More Transaction Processing
Background

• The concurrency control mechanisms studied to date – will work flawlessly, if there are no aborts.

• But, transactions need to be aborted for various reasons: failures/recovery, deadlock (as we’ll see later), etc.

• Aborted transactions cause problems in the non-aborted transactions (next slide).
Topics

• Cascading rollback, Recoverable schedules, Strict schedules

• Deadlocks
Concurrency control & recovery

Example:

\[ T_j \quad \cdots \quad T_i \]

\[ \vdots \quad \vdots \]

\[ W_j(A) \quad \vdots \quad R_i(A) \]

\[ \vdots \quad \text{Commit Ti} \]

\[ \vdots \]

\[ \text{Abort T}_j \quad \vdots \]

\[ \text{Bad!} \]

Cascading rollback
• Schedule is conflict serializable

• But, not recoverable (because we need to abort a committed transaction $T_i$).
• Need to make “final” decision for each transaction:
  - commit decision - system guarantees transaction will or has completed, no matter what
  - abort decision - system guarantees transaction will or has been rolled back (has no effect)
To model this, two new actions:

- \( C_i \) - transaction \( T_i \) commits
- \( A_i \) - transaction \( T_i \) aborts
Reads and writes precede commit or abort.

- If $C_i \in T_i$, then $r_i(A) < C_i$ and $w_i(A) < C_i$
- If $A_i \in T_i$, then $r_i(A) < A_i$ and $w_i(A) < A_i$

- Also, one of $C_i, A_i$ per transaction
Back to example:

\[
\begin{array}{c}
T_j \\
\vdots \\
W_j(A) \\
\vdots \\
Ci & \leftarrow \text{can we commit here?}
\end{array}
\]

\[
\begin{array}{c}
T_i \\
\vdots \\
R_i(A) \\
\vdots \\
\end{array}
\]
Definition

Ti reads from Tj in S (Tj ⇒_S Ti) if

(1) w_j(A) <_S r_i(A)

(2) a_j ≺_S r_i(A)  \quad (≺_S : \text{ does not precede})

(3) If w_j(A) <_S w_k(A) <_S r_i(A) then
    a_k <_S r_i(A)
Recoverable Schedules

S is recoverable if each transaction commits only after all transactions from which it read have committed.

Formally,

Schedule S is recoverable if whenever T_j \implies_{S} T_i and j \neq i and C_i \in S then C_j <_{S} C_i

The above ensures that committed transaction would never need to be aborted.
ACR Schedules

• Recoverable schedules may still result in cascading rollback of other uncommitted transactions.

• Schedule S is ACR, i.e., avoids cascading rollback, if each transaction in S may read only committed values (i.e., those written by committed transactions).
Strict 2PL Locking

- **One way** to achieve ACR (and recoverable) schedules through locking is:
  - Retain the “write” locks until the very end (i.e., unlock just before committing).
  - This is called **Strict** locking. [Strict-2PL locking is different]
  - Yields **Strict schedules**. Herein, essentially, each transaction reads-from and writes-into only committed values.

Himanshu Gupta
Recoverable, ACR, and Strict Schedules

\[ \text{T}_j \quad \text{T}_i \]

\[ W_j(A) \quad \vdots \quad C_j \quad R_i(A) \quad Ci \quad ACR \quad \text{Strict} \]

\[ R_i(A) \quad R_i(A) \quad Ri(A) \quad Wi(A) \]
Examples

• Recoverable:
  \[ w_1(A) \ w_1(B) \ w_2(A) \ r_2(B) \ c_1 \ c_2 \]

• Avoids Cascading Rollback:
  \[ w_1(A) \ w_1(B) \ w_2(A) \ c_1 \ r_2(B) \ c_2 \]

• Strict:
  \[ w_1(A) \ w_1(B) \ c_1 \ w_2(A) \ r_2(B) \ c_2 \]

Assumes \( w_2(A) \) is done without reading
Claims

1. A conflict-serializable schedule may not be recoverable (first slide).

2. Every strict schedule is ACR, and each ACR is recoverable.

3. A strict schedule may not be serializable. Example: 
   \[ r_1(a) \; w_2(a) \; w_2(b) \; c_2 \; w_1(b) \; c_1 \]

4. **Strict-2PL** schedules are produced when all the locks are released at the end.
   - All strict-2PL schedules are strict as well as conflict-serializable.
RC, ACR, and Strict Schedules

- Serializable
  - Conflict-serializable
    - Strict
      - Strict-2PL
        - Serial
Deadlocks

• **Wait-For Graph**
• **Prevention**
  - Resource ordering
• **Detection/Fixing**
  - Timeout
  - Wait-die
  - Wound-wait
Tool for Prevention/Detection

- Build **Wait-For Graph**
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim

![Diagram of wait-for graph]

- \( T_1 \) sends 2
dependences
- \( T_2 \) sends 3
dependences
- \( T_3 \) sends 4
dependences
- \( T_4 \) sends 5
dependences
- \( T_5 \) sends 6
dependences
- \( T_6 \) sends 7
dependences
- \( T_7 \) sends 8
dependences

\( T_1 \) and \( T_2 \) have cycles.
Prevention I: Resource Ordering

- Order all elements $A_1, A_2, \ldots, A_n$
- A transaction $T$ can lock $A_i$ after $A_j$ only if $i > j$

**Problem:** Ordered lock requests not realistic in most cases
Detection I: Timeout

• If transaction waits more than L secs, roll it back!
• Simple scheme
• Hard to select L
Detection II: Wait-die

• Transactions given a timestamp when they arrive …. $ts(T_i)$
• $T_i$ can only wait for $T_j$ if it is older than $T_j$ (i.e., $ts(T_i) < ts(T_j)$) ... else die.

• Restart with the original timestamp. Why?
Example:

\[ T_1 \quad (ts = 10) \]

\[ T_2 \quad (ts = 20) \]

\[ T_3 \quad (ts = 25) \]
Detection III: Wound-wait

- Transactions given a timestamp when they arrive … $ts(T_i)$
- $T_i$ wounds $T_j$ if $T_i$ is older than $T_j$ (i.e., $ts(T_i) < ts(T_j)$) .. else $T_i$ waits

“Wound”: $T_j$ rolls back and gives lock to $T_i$
Example:

T1
(ts = 25)

wait

wait

T2
(ts = 20)

wait

T3
(ts = 10)
Comparison

• Deadlock Detection: Reacts only when there is indeed a deadlock. However, they are difficult to implement.

• Wait-die vs. Wound-wait:
  ➢ If we assume that acquisition of a lock happens early on in a transaction, then wait-die approach kills many transactions that haven’t done much work. In contrast, wound-wait approach kills fewer transactions that may have already done a lot of work.