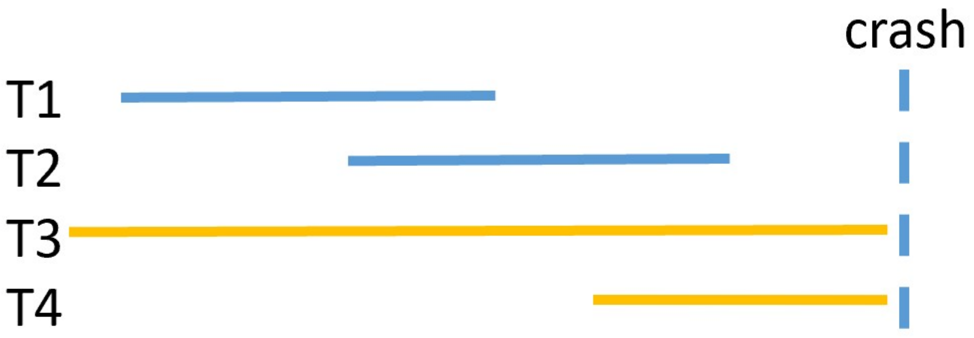
Database Management Systems

Lecture 4

Crash Recovery

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



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

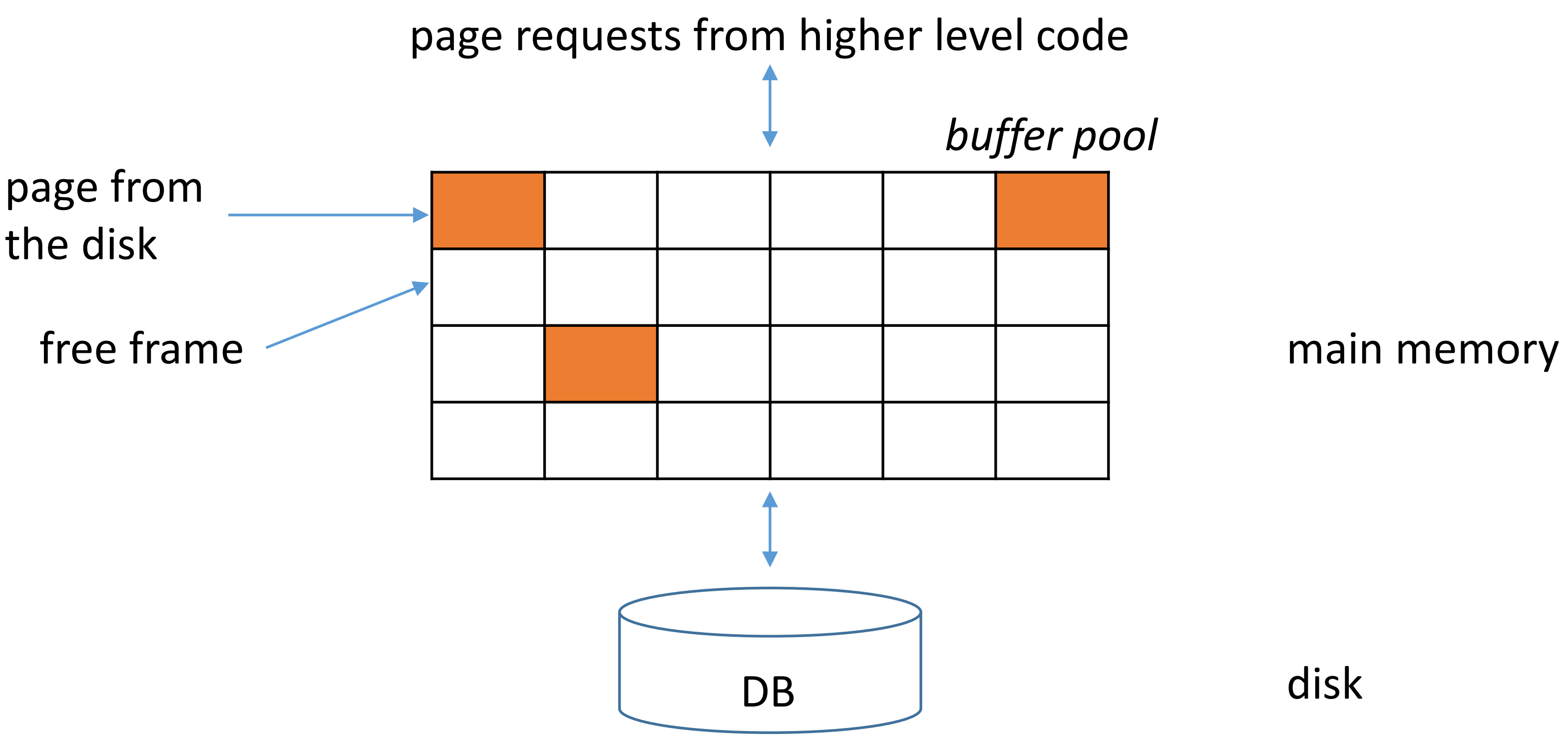
Transaction Failure - Causes

- 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)

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
  - *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

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)

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
- 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	

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 1st 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
- 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)

- *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
  - 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
  - example:

pageID	recLSN
P100	
P2	
P10	

dirty page table

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

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
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3rd Edition, 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
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, http://codex.cs.yale.edu/avi/db-book/