Chapter 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:

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

Cascading rollback (Bad!)

- 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:

- \( C_i \) - transaction \( T_i \) commits
- \( A_i \) - transaction \( T_i \) aborts

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

**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 \( C_i \in T_i \), then \( r_i(A) < C_i \)
  - \( w_i(A) < C_i \)
- 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?
With 2PL, hold write locks to commit (strict 2PL)

\[
\begin{array}{ll}
T_i & T_j \\
\vdots & \vdots \\
W_i(A) & \vdots \\
\vdots & \vdots \\
C_j & \vdots \\
u_i(A) & r_i(A)
\end{array}
\]

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.

Where are serializable schedules?

Examples

- Recoverable:
  \(-w_1(A) w_2(B) w_3(A) r_2(B) c_1 c_2\)

- Avoids Cascading Rollback:
  \(-w_1(A) w_2(B) w_3(A) c_1 r_2(B) c_2\)

- Strict:
  \(-w_1(A) w_2(B) c_1 w_3(A) r_2(B) c_2\)

Assumes \(w_3(A)\) is done without reading.
**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 $A_1, A_2, ..., A_n$
- A transaction $T$ can lock $A_i$ after $A_j$ only if $i > j$

Problem: Ordered lock requests not realistic in most cases

**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:**
- $T_1$ ($ts = 10$)
- $T_2$ ($ts = 20$)
- $T_3$ ($ts = 25$)
Second Example:

T1 \( (ts = 22) \)
- requests A: wait for T2 or T3?

Note: ts between 20 and 25.

T2 \( (ts = 20) \)

T3 \( (ts = 25) \)

Second Example (continued):

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

T1 \( (ts = 22) \)

T2 \( (ts = 20) \)

T3 \( (ts = 25) \)

Second Example (continued):

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

T1 \( (ts = 22) \)

T2 \( (ts = 20) \)

T3 \( (ts = 25) \)

Second Example (continued):

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

T1 \( (ts = 22) \)

T2 \( (ts = 20) \)

T3 \( (ts = 25) \)

Example:

T1 \( (ts = 25) \)

T2 \( (ts = 20) \)

T3 \( (ts = 10) \)

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
Second Example:

T1 (ts = 15) requests A: wait for T2 or T3?

Note: ts between 10 and 20.

T2 (ts = 20)

T3 (ts = 10)

Second Example (continued):

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

T1 (ts = 15)

T2 (ts = 20)

T3 (ts = 10)

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!

T1 (ts = 15)

T2 (ts = 20)

T3 (ts = 10)

Second Example (continued):

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

⇒ T2 is spared!

T1 (ts = 15)

T2 (ts = 20)

T3 (ts = 10)

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
Nested transactions

User program:
Begin_work;
  Begin_work;
    If results_ok, then commit work
    else (abort_work; try something else...)
      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:

T1:
  : Read record r1
  : do record locking
  : Read record r1
  : Modify record r3

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
- Challenge: making actions idempotent
  Example (bad): redo insert \( \Rightarrow \)
  key inserted multiple times!

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

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>
<tr>
<td>...</td>
<td>...</td>
<td>...</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

[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>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</td>
<td>a2</td>
<td>a3</td>
<td>a3</td>
<td>a2</td>
<td>a2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p3</td>
<td>p2</td>
<td>p2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</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</td>
<td>a2</td>
<td>a3</td>
<td>a3</td>
<td>a2</td>
<td>a2</td>
</tr>
<tr>
<td>p1</td>
<td>p2</td>
<td>p3</td>
<td>p3</td>
<td>p2</td>
<td>p2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Pointer to forward action

Pointer to previous T1 action
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
  - $T_1, T_2, \ldots, T_n$
  - $T_1, T_2, \ldots, T_{n-1}, T_{n-1}^{-1}, T_{n-2}^{-1}, \ldots, T_1^{-1}$
  - $T_1, T_2, \ldots, T_{n-2}, T_{n-2}^{-1}, T_{n-3}^{-1}, \ldots, T_1^{-1}$
  - $T_1, T_1^{-1}$
  - nothing

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