>
<td>( w(A); u(A) )</td>
<td>( sl(B) )</td>
<td>granted or denied?</td>
</tr>
<tr>
<td></td>
<td>( xl(B) )</td>
<td>granted or denied?</td>
</tr>
</tbody>
</table>

What About Recoverability?

- 2PL alone does not guarantee recoverability, nor does it prevent cascading rollbacks.

- Example:

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( xl(A); r(A) )</td>
<td>( xl(A); w(A) )</td>
</tr>
<tr>
<td>( w(A); u(A) )</td>
<td>( xl(C); r(C) )</td>
</tr>
<tr>
<td></td>
<td>( u(C) )</td>
</tr>
<tr>
<td></td>
<td>( xli(A); w(C) )</td>
</tr>
</tbody>
</table>

2PL?

- recoverable? why or why not?
- cascadeless? why or why not?
Strict Locking

- To ensure that a schedule is recoverable and cascadeless, we need to also employ strict locking.
- make txns hold all exclusive locks until they commit or abort
- doing so prevents dirty reads, which means schedules are always recoverable and cascadeless

\[ \text{strict + 2PL = strict 2PL} \]

Rigorous Locking

- Under strict locking, it's possible to get something like this:
- "Rigorous locking requires txns to hold all locks until commit/abort."
- It guarantees that transactions commit in the same order as they would in the equivalent serial schedule.
- rigorous + 2PL = rigorous 2PL
Lock Upgrades

- It can be problematic to acquire an exclusive lock earlier than necessary.
  - example: top right
- Instead:
  - acquire a shared lock first
  - only acquire the exclusive lock when you need it.
  - example: bottom right
- Adding an exclusive lock to a shared lock for the same item is known as a *lock upgrade*.
  - need to wait if others hold shared locks for the item

![Lock Upgrade Example]

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>xl(A)</td>
<td>sl(A)</td>
</tr>
<tr>
<td>r(A)</td>
<td></td>
</tr>
<tr>
<td>VERY LONG</td>
<td></td>
</tr>
<tr>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>w(A)</td>
<td></td>
</tr>
<tr>
<td>u(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r(A) finally!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A)</td>
<td></td>
</tr>
<tr>
<td>r(A)</td>
<td></td>
</tr>
<tr>
<td>VERY LONG</td>
<td></td>
</tr>
<tr>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>xl(A)</td>
<td></td>
</tr>
<tr>
<td>w(A)</td>
<td></td>
</tr>
<tr>
<td>u(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r(A) right away! u(A)</td>
</tr>
</tbody>
</table>

Lock Compatibility Matrices

- Used to determine if a lock request should be granted.
- When there are only shared and exclusive locks, we get:

<table>
<thead>
<tr>
<th>mode of existing lock (by other txn)</th>
<th>mode requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shared exclusive</td>
</tr>
<tr>
<td>shared</td>
<td>yes no</td>
</tr>
<tr>
<td>exclusive</td>
<td>no no</td>
</tr>
</tbody>
</table>

- Check all rows that apply, since one txn may hold both a shared and exclusive lock for the same item (after an upgrade).
A Possible Problem with Lock Upgrades

- Upgrades can lead to deadlock:
  - two txns each hold a shared lock for an item
  - both txns attempt to upgrade their locks
  - each txn waits for the other to release its shared lock
  - deadlock!

- Example:

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th></th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(A)</td>
<td>sl(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xl(A)</td>
<td></td>
<td></td>
<td>denied; wait for T₂</td>
</tr>
<tr>
<td>denied;</td>
<td></td>
<td></td>
<td>denied; wait for T₁</td>
</tr>
<tr>
<td>wait for T₂</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Update Locks

- To avoid deadlocks from lock upgrades, some systems take the following approach:
  - don’t allow the upgrading of shared locks
  - provide a third lock mode known as an **update lock**
    - like a shared lock, it allows a txn to read an item
    - it *can* be upgraded to an exclusive lock
  - if read-only $\rightarrow$ acquire a shared lock
  - if read-modify-write $\rightarrow$ acquire an update lock for the read,
    (RMW) and upgrade it to exclusive for the write
Update Locks (cont.)

• Lock compatibility matrix with update locks:

<table>
<thead>
<tr>
<th>mode of existing lock</th>
<th>shared</th>
<th>exclusive</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>exclusive</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>update</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

• When there are one or more shared locks on an item, a txn can still acquire an update lock for that item.
  • allows for concurrency on the read portion of RMW txns

• There can't be more than one update lock on an item.
  • prevents deadlocks from upgrades

• If a txn holds an update lock on an item, other txns can't acquire other locks on that item.
  • prevents the RMW txn from waiting indefinitely to upgrade

Example of Using Update Locks

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
<th>Example (ul(A) = T, requests an update lock for A):</th>
</tr>
</thead>
<tbody>
<tr>
<td>sl(B); ul(C)</td>
<td>sl(A)</td>
<td>granted: the only existing lock on B is a shared lock</td>
</tr>
<tr>
<td>ul(B)</td>
<td>ul(B)</td>
<td>denied: there is already an update lock on C</td>
</tr>
<tr>
<td>u(B)</td>
<td>u(C)</td>
<td>denied: shared locks cannot be upgraded under this scheme</td>
</tr>
<tr>
<td></td>
<td>x(A)</td>
<td>granted: update locks can be upgraded if no other locks</td>
</tr>
<tr>
<td></td>
<td>xl(B)</td>
<td></td>
</tr>
</tbody>
</table>
Handling Deadlocks: Detection

- One way of handling deadlocks is to have the DBMS detect them and abort one of the transactions involved.
- To detect deadlocks, use a \textit{waits-for graph}.
  - the vertices are the transactions
  - an edge from $T_1 \rightarrow T_2$ means that $T_1$ is waiting for $T_2$ to release a lock
  - a cycle in the graph indicates a deadlock

- Example:

  \begin{tabular}{|c|c|c|}
    \hline
    $T_1$ & $T_2$ & $T_3$ \\
    \hline
    xi(A) & sl(B) & xl(C) \\
    xi(B) & sl(C) & xi(A) \\
    \text{denied; wait for T2} & \text{denied; wait for T3} & \text{denied; wait for T1} \\
    \hline
  \end{tabular}

  \begin{tikzpicture}
    \node (T1) at (0,0) {$T_1$};
    \node (T2) at (1,0) {$T_2$};
    \node (T3) at (1,-1) {$T_3$};
    \draw[arrow] (T1) -- (T2);
    \draw[arrow] (T2) -- (T3);
    \draw[arrow] (T3) -- (T1);
  \end{tikzpicture}

- Another Example

- Would the following schedule produce deadlock?
  \begin{itemize}
    \item $r_1(B); w_1(B); r_3(A); r_2(C); r_2(B); r_1(A); w_1(A); w_3(C); w_2(A); r_1(C); w_3(A)$
  \end{itemize}

- assume a lock for an item is acquired just before it is first needed
Handling Deadlocks: Prevention

- Another set of techniques attempt to prevent deadlocks.
  1. lock ordering: require that txns acquire locks in some order
     - example: the order defined by the keys
     - this prevents deadlocks that don’t involve lock upgrades
  2. timeouts: set a maximum time that txns are allowed to wait
     - if a txn waits longer, it *times out* and is aborted
     - this doesn’t completely prevent deadlocks, it just prevents them from lasting indefinitely!
  3. timestamps: assign a timestamp to a transaction when it begins
     - when T1 needs to wait for T2 to release a lock, decide what to do based on their timestamps
     - may abort T1 or T2 instead of making T1 wait
     - use a policy that prevents deadlocks
     - for more info, look up *wound-wait* and *wait-die*

Starvation

- *Starvation* occurs when a transaction never completes.
- Example:

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sl(A)</td>
<td>sl(B)</td>
<td>sl(B)</td>
</tr>
<tr>
<td></td>
<td>xl(B)</td>
<td>xl(A)</td>
<td>xl(A)</td>
</tr>
<tr>
<td></td>
<td>denied; wait for T2</td>
<td>denied; wait for T₁</td>
<td>denied; wait for T₁</td>
</tr>
<tr>
<td></td>
<td>deadlock – abort</td>
<td>deadlock</td>
<td>deadlock</td>
</tr>
<tr>
<td></td>
<td>sl(A)</td>
<td>xl(A)</td>
<td>xl(A)</td>
</tr>
<tr>
<td></td>
<td>xl(B)</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>denied; wait for T₃</td>
<td>u(A); u(B)</td>
<td>u(A); u(B)</td>
</tr>
<tr>
<td></td>
<td>deadlock – abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Most deadlock-handling schemes need an additional policy to prevent this.
Starvation (cont.)

- Starvation can also occur in the absence of deadlocks.
  - example:
    - T1 acquires a shared lock for A
    - T2 requests an exclusive lock for A and waits
    - T3 acquires a shared lock for A
    - T1 unlocks A (but T2 still needs to wait for T3!)
    - T4 acquires a shared lock for A
    - T3 unlocks A (but T2 still needs to wait for T4!)
    - ...

- What policy regarding the granting of lock requests would prevent starvation in the absence of deadlocks?

Locking in Tree Index Structures

- Using 2PL on internal nodes of a tree would be problematic, because there are only a small number of them at the higher levels of the tree.

- In particular, holding an exclusive lock on the root node would prevent any other transaction from accessing the entire tree!

- Instead, systems typically use lock coupling.
  - while holding a lock for page P, acquire a lock for P's child
  - if P's child is not full, release the lock for P
  - if P's child is full, we keep the lock for P, since a split might propagate back up to P
Phantom Phenomenon

- Consider the following interleaving:

  T1
  ```
  for all accts at branch 1
    read(acct.bal)
    total += acct.bal; count += 1
  write(total/count);
  
  for all accts at branch 2
  ```

  T2
  ```
  insert a new acct at branch 1
  insert a new acct at branch 2
  commit
  ```

- It may be allowed by 2PL, even though the result is not equivalent to either serial ordering.
  - if locking records, txns would be allowed; if locking pgs, they might be
  - a similar problem can happen when records are deleted

- The record for the new account at branch 1 is a phantom record.
  - should be seen by T1, but it isn’t

- Conflict serializability does not guarantee serializability in the face of insertions and deletions.

Phantom Phenomenon (cont.)

- How can we prevent interleavings like this one?

  T1
  ```
  for all accts at branch 1
    read(acct.bal)
    total += acct.bal; count += 1
  write(total/count);
  
  for all accts at branch 2
  ```

  T2
  ```
  insert a new acct at branch 1
  insert a new acct at branch 2
  commit
  ```

- Two cases to consider:
  1) there’s an index on the attribute involved the selection (in this case, the branch attribute)
     → use index locking: lock the index page(s) for the selected value(s) at start of txn
  2) there isn’t an index on that attribute
     → use a special per-table lock for queries that affect/are affected by phantom phenom.
Optimistic Concurrency Control

- Locking is *pessimistic*.
  - it assumes serializability will be violated
  - it prevent transactions from performing actions that *might* violate serializability
    - example:

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sl(B); r(B)</td>
<td>xl(A); w(A)</td>
</tr>
<tr>
<td></td>
<td>xl(B)</td>
<td></td>
</tr>
</tbody>
</table>

  - denied, because T₁ might read B again

- There are other approaches that are *optimistic*.
  - they assume serializability will be maintained
  - they only interfere with a transaction if it actually does something that violates serializability

- We’ll look at one such approach – one that uses timestamps.

Timestamp-Based Concurrency Control

- In this approach, the DBMS assigns timestamps to txns.
  - TS(T) = the timestamp of transaction T
  - the timestamps must be unique
  - TS(T₂) > TS(T₁) if and only if T₂ started *after* T₁
  - these timestamps are different than the ones used for deadlock prevention

- The system ensures that all operations are consistent with a serial ordering based on the timestamps.
  - only allows actions if they are consistent with a schedule in which the transactions execute instantaneously at the time at which they start
### Timestamp-Based Concurrency Control (cont.)

- Examples of actions that are not allowed:
  - example 1:
    
    | T₁  | T₂       |
    |-----|---------|
    | TS = 102  | TS = 100 |
    | w(A)    | r(C)    |
    |        | r(A)    |

  **not allowed.** data item A was written by T₁, which should execute after T₂ in a serial order based on their timestamps, and thus T₂ should not be able to see this value. We say that T₂’s read **is too late**.

- example 2:
    
    | T₃  | T₄       |
    |-----|---------|
    | TS = 209  | TS = 205 |
    | r(B)    | r(A)    |
    |        | w(B)    |

  **not allowed.** T₃ should execute after T₄ in a serial ordering. Thus, T₃ should have read this value of B, but instead it read the prior value. We say that T₄’s write **is too late**.

### Timestamp-Based Concurrency Control (cont.)

- When a txn attempts to perform an action that is inconsistent with a timestamp ordering:
  - the offending txn is aborted
  - it is restarted with a new, larger timestamp

- With a larger timestamp, the txn comes later in the equivalent serial ordering.
  - allows it to perform the offending operation

- Aborting the txn ensures that all of its actions will correspond to the new timestamp.
timestamps on data elements

- to determine if an action should be allowed, the dbms associates two timestamps with each data element:
  - read timestamp: \( \text{RTS}(A) = \text{the largest timestamp of any txn that has read A} \)
  - write timestamp: \( \text{WTS}(A) = \text{the largest timestamp of any txn that has written A} \)

timestamp rules for reads

- if t tries to read a, the system compares \( \text{TS}(T) \) and \( \text{WTS}(A) \).
  - if \( \text{TS}(T) < \text{WTS}(A) \), abort t and restart it
    - t comes before the txn that wrote a, so t shouldn't be able to see a's current value
    - t's read is too late (see our earlier example 1)
  - else allow the read
    - t comes after the txn that wrote a, so the read is ok
    - the system also updates \( \text{RTS}(A) \):
      \[
      \text{RTS}(A) = \max(\text{TS}(T), \text{RTS}(A))
      \]
Timestamp Rules for Writes

- If T tries to write A, the system compares TS(T) with both RTS(A) and WTS(A).
  - if TS(T) < RTS(A), abort T and restart it
    - T comes before the txn that read A, so that other txn should have read the value T wants to write
    - T’s write is too late (see example 2)
  - else if TS(T) < WTS(A), ignore the write and let T continue
    - T comes before the txn that wrote A, and thus T’s write should have come first
    - if T’s write had come first, it would have been overwritten, so we can ignore it
  - else allow the write
    - how should the system update WTS(A)?

Thomas Write Rule

- The policy of ignoring out-of-date writes is known as the Thomas Write Rule:
  …else if TS(T) < WTS(A), ignore the write and let T continue

- What if there is a txn that should have read A between the two writes? It’s still okay to ignore T’s write of A.
  - example:
    - TS(T) = 80, WTS(A) = 100.
      what if txn U with TS(U) = 90 is supposed to read A?
    - if U had already read A, Thomas write rule wouldn’t apply:
      - RTS(A) = 90
      - T would be aborted because TS(T) < RTS(A)
    - if U tries to read A after we ignore T’s write:
      - U will be aborted because TS(U) < WTS(A)
Example of Using Timestamps

• Would they prevent our problematic balance-transfer example?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>bal1</th>
<th>bal2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS</td>
<td>TS</td>
<td>RTS = WTS = 0</td>
<td>RTS = WTS = 0</td>
</tr>
<tr>
<td>350</td>
<td>375</td>
<td>RTS = 350</td>
<td>WTS = 350</td>
</tr>
<tr>
<td>r(bal1)</td>
<td>r(bal1); r(bal2)</td>
<td>RTS = 375</td>
<td>WTS = 375</td>
</tr>
<tr>
<td>w(bal1)</td>
<td>w(bal2)</td>
<td>RTS = 375</td>
<td>RTS: no change</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>denied: abort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

what’s the problem here?

Preventing Dirty Reads Using a Commit Bit

• To prevent dirty reads when using timestamps, we associate a commit bit with each data element A.
  • initially, A.commit is true
  • when txn T writes A:
    • set A.commit to false, since T hasn't committed yet
    • update WTS(A) as before
    • A.commit remains false until the most recent writer of A commits or aborts
  • When A.commit == false, we make a txn U wait if:
    • it would be allowed to read A
    • it tries to write A and its write would be ignored
    • its write may be needed if T is rolled back
  • We don’t make a txn wait if its write of A is allowed.
    • it becomes the most recent writer of A
Preventing Dirty Reads Using a Commit Bit (cont.)

- When a txn T commits:
  - set to true the commit bits of all data elements of which T is the most recent writer
  - allow waiting transactions to proceed

- When a txn T is rolled back:
  - restore the prior state (value and timestamps) of all data elements of which T is the most recent writer
  - set the commit bits of those elements based on whether the writer of the prior value has committed
  - make waiting txns try again

Example of Using Timestamps and Commit Bits

- The balance-transfer example would now proceed differently.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>bal1</th>
<th>bal2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS = 350</td>
<td>TS = 375</td>
<td>RTS = WTS = 0 c = true</td>
<td>RTS = WTS = 0 c = true</td>
</tr>
<tr>
<td>r(bal1)</td>
<td>r(bal1)</td>
<td>c = true</td>
<td>c = true</td>
</tr>
<tr>
<td>w(bal1)</td>
<td>r(bal1)</td>
<td>TS = 350 WTS = 350; c = false</td>
<td>RTS = 350 WTS = 350; c = false</td>
</tr>
<tr>
<td>r(bal2)</td>
<td>denied: wait</td>
<td>c = true</td>
<td>c = true</td>
</tr>
<tr>
<td>w(bal2)</td>
<td></td>
<td>TS = 375</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>and completes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another Example

- How will this schedule be executed?
  \[ r_1(B); r_2(B); w_1(B); w_3(A); w_2(A); w_3(B); \text{commit}_3; r_2(A) \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>RTS = WTS = 0</td>
<td>c = true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RTS = WTS = 0</td>
<td>c = true</td>
</tr>
</tbody>
</table>

Example 3

- How will this schedule be executed?
  \[ w_1(A); w_2(A); r_3(B); w_3(B); r_3(A); r_2(B); w_1(B); r_2(A) \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>RTS = WTS = 0</td>
<td>c = true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RTS = WTS = 0</td>
<td>c = true</td>
</tr>
</tbody>
</table>
Multiversion Timestamp Protocol

- To reduce the number of rollbacks, the DBMS can keep old versions of data elements, along with the associated timestamps.
- When a txn T tries to read A, it is given the version of A that it should read, based on the timestamps.
  - the DBMS never needs to roll back a read-only transaction!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>A(0)</th>
<th>A(105)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS = 105</td>
<td>TS = 101</td>
<td>RTS = WTS = 0</td>
<td>c = true; val = “foo”</td>
<td>RTS = 105</td>
</tr>
<tr>
<td>r(A)</td>
<td>w(A)</td>
<td>r(A): get A(0)</td>
<td>TS = 112</td>
<td>r(A) get A(105)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c = false; val = “bar”</td>
<td>c = true</td>
</tr>
</tbody>
</table>

Multiversion Timestamp Protocol (cont.)

- Because each write creates a new version, the WTS of a given version never changes.
- The DBMS maintains RTSs and commit bits for each version, and it updates them using the same rules as before.
- If txn T attempts to write A:
  - find the version of A that T should be overwriting (the one with the largest WTS < TS(T))
  - compare TS(T) with the RTS of that version
  - example: txn T (TS = 50) wants to write A
    - it should be overwriting A(0)
    - show we allow its write and create A(50)?

<table>
<thead>
<tr>
<th>A(0)</th>
<th>A(105)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS = 75</td>
<td>RTS = 0</td>
</tr>
</tbody>
</table>
Multiversion Timestamp Protocol (cont.)

- Let's say that a write would be ignored under the standard timestamp protocol. Should it be still be ignored?

- Versions can be discarded as soon as there are no active transactions that could read them.
  - can discard A(t1) if these two conditions hold:
    - there is another, later version, A(t2), with t2 > t1
    - there is no active transaction T with a TS < t2

Locking vs. Timestamps

- Advantages of timestamps:
  - txns spend less time waiting
  - no deadlocks

- Disadvantages of timestamps:
  - can get more rollbacks, which are expensive
  - may use somewhat more space to keep track of timestamps

- Advantages of locks:
  - only deadlocked txns are rolled back

- Disadvantages of locks:
  - unnecessary waits may occur
The Best of Both Worlds

- Some commercial systems use a combination of two-phase locking (2PL) and multiversion timestamping.
- Transactions that perform writes use 2PL.
- Multiple versions of data elements are maintained, with each write creating a new version, as in multiversion timestamping.
- Read-only transactions are allowed to read the appropriate version, as in multiversion timestamping.

Looking Ahead

- Recall the “ACID” properties:
  - Atomicity: either all of its changes take effect or none do
  - Consistency preservation: its operations take the database from one consistent state to another
  - Isolation: it is not affected by other concurrent transactions
  - Durability: once it completes, its changes survive failures
- We've finished our coverage of how the DBMS provides isolation.
- We'll look next at how the DBMS guarantees atomicity and durability.