Chapter 9  Concurrency Control

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
### Serial Schedule A (“good” by definition)

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

### Serial Schedule B (equally “good”)

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

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

<table>
<thead>
<tr>
<th></th>
<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></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
<td></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>Read(A); A ← A×2;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td>Read(B); B ← B×1;</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td>Write(B);</td>
<td></td>
<td></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>Read(A); A ← A×1;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×1;</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td>Read(B); B ← B×1;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The accident being the particular semantics of T2’

- 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:

\[ SC = \tau_1(A)w_1(A)r_2(A)w_2(A)\tau_1(B)w_1(B)r_2(B)w_2(B) \]
Example:

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

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

\[ T_1 \quad T_2 \]

However, for Schedule D:

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

• as a matter of fact,
  \[ T_2 \text{ must precede } T_1 \]
  in any equivalent schedule,
  i.e.,  \[ T_2 \rightarrow T_1 \]

• And vice versa

•  \[ T_2 \rightarrow T_1 \]

• Also,  \[ T_1 \rightarrow T_2 \]

\[ \text{T}_1 \rightarrow \text{T}_2 \quad \implies \quad \text{SD cannot be rearranged}
\quad \text{into a serial schedule} \]

\[ \implies \quad \text{SD is not “equivalent” to}
\quad \text{any serial schedule} \]

\[ \implies \quad \text{SD is “bad”} \]
Returning to Sc

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

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

Concepts

**Transaction:** sequence of \( r(x), w(x) \) actions

**Conflicting actions:**

\[ r_1(A) \quad w_2(A) \quad w_1(A) \quad w_2(A) \quad r_1(A) \quad w_2(A) \]

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

**Serial schedule:** no interleaving of actions or transactions

What about concurrent actions?

\[ Ti \text{ issues } read(x,t) \quad System \text{ issues } input(x) \quad Input(X) \text{ completes} \quad t \leftarrow x \quad time \]

\[ T2 \text{ issues } write(B,S) \quad System \text{ issues } input(B) \quad \quad \quad B \leftarrow S \quad System \text{ issues } output(B) \quad output(B) \text{ completes} \]
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?

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

• 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 $p_i, q_i$ 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

$S_1, S_2$ conflict equivalent $\Rightarrow P(S_1) = P(S_2)$

Proof:

Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists T_i, T_j: T_i \rightarrow T_j$ in $S_1$ and not in $S_2$

$\Rightarrow S_1 = \ldots p(A) \ldots q(A) \ldots \begin{cases} p_i, q_i \text{ conflict} \\ S_2 = \ldots q(A) \ldots p(A) \ldots \end{cases}$

$\Rightarrow S_1, S_2$ not conflict equivalent

Note: $P(S_1) = P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Counter example:

$S_1 = w_1(A) \ r_2(A) \ w_2(B) \ r_1(B)$

$S_2 = r_2(A) \ w_1(A) \ r_1(B) \ w_2(B)$

Theorem

$P(S_1)$ acyclic $\iff S_1$ conflict serializable

($\Leftarrow$) Assume $S_1$ is conflict serializable

$\Rightarrow \exists S_x: S_x, S_1$ conflict equivalent

$\Rightarrow P(S_x) = P(S_1)$

$\Rightarrow P(S_1)$ acyclic since $P(S_x)$ 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

\[ S_1 = \ldots q_i(A) \ldots p_i(A) \ldots \]

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

---

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

---

Scheduler

DB
A locking protocol

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

Rule #1: Well-formed transactions
\( Ti: \ldots li(A) \ldots pi(A) \ldots ui(A) \ldots \)

Rule #2  Legal scheduler
\( S = \ldots li(A) \ldots ui(A) \ldots \)
\( \text{no } li(A) \)
Exercise:

- What schedules are legal?
  What transactions are well-formed?
  S1 = l_1(\text{A})l_1(\text{B})r_1(\text{A})w_1(\text{B})l_2(\text{B})u_1(\text{A})u_2(\text{B})
  r_2(\text{B})w_2(\text{B})u_2(\text{B})l_3(\text{B})r_3(\text{B})u_3(\text{B})
  S2 = l_1(\text{A})r_1(\text{A})w_1(\text{B})u_1(\text{A})u_1(\text{B})
  l_2(\text{B})r_2(\text{B})w_2(\text{B})l_3(\text{B})r_3(\text{B})u_3(\text{B})
  S3 = l_1(\text{A})r_1(\text{A})w_1(\text{B})u_1(\text{A})u_1(\text{B})
  l_2(\text{B})r_2(\text{B})w_2(\text{B})l_3(\text{B})r_3(\text{B})u_3(\text{B})

Schedule F

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_1(\text{A});\text{Read(A)}</td>
<td>l_1(\text{A});\text{Read(A)}</td>
</tr>
<tr>
<td>A = A+100;\text{Write(A)};u_1(\text{A})</td>
<td>A = A+x2;\text{Write(A)};u_1(\text{A})</td>
</tr>
<tr>
<td>l_1(\text{B});\text{Read(B)}</td>
<td>l_1(\text{B});\text{Read(B)}</td>
</tr>
<tr>
<td>B = B+100;\text{Write(B)};u_1(\text{B})</td>
<td>B = B+x2;\text{Write(B)};u_1(\text{B})</td>
</tr>
</tbody>
</table>
### Schedule F

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1</td>
<td>(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>125</td>
<td></td>
</tr>
<tr>
<td>l2</td>
<td>(A); Read(A)</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A ← Ax2; Write(A); u2(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(B); Read(B)</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B ← Bx2; Write(B); u2(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l3</td>
<td>(B); Read(B)</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B ← B + 100; Write(B); u3(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></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

---

![Graph showing # locks held by Ti over time with phases indicated]

- Growing Phase
- Shrinking Phase
### Schedule H  \hspace{1em} (T_2 \text{ reversed})

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_1(A); Read(A)</td>
<td>l_2(B); Read(B)</td>
</tr>
<tr>
<td>A := A + 100; Write(A)</td>
<td>B := B * 2; Write(B)</td>
</tr>
<tr>
<td></td>
<td></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 $\Rightarrow$ conflict-serializable schedules
Conflict rules for $l(A), u(A)$:

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

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

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

(2PL) serializable schedule

To help in proof:

**Definition** $\text{Shrink}(T_i) = SH(T_i) =$ first unlock action of $T_i$

**Lemma** $T_i \rightarrow T_j$ in $S$ $\Rightarrow$ $SH(T_i) \preceq SH(T_j)$

**Proof of lemma:**

$T_i \rightarrow T_j$ means that $S = ... p(A) ... q_i(A) ...; p,q$ conflict

By rules 1,2:

$S = ... p(A) ... u_i(A) ... l_i(A) ... q_i(A) ...$

By rule 3: $\frac{SH(T_i)}{SH(T_j)}$

So, $SH(T_i) \preceq SH(T_j)$
**Theorem** Rules #1,2,3 ⇒ conflict serializable schedule

**Proof:**
1. Assume P(S) has cycle $T_1 \rightarrow T_2 \rightarrow ... T_n \rightarrow T_1$
2. By lemma: $SH(T_1) < SH(T_2) < ... < SH(T_1)$
3. Impossible, so P(S) acyclic
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:
$S = ... l_1(A) r_1(A) u_1(A) ... l_2(A) r_2(A) u_2(A) ...$

Do not conflict

Instead:
$S = ... ls_1(A) r_1(A) ls_2(A) r_2(A) ... us_1(A) us_2(A)$
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 = ... l-Si(A) ... ri(A) ... u1(A) ...
Ti = ... l-Xi(A) ... wi(A) ... u1(A) ...

• What about transactions that read and write same object?

Option 1: Request exclusive lock
Ti = ... l-Xi(A) ... ri(A) ... wi(A) ... u(A) ...
• What about transactions that read and write same object?

**Option 2: Upgrade**
(E.g., need to read, but don’t know if will write…)

\[ T_i = \ldots l-S_i(A) \ldots r_i(A) \ldots l-X_i(A) \ldots w_i(A) \ldots u(A) \ldots \]

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

---

**Rule #2  Legal scheduler**

\[ S = \ldots l-S_i(A) \ldots \ldots u(A) \ldots \]

no \( l-X_i(A) \)

\[ S = \ldots l-X_i(A) \ldots \ldots u(A) \ldots \]

no \( l-X_i(A) \)

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., $S \rightarrow \{S, X\}$) 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

Lock types beyond S/X

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

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

\[ \begin{align*}
\text{A} = 5 & \quad \text{IN}(A) \quad \text{A} = 7 \\
\quad +10 & \\
\text{IN}(A) & \quad \text{A} = 15 \\
\quad +2 & \\
\text{IN}(A) & \quad \text{A} = 17
\end{align*} \]

Comp

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

A common deadlock problem with upgrades:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l-S_i(A)</td>
<td>l-S_i(A)</td>
<td></td>
</tr>
<tr>
<td>l-X_i(A)</td>
<td>l-X_i(A)</td>
<td></td>
</tr>
</tbody>
</table>

--- Deadlock ---

Solution

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

New request

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

-> symmetric table?
Note: object A may be locked in different modes at the same time...

\[
S_1 = \ldots I-S_1(A) \ldots I-S_2(A) \ldots I-U_3(A) \ldots I-S_t(A) \ldots I-U_t(A) \ldots
\]

• 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

Lock table Conceptually

Every possible object

If null, object is unlocked

Lock info for B

Lock info for C

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Scheduler, part I

Scheduler, part II

DB

Ti

Read(A), Write(B)
But use hash table:

![Diagram]

If object not found in hash table, it is unlocked

Lock info for A - example

![Diagram]

What are the objects we lock?

![Diagram]
• 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
### Multiple granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Holder</td>
<td>IS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parent | Child can be locked in
--- | ---
IS | IS, S
IX | IS, S, IX, X, SIX
S | [S, IS] not necessary
SIX | X, IX, [SIX]
X | none

### 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

<table>
<thead>
<tr>
<th>A</th>
<th>:</th>
<th>Z</th>
<th>α</th>
</tr>
</thead>
</table>

--- 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>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>55</td>
<td>Smith</td>
</tr>
<tr>
<td>o2</td>
<td>75</td>
<td>Jones</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

T1: Insert<99,Gore> T2: Insert<99,Bush>

X1(R) X2(R) delayed

Check constraint Insert<99,Gore> U(R)

Check constraint Oops! e# = 99 already in R!
Instead of using R, can use index on R:

Example:

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

Next:

- Tree-based concurrency control
- Validation concurrency control
Example
- all objects accessed through root, following pointers

![Diagram of object access]

- can we release A lock if we no longer need A??

Idea: traverse like "Monkey Bars"

![Diagram of traversal]

Why does this work?
- Assume all $T_i$ start at root; exclusive lock
- $T_i \rightarrow T_j \Rightarrow T_i$ locks root before $T_j$

![Diagram of traversal]

- 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