CS 245: Database System Principles

Notes 10: More TP

Hector Garcia-Molina

Sections to Skim:

• Section 18.8 [18.8]
• Sections 19.2 19.4, 19.5, 19.6 (none, i.e., read all Ch 19)
• [In the Second Edition, skip all of Chapter 20, and Sections 21.5, 21.6, 21.7, 22.2 through 22.7]

Chapter 19 [19] More on transaction processing

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

Concurrency control & recovery

Example: $T_i \quad T_j$

$W_j(A) \quad r(A)$

Commit $T_i$

Abort $T_j$

Non-Persistent Commit (Bad!)

avoided by recoverable schedules
Concurrency control & recovery

Example:

Example:

Example:

Example:

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

Concurrency control & recovery

Example:

Example:

Example:

Example:

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

• Schedule is conflict serializable
• Tj ——> Ti
• But not recoverable

• 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:

• Ci - transaction Ti commits
• Ai - transaction Ti aborts
Back to example:

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

Definition

Ti reads from Tj in S (Tj \Rightarrow_S T_i) if

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

Note: in transactions, reads and writes precede commit or abort

\( \Leftrightarrow \) If \( C_i \in T_i \), then \( r_i(A) < C_i \)
\( w_i(A) < C_i \)
\( \Leftrightarrow \) If \( A_i \in T_i \), then \( r_i(A) < A_i \)
\( w_i(A) < A_i \)

* Also, one of \( C_i, A_i \) per transaction

How to achieve recoverable schedules?

\[\star\] With 2PL, hold write locks to commit (strict 2PL)

\[
\begin{array}{c|c}
T_j & T_i \\
\vdots & \vdots \\
W_j(A) & ri(A) \\
\vdots & \vdots \\
C_j & \vdots \\
u_j(A) & \vdots
\end{array}
\]
With validation, no change!

- 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.

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

Examples
- Recoverable:
  - $w_1(A) \ w_2(B) \ r_2(B) \ c_1 \ c_2$
- Avoids Cascading Rollback:
  - $w_1(A) \ w_2(B) \ c_1 \ r_2(B) \ c_2$
- Strict:
  - $w_1(A) \ w_2(B) \ c_1 \ w_2(A) \ r_2(B) \ c_2$
Where are serializable schedules?

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

Resource Ordering
- Order all elements A₁, A₂, ..., Aₙ
- A transaction T can lock Aᵢ after Aⱼ only if i > j

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 \( \ldots \) \( ts(T_i) \)
- \( T_i \) can only wait for \( T_j \) if \( ts(T_i) < ts(T_j) \)
  \( \ldots \) else die

---

**Example:**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>(ts = 10)</td>
</tr>
<tr>
<td>T2</td>
<td>(ts = 20)</td>
</tr>
<tr>
<td>T3</td>
<td>(ts = 25)</td>
</tr>
</tbody>
</table>

---

**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:

requests A: wait for T2 or T3?

Note: ts between 20 and 25.

Second Example (continued):

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

Second Example (continued):

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

Second Example (continued):

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

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:
**Example:**

```
T1 (ts = 25) → wait → T2 (ts = 20) → wait → 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` requests `A`: wait for `T2` or `T3`?
- `T2` waits for `T3`
- `T3` waits for `T2` and wounds `T2`.

**Second Example (continued):**

- `T1` waits just for `T3`, transaction holding lock. But when `T2` gets lock, `T1` waits for `T2` and wounds `T2`.

**Second Example (continued):**

- Another option: `T1` only gets a lock after `T2`, `T3` complete, so `T1` waits for both `T2`, `T3` → `T2` wounded right away!

**Second Example (continued):**

- Yet another option: `T2` preempts `T3`, so `T2` only waits for `T3`; `T2` then waits for `T3` and `T2`... → `T2` is spared!
User/Program commands

Lots of variations, but in general
• Begin_work
• Commit_work
• Abort_work

Nested transactions

User program:

Begin_work;

If results_ok, then commit_work
else abort_work

Parallel Nested Transactions

T1: begin-work
parallel:
T11: begin_work
commit_work
T12: begin_work
commit_work
commit_work
Locking

What are we really locking?

Example:

\[
T_i : \\
\vdots \\
\text{Read record } r_1 \\
\vdots \\
\text{Read record } r_1 \\
\vdots \\
\text{Modify record } r_3 \\
\vdots
\]

do record locking

But underneath:

Disk pages

R3

R1

R2

record id

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

But underneath:

Disk pages

R3

R1

R2

record id

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

e.g., 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

Question

• How to deal with spanned record?

Solution: Add Log Sequence Number

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

Still Have a Problem!

T1 Del 4

T2 Ins 5
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>
<th>lsn=27</th>
<th>lsn=35</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>a1</td>
<td>a2</td>
<td>a2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<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 ... ?

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 \( C_k \):
  - 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 \( C_k \) and undo their actions

Example: What To Do After Crash

Log:

<table>
<thead>
<tr>
<th>chk pt</th>
<th>lsn=21</th>
<th>lsn=27</th>
<th>lsn=29</th>
<th>lsn=31</th>
<th>lsn=35</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>a1 p1</td>
<td>a2 p2</td>
<td>a3 p3</td>
<td>a3 p3</td>
<td>a2 p2</td>
<td>a2 p2</td>
</tr>
</tbody>
</table>

During Undo: Skip Undo's

Log:

<table>
<thead>
<tr>
<th>chk pt</th>
<th>lsn=21</th>
<th>lsn=27</th>
<th>lsn=29</th>
<th>lsn=31</th>
<th>lsn=35</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>a1 p1</td>
<td>a2 p2</td>
<td>a3 p3</td>
<td>a3 p3</td>
<td>a2 p2</td>
<td>a2 p2</td>
</tr>
</tbody>
</table>

Related idea: Sagas

- Long running activity: \( T_1, T_2, \ldots, T_n \)
- Each step/transaction \( T_i \) has a compensating transaction \( T_{i-1} \)
- Semantic atomicity: execute one of
  - \( T_1, T_2, \ldots, T_n \)
  - \( T_1, T_2, \ldots, T_{n-1}, T_{n-1}, T_{n-2}, \ldots, T_{i-1}, T_i \)
  - \( T_1, T_2, \ldots, T_{n-2}, T_{n-2}, T_{n-3}, \ldots, T_{i-1} \)
  - \( T_1, T_{i-1} \)
  - nothing

Summary

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