Other CC

Other CC Schemes

- Optimistic concurrency control
  - Also called validation-based

- Intuition
  - Let the transactions execute as they wish
  - At the very end when they are about to commit, check if there might be any problems/conflicts etc
    - If no, let it commit
    - If yes, abort and restart

- Optimistic: The hope is that there won’t be too many problems/aborts
Isolation Levels: Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol

- A type of “optimistic concurrency control”

- Key idea:
  - For each object, maintain past “versions” of the data along with timestamps
    - Every update to an object causes a new version to be generated

Other CC Schemes: Snapshot Isolation

- Read queries:
  - Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of “t”
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of committed database

- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction committed the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)
  - first committer vs. first updater
Other CC Schemes: Snapshot Isolation

- **Advantages:**
  - Read queries do not block, and run very fast
  - As long as conflicts are rare, update transactions don’t abort
  - Overall better performance than locking-based protocols

- **Major disadvantage:**
  - Not serializable!

\[ x = y = 0 \]

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
w(x) & w(y) \\
r(y) & r(x) \\
\end{array}
\]

Other CC Schemes: Snapshot Isolation

- **Banking example**
  - Assume want \( A + B \geq 100 \)

\[ A = B = 75 \]

\[
\begin{array}{c|c}
T_1 & T_2 \\
\hline
r(A) & r(A) \\
r(B) & r(B) \\
w(A) & w(B) \\
commit & commit \\
\end{array}
\]
The “Phantom” problem

- An interesting problem that comes up for dynamic databases
- Schema: `accounts(acct_no, balance, zipcode, …)`
- Transaction 1: Find the number of accounts in `zipcode = 20742`, and divide $1,000,000$ between them
- Transaction 2: Insert `<acctX, …, 20742, …>`
- Execution sequence:
  - T1 locks all tuples corresponding to “zipcode = 20742”, finds the total number of accounts (= num_accounts)
  - T2 does the insert
  - T1 computes $1,000,000 / \text{num_accounts}$
  - When T1 accesses the relation again to update the balances, it finds one new (“phantom”) tuple (the new tuple that T2 inserted)

- Not serializable

- Problem: `locking granularity`

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*
    - **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*
  - **Obvious naïve solutions exist that work, but are too expensive.**
    - E.g. A *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

Buffer Management

- **Buffer manager**
  - sits between DB and disk
  - writing every operation to disk, as it occurs, too slow…
  - ideally only write a block to disk at commit
    - aggregates updates
    - trans might not commit
- **Bottom line**
  - want to *decouple* data writes from DB operations
STEAL vs NO STEAL, FORCE vs NO FORCE

- **STEAL:**
  - The buffer manager *can steal* a (memory) already-used 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:**
  - Stealing not allowed. More control, but less flexibility for the buffer manager → poor performance.

  *Uncommitted changes might be on disk after crash…*

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.

  *Committed changes might NOT be on disk after crash…*
### What if NO STEAL, 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 problems.
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

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

- In lieu of a homework, look at exercises:
  - ch 14, all
  - ch 15 1-3, 10-12, 15-19
  - ch 16 1-3, 9

- Answers:

Log-based Recovery

- Most commonly used recovery method
- 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 $T_1$):
  $T_0$: read (A)
  $A = A - 50$
  write (A)
  $B = B + 50$
  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 $<T_1, COMMIT>$ has been pushed to the disk
  - Also:
    - Log writes are *sequential*
    - They are also typically on a different disk
    - LFS == log-structured file system, and basis of *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

● 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 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.
    - \(<T1, \text{START}>\)
    - \(<T1, A, 200, 300>\)
    - \(<T1, B, 400, 300>\)
    - \(<T1, A, 300, 200>\) \[note: second update of A\]
    - \(<T1, \text{COMMIT}>\)
  - 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 \( F(x) = F(F(x)) = F(F(F(\ldots F(x)))) \)
- 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 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)

- 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
  - $<T_i, \text{START}>$ record is in the log, but no $<T_i, \text{COMMIT}>$

- redo_list: A list of transactions that need to be redone
  - Both $<T_i, \text{START}>$ and $<T_i, \text{COMMIT}>$ 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
  - this is different than the recovery algorithm in 16.4

Restart Recovery (after a crash)

- Must do the UNDOs first before REDO
  - $<T_2, A, 10, 30>$
  - $<T_1, A, 10, 20>$
  - $<T_1, \text{abort}>$  [[ so A was restored back to 10 ]]
  - $<T_2, \text{commit}>$

- If we do UNDO(T1) first, and then REDO(T2), it will be okay
- Trying to do other way around doesn’t work
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 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

Recap

- **STEAL vs NO STEAL, FORCE vs NO FORCE**
  - We studied how to do STEAL and NO FORCE through log-based recovery scheme
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
    - Read 16.7 if interested.

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