Concurrent Control

- How to ensure serializability and recoverability?
- Lock-Based Protocols
  - Lock, 2PL
  - Lock Conversion
  - Lock Implementation
  - Deadlock
  - Multiple Granularity
- Other Protocols
  - Timestamp-Based, Validation-Based, Multiversion Schemes, Snapshot isolation
- Insert and Delete Operations
Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item.

- Data items can be locked in two modes:
  1. **exclusive** (X) *mode*. Data item can be both read as well as written. X-lock is requested using `lock-X` instruction.
  2. **shared** (S) *mode*. Data item can only be read. S-lock is requested using `lock-S` instruction.
**Lock-Based Protocols (Cont.)**

- **Lock-compatibility matrix**

<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.)

Example:

- $T_1$ transfers $50 from account B to account A
- $T_2$ displays the total amount of money in accounts A and B
- Cannot guarantee serializability

$T_1$:  
lock-X($B$); 
read ($B$); 
$B = B - 50$; 
write($B$); 
unlock($B$); 
lock-X($A$); 
read ($A$); 
$A = A + 50$; 
write($A$); 
unlock(X);

$T_2$:  
lock-S($A$); 
read ($A$); 
unlock($A$); 
lock-S($B$); 
read ($B$); 
unlock($B$); 
display($A + B$)
<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>concurrency-control manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-x ($B$)</td>
<td></td>
<td>grant-x ($B, T_1$)</td>
</tr>
<tr>
<td>read ($B$)</td>
<td>lock-s ($A$)</td>
<td>grant-s ($A, T_2$)</td>
</tr>
<tr>
<td>$B := B - 50$</td>
<td>read ($A$)</td>
<td>grant-s ($B, T_2$)</td>
</tr>
<tr>
<td>write ($B$)</td>
<td>unlock ($A$)</td>
<td></td>
</tr>
<tr>
<td>unlock ($B$)</td>
<td>lock-s ($B$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read ($B$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock ($B$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>display ($A + B$)</td>
<td>grant-x ($A, T_2$)</td>
</tr>
<tr>
<td>lock-x ($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read ($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A := A + 50$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write ($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock ($A$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Two-Phase Locking Protocol

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.

- **2PL**
  - (1) If a transaction T wants to read/modify an object, it first requests a shared/exclusive lock on the object.
  - (2) A transaction **cannot request additional locks** once it releases **any** lock.

- Phase 1: Growing Phase
- Phase 2: Shrinking Phase
The Two-Phase Locking Protocol

- **Strict two-phase locking**
  - (1) If a transaction T wants to read/modify an object, it first requests a shared/exclusive lock on the object.
  - (2) **All locks** held by a transaction are released when the transaction is completed.
---

### 2PL (Cont.)

- **Differences between 2PL and Strict 2PL**
  - **2PL**: unlock instructions do not need to appear at the end of the transaction
  - **Strict 2PL**: all unlock must appear at the end

<table>
<thead>
<tr>
<th>Lock-X(B)</th>
<th>Lock-X(B)</th>
<th>Lock-S(A)</th>
<th>Lock-S(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(B)</td>
<td>Read(B)</td>
<td>Read(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>B=B-50</td>
<td>B=B-50</td>
<td>Lock-S(B)</td>
<td>Lock-S(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Write(B)</td>
<td>Read(B)</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>Lock-X(A)</td>
<td>Lock-X(A)</td>
<td>Display(A+B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Read(A)</td>
<td>Unlock(B)</td>
<td>Unlock(A)</td>
<td>Display(A+B)</td>
</tr>
<tr>
<td>A=A+50</td>
<td>Read(A)</td>
<td>Unlock(B)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Write(A)</td>
<td>A=A+50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Write(A)</td>
<td>T4 (strict 2PL)</td>
<td>T4’ (2PL)</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>Unlock(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**T3 (strict 2PL)**  
**T3’ (2PL)**
Both 2PL and Strict 2PL disallow the following schedule

- Guarantee conflict serializability

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>
## 2PL and Strict 2PL

- 2PL allows the following schedule
  - Unrecoverable
- Strict 2PL disallows the following schedule
  - Recoverability

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>Abort</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
2PL - properties

- 2PL does NOT ensure freedom from deadlock

<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock-X(B)</td>
<td>Lock-S(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>B=B-50</td>
<td>Lock-S(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Lock-X(A)</td>
<td>Display(A+B)</td>
</tr>
<tr>
<td>Read(A)</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>A=A+50</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
</tbody>
</table>

T3

T4
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(X)</td>
<td>R(X)</td>
</tr>
<tr>
<td>W(X)</td>
<td>Commit</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

S1 region
-- not conflict serializable
-- not view serializable
-- not recoverable

<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(X)</td>
<td>R(X)</td>
</tr>
<tr>
<td>R(X)</td>
<td>Commit</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
</tr>
</tbody>
</table>

S3 region
-- conflict serializable
-- not recoverable
Implementation of Locking

- Lock and unlock requests are handled by the lock manager
- **Lock table** entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- **Lock upgrade**: transaction that holds a shared lock can be upgraded to hold an exclusive lock
The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests.

The lock table is usually implemented as an **in-memory hash table** indexed on the name of the data item being locked.

**Situations**

- a lock request arrives
- an unlock message comes
- transaction aborts

Keep a list of locks held by each transaction, to implement this efficiently.
Pitfalls of Lock-Based Protocols

- Consider the partial schedule

<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock-X(B)</td>
<td>Lock-X(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>Lock-S(A)</td>
</tr>
<tr>
<td>B = B-50</td>
<td>Read(A)</td>
</tr>
<tr>
<td>W(B)</td>
<td>Lock-S(B)</td>
</tr>
<tr>
<td>T3</td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Display(A+B)</td>
</tr>
<tr>
<td>Unlock(A)</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>T4</td>
<td></td>
</tr>
</tbody>
</table>

- Such a situation is called a **deadlock**.
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
Assign priorities based on timestamps.
- The lower the timestamp, the higher is the transaction’s priority.

Assume $Ti$ wants a lock that $Tj$ holds. Two policies are possible:
- **Wait-Die**: If $Ti$ has higher priority (or smaller timestamp), $Ti$ waits for $Tj$; otherwise $Ti$ aborts
- **Wound-wait**: If $Ti$ has higher priority (or smaller timestamp), $Tj$ aborts; otherwise $Ti$ waits

If a transaction re-starts, make sure it has its original timestamp
- Older transactions thus have precedence over newer ones
- Starvation is hence avoided.
  - **Starvation**: the same transaction is repeatedly rolled back due to deadlocks.
Deadlock prevention (Cont.)

- **Timeout-Based Schemes**
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Thus deadlocks are **not possible**
  - Simple to implement; but **starvation is possible**.
  - Difficult to determine good value of the timeout interval.
Deadlock Detection

- Create a *waits-for graph*:
  - Nodes are transactions
  - There is an edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock
- Periodically check for *cycles* in the waits-for graph
Example

- Schedule: R1(X) W2(X), W2(Y), W3(Y) W1(Y) Commit-T1, Commit-T2, Commit-T3

- Describe how the concurrency control mechanism handles the schedule.
  - Strict 2PL with timestamps used for deadlock prevention.
  - Strict 2PL with deadlock detection.
Example

- R1(X) W2(X), W2(Y), W3(Y) W1(Y) Commit-T1, Commit-T2, Commit-T3
- Note that we do not interchangeably use both Wait-Die or Wound-Wait
- Assume we use deadlock prevention (Wait-Die policy)
  - R1(X): T1 acquires shared-lock on X;
  - W2(X): When T2 asks for an exclusive lock on X, since T2 has a lower priority, it will be aborted; (wait-die)
  - W3(Y): T3 now gets exclusive-lock on Y;
  - W1(Y): When T1 also asks for an exclusive-lock on Y which is still held by T3, T1 will be blocked waiting since T1 has higher priority
  - T3 now finishes write, commits and releases all the lock;
  - T1 wakes up, acquires the lock, proceeds and finishes;
  - T2 now can be restarted successfully.
Example

- R1(X) W2(X), W2(Y), W3(Y) W1(Y) Commit-T1, Commit-T2, Commit-T3

- Strict 2PL with **deadlock detection**.
  - T1 gets a shared-lock on X;
  - T2 **blocks** waiting for an exclusive-lock on X; (No prevention)
    - Think what will happen if we apply deadlock prevention (wait-die/wound-wait)
  - T3 gets an exclusive-lock on Y;
  - T1 **blocks** waiting for an exclusive-lock on Y; (No prevention)
    - Think what will happen if we apply deadlock prevention (wait-die/wound-wait)
  - T3 finishes, commits and releases locks;
  - T1 wakes up, gets an exclusive-lock on Y, finishes up and releases lock on X and Y;
  - T2 now gets both an exclusive-lock on X and Y, and proceeds to finish.
  - No deadlock.

- R1(X), **W2(Y)**, **W2(X)**, W3(Y), W1(Y), Commit-T1, Commit-T2, Commit-T3
Deadlock Detection (Continued)

Example:

T1: S(A), R(A), S(B)
T2: X(B), W(B), X(C)
T3: S(C), R(C), X(A)
T4: X(B)
Deadlock Recovery

- When deadlock is detected, some transaction will have to rolled back (made a victim) to break deadlock.
  - Select a transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation
Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Should not have to decide!
- Data “containers” are nested:
- **Granularity of locking** (level in tree where locking is done):
  - fine granularity (lower in tree): high concurrency, high locking overhead
  - coarse granularity (higher in tree): low locking overhead, low concurrency
Solution: New Lock Modes, Protocol

- **Question**: $Ti$ asks a lock on page, while $Tj$ holds a lock on the file that this page belongs to.

- Allow transactions to lock at each level, but with a special protocol using new “intention” locks:
Intention Lock Modes

- Three additional lock modes with multiple granularity:
  - **intention-shared** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - **intention-exclusive** (IX): indicates explicit X locking at a lower level or shared locks.
  - **shared and intention-exclusive** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with X locks.

- Before locking an item, transaction must set “intention locks” on all its ancestors.
- For unlock, go from specific to general (i.e., bottom-up).
Compatibility Matrix with Intention Lock Modes

The compatibility matrix for all lock modes is:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>IX</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>SIX</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
Examples

- $T1$ scans $R$, and updates a few tuples:
  - $T1$ gets an SIX lock on $R$, then repeatedly gets an $S$ lock on tuples of $R$, and occasionally upgrades to $X$ on the tuples.

- $T2$ uses an index to read only part of $R$:
  - $T2$ gets an IS lock on $R$, and repeatedly gets an $S$ lock on tuples of $R$.

- $T3$ reads all of $R$:
  - $T3$ gets an $S$ lock on $R$.
  - OR, $T3$ could behave like $T2$; can use lock escalation to decide which.
- Concurrency control without locking
- Self-study all the later materials
Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction $T_i$ has time-stamp $TS(T_i)$, a new transaction $T_j$ is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
- System clock, logical counter
- The protocol maintains for each data $Q$ two timestamp values:
  - $W$-timestamp($Q$) is the largest time-stamp of any transaction that executed write($Q$) successfully.
  - $R$-timestamp($Q$) is the largest time-stamp of any transaction that executed read($Q$) successfully.
The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

Suppose a transaction \( T_i \) issues a read\((Q)\):

1. If \( TS(T_i) \leq W\)-timestamp\((Q)\), then \( T_i \) needs to read a value of \( Q \) that was already overwritten.
   - Hence, the read operation is rejected, and \( T_i \) is rolled back.
   - When \( T_i \) restarts, a new and larger timestamp is assigned to \( T_i \).
2. If \( TS(T_i) \geq W\)-timestamp\((Q)\), then the read operation is executed, and R-timestamp\((Q)\) is set to \( \max(R\text{-timestamp}(Q), TS(T_i)) \).
Suppose that transaction $T_i$ issues $\text{write}(Q)$.

1. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of $Q$ that $T_i$ is producing was needed previously, and the system assumed that that value would never be produced.
   - Hence, the $\text{write}$ operation is rejected, and $T_i$ is rolled back.

2. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then $T_i$ is attempting to write an obsolete value of $Q$.
   - Hence, this $\text{write}$ operation is rejected, and $T_i$ is rolled back.

3. Otherwise, the $\text{write}$ operation is executed, and $\text{W-timestamp}(Q)$ is set to $\text{TS}(T_i)$. 
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($Y$)</td>
<td>read ($Y$)</td>
<td>write ($Y$)</td>
<td>read ($X$)</td>
<td></td>
</tr>
<tr>
<td>read ($Z$)</td>
<td>write ($Y$)</td>
<td>write ($Z$)</td>
<td>read ($Z$)</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>write ($Z$)</td>
<td></td>
<td>read ($Z$)</td>
<td></td>
</tr>
<tr>
<td>read ($X$)</td>
<td>abort</td>
<td>write ($W$)</td>
<td></td>
<td>write ($Y$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abort</td>
<td></td>
<td>write ($Z$)</td>
</tr>
</tbody>
</table>
The timestamp-ordering protocol guarantees **serializability** since all the arcs in the precedence graph are of the form:

Thus, there will be no cycles in the precedence graph.

- Timestamp protocol **ensures freedom from deadlock** as no transaction ever waits.
- But the schedule may **not be cascade-free**, and may **not even be recoverable**.

```
T1    T2
--------
W(A)    R(A)
W(B)    W(B)
Commit
```
Correctness of Timestamp-Ordering Protocol (S.S)

- But the schedule may not be cascade-free, and may not even be recoverable.

```
T1          T2
------------
W(A)         
R(A)
W(B)
Commit
```

An Unrecoverable Schedule
Recoverability and Cascade Freedom (S.S)

- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability
Thomas’ Write Rule (S.S.)

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.

- When $T_i$ attempts to write data item $Q$, if $TS(T_i) < W$-timestamp($Q$), then $T_i$ is attempting to write an obsolete value of $\{Q\}$.
  - Rather than rolling back $T_i$ as the timestamp ordering protocol would have done, this $\{write\}$ operation can be ignored.

- Otherwise this protocol is the same as the timestamp ordering protocol.

- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.