CS 44800: Introduction To
Relational Database Systems

Lock Management
Prof. Chris Clifton
29 March 2021

Shared locks

So far:
S = ...l_1(A) r_1(A) u_1(A) ... l_2(A) r_2(A) u_2(A) ...

Do not conflict

Instead:
S = ... ls_1(A) r_1(A) ls_2(A) r_2(A) ... us_1(A) us_2(A)
Operations

Lock actions
l-\text{t}_i(A): lock A in t mode (t is S or X)
u-\text{t}_i(A): unlock t mode (t is S or X)

Shorthand:
u_i(A): unlock whatever modes $T_i$ has locked A

What about transactions that read and write same object?

Option 1: Request exclusive lock
$T_i = \ldots l-X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots$

Option 2: Upgrade
$T_i = \ldots l-S_1(A) \ldots r_1(A) \ldots l-X_1(A) \ldots w_1(A) \ldots u(A) \ldots$

Think of
- Get 2nd lock on A, or
- Drop S, get X lock
Locking Rules

- **Rule #1**  Well formed transactions
  - $T_i = \ldots \text{I-S}_1(A) \ldots \text{r}_1(A) \ldots \text{u}_1(A) \ldots$
  - $T_i = \ldots \text{I-X}_1(A) \ldots \text{w}_1(A) \ldots \text{u}_1(A) \ldots$

- **Rule #2**  Legal scheduler
  - $S = \ldots \text{I-S}_i(A) \ldots \text{u}_i(A) \ldots \text{no I-X}_i(A)$
  - $S = \ldots \text{I-X}_i(A) \ldots \text{u}_i(A) \ldots \text{no I-X}_i(A) \text{no I-S}_i(A)$

A way to summarize Rule #2

- 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>
Rule # 3  2PL transactions

No change except for upgrades:
(I) If upgrade gets more locks
   (e.g., S → {S, X}) then no change!
(II) If upgrade releases read (shared) lock (e.g., S → X)
    - can be allowed in growing phase

Why this works

- **Theorem**: Rules 1,2,3 ⇒ Conflict serializable for S/X lock schedules
- **Proof**: similar to X locks case
  - **Detail**:
    - l-ti(A), l-rj(A) do not conflict if comp(t,r)
    - l-ti(A), u-rj(A) do not conflict if comp(t,r)
Implementation of Locking

- A lock manager can be implemented as a separate process
- Transactions can send lock and unlock requests as messages
- The lock manager replies to a lock request by sending a lock grant messages (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 an in-memory data-structure called a lock table to record granted locks and pending requests

Lock table: Conceptually

Every possible object

A
B
C

If null, object is unlocked

Lock info for B
Lock info for C
**Lock Table**

- Dark rectangles indicate granted locks, light colored 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.

**Deadlock Handling**

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-$X(B)$</td>
<td>lock-$S(A)$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$B := B - 50$</td>
<td>lock-$S(B)$</td>
</tr>
<tr>
<td>write($B$)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-$X(A)$</td>
<td></td>
</tr>
</tbody>
</table>
Deadlock Detection

- **Wait-for graph**
  - **Vertices**: transactions
  - **Edge from** \( T_i \rightarrow T_j \): if \( T_i \) is waiting for a lock held in conflicting mode by \( T_j \)

- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.

![Wait-for graph without a cycle](image1)
![Wait-for graph with a cycle](image2)

Deadlock Recovery

- **When deadlock is detected**:
  - Some transaction will have to rolled back (made a victim) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
    - Rollback -- determine how far to roll back transaction
      - **Total rollback**: Abort the transaction and then restart it.
      - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
  - Starvation can happen (why?)
    - One solution: oldest transaction in the deadlock set is never chosen as victim
Deadlock Handling

- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre-declaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock.

- **wound-wait** scheme — preemptive
  - Older transaction **wounds** (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.

- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.
Deadlock prevention (Cont.)

- **Timeout-Based Schemes:**
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set $D = \{d_1, d_2, ..., d_n\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both $d_i$ and $d_j$ must access $d_i$ before accessing $d_j$.
  - Implies that the set $D$ may now be viewed as a directed acyclic graph, called a database graph.
- The tree-protocol is a simple kind of graph protocol.
**Tree Protocol**

- Only exclusive locks are allowed.
- The first lock by $T_i$ may be on any data item. Subsequently, a data $Q$ can be locked by $T_i$ only if the parent of $Q$ is currently locked by $T_i$.
- Data items may be unlocked at any time.
- A data item that has been locked and unlocked by $T_i$ cannot subsequently be relocked by $T_i$.

**Graph-Based Protocols (Cont.)**

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - Shorter waiting times, and increase in concurrency
  - Protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - Increased locking overhead, and additional waiting time
    - Potential decrease in concurrency
  - Schedules not possible under two-phase locking are possible under the tree protocol, and vice versa.
What are the objects we lock?

<table>
<thead>
<tr>
<th>Relation A</th>
<th>Tuple A</th>
<th>Disk block A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation B</td>
<td>Tuple B</td>
<td>Disk block B</td>
</tr>
<tr>
<td></td>
<td>Tuple C</td>
<td></td>
</tr>
</tbody>
</table>

DB

DB

DB

Locking works in any case, but should we choose small or large objects?

- If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency
- If we lock small objects (e.g., tuples, fields)
  - Need more locks
  - More concurrency