Chapter 15 : Concurrency Control
Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency

- Testing a schedule for serializability after it has executed is a little too late

- **Goal** – to develop concurrency control protocols that will assure serializability
  - Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes:
  1. *exclusive* (*X*) *mode*. Data item can be both read as well as written. X-lock is requested using *lock-X* instruction.
  2. *shared* (*S*) *mode*. Data item can only be read. S-lock is requested using *lock-S* instruction.
- Lock requests are made to concurrency-control manager.
- Transaction can proceed only after request is granted.
Granting of Locks

- **Lock-compatibility matrix**

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item.
- If any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released.
Example of a transaction performing locking:

\[ T_2: \text{lock-S}(A); \]
\[ \text{read}(A); \]
\[ \text{unlock}(A); \]
\[ \text{lock-S}(B); \]
\[ \text{read}(B); \]
\[ \text{unlock}(B); \]
\[ \text{display}(A+B) \]

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

Locking protocols restrict the set of possible schedules.
Pitfalls of Lock-Based Protocols

- Consider the partial schedule

<table>
<thead>
<tr>
<th></th>
<th>( T_3 )</th>
<th>( T_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>lock-x (( B ))</td>
<td>lock-s (( A ))</td>
</tr>
<tr>
<td></td>
<td>read (( B ))</td>
<td>read (( A ))</td>
</tr>
<tr>
<td></td>
<td>( B := B - 50 )</td>
<td>lock-s (( B ))</td>
</tr>
<tr>
<td></td>
<td>write (( B ))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock-x (( A ))</td>
<td></td>
</tr>
</tbody>
</table>

- Neither \( T_3 \) nor \( T_4 \) can make progress
  - Executing \textbf{lock-S}(\( B \)) causes \( T_4 \) to wait for \( T_3 \) to release its lock on \( B \)
  - Executing \textbf{lock-X}(\( A \)) causes \( T_3 \) to wait for \( T_4 \) to release its lock on \( A \)
- Such a situation is called a \textbf{deadlock}
  - To handle a deadlock one of \( T_3 \) or \( T_4 \) must be rolled back and its locks released
Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols
  - Deadlocks are a necessary evil
- **Starvation** is also possible if concurrency control manager is badly designed
- Example
  - A transaction may be waiting for an X-lock on an item,
  - while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks
- Concurrency control manager can be designed to prevent starvation
Two-Phase Locking Protocol (2PL)

- This is a protocol which ensures conflict-serializable schedules
- **Phase 1: Growing Phase**
  - transaction may obtain locks
  - transaction may not release locks
- **Phase 2: Shrinking Phase**
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures (conflict) serializability
  - The transactions can be serialized in the order of their **lock points**
    (i.e., the point where a transaction acquired its final lock)
  - There can be conflict serializable schedules that cannot be obtained if two-phase locking is used
## Partial Schedule Under 2PL

<table>
<thead>
<tr>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock-S($B$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read($B$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlock($A$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock-X($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock($A$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock-S($A$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read($A$)</td>
</tr>
</tbody>
</table>
Strict / Rigorous 2PL

- Two-phase locking does not ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking

- **Strict two-phase locking**
  - Here a transaction must hold all its exclusive locks till it commits/aborts
  - No cascading rollback

- **Rigorous two-phase locking**
  - Here all locks (shared and exclusive) are held till commit/abort
  - No cascading rollback (of course)
  - In this protocol transactions can be serialized in the order in which they commit
Lock Conversions

- The original lock mode with (lock-X, lock-S)
  - assign lock-X on a data D when D is both read and written
- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability
- The refined 2PL gets more concurrency than the original 2PL
Deadlock Detection

- Deadlocks can be described as a **wait-for graph**, which consists of a pair \( G = (V,E) \),
  - \( V \) is a set of vertices (all the transactions in the system)
  - \( E \) is a set of edges; each element is an ordered pair \( T_i \rightarrow T_j \).
- If \( T_i \rightarrow T_j \) is in \( E \), then there is a directed edge from \( T_i \) to \( T_j \), implying that \( T_i \) is waiting for \( T_j \) to release a data item
- When \( T_i \) requests a data item currently being held by \( T_j \), then the edge \( T_i \rightarrow T_j \) is inserted in the wait-for graph
  - This edge is removed only when \( T_j \) is no longer holding a data item needed by \( T_i \)
- The system is in a deadlock state if and only if the wait-for graph has a cycle
  - Must invoke a deadlock-detection algorithm **periodically** to look for cycles
Deadlock Detection (Cont.)

Wait-for graph without a cycle

Wait-for graph with a cycle
Multiversion Two-Phase Locking

- **Motivation:** Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- **Multiversion schemes** keep old versions of data item to increase concurrency
  - Differentiates between read-only transactions and update transactions
  - **Ts-counter** is a global time-stamp clock
    - This is incremented during commit processing
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction
  - 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 **ts-counter**
- **Read-only transactions** are assigned a timestamp by reading the current value of **ts-counter** before they start execution
Multiversion Two-Phase Locking (Cont.)

- 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
- Problem: works well, but how does system know a transaction is read only?
Snapshot Isolation

A transaction T2 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 T2
- writes of T2 complete when it commits
- **First-committer-wins rule:**
  - Commits only if no other concurrent transaction has already written data that T2 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></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(X) \rightarrow 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(Y) \rightarrow 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(X:=2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(Z:=3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>R(Z) \rightarrow 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(Y) \rightarrow 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(X:=3)</td>
<td>Commit-Req</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abort</td>
</tr>
</tbody>
</table>

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back
Benefits of Snapshot Isolation

- Reading is *never* blocked
  - and also doesn’t block other txs 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
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txs, one sees the effects of the other
    - SI: neither sees the effects of the other
  - Result: integrity constraints can be violated
- Variants implemented in many database systems
  - E.g., Oracle, PostgreSQL, SQL Server 2005
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**: only committed records can be read, but successive reads of record may return different (but committed) values
  - **Read uncommitted**: even uncommitted records may 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`
End of Chapter 15