

Transaction Processing: Recovery

CPS 216
Advanced Database Systems

Announcements

- ❖ Homework #4 due in 7 days (April 23)
- ❖ Recitation session this Friday (April 18)
 - Homework #4 Q&A
- ❖ Project demo period starting in 12 days (April 28)
 - Sign-up sheet will be available next Monday
- ❖ Final exam in 17 days (May 1, 2-5pm)
 - More details will be available next Monday

Review

❖ ACID

- Atomicity
- Consistency
- Isolation → Concurrency control
- Durability → Recovery

Execution model



- ❖ Before it can be operated upon, disk-resident data must first be brought into memory
 - ~~input(X): copy the disk block containing object X to memory~~
 - ~~$v = \text{read}(X)$: read the value of X into a local variable v~~
 - Execute ~~input(X)~~ first if necessary → Issued by transactions
 - ~~write(X, v): write value v to X in memory~~
 - Execute ~~input(X)~~ first if necessary → Issued by DBMS
 - ~~output(X): write the memory block containing X to disk~~

Failures

- ❖ System crashes in the middle of a transaction T ; partial effects of T were written to disk
 - How do we undo T (atomicity)?
- ❖ System crashes right after a transaction T commits; not all effects of T were written to disk
 - How do we complete T (durability)?
- ❖ Media fails; data on disk corrupted
 - How do we reconstruct the database (durability)?

Naïve approach

- ❖ Force: When a transaction commits, all writes of this transaction must be reflected on disk
 - Without force, if system crashes right after T commits, effects of T will be lost
 - ☞ Problem: Lots of random writes hurt performance
- ❖ No steal: Writes of a transaction can only be flushed to disk at commit time
 - With steal, if system crashes before T commits but after some writes of T have been flushed to disk, there is no way to undo these writes
 - ☞ Problem: Holding on to all dirty blocks requires lots of memory

Logging

7

❖ Log

- Sequence of log records, recording all changes made to the database
- Written to stable storage (e.g., disk) during normal operation
- Used in recovery

❖ One change turns into two—bad for performance?

- But writes are sequential (append to the end of log)
- Can use dedicated disk(s) to improve performance

Undo logging

8

Basic idea

- ❖ Every time something is modified on disk, record its old value in the log
- ❖ If system crashes, undo the writes of partially executed transactions by restoring the old values

No steal?

- ❖ Not needed because we can undo; can flush dirty blocks before commit

Force?

- ❖ Still needed; otherwise effects of committed transactions can be lost after crash

Undo logging example

9

T_1 (balance transfer of \$100 from A to B)

$a = \text{read}(A); a = a - 100;$

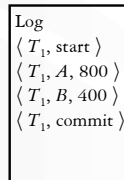
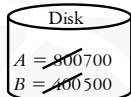
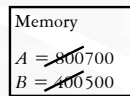
$\text{write}(A, a);$

$b = \text{read}(B); b = b + 100;$

$\text{write}(B, b);$

$\text{output}(A);$

$\text{output}(B);$



One technicality

10

T_1 (balance transfer of \$100 from A to B)

$a = \text{read}(A); a = a - 100;$

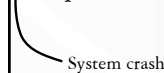
$\text{write}(A, a);$

$b = \text{read}(B); b = b + 100;$

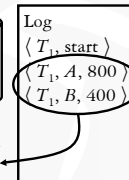
$\text{write}(B, b);$

$\text{output}(A);$

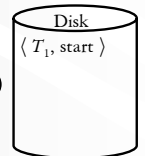
$\text{output}(B);$



Haven't been flushed yet



Log is first written to memory—when is flushing needed?



WAL

11

❖ Recap of the situation to be avoided

- T_1 has not completed yet
- A is modified on disk already
- But there is no log record for A
- Cannot undo the modification of A !

❖ Solution: WAL (Write-Ahead Logging)

- Before any database object X is modified on disk, the log record pertaining to X must be flushed

Undo logging: normal operations

12

❖ For every write, generate an undo log record containing the old value being overwritten

$\langle T_i, X, \text{old_value_of_}X \rangle$

▪ Typically (assuming physical logging)

- T_i : transaction id
- X : physical address of X (block id, offset)
- $\text{old_value_of_}X$: bits

❖ Also log $\langle T_i, \text{start} \rangle$, $\langle T_i, \text{commit} \rangle$, $\langle T_i, \text{abort} \rangle$

❖ WAL

- ❖ Force: before the commit log record of T_i is flushed, all writes of T_i must be reflected on disk

Undo logging: recovery

13

- ❖ Identify U , the set of active transactions at time of crash
 - Log contains $\langle T, \text{start} \rangle$, but neither $\langle T, \text{commit} \rangle$ nor $\langle T, \text{abort} \rangle$
- ❖ Process log backward (why?)
 - For each $\langle T, X, \text{old_value} \rangle$ where T is in U , issue $\text{write}(X, \text{old_value})$, $\text{output}(X)$ (why flush?)
- ❖ For each T in U , append $\langle T, \text{abort} \rangle$ to the end of the log (why?)

Additional issues with undo logging

14

- ❖ Failure during recovery?
 - No problem, run recovery procedure again
 - Undo is idempotent!
- ❖ Can you truncate log?
 - Yes, after a successful recovery
 - Or, truncate any prefix that contain no log records for active transactions

Redo logging

15

Basic idea

- ❖ Every time something is modified on disk, record its new value in the log
- ❖ If system crashes, redo the writes of committed transactions and ignore those that did not commit

Force?

- ❖ Not needed because we can redo; commit does not trigger writing of dirty blocks

No steal?

- ❖ Still needed; otherwise there is no way to roll back changes made by incomplete transactions

Redo logging example

16

T_1 (balance transfer of \$100 from A to B)

$a = \text{read}(A); a = a - 100;$

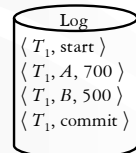
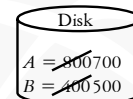
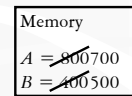
$\text{write}(A, a);$

$b = \text{read}(B); b = b + 100;$

$\text{write}(B, b);$

$\text{output}(A);$

$\text{output}(B);$



Redo logging: normal operations

17

- ❖ For every write, generate a redo log record containing the new value being written $\langle T_i, X, \text{new_value_of_}X \rangle$
- ❖ Also log $\langle T_i, \text{start} \rangle$, $\langle T_i, \text{commit} \rangle$, $\langle T_i, \text{abort} \rangle$
- ❖ WAL
- ❖ No steal
 - Before the commit log record of T_i is flushed, no writes of T_i can be flushed to disk
 - Requires keeping all dirty blocks in memory before commit

Redo logging: recovery

18

- ❖ Identify C , the set of all committed transactions (those with commit log records in log)
- ❖ Process log forward
 - For each $\langle T, X, \text{new_value} \rangle$ where T is in C , issue $\text{write}(X, \text{new_value})$ ($\text{output}(X)$ is optional; why?)
- ❖ For each incomplete transaction T (with neither commit nor abort log record), append $\langle T, \text{abort} \rangle$ to the end of the log

Additional issues with redo logging 19

- ❖ Failure during recovery?
 - No problem—redo is idempotent!
- ❖ Extremely slow recovery process!
 - I transferred the balance last year...
- ❖ Can you truncate log?
 - No, unless ...

Checkpointing 20

- ❖ Naïve approach
 - Stop accepting new transactions (lame!)
 - Finish all active transactions
 - Take a database dump
 - Now safe to truncate the redo log
- ☞ Alternative: fuzzy checkpointing (more later)

Summary of redo and undo logging 21

- ❖ Undo logging—immediate write
 - Force
 - Excessive disk I/Os
 - Imagine many small transactions updating the same block
- ❖ Redo logging—deferred write
 - No steal
 - High memory requirement
 - Imagine a big transaction updating many blocks

Logging taxonomy 22

	No steal	Steal
Force	No logging	Undo logging
No force	Redo logging	Undo/redo logging

↓
Next!

Undo/redo logging: normal operations 23

- ❖ Log both old and new values
 - ⟨ $T_i, X, old_value_of_X, new_value_of_X$ ⟩
- ❖ Also log ⟨ $T_i, start$ ⟩, ⟨ $T_i, commit$ ⟩, ⟨ $T_i, abort$ ⟩
- ❖ WAL
- ❖ Steal: If chosen for replacement, modified memory blocks can be flushed to disk anytime
- ❖ No force: When a transaction commits, modified memory blocks are not forced to disk
- ☞ Buffer manager has complete freedom!

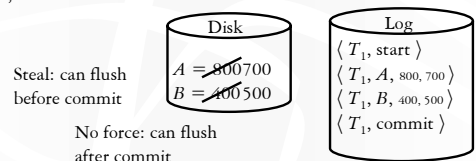
Undo/redo logging example 24

T_1 (balance transfer of \$100 from A to B)

```

a = read(A); a = a - 100;
write(A, a);
b = read(B); b = b + 100;
write(B, b);
commit;
    
```

Memory
~~A = 800700~~
~~B = 400500~~



No restriction on when memory blocks can/should be flushed

Fuzzy checkpointing

25

- ❖ Determine S , the set of currently active transactions, and log \langle begin-checkpoint $S \rangle$
- ❖ Write to disk all memory blocks currently dirty
 - Regardless whether they are written by committed or uncommitted transactions (but do follow WAL)
 - Blocks that become dirty after begin-checkpoint do not need to be flushed
- ❖ Log \langle end-checkpoint \rangle
- ❖ Between begin and end, continue processing old and new transactions

Undo/redo logging: recovery

26

- ❖ Scan log from the last \langle start-checkpoint $S \rangle$ with a matching \langle end-checkpoint \rangle to find
 - C , the set of committed transactions since last checkpoint
 - A , the set of active transactions at the time of crash
 - ☞ C and A may contain transactions that started after checkpoint
- ❖ Scan forward from that start-checkpoint to end of the log, and redo transactions in C
 - No need to look before start-checkpoint for redo (why?)
- ❖ Scan the log backward to undo transactions in A
 - May scan past start-checkpoint (why?)
 - Optimization: each log record stores a pointer to the previous log record for the same transaction; follow the pointer during undo
- ❖ Append $\langle T, \text{abort} \rangle$ to the log for each T in A

Physical vs. logical logging

27

- ❖ Physical logging (what we have assumed so far)
 - Log before and after images of data
 - ❖ Logical logging
 - Log operations (e.g., insert a row into a table)
 - Smaller log records
 - An insertion could cause rearrangement of things on disk
 - Or trigger hundreds of other events
 - Sometimes necessary
 - Assume row-level rather than page(block)-level locking
 - Data might have moved to another block at time of undo!
 - Much harder to make redo/undo idempotent
- ☞ See solution offered by ARIES

ARIES

28

"ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," by Mohan et al. *TODS* 1992

- ❖ Same basic ideas: steal, no force, WAL
- ❖ Three phases: analysis, redo, undo
 - Repeats history (redo even incomplete transactions)
- ❖ Better than our simple algorithm
 - CLR (Compensation Log Record) for transaction aborts
 - Redo/undo on an object is only performed when necessary → idempotency requirement lifted → logical logging supported
 - Each disk block records the LSN (log sequence number) of the last change
 - Can take advantage of a partial checkpoint
 - Recovery can start from any start-checkpoint, not necessarily one that corresponds to an end-checkpoint