
Recovery

Christos Faloutsos
www.cs.cmu.edu/~christos

Roadmap

1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
3) Transactions: locking, recovery
   granularity of locks
   recovery
   B-trees
   ...
4) Distributed DBMSs
5) Parallel DBMSs: Gamma, Alphasort
   ...

Reference

Principles of Transaction Oriented Database
Recovery T. Haerder and A. Reuter, ACM

Detailed roadmap

• Problem definition - ACID properties
• Mapping hierarchy
• Crash recovery
• Archive recovery
• Conclusions

Transactions - dfn

= unit of work, eg.
move $10 from savings to checking

Atomicity (all or none)
Consistency
Isolation (as if alone)
Durability

recovery
concurrency control

Recovery

• Durability - types of failures?
Recovery

- Durability - types of failures?
- disk crash (ouch!)
- power failure
- software errors (deadlock, division by zero)

Reminder: types of storage

- volatile (eg., main memory)
- non-volatile (eg., disk, tape)
- “stable” (“never” fails - how to implement it?)

Classification of failures:

- frequent; ‘cheap’
  - logical errors (eg., div. by 0)
  - system errors (eg. deadlock - pgm can run later)
  - system crash (eg., power failure - volatile storage is lost)
  - disk failure

- rare; expensive

Goals of recovery

- Guarantee atomicity and durability

For good performance:

- minimize time to restore db (after failure)
- minimize overhead during normal op.

Actions

- UNDO xact
- UNDO all
- REDO partial
- REDO all

after failure (system crash)
archive recovery - destroyed db

Detailed roadmap

- Problem definition - ACID properties
- Mapping hierarchy
  - Crash recovery
  - Archive recovery
- Conclusions
Views of a db

- current db -> on disk + buffers
- materialized db -> on disk only

Problem definition

- Records are on disk
- for updates, they are copied in memory
- and flushed back on disk, \textit{at the discretion of the O.S.}! (unless forced-output: `output(B)` = fflush())
Classification

- ATOMIC / ∼ATOMIC (‘in place’, or not)
- STEAL / ∼STEAL (DBMS pins buffers)
- FORCE / ∼FORCE (DBMS forces buffers out)

(we’ll mainly focus on ∼ATOMIC -> WAL)

Problem definition - eg.:

```
read(X)
X=X+1
write(Y)
Y=Y-1
write(X)
X=X+1
```

Problem definition - eg.:

```
read(X)
X=X+1
write(X)
buffer joins an output queue,
but it is NOT flushed immediately!
Q1: why not?
Q2: so what?
```

Problem definition - eg.:

```
read(X)
read(Y)
X=X+1
Y=Y-1
write(X)
write(Y)
```

Q2: so what?

Problem definition - eg.:

```
read(X)
read(Y)
X=X+1
Y=Y-1
write(X)
write(Y)
write(Y)
```

Q2: so what?

Q3: how to guard against it?
Solution #1: W.A.L.
- ¬ATOMIC (¬ ‘in place’ updates)
- redundancy, namely
- write-ahead log, on ‘stable’ storage
- Q: what to replicate? (not the full page!!)
- A:
- Q: how exactly?

W.A.L. - intro
- replicate intentions: eg:
  <T1 start>
  <T1, X, 5, 6>
  <T1, Y, 4, 3>
  <T1 commit> (or <T1 abort>)

Other log data?
- we store <tid, item-id, v-before, v-after>
- any other alternative?

W.A.L. - intro
- when done, write a <commit> record & exit

Other log data?
- we store <tid, item-id, v-before, v-after>
- any other alternative?

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>DML</td>
</tr>
<tr>
<td>Physical</td>
<td>Before/after XOR</td>
</tr>
</tbody>
</table>

W.A.L. - deferred updates
- (¬STEAL, ¬ FORCE)
- idea: prevent OS from flushing buffers, until ‘commit’.
- After a failure, “replay” the log
W.A.L. - deferred updates

- Q: how, exactly?
  - value of W on disk?
  - value of W after recov.?
  - value of Z on disk?
  - value of Z after recov.?

- Disadvantages?
  (e.g., “increase all balances by 5%”)
  May run out of buffer space!
  Hence: ‘STEAL’!

W.A.L. - deferred updates

- Thus, the recovery algo:
  - redo committed transactions
  - ignore uncommitted ones

- Observations:
  - no need to keep ‘old’ values
  - Disadvantages?

W.A.L. - incremental updates

(→ ATOMIC, STEAL, ~FORCE)
- log records have ‘old’ and ‘new’ values.
- modified buffers can be flushed at any time
W.A.L - incremental updates

Each transaction:
- writes a log record first, before doing the change
- writes a ‘commit’ record (if all is well)
- exits

W.A.L. - incremental updates

• Q: how, exactly?
  – value of W on disk?
  – value of W after recov.?
  – value of Z on disk?
  – value of Z after recov.?

W.A.L. - incremental updates

• Q: recovery algo?
  • A:
    – redo committed xacts
    – undo uncommitted ones
  • (more details: soon)

W.A.L. - incremental updates

Observations
• “increase all balances by 5%” - problems?
  • what if the log is huge?

Overview - recovery

• problem definition
  – types of failures
  – types of storage
• solution#1: Write-ahead log
  – deferred updates
  – incremental updates
  – checkpoints
• (solution #2: shadow paging)
W.A.L. - check-points

Idea: periodically, flush buffers
Q: should we write anything on the log?

W.A.L. - check-points

Q: how does it help us?
A: yes!
Q: how does it help us?

W.A.L. - check-points

Q: how does it help us?
A=? on disk?
B=? on disk?
C=? on disk?
C=? after recovery?

W.A.L. - check-points

Q: how is the recovery algorithm?
A:
- undo uncommitted xacts (eg., T500)
- redo the ones committed after the last checkpoint (eg., none)

W.A.L. - w/ concurrent xacts

Log helps to rollback transactions (eg., after a deadlock + victim selection)
Eg., rollback(T500): go backwards on log: restore old values
W.A.L. - w/ concurrent xacts

- recovery algo?
- undo uncommitted ones
- redo ones committed
  after the last checkpoint

<checkpoint>

<checkpoints>

<checkpoints>

<checkpoints>

W.A.L. - w/ concurrent xacts

- recovery algo?
- undo uncommitted ones
- redo ones committed
  after the last checkpoint

- Eg.?

Recovery algo:
- build ‘undo’ and ‘redo’ lists
- scan backwards, undoing ops
  by the ‘undo’-list transactions
- go to most recent checkpoint
- scan forward, redoing ops by the ‘redo’-list xacts

T4 commit

T3 commit

T2 commit

T1 commit

<checkpoint>

<checkpoints>

<checkpoints>

<checkpoints>

W.A.L. - w/ concurrent xacts

- recovery algo?
- undo uncommitted ones
- redo ones committed
  after the last checkpoint

- Eg.?

Recovery algo:
- build ‘undo’ and ‘redo’ lists
- scan backwards, undoing ops
  by the ‘undo’-list transactions
- go to most recent checkpoint
- scan forward, redoing ops by the ‘redo’-list xacts

T4 commit

T3 commit

T2 commit

T1 commit

<checkpoint>

<checkpoints>

<checkpoints>

<checkpoints>

W.A.L. - w/ concurrent xacts

- recovery algo?
- undo uncommitted ones
- redo ones committed
  after the last checkpoint

- Eg.?

Recovery algo:
- build ‘undo’ and ‘redo’ lists
- scan backwards, undoing ops
  by the ‘undo’-list transactions
- go to most recent checkpoint
- scan forward, redoing ops by the ‘redo’-list xacts

T4 commit

T3 commit

T2 commit

T1 commit

<checkpoint>

<checkpoints>

<checkpoints>

<checkpoints>

W.A.L. - w/ concurrent xacts

- recovery algo?
- undo uncommitted ones
- redo ones committed
  after the last checkpoint

- Eg.?

Recovery algo:
- build ‘undo’ and ‘redo’ lists
- scan backwards, undoing ops
  by the ‘undo’-list transactions
- go to most recent checkpoint
- scan forward, redoing ops by the ‘redo’-list xacts

T4 commit

T3 commit

T2 commit

T1 commit

<checkpoint>

<checkpoints>

<checkpoints>

<checkpoints>
W.A.L. - w/ concurrent xacts

Observations:
- during checkpoints: assume that no changes are allowed by xacts (otherwise, "fuzzy checkpoints")
- recovery algo: is idempotent (i.e., can work, even if there is a failure during recovery)
- how to handle buffers of stable storage?

Checkpoints - more details

- when and how often to do check-points?

Checkpoints - more details

- A1: TCO: x-act oriented: at EOT (~FORCE)
- A2: TCC: x-act consistent (global checkpoints):
  - complete pending updates & block new xacts
  - flush
  - continue
- A3: ACC: action consistent

Checkpoints - more details

- write on log: <begin ck>
- flush pages
- write on log: <end ck>

Checkpoints - more details

- when and how often to do check-points?
- A1: TCO: x-act oriented: at EOT (~FORCE)

Checkpoints - more details

- A4: fuzzy checkpoints (see book)
Overview - recovery

- problem definition
- crash recovery
  - solution #1: Write-ahead log
  - (solution #2: shadow paging)

Shadow paging

- (ATOMIC)
- keep old pages on disk
- write updated records on new pages on disk
- if successful, release old pages; else release ‘new’ pages
- not used in practice - why not?

Shadow paging

- not used in practice - why not?
- may need too much disk space (“increase all by 5%”)
- may destroy clustering/contiguity of pages.

Detailed roadmap

- Problem definition - ACID properties
- Mapping hierarchy
- Crash recovery
  - Archive recovery
  - Conclusions

Other topics

- against loss of non-volatile storage: dumps of the whole database on stable storage.
- to recover: get the most recent dump; replay the log

Conclusions

- ¬ATOMIC, STEAL, ¬FORCE
- Write-Ahead Log, for loss of volatile storage,
  - with incremental updates,
  - and checkpoints.
- On recovery: undo uncommitted; redo committed transactions.
Conclusions

- On normal operation
  - UNDO info: write on log, before updates are flushed (‘propagated’)
  - REDO info: write on log, before EOT is ack-ed
- On recovery:
  - **undo** uncommitted;
  - **redo** committed transactions.

Addendum: ARIES

very popular recovery scheme

- ARIES paper (Mohan et al., ACM TODS, March 1992)

Excellent intro:


ARIES - main idea

- before ARIES:
  - for system crash (disk intact)
    - go backwards; undo uncommitted
    - go forward: redo committed (carefully)
  - for disk crash, (‘archive recovery’)
    - get a dump
    - go forward and do redo

ARIES overview

- Main idea: repeat history
  - supports STEAL / NO FORCE, fuzzy checkpoints
- Pages are tagged with Log Seq. Number (LSN)
  - = last log record applied to page
  - log records of a xact are linked backwards
ARIES overview

- checkpoints have
  - list of active transactions
  - list of dirty pages and LSNs that dirtied them

ARIES - ph1

three pass algorithm

- ANALYSIS
  - which pages could have been dirty @ crash
  - uncommitted xacts
  - where to start redo pass (firstLSN)

ARIES - ph2

- REDO: repeat history
  - updates for all transactions are redone
  - scan log forward from firstLSN
  - save work by only redoing what needs to be redone

ARIES - ph3

- UNDO:
  - remove uncommitted updates
  - undo is unconditional
  - undo writes CLR (compensation log record), in case we have a crash during the undo (!!)

Conclusions

ARIES: very popular (IBM DB2, MS SQLServer, etc)