Other CC Schemes: Snapshot Isolation

- **Banking example**
  - Assume want $A + B = 100$

\[ A = B = 75 \]

- **T1**
  - `r(A) 75`
  - `r(B) 75`
  - `w(B) 25`

- **T2**
  - `r(A) 75`
  - `r(B) 75`
  - `w(A) 25`

\[ \text{commit} \]

**Other CC Schemes: Snapshot Isolation**

- Initially, $A=1$, $B = 2$
  - **T5** will read $A = 2$
  - **T2** will read $A = 1$
  - **T2** commits
  - **T4** commits

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Start</td>
<td>Start</td>
<td>Start</td>
<td>Start</td>
</tr>
<tr>
<td><code>R(A) \rightarrow 1</code></td>
<td><code>R(A) \rightarrow 1</code></td>
<td><code>R(A) \rightarrow 1</code></td>
<td><code>R(A) \rightarrow 1</code></td>
<td><code>R(A) \rightarrow \text{??????}</code></td>
</tr>
<tr>
<td><code>R(B) \rightarrow 2</code></td>
<td><code>R(B) \rightarrow 2</code></td>
<td><code>R(B) \rightarrow 2</code></td>
<td><code>R(B) \rightarrow 2</code></td>
<td><code>R(B) \rightarrow \text{??????}</code></td>
</tr>
<tr>
<td><code>W(A) = 2</code></td>
<td><code>Commit-Req</code></td>
<td><code>Commit</code></td>
<td></td>
<td><code>Commit-Req</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>W(B) = 0</code></td>
<td></td>
</tr>
</tbody>
</table>
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/num_accounts
  - When T1 accesses the relation again to update the balances, it finds one new (“phantom”) tuple (the new tuple that T2 inserted)
    - update accounts set balance += 1000000/num_accounts where zipcode=20742

- Not serializable

- Problem: locking granularity
  - needed to have lock on whole table

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) an already-used page from the database
  - i.e., 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*, i.e., writes/updates by a transaction that hasn’t committed
  - But, we must allow steal for performance reasons.
    - *Uncommitted changes might be on disk after crash...*

**NO STEAL:**
- Stealing not allowed. More control, but less flexibility for the buffer manager ➔ poor performance.

---

**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?
  - Usually random I/Os ➔ 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...*
### STEAL vs NO STEAL, FORCE vs NO FORCE

<table>
<thead>
<tr>
<th>No Force</th>
<th>Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force</td>
<td>Trivial</td>
</tr>
<tr>
<td>No Steal</td>
<td>Steal</td>
</tr>
</tbody>
</table>

#### 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.
- 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>
  - <T2, ABORT>
  - <T1, A, 100, 200>
    - T1 modified A; old value = 100, new value = 200
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
  - \(<T_1, \text{START}>\)
  - \(<T_2, \text{COMMIT}>\)
  - \(<T_2, \text{ABORT}>\)
  - \(<T_1, A, 100, 200>\)
    - 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)
  - read (B)
  - \(B = B + 50\)
  - write (B)

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

- Possible Logs:

  \(<T_0, \text{start}>\)
  \(<T_0, A, 950, 900>\)
  \(<T_0, B, 2000, 2050>\)
  \(<T_0, \text{commit}>\)
  \(<T_1, \text{start}>\)
  \(<T_1, C, 500, 400>\)

  (a) \(\quad\) (b) \(\quad\) (c)
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.
     - As a special case, a transaction 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?)
  - 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.
    - $<T1, START>$
    - $<T1, A, 200, 300>$
    - $<T1, B, 400, 300>$
    - $<T1, A, 300, 200>$  
      [[ 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 be *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, *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)

- 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
  - this is different than the recovery algorithm in 16.4
Restart Recovery (after a crash)

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