Transactional Information Systems:

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Gerhard Weikum and Gottfried Vossen

© 2002 Morgan Kaufmann
ISBN 1-55860-508-8

“Teamwork is essential. It allows you to blame someone else.” (Anonymous)
Part III: Recovery

- 11 Transaction Recovery
- 12 Crash Recovery: Notion of Correctness
- 13 Page-Model Crash Recovery Algorithms
- 14 Object-Model Crash Recovery Algorithms
- 15 Special Issues of Recovery
- 16 Media Recovery
- 17 Application Recovery
Chapter 14: Object-Model Crash Recovery

• 14.2 Overview of Redo-History Algorithms
• 14.3 Simple Algorithm for 2-Layered Systems
• 14.4 Enhanced Algorithm for 2-Layered Systems
• 14.5 Complete Algorithm for General Executions
• 14.6 Lessons Learned

“This we know. All things are connected.” (Chief Seattle)
Conceptual Overview of Redo-History Algorithms

- **Analysis pass**: as in page model
- **Redo pass**: page-level redo for efficiency
- **Undo pass**: needs to invoke inverse high-level operations

**Problems:**
- atomicity of high-level operations:
  how to deal with partial effects of high-level operations
- idempotence of high-level operations:
  how to detect and prevent duplicate executions
  in situations where winners can follow losers

**Solutions:**
- page-level undo for partial high-level operations
- create CLEs for high-level inverse operations
Example for Object-Model Crash Recovery

 incr(x)  incr(y)  incr(z)

 t_1  t_2  t_12  t_13

 r(p) w(p) r(q) w(q)  r(s) w(s) r(r) w(r)  r(s) w(s)

 L_0  L_1

crash
Chapter 14: Object-Model Crash Recovery

• 14.2 Overview of Redo-History Algorithms
• 14.3 Simple Algorithm for 2-Layered Systems
• 14.4 Enhanced Algorithm for 2-Layered Systems
• 14.5 Complete Algorithm for General Executions
• 14.6 Lessons Learned
Actions During Normal Operation

- Introduce separate logs for each layer, with separate instances of the log manager's data structures (e.g., log buffer)
- Maintain $L_0$ log for page writes on behalf of subtransactions along with subcommit log entries for redo of completed subtransactions and undo of incomplete subtransactions
- Maintain $L_1$ log for high-level inverse operations and transaction commit log entries
- Both logs make use of CLEs

Log buffer forcing necessary for:
- $L_0$ log before a dirty page can be flushed
- $L_1$ log upon transaction commit, with $L_0$ log forced beforehand for transaction redo guarantee
- $L_1$ log before $L_0$ log is forced for transaction undo guarantee
Execution of High-Level Operations

```
exec (op, transid, inputparams, ↑returnvalues, s):
    subbegin ( ) ↑subtransid;
    execute operation;
    newlogentry.LogSeqNo := s;
    newlogentry.ActionType := exec;
    newlogentry.TransId := transid;
    newlogentry.SubtransId := subtransid;
    newlogentry.UndoInfo :=
        information on the inverse operation and its parameters;
    newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
    ActiveTrans[transid].LastSeqNo := s;
    L1LogBuffer += newlogentry;
    subcommit (subtransid);
```
Simple 2-Level Crash Recovery Algorithm

- $L_0$ recovery first identifies winner subtransactions, performs redo for these
  and undo for incomplete subtransactions
- $L_1$ recovery then identifies loser transactions, performs undo by traversing the corresponding
  NextUndoSeqNo backward chains:
  - an inverse operation is initiated iff
    the corresponding forward subtransaction was a winner
  - inverse operation executions create CLEs in the $L_1$ log and
    are treated like subtransactions during normal operation

```markdown
restart ( ):
  $L_0$ analysis pass ( ) returns losers, winners, DirtyPages;
  $L_0$ redo pass ( );
  $L_0$ undo pass ( );
  $L_1$ analysis pass ( );
  $L_1$ undo pass ( );
```
Efficient Testing of Winner Subtransactions

**Problem:** L₁ undo step needs to be invoked iff the corresponding subtransaction is an L₀ winner

→ need efficient test without explicit L₀ winner list

**Solution:**
- Include L₀ subbegin LSN in L₁ log entry for high-level operation
- L₀ analysis pass should identify maximum subbegin LSN as a “survivor mark” and explicitly identifies loser subtransactions
- L₁ undo pass test “presence” of high-level operation fᵢⱼ as follows:
  - if subbegin LSN in L₁ log entry for fᵢⱼ is larger than survivor mark
    then fᵢⱼ must be a loser subtransaction
  - otherwise (i.e., there is L₀ evidence of fᵢⱼ), if fᵢⱼ is not a loser subtransaction then it must be a winner subtransaction
L₁ Undo Pass of Simple 2-Level Algorithm (1)

L₁ undo pass ( ):
ActiveTrans := empty;
for each t in L₁ losers do
    ActiveTrans += t;
    ActiveTrans[t].LastSeqNo := losers[t].LastSeqNo;
end /*for*/;
while there exists t in losers such that losers[t].LastSeqNo <> nil do
    nexttrans := TransNo in losers such that losers[nexttrans].LastSeqNo =
        max {losers[x].LastSeqNo | x in losers};
    nextentry := losers[nexttrans].LastSeqNo;
    if StableLog[nextentry].ActionType = compensation then
        if StableLog[nextentry].CompensatingSubtransId is in L₀ winners then
            losers[nexttrans].LastSeqNo :=
                StableLog[nextentry].NextUndoSeqNo else
            losers[nexttrans].LastSeqNo :=
                StableLog[nextentry].PreviousSeqNo;
        end /*if*/;
    end /*if*/;
end /*if*/;
L₁ Undo Pass of Simple 2-Level Algorithm (2)

if StableLog[nextentry]. ActionType = exec then
    if StableLog[nextentry]. SubtransId is in L₀ winners then
        subbegin ( );
        newlogentry.LogSeqNo := new sequence number;
        newlogentry.ActionType := compensation;
        newlogentry.PreviousSeqNo :=
            ActiveTrans[transid]. LastSeqNo;
        newlogentry.NextUndoSeqNo := nextentry. PreviousSeqNo;
        ActiveTrans[transid]. LastSeqNo :=
            newlogentry.LogSeqNo;
        LogBuffer += newlogentry;
        execute inverse operation
            according to StableLog[nextentry]. UndoInfo;
        subcommit ( );
    end /*if*/;
losers[nexttrans]. LastSeqNo :=
    StableLog[nextentry]. PreviousSeqNo;
end /*if*/;
L₁ Undo Pass of Simple 2-Level Algorithm (3)

if StableLog[nextentry].ActionType = begin
then
    newlogentry.LogSeqNo := new sequence number;
    newlogentry.ActionType := rollback;
    newlogentry.TransId := StableLog[nextentry].TransId;
    newlogentry.PreviousSeqNo :=
        ActiveTrans[transid].LastSeqNo;
    LogBuffer += newlogentry;
    ActiveTrans[transid] -= transid;
    losers -= transid;
end /*if*/;

end /*while*/;
force ( );
Example for Simple 2-Level Algorithm

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1: begin (t₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: incr (x, t₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: subbegin (t₁₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: write (p, t₁₁)</td>
<td>p: 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: write (q, t₁₁)</td>
<td>q: 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: subcommit (t₁₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: begin (t₂)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: incr (x, t₂)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: subbegin (t₂₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: write (p, t₂₁)</td>
<td>p: 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11: incr (y, t₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12: subbegin (t₁₂)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: write (s, t₁₂)</td>
<td>s: 13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14: flush (p)</td>
<td></td>
<td>p: 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15: write (r, t₂₁)</td>
<td>r: 15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16: flush (s)</td>
<td></td>
<td>s: 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17: subcommit (t₂₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18: commit (t₂)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19: write (r, t₁₂)</td>
<td>r: 19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20: subcommit (t₁₂)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21: incr (z, t₁)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22: subbegin (t₁₃)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23: write (s, t₁₃)</td>
<td>s: 23</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† SYSTEM CRASH †
<table>
<thead>
<tr>
<th>Sequence number: action</th>
<th>Cached changes [PageNo: SeqNo]</th>
<th>Stable Changes [PageNo: SeqNo]</th>
<th>Log entry added to $L_0$ log [LogSeqNo: action]</th>
<th>Log entry added to $L_1$ log [LogSeqNo: action]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESTART</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_0$ analysis pass: $L_0$ losers = {t_{13}}, $L_0$ winners = {t_{11}, t_{21}, t_{12}}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consider-redo (4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redo (5)</td>
<td>q: 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consider-redo (10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redo (15)</td>
<td>r: 15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redo (19)</td>
<td>r: 19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>redo (23)</td>
<td>s: 23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24: compensate (23)</td>
<td>s: 24</td>
<td>24: CLE (23), next=nil</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25: subrollback (t_{13})</td>
<td>25: subrollback (t_{13})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L_1</strong> analysis pass: $L_1$ losers = {t_1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consider-compensate (21, t_{13})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26: compensate (11, t_{12}) $\uparrow$ t_{14}</td>
<td>26: CLE (11, t_{12}, t_{14}), next = 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27: subbegin (t_{13})</td>
<td>27: subbegin (t_{14})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28: write (s, t_{14})</td>
<td>s: 28</td>
<td>28: write (s, t_{14})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29: write (r, t_{14})</td>
<td>r: 29</td>
<td>29: write (r, t_{14})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30: flush (r)</td>
<td>r: 29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31: subcommit (t_{14})</td>
<td>31: subcommit (t_{14})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32: flush (q)</td>
<td>q: 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33: compensate (2, t_{11}) $\uparrow$ t_{15}</td>
<td>33: CLE(2, t_{11}, t_{15}), next = nil</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\* SECOND SYSTEM CRASH \*
### SECOND RESTART

**L₀ analysis pass:** $L₀$ losers = \{t₁₃\}, $L₀$ winners = \{t₁₁, t₂₁, t₁₂, t₁₃, t₁₄\}

- consider-redo (4)
- consider-redo (5)
- consider-redo (10)
- consider-redo (13)
- consider-redo (15)
- consider-redo (19)

- redo (23)  
  - s: 23
- redo (24)  
  - s: 24
- redo (28)  
  - s: 28
- consider-redo (29)

- 34: subrollback (t₁₅)
- 34: subrollback (t₁₅)

**L₁ analysis pass:** $L₁$ losers = \{t₁\}

- 35: compensate (2, t₁₁) ↑ t₁₆
- 35: CLE (2, t₁₁, t₁₆), nex t= nil

- 36: subbegin (t₁₆)

- 37: write (p, t₁₆)  
  - p: 37
- 38: write (q, t₁₆)  
  - q: 38

- 39: subcommit (t₁₆)

- 40: rollback (t₁)
- 40: rollback (t₁)

**SECOND RESTART COMPLETE:** RESUME NORMAL OPERATION
Chapter 14: Object-Model Crash Recovery

- 14.2 Overview of Redo-History Algorithms
- 14.3 Simple Algorithm for 2-Layered Systems
- **14.4 Enhanced Algorithm for 2-Layered Systems**
- 14.5 Complete Algorithm for General Executions
- 14.6 Lessons Learned
Enhanced 2-Level Crash Recovery Algorithm

combine $L_0$ log and $L_1$ log into a single log
• simplifies log forcing: log buffer forcing as in page model
• simplifies state testing by $L_1$ undo:
  by creating the $L_1$ log entry for the inverse operation at the end
  of the subtransaction and interpreting it also as an $L_0$ subcommit,
  the $L_1$ undo pass does no longer need to to test for $L_0$ winners
• can combine two analysis passes into one
• can combine two undo passes into one
  by using the NextUndoSeqNo backward chain as follows:
  • an $L_0$ write log entry points to the preceding write
  • in the same subtransaction
  • the very first $L_0$ write log entry of a subtransaction points to
    the $L_1$ log entry of the preceding subtransaction
  • an $L_0$ or $L_1$ CLE points to the predecessor of the
    compensated action
combined $L_0/L_1$ log ...

... during normal operation

... continued during restart

NextUndoSeqNo Backward Chaining in Enhanced 2-Level Crash Recovery Algorithm
undo pass ( ):  
  ActiveTrans := empty; 
  for each t in losers do  
    ActiveTrans += t; 
    ActiveTrans[t].LastSeqNo := losers[t].LastSeqNo; 
  end /*for*/; 
  while there exists t in losers such that 
    losers[t].LastSeqNo <> nil do 
    nexttrans = TransNo in losers 
      such that losers[nexttrans].LastSeqNo = 
      max {losers[x].LastSeqNo | x in losers}; 
    nextentry := losers[nexttrans].LastSeqNo; 
    if StableLog[nextentry].ActionType = compensation then 
      losers[nexttrans].LastSeqNo := 
      StableLog[nextentry].NextUndoSeqNo; 
    end /*if*/;
if StableLog[nextentry].ActionType = write or full-write
then
  pageo := StableLog[nextentry].PageNo; fetch (pageo);
  if DatabaseCache[pageo].PageSeqNo
    >= nextentry.LogSeqNo then
    newlogentry.LogSeqNo := new sequence number;
    newlogentry.ActionType := compensation;
    newlogentry.PreviousSeqNo :=
      ActiveTrans[transid].LastSeqNo;
    newlogentry.NextUndoSeqNo := nextentry.PreviousSeqNo;
    newlogentry.RedoInfo :=
      inverse action of the action in nextentry;
    ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
    LogBuffer += newlogentry;
    read and write (StableLog[nextentry].PageNo)
      according to StableLog[nextentry].UndoInfo;
    DatabaseCache[pageo].PageSeqNo := newlogentry.LogSeqNo;
  end /*if*/;
losers[nexttrans].LastSeqNo :=
  StableLog[nextentry].NextUndoSeqNo;
end /*if*/;
if StableLog[nextentry].ActionType = exec then
    subbegin ( );
    execute inverse operation
        according to StableLog[nextentry].UndoInfo;
    newlogentry.LogSeqNo := new sequence number;
    newlogentry.ActionType := compensation;
    newlogentry.PreviousSeqNo :=
        ActiveTrans[transid].LastSeqNo;
    newlogentry.NextUndoSeqNo := nextentry.NextUndoSeqNo;
    ActiveTrans[transid].LastSeqNo :=
        newlogentry.LogSeqNo;
    LogBuffer += newlogentry;
    subcommit ( );
    losers[nexttrans].LastSeqNo :=
        StableLog[nextentry].NextUndoSeqNo;
end /*if*/;
if StableLog[nextentry].ActionType = begin then
    newlogentry.LogSeqNo := new sequence number;
    newlogentry.ActionType := rollback;
    newlogentry.TransId := StableLog[nextentry].TransId;
    newlogentry.PreviousSeqNo :=
        ActiveTrans[transid].LastSeqNo;
    LogBuffer += newlogentry;
    ActiveTrans[transid].losers -= transid;
end /*if*/;
end /*while*/;
force ( );
**Example for Enhanced 2-Level Algorithm**

<table>
<thead>
<tr>
<th>Sequence number: action</th>
<th>Cached changes [PageNo: SeqNo]</th>
<th>Stable Changes [PageNo: SeqNo]</th>
<th>Log entry added [LogSeqNo: action] [NextUndoSeqNo]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: begin ((t_1))</td>
<td></td>
<td></td>
<td>1: begin ((t_1)), next = nil</td>
</tr>
<tr>
<td>2: incr ((x, t_1))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: subbegin ((t_{11}))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: write ((p, t_{11}))</td>
<td>(p: 4)</td>
<td></td>
<td>4: write ((p, t_{11})), next = nil</td>
</tr>
<tr>
<td>5: write ((q, t_{11}))</td>
<td>(q: 5)</td>
<td></td>
<td>5: write ((q, t_{11})), next = 4</td>
</tr>
<tr>
<td>6: subcommit ((t_{11}))</td>
<td></td>
<td></td>
<td>6: incr(^{-1}) ((x, t_1)), next = nil</td>
</tr>
<tr>
<td>7: begin ((t_2))</td>
<td></td>
<td></td>
<td>7: begin ((t_2))</td>
</tr>
<tr>
<td>8: incr ((x, t_2))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: subbegin ((t_{21}))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10: write ((p, t_{21}))</td>
<td>(p: 10)</td>
<td></td>
<td>10: write ((p, t_{21})), next = nil</td>
</tr>
<tr>
<td>11: incr ((y, t_1))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12: subbegin ((t_{12}))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13: write ((s, t_{12}))</td>
<td>(s: 13)</td>
<td></td>
<td>13: write ((s, t_{12})), next = 6</td>
</tr>
<tr>
<td>14: flush ((p))</td>
<td>(p: 10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15: write ((r, t_{21}))</td>
<td>(r: 15)</td>
<td></td>
<td>15: write ((r, t_{21})), next = 10</td>
</tr>
<tr>
<td>16: flush ((s))</td>
<td>(s: 13)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17: subcommit ((t_{21}))</td>
<td></td>
<td></td>
<td>17: incr(^{-1}) ((x, t_2)), next = nil</td>
</tr>
<tr>
<td>18: commit ((t_2))</td>
<td></td>
<td></td>
<td>18: commit ((t_2))</td>
</tr>
<tr>
<td>19: write ((r, t_{12}))</td>
<td>(r: 19)</td>
<td></td>
<td>19: write ((r, t_{12})), next = 13</td>
</tr>
<tr>
<td>20: subcommit ((t_{12}))</td>
<td></td>
<td></td>
<td>20: incr(^{-1}) ((y, t_1)), next = 6</td>
</tr>
<tr>
<td>21: incr ((z, t_1))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22: subbegin ((t_{13}))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23: write ((s, t_{13}))</td>
<td>(s: 23)</td>
<td></td>
<td>23: write ((s, t_{13})), next = 20</td>
</tr>
</tbody>
</table>

**SYSTEML CRASH**
<table>
<thead>
<tr>
<th>Sequence number: action</th>
<th>Cached changes [PageNo: SeqNo]</th>
<th>Stable Changes [PageNo: SeqNo]</th>
<th>Log entry added [LogSeqNo: action] [NextUndoSeqNo]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESTART</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**analysis pass**: losers = \{t_1\}, LastSeqNo (t_1) = 23

- consider-redo (4)
- redo (5) \> q: 5
- consider-redo (10)
- consider-redo (13)
- redo (15) \> r: 15
- redo (19) \> r: 19
- redo (23) \> s: 23

24: compensate (23) \> s: 24  
24: CLE (23), next = 20

25: compensate (20, t_{12}) \uparrow t_{14}

26: subbegin (t_{14})

27: write (s, t_{14}) \> s: 27  
27: write (s, t_{14}), next = 20

28: write (r, t_{14}) \> r: 28  
28: write (r, t_{14}), next = 27

29: flush (r) \> r: 28

30: subcommit (t_{14}) \> 30: CLE (20, t_{12}, t_{14}), next = 6

31: flush (q) \> q: 5

32: compensate (6, t_{11}) \uparrow t_{15}

\* SECOND SYSTEM CRASH \*
### SECOND RESTART

<table>
<thead>
<tr>
<th>Analysis pass: losers = {t_{1}}, Last SeqNo (t_{1}) = 30</th>
</tr>
</thead>
<tbody>
<tr>
<td>consider-redo (4)</td>
</tr>
<tr>
<td>consider-redo (5)</td>
</tr>
<tr>
<td>consider-redo (10)</td>
</tr>
<tr>
<td>consider-redo (13)</td>
</tr>
<tr>
<td>consider-redo (15)</td>
</tr>
<tr>
<td>consider-redo (19)</td>
</tr>
<tr>
<td>redo (23) s: 23</td>
</tr>
<tr>
<td>redo (24) s: 24</td>
</tr>
<tr>
<td>redo (27) s: 27</td>
</tr>
<tr>
<td>consider-redo (28)</td>
</tr>
<tr>
<td>33: compensate (6, t_{11}) \uparrow t_{15}</td>
</tr>
<tr>
<td>34: subbegin (t_{15})</td>
</tr>
<tr>
<td>35: write (p, t_{15})                                      p: 35  35: write (p, t_{15}), next = 6</td>
</tr>
<tr>
<td>36: write (q, t_{15})                                      q: 36  36: write (q, t_{15}), next = 35</td>
</tr>
<tr>
<td>37: subcommit (t_{15})                                     37: CLE (6, t_{11}, t_{15}), next = nil</td>
</tr>
<tr>
<td>38: rollback (t_{1})                                       38: rollback (t_{1})</td>
</tr>
</tbody>
</table>

SECOND RESTART COMPLETE: RESUME NORMAL OPERATION
Theorem 14.1:
The enhanced 2-level crash recovery method, with 3 passes over the combined log, performs correct recovery.

Proof sketch:
The following invariant holds at each point of the undo pass:
\[ \forall \text{log sequence numbers } s \in \text{StableLog such that } s = \text{ActiveTrans}[t].\text{LastSeqNo for some loser transaction } t: \]
\[ \forall \text{operations } o \in \text{StableLog}: \]
\[ (o \text{ belongs to } t) \implies \]
\[ (o \text{ is reachable along } \text{ActiveTrans}[t].\text{NextUndoSeqNo} \iff o \in \text{CachedDatabase}) \]
Chapter 14: Object-Model Crash Recovery

• 14.2 Overview of Redo-History Algorithms
• 14.3 Simple Algorithm for 2-Layered Systems
• 14.4 Enhanced Algorithm for 2-Layered Systems
• 14.5 Complete Algorithm for General Executions
• 14.6 Lessons Learned
Lessons Learned

• The redo-history paradigm can be extended to object-model crash recovery.
• State-of-the-art algorithms are based on:
  • page-oriented redo of winners and losers
  • log entries of all levels in a single log, to facilitate a single undo pass
  • log entries for high-level operations are at the same time sub-commit log entries to ensure the operation atomicity
  • for undo, log entries of all levels are appropriately linked in the NextUndoSeqNo backward chain
  • during undo, CLEs are created to track progress and ensure idempotence
  • during undo, the execution of high-level inverse operations requires the creation of low-level redo log entries to ensure operation atomicity