Lecture 9

# Database Management Systems

Winter 2004

# CMPUT 391: Implementing Durability

Dr. Osmar R. Zaïane

University of Alberta

Chapter 25
of Textbook

Based on slides by Lewis, Bernstein and Kifer.

---

# Recovery and the ACID properties

Atomicity: "all or nothing"
Consistency: "DB starts consistent and ends consistent"
Isolation: "isolated from the effects of other transactions"
Durability: "effects of commited transaction persist"

- The **Recovery Manager** is responsible for ensuring *Atomicity* and *Durability*.
  - ***Atomicity*** is guaranteed by undoing the actions of the transactions that did not commit (aborted).
  - ***Durability*** is guaranteed by making sure that all actions of committed transactions survive crashes and failures.

---

# Types of Failures

- *Transaction failures*
  - overflow, interrupt, data not available, explicit rollback, concurrency enforcement, programming errors
  - no memory loss.
- *System crashes*
  - due to hardware or software errors
  - main memory content is lost
- *Media failures*
  - problems with disk head, unreadable media surface
  - (parts of ) information on secondary storage may be lost
- *Natural disasters*
  - fire, flood, earthquakes, theft, etc.
  - physical loss of all information on all media

---

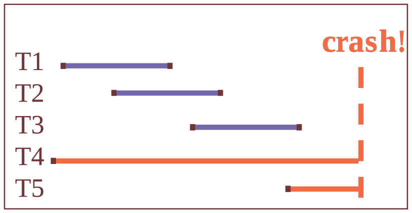
# General Idea

- If a transaction $T_i$ is aborted (e.g., for concurrency control reasons), all its actions have to be *undone*.
- Active transactions at the time of the crash have to be aborted, i.e., their effects have to be *undone* when the system comes back.

➢ DBMS has to maintain enough information to undo actions of transactions (the LOG File)

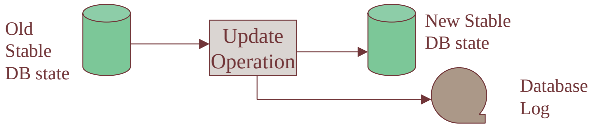
Desired Behavior after system restarts:

- T1, T2 & T3 should be durable.
- T4 & T5 should be rolled back, i.e., effects undone.



# Log

- Sequence of records (sequential file)
  - Modified by appending (no updating)
- Contains information from which database can be restored
- Log and database stored on different mass storage devices
- Often replicated to survive single media failure
- Contains valuable historical data not in database
  - How did database reach current state?



---

# Log

- Each modification of the database causes an *update record* to be appended to the log
- Update record contains:
  - Identity of data item modified
  - Identity of transaction (tid) that did the modification
  - *Before image* (undo record) - value of data item before update occurred
    - Referred to as *physical logging*

---

# Log

| | | | | | | |
|---|---|---|---|---|---|---|
| *x* | *y* | *z* | *u* | *y* | *w* | *z* |
| $T_1$ | $T_1$ | $T_2$ | $T_3$ | $T_1$ | $T_4$ | $T_2$ |
| 17 | A | 2.4 | 18 | ab | 3 | 4.5 |

- Update records in a log

---

# Transaction Abort Using Log

- Scan log backwards using *tid* to identify transaction's update records
- Reverse each update using before image
- In a strict system, new values are unavailable to concurrent transactions (as a result of long term exclusive locks); hence rollback makes transaction atomic
- **Problem**: terminating scan (log can be long)
- **Solution**: append *begin record* containing *tid* prior to first update record

# Transaction Abort Using Log

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | U | U | U | U | U | U | U |
| | $x$ | $y$ | $z$ | $u$ | $y$ | $w$ | $z$ |
| $T_1$ | $T_1$ | $T_1$ | $T_2$ | $T_3$ | $T_1$ | $T_4$ | $T_2$ |
| | 17 | A | 2.4 | 18 | ab | 3 | 4.5 |

abort $T_1$

Key:
B – begin record
U – update record

- Scan back to begin record to abort a transaction

---

# Crash Recovery Using Log

- Abort all transactions active at time of crash
- **Problem**: How do you identify them?
- **Solution**: *abort record* or *commit record* appended to log when transaction terminates
- Recovery Procedure:
  - – Scan log backwards - if first of T's records is update record, T was active at time of crash. Roll it back
    - Transaction not committed until commit record in log

---

# Crash Recovery Using Log

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| B | U | U | U | U | U | C | U | A | U |
| | $x$ | $y$ | $z$ | $u$ | $y$ | | $w$ | | $z$ |
| $T_1$ | $T_1$ | $T_1$ | $T_2$ | $T_3$ | $T_1$ | $T_3$ | $T_4$ | $T_1$ | $T_2$ |
| | 17 | A | 2.4 | 18 | ab | | 3 | | 4.5 |

crash

Key:
B – begin record
U – update record
C – commit record
A – abort record

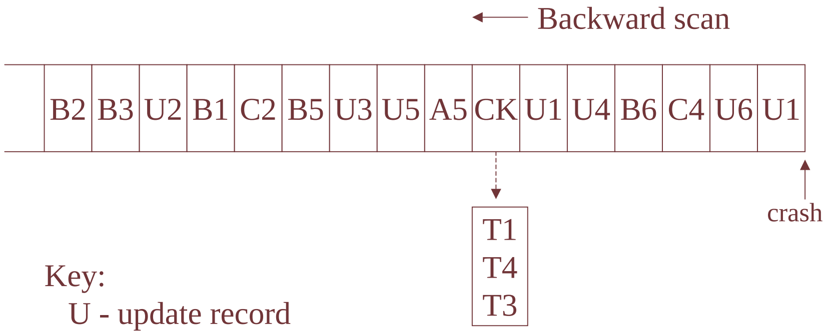
- $T_1$ and $T_3$ were not active at time of crash

---

# Crash Recovery Using Log

- **Problem**: Scan must retrace entire log
- **Solution**: Periodically append *checkpoint record* to log. Contains tid's of all active transactions at time of append
  - – Backward scan goes at least as far as last checkpoint record appended
  - – Transactions active at time of crash determined from log suffix that includes last checkpoint record
  - – Scan continues until those transactions have been rolled back

# Example



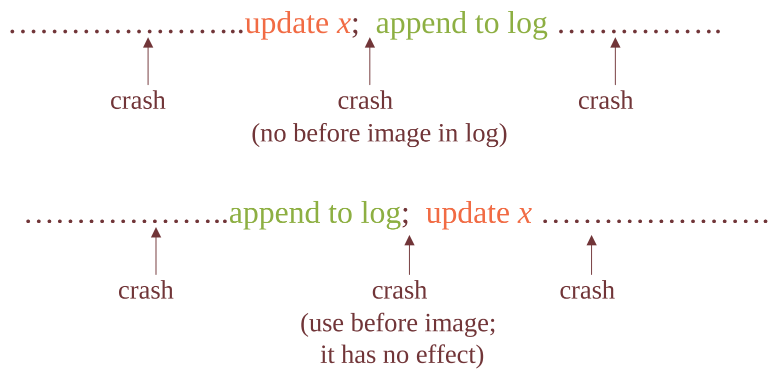
Key:
U - update record
B - begin record
C - commit record
A - abort record
CK - checkpoint record

$T_1$, $T_3$ and $T_6$ active
at time of crash

---

# Write-Ahead Log

- When $x$ is updated two writes must occur: update $x$ in database, append of update log record
  - Which goes first?



---

# Write-Ahead Logging

- An update record must always be appended to the Log before the database is updated on disk.
- The Write-Ahead Logging Protocol:

Must force the log record for an update *before* the corresponding data page gets to disk.

guarantees Atomicity (and Durability).

---

# Write-Ahead Log: Performance

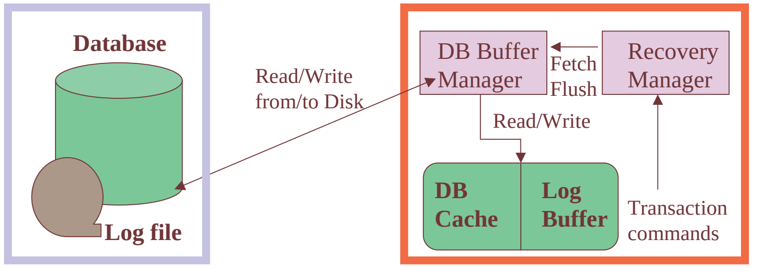
- **Problem**: two I/O operations for each database update
- **Solution**: log buffer in main memory
  - Extension of log on mass store
  - Periodically *flushed* to mass store
  - Flush cost pro-rated over multiple log appends

# Performance

- **Problem**: one I/O operation for each database access
- **Solution**: database page cache in main memory
  - Page is unit of transfer
  - Page containing requested item brought to cache; then copy of item transferred to application
  - Retain page in cache for future use
  - Check cache for requested item before doing I/O (I/O can be avoided)

---

# Basic Architecture


**Database**

**Log file**

Read/Write from/to Disk

DB Buffer Manager

Fetch

Flush

Recovery Manager

Read/Write

**DB Cache**

**Log Buffer**

Transaction commands

## Stable Storage

- ***Persistent storage***, loses content only if media fails or is otherwise lost
- Contains DB and Transaction Log
- Disks and other Media

## Main Memory

- “volatile” memory, loses content if system crashes
  ➔ DB Cache & Log Buffer may be lost.
- Different strategies for the Interaction Buffer Manager ↔ Recovery Manager

---

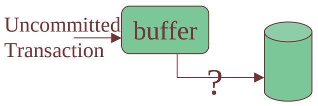
# The Role of the Database Buffer in Main Memory

- Database pages are read from disk, if needed, and put into the cache in main memory. They stay there until explicitly written back to disk.
- Read and Write operations of transactions are executed on pages in the cache! Cache pages that have been updated are marked *dirty*; others are *clean*.
- Changed pages may be kept in the buffer (for efficiency)
  - Update of the page is not reflected on disk immediately (saves write access to the disc)
  - Other transaction can read the value from the buffer (saves read access to the disc)
- Cache can hold several pages, but ultimately fills
  - Clean pages can simply be overwritten
  - Dirty pages must be written to DB before page frame can be reused

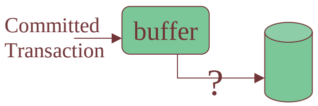
---

# Recovery Manager (*RM*) / Buffer Manager (*BM*) Interaction

- Can a BM decide to write some of it’s buffer pages (possibly changed by some uncommitted transaction) to stable storage or does it wait for the RM to instruct it?
  - **Steal** / **No-Steal** decision
  - No-steal means RM fixes pages in buffer



- Does the RM force the BM to write certain buffer pages to stable database at the end of a transaction’s execution?
  - **Force** / **No-Force** decision



# Possible Execution Strategies

- **Steal / No-force**

BM may have written some of the updated pages into disk. RM writes a commit

Assumed in the following

- Steal / force

BM may have written some of the updated pages into disk. RM issues a *flush* and writes a commit

- No-steal / no-force

None of the updated pages have been written. RM writes a commit and sends unpin to BM for all pinned pages.

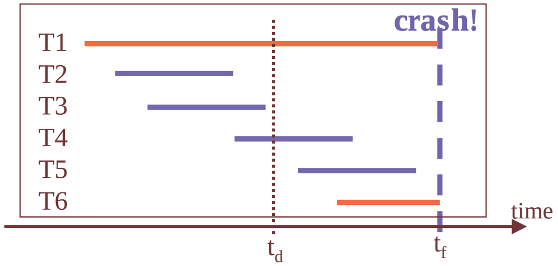
- No-steal / force

None of the updated pages have been written. RM issues a *flush* and writes a commit

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted transaction?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

# General Idea with Buffering

$t_f$ is the time of failure. $t_d$ is the time at which the cache was *certainly* reflected on disk. ➔ Anything after $t_d$ may still be in buffer pages in main memory.



- T2 and T3 made it to secondary storage before $T_d$ and the failure
- T1 and T6 are not committed at time of crash
  - **Undo** T1 and T6 at restart.
- T4 and T5 committed before the crash, but some of their changes may have been only to the volatile database buffer and may not be reflected on disc; some of the changes of T4 may already be reflected on disc
  - **Undo** operations of T4 that are reflected on disc already
  - **Redo** T4, and T5

# Atomicity and Durability with Buffering

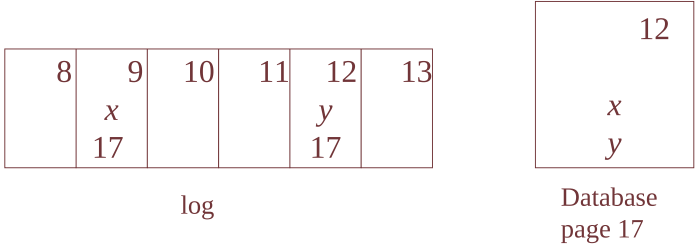
- **Problem**: page and log buffers are volatile
  - Their use affects the time data becomes non-volatile
  - Complicates algorithms for atomicity and durability
- **Requirements**:
  - Write-ahead feature (move update records to log before database is updated) is necessary to preserve atomicity
  - New values written by a transaction must be on mass store when its commit record is written to log (move new values to mass store before commit record) to preserve durability
- **Solution**: requires new mechanisms

# New Mechanism 1

- Forced vs. Unforced Writes:
  - *On database page –*
    - Unforced write updates cache copy, marks it as dirty and returns control immediately.
    - Forced write updates cache copy, marks it as dirty, uses it to update database page on disk, and returns control when I/O completes.
  - *On log –*
    - Unforced append adds record to log buffer and returns control immediately.
    - Forced append, adds record to log buffer, writes buffer to log, and returns control when I/O completes

# New Mechanism 2

- Log Sequence Number (LSN):
  - Log records are numbered sequentially
  - Each database page contains the LSN of the update record describing the most recent update of any item in the page



# Preserving Atomicity: the Write-Ahead Property and Buffering

- **Problem**: When the cache page replacement algorithm decides to write a dirty page, *p*, to mass store, an update record corresponding to *p* might still be in the log buffer.
- **Solution**: Force the log buffer if the LSN stored in *p* is greater than or equal to the LSN of the oldest record in the log buffer. Then write *p*. This preserves write-ahead policy.

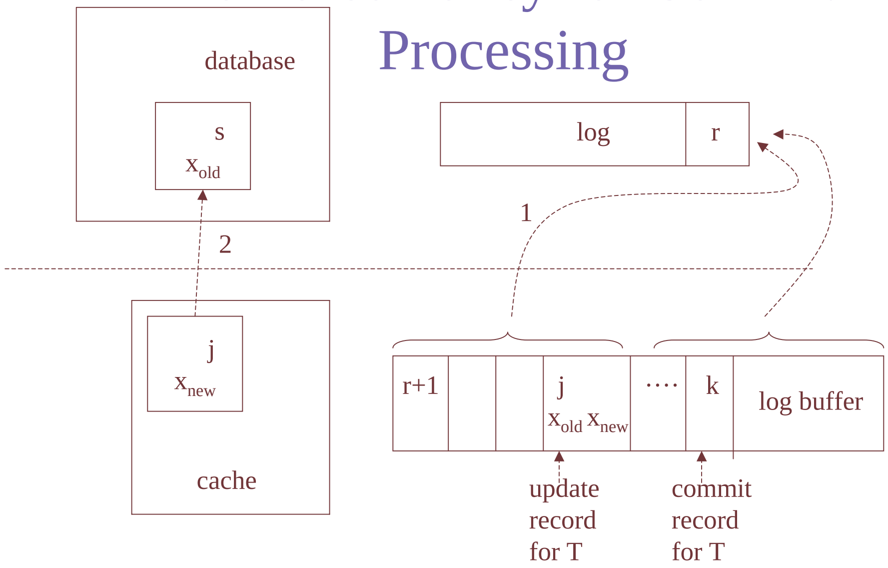
# Preserving Durability

- **Problem**: Pages updated by T might still be in cache when T's commit record is appended to log buffer.
- **Solution**: Update record contains *after image* (called a *redo* record) as well as before image
  - Write-ahead property still requires that update record be written to mass store before page
  - But it is not necessary to force dirty pages when commit record is written to log on mass store (*no-force* policy) since all after images precede commit record in log

# No Force Commit Processing

- No force policy for commit processing:
  - (1) Force the log buffer (immediate commit)
    - Log contains both T's update records and its commit record
    - update records precede commit record in log buffer, ensuring transaction's updates are durable before (or at the same time as) commit
  - (2) T's dirty pages can be flushed from cache at any time after update records have been written
    - Necessary for write-ahead policy
    - Dirty pages can be written before or after commit record

# No Force Policy for Commit Processing



---

# No-Force Policy

- **Advantages**:
  - Commit does not have to wait while dirty pages are forced
  - Pages with hotspots do not have to be written out as frequently
- **Disadvantage**:
  - Crash recovery complicated: some updates of committed transactions (contained in redo records) might not be in database on restart after crash
  - Update records are larger

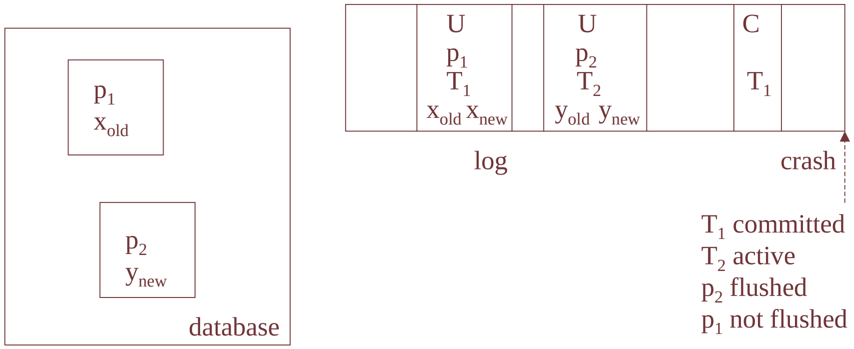
---

# Recovery With No-Force/Steal Policy

- **Problem**: When a crash occurs there might exist
  - Some pages in database containing updates of uncommitted transaction: they must be rolled back
  - Some pages in database that do not (but should) contain the updates of committed transactions: they must be rolled forward
- **Solution**: Use a *sharp checkpoint* (all dirty pages are forced to disk at checkpoint)

---

# Recovery With No-Force/Steal Policy



$T_1$ committed
$T_2$ active
$p_2$ flushed
$p_1$ not flushed

$p_1$ must be rolled forward using $x_{new}$
$p_2$ must be rolled back using $y_{old}$

# Algorithm ARIES

- A recovery algorithm that works with the steal/no-force strategy (called ARIES) has *3 Passes*:
  - *PASS 1 – Analysis*: Scan the log backward to the most recent checkpoint to identify all transactions that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *PASS 2 – Redo*: The log is scanned forward (replayed) from the checkpoint to ensure that all logged updates are in fact carried out and written to disk.
  - *PASS 3 – Undo*: The writes of all transactions that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)