More on Transaction Processing

Topics:
- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Distributed transactions
- Long transactions (nested, compensation)

Concurrency Control & Recovery

Example:

<table>
<thead>
<tr>
<th>Tj</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Wj(A)</td>
<td>ri(A)</td>
</tr>
<tr>
<td></td>
<td>Commit Ti</td>
</tr>
<tr>
<td>Abort Tj</td>
<td></td>
</tr>
</tbody>
</table>

→ Non-Persistent Commit (Bad!)

Avoided by recoverable schedules
### Concurrency Control & Recovery

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>Tj</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wj(A)</td>
<td>ri(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wi(B)</td>
<td></td>
</tr>
<tr>
<td>Abort Tj</td>
<td>[Commit Ti]</td>
<td></td>
</tr>
</tbody>
</table>

**Cascading rollback (Bad!)**

- Schedule is conflict serializable
- Tj → Ti

But not recoverable

### Concurrency Control & Recovery

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>Tj</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wj(A)</td>
<td>ri(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wi(B)</td>
<td></td>
</tr>
<tr>
<td>Abort Tj</td>
<td>[Commit Ti]</td>
<td></td>
</tr>
</tbody>
</table>

**avoids-cascading-rollback (ACR) schedules**

- 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

Back To Example:

Definition

$T_i$ reads from $T_j$ in $S$ ($T_j \Rightarrow_s T_i$) if

1. $w_j(A) <_s r_i(A)$
2. $a_j \not<_s r_i(A)$ ($\not<$: does not precede)
3. If $w_j(A) <_s w_k(A) <_s r_i(A)$ then $a_k <_s r_i(A)$

Definition

Schedule $S$ is recoverable if whenever $T_j \Rightarrow_s T_i$ and $j \neq i$ and $C_i \in S$ then $C_j <_s C_i$
Note: in transactions, reads and writes precede commit or abort

- If $Ci \in Ti$, then $ri(A) < Ci$
  $wi(A) < Ci$
- If $Ai \in Ti$, then $ri(A) < Ai$
  $wi(A) < Ai$

* Also, one of $Ci, Ai$ per transaction

How to achieve recoverable schedules?

» With 2PL, hold write locks to commit (strict 2PL)

<table>
<thead>
<tr>
<th>Tj</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Wj(A)</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Cj</td>
<td>:</td>
</tr>
<tr>
<td>uj(A)</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>ri(A)</td>
</tr>
</tbody>
</table>
• S is recoverable if each transaction commits only after all transactions from which it read have committed.

• S avoids cascading rollback if each transaction may read only those values written by committed transactions.

Where are serializable schedules?

- 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:
  - $w_1(A) \, w_1(B) \, c_1 \, w_2(A) \, r_2(B) \, c_2$

S is strict if each transaction may read and write only items previously written by committed transactions.
**Deadlocks**

- **Detection**
  - Wait-For Graph
- **Prevention**
  - Resource Ordering
  - Timeout
  - Wait-Die
  - Wound-Wait

**Deadlock Detection**

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

![Deadlock Detection Diagram](image)

**Resource Ordering**

- Order all elements A1, A2, …, An
- A transaction T can lock Ai after Aj only if \( i > j \)

Problem: Ordered lock requests not realistic in most cases

**Timeout**

- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L
### Wait-Die

- Transactions given a timestamp when they arrive... $ts(T_i)$
- $T_i$ can only wait for $T_j$ if $ts(T_i) < ts(T_j)$...
  else die

### Example

- $T_1$ ($ts = 10$)
- $T_2$ ($ts = 20$)
- $T_3$ ($ts = 25$)

### Starvation with Wait-Die

- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

### Starvation with Wait-Die

- Resubmit with original timestamp
- Guarantees no starvation
  - Transaction with oldest $ts$ never dies
  - A transaction that dies will eventually have oldest $ts$ and will complete...
Second Example

One option:
T1 waits just for T3, transaction holding lock. But when T2 gets lock, T1 will have to die!

Another option:
T1 only gets A lock after T2, T3 complete, so T1 waits for both T2, T3 ⇒ T1 dies right away!

Yet another option:
T1 preempts T2, so T1 only waits for T3; T2 then waits for T3 and T1... ⇒ T2 may starve?

Note: ts between 20 and 25.
**Wound-Wait**

- Transactions given a timestamp when they arrive ... ts(Ti)
- Ti wounds Tj if ts(Ti) < ts(Tj)
  else Ti waits

“Wound”: Tj rolls back and gives lock to Ti

**Example**

```
T1 (ts = 25)  
T2 (ts = 20)  
T3 (ts = 10)  
```

**Starvation with Wound-Wait**

- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

**Second Example**

```
T1 (ts = 15)  
T2 (ts = 20)  
T3 (ts = 10)  
```

Note: ts between 10 and 20.
Second Example (cont’d)

One option:
T1 waits just for T3, transaction holding lock.
But when T2 gets lock, T1 waits for T2 and wounds T2.

\[
\begin{align*}
T1 & (ts = 15) \\
& \downarrow \text{wait}(A) \\
T2 & \downarrow \text{wait}(A) (ts = 20) \\
T3 & (ts = 10)
\end{align*}
\]

Second Example (cont’d)

Another option:
T1 only gets A lock after T2, T3 complete, so T1 waits for both T2, T3 \(\Rightarrow\) T2 wounded right away!

\[
\begin{align*}
T1 & (ts = 15) \\
& \downarrow \text{wait}(A) \\
T2 & \downarrow \text{wait}(A) (ts = 20) \\
T3 & (ts = 10)
\end{align*}
\]

Second Example (cont’d)

Yet another option:
T1 preempts T2, so T1 only waits for T3; T2 then waits for T3 and T1... \(\Rightarrow\) T2 is spared!

\[
\begin{align*}
T1 & (ts = 15) \\
& \downarrow \text{wait}(A) \\
& \downarrow \text{wait}(A) (ts = 20) \\
T2 & \downarrow \text{wait}(A) \\
T3 & (ts = 10)
\end{align*}
\]

User/Program Commands

Lots of variations, but in general
- Begin_work
- Commit_work
- Abort_work
Nested Transactions

User program:

\[\text{Begin\_work;}\]
\[\text{If results\_ok, then commit work} \]
\[\text{else abort\_work}\]

Parallel Nested Transactions

\[T_1: \begin{align*}
\text{begin\_work} \\
\text{parallel:} \\
T_{11}: \text{begin\_work} \\
& \text{commit\_work} \\
T_{12}: \text{begin\_work} \\
& \text{commit\_work} \\
& \text{commit\_work}
\end{align*}\]

Locking

What are we really locking?
Example

```
Ti
: Read record r1
: Read record r1
: do record locking
: Modify record r3
```

But Underneath

- If we lock all data involved in read of R1, we may prevent an update to R2 (which may require reorganization within block)

Solution: View DB At Two Levels

Top level: record actions
- record locks
- undo/redo actions — logical

  - e.g., Insert record(X,Y,Z)
    - Redo: insert(X,Y,Z)
    - Undo: delete

Low level: deal with physical details
- latch page during action
- (release at end of action)
Note: undo does not return physical DB to original state; only same logical state

- Insert R3
- Undo (delete R3)

Logging Logical Actions

- Logical action typically span one block (physiological actions)
- Undo/redo log entry specifies undo/redo logical action
- Challenge: making actions idempotent
  - Example (bad): redo insert ⇒ key inserted multiple times!

Solution: Add Log Sequence Number

Log record:
- LSN=26
- OP=insert(5,v2) into P
- ...

Still Have a Problem!

Make log entry for undo

T1 Del 4
T2 Ins 5
undo Del 4
Compensation Log Records

- Log record to indicate undo (not redo) action performed
- Note: Compensation may not return page to exactly the initial state

At Recovery: Example

Log:

<table>
<thead>
<tr>
<th>lsn=21</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>a1</td>
</tr>
<tr>
<td>p1</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>lsn=27</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>a2</td>
</tr>
<tr>
<td>p2</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>lsn=35</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>a2⁻¹</td>
</tr>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

What To Do With p2 (During T1 Rollback)?

- If lsn(p2)<27 then ... ?
- If 27 ≤ lsn(p2) < 35 then ... ?
- If lsn(p2) ≥ 35 then ... ?

Note: lsn(p2) is lsn of p copy on disk

Recovery Strategy

[1] Reconstruct state at time of crash
   - Find latest valid checkpoint, Ck, and let ac be its set of active transactions
   - Scan log from Ck to end:
     - For each log entry [lsn, page] do:
       - if lsn(page) < lsn then redo action
       - If log entry is start or commit, update ac
Recovery Strategy

[2] Abort uncommitted transactions
- Set ac contains transactions to abort
- Scan log from end to Ck:
  - For each log entry (not undo) of an ac transaction, undo action (making log entry)
  - For ac transactions not fully aborted, read their log entries older than Ck and undo their actions

Example: What To Do After Crash

Log:

<table>
<thead>
<tr>
<th>lsn</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>a1p1</td>
</tr>
<tr>
<td>27</td>
<td>a2p2</td>
</tr>
<tr>
<td>29</td>
<td>a3p3</td>
</tr>
<tr>
<td>31</td>
<td>a3'p3</td>
</tr>
<tr>
<td>35</td>
<td>a2'p2</td>
</tr>
</tbody>
</table>

Related idea: Sagas

- Long running activity: T₁, T₂, ... Tₙ
- Each step/transaction Tᵢ has a compensating transaction Tᵢ₋₁
- Semantic atomicity: execute one of
  - T₁, T₂, ... Tₙ
  - T₁, T₂, ... Tₙ₋₁ T₋₁ₙ₋₁, T₋₁ₙ₋₂, ... T₋₁₁ᵢ
  - T₁, T₂, ... Tₙ₋₂ T₋₁ₙ₋₂, T₋₁ₙ₋₃, ... T₋₁₁ᵢ
  - T₁, T₋₁₁ᵢ
  - nothing
Summary

- Cascading rollback
  Recoverable schedule
- Deadlock
  - Prevention
  - Detection
- Nested transactions
- Multi-level view