Concurrency Control

Recap, Next….

- Deadlocks
  - Detection, prevention, recovery

- Locking granularity
  - Arranged in a hierarchy
  - Intentional locks

- Next…
  - Brief discussion of some other concurrency schemes
Time-stamp based CC

- Transactions are issued time-stamps
  - When they enter the system
  - Time-stamps determine the serializability order
  - If T1 entered before T2, Then T1 before T2 in the serializability order
- Say \(\text{timestamp}(T1) < \text{timestamp}(T2)\)
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is aborted
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read or written that data item, then the write is rejected and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of starvation

Time-stamp based CC

- Example

<table>
<thead>
<tr>
<th>(T_1)</th>
<th>(T_2)</th>
<th>(T_3)</th>
<th>(T_4)</th>
<th>(T_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read((Y))</td>
<td>read((Y))</td>
<td>write((Y))</td>
<td>write((X))</td>
<td></td>
</tr>
<tr>
<td>read((X))</td>
<td>read((X))</td>
<td>write((Z))</td>
<td></td>
<td>read((Z))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write((Z))</td>
<td></td>
<td>write((Y)) write((Z))</td>
</tr>
</tbody>
</table>
**Time-stamp based CC**

- Example

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>write($X$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read($X$)</td>
<td>abort</td>
<td>write($Z$)</td>
<td>read($Z$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>abort</td>
<td></td>
<td>write($Z$)</td>
<td>abort</td>
<td></td>
</tr>
</tbody>
</table>

$\text{TS}(T_1) < \text{TS}(T_2) < \text{TS}(T_3) < \text{TS}(T_4) < \text{TS}(T_5)$

**Time-stamp based CC**

- The following set of instructions is not conflict-serializable:

<table>
<thead>
<tr>
<th></th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($Q$)</td>
<td>write($Q$)</td>
</tr>
<tr>
<td></td>
<td>write($Q$)</td>
<td></td>
</tr>
</tbody>
</table>

- Also not view-serializable:
  - if $T_i$ reads initial value of $Q$ in $S$, must also in $S'$
    - not true if $T_4 \rightarrow T_3$
  - if $T_i$ reads value written from $T_j$ in $S$, must also in $S'$
    - not true if $T_3 \rightarrow T_4$
  - if $T_i$ performs final write to $Q$ in $S$, must also in $S'$
**Time-stamp based CC**

- **Thomas’ Write Rule**
  - Ignore obsolete writes

<table>
<thead>
<tr>
<th>Time-stamp based CC</th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td></td>
<td>write($Q$)</td>
</tr>
<tr>
<td>write($Q$)</td>
<td>ignored</td>
<td></td>
</tr>
</tbody>
</table>

- **Say $\text{timestamp}(T_1) < \text{timestamp}(T_2)$**
  - If $T_1$ wants to read data item $A$
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and $T_1$ is aborted
  - If $T_1$ wants to write data item $A$
    - If a transaction with larger time-stamp already read or written that data item, then the write is rejected and $T_1$ is aborted
    - If a transaction with larger time-stamp already written that data item, then the write is ignored

**Other CC Schemes**

- **Time-stamp based**
  - Many potential problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
  - Most can be solved fairly easily
    - Read up
  - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to find the minimal set of restrictions to as to not hinder concurrency
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

Other CC Schemes

Example Optimistic: Snapshot Isolation

- Primary scheme of Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol

- 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
  - All reads use versions at a single timestamp
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!

\[ x = y = 0 \]

\[
\begin{array}{cc}
T_1 & T_2 \\
\text{w(x)} & \text{w(y)} \\
\text{r(y)} & \text{r(x)} \\
1 & 1 \\
0 & 0
\end{array}
\]
Other CC Schemes: Snapshot Isolation

- Banking example
  - Assume (checking+savings) must be $\geq 100$

\[ ch = sa = 75 \]

<table>
<thead>
<tr>
<th></th>
<th>T₁</th>
<th>T₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(ch)</td>
<td>75</td>
<td>r(ch) 75</td>
</tr>
<tr>
<td>r(sa)</td>
<td>75</td>
<td>r(sa) 75</td>
</tr>
<tr>
<td>w(ch)</td>
<td>25</td>
<td>w(sa) 25</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

Isolation Levels

- serializability
  - as before
- read committed
  - only read values committed to DB
- repeated read
  - reads do not change
- read uncommitted
  - can read dirty data

All of these also prohibit *dirty writes*, i.e. modifications to data already modified by concurrent transactions.
Isolation Levels: Snapshot Isolation

\[ X = Y = 1 \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_trans()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r(X) = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ X = Y = 1 \]

begin_trans()

w(X)

w(Y)

r(Y) = 1

commit()

Isolation Levels: Read-committed

- features:
  - reads only see committed data
  - returning prior versions if current data modified

\[ X = Y = 1 \]

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_trans()</td>
<td></td>
</tr>
<tr>
<td>r(X) = 1</td>
<td></td>
</tr>
</tbody>
</table>

begin_trans()

w(X)

w(Y)

r(Y) = 1

commit()

r(Y) = 2

commit()
Isolation Levels: Repeatable read

- **features:**
  - *read-committed*
  - once *v* returned for data, either it or transac write must be returned

\[
X = Y = 1
\]

\[
\begin{align*}
&T1 &T2 \\
&\text{begin\_trans()} &\text{begin\_trans()} \\
&r(X) = 1 &w(X)2 \\
&w(X)2 &w(Y)2 \\
&r(Y) = 1 &\text{commit()} \\
&r(Y) = 1 &
\end{align*}
\]

The “Phantom” problem (what to lock)

- **An interesting problem in 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”) tuples (the new tuple that T2 inserted)

- **Not serializable**
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; \textit{for now we will assume}
    - \textbf{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 \textit{disk is durable}
  - The only \textit{atomicity} guarantee is that a \textit{disk block write} is \textit{atomic}
- **Obvious naïve solutions exist that work, but are too expensive.**
  - E.g. A \textit{shadow copy} solution
    - Make a copy of the database; do the changes on the copy; do an atomic switch of the \textit{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) 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/uploads 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…*