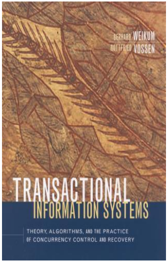
Transactional Information Systems:

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

Gerhard Weikum and Gottfried Vossen

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

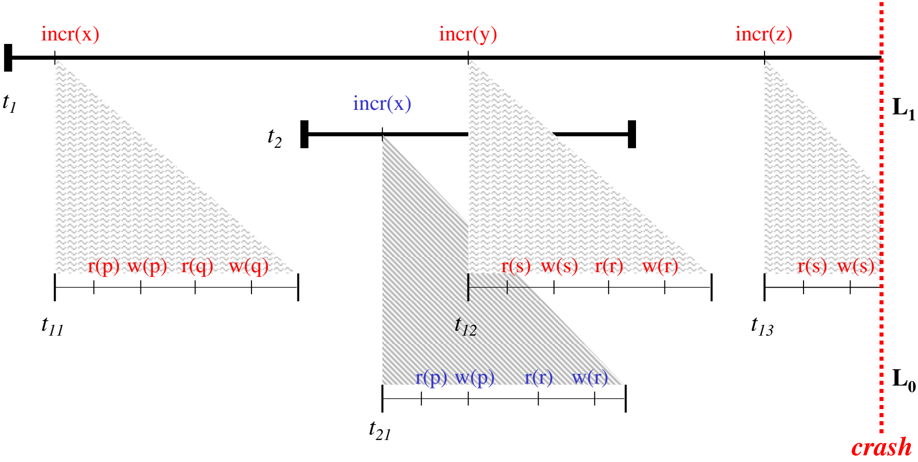
Problems:

- 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



Chapter 14: Object-Model Crash Recovery

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 } *page-model crash recovery for sub-transactions*
- 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

```
restart ( ):
   L0 analysis pass ( ) returns losers, winners, DirtyPages;
   L0 redo pass ( );
   L0 undo pass ( );
   L1 analysis pass ( );
   L1 undo pass ( );
```

Efficient Testing of Winner Subtransactions

Problem: L_1 undo step needs to be invoked iff the corresponding subtransaction is an L_0 winner
$\rightarrow$ need efficient test without explicit L_0 winner list

Solution:

- Include L_0 subbegin LSN in L_1 log entry for high-level operation
- L_0 analysis pass should identify maximum subbegin LSN as a “survivor mark” and explicitly identifies loser subtransactions
- L_1 undo pass test “presence” of high-level operation f_{ij} as follows:
    - if subbegin LSN in L_1 log entry for f_{ij} is larger than survivor mark then f_{ij} must be a loser subtransaction
    - otherwise (i.e., there is L_0 evidence of f_{ij}), if f_{ij} is not a loser subtransaction then it must be a winner subtransaction

L_1 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*/;
```

L_1 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*/;
```

L_1 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;
       losers -= transid;
    end /*if*/;

end /*while*/;
force ( );
```

Example for Simple 2-Level Algorithm

Sequence number: action	Cached changes [PageNo: SeqNo]	Stable Changes [PageNo: SeqNo]	Log entry added to L_0 log [LogSeqNo: action]	Log entry added to L_1 log [LogSeqNo: action]
1: begin (t_1)				1: begin (t_1)
2: incr (x, t_1)				2: $incr^{-1}$(x, t_1)
3: subbegin (t_{11})			3: subbegin (t_{11})	
4: write (p, t_{11})	p: 4		4: write (p, t_{11})	
5: write (q, t_{11})	q: 5		5: write (q, t_{11})	
6: subcommit (t_{11})			6: subcommit (t_{11})	
7: begin (t_2)				7: begin (t_2)
8: incr (x, t_2)				8: $incr^{-1}$ (x, t_2)
9: subbegin (t_{21})			9: subbegin (t_{21})	
10: write (p, t_{21})	p: 10		10: write (p, t_{21})	
11: incr (y, t_1)				11: $incr^{-1}$ (y, t_1)
12: subbegin (t_{12})			12: subbegin (t_{12})	
13: write (s, t_{12})	s: 13		13: write (s, t_{12})	
14: flush (p)		p: 10		
15: write (r, t_{21})	r: 15		15: write (r, t_{21})	
16: flush (s)		s: 13		
17: subcommit (t_{21})			17: subcommit (t_{21})	
18: commit (t_2)				18: commit (t_2)
19: write (r, t_{12})	r: 19		19: write (r, t_{12})	
20: subcommit (t_{12})			20: subcommit (t_{12})	
21: incr (z, t_1)				21: $incr^{-1}$ (z, t_1)
22: subbegin (t_{13})			22: subbegin (t_{13})	
23: write (s, t_{13})	s: 23		23: write (s, t_{13})	
⚡ SYSTEM CRASH ⚡				

actions during normal operation

Sequence number: action	Cached changes [PageNo: SeqNo]	Stable Changes [PageNo: SeqNo]	Log entry added to L_0 log [LogSeqNo: action]	Log entry added to L_1 log [LogSeqNo: action]
RESTART				
L_0 analysis pass: L_0 losers = $\{t_{13}\}$, L_0 winners = $\{t_{11}, t_{21}, t_{12}\}$				
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=nil	
25: subrollback (t_{13})			25: subrollback (t_{13})	
L_1 analysis pass: L_1 losers = $\{t_1\}$				
consider-compensate (21, t_{13})				
26: compensate (11, t_{12}) ↑ t_{14}				26: CLE (11, t_{12}, t_{14}), next = 2
27: subbegin (t_{14})			27: subbegin (t_{14})	
28: write (s, t_{14})	s: 28		28: write (s, t_{14})	
29: write (r, t_{14})	r: 29		29: write (r, t_{14})	
30: flush (r)		r: 29		
31: subcommit (t_{14})			31: subcommit (t_{14})	
32: flush (q)		q: 5		
33: compensate (2, t_{11}) ↑ t_{15}				33: CLE(2, t_{11}, t_{15}), next = nil
⚡ SECOND SYSTEM CRASH ⚡				

Sequence number: action	Cached changes [PageNo: SeqNo]	Stable Changes [PageNo: SeqNo]	Log entry added to L_0 log [LogSeqNo: action]	Log entry added to L_1 log [LogSeqNo: action]
SECOND RESTART				
L_0 analysis pass: L_0 losers = $\{t_{13}\}$, L_0 winners = $\{t_{11}, t_{21}, t_{12}, t_{13}, t_{14}\}$				
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 (t_{15})			34: subrollback (t_{15})	
L_1 analysis pass: L_1 losers = $\{t_1\}$				
35: compensate (2, t_{11}) ↑ t_{16}				35: CLE (2, t_{11}, t_{16}), nex t= nil
36: subbegin (t_{16})			36: subbegin (t_{16})	
37: write (p, t_{16})	p: 37			
38: write (q, t_{16})	q: 38			
39: subcommit (t_{16})			39: subcommit (t_{16})	
40: rollback (t_1)				40: rollback (t_1)
SECOND RESTART COMPLETE: RESUME NORMAL OPERATION				

Chapter 14: Object-Model Crash Recovery

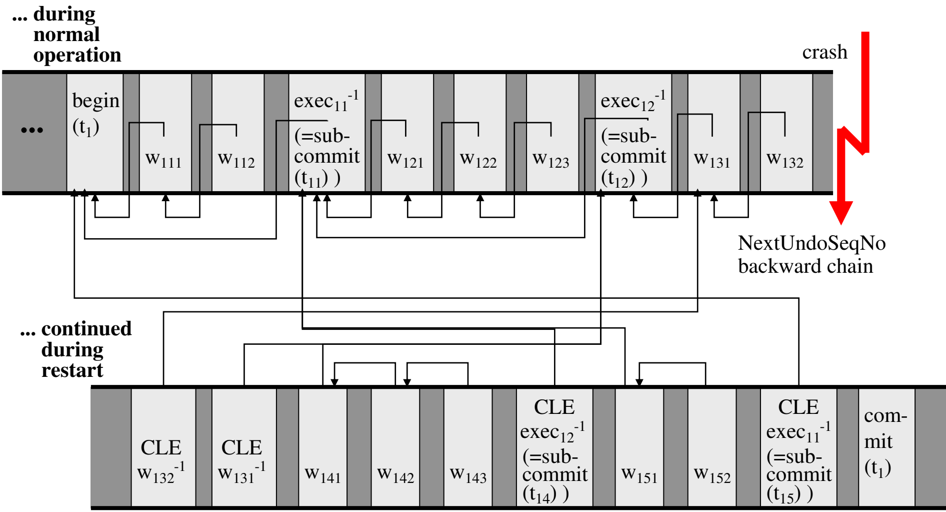
Enhanced 2-Level Crash Recovery Algorithm

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

NextUndoSeqNo Backward Chaining in Enhanced 2-Level Crash Recovery Algorithm

combined L_0/L_1 log ...



L_0/L_1 Undo Pass of Enhanced 2-Level Algorithm (1)

```
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*/;
```

L_0/L_1 Undo Pass of Enhanced 2-Level Algorithm (2)

```
if StableLog[nextentry].ActionType = write or full-write
then
   pageno := StableLog[nextentry].PageNo; fetch (pageno);
   if DatabaseCache[pageno].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[pageno].PageSeqNo := newlogentry.LogSeqNo;
   end /*if*/;
   losers[nexttrans].LastSeqNo :=
      StableLog[nextentry].NextUndoSeqNo;
end /*if*/;
```

L_0/L_1 Undo Pass of Enhanced 2-Level Algorithm (3)

```
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*/;
```

L_0/L_1 Undo Pass of Enhanced 2-Level Algorithm (4)

```
    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

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

Sequence number: action	Cached changes [PageNo: SeqNo]	Stable Changes [PageNo: SeqNo]	Log entry added [LogSeqNo: action] [NextUndoSeqNo]
RESTART			
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}) ↑ 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}) ↑ t_{15}			
⚡ SECOND SYSTEM CRASH ⚡			

Sequence number: action	Cached changes [PageNo: SeqNo]	Stable Changes [PageNo: SeqNo]	Log entry added [LogSeqNo: action] [NextUndo SeqNo]
SECOND RESTART			
analysis pass: losers = {t_1}, Last SeqNo (t_1) = 30			
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 (27)	s: 27		
consider-redo (28)			
33: compensate (6, t_{11}) ↑ t_{15}			
34: subbegin (t_{15})			
35: write (p, t_{15})	p: 35		35: write (p, t_{15}) , next = 6
36: write (q, t_{15})	q: 36		36: write (q, t_{15}), next = 35
37: subcommit (t_{15})			37: CLE (6, t_{11}, t_{15}), next = nil
38: rollback (t_1)			38: rollback (t_1)
SECOND RESTART COMPLETE: RESUME NORMAL OPERATION			

Correctness of Enhanced 2-Level Algorithm

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:
$\forall$ log sequence numbers s $\in$ StableLog such that
s = ActiveTrans[t].LastSeqNo for some loser transaction t:
$\forall$ operations o $\in$ StableLog:
(o belongs to t) $\Rightarrow$
(o is reachable along ActiveTrans[t].NextUndoSeqNo
$\Leftrightarrow$ o $\in$ 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