CS 245: Database System Principles

Notes 08: Failure Recovery

Hector Garcia-Molina
(Some modifications by Chris Olston)

PART II

- Crash recovery (2 lectures) Ch.8
- Concurrency control (3 lectures) Ch.9
- Transaction processing (2 lects) Ch.10
- Information integration (1 lect) Ch.11

ACID Properties

- Atomicity
  - Actions are never left partially executed
- Consistency
  - Actions leave the DB in a consistent state
- Isolation
  - Actions are not affected by other concurrent actions
- Durability
  - Effects of completed actions are resilient against system failures

Integrity or correctness of data

A
Consistency
I
D

- Would like data to be "accurate" or "correct" at all times

EMP

Name	Age
White	52
Green	3421
Gray	1

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

Definition:

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

Constraints (as we use here) may not capture "full correctness"

Example 1 Transaction constraints

- When salary is updated,
  new salary > old salary
- When account record is deleted,
  balance = 0

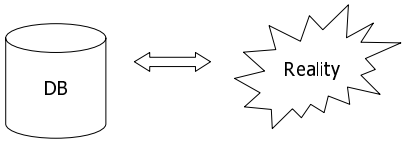
Note: could be "emulated" by simple constraints, e.g.,

account

Acct #	....	balance	deleted?

Constraints (as we use here) may not capture "full correctness"

Example 2 Database should reflect real world



☞ in any case, continue with constraints...

Observation: DB cannot be consistent always!

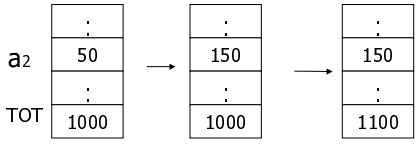
Example: $a_1 + a_2 + \ldots a_n = TOT$ (constraint)

Deposit \$100 in a_2: $a_2 \leftarrow a_2 + 100$; $TOT \leftarrow TOT + 100$

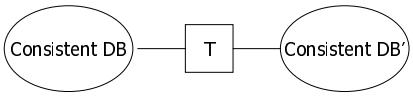
Example: $a_1 + a_2 + \ldots a_n = TOT$ (constraint)

Deposit \$100 in a_2: $a_2 \leftarrow a_2 + 100$

$TOT \leftarrow TOT + 100$



Transaction: collection of actions that preserve consistency



Big assumption:

If T starts with consistent state +
T executes in isolation
⇒ T leaves consistent state

Correctness (informally)

- If we stop running transactions,
  DB left consistent
- Each transaction sees a consistent DB

How can constraints be violated?

- Transaction bug
- DBMS bug
- Hardware failure
  e.g., disk crash alters balance of account
- Data sharing
  e.g.: T1: give 10% raise to programmers
  T2: change programmers ⇒ systems analysts

How can we prevent/fix violations?

- Chapter 8: due to failures only
- Chapter 9: due to data sharing only
- Chapter 10: due to failures and sharing

Will not consider:

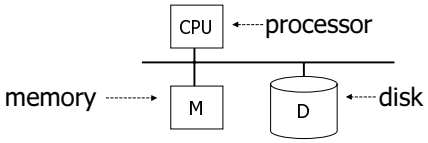
- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair
  That is, solutions studied here do not need to know constraints

Chapter 8 Recovery

- First order of business:
  Failure Model

Events — Desired
Undesired — Expected
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 catches fire
- Car smashes into data center
- (You get the idea …)

Is this model reasonable?

In our model, data is either okay or "lost"

Assumes we can detect corruption and treat it as lost data

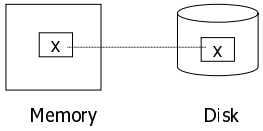
Detecting corruption

Approach: Add low level checks + redundancy to increase probability model holds

E.g.,
- Disk checksums
- Memory parity
- CPU checks: run code twice

Second order of business:

Storage hierarchy



Operations:

- 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

Failure modes:

- Undesired/expected: system crash
  - Data on disk still there on restart ← next
- Undesired/unexpected:
  media failure
  catastrophic failure
  - Data on disk/disks lost! ← later

System Crash

A
C
I
Durability

Problem #1 Transaction completed, but results only in memory

If system crashes, effects of transaction lost

Solution: Don't tell user the transaction has "committed" until all effects are on disk

System Crash

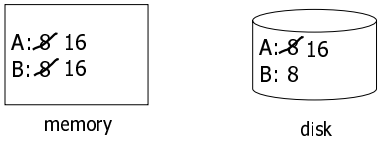
Problem #2 Unfinished transaction

Example Constraint: A=B

T_1: A ← A × 2

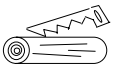
B ← B × 2

T_1: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B); crash!



Atomicity
C
I
D

- Need atomicity: execute all actions of a transaction or none at all
- Solution: logging



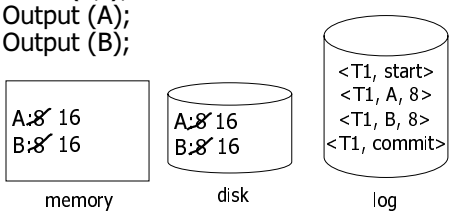
One variation: undo logging (immediate modification)

due to: Hansel and Gretel, 782 AD

- Improved in 784 AD to durable undo logging

Undo logging (Immediate modification)

T_1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

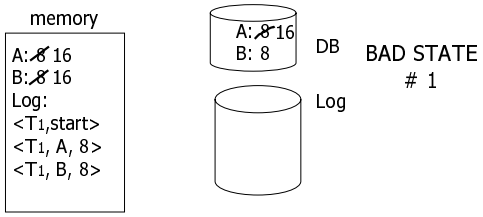


Assumption

- Logging "fakes" atomicity by undoing writes of partially completed xacts
- Assumption: writing a block is atomic
  - Atomicity of block writes is itself "faked" by low-level checks
  - And so it goes …
  - At the bottom, there must be some intrinsically atomic action (writing a bit?)

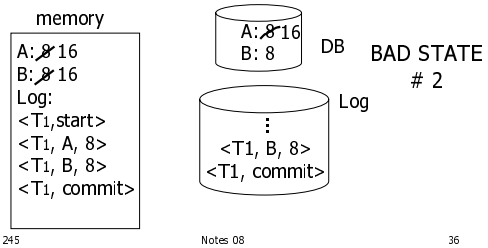
One "complication"

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



One "complication"

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



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

Recovery rules: Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti,commit> or <Ti,abort> in log, do nothing
  - Else { For all <Ti, X, v> in log: { write (X, v); output (X) }; Write <Ti, abort> to log }

➨IS THIS CORRECT??

Recovery rules: Undo logging

(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

What if failure during recovery?

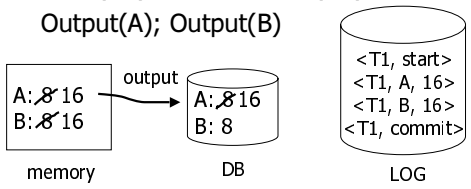
No problem! ⇒ Undo idempotent

To discuss:

- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures
- Catastrophic failures

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 disk

(3) Flush log at commit

Recovery rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:
    - Write(X, v)
    - Output(X)

➡IS THIS CORRECT??

Recovery rules: Redo logging

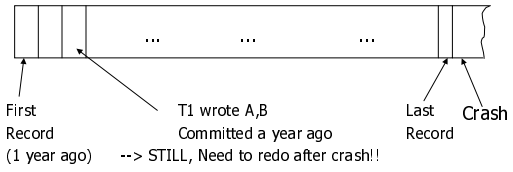
(1) Let S = set of transactions with <Ti, commit> in log

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

- if Ti ∈ S then
  - Write(X, v)
  - Output(X) ← optional

Recovery is very, very SLOW !

Redo log:



Solution: Checkpoint (simple version)

Periodically:

(1) Do not accept new transactions

(2) Wait until all transactions finish

(3) Flush all log records to disk (log)

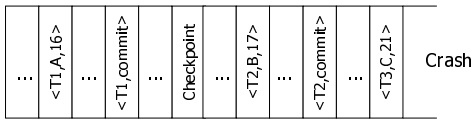
(4) Flush all buffers to disk (DB) (do not discard buffers)

(5) Write "checkpoint" record on disk (log)

(6) Resume transaction processing

Example: what to do at recovery?

Redo log (disk):



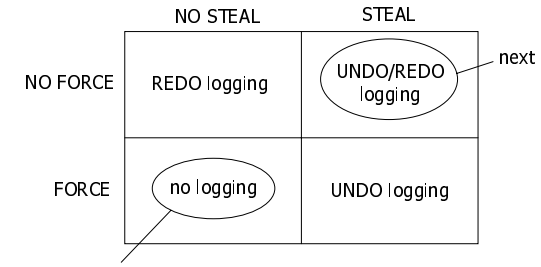
Better: Fuzzy Checkpointing

- Want to be able to checkpoint without bringing the system to a halt
- Especially if transactions run for a long time
- Solution: Nonquiescent checkpointing
  - Will get to this soon…

Key drawbacks (with respect to buffer management)

- *Undo logging:*
  must FORCE data to disk to commit
- *Redo logging:*
  need to keep all modified blocks in memory until commit
  (other xacts can't STEAL buffer pages)

Logging Taxonomy



Only works if xacts only modify one block, and no two concurrent xacts modify same block

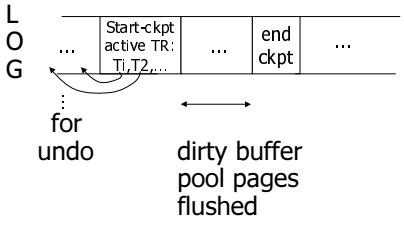
Solution: undo/redo logging!

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

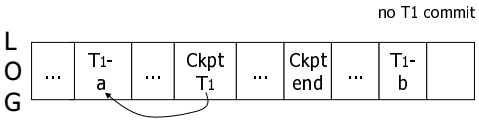
Undo/Redo Logging Rules

- Element X can be flushed before or after Ti commit
  (Buffer manager has complete freedom)
- Before flushing a page, all corresponding log records flushed (WAL)
- Flush log before commit

Non-quiesce checkpoint

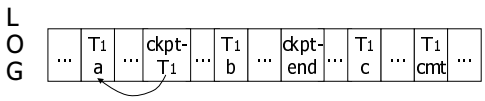


Examples what to do at recovery time?



➡ Undo T_1 (undo a,b)

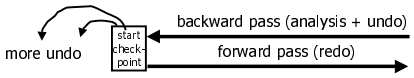
Example



➡ Redo T1: (redo b,c)

Recovery process:

- Backward pass (end of log ➜ latest checkpoint start)
  - construct set C of transactions that committed since checkpoint start
  - Undo actions of uncommitted transactions
- Finish undo (follow undo chains)
  - Undo all actions of transactions not in C
- Forward pass (latest checkpoint start ➜ end of log)
  - Redo actions of C transactions



Real world actions

E.g., fire missile

$T_i = a_1 a_2 \ldots\ldots a_j \ldots\ldots a_n$

Can't undo!
(Well, not yet anyway … ☺)

Solution

Execute real-world actions after commit

Works for actions that are idempotent

If missile already launched, "file missile" command has no effect (assuming one missile per launcher/silo)

Non-idempotent real world actions

E.g., dispense cash at ATM

$T_i = a_1 a_2 \ldots\ldots a_j \ldots\ldots a_n$

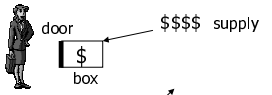
$

Can't undo!
("Please reinsert cash" ☺)

Not idempotent!
(Avoid dispensing cash twice)

Solution

Design system & transaction so final action is idempotent



What happens if crash here?

- Put cash in box
- Write commit record

here?

- Open door *(idempotent)*

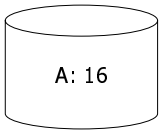
Failure modes:

- Undesired/expected: system crash
  - Data on disk still there on restart
  - Solution: atomicity via logging
- Undesired/unexpected:
  - media failure
  - catastrophic failure
  - Data on disk/disks lost! ← next

Media failure

A
C
I
Durability+

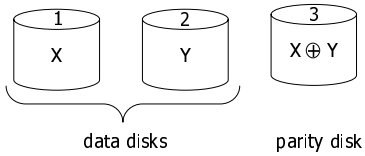
(loss of non-volatile storage)



Solution: Make copies of data!

Example 1 RAID

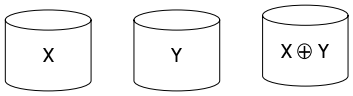
- Array of disks with redundancy
- RAID-4: one extra disk for "parity"
- Output(X) --> write to disk 1, update 3
- Input(X) --> read disk 1



Recovery in RAID-4

- If one disk fails, reconstruct it using the other two disks

⇨ Assumes bad data can be detected



Example #2 Fully redundant writes, Single reads

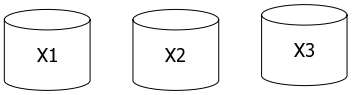
- Keep N copies on separate disks
- Output(X) --> N outputs
- Input(X) --> Input one copy
  - if ok, done
  - else try another one

⇨ Assumes bad data can be detected

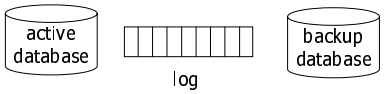
Example 3 Triple modular redundancy

- Keep 3 copies on separate disks
- Output(X) --> three outputs
- Input(X) --> three inputs + vote

⇨ Protects against undetected bad data



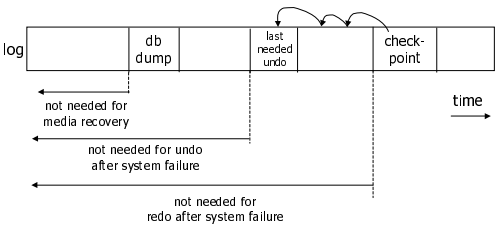
Example 4: DB Dump + Log



- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

☆ Can't use undo-only logging!

When can log be discarded?



Catastrophic Failure

- Array of disks won't help in case of:
  - Fire
  - Explosion
  - Earthquake
  - Godzilla
  - Also: vandalism, viruses

Solution

A
C
I
Durability++

- Geographically distributed copies!

Summary

- Consistency of data
- One source of problems: failures
  - Logging

  - Redundancy

- Another source of problems:
  Data Sharing..... next