Transactions – Recovery

CS 317/387

1

ACID Properties and Recovery

- 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.

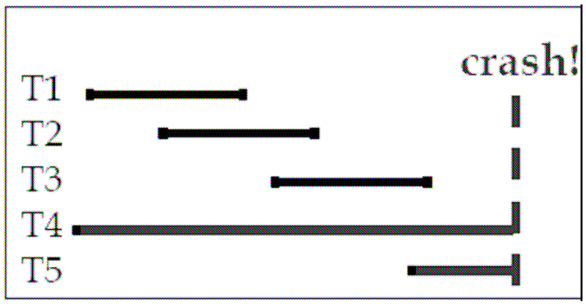
2

Types of Failures

- ***System crashes***
  - due to hardware or software errors
  - main memory content is lost
- ***Transaction failures***
  - overflow, interrupt, data not available, explicit rollback, concurrency enforcement, programming errors
  - no memory loss.
- *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

Strategy

- If a transaction *Ti* 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 & T3should 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?

- Each modification of the database causes an *update record* to be appended to 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 file example

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

Transaction Abort using Logs

- Scan log backwards using tid to identify transaction's update records
- Reverse each update using before image
- In a strict system new values 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

9

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

B – Begin record
U – update record

Scan back to Begin record to abort a transaction

10

Crash Recovery using Logs

- 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:

1. Scan log backwards
2. If first of T's records is update record, T was active at time of crash. Roll it back
3. Transaction not committed until commit record in 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

B – Begin Record
U – Update Record
C – Commit
A - Abort

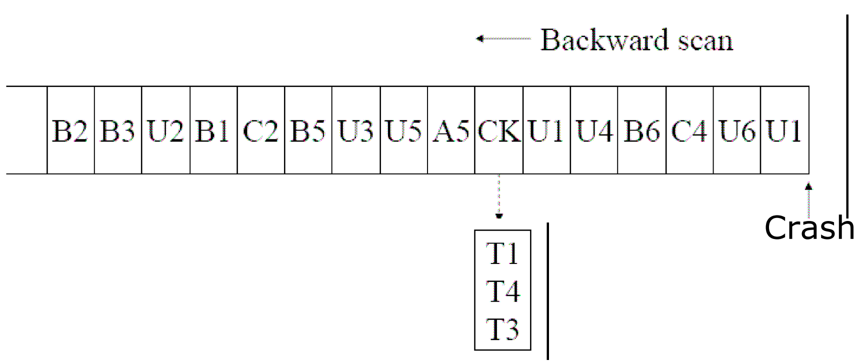
T1and T3 were not active at time of crash

Crash Recovery

- **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

13

Checkpoints



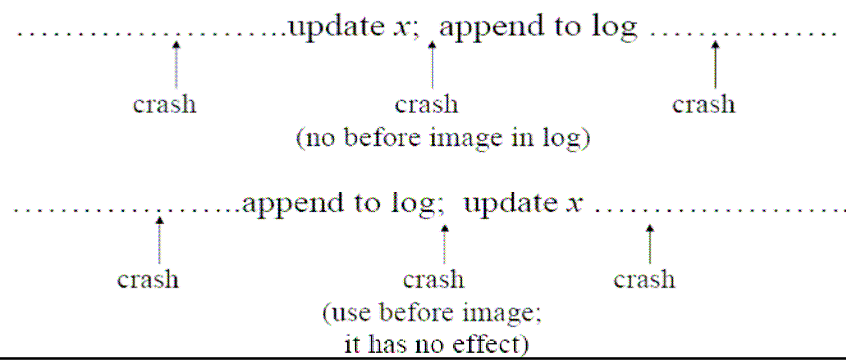
T1, T3 and T6 Active at time of crash

14

Write Ahead Logging

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
- Force the log record for an update *before* the corresponding data page gets to disk.
  - Guarantees Atomicity

.

16

Performance Issues

- **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

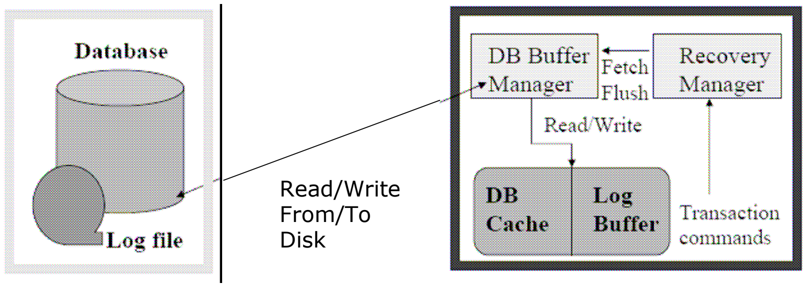
17

Performance Concerns

- **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)

18

Architecture



Persistent storage,

•loses content only if media fails

or is otherwise lost

•Contains DB and Transaction Log

•Disks and other Media

•***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

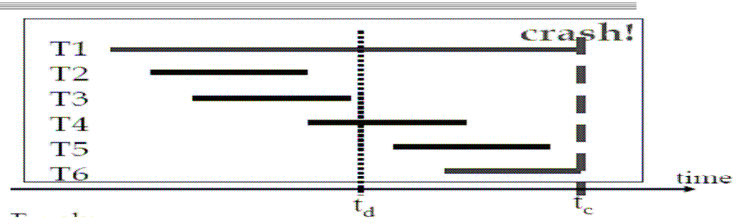
- Database pages are read from disk, if needed, and put into the cache in main memory. They stay there until explicitly written back to disk.
- Read and Write operations of transactions are executed on pages in the cache! Cache pages that have been updated are marked *dirty*; others are *clean*.
- Changed 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

What about Atomicity and Durability?

- **Problem**: page and log buffers are volatile
  - Their use affects the time data becomes non-volatile
  - Complicates algorithms for atomicity and durability

21

Illustrated



Assume T_d is the last time point before a crash at T_c where all changes to the DB were definitely reflected on disk.

⇒T2 and T3 made it to disk before T_d

⇒T1 and T6 are not committed at T_c and have to be undone

⇒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 disk; some of the changes of T4 may already be reflected on disk

22

Atomicity and Durability with Buffering

- **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

23

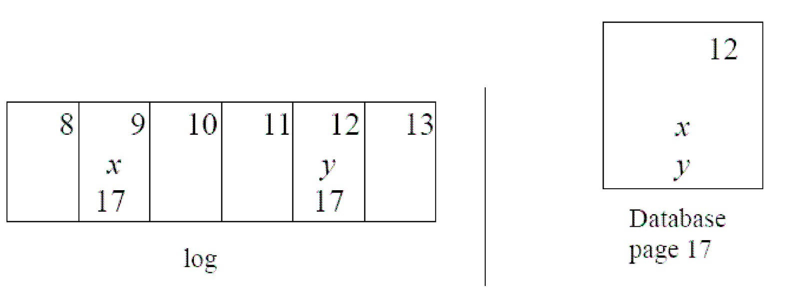
One solution

- 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

24

Solution 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.

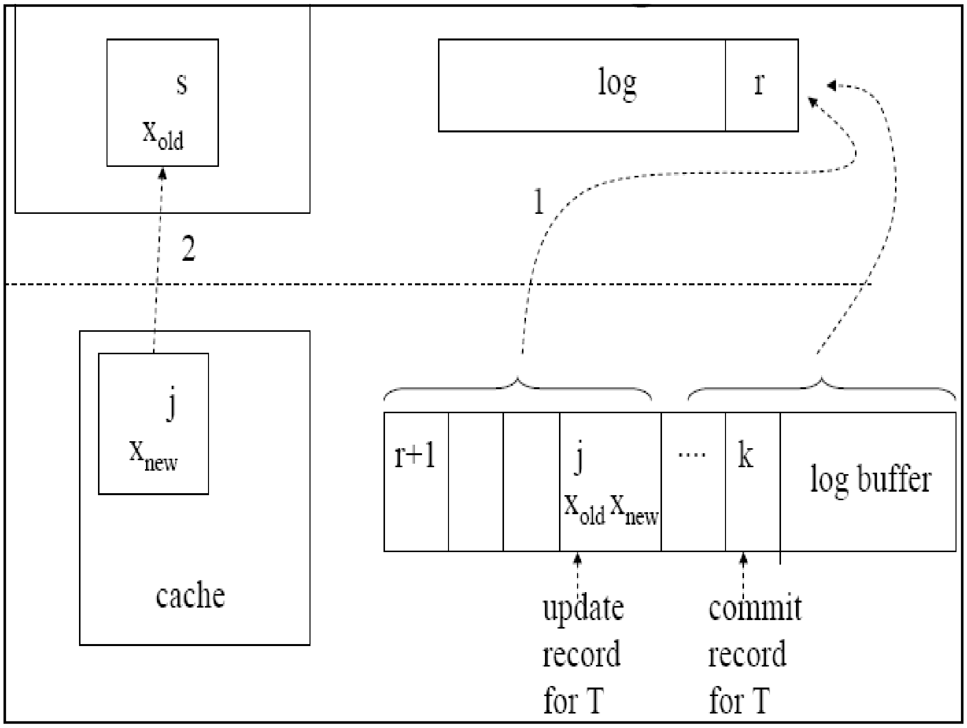
- **Solution**: Update record contains *after image*(called a *redo* record) as well as before image
  - Write-ahead property still requires that update record be written to mass store before page
  - But it is not necessary to force dirty pages when commit record is written to log on mass store (*no-force*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

27

No Force 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 database pages can be written before or after commit record

28



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

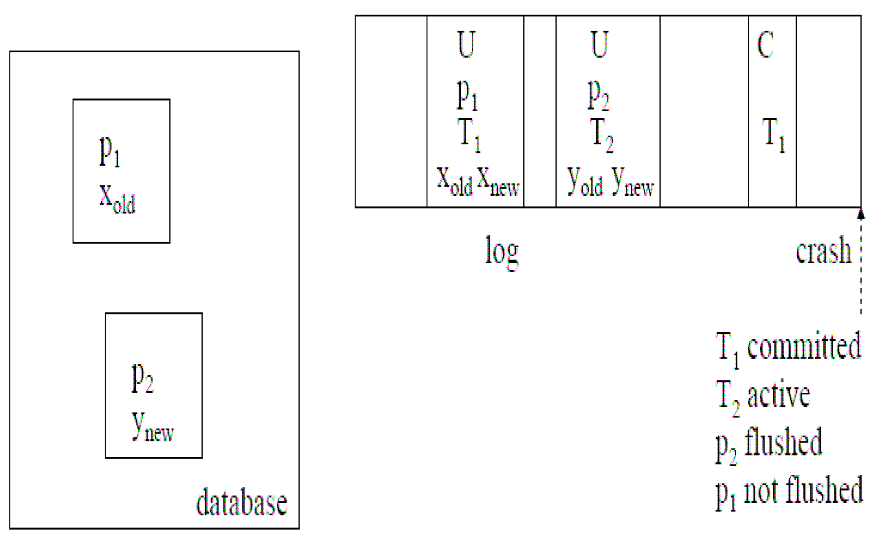
- **Disadvantage**:
  - Crash recovery complicated: some updates of committed transactions (contained in redo records) might not be in database on restart after crash
  - Update records are larger

30

Recovery with a No Force 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)

31



p_1 must be rolled forward using x_{new}

p_2 must be rolled back using y_{old}

ARIES

- A recovery algorithm that works with the steal/no-force strategy (called ARIES) has *3 Passes*:

1. ***Analysis***: Scan the log backward to the most recent sharp checkpoint to identify all transactions that were active, and all dirty pages in the buffer pool at the time of the crash.
2. ***Redo***: The log is scanned forward (replayed) from the checkpoint to ensure that logged updates are in fact carried out and written to disk.
3. ***Undo***: The writes of all transactions that were active at the crash are undone (by restoring the *before image* of the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

33