Recovery

Context

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability ?
    - Atomicity: Two problems
      - Part of the transaction is done, but we want to cancel it
        - ABORT/ROLLBACK
      - System crashes during the transaction. Some changes made it to the disk, some didn't.
    - Durability:
      - Transaction finished. User notified. But changes not sent to disk yet (for performance reasons). System crashed.
- Essentially similar solutions

Reasons for crashes

- Transaction failures
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash
  - Power failures, operating system bugs etc
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure
  - Head crashes; *for now we will assume*
    - *STABLE STORAGE: Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data*

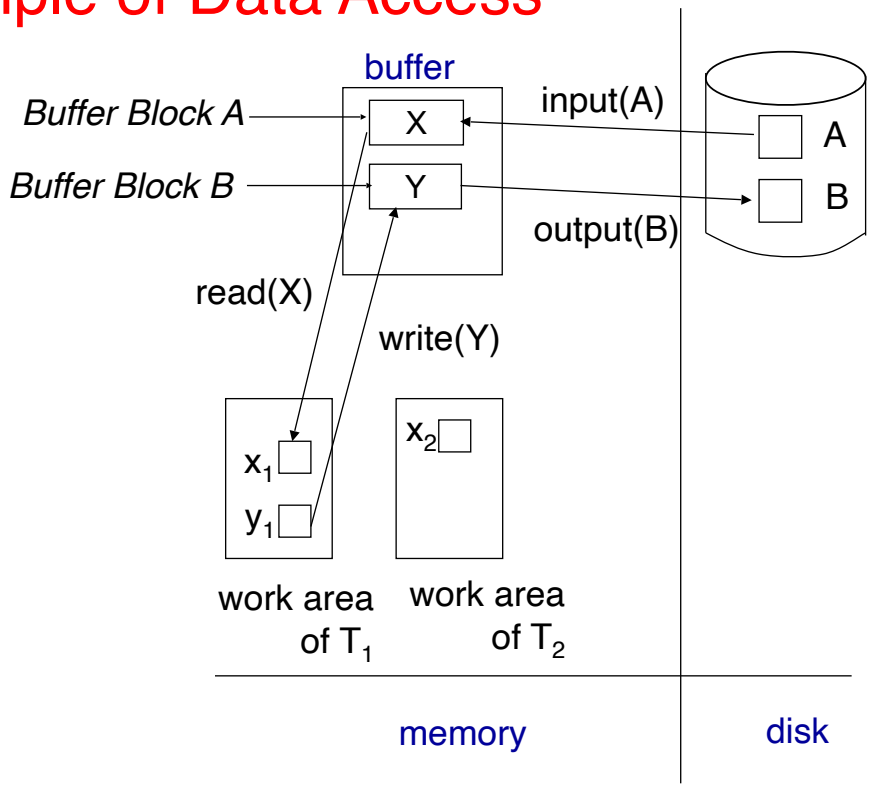
Approach, Assumptions etc..

- Approach:
  - Guarantee A and D:
    - by controlling how the disk and memory interact,
    - by storing enough information during normal processing to recover from failures
    - by developing algorithms to recover the database state
- Assumptions:
  - System may crash, but the *disk is durable*
  - The only *atomicity* guarantee is that *a disk block write* is *atomic*
- Naive solutions work, but are too expensive.
  - E.g. The shadow copy solution we saw earlier
    - Make a copy of the database; do the changes on the copy; do an atomic switch of the *dbpointer* at commit time
  - Goal is to do this as efficiently as possible

Data Access

- Physical blocks are those residing on the disk.
- Buffer blocks are those temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - input(B) transfers the physical block B to main memory.
  - output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - read(X) assigns the value of data item X to the local variable x_i.
  - write(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - Note: output(B_X) need not immediately follow write(X). System can perform the output operation when it deems fit.
- Transactions
  - Must perform read(X) before accessing X for the first time (subsequent reads can be from local copy)
  - write(X) can be executed at any time before the transaction commits*

STEAL vs NO STEAL, FORCE vs NO FORCE

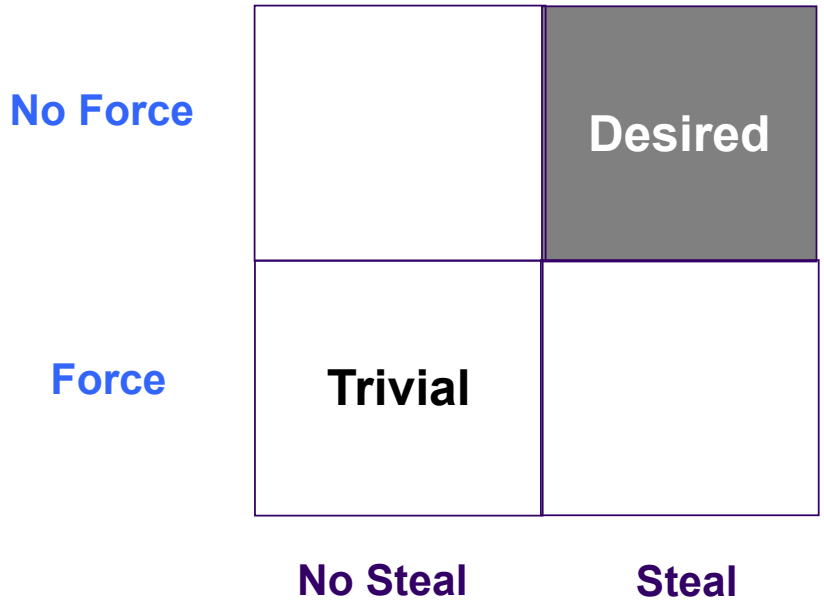
- STEAL:
  - The buffer manager *can steal* a (memory) page from the database
    - ie., it can write an arbitrary page to the disk and use that page for something else from the disk
    - In other words, the database system doesn't control the buffer replacement policy
  - Why a problem ?
    - The page might contain *dirty writes,* ie., writes/updates by a transaction that hasn't committed
  - But, we must allow *steal* for performance reasons.

- NO STEAL:
  - Not allowed. More control, but less flexibility for the buffer manager.

STEAL vs NO STEAL, FORCE vs NO FORCE

- FORCE:
  - The database system *forces* all the updates of a transaction to disk before committing
  - Why ?
    - To make its updates permanent before committing
  - Why a problem ?
    - Most probably random I/Os, so poor response time and throughput
    - Interferes with the disk controlling policies
- NO FORCE:
  - Don't do the above. Desired.
  - Problem:
    - Guaranteeing durability becomes hard
  - Might still have to *force* pages to disk, but hopefully minimal.

STEAL vs NO STEAL, FORCE vs NO FORCE



What if NO STEAL, FORCE ?

start trans
write(A)
write(B)
start commit
push A
pu…
……crash!

- Only updates from committed transaction are written to d
  (since no steal)
- Updates from a transaction are forced to disk before com
  (since force)
  - A minor problem: how do you guarantee that all updates from a transaction make it to the disk atomically ?
    - Remember we are only guaranteed an atomic *block write*
    - What if some updates make it to disk, and other don't ?
  - Can use something like shadow copying/shadow paging
- No durability problems.
- Slow

What if STEAL, NO FORCE ?

- After crash:
  - Disk might have DB data from uncommitted transactions
  - Disk might not have DB data from committed transactions
- How to recover?

"Log-based recovery"

Log-based Recovery

- Most commonly used recovery method
- A log is a record of everything the database system does
  - the "DB" are the files where relations are stored

- For every operation done by the database, a *log record* is generated and stored *typically on a different disk*
  - *<T1, START>*
  - *<T2, COMMIT>*
  - *<T3, ABORT>*
  - *<T1, A, 100, 200>*
    - T1 modified A; old value = 100, new value = 200

Log

- Example transactions T_0 and T_1 (T_0 serialized before T_1):

T_0: read (A)
$A = A - 50$
write (A)
read (B)
$B = B + 50$
write (B)

T_1: read (C)
$C = C - 100$
write (C)

- Possible logs after crash and restart:

(a)	(b)	(c)
$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 950, 900>$	$<T_0, A, 950, 900>$	$<T_0, A, 950, 900>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$, commit$>$	$<T_0$, commit$>$
	$<T_1$, start$>$	$<T_1$, start$>$
	$<T_1, C, 500, 400>$	$<T_1, C, 500, 400>$
		$<T_1$, commit$>$

Log-based Recovery

- *Starting* assumptions:
  1. Log records are *immediately pushed to the disk* as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated *before* the actual data value is updated
  4. *Strict two-phase locking*
     - The first assumption can be relaxed
     - *A transaction T1 is considered committed only after record <T1, COMMIT> has been pushed to the disk*
- Also:
  - Log writes are *sequential*
  - They are also often on a different disk (why important?)
  - File systems:
    - LFS == log-structured file system
    - *journaling* file systems

Recovery

STEAL is allowed, so changes of a transaction may have made it to the disk

- UNDO(T1):
  - Procedure executed to *rollback/undo* the effects of a transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - T1 decides to abort
  - Any of the changes might have made it to the disk

Using the log to *abort/rollback*

<T1, START>
<T1, A, 200, 300>
<T1, B, 400, 300>
<T1, A, 300, 200>

- UNDO(T1):
  - Go *backwards* in the *log* looking for log records belonging to T1
  - Restore the values to the old values
  - NOTE: Going backwards is important.
    - *A* was updated twice
  - In the example, we simply:
    - Restore A to 300
      - *Write <T1, CLR, A, 300> record* (compensating log record)
    - Restore B to 400
      - *Write <T1, CLR, B, 400> record*
    - Restore A to 200
      - *Write <T1, CLR, A, 200> record*
    - *Write <T1, ABORT>* **(abort comes *after* CLR records)**
  - Note: No other transaction could have changed A or B in the meantime
    - *Strict two-phase locking*

Using the log to *recover*

- We don't require FORCE, so a change made by the committed transaction may not have made it to the disk before the system crashed
  - BUT, the log record did (recall our assumptions)
- REDO(T1):
  - Procedure executed to recover a committed transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - *<T1, COMMIT>*
  - By our assumptions, all the log records made it to the disk (since the transaction committed)
  - But any or none of the changes to A or B might have made it to disk

Using the log to *recover*

<T1, START>
<T1, A, 200, 300>
<T1, B, 400, 300>
<T1, A, 300, 200>

- REDO(T1):
  - Go *forward* in the *log* looking for log records belonging to T1
  - Set the values to the new values
  - NOTE: Going forward is important.
  - In the example, we simply:
    - Set A to 300
    - Set B to 300
    - Set A to 200

Idempotency

- Both redo and undo are required to be *idempotent*
  - *F is idempotent, if F(x) = F(F(x)) = F(F(F(F(…F(x)))))*
- Multiple applications shouldn't change the effect
  - Important as *we don't know* what made it to the disk
  - E.g., consider a log record of the type
    - <T1, A, *incremented by 100*>
    - Old value was 200, and so new value was 300
  - But the on disk value might be 200 or 300 (since we have no control over the buffer manager)
  - So we have no idea whether to apply this log record or not
  - Hence, we use *value based logging* (*physical logging*), not operation based (*logical logging*)

Log-based recovery

- Log is maintained
- If during the normal processing, a transaction needs to abort
  - UNDO() is used for that purpose
- If the system crashes, then we need to do recovery using both UNDO() and REDO()
  - Some transactions that were going on at the time of crash may not have completed, and must be *aborted/undone*
  - Some transactions may have committed, but their changes didn't make it to disk, so they must be *redone*
  - Called *restart recovery*

Restart Recovery (after a crash)

1. Initialize the undo-list to empty list.
2. Roll forward through the log re-executing everything
   a. Add transaction STARTs to the undo-list as you go
   b. Remove transactions from the undo-list if a corresponding commit record is found
3. Roll back from end of log undo-ing the effects of transactions in your undo-list

Checkpointing

- How far should we go back in the log while recovering ??
  - It is possible that a transaction made an update at the very beginning of the system, and that update never made it to disk
    - very very unlikely, but possible (because we don't do force)
  - For correctness, we have to go back all the way to the beginning of the log
  - Bad idea !!

- Checkpointing is a mechanism to reduce this

Checkpointing

- Periodically, the database system writes out everything in the memory to disk
  - Goal is to get the database in a state that we know (not necessarily consistent state)
- Steps:
  - Stop all other activity in the database system
  - Write out the entire contents of the memory to the disk
    - Only need to write updated pages, so not so bad
    - Entirely write all updates, whether committed or not
  - Write out all the log records to the disk
  - Write out a special log record to disk
    - *<CHECKPOINT LIST_OF_ACTIVE_TRANSACTIONS>*
    - The second component is the list of all active transactions in the system right now
  - Continue with the transactions again

Restart Recovery w/ checkpoints

- implement the *redo* phase of Section 19.4
  - Roll forward from the last checkpoint or the beginning of the log: keep track of active transactions, taking into account any information from the checkpoint
  - redo any UPDATE and CLR records encountered
- implement the *undo* phase of Section 19.4
  - roll back from the end of the log: reversing the effects of any encountered UPDATE records of active transactions by
    - changing the data in the relation back to the original, and
    - append ing CLR records
  - add abort records when encountering the START record for any active transaction
- finish
  - push all changes to the relation file (using BufferPool.writeAllToDisk)
  - write a checkpoint record to the log at the end.

(assignment 8 or 9)

Write-ahead logging

- So far assumed that log records are written to disk as soon as generated
  - Too restrictive
- Write-ahead logging:
  - Before an update on a data item (say A) makes it to disk, the log records referring to the update must be forced to disk
  - How ?
    - Each log record has a log sequence number (LSN)
      - Monotonically increasing
    - For each page in the memory, we maintain the LSN of the *last log record* that updated a record on this page
      - *pageLSN*
    - If a page *P* is to be written to disk, all the log records till *pageLSN(P)* are forced to disk first

Write-ahead logging

- Write-ahead logging (WAL) is sufficient for all our purposes
  - All the algorithms discussed before work
- Note the special case:
  - A transaction is not considered committed unless the **<T, commit>** record is on disk

Other issues

- ARIES: Considered *the canonical description of log-based recovery*
  - Used in most systems
  - Has many other types of log records that simplify recovery significantly
- Loss of disk:
  - Can use a scheme similar to checkpointing to periodically dump the database onto *tapes* or *optical storage*
  - Techniques exist for doing this while the transactions are executing (called *fuzzy dumps)*
- Shadow paging:
  - Read up

Other issues

- The system halts during checkpointing
  - Not acceptable
  - Advanced recovery techniques allow the system to continue processing while checkpointing is going on
- System may crash during recovery
  - Our simple protocol is actually fine
  - In general, this can be painful to handle
- B+-Tree and other indexing techniques
  - Strict 2PL is typically not followed (we didn't cover this)
  - So physical logging is not sufficient; must have logical logging (section 19.7

Recap

- STEAL vs NO STEAL, FORCE vs NO FORCE
  - We studied how to do STEAL and NO FORCE through log-based recovery scheme