Admin

• Signup sheet for project presentations: contact Sharath ASAP – also check forum

• Course evaluation:
  http://www.CourseEvalUM.umd.edu

• Additional queries (deadline – your demo day)
• Find the authors, their skills and education, of the top 5 highest cited publications.

• Find the most accomplished author in your database. (Use any ranking function for "accomplishment": e.g # of Papers + # of awards)
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:

• Essentially similar solutions
Reasons for crashes

• Transaction failures
  – Logical errors, deadlocks
• System crash
  – Power failures, operating system bugs etc
• Disk failure
  – Head crashes; for now we will assume that either this does not happen or that RAID is used to handle this
  – 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

• Once again, obvious naïve solutions exist that work, but that 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
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
  – We might still have to *force* some pages to disk, but minimal.
STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

- No Force
  - No Steal
  - Steal
- Force
  - Desired
  - Trivial
STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

• How to implement A and D when No Steal and Force?
  – Only updates from committed transaction are written to disk (since no steal)
  – Updates from a transaction are forced to disk before commit (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 atomicity/durability problem arise.
Terminology

• Deferred Database Modification (write at commit time):
  – Similar to NO STEAL, NO FORCE
    • Not identical
  – Only need *redos, no undos*
  – We won’t cover this today

• Immediate Database Modification (write anytime):
  – Similar to STEAL, NO FORCE
  – Need both *redos, and undos*
Log-based Recovery

• Most commonly used recovery method
• Intuitively, a log is a record of everything the database system does
• For every operation done by the database, a log record is generated and stored typically on a different (log) disk
• <T1, START>
• <T2, COMMIT>
• <T2, ABORT>
• <T1, A, 100, 200>
  – T1 modified A; old value = 100, new value = 200
Log

- Example transactions $T_0$ and $T_1$ ($T_0$ executes before $T_1$):

  $T_0$:  
  \[ \text{read} (A) \]
  
  \[ A: - A - 50 \]

  \[ \text{write} (A) \]
  
  \[ B: - B + 50 \]

  \[ \text{read} (B) \]

  \[ C: - C - 100 \]

  \[ \text{write} (C) \]

- Log:

  \[
  \begin{array}{ccc}
    <T_0 \text{ start}> & <T_0 \text{ start}> & <T_0 \text{ start}> \\
    <T_0, A, 950> & <T_0, A, 950> & <T_0, A, 950> \\
    <T_0, B, 2050> & <T_0, B, 2050> & <T_0, B, 2050> \\
    <T_0 \text{ commit}> & <T_0 \text{ commit}> & <T_0 \text{ commit}> \\
    <T_1 \text{ start}> & <T_1 \text{ start}> & <T_1 \text{ start}> \\
    <T_1, C, 600> & <T_1, C, 600> & <T_1 \text{ commit}> \\
  \end{array}
  \]

(a) (b) (c)
Log-based Recovery

**Assumptions:**

- Log records are immediately pushed to the disk as soon as they are generated
- Log records are written to disk in the order generated
- A log record is generated *before* the actual data value is updated
- *Strict two-phase locking*

The first assumption can be relaxed

As a special case, a transaction is considered *committed* only after the `<T1, COMMIT>` has been pushed to the disk

But, this seems like exactly what we are trying to avoid??

- Log writes are *sequential*
- They are also typically on a different disk

Aside: LFS == log-structured file system
Log-based Recovery

Assumptions:

- Log records are immediately pushed to the disk as soon as they are generated.
- Log records are written to disk in the order generated.
- A log record is generated before the actual data value is updated.
- \textit{Strict two-phase locking}.
- The first assumption can be relaxed.
- As a special case, a transaction is considered \textit{committed} only after the \textit{<T1, COMMIT>} has been pushed to the disk.

NOTE: As a result of assumptions 1 and 2, if \textit{data item A} is updated, the log record corresponding to the update is always forced to the disk before \textit{data item A} is written to the disk.

This is actually the only property we need; assumption 1 can be relaxed to just guarantee this (called \textit{write-ahead logging}).
Using the log to *abort/rollback*

- 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, 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*

- **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
    - Restore *B* to 400
    - Restore *A* to 200
  - **Note:** No other transaction better 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

• REDO(T1):
  – Go *forwards* in the log looking for log records belonging to T1
  – Set the values to the new values
  – NOTE: Going forwards 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 idempotent
  – *F is idempotent, if F(x) = F(F(x)) = F(F(F(F(…F(x)))))*

• Multiple applications shouldn’t change the effect
  – This is important because we don’t know exactly what made it to the disk, and we can’t keep track of that
  – E.g. consider a log record of the type
    • <T1, A, *incremented by 100*><br>    • 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, *value based logging* is used (also called *physical*), not operation based (also called *logical*)
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 transaction 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)

• After restart, go backwards into the log, and make two lists
  – How far ?? For now, assume till the beginning of the log.

• undo_list: A list of transactions that must be undone
  – $<Ti, \text{START}>$ record is in the log, but no $<Ti, \text{COMMIT}>$

• redo_list: A list of transactions that need to be redone
  – Both $<Ti, \text{START}>$ and $<Ti, \text{COMMIT}>$ records are in the log

• After that:
  – UNDO all the transactions on the undo_list one by one
  – REDO all the transaction on the redo_list one by one
Restart Recovery (after a crash)

• Must do the UNDOs first before REDO
  – <T1, A, 10, 20>
  – <T1, abort>  
  [[ so A was restored back to 10 ]]
  – <T2, A, 10, 30>
  – <T2, commit>

• If we do UNDO(T1) first, and then REDO(T2), it will be okay

• Trying to do other way around doesn’t work

• NOTE: In reality, most system generate special log records when transactions are aborted, and in that case, they have to do REDO before UNDO
  – However, our scheme doesn’t, so we must do UNDO before REDO
Checkpointing

• How far should we go back in the log while constructing redo and undo lists??
  – 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
    • Entire === 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

• Key difference: Only need to go back till the last checkpoint

• Steps:
  – undo_list:
    • Go back till the checkpoint as before.
    • Add all the transactions that were active at that time, and that didn’t commit
      – e.g. possible that a transactions started before the checkpoint, but didn’t finish till the crash
  – redo_list:
    • Similarly, go back till the checkpoint constructing the redo_list
    • Add all the transactions that were active at that time, and that did commit
  – Do UNDOs and REDOs as before
Recap

• Log-based recovery
  – Uses a log to aid during recovery

• UNDO()
  – Used for normal transaction abort/rollback, as well as during restart recovery

• REDO()
  – Used during restart recovery

• Checkpoints
  – Used to reduce the restart recovery time
Write-ahead logging

• We 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
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 \(<T, \text{commit}>\) record is on disk
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
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
Recap

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

• ACID Properties
  – Atomicity and Durability:
    • Logs, undo(), redo(), WAL etc

  – Consistency and Isolation:
    • Concurrency schemes

  – Strong interactions:
    • We had to assume Strict 2PL for proving correctness of recovery