CSE232A: Database System
Principle

Concurrency Control

Chapter 9
Concurrency Control

Example:

T1: Read(A)  
    A ← A+100  
    Write(A)  
    Read(B)  
    B ← B+100  
    Write(B)  
    Constraint: A=B

T2: Read(A)  
    A ← A×2  
    Write(A)  
    Read(B)  
    B ← B×2  
    Write(B)
Serial Schedule A ("good" by definition)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A+2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

Serial Schedule B (equally "good")

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

Interleaved Schedule C (good because it is equivalent to A)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

Scoping "equivalence" is tricky; for now think that A and C are equivalent because if they start from same initial values they end up with same results.
Interleaved Schedule D (bad!)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td>Read(B); B ← B+100;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(B);</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>

Schedule E (good by “accident”)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2’</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×1;</td>
<td>Read(B); B ← B×1;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(B);</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>

The accident being the particular semantics

• Want schedules that are “good”, i.e., equivalent to serial regardless of
  – initial state and
  – transaction semantics
• Only look at order of read and writes

Example:
SC=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)
Example:
\[ SC = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\[ SC' = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\[ T_1 \quad T_2 \]

However, for SD:
\[ SD = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

- as a matter of fact,
  \( T_2 \) must precede \( T_1 \)
in any equivalent schedule,
i.e., \( T_2 \rightarrow T_1 \)
- And vice versa

\[ T_2 \rightarrow T_1 \]

\[ \text{Also, } T_1 \rightarrow T_2 \]

\[ T_1 \rightarrow T_2 \]

\[ SD \text{ cannot be rearranged into a serial schedule} \]
\[ SD \text{ is not “equivalent” to any serial schedule} \]
\[ SD \text{ is “bad”} \]
Returning to Sc

\[ SC = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\( T_1 \rightarrow T_2 \quad T_1 \rightarrow T_2 \)

- no cycles \( \Rightarrow \) SC is “equivalent” to a serial schedule
  (in this case \( T_1, T_2 \))

Concepts

*Transaction*: sequence of \( r_i(x), w_i(x) \) actions

*Conflicting actions*: \( r_1(A), w_2(A), w_1(A), w_2(A), r_1(A), w_2(A) \)

*Schedule*: represents chronological order in which actions are executed

*Serial schedule*: no interleaving of actions or transactions

What about concurrent actions?

\( T_i \) issues \( \text{System Input}(X) \)

\( \text{read}(x, t) \) issues completes

\( T_2 \) issues \( \text{write}(B, S) \)

\( \text{System issues input}(B) \)

\( B \leftarrow S \)

\( \text{System issues output}(B) \)

\( \text{output}(B) \) completes
So net effect is either
• $S = \ldots r_1(x) \ldots w_2(B) \ldots$ or
• $S = \ldots w_2(B) \ldots r_1(x) \ldots$

What about conflicting, concurrent actions on same object?

\begin{center}
\begin{tikzpicture}
\draw[->] (-1,0) -- (5,0) node{time};
\draw (0,0) -- (0,-0.5) node{start $w_2(A)$};
\draw (0,0) -- (0,0.5) node{start $r_1(A)$};
\draw (5,0) -- (5,-0.5) node{end $w_2(A)$};
\draw (5,0) -- (5,0.5) node{end $r_1(A)$};
\end{tikzpicture}
\end{center}

• Assume equivalent to either $r_1(A) w_2(A)$ or $w_2(A) r_1(A)$
• $\Rightarrow$ low level synchronization mechanism
• Assumption called “atomic actions”

Definition

$S_1$, $S_2$ are conflict equivalent schedules if $S_1$ can be transformed into $S_2$ by a series of swaps on non-conflicting actions.
Definition
A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

Precedence graph $P(S)$ ($S$ is schedule)
Nodes: transactions in $S$
Arcs: $T_i \rightarrow T_j$ whenever
- $p(A), q(A)$ are actions in $S$
- $p(A) <_S q(A)$
- at least one of $p, q$ is a write

Exercise:
- What is $P(S)$ for $S = w_3(A) w_2(C) r_1(A) w_1(B) r_1(C) w_2(A) r_4(A) w_4(D)$

- Is $S$ serializable?
**Lemma**

S₁, S₂ conflict equivalent \(\Rightarrow P(S₁) = P(S₂)\)

**Proof:**

Assume \(P(S₁) \neq P(S₂)\)

\(\Rightarrow \exists Tᵢ, Tⱼ: Tᵢ \rightarrow Tⱼ\) in S₁ and not in S₂

\(\Rightarrow S₁ = ...p(A)... q(A)... pᵢ, qᵢ\)

\(S₂ = ...q(A)...p(A)...\) conflict

\(\Rightarrow S₁, S₂\) not conflict equivalent

**Note:** \(P(S₁) = P(S₂) \neq S₁, S₂\) conflict equivalent

**Counter example:**

\(S₁ = w₁(A) r₂(A) w₂(B) r₁(B)\)

\(S₂ = r₂(A) w₁(A) r₁(B) w₂(B)\)

**Theorem**

\(P(S₁)\) acyclic \(\iff S₁\) conflict serializable

(\(\Leftarrow\)) Assume \(S₁\) is conflict serializable

\(\Rightarrow \exists Sᵦ: Sᵦ, S₁\) conflict equivalent

\(\Rightarrow P(Sᵦ) = P(S₁)\)

\(\Rightarrow P(S₁)\) acyclic since \(P(Sᵦ)\) is acyclic
Theorem

\[ P(S_1) \text{ acyclic } \iff S_1 \text{ conflict serializable} \]

(\(\Rightarrow\)) Assume \(P(S_1)\) is acyclic

Transform \(S_1\) as follows:

1. Take \(T_1\) to be transaction with no incident arcs
2. Move all \(T_1\) actions to the front
   \[ S_1 = \ldots \ q_j(A) \ldots \ p_1(A) \ldots \]
3. we now have \(S_1 = < T_1 \text{ actions }> \ldots \text{rest} \ldots >\)
4. repeat above steps to serialize rest!

How to enforce serializable schedules?

Option 1: run system, recording \(P(S)\);
   check for \(P(S)\) cycles and declare if execution was good;
   or abort transactions as soon as they generate a cycle.

How to enforce serializable schedules?

Option 2: prevent \(P(S)\) cycles from occurring

Scheduler

DB
A locking protocol

Two new actions:
lock (exclusive): \( l_i(A) \)
unlock: \( u_i(A) \)

Rule #1: Well-formed transactions

\( T_i: \ldots l_i(A) \ldots p_i(A) \ldots u_i(A) \ldots \)

Rule #2 Legal scheduler

\( S = \ldots l_i(A) \ldots u_i(A) \ldots \)
no \( l_i(A) \)
Exercise:

- What schedules are legal?
- What transactions are well-formed?

S1 = l1(A)l1(B)r1(A)w1(B)u1(A)u1(B) 
   r1(B)w1(B)u1(B)
S2 = l1(A)r1(A)w1(B)u1(A)u1(B)
   l1(B)r2(B)w2(B)u2(B)u2(B)
S3 = l1(A)r1(A)u1(A)l1(B)w1(B)u1(B) 
   l1(B)r2(B)w2(B)u2(B)u2(B)

Schedule F

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A);Read(A)</td>
<td>l1(A);Read(A)</td>
</tr>
<tr>
<td>A←A+100;Write(A);u1(A)</td>
<td>A←A+2x;Write(A);u1(A)</td>
</tr>
<tr>
<td>l1(B);Read(B)</td>
<td>l1(B);Read(B)</td>
</tr>
<tr>
<td>B←B+2x;Write(B);u1(B)</td>
<td>B←Bx2;Write(B);u1(B)</td>
</tr>
</tbody>
</table>

Exercise:

- What schedules are legal?
- What transactions are well-formed?

S1 = l1(A)l1(B)r1(A)w1(B)u1(A)u1(B) 
   r1(B)w1(B)u1(B)
S2 = l1(A)r1(A)w1(B)u1(A)u1(B)
   l1(B)r2(B)w2(B)u2(B)u2(B)
S3 = l1(A)r1(A)u1(A)l1(B)w1(B)u1(B) 
   l1(B)r2(B)w2(B)u2(B)u2(B)
## Schedule F

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>l1(A); Read(A)</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>A = A + 100; Write(A); u1(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>l2(A); Read(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A = A * 2; Write(A); u2(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>l1(B); Read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B = B + 100; Write(B); u1(B)</td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>

### Rule #3  Two phase locking (2PL)

for transactions

\[ T_i = \ldots l_i(A) \ldots u_i(A) \ldots \]

- no unlocks
- no locks

#### # locks held by Ti

<table>
<thead>
<tr>
<th>Growing Phase</th>
<th>Shrinking Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time</td>
</tr>
</tbody>
</table>

34

35

36
<table>
<thead>
<tr>
<th>Schedule G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td><strong>T2</strong></td>
</tr>
<tr>
<td>l1(A); Read(A)</td>
<td>l1(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>l1(B); u1(A)</td>
<td>A ← Ax2; Write(A)</td>
</tr>
<tr>
<td>l2(A); Read(A)</td>
<td>Read(B); B ← B + 100</td>
</tr>
<tr>
<td>A ← Ax2; Write(A)</td>
<td>Write(B); u1(B)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td>u2(B); delayed</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

**Schedule G**

<table>
<thead>
<tr>
<th>Schedule G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td><strong>T2</strong></td>
</tr>
<tr>
<td>l1(A); Read(A)</td>
<td>l1(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>l1(B); u1(A)</td>
<td>A ← Ax2; Write(A)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td></td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

**Schedule G**

<table>
<thead>
<tr>
<th>Schedule G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td><strong>T2</strong></td>
</tr>
<tr>
<td>l1(A); Read(A)</td>
<td>l1(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>l1(B); u1(A)</td>
<td>A ← Ax2; Write(A)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td></td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

**Schedule G**

<table>
<thead>
<tr>
<th>Schedule G</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td><strong>T2</strong></td>
</tr>
<tr>
<td>l1(A); Read(A)</td>
<td>l1(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>l1(B); u1(A)</td>
<td>A ← Ax2; Write(A)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td></td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>
Schedule H  (T2 reversed)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A); Read(A)</td>
<td>l1(B); Read(B)</td>
</tr>
<tr>
<td>A=A+100; Write(A)</td>
<td>B=Bx2; Write(B)</td>
</tr>
</tbody>
</table>

• Assume deadlocked transactions are rolled back
  – They have no effect
  – They do not appear in schedule

E.g., Schedule H =

Next step:
Show that rules #1,2,3 ⇒ conflict-serializable schedules
Conflict rules for $l_i(A), u_i(A)$:

- $l_i(A), l_j(A)$ conflict
- $l_i(A), u_j(A)$ conflict

Note: no conflict $< u_i(A), u_j(A)>, < l_i(A), r_j(A)>,...$

Theorem  Rules #1,2,3 $\Rightarrow$ conflict

(2PL) serializable schedule

To help in proof:
Definition  Shrink$(T_i) = SH(T_i) =$ first unlock action of $T_i$

Lemma  $T_i \rightarrow T_j$ in $S \Rightarrow SH(T_i) <_S SH(T_j)$

Proof of lemma:
$T_i \rightarrow T_j$ means that
$S = ... p(A)... q(A) ... ; \ p,q$ conflict

By rules 1,2:
$S = ... p(A)... u_i(A)... l_j(A) ... q(A) ...$

By rule 3:  $SH(T_i)$ $\rightarrow$ $SH(T_j)$

So,  $SH(T_i) <_S SH(T_j)$
Theorem: Rules #1,2,3 $\Rightarrow$ conflict serializable

Proof:
(1) Assume P(S) has cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$
(2) By lemma: $SH(T_1) < SH(T_2) < \ldots < SH(T_1)$
(3) Impossible, so P(S) acyclic
(4) $\Rightarrow$ S is conflict serializable

- Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency....
  - Shared locks
  - Multiple granularity
  - Inserts, deletes and phantoms
  - Other types of C.C. mechanisms

Shared locks
So far:
$S = \ldots l_1(A) r_1(A) u_1(A) \ldots l_2(A) r_2(A) u_2(A) \ldots$

Instead:
$S = \ldots l_{s_1}(A) r_{s_1}(A) l_{s_2}(A) r_{s_2}(A) \ldots u_{s_1}(A) u_{s_2}(A)$
**Lock actions**

l-t(A): lock A in t mode (t is S or X)

u-t(A): unlock t mode (t is S or X)

**Shorthand:**

u(A): unlock whatever modes

Ti has locked A

---

**Rule #1**  Well formed transactions

\[ Ti = \ldots l-S_t(A) \ldots r_t(A) \ldots u_t(A) \ldots \]

\[ Ti = \ldots l-X_t(A) \ldots w_t(A) \ldots u_t(A) \ldots \]

---

- What about transactions that read and write same object?

**Option 1:** Request exclusive lock

\[ Ti = \ldots l-X_t(A) \ldots r_t(A) \ldots w_t(A) \ldots u(A) \ldots \]
Option 2: Upgrade

(E.g., need to read, but don’t know if will write...)

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

Rule #2 Legal scheduler

\[ S = \ldots l-S_i(A) \ldots u_i(A) \ldots \]
no \( l-X_j(A) \)

\[ S = \ldots l-X_i(A) \ldots u_i(A) \ldots \]
no \( l-X_j(A) \)
no \( l-S_i(A) \)

A way to summarize Rule #2

Compatibility matrix

<table>
<thead>
<tr>
<th>Comp</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

Proof: similar to X locks case

Detail:
l-ti(A), l-rj(A) do not conflict if comp(t,r)
l-t(A), u-rj(A) do not conflict if comp(t,r)

Theorem  Rules 1,2,3 ⇒ Conf.serializable
          for S/X locks        schedules

Lock types beyond S/X

Examples:
   (1) increment lock
   (2) update lock
Example (1): increment lock

- Atomic increment action: \( IN_i(A) \)
  \{Read(A); A ← A+k; Write(A)\}
- \( IN_i(A), IN_j(A) \) do not conflict!

\[
\begin{array}{l}
\text{Comp S X I} \\
S \quad T \\
X \quad F \\
I \quad F \\
\end{array}
\]

\[
\begin{array}{l}
\text{Comp S X I} \\
S \quad T \\
X \quad F \\
I \quad F \\
\end{array}
\]
Update locks

A common deadlock problem with upgrades:

T1
l-S1(A)
l-X1(A)

T2
l-S2(A)
l-X2(A)

--- Deadlock ---

Solution

If Ti wants to read A and knows it may later want to write A, it requests update lock (not shared)

<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock already held in</td>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$ symmetric table?
Note: object A may be locked in different modes at the same time...

\[ S_1 = ... \cdot S_1(A) ... \cdot S_2(A) ... \cdot U_3(A) ... \cdot S_4(A) ... \cdot U_4(A) ... \]

- To grant a lock in mode \( t \), mode \( t \) must be compatible with all currently held locks on object

How does locking work in practice?

- Every system is different

But here is one (simplified) way ...
Sample Locking System:

1. Don't trust transactions to request/release locks
2. Hold all locks until transaction commits

<table>
<thead>
<tr>
<th># locks</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Locking System:

- Ti
  - Read(A), Write(B)
- Scheduler, part I
  - l(A), Read(A), l(B), Write(B)...
- Scheduler, part II
  - Read(A), Write(B)

Lock table:

- A
- B
- C
- ...

If null, object is unlocked

Every possible object:

- Lock info for B
- Lock info for C
But use hash table:

If object not found in hash table, it is unlocked

Lock info for A - example

What are the objects we lock?
• 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

We can have it both ways!!
Ask any janitor to give you the solution...

Example

R1
  t1
  t2
  t3
  t4

T1(IS), T2(S)
Example

\[ \text{T}_1(\text{IS}), \text{T}_2(\text{IX}) \]

\[ \text{T}_1(\text{S}) \]

\[ \text{T}_2(\text{X}) \]

Multiple granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IX</td>
</tr>
<tr>
<td>S</td>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
</tr>
<tr>
<td>IX</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

Multiple granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IX</td>
</tr>
<tr>
<td>S</td>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
</tr>
<tr>
<td>IX</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>
Rules

1. Follow multiple granularity comp function
2. Lock root of tree first, any mode
3. Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
4. Node Q can be locked by Ti in X, SIX, IX only if parent(Q) locked by Ti in IX, SIX
5. Ti is two-phase
6. Ti can unlock node Q only if none of Q's children are locked by Ti
Insert + delete operations

$$
\begin{array}{c}
| A | \\
| \vdots | \\
| Z | \\
| \alpha |
\end{array}
$$

Insert

Modifications to locking rules:

1. Get exclusive lock on A before deleting A
2. At insert A operation by Ti, Ti is given exclusive lock on A

Still have a problem: Phantoms

Example: relation R (E#, name, ...)
constraint: E# is key
use tuple locking

<table>
<thead>
<tr>
<th>R</th>
<th>E#</th>
<th>Name</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>55</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>o2</td>
<td>75</td>
<td>Jones</td>
<td></td>
</tr>
</tbody>
</table>
T1: Insert <99,Gore,...> into R
T2: Insert <99,Bush,...> into R

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(o1)</td>
<td>S2(o1)</td>
</tr>
<tr>
<td>S1(o2)</td>
<td>S2(o2)</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
<tr>
<td>Insert o3[99,Gore,...]</td>
<td>Insert o4[99,Bush,...]</td>
</tr>
</tbody>
</table>

Solution

- Use multiple granularity tree
- Before insert of node Q,
  lock parent(Q) in X mode

Back to example

<table>
<thead>
<tr>
<th>T1: Insert&lt;99,Gore&gt;</th>
<th>T2: Insert&lt;99,Bush&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1(R)</td>
<td>X2(R)</td>
</tr>
<tr>
<td>Check constraint</td>
<td>Check constraint</td>
</tr>
<tr>
<td>Insert&lt;99,Gore&gt;</td>
<td>Oops! e# = 99 already in R!</td>
</tr>
<tr>
<td>U(R)</td>
<td></td>
</tr>
</tbody>
</table>
Instead of using R, can use index on R:

Example:

\[ \begin{array}{c}
\text{Index} \\
0 < E# \leq 100
\end{array} \quad \begin{array}{c}
\text{Index} \\
100 < E# \leq 200
\end{array} \quad \begin{array}{c}
E# = 2 \\
E# = 5 \\
E# = 107 \\
E# = 109 \\
\ldots
\end{array} \]

• This approach can be generalized to multiple indexes...

Next:

• Tree-based concurrency control
• Validation concurrency control
Example

- all objects accessed through root, following pointers

(can we release A lock if we no longer need A?)

Idea: traverse like "Monkey Bars"

Why does this work?

- Assume all Ti start at root; exclusive lock
- Ti \rightarrow Tj \Rightarrow Ti locks root before Tj
- Actually works if we don’t always start at root
Rules: tree protocol (exclusive locks)

1. First lock by Ti may be on any item
2. After that, item Q can be locked by Ti only if parent(Q) locked by Ti
3. Items may be unlocked at any time
4. After Ti unlocks Q, it cannot relock Q

Tree-like protocols are used typically for B-tree concurrency control

E.g., during insert, do not release parent lock, until you are certain child does not have to split