Advanced Databases

Lecture 10 - Concurrency Control (continued)

Masood Niazi Torshiz
Islamic Azad University- Mashhad Branch
www.mniazi.ir
Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful write results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a read($Q$) operation is issued, select an appropriate version of $Q$ based on the timestamp of the transaction, and return the value of the selected version.
- reads never have to wait as an appropriate version is returned immediately.
Multiversion Timestamp Ordering

- Each data item $Q$ has a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$. Each version $Q_k$ contains three data fields:
  - **Content** -- the value of version $Q_k$.
  - **W-timestamp**($Q_k$) -- timestamp of the transaction that created (wrote) version $Q_k$.
  - **R-timestamp**($Q_k$) -- largest timestamp of a transaction that successfully read version $Q_k$.

- When a transaction $T_i$ creates a new version $Q_k$ of $Q$, $Q_k$'s W-timestamp and R-timestamp are initialized to $TS(T_i)$.
- R-timestamp of $Q_k$ is updated whenever a transaction $T_j$ reads $Q_k$, and $TS(T_j) > R$-timestamp($Q_k$).
Multiversion Timestamp Ordering (Cont)

- Suppose that transaction $T_i$ issues a read($Q$) or write($Q$) operation. Let $Q_k$ denote the version of $Q$ whose write timestamp is the largest write timestamp less than or equal to TS($T_i$).
  1. If transaction $T_i$ issues a read($Q$), then the value returned is the content of version $Q_k$.
  2. If transaction $T_i$ issues a write($Q$)
     1. if TS($T_i$) < R-timestamp($Q_k$), then transaction $T_i$ is rolled back.
     2. if TS($T_i$) = W-timestamp($Q_k$), the contents of $Q_k$ are overwritten
     3. else a new version of $Q$ is created.

- Observe that
  - Reads always succeed
  - A write by $T_i$ is rejected if some other transaction $T_j$ that (in the serialization order defined by the timestamp values) should read $T_i$'s write, has already read a version created by a transaction older than $T_i$.

- Protocol guarantees serializability
Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions

  - *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
    - Each successful *write* results in the creation of a new version of the data item written.
    - Each version of a data item has a single timestamp whose value is obtained from a counter *ts-counter* that is incremented during commit processing.

- *Read-only transactions* are assigned a timestamp by reading the current value of *ts-counter* before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.
When an update transaction wants to read a data item:
- it obtains a shared lock on it, and reads the latest version.

When it wants to write an item:
- it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to $\infty$.

When update transaction $T_i$ completes, commit processing occurs:
- $T_i$ sets timestamp on the versions it has created to $\text{ts-counter} + 1$
- $T_i$ increments $\text{ts-counter}$ by 1

Read-only transactions that start after $T_i$ increments $\text{ts-counter}$ will see the values updated by $T_i$.

Read-only transactions that start before $T_i$ increments the $\text{ts-counter}$ will see the value before the updates by $T_i$.

Only serializable schedules are produced.
MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
Snapshot Isolation

- **Motivation:** Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- **Solution 1:** Give logical “snapshot” of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- **Solution 2:** Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005
A transaction T1 executing with Snapshot Isolation:
- Takes snapshot of committed data at start.
- Always reads/modifies data in its own snapshot.
- Updates of concurrent transactions are not visible to T1.
- Writes of T1 complete when it commits.

First-committee-wins rule:
- Commits only if no other concurrent transaction has already written data that T1 intends to write.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(Y := 1)</td>
<td>Start</td>
<td>W(X:=2)</td>
</tr>
<tr>
<td>Commit</td>
<td>R(X) → 0</td>
<td>W(Z:=3)</td>
</tr>
<tr>
<td></td>
<td>R(Y) → 1</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>R(Z) → 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(Y) → 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(X:=3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit-Req</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Abort</td>
<td></td>
</tr>
</tbody>
</table>

Concurrent updates not visible
Own updates are visible
Not first-committee of X
Serialization error, T2 is rolled back
**Snapshot Read**

- Concurrent updates invisible to snapshot read

\[ X_0 = 100, \ Y_0 = 0 \]

<table>
<thead>
<tr>
<th></th>
<th>( T_1 ) deposits 50 in ( Y )</th>
<th>( T_2 ) withdraws 50 from ( X )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1(X_0, 100) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r_1(Y_0, 0) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( w_1(Y_1, 50) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( r_1(X_0, 100) ) (update by ( T_2 ) not seen)</td>
<td></td>
<td>( r_2(Y_0, 0) ) (update by ( T_1 ) not seen)</td>
</tr>
<tr>
<td>( r_1(Y_1, 50) ) (can see its own updates)</td>
<td></td>
<td>( r_2(X_0, 100) )</td>
</tr>
</tbody>
</table>

\[ X_2 = 50, \ Y_1 = 50 \]
Snapshot Write: First Committer Wins

<table>
<thead>
<tr>
<th>$T_1$ deposits 50 in $X$</th>
<th>$T_2$ withdraws 50 from $X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1(X_0, 100)$</td>
<td>$r_2(X_0, 100)$</td>
</tr>
<tr>
<td>$w_1(X_1, 150)$</td>
<td>$w_2(X_2, 50)$</td>
</tr>
<tr>
<td>$commit_1$</td>
<td>$commit_2$ (Serialization Error $T_2$ is rolled back)</td>
</tr>
</tbody>
</table>

$X_0 = 100$
$X_1 = 150$

- **Variant: “First-updater-wins”**
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent
Benefits of SI

- Reading is *never* blocked,
  - and also doesn’t block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated
Snapshot Isolation

- E.g. of problem with SI
  - T1: x := y
  - T2: y := x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ??, y = ??

- Called skew write

- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
Snapshot Isolation Anomalies

- SI breaks serializability when txns modify different items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot
SI In Oracle and PostgreSQL

**Warning**: SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1

- PostgreSQL’s implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
- Oracle implements “first updater wins” rule (variant of “first committer wins”)
  - concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution
- PostgreSQL 9.1 introduced new protocol called “Serializable Snapshot Isolation” (SSI)
  - Which guarantees true serializability including handling predicate reads (coming up)
SI In Oracle and PostgreSQL

- Can sidestep SI for specific queries by using `select .. for update` in Oracle and PostgreSQL.
  - E.g.,
    1. `select max(orderno) from orders for update`
    2. Read value into local variable `maxorder`
    3. Insert into orders `(maxorder+1, ..)`

  - Select for update (SFU) treats all data read by the query as if it were also updated, preventing concurrent updates.
  - Does not always ensure serializability since phantom phenomena can occur (coming up).

- In PostgreSQL versions < 9.1, SFU locks the data item, but releases locks when the transaction completes, even if other concurrent transactions are active.
  - Not quite same as SFU in Oracle, which keeps locks until all concurrent transactions have completed.
Insert and Delete Operations

- If two-phase locking is used:
  - A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple.

- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge)
    and a transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)
    (conceptually) conflict in spite of not accessing any tuple in common.

- If only tuple locks are used, non-serializable schedules can result
  - E.g. the scan transaction does not see the new account, but reads some other tuple written by the update transaction.
The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.

The conflict should be detected, e.g. by locking the information.

One solution:

- Associate a data item with the relation, to represent the information about what tuples the relation contains.
- Transactions scanning the relation acquire a shared lock in the data item,
- Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)

Above protocol provides very low concurrency for insertions/deletions.

Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.
Index Locking Protocol

- Index locking protocol:
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation.
  - A transaction $T_i$ that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode.
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range).
  - A transaction $T_i$ that inserts, updates or deletes a tuple $t_i$ in a relation $r$
    - must update all indices to $r$
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete.
  - The rules of the two-phase locking protocol must be observed.
  - Guarantees that phantom phenomenon won’t occur.
Next-Key Locking

- Index-locking protocol to prevent phantoms required locking entire leaf
  - Can result in poor concurrency if there are many inserts
- Alternative: for an index lookup
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures that range queries will conflict with inserts/deletes/updates
  - Regardless of which happens first, as long as both are concurrent
Concurreny in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a B+-tree are irrelevant so long as we land up in the correct leaf node.
Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
  - Use **crabbing** instead of two-phase locking on the nodes of the B+-tree, as follows. During search/insertion/deletion:
    - First lock the root node in shared mode.
    - After locking all required children of a node in shared mode, release the lock on the node.
    - During insertion/deletion, upgrade leaf node locks to exclusive mode.
    - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
  - Above protocol can cause excessive deadlocks
    - Searches coming down the tree deadlock with updates going up the tree
    - Can abort and restart search, without affecting transaction
  - Better protocols are available; see Section 16.9 for one such protocol, the B-link tree protocol
    - Intuition: release lock on parent before acquiring lock on child
      - And deal with changes that may have happened between lock release and acquire
Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur

- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency
Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - set isolation level serializable
Transactions across User Interaction

- Many applications need transaction support across user interactions
  - Can’t use locking
  - Don’t want to reserve database connection per user
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - `select r.balance, r.version into :A, :version`
    - `from r where acctId = 23`
    - When writing tuple, check that current version number is same as the version when tuple was read
      - `update r set r.balance = r.balance + :deposit`
      - `where acctId = 23 and r.version = :version`
- Equivalent to optimistic concurrency control without validating read set
- Used internally in Hibernate ORM system, and manually in many applications
- Version numbering can also be used to support first committer wins check of snapshot isolation
  - Unlike SI, reads are not guaranteed to be from a single snapshot