Transactions: Recovery

Amol Deshpande
CMSC424
Spring 2020 – Online Instruction Plan

- Week 1: File Organization and Indexes
- Week 2: Query Processing
- Week 3: Query Optimization; Parallel Databases 1
- Week 4: Parallel Databases; Mapreduce; Transactions 1
- Week 5: Transactions 2
- Week 6: Homework Due May 8
  - Transactions: Recovery
  - Misc 1: Distributed Transactions, and Object-oriented/Object-relational databases
  - Misc 2: OLAP and Data Cubes, and Information Retrieval
Transactions: Recovery

- Book Chapters
  - 16.1 – 16.4

- Key topics:
  - Challenges in guaranteeing Atomicity and Durability
  - STEAL and NO FORCE: Why those are desirable
  - How to use “logging” to support A and D
  - Key properties including write-ahead logging
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**: 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: 

- 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
  - 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 blocks residing on the disk.
- **Buffer blocks** are the blocks residing 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

Buffer Block A

Buffer Block B

read(X)

write(Y)

work area of T₁

work area of T₂

memory
disk

buffer

input(A)

output(B)

x₁

y₁

x₂

A

B
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
  - We might still have to *force* some pages to disk, but minimal.
STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

- Force
  - No Steal: Desired
  - Steal: Trivial
- No Force
  - No Steal: Trivial
  - Steal: Desired
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.
Deferred Database Modification:
- Similar to NO STEAL, NO FORCE
  - Not identical
- Only need *redos, no undos*
- We won’t cover this today

Immediate Database Modification:
- 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
Example transactions $T_0$ and $T_1$ ($T_0$ executes 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)

Log:
Log-based Recovery

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.
   - 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:

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

**NOTE:** As a result of assumptions 1 and 2, if *data item A* is updated, the log record corresponding to the update is always forced to the disk before *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 **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 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
- Write a log record \(<T_i, X_j, V_1>\)
  - such log records are called *compensation log records*
  - \(<T1, A, 300>, <T1, B, 400>, <T1, A, 200>\)
- **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* \( F(x) = F(F(x)) = F(F(F(F(\ldots F(x)))))) \)*

- 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
    - \(<T1, A, \text{incremented by 100}>\)
    - 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, *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
Recovery from failure: Two phases

- **Redo phase**: replay updates of all transactions, whether they committed, aborted, or are incomplete
- **Undo phase**: undo all incomplete transactions

Redo phase:

1. Find last `<checkpoint L>` record, and set undo-list to $L$.
   - If no checkpoint record, start at the beginning
2. Scan forward from above `<checkpoint L>` record
   1. Whenever a record `<$T_i$, $X_j$, $V_1$, $V_2$>` is found, redo it by writing $V_2$ to $X_j$
   2. Whenever a log record `<$T_i$ start>` is found, add $T_i$ to undo-list
   3. Whenever a log record `<$T_i$ commit>` or `<$T_i$ abort>` is found, remove $T_i$ from undo-list
Recovery Algorithm (Cont.)

■ Undo phase:

1. Scan log backwards from end
   1. Whenever a log record \( <T_i, X_j, V_1, V_2> \) is found where \( T_i \) is in undo-list perform same actions as for transaction rollback:
      1. perform undo by writing \( V_1 \) to \( X_j \).
      2. write a log record \( <T_i, X_j, V_1> \)
   2. Whenever a log record \( <T_i \text{start}> \) is found where \( T_i \) is in undo-list,
      1. Write a log record \( <T_i \text{abort}> \)
      2. Remove \( T_i \) from undo-list
   3. Stop when undo-list is empty
      • i.e. \( <T_i \text{start}> \) has been found for every transaction in undo-list

• After undo phase completes, normal transaction processing can commence
Example of Recovery

Beginning of log

\(<T_0 \text{ start}>\)
\(<T_0, B, 2000, 2050>\)
\(<T_1 \text{ start}>\)
\(<\text{checkpoint \{} T_0, T_1\text{\}}\>
\(<T_1, C, 700, 600>\)
\(<T_1 \text{ commit}>\)
\(<T_2 \text{ start}>\)
\(<T_2, A, 500, 400>\)
\(<T_0, B, 2000>\)
\(<T_0 \text{ abort}>\)
\(<T_2 \text{ abort}>\)

End of log at crash!

Log records added during recovery

Start log records found for all transactions in undo list

Redo Pass

\(T_0\) rollback (during normal operation) begins

\(T_0\) rollback complete

\(T_2\) is incomplete at crash

Undo list: \(T_2\)

\(T_2\) rolled back in undo pass

Undo Pass

newer

older
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 so far …

- 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 \( \text{pageLSN}(P) \) 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