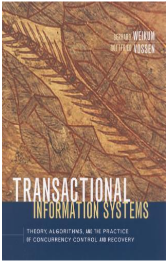
Transactional Information Systems:

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 13: Page-Model Crash Recovery Algorithms

- **13.2 Basic Data Structures**
- 13.3 Redo-Winners Paradigm
- 13.4 Redo-History Paradigm
- 13.5 Lessons Learned

“History is written by the winners.” (Alex Haley)

“History is a people's memory, and without a memory, man is demoted to the lower animals.” (Malcolm X)

Basic Data Structures for Crash Recovery (1)

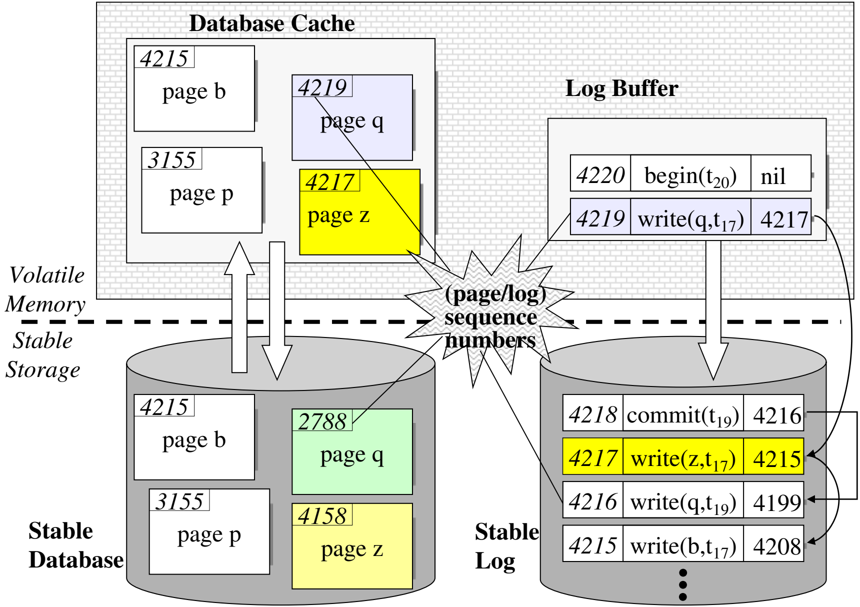
```
type Page: record of
        PageNo: identifier;
        PageSeqNo: identifier;
        Status: (clean, dirty) /* only for cached pages*/;
        Contents: array [PageSize] of char;
     end;
persistent var StableDatabase:
        set of Page indexed by PageNo;
var DatabaseCache:
        set of Page indexed by PageNo;
```

Basic Data Structures for Crash Recovery (2)

```
type LogEntry: record of
        LogSeqNo: identifier;
        TransId: identifier;
        PageNo: identifier;
        ActionType:(write, full-write, begin, commit, rollback);
        UndoInfo: array of char;
        RedoInfo: array of char;
        PreviousSeqNo: identifier;
     end;
persistent var StableLog:
        ordered set of LogEntry indexed by LogSeqNo;
var LogBuffer:
        ordered set of LogEntry indexed by LogSeqNo;
type TransInfo: record of
        TransId: identifier;
        LastSeqNo: identifier;
     end;
var ActiveTrans:
        set of TransInfo indexed by TransId;
```

Remark: log entries can be physical or physiological

Recall: (Log) Sequence Numbers



Correspondence of Data Structures and Abstract Model

0) action with sequence number $s \in$ StableLog
   $\Leftrightarrow$ LSN s is in StableLog

1) write action with sequence number s on page $p \in$ StableDatabase
   $\Leftrightarrow$ StableDatabase[p].PageSeqNo $\geq$ s

2) write action with sequence number s on page $p \in$ CachedDatabase
   $\Leftrightarrow$ DatabaseCache[p].PageSeqNo $\geq$ s $\vee$
   StableDatabase[p].PageSeqNo $\geq$ s

Typical implementation for 1) and 2):
   DatabaseCache[p].PageSeqNo :=
   max{s | there is a write action on p with sequence number s}

Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- **13.3 Redo-Winners Paradigm**
  - **13.3.1 Actions During Normal Operation**
  - 13.3.2 Simple Three-Pass Algorithm
  - 13.3.3 Enhanced Algorithm:
    Log Truncation, Checkpoints, Redo Optimization
  - 13.3.4 Complete Algorithm:
    Handling Transaction Aborts and Undo Completion
- 13.4 Redo-History Paradigm
- 13.5 Lessons Learned

Actions During Normal Operation (1)

```
write or full-write (pageno, transid, s):
   DatabaseCache[pageno].Contents := modified contents;
   DatabaseCache[pageno].PageSeqNo := s;
   DatabaseCache[pageno].Status := dirty;
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := write or full-write;
   newlogentry.TransId := transid;
   newlogentry.PageNo := pageno;
   newlogentry.UndoInfo := information to undo update
        (before-image for full-write);
   newlogentry.RedoInfo := information to redo update
        (after-image for full-write);
   newlogentry.PreviousSeqNo :=
         ActiveTrans[transid].LastSeqNo;
   ActiveTrans[transid].LastSeqNo := s;
   LogBuffer += newlogentry;
```

Actions During Normal Operation (2)

```
fetch (pageno):
   DatabaseCache += pageno;
   DatabaseCache[pageno].Contents :=
        StableDatabase[pageno].Contents;
   DatabaseCache[pageno].PageSeqNo :=
        StableDatabase[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;

flush (pageno):
   if there is logentry in LogBuffer
      with logentry.PageNo = pageno
   then force ( ); end /*if*/;
   StableDatabase[pageno].Contents :=
        DatabaseCache[pageno].Contents;
   StableDatabase[pageno].PageSeqNo :=
        DatabaseCache[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;

force ( ):
     StableLog += LogBuffer;
     LogBuffer := empty;
```

Actions During Normal Operation (3)

```
begin (transid, s):
   ActiveTrans += transid;
   ActiveTrans[transid].LastSeqNo := s;
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := begin;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := nil;
   LogBuffer += newlogentry;

commit (transid, s):
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := commit;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo :=
        ActiveTrans[transid].LastSeqNo;
   LogBuffer += newlogentry;
   ActiveTrans -= transid;
   force ( );
```

Correctness and Efficiency Considerations for Actions During Normal Operation

Theorem 13.1:
During normal operation, the redo logging rule, the undo logging rule, and the garbage collection rule are satisfied.

Forced log I/O is potential bottleneck during normal operation
$\rightarrow$ **group commit** for log I/O batching

Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- **13.3 Redo-Winners Paradigm**
  - 13.3.1 Actions During Normal Operation
  - **13.3.2 Simple Three-Pass Algorithm**
  - 13.3.3 Enhanced Algorithm:
    Log Truncation, Checkpoints, Redo Optimization
  - 13.3.4 Complete Algorithm:
    Handling Transaction Aborts and Undo Completion
- 13.4 Redo-History Paradigm
- 13.5 Lessons Learned

Overview of Simple Three-Pass Algorithm

- **Analysis pass:**
  determine start of stable log from master record
  perform forward scan
  to determine winner and loser transactions
- **Redo pass:**
  perform forward scan
  to redo all winner actions in chronological (LSN) order
  (until end of log is reached)
- **Undo pass:**
  perform backward scan
  to traverse all loser log entries in reverse chronological order
  and undo the corresponding actions

Simple Three-Pass Algorithm (1)

```
restart ( ):
   analysis pass ( ) returns losers;
   redo pass ( );
   undo pass ( );

analysis pass ( ) returns losers:
var losers: set of record
               TransId: identifier;
               LastSeqNo: identifier;
            end indexed by TransId;
   losers := empty;
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max do
       case StableLog[i].ActionType:
          begin: losers += StableLog[i].TransId;
                 losers[StableLog[i].TransId].LastSeqNo := nil;
          commit: losers -= StableLog[i].TransId;
          full-write: losers[StableLog[i].TransId].LastSeqNo := i;
      end /*case*/;
   end /*for*/;
```

Simple Three-Pass Algorithm (2)

```
redo pass ( ):
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max
   do
       if StableLog[i].ActionType = full-write and
          StableLog[i].TransId not in losers
       then
          pageno = StableLog[i].PageNo;
          fetch (pageno);
          full-write (pageno)
             with contents from StableLog[i].RedoInfo;
       end /*if*/;
   end /*for*/;
```

Simple Three-Pass Algorithm (3)

```
undo pass ( ):
   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 = full-write
       then
          pageno = StableLog[nextentry].PageNo;
          fetch (pageno);
          full-write (pageno)
             with contents from StableLog[nextentry].UndoInfo;
          losers[nexttrans].LastSeqNo :=
             StableLog[nextentry].PreviousSeqNo;
       end /*if*/;
   end /*while*/;
```

Correctness of Simple Three-Pass Algorithm

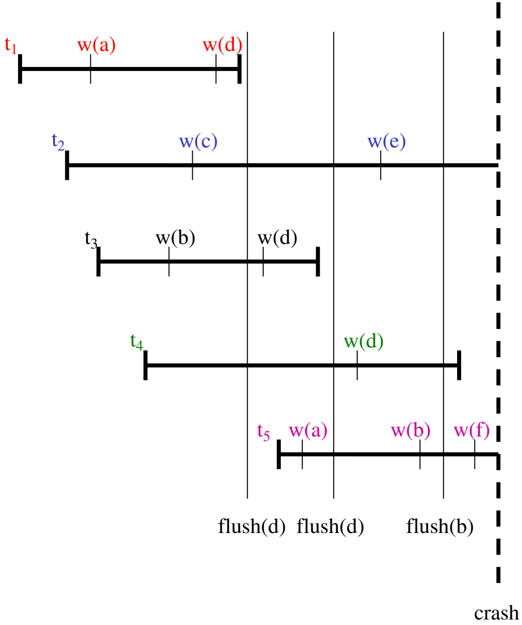
Theorem 13.2:
When restricted to full-writes as data actions,
the simple three-pass recovery algorithm performs correct recovery.

Proof sketch:

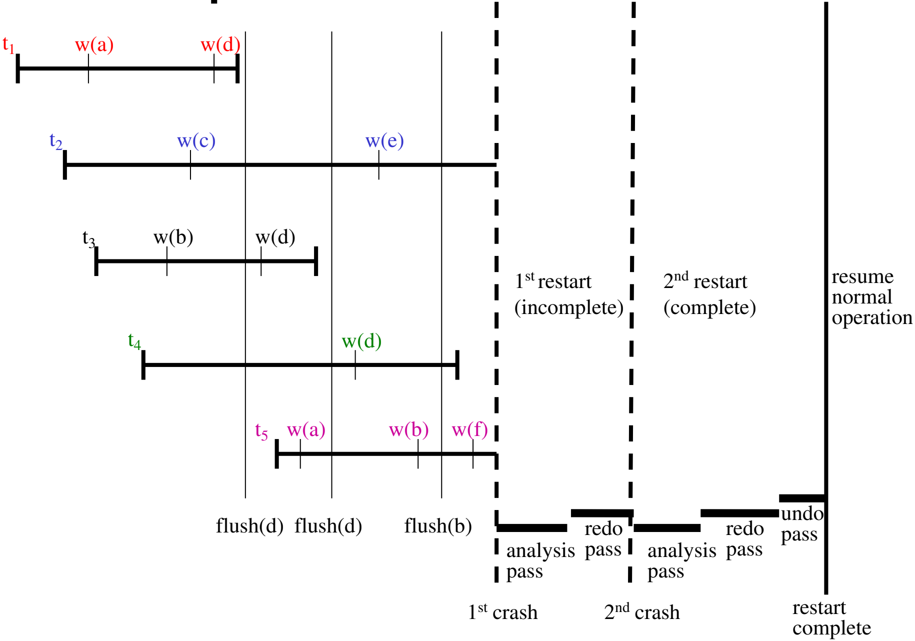
1) all winners must have a commit log entry on stable log
   losers without any stable log entries are irrelevant
2) redo restores last committed write for each page
   (which absorbs all earlier winner writes)
3) LRC implies that losers follow winners for each page
   ⇒ undo restores page state as of the time
   before the first loser write and after the last winner write

⇒ resulting cached database contains exactly the last committed write of the original history

Example Scenario: up to Crash



Example Scenario: from Crash on



Example under Simple Three-Pass Algorithm

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin(t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
⚡ SYSTEM CRASH ⚡				

RESTART

analysis pass: losers = {t_2, t_5}

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
redo (6)	b: 6			
flush (a)		a: 3		
redo (8)	d: 8			
flush (d)		d: 8		
redo (11)	d:11			

⚡ SECOND SYSTEM CRASH ⚡

SECOND RESTART
analysis pass: losers = $\{t_2, t_5\}$

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo(3)	a: 3			
redo(6)	b: 6			
redo(8)	d: 8			
redo(11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 6			
undo(17)	e: 0			
undo(13)	a: 3			
undo(7)	c: 0			

SECOND RESTART COMPLETE: RESUME NORMAL OPERATION

Incorporating General Writes As Physiological Log Entries

Principle:

- state testing during the redo pass:
  for log entry for page p with log sequence number i,
  redo write only if i > p.PageSeqNo
  and subsequently set p.PageSeqNo := i

- state testing during the undo pass:
  for log entry for page p with log sequence number i,
  undo write only if i ≤ p.PageSeqNo
  and subsequently set p.PageSeqNo := i-1

Simple Three-Pass Algorithm with General Writes

```
redo pass ( ):
          ...
          fetch (pageno);
          if DatabaseCache[pageno].PageSeqNo < i
          then
             read and write (pageno)
                  according to StableLog[i].RedoInfo;
             DatabaseCache[pageno].PageSeqNo := i;
          end /*if*/;
          ...
undo pass ( ):
          ...
          fetch (pageno);
          if DatabaseCache[pageno].PageSeqNo >= nextentry.LogSeqNo
          then
             read and write (pageno)
                 according to StableLog[nextentry].UndoInfo;
             DatabaseCache[pageno].PageSeqNo :=
                 nextentry.LogSeqNo - 1;
          end /*if*/;
          ...
```

Correctness of Simple Three-Pass Algorithm for General Writes

Theorem 13.3:
The simple three-pass recovery algorithm with sequence number testing performs correct recovery for general writes.

Example under Simple Three-Pass Algorithm with General Writes

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin(t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
⚡ SYSTEM CRASH ⚡				

RESTART
analysis pass: losers = $\{t_2, t_5\}$

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
consider-redo (6)	b: 18			
flush (a)		a: 3		
consider-redo (8)	d: 11			
consider-redo (11)	d: 11			
⚡ SECOND SYSTEM CRASH ⚡				

Redo steps on d with LSN <= 11 are suppressed

SECOND RESTART
analysis pass: losers = {t_2, t_5}

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
consider-redo(3)	a: 3			
consider-redo(6)	b: 18			
consider-redo(8)	d: 11			
consider-redo(11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 17			
consider-undo(17)	e: 0			
consider-undo(13)	a: 3			
consider-undo(7)	c: 0			

SECOND RESTART COMPLETE: RESUME NORMAL OPERATION

Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- **13.3 Redo-Winners Paradigm**
  - 13.3.1 Actions During Normal Operation
  - 13.3.2 Simple Three-Pass Algorithm
  - **13.3.3 Enhanced Algorithm: Log Truncation, Checkpoints, Redo Optimization**
  - 13.3.4 Complete Algorithm: Handling Transaction Aborts and Undo Completion
- 13.4 Redo-History Paradigm
- 13.5 Lessons Learned

Need and Opportunity for Log Truncation

Major cost factors and potential availability bottlenecks:

1) analysis pass and redo pass scan entire log
2) redo pass performs many random I/Os on stable database

Improvement:

continuously advance the log start pointer (garbage collection)

- for redo, can drop all log entries for page p that precede the last flush action for p =: RedoLSN (p); min{RedoLSN (p) | dirty page p} =: SystemRedoLSN
- for undo, can drop all log entries that precede the oldest log entry of a potential loser =: OldestUndoLSN

Remarks:

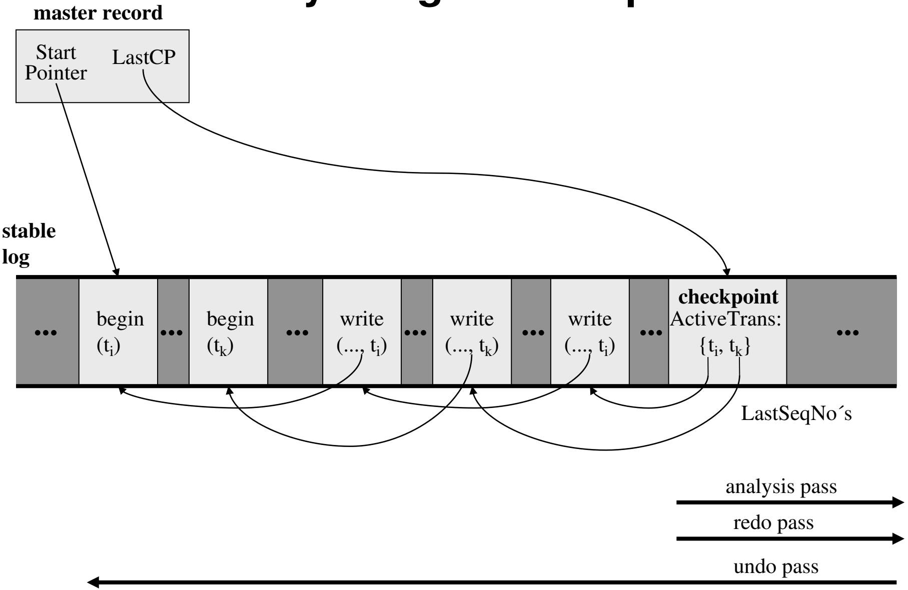
for full-writes, all but the most recent after-image can be dropped

log truncation after complete undo pass requires global flush

Log Truncation

```
log truncation ( ):
    OldestUndoLSN :=
         min {i | StableLog[i].TransId is in ActiveTrans};
    SystemRedoLSN := min {DatabaseCache[p].RedoLSN};
    OldestRedoPage := page p such that
         DatabaseCache[p].RedoLSN = SystemRedoLSN;
    NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    OldStartPointer := MasterRecord.StartPointer;
    while OldStartPointer - NewStartPointer
              is not sufficiently large
    and SystemRedoLSN < OldestUndoLSN
    do
        flush (OldestRedoPage);
        SystemRedoLSN := min{DatabaseCache[p].RedoLSN};
        OldestRedoPage := page p such that
              DatabaseCache[p].RedoLSN = SystemRedoLSN;
        NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    end /*while*/;
    MasterRecord.StartPointer := NewStartPointer;
```

Heavy-Weight Checkpoints



Recovery with Heavy-Weight Checkpoints (1)

```
checkpoint ( ):
    for each p in DatabaseCache do
         if DatabaseCache[p].Status = dirty
         then flush (p);
         end /*if*/;
     end /*for*/;
    logentry.ActionType := checkpoint;
    logentry.ActiveTrans :=
         ActiveTrans (as maintained in memory);
    logentry.LogSeqNo := new sequence number;
    LogBuffer += logentry;
    force ( );
    MasterRecord.LastCP := logentry.LogSeqNo;
```

Recovery with Heavy-Weight Checkpoints (2)

```
analysis pass ( ) returns losers:
   cp := MasterRecord.LastCP;
   losers := StableLog[cp].ActiveTrans;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
        case StableLog[i].ActionType:
             ...
             maintenance of losers
                  as in the algorithm without checkpoints
             ...
        end /*case*/;
   end /*for*/;
redo pass ( ):
   cp := MasterRecord.LastCP;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
         ...
         page-state-testing and redo steps
              as in the algorithm without checkpoints
         ...
   end /*for*/;
```

Example with Heavy-Weight Checkpoints

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: checkpoint		a: 13, b: 6, c: 7, d: 11	14: CP ActiveTrans: {t2, t3, t4, t5}	11, 12, 13 14

Example with Heavy-Weight Checkpoints

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
14: Checkpoint		a: 13, b: 6, c: 7, d: 11	14: CP ActiveTrans: $\{t_2, t_3, t_4, t_5\}$	11, 12, 13 14
15: commit (t_3)			15: commit (t_3)	15
[16: flush (d)]		d: 11		
17: write (d, t_4)	d: 17		17: write (d, t_4)	
18: write (e, t_2)	e: 18		18: write (e, t_2)	
19: write (b, t_5)	b: 19		19: write (b, t_5)	
20: flush (b)		b: 19		17, 18, 19
21: commit (t_4)			21: commit (t_4)	21
22: write (f, t_5)	f: 22		22: write (f, t_5)	
⚡ SYSTEM CRASH ⚡				

RESTART

analysis pass: losers = {t_2, t_5}

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo(17)	d: 17			
undo(19)	b: 18			
consider-undo(18)	e: 0			
undo(13)	a: 12			
undo(7)	c: 6			

RESTART COMPLETE: RESUME NORMAL OPERATION

Dirty Page List for Redo Optimization

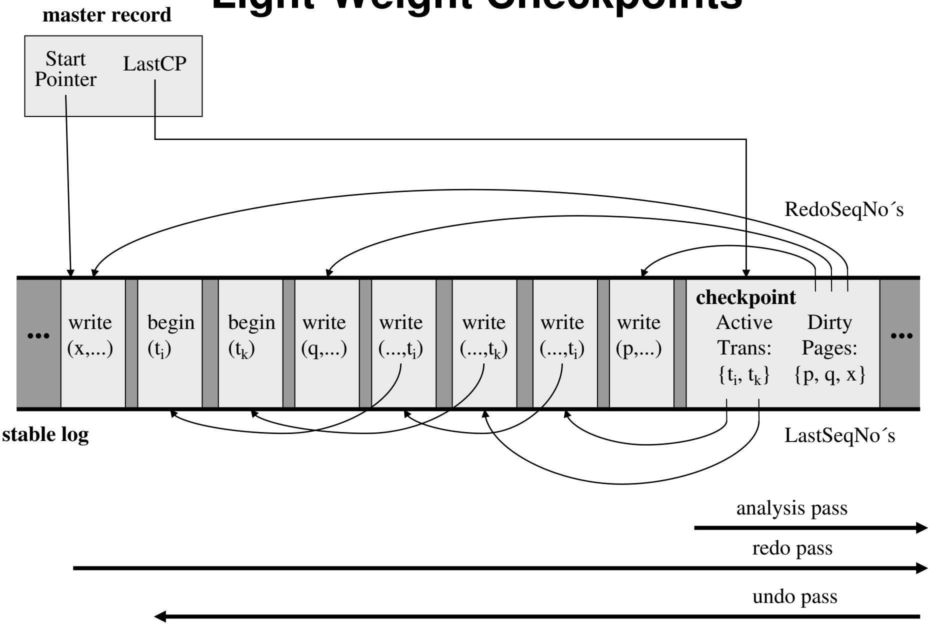
Keep track of
- the set of dirty cached pages
- for each such page the sequence number of the oldest write action that followed the most recent flush action (redo sequence numbers)

Avoid very old RedoSeqNo's by write-behind demon

```
type DirtyPageListEntry: record of
        PageNo: identifier;
        RedoSeqNo: identifier;
     end;
var DirtyPages:
        set of DirtyPageListEntry indexed by PageNo;
```

Record dirty page list in checkpoint log entry and reconstruct (conservative approximation of) dirty page list during analysis pass
→ exploit knowledge of dirty page list and redo sequence numbers for I/O optimizations during redo

Light-Weight Checkpoints



Recovery with Light-Weight Checkpoints (1)

```
checkpoint ( ):
    DirtyPages := empty;
    for each p in DatabaseCache do
         if DatabaseCache[p].Status = dirty
         then
              DirtyPages += p;
              DirtyPages[p].RedoSeqNo :=
                   DatabaseCache[p].RedoLSN;
         end /*if*/;
     end /*for*/;
    logentry.ActionType := checkpoint;
    logentry.ActiveTrans :=
         ActiveTrans (as maintained in memory);
    logentry.DirtyPages := DirtyPages;
    logentry.LogSeqNo := new sequence number;
    LogBuffer += logentry;
    force ( );
    MasterRecord.LastCP := logentry.LogSeqNo;
```

Recovery with Light-Weight Checkpoints (2)

```
analysis pass ( ) returns losers, DirtyPages:
   cp := MasterRecord.LastCP;
   losers := StableLog[cp].ActiveTrans;
   DirtyPages := StableLog[cp].DirtyPages;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
        case StableLog[i].ActionType:
             ...
             maintenance of losers
                  as in the algorithm without checkpoints
             ...
        end /*case*/;
        if StableLog[i].ActionType = write or full-write
              and StableLog[i].PageNo not in DirtyPages
        then
              DirtyPages += StableLog[i].PageNo;
              DirtyPages[StableLog[i].PageNo].RedoSeqNo := i;
        end /*if*/;
   end /*for*/;
```

Recovery with Light-Weight Checkpoints (3)

```
redo pass ( ):
   cp := MasterRecord.LastCP;
   SystemRedoLSN := min{cp.DirtyPages[p].RedoSeqNo};
   max := LogSeqNo of most recent log entry in StableLog;
   for i := SystemRedoLSN to max do
       if StableLog[i].ActionType = write or full-write
            and StableLog[i].TransId not in losers
       then
           pageno := StableLog[i].PageNo;
             if pageno in DirtyPages
                  and i >= DirtyPages[pageno].RedoSeqNo
             then
                  fetch (pageno);
                  if DatabaseCache[pageno].PageSeqNo < i
                  then
                       read and write (pageno)
                          according to StableLog[i].RedoInfo;
                       DatabaseCache[pageno].PageSeqNo := i;
                  else
                       DirtyPages[pageno].RedoSeqNo :=
                            DatabaseCache[pageno].PageSeqNo + 1;
                  end/*if*/; end/*if*/; end/*if*/; end/*for*/;
```

Example with Light-Weight Checkpoints

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: checkpoint			14: CP DirtyPages: {a, b, c, d} RedoLSNs: {a: 3, b: 6, c: 7, d: 11} ActiveTrans: {t_2, t_3, t_4, t_5}	11, 12, 13, 14

Example with Light-Weight Checkpoints

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
14: Checkpoint			14: CP DirtyPages: {a, b, c, d} RedoLSNs: {a: 3, b: 6, c: 7, d: 11} ActiveTrans: $\{t_2, t_3, t_4, t_5\}$	11, 12, 13, 14
15: commit (t_3)			15: commit (t_3)	15
16: flush (d)		d: 11		
17: write (d, t_4)	d: 17		17: write (d, t_4)	
18: write (e, t_2)	e: 18		18: write (e, t_2)	
19: write (b, t_5)	b: 19		19: write (b, t_5)	
20: flush (b)		b: 19		17, 18, 19
21: commit (t_4)			21: commit (t_4)	21
22: write (f, t_5)	f: 22		22: write (f, t_5)	
⚡ SYSTEM CRASH ⚡				

RESTART
analysis pass: losers = {t_2, t_5} **DirtyPages = {a, b, c, d, e}** **RedoLSNs: a: 3, b: 6, c: 7, d: 11, e: 18**

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
consider-redo(3)	a: 3			
consider-redo(6)	b: 19			
skip-redo(8)				
consider-redo(11)	d: 11			
redo(17)	d: 17			
undo(19)	b: 18			
consider-undo(18)	e: 0			
consider-undo(13)	a: 3			
consider-undo(7)	c: 0			

RESTART COMPLETE: RESUME NORMAL OPERATION

Recovery with Flush Log Entries

```
analysis pass ( ) returns losers, DirtyPages:
   cp := MasterRecord.LastCP;
   losers := StableLog[cp].ActiveTrans;
   DirtyPages := StableLog[cp].DirtyPages;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
        case StableLog[i].ActionType:
             ...
             maintenance of losers
                  as in the algorithm without checkpoints
             ...
        end /*case*/;
        if StableLog[i].ActionType = write or full-write
              and StableLog[i].PageNo not in DirtyPages
        then
              DirtyPages += StableLog[i].PageNo;
              DirtyPages[StableLog[i].PageNo].RedoSeqNo := i;
        end /*if*/;
        if StableLog[i].ActionType = flush then
               DirtyPages -= StableLog[i].PageNo;
        end /*if*/; end /*for*/;
```

Example with Light-Weight Checkpoints

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8	**10: flush (d)**	
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: checkpoint			14: CP DirtyPages: {a, b, c, d} RedoLSNs: {a: 3, b: 6, c: 7, d: 11} ActiveTrans: {t_2, t_3, t_4, t_5}	10, 11, 12, 13, 14

Example with Light-Weight Checkpoints

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
14: Checkpoint			14: CP DirtyPages: {a, b, c, d} RedoLSNs: {a: 3, b: 6, c: 7, d: 11} ActiveTrans: {t_2, t_3, t_4, t_5}	10, 11, 12, 13, 14
15: commit (t_3)			15: commit (t_3)	15
16: flush (d)		d: 11	**16: flush (d)**	
17: write (d, t_4)	d: 17		17: write (d, t_4)	
18: write (e, t_2)	e: 18		18: write (e, t_2)	
19: write (b, t_5)	b: 19		19: write (b, t_5)	
20: flush (b)		b: 19	**20: flush (b)**	16, 17, 18, 19
21: commit (t_4)			21: commit (t_4)	20, 21
22: write (f, t_5)	f: 22		22: write (f, t_5)	
⚡ SYSTEM CRASH ⚡				

RESTART
analysis pass: losers = $\{t_2, t_5\}$ **DirtyPages = {a, c, d, e}** **RedoLSNs: a: 3, c: 7, d: 17, e: 18**

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
consider-redo(3)	a: 3			
consider-redo(6)	b: 19			
skip-redo(8)				
skip-redo(11)				
redo(17)	d: 17			
undo(19)	b: 18			
consider-undo(18)	e: 0			
consider-undo(13)	a: 3			
consider-undo(7)	c: 0			

RESTART COMPLETE: RESUME NORMAL OPERATION

Correctness of Enhanced Three-Pass Algorithm

Theorem 13.4:
Extending the simple three-pass recovery algorithm with log truncation, heavy-weight or light-weight checkpoints, and flush action logging (or any subset of these features) preserves the correctness of crash recovery.

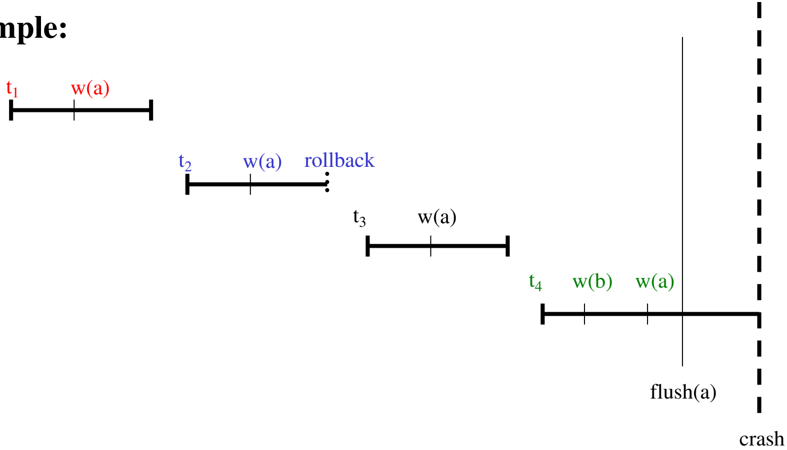
Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- **13.3 Redo-Winners Paradigm**
  - 13.3.1 Actions During Normal Operation
  - 13.3.2 Simple Three-Pass Algorithm
  - 13.3.3 Enhanced Algorithm:
    Log Truncation, Checkpoints, Redo Optimization
  - **13.3.4 Complete Algorithm:**
    Handling Transaction Aborts and Undo Completion
- 13.4 Redo-History Paradigm
- 13.5 Lessons Learned

Problems with Aborted Transactions as Losers

- identifying losers would require full log scan (without advantage from checkpoints)
- losers would precede winners in serialization order

Example:



Example Scenario with Aborted Transactions

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: write (a, t_1)	a: 2		2: write (a, t_1)	
3: commit (t_1)			3: commit (t_1)	1, 2, 3
4: begin (t_2)			4: begin (t_2)	
5: write (a, t_2)	a: 5		5: write (a, t_2)	
6: abort (t_2)			6: abort (t_2)	4, 5, 6
7: begin (t_3)			7: begin (t_3)	
8: write (a, t_3)	a: 8		8: write (a, t_3)	
9: commit (t_3)			9: commit (t_3)	7, 8, 9
10: begin (t_4)			10: begin (t_4)	
11: write (b, t_4)	b: 11		11: write (b, t_4)	
12: write (a, t_4)	a: 12		12: write (a, t_4)	
13: flush (a)		a: 12	13: flush (a)	10, 11, 12
⚡ SYSTEM CRASH ⚡				
RESTART				
Analysis pass: "losers" = {t_2, t_4}				
consider-redo (2)	a: 12			
consider-redo (8)	a: 12			
undo (12)				
consider-undo (11)				
undo (5)				

Handling Aborted Transactions as Winners

- create compensation log entries for
  inverse operations of transaction rollback
- complete rollback by creating rollback log entry
- during crash recovery,
  aborted transactions with complete rollback are winners,
  incomplete aborted transactions are losers

Theorem 13.5:
The extension for handling transaction rollbacks during normal operation preserves the correctness of the three-pass algorithm.

Completion of Transaction Rollback

```
abort (transid):
  logentry := ActiveTrans[transid].LastSeqNo;
  while logentry is not nil and
        logentry.ActionType = write or full-write do
     newlogentry.LogSeqNo := new sequence number;
     newlogentry.ActionType := compensation;
     newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
     newlogentry.RedoInfo :=
        inverse action of the action in logentry;
     newlogentry.UndoInfo :=
        inverse action of inverse action of action in logentry;
     ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
     LogBuffer += newlogentry;
     write (logentry.PageNo) according to logentry.UndoInfo;
     logentry := logentry.PreviousSeqNo;
   end /*while*/
   newlogentry.LogSeqNo := new sequence number;
   newlogentry.ActionType := rollback;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   LogBuffer += newlogentry; ActiveTrans -= transid; force ( );
```

Example with Aborted Transactions as Winners

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: write (a, t_1)	a: 2		2: write (a, t_1)	
3: commit (t_1)			3: commit (t_1)	1, 2, 3
4: begin (t_2)			4: begin (t_2)	
5: write (a, t_2)	a: 5		5: write (a, t_2)	
6: abort (t_2)				
7: compensate (5: write (a, t_2))			7: compensate (a, t_2)	
8: rollback (t_2)			8: rollback (t_2)	4, 5, 7, 8
9: begin (t_3)			9: begin (t_3)	
10: write (a, t_3)	a: 10		10: write (a, t_3)	
11: commit (t_3)			11: commit (t_3)	9, 10, 11
12: begin (t_4)			12: begin (t_4)	
13: write (b, t_4)	b: 13		13: write (b, t_4)	
14: write (a, t_4)	a: 14		14: write (a, t_4)	
15: abort (t_4)				
16: compensate (14: write (a, t_4))			16: compensate (a, t_4)	
17: flush (a)		a: 16		12, 13, 14, 16
⚡ SYSTEM CRASH ⚡				

RESTART
analysis pass: “losers” = {t_4}

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
consider-redo (2)	a: 16			
consider-redo (5)	a: 16			
consider-redo (7)	a: 16			
consider-redo (10)	a: 16			
undo (16)	a: 15			
undo (14)	a: 13			
consider-undo (13)	b: 0			
RESTART COMPLETE: RESUME NORMAL OPERATION				

Undo Completion

- create undo-complete log entry for each loser,
- flush pages modified during undo, and
- set OldestUndoLSN to nil (to facilitate log truncation)

Theorem 13.6:
The method for undo completion
preserves the correctness of the three-pass algorithm.

Complete Undo Algorithm (1)

```
undo pass ( ):
   FlushList := empty;
   while there exists t in losers
         such that losers[t].LastSeqNo <> nil do
      nexttrans := TransNo in losers
             such that losers[TransNo].LastSeqNo =
             max {losers[x].LastSeqNo | x in losers};
      nextentry = losers[nexttrans].LastSeqNo;
      if StableLog[nextentry].ActionType = write then
        pageno := StableLog[nextentry].PageNo; fetch (pageno);
        if DatabaseCache[pageno].PageSeqNo >= nextentry.LogSeqNo;
        then
          read and write (StableLog[nextentry].PageNo)
              according to StableLog[nextentry].UndoInfo;
          DatabaseCache[pageno].PageSeqNo:=nextentry.LogSeqNo - 1;
          FlushList += pageno;
        end /*if*/;
        losers[nexttrans].LastSeqNo :=
            StableLog[nextentry].PreviousSeqNo;
      end /*if*/;
end /*while*/;
```

Complete Undo Algorithm (2)

```
for each p in FlushList do
     flush (p);
end /*for*/;
for each t in losers do
     newlogentry.LogSeqNo := new sequence number;
     newlogentry.ActionType := undo-complete;
     newlogentry.TransId := losers[t].TransId;
     LogBuffer += newlogentry;
end /*for*/;
force ( );
```

Example with Undo Completion

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: write (a, t_1)	a: 2		2: write (a, t_1)	
3: commit (t_1)			3: commit (t_1)	1, 2, 3
4: begin (t_2)			4: begin (t_2)	
5: write (a, t_2)	a: 5		5: write (a, t_2)	
6: abort (t_2)				
7: compensate (5: write (a, t_2))	a: 7		7: compensate (a, t_2)	
8: rollback (t_2)			8: rollback (t_2)	4, 5, 7, 8
9: begin (t_3)			9: begin (t_3)	
10: write (b, t_3)	b: 10		10: write (b, t_3)	
11: commit (t_3)			11: commit (t_3)	9, 10, 11
12: begin (t_4)			12: begin (t_4)	
13: write (b, t_4)	b: 13		13: write (b, t_4)	
14: write (a, t_4)	a: 14		14: write (a, t_4)	
15: abort (t_4)				
16: compensate (14: write (a, t_4))	a: 16		16: compensate (a, t_4)	
17: flush (a)		a: 16		12, 13, 14, 16
18: begin (t_5)			18: begin (t_5)	
19: write (c, t_5)	c: 19		19: write (c, t_5)	
20: begin (t_6)			20: begin (t_6)	
21: write (d, t_6)	d: 21		21: write (d, t_6)	
22: flush (c)		c: 19		18, 19, 20, 21
⚡ SYSTEM CRASH ⚡				

RESTART

analysis pass: "losers" = {t_4, t_5, t_6}

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
consider-redo (2)	a: 16			
consider-redo (5)	a: 16			
consider-redo (7)	a: 16			
redo (10)	b: 16			
consider-undo (21)	d: 0			
undo (19)	c: 18			
undo (16)	a: 15			
undo (14)	a: 13			
consider-undo (13)	b: 13			
flush (a)		a: 13		
flush (c)		c: 18		
23: undo-complete (t_4)			23: undo-complete (t_4)	
24: undo-complete (t_5)			24: undo-complete (t_5)	
25: undo-complete (t_6)			25: undo-complete (t_6)	
force				23, 24, 25

RESTART COMPLETE: RESUME NORMAL OPERATION

Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- 13.3 Redo-Winners Paradigm
- **13.4 Redo-History Paradigm**
  - 13.4.1 Actions During Normal Operation
  - 13.4.2 Simple Three-Pass and Two-Pass Algorithms
  - 13.4.3 Enhanced Algorithms
  - 13.4.4 Complete Algorithms
- 13.5 Lessons Learned

Basic Idea

- In **Redo-Winners**, the redo pass considers only winners, at the expense of complicating transaction aborts and log truncation.

- In **Redo-History**, *all* actions are repeated in chronological order, i.e.,

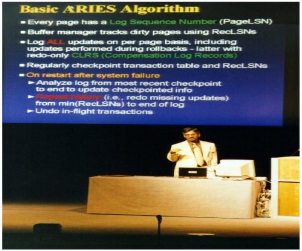
  1. it first reconstructs the cached database,

  2. then undoes losers from there.

Redo-History: ARIES

Adresse http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html

ARIES Family of Locking and Recovery Algorithms



This page is devoted to tracking information on the **ARIES** (*Algorithms for Recovery and Isolation Exploiting Semantics*) family of locking, logging and recovery algorithms for persistent data management. I have included information on the books and university courses which cover ARIES with links to course materials, teachers and authors. The impact of ARIES on products, prototypes and researchers is also outlined. A listing of our papers and patents on ARIES is also included.

The impact of ARIES on the research and the commercial worlds was recognized with the "10 Year Best Impact Paper Award" at VLDB99. The birth and evolution of ARIES is described in my VLDB99 paper. ARIES is covered in 14 books and more than 80 universities' computer science courses across the world (Australia, Canada, Denmark, England, Finland, France, Germany, Greece, India, Iran, Israel, Italy, Korea, New Zealand, Norway, Singapore, Spain, Sweden, Taiwan, USA). Excluding self-citations, so far, the main ARIES paper (TODS, March 1992) has been cited more than 230 times, the ARIES/IM paper (SIGMOD92) 90 times, and the ARIES/KVL paper (VLDB90) 60 times. The referenced citation lists are much more complete than the ones at DBLP, ACM and ResearchIndex.

I am very thankful to the professors, authors and systems builders who have made the ARIES algorithms extremely popular via their books, courses, papers and implementations. Any comments, corrections and additions to this page's contents would be most welcome!

Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- 13.3 Redo-Winners Paradigm
- **13.4 Redo-History Paradigm**
  - 13.4.1 Actions During Normal Operation
  - **13.4.2 Simple Three-Pass and Two-Pass Algorithms**
  - 13.4.3 Enhanced Algorithms
  - 13.4.4 Complete Algorithms
- 13.5 Lessons Learned

Key Properties of Redo-History Algorithms

- *Optional* analysis pass
  - determines losers and
  - reconstructs DirtyPages list,
    using the analysis algorithm of the redo-winners paradigm
- Redo pass starts from SystemRedoLSN and
  - redoes *both* winner and loser updates,
    with LSN-based state testing for idempotence,
    to reconstruct the database state as of the time of the crash
- Undo pass initiates *rollback* for all loser transactions,
  using the code for rollback during normal operation,
  with undo steps (without page state testing)
  - creating *compensation log entries* and
  - *advancing* page sequence numbers

Redo Pass of Redo-History Algorithms

```
redo pass ( ):
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max do
          pageno = StableLog[i].PageNo;
          fetch (pageno);
          if DatabaseCache[pageno].PageSeqNo < i
          then
             read and write (pageno)
                  according to StableLog[i].RedoInfo;
             DatabaseCache[pageno].PageSeqNo := i;
          end /*if*/;
   end /*for*/;
```

Undo Pass of Redo-History Algorithms (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;
```

Undo Pass of Redo-History Algorithms (2)

```
if StableLog[nextentry].ActionType in {write, compensation}
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.RedoInfo :=
          inverse action of the action in nextentry;
     newlogentry.UndoInfo := inverse action of the
          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].PreviousSeqNo;
end /*if*/;
```

Undo Pass of Redo-History Algorithms (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 ( );
```

Simple Three-Pass Redo-History Algorithm

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin(t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
⚡ SYSTEM CRASH AND RESTART⚡				

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
Analysis pass: losers = $\{t_2, t_5\}$				
redo (3)	a: 3			
consider-redo (6)	b: 18			
flush (a)		a: 3		
redo (7)	c: 7			
consider-redo (8)	d: 11			
consider-redo (11)	d: 11			
redo (13)	a: 13			
redo (16)	d: 16			
redo (17)	e: 17			
consider-redo (18)	b: 18			
flush (a)		a: 13		
22: compensate (18)	b: 22		22: compensate (18: b, t_5)	
23: compensate (17)	e: 23		23: compensate (17: e, t_2)	
flush (b)		b: 22		22, 23
24: compensate (13)	a: 24		24: compensate (13: a, t_5)	
25: rollback (t_5)			25: rollback (t_5)	
⚡ SECOND SYSTEM CRASH AND SECOND RESTART⚡				

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
Analysis pass: losers = {t_2, t_5}				
redo (3)	a: 13			
consider-redo (6)	b: 22			
redo (7)	c: 7			
consider-redo (8)	d: 11			
consider-redo (11)	d: 11			
consider-redo (13)	a: 13			
redo (16)	d: 16			
redo (17)	e: 17			
consider-redo (18)	b: 22			
consider-redo (22)	b: 22			
redo (23)	e: 23			
26: compensate (23)	e: 26		26: compensate (23, e: t_2)	
27: compensate (22)	b: 27		27: compensate (22, e: t_5)	
28: compensate (18)	b: 28		28: compensate (18, b: t_5)	
29: compensate (17)	e: 29		29: compensate (17, e: t_2)	
30: compensate (13)	a: 30		30: compensate (13, a: t_5)	
31: rollback (t_5)			31: rollback (t_5)	
32: compensate (7)	c: 32		32: compensate (7: c, t_2)	
33: rollback (t_2)			31: rollback (t_2)	
force				26, 27, 28, 29, 30, 31, 32, 33

SECOND RESTART COMPLETE: RESUME NORMAL OPERATION

Correctness of Simple Redo-History Algorithm

Theorem 13.7:
The simple three-pass redo-history recovery algorithm performs correct recovery.

Proof sketch:

- redo pass establishes the postcondition
  $\forall p\ \forall t\ \forall o \in$ stable log: (o belongs to t and refers to p) $\Rightarrow$ o $\in$ cached db
- undo pass performs rollback like during normal operation
  and establishes the postcondition
  $\forall p\ \forall t\ \forall o \in$ stable log: (o belongs to t and refers to p and t $\in$ losers) $\Rightarrow$ o $\notin$ cached db
- as losers follow winners in the serialization order,
  the final postcondition of the entire restart is
  $\forall p\ \forall t\ \forall o \in$ stable log: (o belongs to t and refers to p and t $\in$ winners) $\Rightarrow$ o $\in$ cached db
- a second crash during redo does not affect the second restart
- a second crash during undo could leave losers prolonged with some (but not all) inverse actions; the second restart will treat them as if the inverse actions were forward actions, and thus is no different from the first restart

Chapter 13: Page-Model Crash Recovery Algorithms

- 13.2 Basic Data Structures
- 13.3 Redo-Winners Paradigm
- **13.4 Redo-History Paradigm**
  - 13.4.1 Actions During Normal Operation
  - 13.4.2 Simple Three-Pass and Two-Pass Algorithms
  - 13.4.3 Enhanced Algorithms
  - **13.4.4 Complete Algorithms**
- 13.5 Lessons Learned

Undo Completion for Redo-History Algorithms

By completing losers, creating CLEs, and advancing page sequence numbers during undo, upon completed restart the log can be truncated at the SystemRedoLSN (without need for flushing)

(Minor) problem:
repeated crashes during undo lead to multiple-times inverse actions that could make successive restarts longer

Example:

10: write(t_i,a)
20: write(t_i,b)
30: write(t_i,c)
first crash: redo 10, 20, 30
need to undo 30, 20, 10
40: write(t_i,c)$^{-1}$
50: write(t_i, b)$^{-1}$
second crash: redo 10, 20, 30, 40, 50
need to undo 50, 40, 30, 20, 10
60: (write(t_i, b)$^{-1}$)$^{-1}$
70: (write(t_i, c)$^{-1}$)$^{-1}$
80: write(t_i,c)$^{-1}$
90: write(t_i, b)$^{-1}$
100: write(t_i, a)$^{-1}$
second restart complete

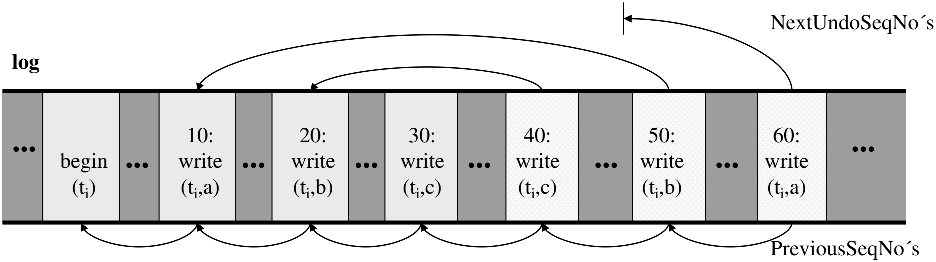
Next Undo Sequence Number Backward Chaining

Multiple-times inverse actions can be avoided by backward chaining a CLE to the predecessor of its corresponding forward action and following this **NextUndoSeqNo** backward chain during undo

Example:

10: write(t_1,a), NextUndoSeqNo=nil
20: write(t_i,b), NextUndoSeqNo=10
30: write(t_i,c), NextUndoSeqNo=20
first crash: redo 10, 20, 30; need to undo 30, 20, 10
40: write(t_i,c)$^{-1}$, NextUndoSeqNo=20
50: write(t_i, b)$^{-1}$, NextUndoSeqNo=10
second crash: redo 10, 20, 30, 40, 50; need to undo 10
60: write(t_i, a)$^{-1}$, NextUndoSeqNo=nil
second restart complete

Illustration of Next Undo Sequence Number Backward Chaining



Undo Pass with CLEs and NextUndoSeqNo Backward Chaining (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*/;
```

Undo Pass with CLEs and NextUndoSeqNo Backward Chaining (2)

```
if StableLog[nextentry].ActionType = 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*/;
```

Undo Pass with CLEs and NextUndoSeqNo Backward Chaining (3)

```
      losers[nexttrans].LastSeqNo =
            StableLog[nextentry].PreviousSeqNo;
    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 ( );
```

Transaction Abort During Normal Operation with CLEs and NextUndoSeqNo Backward Chaining (1)

```
abort (transid):
   logentry := ActiveTrans[transid].LastSeqNo;
   while logentry is not nil and
         logentry.ActionType = write or full-write do
         newlogentry.LogSeqNo := new sequence number;
         newlogentry.ActionType := compensation;
         newlogentry.PreviousSeqNo :=
              ActiveTrans[transid].LastSeqNo;
         newlogentry.RedoInfo :=
               inverse action of the action in logentry;
         newlogentry.NextUndoSeqNo :=
               logentry.PreviousSeqNo;
         ActiveTrans[transid].LastSeqNo :=
               newlogentry.LogSeqNo;
         LogBuffer += newlogentry;
         write (logentry.PageNo)
               according to logentry.UndoInfo;
         logentry := logentry.PreviousSeqNo;
   end /*while*/
```

Transaction Abort During Normal Operation with CLEs and NextUndoSeqNo Backward Chaining (2)

```
newlogentry.LogSeqNo := new sequence number;
newlogentry.ActionType := rollback;
newlogentry.TransId := transid;
newlogentry.PreviousSeqNo :=
     ActiveTrans[transid].LastSeqNo;
newlogentry.NextUndoSeqNo := nil;
LogBuffer += newlogentry;
ActiveTrans -= transid;
force ( );
```

Example with Undo Completion of Three-Pass Redo-History Recovery

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin(t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
⚡ SYSTEM CRASH AND RESTART⚡				

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
Analysis pass: losers = {t_2, t_5}				
redo (3)	a: 3			
consider-redo (6)	b: 18			
flush (a)		a: 3		
redo (7)	c: 7			
consider-redo (8)	d: 11			
consider-redo (11)	d: 11			
redo (13)	a: 13			
redo (16)	d: 16			
redo (17)	e: 17			
consider-redo (18)	b: 18			
flush (a)		a: 13		
22: compensate (18)	b: 22		**22: compensate (18: b, t_5) NextUndoSeqNo: 13**	
23: compensate (17)	e: 23		**23: compensate (17: e, t_2) NextUndoSeqNo: 7**	
flush (b)		b: 22		22, 23
24: compensate (13)	a: 24		**24: compensate (13: a, t_5) NextUndoSeqNo: nil**	
25: rollback (t_5)			25: rollback (t_5)	
⚡ SECOND SYSTEM CRASH AND SECOND RESTART⚡				

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
Analysis pass: losers = {t_2, t_5}				
consider-redo (3)	a: 13			
consider-redo (6)	b: 22			
redo (7)	c: 7			
consider-redo (8)	d: 11			
consider-redo (11)	d: 11			
consider-redo (13)	a: 13			
redo (16)	d: 16			
redo (17)	e: 17			
consider-redo (18)	b: 22			
consider-redo (22)	b: 22			
redo (23)	e: 23			
26: compensate (13)	a: 26		**26: compensate (13, e: t_2) NextUndoSeqNo: nil**	
27: rollback (t_5)			27: rollback (t_5)	
28: compensate (7)	c: 28		**32: compensate (7: c, t_2) NextUndoSeqNo: nil**	
33: rollback (t_2)			31: rollback (t_2)	
force				26, 27, 28, 29
SECOND RESTART COMPLETE: RESUME NORMAL OPERATION				

Correctness of Undo Completion with CLEs and NextUndoSeqNo Backward Chaining

Theorem 13.8:
The method for undo completion, based on executing inverse actions and creating CLEs that are backward-chained to reflect the next undo log sequence numbers, preserves the correctness of the three-pass or two-pass redo-history recovery algorithms.

Proof sketch:
The following invariant holds:

$\forall$log sequence numbers s $\in$ stable log such that
all more recent log entries of losers, including s,
have been processed by the undo pass:
$\forall$u $\in$ stable log with u.LogSeqNo $\geq$ s.LogSeqNo: $\forall$o $\in$ stable log:
(u.TransId $\in$ losers and o.TransId = u.TransId and
o.LogSeqNo > u.NextUndoSeqNo) $\Rightarrow$ o $\notin$ cached db

Chapter 13: Page-Model Crash Recovery Algorithms

Lessons Learned

- Redo-history algorithm preferable

  because of uniformity, no need for page flush during restart,

  simplicity, and robustness

  (and extensibility towards object model, see Chapter 14)

- Main ingredients are:
  - three passes for log analysis, redo, undo
  - light-weight checkpoints for log truncation
  - additional flush log entries for further savings of redo cost
  - compensation log entries

    for transaction rollback and undo completion