Log-Based Failure Recovery

(when database management systems fail)

Objectives:

1) review transactional requirements of a DBMS
2) Log-based algorithms to support durability and atomicity

Reading: Ch. 17

Slide thanks: many from textbook web site

ACID Properties

(Hard Consistency)

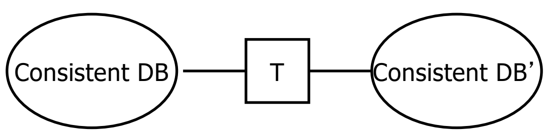
*A*tomicity: **all** actions of a transaction happen, or **none** happen.

*C*onsistency: if a transaction is consistent, and the database starts from a consistent state, then it will end in a consistent state.

*I*solation: the execution of one transaction is isolated from other transactions.

*D*urability: if a transaction commits, its effects persist in the database.

<u>Transaction:</u> collection of actions that preserve consistency



Integrity: Two Schools

Physical vs. Logical

use of locks vs. semantic predicates

Early: Locks won

Future: Semantics

- means (and increasing culture) re: semantics
- locks are less attractive in distributed computation

Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - $x \rightarrow y$ holds in R
  - Domain(x) = {Red, Blue, Green}
  - α is valid index for attribute x of R
  - no employee should make more than twice the average salary

Why define consistency wrt integrity constraints?

- Conventional database concurrency issues:
  →has a row been written --> a page dirty
- Thought to have been taken as far as it can go
- What if you know,
  - e.g. airplane flight #1234 has 56 available seats
  - if row is busy for one reservation, other reservations must wait.
  - you and I know
    - not have 56 concurrent reservations

NoSQL Took a Completely Different Track
(soft consistency)

- Usage: Hard vs. Soft consistency

NoSQL Took a Completely Different Track
(soft consistency)

- Usage: Hard vs. Soft consistency
- Soft consistency: *Eventual Consistency*
  - Recall, Cloud-native (e.g. HDFS), storage
    - Sharded (partitioned)
    - **Replicate the partitions** on different servers
      - Server can even be geographically distributed
  - DB Updates propagate to all copies
    - **Eventually,** all replicates get the update.

concurrency control -
integrate application semantics

- Consistent state: satisfies all constraints
- Consistent DB: DB in consistent state

Correctness (informally)

- If we stop running new transactions, when done, DB left consistent
- Each transaction sees a consistent DB
- Later we will formalize, a *serial* execution of a set of transactions will represent correctness.

Independent of the grounding of the theory:

Assume:
If T starts with consistent state
and T executes in isolation
⇒T leaves consistent state

I.e. programs are correct

D urability

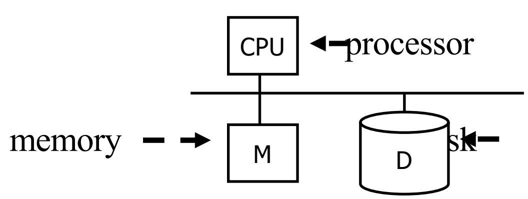
- Computers *Do* Break
- Disks are slow

Computer's break

- First order of business:
  Failure Model - (I won't formally define)

Events — Desired
Events — Undesired — Expected
Events — Undesired — Unexpected

Our failure model



Desired events: see product manuals….

Undesired expected events:
System crash
- memory lost
- cpu halts, resets

that's it!!

Undesired & Unexpected: Everything else!

Undesired Unexpected: Everything else!

Examples:

- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe….

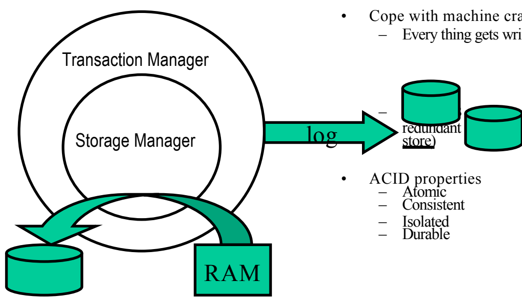
Disks are Slow

- Disk pages buffered (like virtual memory)
- Dirty pages written back to disk, (LRU)
  - No connection (until today) to
    - write to a log file
    - sequential execution
    - commits

DBMS Architecture, 2

Transaction Manager

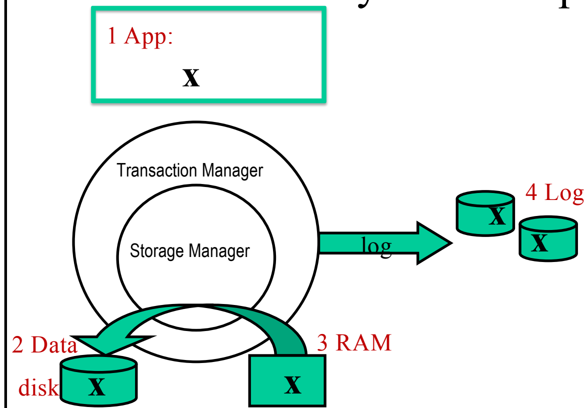
- Manage many users sharing a database, (speed)
- Cope with machine crashes
  - Every thing gets written at least 3 times.
  - write, also logged to redundant disks, (a.k.a. stable store)
- ACID properties
  - Atomic
  - Consistent
  - Isolated
  - Durable



DBMS Architecture

- Updates and transactions details logged
- Log data to a stable store
  - Smaller
  - Write only
  - Sequentially written
- Log used to repair to consistent states

Locations: By address space



Transaction Operations: (installment 1)

- Input (x): block with x → memory
- Output (x): block with x → disk
- Read (x,t): do input(x) if necessary
  t ← value of x in block
- Write (x,t): do input(x) if necessary
  value of x in block ← t
- Commit
- Abort or Roll back

// Operations will soon expand to include a transaction id.

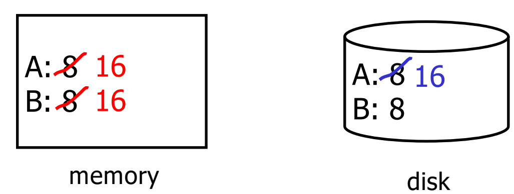
Central issue, machine breaks with an unfinished transaction (atomicity)

Example Constraint: A=B

T_1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B); failure!



- Need atomicity:

execute all actions of a transaction or none at all

Commit

Operational definition:

- A transaction does not commit until changes are written to [reliable] disk.

Where we are going:

- Exploit a sequentially written log file in stable store to support durability and atomicity
- Consider the interaction of writes
  - to the log
  - writes to disk (tables)
  - commit
- Algorithms
  1. undo logging *// start with strict requirement on larger behaviors.*
  2. redo logging *// then relax them*
  3. undo/redo logging
  4. non-quiescent checkpoints

First solution: undo logging (immediate modification)

due to: Hansel and Gretel, 782 AD

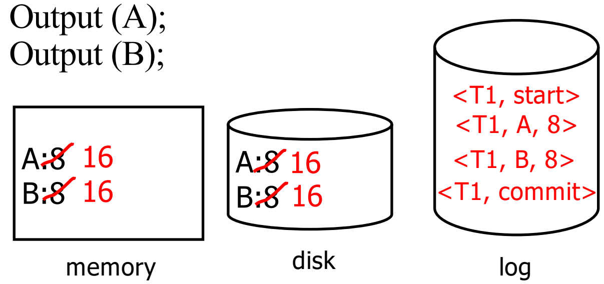
- Improved in 784 AD to durable undo logging

Transaction Log Entries

- <transaction-id, transaction-operation>
- Examples
  - <T1, A, 8> Transaction 1 write 8 to A
    // what about reads?
  - <T2, start>
  - <T2, commit>

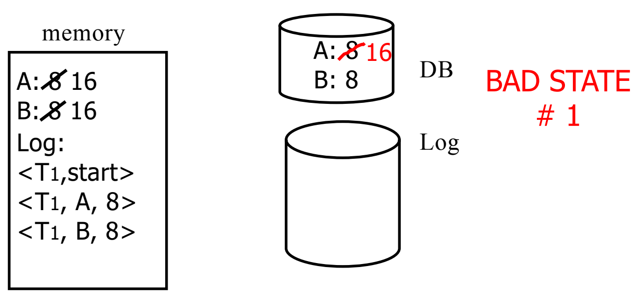
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

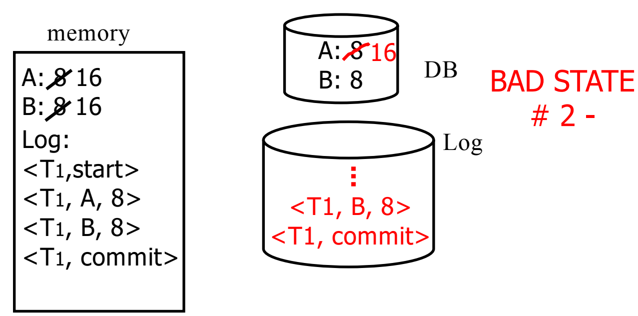


That was silly: Real memory organization and improvements

- Log is first written in memory //buffered
- Not written to disk on every action



- Log must be written before disk
- Disk must be written before commit



Summary: Undo logging rules

(1) For every action generate undo log record (containing old value)

(2) Before *x* is modified on disk, log records pertaining to *x* must be on disk (write ahead logging: WAL)

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk

Example [https://medium.com/brandons-computer-science-notes/second-class-test-for-databases-b1800ddaca3e]

Time	Transaction T_1	Transaction T_2	Log (buffer)	Log (disk)
0			<START T_1>	
1	read_item(X)			
2	X := X * 2			
3	write_item(X)		<T_1, X, 1>	
4			<START T_2>	
5		read_item(X)		
6	read_item(Y)			
7		X := X * 3		
8		write_item(X)	<T_2, X, 2>	
9	Y := X + Y			
10	write_item(Y)		<T_1, Y, 2>	
11	**flush_log**			
12	**output(X)**			
13	**output(Y)**			
14			<COMMIT T_1>	
15	**flush_log**			
16		**flush_log**		
17		**output(X)**		
18			<COMMIT T_2>	
19		**flush_log**		

Undo logging Recovery: Suppose

Time	Transaction T_1	Transaction T_2	Log
0			<START T_1>
1	read_item(X)		
2	X := X * 2		
3	write_item(X)		<T_1, X, 1>
4			<START T_2>
5		read_item(X)	
6	read_item(Y)		
7		X := X * 3	
8		write_item(X)	<T_2, X, 2>
9	Y := X + Y		
10	write_item(Y)		<T_1, Y, 2>
11	**flush_log**		

T I m e

Crash

- Nothing is committed
  - But the data disk could have new values on it
- Must restore the old values – go through the log backwards, and write old values; why backwards?

Undo logging Recovery: Suppose

Time	Transaction T_1	Transaction T_2	Log
0			<START T_1>
1	read_item(X)		
2	X := X * 2		
3	write_item(X)		<T_1, X, 1>
4			<START T_2>
5		read_item(X)	
6	read_item(Y)		
7		X := X * 3	
8		write_item(X)	<T_2, X, 2>
9	Y := X + Y		
10	write_item(Y)		<T_1, Y, 2>
11	**flush_log**		
12	**output(X)**		
13	**output(Y)**		
14			<COMMIT T_1>

T I m e

Crash

- T1 committed, but not T2
- Undo, on data disk, T2's (possible) updates, but not T1's

Undo logging Recovery:

(1) Let S = set of transactions with
<Ti, start> in log, but no
<Ti, commit> (or <Ti, abort>) record in log

(2) For each <Ti, X, v> in log,
in reverse order (latest → earliest) do:

- if Ti ∈ S then { write (X, v)
  - output (X) }

(3) For each Ti ∈ S do
- write <Ti, abort> to log

BTW: How far back in the log do we have to go?

Time	Transaction T_1	Transaction T_2	Log
0			<START T_1>
1	read_item(X)		
2	X := X * 2		
3	write_item(X)		<T_1, X, 1>
4			<START T_2>
5		read_item(X)	
6	read_item(Y)		
7		X := X * 3	
8		write_item(X)	$\boxed{\text{<}T_2\text{, X, 2>}}$
9	Y := X + Y		
10	write_item(Y)		<T_1, Y, 2>
11	**flush_log**		
12	**output(X)**		
13	**output(Y)**		
14			<COMMIT T_1>

T
I
m
e

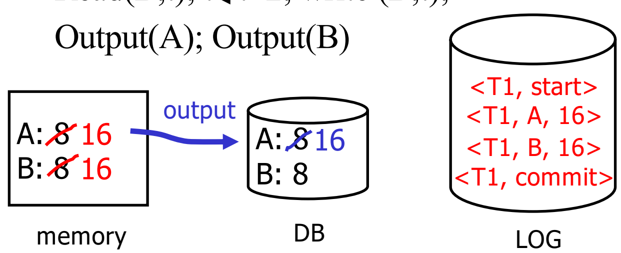
Crash

- T1 committed, but not T2
- Undo, on data disk, T2's (possible) updates, but not T1's

- Don't know how long a transaction may take, so
  - Back to the beginning
  - That could be a long time (i.e. a big log)

Redo logging (deferred modification)

T1: Read(A,t); t ← t×2; write (A,t);
Read(B,t); t ← t×2; write (B,t);
Output(A); Output(B)



Redo logging rules

(1) For every action, generate redo log record (containing new value)

(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on log.

(3) Flush log at commit

Redo logging rules

(1) For every action, generate redo log record (containing new value)

(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on log.

(3) Flush log at commit

Redo Logging Recovery:

(1) Let S = set of transactions with <Ti, commit> in log

(2) For each <Ti, X, v> in log, in forward order (earliest $\rightarrow$ latest) do:

- if Ti $\in$ S then $\begin{cases} \text{Write(X, v)} \\ \text{Output(X)} \end{cases}$ ◄ optional

Redo Recovery

Transaction T_1	Transaction T_2	Log (buffer)	Log (dis
		<START T_1>	
read_item(X)			
X := X * 2			
write_item(X)		<T_1, X, 2>	
		<START T_2>	
	read_item(X)		
read_item(Y)			
	X := X * 3		
	write_item(X)	<T_2, X, 6>	
Y := X + Y			
write_item(Y)		<T_1, Y, 4>	
		<COMMIT T_1>	
flush_log			
output(X)			
output(Y)			

Crash

T1 committed, T2 not committed

- Redo T1 writes to data disk, T2, do nothing
- Direction?
  - Run backwards, collecting commits, Go forward redoing

Redo Logging Recovery:

(1) Let S = set of transactions with
<Ti, commit> in log

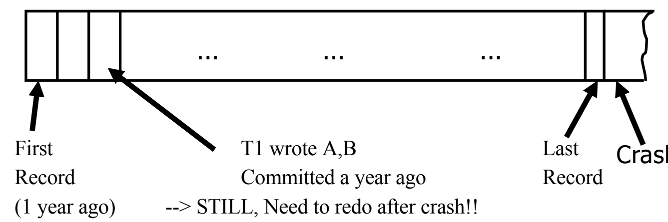
(2) For each <Ti, X, v> in log, in forward order (earliest → latest) do:

- if Ti ∈ S then $\begin{cases} \text{Write(X, v)} \\ \text{Output(X)} \end{cases}$ ◄-- optional

Recovery is very, SLOW:

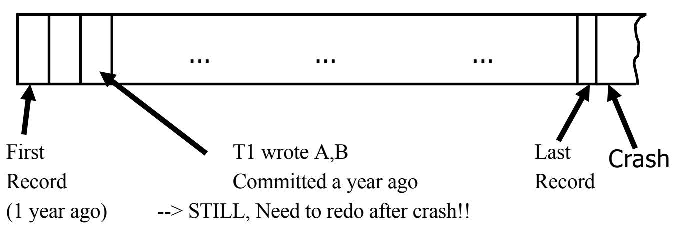
must consider the whole log

Redo log:



But: we also have the whole version history of the database and can reconstruct the database

Redo log:



Checkpoint: n. A point in a log file representing a point in time when the disks and the log are synchronized v: to create a checkpoint

Periodically: (simple version)

(1) Do not accept new transactions
(2) Wait until all transactions finish
(3) Flush all log records to disk (log)
(4) Flush all buffers to disk (DB) (do not discard buffers)
(5) Write “checkpoint” record on disk (log)
(6) Resume transaction processing

Making Backups

- Make database backup upon checkpoint
  - Make an entry in the log
    - Enable matching checkpoint to the backup
- If we lose the data disk,
    - Restore a disk from backup
    - Then use the log to complete putting committed data values on the disk

Imperfect ingredients:

- *Undo logging:* cannot bring backup DB copies up to date
- *Redo logging:* need to keep all modified blocks in memory until commit
- Commit --> stopping the data subsystem

Solution (1): undo/redo logging!

Update ⇒ record both the old and new values:

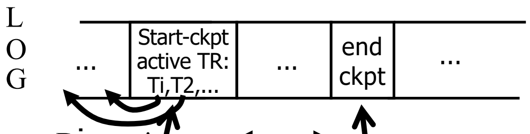
given data on page X

<Ti, Xid, New X val, Old X val>

Undo/redo recovery:

- Page X can be flushed before or after Ti commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

Non-quiescent checkpoint



Determine
all transactions
in progress
record in log

as dirty buffer pool pages
flushed, track when those
transactions have committed
when all have committed,
record in log

Recovery process:

- Backwards pass (go only as far as the latest checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Undo pending transactions
  - follow undo chains for transactions in (checkpoint active list) - S
- Forward pass redo actions of S transactions