Guarantees of Atomicity and Isolation

- Serializability
  - conflict, view
- Concurrency Control
  - locking
    - 2-phase locking, strict, rigorous
    - granularity, intention locks
- Other approaches to concurrency control
  - snapshot isolation
  - timestamp-based
  - optimistic
- Quiz 8
- Mid-term
- Weak levels of concurrency
- Recovery

Midterm 2

\textit{functional dependences < midterm 2 < serializability}

- quizzes 7, 8, 9
- assignments 5, 6, 7
- mid2s22

Not-necessarily-complete Set of Topics:
- mapping relations to files
- mapping tuples to files
  - no ordering (heap organization)
  - ordered
  - hash (a bucket per page, for example)
- insertions and deletions
- indexes
  - primary vs secondary, dense vs sparse, B+-trees
- query processing
  - cost estimation (selectivity, histograms)
  - query costs: costs to access B+ tree, cost to return
  - sorting (sort-merge)
  - \texttt{joins} (block nested loop, loop w/ index, merge, hash)
Optimistic Concurrency Control

- Each transaction $T_i$ has 3 timestamps
  - $\text{Start}(T_i)$: the time when $T_i$ started its execution
  - $\text{Validation}(T_i)$: the time when $T_i$ enters its validation phase
  - $\text{Finish}(T_i)$: the time when $T_i$ finished its write phase

- Serializability order is validation order
  - $TS(T_i) = \text{Validation}(T_i)$
  - increases concurrency.

- Higher degree of concurrency if conflicts low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Optimistic Concurrency Control

- If for all $T_i$ with $TS(T_i) < TS(T_k)$ either one of the following condition holds:
  - $\text{finish}(T_i) < \text{start}(T_k)$ or
  - $\text{start}(T_k) < \text{finish}(T_i) < \text{validation}(T_k)$ and the set of data items written by $T_i$ does not intersect with the set of data items read by $T_k$.

  then validation succeeds and $T_k$ can be committed. Otherwise, validation fails and $T_k$ is aborted.

- Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of $T_k$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.
  - the writes of $T_i$ do not affect reads of $T_k$ since $T_k$ does not read any item written by $T_i$. 

Optimistic Concurrency Control

- $T_{25} < T_{26}$
- $T_{25}$ validates because first
- $T_{26}$ validates because $T_{25}$ had no writes

<table>
<thead>
<tr>
<th>$T_{25}$</th>
<th>$T_{26}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td></td>
<td>$B := B - 50$</td>
</tr>
<tr>
<td></td>
<td>read($A$)</td>
</tr>
<tr>
<td></td>
<td>$A := A + 50$</td>
</tr>
<tr>
<td>read($A$)</td>
<td>&lt;validate&gt;</td>
</tr>
<tr>
<td>&lt;validate&gt;</td>
<td>display($A + B$)</td>
</tr>
<tr>
<td>&lt;validate&gt;</td>
<td>write($B$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>&lt;commit&gt;</td>
</tr>
</tbody>
</table>

Optimistic Concurrency Control

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;start&gt;</td>
<td>&lt;start&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write($X$) = 1</td>
<td>read($X$) = 0</td>
</tr>
<tr>
<td></td>
<td>&lt;validate&gt;</td>
<td>&lt;validate&gt;</td>
<td>&lt;validate&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;commit&gt;</td>
<td>&lt;commit&gt;</td>
<td>&lt;commit&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;start&gt;</td>
<td>&lt;start&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read($X$) = 1</td>
<td>read($X$) = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;validate&gt;</td>
<td>&lt;validate&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;commit&gt;</td>
<td>&lt;commit&gt;</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 by that number
- 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 bonus = 1,000,000/num_accounts
  - T1 gives bonus to all matching accounts (including T2’s insert!)
    - update accounts set balance += 1000000/num_accounts where zipcode=20742
- Not serializable
- Root problem: locking granularity
  - needed to have lock on whole table

Weak Levels of Isolation in SQL

- SQL allows non-serializable executions
  - Serializable: default, strongest (except for linearizable)
  - Repeatable read: allows only committed records to be read, and repeating a read should return the same value
    - so read locks should be retained or caching used
    - transaction-local writes can change subsequent reads
    - Phantom problem not necessarily prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - Read committed: only read committed data, repeated reads of same data might return different values as other transactions commit
  - Read uncommitted: allows even uncommitted data to be read
- In many database systems, read committed is the default
  - has to be explicitly changed to serializable when required
    - set isolation level serializable
  - Oracle calls snapshot isolation “serializable”
### Weak Isolation Levels: Read Uncommitted

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\hline
x = 0 & & \\
\text{start} & \text{write}(X) = 1 & \\
\text{commit} & & \\
\text{start} & \text{read}(X) = 1 & \text{write}(X) = 3 \\
\text{commit} & & \text{read}(X) = 3 \\
\text{start} & \text{write}(X) = 2 & \\
\text{commit} & & \\
\end{array}
\]

- Not serializable
- Doesn’t guarantee recoverable scheds
- Not free from cascading aborts
- Not serializable

### Weak Isolation Levels: Read Committed

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
\hline
x = 0 & & \\
\text{start} & \text{write}(X) = 1 & \\
\text{commit} & & \\
\text{start} & \text{read}(X) = 0 & \text{write}(X) = 3 \\
\text{commit} & & \text{read}(X) = 1 \\
\text{start} & \text{write}(X) = 2 & \\
\text{commit} & & \\
\end{array}
\]

- Not serializable
- Guarantees recoverable scheds
- Free from cascading aborts
- Not serializable but stronger isolation
### Weak Isolation Levels: Repeatable Reads II

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Start write(X) = 1
- Start read(X) = 0
- Commit
- Start read(X) = 0
- Read(X) = 0
- Commit
- Start read(X) = 1
- Read(X) = 0
- Commit
- Start write(X) = 2
- Commit

- Not serializable
- Guarantees recoverable scheds
- Free from cascading aborts
- Not serializable, but even stronger isolation

### Weak Isolation Levels: Snapshot Iso

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₀</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Start write(X) = 1
- Start read(X) = 0
- Write(X) = 3
- Read(X) = 3
- Commit
- Start write(X) = 2
- Commit
- Start read(X) = 0
- Write(X) = 3
- Read(X) = 3

- Not serializable
- Guarantees recoverable scheds
- Free from cascading aborts
- Faster
- Abort
The Real World

- Shopping cart
  - add to cart
  - becomes unavailable before you check out
  - definitely < repeatable reads
- Airline reservation w/ seat selections
  - multiple passengers reserving concurrently
  - all good until you hit the PAY button
    - then disappears
  - snapshot isolation?

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**: 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, 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

Data Access

• Physical blocks are those residing on the disk.
• Buffer blocks are those 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.
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:
    - $\text{read}(X)$ assigns the value of data item $X$ to the local variable $x_i$.
    - $\text{write}(X)$ 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
    • 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.

• 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.
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"