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 15: Special Issues of Recovery

• 15.2 Logical Logging for Indexes and Large Objects
• 15.3 Intra-transaction Savepoints
• 15.4 Exploiting Parallelism During Restart
• 15.5 Main-Memory Data Servers
• 15.6 Data-Sharing Clusters
• 15.7 Lessons Learned

“Success is a lousy teacher.” (Bill Gates)
2-Level Logging for Index Operations

log entries for insert\(_{ij}\) (k, @x)
on B-tree path along pages r, n, l, with split of l into l and m:
  write\(_{ij1}\)(l)
  write\(_{ij2}\)(m)
  write\(_{ij3}\)(n)
  insert\(^{-1}\)\(_{ij}\)(k, @x)

→ writes the original contents of l
twice on the log (undo/redo info for l and m)
Logical Logging for Redo of Index Splits

log only $L_1$ operation for transaction redo (to save log space) and rely on careful flush ordering for subtransaction atomicity

possible cases after a crash (because of arbitrary page flushing):

1) $l$, $m$, and $n$ are in old state (none were flushed)
2) $l$ is new, $m$ and $n$ are old
3) $m$ is new, $l$ and $n$ are old
4) $n$ is new, $l$ and $m$ are old
5) $l$ and $m$ are new, $n$ is old
6) $l$ and $n$ are new, $m$ is old
7) $m$ and $n$ are new, $l$ is old
8) $l$, $m$, and $n$ are in new state (all were flushed)

must avoid cases 2 and 6 (all other cases are recoverable) by enforcing flush order $m < l < n$

in addition, posting $(n)$ could be detached from half-split ($l$ and $m$) by link technique, so that $m < l$ is sufficient
The Need for Redo and Flush-Order Dependencies

Problem: if a were flushed before b and the system crashed in between, the copy operation with LSN 100 could not be redone.
Redo and Flush-Order Dependencies

**Opportunity:** operations on large objects (BLOBs, stored procedure execution state) can achieve significant savings on log space by logical logging

**Difficulty:** redo of partially surviving multi-page operations

**Definition:**
There is a **redo dependency** from logged operation \( f(...) \) to logged operation \( g(...) \) if
- \( f \) precedes \( g \) on the log and
- there exists page \( x \) such that \( x \in \text{readset}(f) \) and \( x \in \text{writeset}(g) \)

**Definition:**
There is a **flush order dependency** from page \( y \) to page \( z \) (i.e., page \( y \) must be flushed before page \( z \)) if
- there are logged operations \( f \) and \( g \) with
  - \( y \in \text{writeset}(f) \) and \( z \in \text{writeset}(g) \)
  - and a redo dependency from \( f \) to \( g \).
Cyclic Flush-Order Dependencies

Need to flush all pages on the cycle atomically or force physical, full-write, log entries (i.e., after-images) atomically.
Intra-Operation Flush-Order Dependencies

**redo dependencies**

- LSN 500
  - swap (a, b)
  - readset: {a, b}
  - writeset: {a, b}

**read-write dependencies**

- LSN 1000
  - half-split (l)
  - readset: {l}
  - writeset: {l, m}

**flush-order dependencies**

- page a
  - written by: 500
- page b
  - written by: 500
- page m
  - written by: 1000
- page l
  - written by: 1000
Chapter 15: Special Issues of Recovery

• 15.2 Logical Logging for Indexes and Large Objects

• **15.3 Intra-transaction Savepoints**

• 15.4 Exploiting Parallelism During Restart

• 15.5 Main-Memory Data Servers

• 15.6 Data-Sharing Clusters

• 15.7 Lessons Learned
The Case for Partial Rollbacks

Additional calls during normal operation (for partial rollbacks to resolve deadlocks or application-defined intra-transaction consistency points):

- **save** (trans) $\uparrow s$
- **restore** (trans, s)

**Approach:**
savepoints are recorded on the log, and restore creates CLEs

**Problem with nested rollbacks:**
\[ l_1(x) \ w_1(x) \ l_1(y) \ w_1(y) \ w_1^{-1}(y) \ u_1(y) \ l_2(y) \ w_2(y) \ c_2 \ l_1(y) \ (w_1^{-1}(y))^{-1} \ w^{-1}(y) \ w^{-1}(x) \]
\[ \rightarrow \text{not prefix reducible} \]

**Problem eliminated with NextUndoSeqNo backward chaining:**
\[ l_1(x) \ w_1(x) \ l_1(y) \ w_1(y) \ w_1^{-1}(y) \ u_1(y) \ l_2(y) \ w_2(y) \ c_2 \ w^{-1}(x) \]
\[ \rightarrow \text{prefix reducible} \]
NextUndoSeqNo Backward Chain for Nested Rollbacks

log during normal operation

log continued during restart

10: write \((t_i, a)\)
20: write \((t_i, b)\)
30: save-point \((t_i)\)
40: write \((t_i, c)\)
45: save-point \((t_i)\)
50: write \((t_i, d)\)
60: write \((t_i, e)\)
63: \(t_i\), e, 60
64: \(t_i\), d, 50
65: restore \((t_i, 45)\)
70: write \((t_i, f)\)
73: \(t_i\), f, 70
74: \(t_i\), c, 40
75: restore \((t_i, 30)\)
83: \(t_i\), b, 20

84: CLE \((t_i, a, 10)\)
85: rollback \((t_i)\)

first restore initiated
second restore initiated
abort initiated

crash

...
Savepoint Algorithm

savepoint (transid):
   
   newlogentry.LogSeqNo := new sequence number;
   newlogentry.ActionType := savepoint;
   newlogentry.PreviousSeqNo :=
      ActiveTrans[transid].LastSeqNo;
   newlogentry.NextUndoSeqNo :=
      ActiveTrans[transid].LastSeqNo;
   ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
   LogBuffer += newlogentry;
Restore Algorithm

\[ \text{restore (transid, s):} \]
\[ \quad \text{logentry := ActiveTrans[transid].LastSeqNo;} \]
\[ \quad \text{while logentry is not equal to s do} \]
\[ \quad \quad \text{if logentry.ActionType = write or full-write then} \]
\[ \quad \quad \quad \text{newlogentry.LogSeqNo := new sequence number;} \]
\[ \quad \quad \quad \text{newlogentry.ActionType := compensation;} \]
\[ \quad \quad \quad \text{newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;} \]
\[ \quad \quad \quad \text{newlogentry.RedoInfo := inverse action of the action in logentry;} \]
\[ \quad \quad \quad \text{newlogentry.NextUndoSeqNo := logentry.PreviousSeqNo;} \]
\[ \quad \quad \text{ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;} \]
\[ \quad \quad \text{LogBuffer += newlogentry;} \]
\[ \quad \quad \text{write (logentry.PageNo) according to logentry.UndoInfo;} \]
\[ \quad \quad \text{logentry := logentry.PreviousSeqNo;} \]
\[ \quad \quad \text{end /*if*/;} \]
\[ \quad \quad \text{if logentry.ActionType = restore then} \]
\[ \quad \quad \quad \text{logentry := logentry.NextUndoSeqNo;} \]
\[ \quad \quad \text{end /*if*/} \]
\[ \quad \text{end /*while*/} \]
\[ \quad \text{newlogentry.LogSeqNo := new sequence number;} \]
\[ \quad \text{newlogentry.ActionType := restore;} \]
\[ \quad \text{newlogentry.TransId := transid;} \]
\[ \quad \text{newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;} \]
\[ \quad \text{newlogentry.NextUndoSeqNo := s.NextUndoSeqNo;} \]
\[ \quad \text{LogBuffer += newlogentry;} \]
Savepoints in Nested Transactions

**beginnings of active subtransactions are feasible savepoints**
Chapter 15: Special Issues of Recovery

• 15.2 Logical Logging for Indexes and Large Objects
• 15.3 Intra-transaction Savepoints

• **15.4 Exploiting Parallelism During Restart**

• 15.5 Main-Memory Data Servers
• 15.6 Data-Sharing Clusters
• 15.7 Lessons Learned
Exploiting Parallelism During Restart

- **Parallelize redo** by spawning multiple threads for different page subsets (driven by DirtyPages list), assuming physical or physiological log entries
- **Parallelize log scans** by partitioning the stable log across multiple disks based on hash values of page numbers
- **Parallelize undo** by spawning multiple threads for different loser transactions

**Incremental restart** with early admission of new transactions right after redo
- by re-acquiring locks of loser transactions (or coarser locks) during redo of history, or
- right after log analysis by allowing access, already during redo, to all non-dirty pages \( p \) with \( p.\text{PageSeqNo} < \text{OldestUndoLSN}(p) \)
Chapter 15: Special Issues of Recovery

- 15.2 Logical Logging for Indexes and Large Objects
- 15.3 Intra-transaction Savepoints
- 15.4 Exploiting Parallelism During Restart
- **15.5 Main-Memory Data Servers**
- 15.6 Data-Sharing Clusters
- 15.7 Lessons Learned
Considerations for Main-Memory Data Servers

Main-memory databases are particularly attractive for telecom or financial apps with < 50 GB of data, fairly uniform workload of short transactions, and very stringent response time requirements.

Specific opportunities:
- crash recovery amounts to reloading the database
  → physical (after-image) logging attractive
- eager page flushing in the background
  amounts to “fuzzy checkpoint”
- in-memory versioning (with no-steal caching)
  can eliminate writing undo information to stable log
- log buffer forcing can be avoided by “safe RAM”
- incremental, page-wise, redo (and undo) on demand
  may deviate from chronological order
Chapter 15: Special Issues of Recovery

• 15.2 Logical Logging for Indexes and Large Objects
• 15.3 Intra-transaction Savepoints
• 15.4 Exploiting Parallelism During Restart
• 15.5 Main-Memory Data Servers

• **15.6 Data-Sharing Clusters**

• 15.7 Lessons Learned
Architecture of Data-Sharing Clusters

Data-sharing cluster:
multiple computers (as data servers) with local memory
and shared disks via high-speed interconnect
for load sharing, failure isolation, and very high availability

During normal operation:
• transactions initiated and executed locally
• pages transferred to local caches on demand (data shipping)
• coherency control eliminates stale page copies:
  • multiple caches can hold up-to-date copies read-only
  • upon update in one cache, all other caches drop their copies
  • can be combined with page-model or object-model CC
• logging to global log on shared disk or
  partitioned log with static assignment of server responsibilities or
  private logs for each server for perfect scalability

Upon failure of a single server:
failover to surviving servers
Illustration of Data-Sharing Cluster

Server 1

Cache

- 4215 page p
- 4299 page x
- 3155 page q

Stable Log

- 4299 write(x, ...)
- 4158 write(y, ...)

Server 2

Cache

- 4215 page p
- 3155 page q

Stable Log

- 4215 write(p, ...)
- 4088 write(x, ...)

Server n

Cache

- 4309 page y
- 3155 page q

Stable Log

- 4309 write(y, ...)
- 4218 write(x, ...)

Stable Database

- 4011 page p
- 4088 page x
- 3155 page q
- 4158 page y
Recovery with “Private” Logs

needs page-wise globally monotonic sequence numbers, e.g., upon update to page p (without any extra messages):
\[ p.\text{PageSeqNo} := \max\{p.\text{PageSeqNo}, \text{largest local seq no}\} + 1 \]

surviving server performs crash recovery on behalf of the failed one,
• with analysis pass on private log of failed server to identify losers,
• scanning and “merging” all private logs for redo, possibly with DirtyPages info from the failed server, (merging can be avoided by flushing before each page transfer across servers),
• scanning private log of failed server for undo

recovery from failure of entire cluster needs analysis passes, merged redo passes, and undo passes over all private logs
Chapter 15: Special Issues of Recovery

• 15.2 Logical Logging for Indexes and Large Objects
• 15.3 Intra-transaction Savepoints
• 15.4 Exploiting Parallelism During Restart
• 15.5 Main-Memory Data Servers
• 15.6 Data-Sharing Clusters

• 15.7 Lessons Learned
Lessons Learned

• The redo-history algorithms from Chapter 13 and 14 can be extended in a fairly localized and incremental manner.
• Practically important extensions are:
  • logical log entries for multi-page operations
  • as an additional option
  • intra-transaction savepoints and partial rollbacks
  • parallelized and incremental restart for higher availability
  • special architectures like
    - main-memory data servers
    - for sub-second responsiveness and
    - data-sharing clusters
    - for very high availability