Database Management Systems

Lecture 4

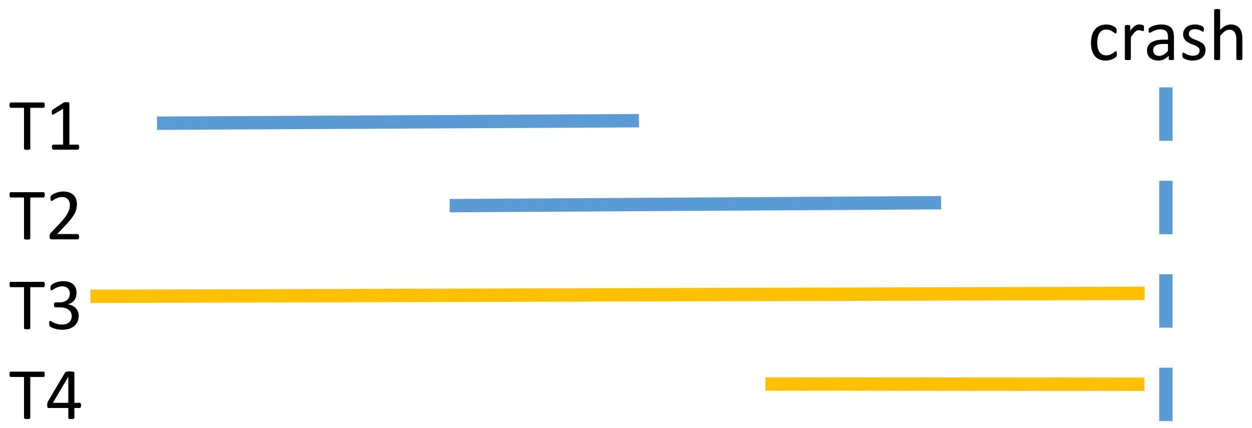
Crash Recovery

Sabina S. CS

Recovery Manager

- the Recovery Manager in a DBMS ensures two important properties of transactions:
  - atomicity
    - the effects of uncommitted transactions are undone
  - durability
    - the effects of committed transactions survive system crashes

Recovery Manager



- T1, T2 commit before the crash
- T3 and T4 are still active when the system crashes

- the system comes back up:
  - the effects of T1 & T2 must persist
  - T3 & T4 are undone (their effects are not persisted in the DB)

Transaction Failure - Causes

- system failure
- application error
- action by the Transaction Manager
- self-abort

- system failure (hardware failures, bugs in the operating system, database system, etc)
  - all running transactions terminate
  - contents of internal memory – affected (i.e., lost)
  - contents of external memory – not affected

Transaction Failure - Causes

- system failure
- application error
- action by the Transaction Manager
- self-abort

- application error (“bug”, e.g., division by 0, infinite loop, etc)
  => transaction fails; it should be executed again only after the error is corrected

Transaction Failure - Causes

- system failure
- application error
- action by the Transaction Manager
- self-abort

- action by the Transaction Manager (TM)
  - e.g., deadlock resolution scheme
  - a transaction is chosen as the deadlock victim and terminated
  - the transaction might complete successfully if executed again

Transaction Failure - Causes

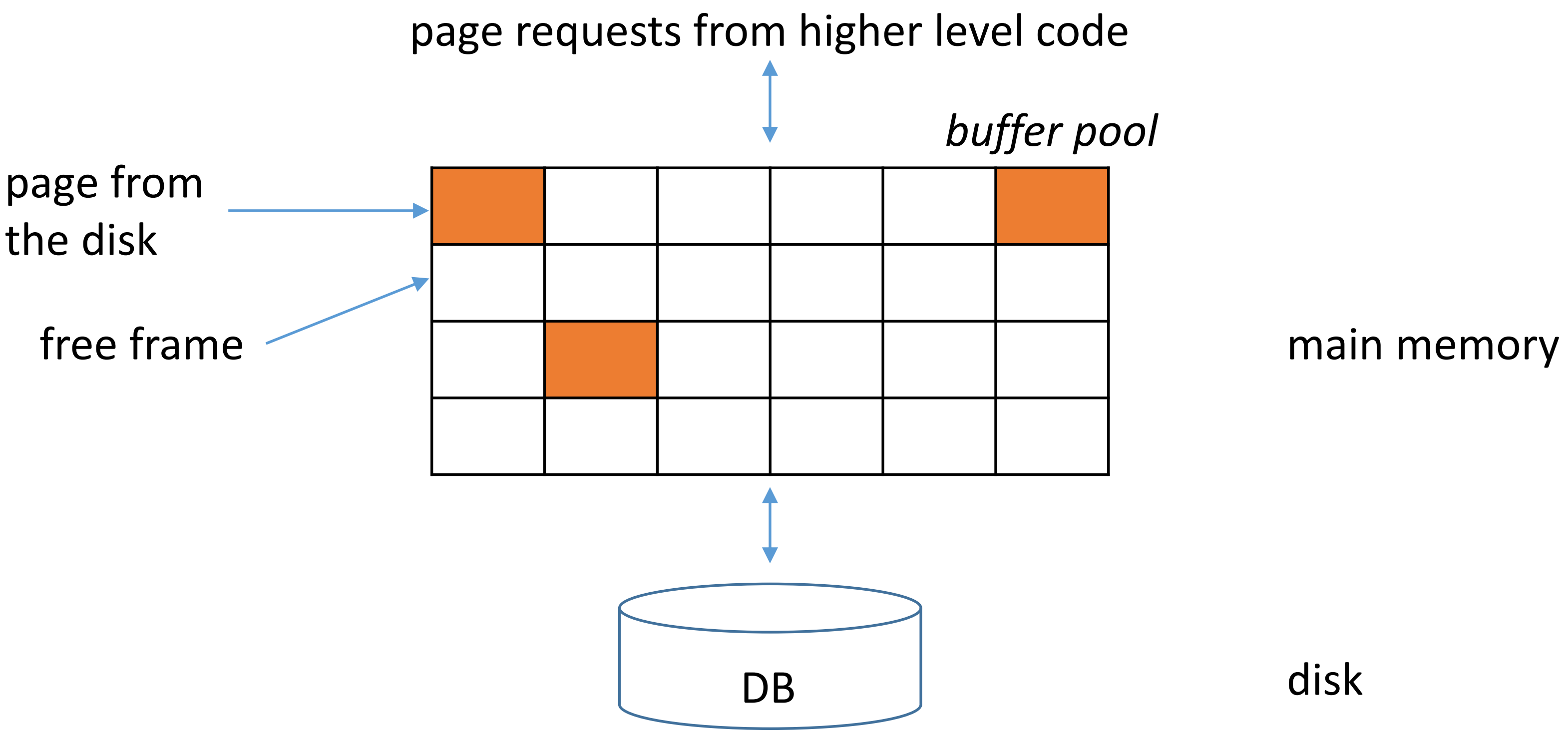
- system failure
- application error
- action by the Transaction Manager
- self-abort

- self-abort
  - based on some computations, a transaction can decide to terminate and undo its actions
  - there are special statements for this purpose, e.g., *ABORT, ROLLBACK*
  - can be seen as a special case of *action by the TM*

Normal Execution

- during normal execution, transactions read / write database objects
- reading database object O:
    - bring O from disk into a frame in the Buffer Pool (BP)
    - copy O's value into a program variable
- writing database object O:
    - modify an in-memory copy of O (in the BP)
    - write the in-memory copy to disk

Buffer Manager*



*see the *Databases* course in the 1st semester (lecture 8 - Buffer Manager)

Buffer Manager

- higher level layer L in the DBMS asks the Buffer Manager (BM) for page P
- if P is not in the BP, the BM brings it into a frame F in the BP
- when P is no longer needed, L notifies the BM (it releases P), so F can be reused
- if P has been modified, L notifies the BM, which propagates the changes in F to the disk

Buffer Manager

- BM maintains 2 variables for each frame F
    - *pin_count*
        - number of current users (requested the page in F but haven't released it yet)
        - only frames with pin_count = 0 can be chosen as replacement frames
    - *dirty*
        - boolean value indicating whether the page in F has been changed since being brought into F
- incrementing pin_count
    - pinning a page P in a frame F
- decrementing pin_count
    - unpinning a page

Buffer Manager

- initially, pin_count = 0, dirty = off, ∀ F ∈ BP
- L asks for a page P; the BM:

1. checks whether page P is in the BP; if so, pin_count(F)++, where F is the frame containing P

otherwise:

a. BM chooses a frame FR for replacement

- if the BP contains multiple frames with pin_count = 0, one frame is chosen according to the BM's replacement policy
- pin_count(FR)++;

b. if dirty(FR) = on, BM writes the page in FR to disk

c. BM reads page P in frame FR

2. the BM returns the address of the BP frame that contains P to L

Buffer Manager

- replacement policies
  - Least Recently Used
  - Most Recently Used
  - random
  - ...

Writing Objects

- options: *steal / no-steal, force / no-force*

- transaction T changes object O (in frame F in the BP)

- *steal* approach:
  - T's changes can be written to disk before it commits
  - transaction T2 needs a page; the BM chooses F as a replacement frame (while T is in progress); T2 steals a frame from T

- *no-steal* approach:
  - T's changes cannot be written to disk before it commits

Writing Objects

- options: *steal / no-steal, force / no-force*

- transaction T changes object O

- *force* approach
  - T's changes are immediately forced to disk when it commits

- *no-force* approach
  - T's changes are not forced to disk when it commits

Writing Objects

- *no-steal* approach
  - advantage - changes of aborted transactions don't have to be undone (such changes are never written to disk!)
  - drawback - assumption: all pages modified by active transactions can fit in the BP
- *force* approach
  - advantage - actions of committed transactions don't have to be redone
    - by contrast, when using *no-force*, the following scenario is possible: transaction T commits at time t_0; its changes are not immediately forced to disk; the system crashes at time t_1 => T's changes have to be redone!
  - drawback - can result in excessive I/O

- *steal, no-force* approach – used by most systems

ARIES

- recovery algorithm; *steal, no-force* approach
- system restart after a crash - three phases:
    - analysis
        - determine:
            - active transactions at the time of the crash
            - *dirty pages*, i.e., pages in BP whose changes have not been written to disk
    - redo
        - reapply all changes (starting from a certain record in the log), i.e., bring the DB to the state it was in when the crash occurred
    - undo
        - undo changes of uncommitted transactions

ARIES

- fundamental principle - *Write-Ahead Logging*
  - a change to an object O is first recorded in the log (in a log record LR)
  - LR must be written to stable storage before the change to O is written to disk

ARIES

* example
- analysis
  - active transactions at crash time: T1, T3 (to be undone)
  - committed transactions: T2 (its effects must persist)
  - potentially dirty pages: P1, P2, P3
- redo
  - reapply all changes in order (1, 2, …)
- undo
  - undo changes of T1 and T3 in reverse order (6, 5, 1)

LSN	Log
1	update: T1 writes P1
2	update: T2 writes P2
3	T2 commit
4	T2 end
5	update: T3 writes P3
6	update: T3 writes P2
crash, restart	

Storage Media

- volatile storage
  - information doesn't usually survive system crashes (e.g., main memory)
- non-volatile storage
  - information survives system crashes (e.g., magnetic disks, flash storage)
- stable storage
  - information is never lost
  - techniques that approximate stable storage (e.g., store information on multiple disks, in several locations)

The Log (journal)

- history of actions executed by the DBMS
- file of records
- stored in *stable storage* (keep >= 2 copies of the log on different disks (locations) - ensures the durability of the log)
- records are added to the end of the log
- *log tail*
  - the most recent fragment of the log
  - kept in main memory and periodically forced to stable storage
- *Log Sequence Number* (LSN)
  - unique id for every log record
  - monotonically increasing (e.g., address of 1^{st} byte of log record)

The Log

- *pageLSN*
    - every page P in the DB contains the *pageLSN*: the LSN of the most recent record in the log describing a change to P
- *log record* – fields:
    - prevLSN
        - linking a transaction’s log records
    - transID
        - id of the corresponding transaction
    - type
        - type of the log record

The Log

- a log record is written for each of the following actions:
  - *update page*
  - *commit*
  - *abort*
  - *end*
  - *undo an update*

The Log

- update page P
  - add an *update type* log record ULR to the log tail (with LSN_{ULR})
  - pageLSN(P) is set to LSN_{ULR}
- transaction T commits*
  - add a *commit type* log record CoLR to the log
  - force log tail to stable storage (including CoLR)
  - complete subsequent actions (remove T from transaction table)
- transaction T aborts
  - add an *abort type* log record to the log
  - initiate Undo for T

* obs. committed transaction – a transaction whose log records (including the *commit log record*) have been written to stable storage

The Log

- transaction T ends
  - T commits / aborts - complete required actions
  - add an *end type* log record to the log
- undo an update
  - i.e., when the change described in an update log record is undone
  - write a *compensation log record* (CLR)

The Log

- *update log record*
    - additional fields
        - *pageID* (id of the changed page)
        - *length* (length of the change - in bytes)
        - *offset* (offset of the change)
        - *before-image* (value before the change)
        - *after-image* (value after the change)
    - can be used to undo / redo the change

The Log

- *compensation log record*
  - let U be an update log record describing an update of transaction T
  - let C be the compensation log record for U, i.e., C describes the action taken to undo the changes described by U
  - C has a field named *undoNextLSN*:
    - the LSN of the next log record to be undone for T
    - set to the value of prevLSN in U

->

The Log

- *compensation log record*

* example: undo T10's update to P10

=> CLR with:

transID = T10

pageID = P10

length = 2

offset = 10

before-image = JH

undoNextLSN = LSN of 1st log record (i.e., the next record that is to be undone for transaction T10)

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

log

The Transaction Table and the Dirty Page Table

- contain important information for the recovery process
- *transaction table*
  - 1 entry / active transaction
  - fields
    - *transID*
    - *status*
      - in progress
      - committed
      - aborted
    - *lastLSN*
      - LSN of the most recent log record for the transaction

->

The Transaction Table and the Dirty Page Table

- *transaction table*
  - example (*status* = in progress, not displayed)

transID	lastLSN
T10	
T15	

transaction table

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

log

The Transaction Table and the Dirty Page Table

- *dirty page table*
  - 1 entry / dirty page in the Buffer Pool
  - fields
    - *pageID*
    - *recLSN*
      - the LSN of the 1st log record that dirtied the page

->

The Transaction Table and the Dirty Page Table

- *dirty page table*
  - example

pageID	recLSN
P100	
P2	
P10	

dirty page table

prevLSN	transID	type	pageID	length	offset	before-image	after-image
	T10	update	P100	2	10	AB	CD
	T15	update	P2	2	10	YW	ZA
	T15	update	P100	2	9	EC	YW
	T10	update	P10	2	10	JH	AB

log

transID	lastLSN
T10	
T15	

transaction table

Checkpointing

- objective
  - reduce the amount of work performed by the system when it comes back up after a crash
- *checkpoints* taken periodically

- checkpointing in ARIES - 3 steps:
  - write a *begin_checkpoint* record (it indicates when the checkpoint starts)
    - LSN_{BCK} - LSN of *begin_checkpoint* record
  - write an *end_checkpoint* record
    - it includes the current Transaction Table and the current Dirty Page Table

Checkpointing

- checkpointing in ARIES - 3 steps:
  - after the *end_checkpoint* record is written to stable storage:
    - write a *master* record to a known place on stable storage
    - *master* record includes LSN_{BCK}

- crash -> restart -> system looks for the most recent checkpoint
- normal execution begins with a checkpoint with an empty Transaction Table and an empty Dirty Page Table

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019