CS 245: Database System Principles

Notes 09: Concurrency Control

Hector Garcia-Molina
(Some modifications by Chris Olston

ACID Properties

- Atomicity
  - Actions are never left partially executed
- Consistency
  - Actions leave the DB in a consistent state
- Isolation
  - Actions are not affected by other concurrent actions
- Durability
  - Effects of completed actions are resilient against system failures

Chapter 9
Concurrency Control

T1, T2, ..., Tn

Transactions not affected by other concurrent transactions

Example:

T1: Read(A)  T2: Read(A)
A ← A+100    A ← A×2
Write(A)      Write(A)
Read(B)       Read(B)
B ← B+100    B ← B×2
Write(B)      Write(B)
Constraint: A=B

Schedule A

T1       T2
Read(A); A ← A+100          25
Write(A);                   25
Read(B); B ← B+100;         125
Write(B);                   125
Read(A); A ← A×2;           250
Write(A);                   250
Read(B); B ← B×2;           250
Write(B);                   250
**Schedule B**

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

**Schedule C**

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

**Schedule D**

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

**Schedule E**

<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>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A+1;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+1;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
</tbody>
</table>

- Want schedules that are “good”, regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

Example:
\[ Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

Example:
\[ Sc' = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]
However, for S_d:
\[ S_d = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

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

\[ \text{T}_1 \rightarrow \text{T}_2 \quad \Rightarrow \quad \text{S}_d \text{ cannot be rearranged into a serial schedule} \]
\[ \text{T}_1 \rightarrow \text{T}_2 \quad \Rightarrow \quad \text{S}_d \text{ is not "equivalent" to any serial schedule} \]
\[ \text{T}_1 \rightarrow \text{T}_2 \quad \Rightarrow \quad \text{S}_d \text{ is "bad"} \]

Returning to S_c:
\[ S_c = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\[ \text{T}_1 \rightarrow \text{T}_2 \quad \Rightarrow \quad \text{S}_c \text{ is "equivalent" to a serial schedule (in this case T}_1, \text{T}_2) \]

Concepts

Transaction: sequence of r_i(x), w_i(x) actions
Conflicting actions: \[ r_1(A) \quad W_2(A) \quad W_1(A) \]
\[ W_2(A) \quad r_1(A) \quad W_2(A) \]

Schedule: represents chronological order in which actions are executed
Serial schedule: no interleaving of actions or transactions

What about concurrent actions?

So net effect is either
- \[ S = \ldots r_1(x) \ldots w_2(y) \ldots \text{ or} \]
- \[ S = \ldots w_2(y) \ldots r_1(x) \ldots \]
What about conflicting, concurrent actions on same object?

\[
\begin{array}{c}\text{start } r_1(A) \quad \text{end } r_1(A) \\
\text{start } w_2(A) \quad \text{end } w_2(A) \quad \text{time}
\end{array}
\]

- 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?

**Exercise 2:**

- What is \( P(S) \) for \( S = r_1(A) \) \( w_1(B) \) \( r_1(C) \) \( w_2(C) \) \( w_2(A) \) \( w_3(A) \) \( r_4(A) \) \( w_4(D) \)
- \( S \) serial \( \Rightarrow \) \( P(S) \) acyclic
**Lemma**

S₁, S₂ conflict equivalent ⇒ P(S₁) = P(S₂)

**Proof:**
Assume P(S₁) ≠ P(S₂)
⇒ ∃ Ti: Ti → Tj in S₁ and not in S₂
⇒ S₁ = ...p(A)... q(A)... p. q_i
    S₂ = ...q_i(A)...p(A)... conflict
⇒ S₁, S₂ not conflict equivalent

**Note:** P(S₁) ≠ P(S₂) ≠ S₁, S₂ conflict equivalent

**Counter example:**

S₁ = w₁(A) r₁(A) w₂(B) r₁(B)
S₂ = r₁(A) w₁(A) r₁(B) w₂(B)

**Theorem**

P(S₁) acyclic ⇐⇒ S₁ conflict serializable

(⇐) Assume S₁ is conflict serializable
⇒ ∃ Sᵢ: Sᵢ, Sᵢ conflict equivalent
⇒ P(Sᵢ) = P(S₁)
⇒ P(S₁) acyclic since P(Sᵢ) is acyclic

(⇒) Assume P(S₁) is acyclic
Transform S₁ as follows:
(1) Take T₁ to be transaction with no incident actions
(2) Move all T₁ actions to the front
    S₁ = ...... q₁(A) ...... p₁(A) ......
(3) we now have S₁ = < T₁ actions > < ... rest ... >
(4) repeat above steps to serialize rest!

**How to enforce serializable schedules?**

**Option 1:** run system, recording P(S);
at end of day, check for P(S) cycles and declare if execution was good

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

How to enforce serializable schedules?
A locking protocol

Two new actions:
lock (exclusive): \text{li} (A)
unlock: \text{ui} (A)

\begin{center}
\begin{tikzpicture}
\node (scheduler) at (0,0) {scheduler};
\node (lock) at (1,0) {lock table};
\draw [->] (scheduler) -- (lock);
\end{tikzpicture}
\end{center}

\textbf{Rule #1:} Well-formed transactions

\text{Ti}: \ldots \text{li}(A) \ldots \text{pi}(A) \ldots \text{ui}(A) \ldots

\textbf{Rule #2} Legal scheduler

\[ S = \ldots \ldots \text{li}(A) \ldots \ldots \text{ui}(A) \ldots \ldots \]
\[ \text{no li}(A) \]

\textbf{Exercise:}

\begin{itemize}
  \item What schedules are legal?
  \item What transactions are well-formed?
\end{itemize}

\text{S1} = \text{li}(A)\text{li}(B)\text{ri}(A)\text{wi}(B)\text{bi}(B)\text{ui}(A)\text{ui}(B)
\text{ri}(B)\text{wi}(B)\text{bi}(B)\text{ui}(B)\text{ui}(B)
\text{S2} = \text{li}(A)\text{ri}(A)\text{ri}(B)\text{ui}(A)\text{ui}(B)
\text{ri}(B)\text{wi}(B)\text{bi}(B)\text{ui}(B)\text{ui}(B)
\text{S3} = \text{li}(A)\text{ri}(A)\text{ui}(A)\text{li}(B)\text{wi}(B)\text{ui}(B)
\text{bi}(B)\text{wi}(B)\text{ui}(B)\text{ui}(B)\text{ui}(B)

\textbf{Schedule F}

\begin{center}
\begin{tabular}{c|c|c}
  \text{T1} & \text{T2} \\
  \text{li}(A); \text{Read}(A) & \text{b}(A); \text{Read}(A) \\
  \text{A} \rightarrow \text{A} + 100; \text{Write}(A); \text{u}(A) & \text{A} \rightarrow \text{Ax} + 2; \text{Write}(A); \text{u}(A) \\
  \text{b}(B); \text{Read}(B) & \text{b}(B); \text{Read}(B) \\
  \text{B} \rightarrow \text{B} + 100; \text{Write}(B); \text{u}(B) & \text{B} \rightarrow \text{Bx} + 2; \text{Write}(B); \text{u}(B) \\
  \text{li}(B); \text{Read}(B) & \text{li}(B); \text{Read}(B) \\
  \text{b}(B) & \text{b}(B) \\
\end{tabular}
\end{center}
Schedule F

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

Rule #3 Two phase locking (2PL)

for transactions

Ti = .......... l(A) ........... u(A) ........
no locks no locks

Schedule G

![Graph showing the number of locks held by each transaction over time]

Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(A); Read(A)</td>
<td></td>
<td>l(A); Read(A)</td>
<td></td>
</tr>
<tr>
<td>A++; A+100; Write(A)</td>
<td></td>
<td>A++; A+100; Write(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>l(B); u(A)</td>
<td></td>
<td>b(A); Read(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B); B++; B+100</td>
<td></td>
<td>A++; A+2; Write(A); l(B)</td>
<td></td>
</tr>
<tr>
<td>Write(B); u(B)</td>
<td></td>
<td>Write(B); u(B)</td>
<td></td>
</tr>
<tr>
<td>t(A); Read(A)</td>
<td></td>
<td>t(A); Read(A)</td>
<td>delayed</td>
</tr>
<tr>
<td>A++; A+2; Write(A); l(B)</td>
<td></td>
<td>A++; A+2; Write(A); l(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B); B++; B+100</td>
<td></td>
<td>Read(B); B++; B+100</td>
<td></td>
</tr>
<tr>
<td>Write(B); u(B)</td>
<td></td>
<td>Write(B); u(B)</td>
<td></td>
</tr>
<tr>
<td>l(B); u(A); Read(B)</td>
<td></td>
<td>l(B); u(A); Read(B)</td>
<td></td>
</tr>
<tr>
<td>B++; B+2; Write(B); u(B)</td>
<td></td>
<td>B++; B+2; Write(B); u(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>(l(A); \text{Read}(A))</td>
<td>(l(B); \text{Read}(B))</td>
</tr>
<tr>
<td>(A \leftarrow A + 100; \text{Write}(A))</td>
<td>(B \leftarrow B + 2; \text{Write}(B))</td>
</tr>
<tr>
<td>(l(B))</td>
<td>(l(A))</td>
</tr>
</tbody>
</table>

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

E.g., Schedule H =

This space intentionally left blank!

### Next step:

Show that rules #1,2,3 ⇒ conflict-serializable schedules

### Conflict rules for \(l(A), u(A)\):

- \(l(A), l(A)\) conflict
- \(l(A), u(A)\) conflict

Note: no conflict < \(u(A), u(A)\>, < l(A), r(A)>,. . .

### Lemma

\(Tᵢ \rightarrow Tⱼ\) in \(S\) ⇒ \(SH(Tᵢ) <ₛ SH(Tⱼ)\)

**Proof of lemma:**

\(Tᵢ \rightarrow Tⱼ\) means that

\(S = ... \ p(A) ... q(A) ... ; \ p, q \text{ conflict}\)

By rules 1,2:

\(S = ... \ p(A) ... u(A) ... l(A) ... q(A) ...\)

By rule 3:

\(SH(Tᵢ) <ₛ SH(Tⱼ)\)

So, \(SH(Tᵢ) <ₛ SH(Tⱼ)\)
Theorem: Rules #1, 2, 3 ⇒ conflict (2PL) serializable schedule

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) cyclic
(4) ⇒ 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:
\[ S = \ldots l_s(A) r_t(A) u_s(A) \ldots l_t(A) r_t(A) u_t(A) \ldots \]

Do not conflict

Instead:
\[ S = \ldots l_{s}(A) r_{t}(A) l_{s}(A) r_{t}(A) \ldots u_{s}(A) u_{s}(A) \]

Lock actions
\( l\cdot t(A) \): lock A in t mode (t is S or X)
\( u\cdot t(A) \): unlock t mode (t is S or X)

Shorthand:
\( u(A) \): unlock whatever modes

\( Ti \) has locked A

Rule #1: Well formed transactions
\[ Ti = \ldots l\cdot S_i(A) \ldots r_t(A) \ldots u_t(A) \ldots \]
\[ Ti = \ldots l\cdot X_i(A) \ldots w_t(A) \ldots u_t(A) \ldots \]

What about transactions that read and write same object?

Option 1: Request exclusive lock
\[ Ti = \ldots l\cdot X_i(A) \ldots r_t(A) \ldots w_t(A) \ldots u(A) \ldots \]
What about transactions that read and write same object?

Option 2: Upgrade

(I.e., need to read, but don’t know if will write...)

$T_i = \cdots \text{l-} S_i(A) \cdots \text{r}_i(A) \cdots \text{l-X}_i(A) \cdots \text{w}_i(A) \cdots \text{u}(A) \cdots$

Think of:
- Get 2nd lock on A, or
- Drop $S_i$ get $X$ lock

Rule #2 Legal scheduler

$S = \cdots \text{l-} S_i(A) \cdots \text{u}(A) \cdots$

no $\text{l-X}_i(A)$

$S = \cdots \text{l-X}(A) \cdots \text{u}(A) \cdots$

no $\text{l-X}(A)$

no $\text{l-S}_i(A)$

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., $S \rightarrow \{S, X\}$) then no change!

(II) If upgrade releases read (shared)

lock (e.g., $S \rightarrow X$)

- can be allowed in growing phase

Theorem Rules 1, 2, 3 $\Rightarrow$ Conf. serializable for $S/X$ locks schedules

Proof: similar to $X$ locks case

Detail:

$\text{l-}t(A), \text{l-r}(A)$ do not conflict if $\text{comp}(t,r)$

$\text{l-t}(A), \text{u-}r(A)$ do not conflict if $\text{comp}(t,r)$

Lock types beyond $S/X$

Examples:

(1) increment lock

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

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

\[
\begin{align*}
\text{IN}_{\text{A}}(A &= 5
\quad +2
\quad +10
\quad \text{IN}_{\text{A}}(A
\quad A &= 7
\quad A &= 17
\quad A &= 19
\end{align*}
\]

Comp

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Update locks

A common deadlock problem with upgrades:

\[
\begin{align*}
\text{T1} &\quad \text{T2} \\
\text{I-Si(A) &} & \quad \text{I-Sz(A)} \\
\text{I-Xi(A) &} & \quad \text{I-Xz(A)} \\
\text{--- Deadlock ---} \\
\end{align*}
\]

Solution

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

Comp

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New request

Lock already held in

Comp

<table>
<thead>
<tr>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How does locking work in practice?

- Every system is different
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Note: object A may be locked in different modes at the same time...

\[ S_1 = \ldots S_r(A) \ldots s_2(A) \ldots U_r(A) \ldots U_2(A) \ldots \]

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

Sample Locking System:

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

Lock table

Conceptually

If null, object is unlocked
But use hash table:

![Diagram of hash table]

If object not found in hash table, it is unlocked

Lock info for A - example

![Diagram of lock info for A]

What are the objects we lock?

![Diagram of objects]

- 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!!

![Diagram of application structure]

Example

![Diagram of example structure]
Example

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

Multiple granularity

<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IS, S</td>
</tr>
<tr>
<td>IX</td>
<td>IS, S, IX, X, SIX</td>
</tr>
<tr>
<td>S</td>
<td>S, IS not necessary</td>
</tr>
<tr>
<td>SIX</td>
<td>X, IX, [SIX]</td>
</tr>
<tr>
<td>X</td>
<td>none</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
**Exercise:**
- Can T2 access object f.2 in X mode?
  - What locks will T2 get?

![Diagram](Diagram1.png)

**Exercise:**
- Can T2 access object f.2 in X mode?
  - What locks will T2 get?

![Diagram](Diagram2.png)

**Exercise:**
- Can T2 access object f.1 in X mode?
  - What locks will T2 get?

![Diagram](Diagram3.png)

**Exercise:**
- Can T2 access object f.2 in S mode?
  - What locks will T2 get?

![Diagram](Diagram4.png)

**Exercise:**
- Can T2 access object f.2 in X mode?
  - What locks will T2 get?

![Diagram](Diagram5.png)

**Insert + delete operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Insert</td>
</tr>
<tr>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>α</td>
<td></td>
</tr>
</tbody>
</table>

---
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>E#</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>Smith</td>
</tr>
<tr>
<td>75</td>
<td>Jones</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,Ullman,...&gt; into R</th>
<th>T2: Insert &lt;99,Widom,...&gt; into R</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>S1(o1)</td>
<td>S2(o1)</td>
</tr>
<tr>
<td>S1(∞)</td>
<td>S2(∞)</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
<tr>
<td>Insert o1[99,Ullman,...]</td>
<td>Insert o2[99,Widom,...]</td>
</tr>
</tbody>
</table>

Instead of using R, can use index on R:

Example:

Index
O<E#≤100

E#=2 E#=5 ...

E#=107 E#=109 ...

Index
100<E#≤200
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”
- Idea: don’t let another transaction “slip past”

Why does this work?
- Assume all Ti start at root; exclusive lock
- Ti → Tj ⇒ 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
Validation
Transactions have 3 phases:
(1) Read
   - all DB values read
   - writes to temporary storage
   - no locking
(2) Validate
   - check if schedule so far is serializable
(3) Write
   - if validate ok, write to DB

Key idea
- Make validation atomic
- If $T_1, T_2, T_3, \ldots$ is validation order, then resulting schedule will be conflict equivalent to $S_\circ = T_1 T_2 T_3 \ldots$

To implement validation, system keeps two sets:
- $\text{FIN} =$ transactions that have finished phase 3 (and are all done)
- $\text{VAL} =$ transactions that have successfully finished phase 2 (validation)

Example of what validation must prevent:

$$\begin{align*}
\text{RS}(T_2) &= \{B\} \\
\text{WS}(T_2) &= \{B, D\} \\
\text{RS}(T_3) &= \{A, B\} \neq \emptyset \\
\text{WS}(T_3) &= \{C\}
\end{align*}$$

Another thing validation must prevent:

$$\begin{align*}
\text{RS}(T_2) &= \{A\} \\
\text{RS}(T_3) &= \{A, B\} \\
\text{WS}(T_2) &= \{D, E\} \\
\text{WS}(T_3) &= \{C, D\}
\end{align*}$$

Example of what validation must prevent:

$\begin{align*}
\text{RS}(T_2) &= \{B\} \\
\text{WS}(T_2) &= \{B, D\} \\
\text{RS}(T_3) &= \{A, B\} \neq \emptyset \\
\text{WS}(T_3) &= \{C\}
\end{align*}$

Another thing validation must prevent:

$\begin{align*}
\text{RS}(T_2) &= \{A\} \\
\text{RS}(T_3) &= \{A, B\} \\
\text{WS}(T_2) &= \{D, E\} \\
\text{WS}(T_3) &= \{C, D\}
\end{align*}$
Another thing validation must prevent:

- RS(T2) = {A} and RS(T3) = {A, B}
- WS(T2) = {D, E} and WS(T3) = {C, D}

Validation rules for Tj:

1. When Tj starts phase 1:
   - If check(Tj) then
     - Enter phase
     - FIN ← FIN U {Tj}

2. At Tj validation:
   - If check(Tj) then
     - Enter phase
     - FIN ← FIN U {Tj}

Improving Check(Tj):

For Ti ∈ VAL - IGNORE (Tj) DO

- IF [ WS(Ti) ∩ RS(Tj) ≠ ∅ OR Ti ∉ FIN ] THEN RETURN false;
- RETURN true;

Is this check too restrictive?

Exercise:

- U: RS(U) = {B} and WS(U) = {D}
- W: RS(W) = {A, D} and WS(W) = {A, C}
- T: RS(T) = {A, B} and WS(T) = {A, C}
- V: RS(V) = {B} and WS(V) = {D, E}

Validation (also called optimistic concurrency control) is useful in some cases:
- Conflicts rare
- System resources plentiful
- Have real time constraints
Summary

Have studied C.C. mechanisms used in practice
- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation