CS 5614: (Big) Data Management Systems

B. Aditya Prakash

Lecture #8: Logging and Recovery 1
General Overview

- Preliminaries
- Write-Ahead Log - main ideas
- (Shadow paging)
- Write-Ahead Log: ARIES
NOTICE:

- **NONE** of the methods in this lecture is used ‘as is’
- we mention them for clarity, to illustrate the concepts and rationale behind ‘ARIES’, which is the **industry standard**.
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
Overview - recovery

- problem definition
  - types of failures
  - types of storage
- solution#1: Write-ahead log - main ideas
  - deferred updates
  - incremental updates
  - checkpoints
- (solution #2: shadow paging)
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 (e.g., main memory)
- non-volatile (e.g., 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**
Problem definition

- Records are on disk
- for updates, they are copied in memory
- and flushed back on disk, at the discretion of the O.S.! (unless forced-output: ‘output(B)’ = fflush())
Problem definition - eg.:

\[ \text{read}(X) \]
\[ X = X + 1 \]
\[ \text{write}(X) \]
Problem definition - eg.:

read(X)

→ X = X + 1

write(X)
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.:

\[
\begin{align*}
\text{read}(X) \\
\text{read}(Y) \\
X &= X + 1 \\
Y &= Y - 1 \\
\text{write}(X) \\
\rightarrow \text{write}(Y)
\end{align*}
\]

Q2: so what?
Q3: how to guard against it?
Overview - recovery

- problem definition
  - types of failures
  - types of storage

- solution#1: Write-ahead log - main ideas
  - deferred updates
  - incremental updates
  - checkpoints

- (solution #2: shadow paging)
Solution #1: W.A.L.

- 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:
  
  \(<\text{T1 start}>\)
  
  \(<\text{T1, X, 5, 6}>\)
  
  \(<\text{T1, Y, 4, 3}>\)
  
  \(<\text{T1 commit}>\) (or \(<\text{T1 abort}>\))
W.A.L. - intro

- in general: transaction-id, data-item-id, old-value, new-value
- (assumption: each log record is immediately flushed on stable store)
- each transaction writes a log record first, before doing the change
- when done, write a <commit> record & exit
W.A.L. - deferred updates

- idea: prevent OS from flushing buffers, until (partial) ‘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.?

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
<T1 commit>

crash
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.?

before
<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>

crash
W.A.L. - deferred updates

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

- Before crash:
  - <T1 start>
  - <T1, W, 1000, 2000>
  - <T1, Z, 5, 10>

- After crash:
W.A.L. - deferred updates

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

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>

crash
W.A.L. - deferred updates

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

- log records have ‘old’ and ‘new’ values.
- modified buffers can be flushed at any time

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.?

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
<T1 commit>

crash
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.?

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>

before

crash
W.A.L. - incremental updates

- Q: recovery algo?
- A:
  - redo committed xacts
  - undo uncommitted ones
- (more details: soon)
High level conclusion:

- Buffer management plays a key role
- FORCE policy: DBMS immediately forces dirty pages on the disk (easier recovery; poor performance)
- STEAL policy == ‘incremental updates’ : the O.S. is allowed to flush dirty pages on the disk
Buffer Management summary

No UNDO

No Force

No Steal

Steal

Fastest

Slowest

Force

No REDO

UNDO

REDO

Performance Implications

Logging/Recovery Implications
W.A.L. - incremental updates

Observations

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

before
<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>

crash
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?

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
...
<T500, B, 10, 12>

before

crash
W.A.L. - check-points

Q: should we write anything on the log?
A: yes!

Q: how does it help us?

before

<T1 start>
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
<checkpoint>
...
<checkpoint>
<T500, B, 10, 12>

crash
W.A.L. - check-points

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

<T1 start>
... 
<T1 commit>
... 
<T499, C, 1000, 1200> 
<checkpoint>
<T499 commit>
<T500 start>
<T500, A, 200, 400>  
<checkpoint>
<T500, B, 10, 12>

before crash

crash
W.A.L. - check-points

Q: how does it help us?
I.e., how is the recovery algorithm?

<T1 start>
...
<T1 commit>
...
<T499, C, 1000, 1200>
<checkpoint>
<T499 commit>
<T500 start>
<T500, A, 200, 400>
<checkpoint>
<T500, B, 10, 12>

before

crash
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

Assume: strict 2PL
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

<T1 start>
<T499 commit>
<T500 start>
<T500, A, 200, 400>
<T300 commit>
<T500 abort>
<T500, B, 10, 12>
W.A.L. - w/ concurrent xacts

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

<T1 start>
...
<T300 start>
...
<T300 commit>
<T500 start>
<T500, A, 200, 400>
<T300 commit>
<T500, B, 10, 12>

before
W.A.L. - w/ concurrent xacts

- recovery algo?
- undo uncommitted ones
- redo ones committed after the last checkpoint
- Eg.?
W.A.L. - w/ concurrent xacts

-recovery algo? specifically:
- find latest checkpoint
- create the ‘undo’ and ‘redo’ lists
W.A.L. - w/ concurrent xacts

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint>
<T3 start>
<T2 commit>
<checkpoint>
<T3 commit>

ck ck crash

T1 T2 T3 T4

Prakash 2017
VT CS 5614
W.A.L. - w/ concurrent xacts

<checkpoint> should also contain a list of 'active' transactions (= not committed yet)

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint>
<T3 start>
<T2 commit>
<checkpoint>
<T3 commit>
W.A.L. - w/ concurrent xacts

<checkpoint> should also contain a list of ‘active’ transactions

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3}>
<T3 commit>
W.A.L. - w/ concurrent xacts

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

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3}>
<T3 commit>
W.A.L. - w/ concurrent xacts

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

Actual ARIES algorithm: more clever (and more complicated) than that

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3}>
<T3 commit>
W.A.L. - w/ concurrent xacts

Observations/Questions
1) what is the right order to undo/redo?
2) during checkpoints: assume that no changes are allowed by xacts (otherwise, ‘fuzzy checkpoints’)
3) recovery algo: must be idempotent (ie., can work, even if there is a failure during recovery!
4) how to handle buffers of stable storage?

<T1 start>
<T2 start>
<T4 start>
<T1 commit>
<checkpoint {T4, T2}>
<T3 start>
<T2 commit>
<checkpoint {T4,T3} >
<T3 commit>
Observations

ARIES (coming up soon) handles all issues:
1) redo everything; undo after that
2) ‘fuzzy checkpoints’
3) idempotent recovery
4) buffer log records;
   – flush all necessary log records before a page is written
   – flush all necessary log records before a x-act commits
Overview - recovery

- problem definition
  - types of failures
  - types of storage
- solution#1: Write-ahead log
  - deferred updates
  - incremental updates
  - checkpoints

(solution #2: shadow paging)
Shadow paging

- keep old pages on disk
- write updated records on new pages on disk
- if successful, release old pages; else release ‘new’ pages
- tried in early IBM prototype systems, but
- 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.
Other topics

- against loss of non-volatile storage: dumps of the whole database on stable storage.
Conclusions

- Write-Ahead Log, for loss of volatile storage, with incremental updates (STEAL, NO FORCE)
- and checkpoints
- On recovery: **undo** uncommitted; **redo** committed transactions.
Next time:

ARIES, with full details on

– fuzzy checkpoints
– recovery algorithm