CA420 Databases II

Centralized Recovery

Failures

Various types of failure:

- *transaction*, *system* and *media* failures

Transaction failures:

- *all of the failed transaction's effects* must be removed form the database system

System and media failures:

- the effects of all *aborted transactions must be removed* from the database system
- the effects of all *committed transactions must be restored* to the database system

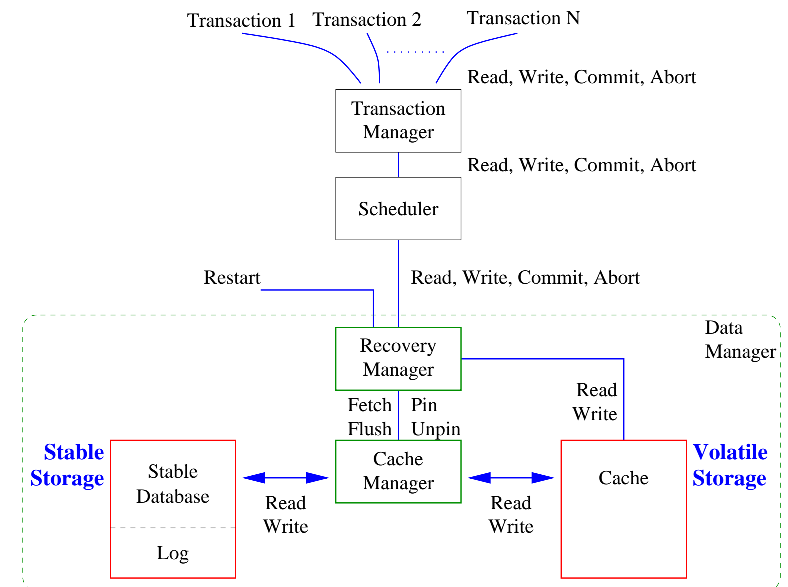
Idempotence of Restart

`Restart` must be *idempotent*:

- it must return the system to the same state,
  even in the presence of repeated failures during restart processing

As a result,
care is required writing updates to stable storage during `restart` processing

System Architecture – Reviewed



Stable and Volatile Storage

Stable storage:

- usually hard disk drives, often RAID devices
  low cost per megabyte
- unit of I/O is the disk block (often 512 bytes)
- I/O operations costly
  seek time, rotational delay, transfer time
- contents survive system failures and restarts

Volatile storage:

- usually random access main memory
  high cost per megabyte
- managed in terms of “pages” (often of 4KB or 16KB)
- contents do not survive system failures or restarts

Initially, assume “items” are 4KB or 16KB pages

Cache Manager

Cache manager:

- controls all data movement between stable and volatile storage
- manages a pool of 4KB or 16KB “cache slots”
- “dirty” slots:
  contain updates that have not yet been propagated to stable storage
- each cache slot has a “dirty” bit to indicate whether it is dirty:
  - slot becomes dirty when updated by transaction writes
  - slot becomes clean when read from or written to stable storage

Database operations (reads and writes):

- operate against copies of database pages in cache slots
- updates are propagated to stable storage under control of the cache manager and the recovery manager

Cache Manager Operations

Cache manager operations:

Read: allocated a cache slot and read page from stable storage

Write: force write updated page from cache slot back to stable database

Pin: hold page in cache slot
do not free it or write it back to stable storage

Unpin: release page in cache slot
becomes eligible to be freed or written back to stable storage

Dirty pages:

- depending upon the recovery algorithm,
  the cache manager may or may not be able to flush dirty pages to stable storage asynchronously

Atomicity:

- cache manager writes to stable storage are assumed to be atomic

Recovery Manager Operations

`Commit`:

- ensure that the transaction’s effects are made permanent

`Abort`:

- ensure that the transaction’s effects are removed from the database

`Restart`:

- return the system to a state in which:
  - the effects of all committed transactions are restored, and
  - the effects of all aborted and (previously) active transactions are removed from the database
- invoked on system startup, possibly following a failure

Classes of Recovery Manager

Recovery managers can be classified into two broad types

In-place updating:

- cache writes to stable storage overwrite previous values
- *write-ahead logging* (WAL) is used to preserve values necessary for recovery processing

Shadow paging:

- cache writes to stable storage are written to a new location
  previous values are preserved

Unless care is taken, shadow paging compromises locality on disk

Sequential versus Random Disk Operations

Disk operation costs:

- seek time, rotational delay, transfer time

Two disk operations to random blocks incur:

- two seek-time costs
- two rotational delays
- two transfer-time costs

Two disk operations to sequential blocks incur:

- one seek-time cost
- one rotational delay
- two transfer-time costs

Generally:
sequential operations substantially outperform random operations
particularly for synchronous operations

Undo and Redo Rules

Generally, a recovery method:

requires undo:
if it allows a transaction's updates to be recorded in the stable database before it commits

requires redo:
if it allows a transaction to commit before all its updates have been recorded in the stable database

Undo Rule:
the last committed value of x must be saved (elsewhere) on stable storage before being overwritten by an uncommitted value

Redo rule:
before a transaction can commit, the value it wrote for each data item must be in stable storage (somewhere)

Implications for the Cache Manager

If recovery manager cannot *redo* a transaction's effect:

- updated values must be installed in the stable database before a transaction commits
  (follows from redo rule)
- consequence:
  many possibly synchronous, random I/O operations

If recovery manager cannot *undo* a transaction's effect:

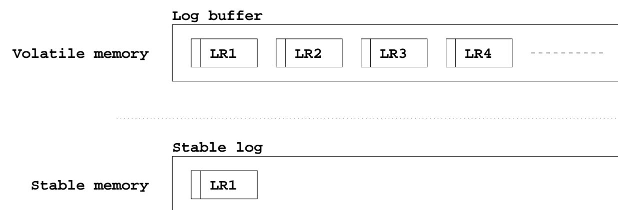
- updated values from uncommitted transaction may not be written to the stable database
  (follows from undo rule)
- cache manager cannot use in-place updating for dirty pages from non-committed transactions

Cache manager has most flexibility if recovery manager can both undo and redo transactions' effects

Write-Ahead Logging (WAL)

Updates are recorded on a log during normal processing:

- updates are recorded initially in a volatile log buffer
- only the stable log survives failures and system restarts



The stable log is a sequential file

Log Operations

Log sequence numbers (LSNs):

- log records are identified on disk by log sequence numbers
- LSNs are increasing
- usually identify a physical location on disk

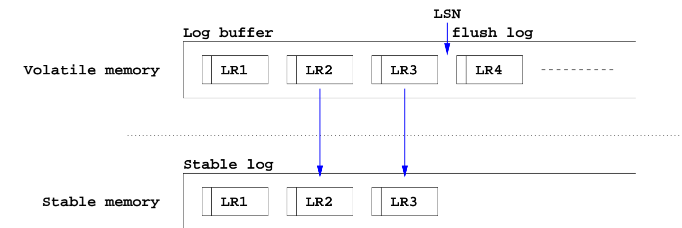
Log operations:

`Append`:
  append new record to log buffer, returning LSN of new record

`Flush`:
  flush log buffer to stable log up to given LSN

Flushing the Log

At key points,
the recovery manager must flush the log buffer to the stable log



It is always safe for the log manager to flush the log asynchronously

Log Records

Recovery methods
may require one or more of the following to be recorded on the log:

- T_i active
- T_i aborted
- T_i committed
- $[T_i, x, v]$
  - T_i wrote the value v into x

(many other types of log record are also used)

An Undo / Redo Recovery Algorithm

Next,
we describe a simple (actually, naïve) undo/redo recovery algorithm

Later, we will describe a more realistic algorithm

An Undo / Redo Recovery Algorithm – 1

`RM-Write`(T_i, x, v):

1. if x is not in the cache, then fetch it
2. append$[T_i, x, v]$ to the log buffer
3. write v into the cache slot for x
4. acknowledge processing of `RM-Write`(T_i, x, v) to the scheduler

Notes:

- `pin` and `un-pin` requests are implicit here in step (3)
  as is the setting of the cache slot's dirty bit
  (true throughout)
- at step (2), the log need not be flushed to disk

An Undo / Redo Recovery Algorithm – 2

`RM-Read`(T_i, x):

1. if x is not in the cache, then fetch it
2. return the value in x's cache slot to the scheduler

An Undo / Redo Recovery Algorithm – 3

`RM-Commit`(T_i):

1. add `Commit-`T_i to the log
2. flush the log to stable storage $\longleftarrow$ *this is the commit point!*
3. acknowledge the commit to the scheduler

Flushing the log at step (2) ensures both:

1. that the last-committed value is on stable storage
   (redo rule)
2. that the before image for subsequent transactions is on stable storage
   (undo rule)

An Undo / Redo Recovery Algorithm – 4

RM-Abort(T_i):

1. for each item x updated by T_i:
   (a) if x is not in the cache, allocate a slot for it
   (b) copy the before image of x for T_i into x's cache slot
2. acknowledge the abort to the scheduler

Notes:

- in step 1(a),
  x need not be fetched from the stable database
- in step 1(b),
  before image must be available *somewhere* on the log
- after step 1(b),
  x's value is only restored in the cache,
  it will restored in the stable database subsequently

An Undo / Redo Recovery Algorithm – 5

RM-Restart():

1. discard all cache slots
   set undone={}, redone={}, CL={}
   scan backwards through the log
2. for each Commit-T_i, add T_i to CL (commit list)
3. for each [T_i,x,v]:
   (a) skip if $x \in$ undone $\cup$ redone
   (b) allocate a cache slot for x
   (c) if T_i is in the commit list (CL):
      i. copy v into the cache slot for x
      ii. add x to redone
   (d) otherwise:
      i. copy *before image* of x into the cache slot for x
      ii. add x to undone
   (e) finished if undone $\cup$ redone equals all the items in the database

Undo / Redo Recovery Algorithm – Restart Notes

Notes:

- the commit list:
  - is regenerated by scanning the log
  - all non-committed transactions are assumed to have aborted
- "before image of x":
  - scan backwards through the log from [T_i,x,v]
  - locate the first record [T_j,x,u] encountered such that T_j is on the commit list
  - u is the before image of x
- last committed values of all updated items in the cache:
  - CM will later reinstall those values in the stable database asynchronously