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)
  - $B = B + 50$

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

- Possible logs after crash and restart:

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

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 $T_1$ is considered committed *only after* record $<T_1, COMMIT>$ 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.
    - \(<T1, \text{START}>\)
    - \(<T1, A, 200, 300>\)
    - \(<T1, B, 400, 300>\)
    - \(<T1, A, 300, 200>\) \[\text{[note: second update of A]}\]
  - 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
      - Write \(<T1, \text{CLR, A, 300}>\) record (compensating log record)
    - Restore B to 400
      - Write \(<T1, \text{CLR, B, 400}>\) record
    - Restore A to 200
      - Write \(<T1, \text{CLR, A, 200}>\) record
      - Write \(<T1, \text{ABORT}>\) \[\text{(abort comes after CLR records)}\]
  - 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

- 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 $F(x) = F(F(x)) = F(F(F(...F(x))))$
- 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
    - $<T1, A, \text{incremented by 100}>$
    - 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, 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
    - appending 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
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

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
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 $<T, \text{commit}>$ 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 (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
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