CSI T5300: Advanced Database Systems

L12: Locking Protocols

Dr. Kenneth LEUNG

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China
The GOAL is to generate schedules that enforce the following transaction properties:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
Concurrent is Based on the Concept of Conflict Serializability

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read(Z)</td>
<td>read(Y)</td>
</tr>
<tr>
<td></td>
<td>read(Y)</td>
<td>write(Y)</td>
</tr>
<tr>
<td></td>
<td>write(X)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(Y)</td>
</tr>
<tr>
<td></td>
<td>write(Y)</td>
<td>write(X)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write(Z)</td>
</tr>
</tbody>
</table>

Schedule is **NOT** serializable (circle).
Therefore, it should not be generated by any concurrency control scheme.
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 the concurrency-control manager. Transaction can proceed only after request is granted.

Should only allow conflict-serializable schedules.
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, but if any transaction holds an exclusive 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. The lock is then granted.

<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 that displays \( A + B \):

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

- Locking as above is not sufficient to guarantee serializability: if \( A \) and \( B \) get updated in-between the read of \( A \) and \( B \), the displayed sum would be wrong.

- 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.
Neither $T_3$ nor $T_4$ can make progress: executing `lock-S(B)` causes $T_4$ to wait for $T_3$ to release its lock on $B$, while executing `lock-X(A)` causes $T_3$ to wait for $T_4$ to release its lock on $A$.

Such a situation is called a **deadlock**.

- To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released.
The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

**Starvation** is also possible if the 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.
2PL 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 serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).

- If a schedule is executed by 2PL it must be conflict serializable.
- If a schedule is conflict serializable it may or may not be executed by 2PL.
• 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)
A lock manager can be implemented as a separate process to which transactions send lock and unlock requests. The lock manager replies to a lock request by sending a lock grant message (or a message asking the transaction to roll back, in case of a deadlock). The requesting transaction waits until its request is answered. The lock manager maintains a data structure called a lock table to record granted locks and pending requests. The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.
Black rectangles indicate granted locks, white ones indicate waiting requests. Lock table also records the type of lock granted or requested. New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks. Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted. If transaction aborts, all waiting or granted requests of the transaction are deleted. Lock manager may keep a list of locks held by each transaction, to implement this efficiently.
• **Cascading roll-back** is the situation where the failure of a transaction $T_i$ may lead to failures of other transactions (because they read items written by $T_i$ before its commitment)

• Cascading roll-back is possible under two-phase locking.

• To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.

• **Rigorous two-phase locking** is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.