Chapter 18: Concurrency Control

Example:

T1: Read(A) T2: Read(A)
A ← A+100
Write(A)
Read(B)
B ← B+100
Write(B)

Constraint: A=B

Schedule A

T1 T2
Read(A); A ← A+100; 125
Write(A);
Read(B); B ← B+100;
Write(B);

A   B
25  25
125
250
250
Schedule B

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A×2; Write(A); Read(B); B ← B×2; Write(B);</td>
<td></td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Read(A); A ← A+100 Write(A); Read(B); B ← B+100; Write(B);</td>
<td>150</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>150</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>T1</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100 Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2; Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100; Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2; Write(B);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</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>T1</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100 Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2; Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×2; Write(B);</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100; Write(B);</td>
<td>150</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>150</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>T1</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100 Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×1; Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B×1; Write(B);</td>
<td></td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100; Write(B);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>125</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=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

However, for Sd:
Sd=r1(A)w1(A)r2(A)w2(A)r2(B)w2(B)r1(B)w1(B)

• as a matter of fact,
  T2 must precede T1
  in any equivalent schedule,
  i.e., T2 → T1

**Example:**
Sc=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

Sc’=r1(A)w1(A) r1(B)w1(B)r2(A)w2(A)r2(B)w2(B)

T1 T2

• T2 → T1
• Also, T1 → T2

T1 T2

Sd cannot be rearranged into a serial schedule

Sd is not “equivalent” to any serial schedule

Sd is “bad”
Returning to Sc

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

\[ T_1 \to T_2 \]

\[ T_1 \to T_2 \]

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

Concepts

Transaction: sequence of \(\text{ri}(x), \text{wi}(x)\) actions

Conflicting actions:

\[ \text{r}_1(A) \quad \text{w}_2(A) \quad \text{w}_1(A) \]
\[ \text{w}_2(A) \quad \text{r}_1(A) \quad \text{w}_2(A) \]

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

What About Concurrent Actions?

<table>
<thead>
<tr>
<th>Ti issues</th>
<th>System issues</th>
<th>Input(X)</th>
<th>t (\leftarrow x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x,t)</td>
<td>input(x)</td>
<td>completes</td>
<td>time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T2 issues</th>
<th>input(B)</th>
<th>System issues</th>
<th>output(B)</th>
<th>B (\leftarrow S)</th>
<th>output(B) completes</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(B,S)</td>
<td>completes</td>
<td>System issues</td>
<td>output(B)</td>
<td>B (\leftarrow S)</td>
<td>completes</td>
</tr>
</tbody>
</table>

So net effect is either

- \(S=\ldots \text{r}_1(x) \ldots \text{w}_2(B) \ldots\) or
- \(S=\ldots \text{w}_2(B) \ldots \text{r}_1(x) \ldots\)
What about conflicting, concurrent actions on same object?

\[ \begin{align*}
\text{start } r1(A) & \quad \text{end } r1(A) \\
\text{start } w2(A) & \quad \text{end } w2(A)
\end{align*} \]

\[ \text{time} \]

- Assume equivalent to either \( r1(A) \) \( w2(A) \)
  or \( w2(A) \) \( r1(A) \)
- \implies\ low level synchronization mechanism
- Assumption called “atomic actions”

**Definition**

S1, S2 are conflict equivalent schedules if S1 can be transformed into S2 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: \( Ti \rightarrow Tj \) whenever

- \( pi(A) \), \( qj(A) \) are actions in S
- \( pi(A) \) \( \leq_S \) \( qj(A) \)
- at least one of \( pi \), \( qj \) is a write
Exercise:

• What is P(S) for
  S = w3(A) w2(C) r1(A) w1(B) r1(C) w2(A) r4(A) w4(D)

• Is S serializable?

Another Exercise:

• What is P(S) for
  S = w1(A) r2(A) r3(A) w4(A) ?

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 = ...pi(A)... qj(A)...
   S2 = ...qj(A)...pi(A)...
   conflict

⇒ S1, S2 not conflict equivalent

Note: P(S1)=P(S2) \not\Rightarrow 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**

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

\( (\iff) \) Assume \( S_1 \) is conflict serializable
\( \Rightarrow \exists Ss: Ss, S_1 \text{ conflict equivalent} \)
\( \Rightarrow P(Ss) = P(S_1) \)
\( \Rightarrow P(S_1) \text{ acyclic since } P(Ss) \text{ is acyclic} \)

\( (\Rightarrow) \) 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
3. we now have \( S_1 = T_1 \text{ actions } \ldotsq_j(A) \ldots p_1(A) \ldots \)
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
A Locking Protocol

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

Rule #1: Well-Formed Transactions

\[ \text{Ti: } \text{... li}(A) \text{... pi}(A) \text{... ui}(A) \text{...} \]

Rule #2: Legal Scheduler

\[ S = \ldots \text{ li}(A) \ldots \text{ ui}(A) \ldots \]

Exercise:

- What schedules are legal?
  What transactions are well-formed?
  \[ S1 = l1(A)r1(B)w1(B)r2(B)u1(A)u1(B) \]
  \[ r2(B)w2(B)u2(B)r3(B)u3(B) \]
  \[ S2 = l1(A)r1(A)w1(B)u1(A)u1(B) \]
  \[ r2(B)w2(B)u2(B)r3(B)u3(B) \]
  \[ S3 = l1(A)r1(A)w1(B)u1(B)u1(A) \]
  \[ r2(B)w2(B)u2(B)r3(B)u3(B) \]
**Schedule F**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A); Read(A)</td>
<td>l2(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A); u1(A)</td>
<td>A ← A; x2; Write(A); u2(A)</td>
</tr>
<tr>
<td>l1(B); Read(B)</td>
<td>l2(B); Read(B)</td>
</tr>
<tr>
<td>B ← B + 100; Write(B); u1(B)</td>
<td>B ← B; x2; Write(B); u2(B)</td>
</tr>
</tbody>
</table>

**Rule #3: Two Phase Locking (2PL) for Transactions**

\[ Ti = \ldots \, li(A) \ldots \ldots \, ui(A) \ldots \]

- no unlocks
- no locks

<table>
<thead>
<tr>
<th># locks held by Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>Growing Phase</td>
</tr>
<tr>
<td>Shrinkin Phase</td>
</tr>
<tr>
<td>Time</td>
</tr>
</tbody>
</table>
Schedule G

\[ T1 \quad l1(\text{A}); \text{Read(A)} \quad A \leftarrow A + 100; \text{Write(A)} \quad l1(\text{B}); \ u1(\text{A}) \]

\[ l2(\text{A}); \text{Read(A)} \quad A \leftarrow Ax2; \text{Write(A)}; \quad l2(\text{B}) \quad \text{delayed} \]

Schedule G

\[ T1 \quad l1(\text{A}); \text{Read(A)} \quad A \leftarrow A + 100; \text{Write(A)} \quad l1(\text{B}); \ u1(\text{A}) \]

\[ l2(\text{A}); \text{Read(A)} \quad A \leftarrow Ax2; \text{Write(A)}; \quad l2(\text{B}) \quad \text{delayed} \]

\[ \text{Read(B)}; B \leftarrow B + 100 \quad \text{Write(B)}; \ u1(\text{B}) \]

Schedule G

\[ T1 \quad l1(\text{A}); \text{Read(A)} \quad A \leftarrow A + 100; \text{Write(A)} \quad l1(\text{B}); \ u1(\text{A}) \]

\[ l2(\text{A}); \text{Read(A)} \quad A \leftarrow Ax2; \text{Write(A)}; \quad l2(\text{B}) \quad \text{delayed} \]

\[ \text{Read(B)}; B \leftarrow B + 100 \quad \text{Write(B)}; \ u1(\text{B}) \]

Schedule G

\[ T1 \quad l1(\text{A}); \text{Read(A)} \quad A \leftarrow A + 100; \text{Write(A)} \quad l1(\text{B}); \ u1(\text{A}) \]

\[ l2(\text{A}); \text{Read(A)} \quad A \leftarrow Ax2; \text{Write(A)}; \quad l2(\text{B}) \quad \text{delayed} \]

\[ \text{Read(B)}; B \leftarrow B + 100 \quad \text{Write(B)}; \ u1(\text{B}) \]

Schedule H (T2 reversed)

\[ T1 \quad l1(\text{A}); \text{Read(A)} \quad A \leftarrow A + 100; \text{Write(A)} \quad l1(\text{B}); \ u1(\text{A}) \]

\[ l2(\text{A}); \text{Read(A)} \quad A \leftarrow Ax2; \text{Write(A)}; \quad l2(\text{B}) \quad \text{delayed} \]

\[ \text{Read(B)}; B \leftarrow B + 100 \quad \text{Write(B)}; \ u1(\text{B}) \]

Schedule H (T2 reversed)

\[ T1 \quad l1(\text{A}); \text{Read(A)} \quad A \leftarrow A + 100; \text{Write(A)} \quad l1(\text{B}); \ u1(\text{A}) \]

\[ l2(\text{A}); \text{Read(A)} \quad A \leftarrow Ax2; \text{Write(A)}; \quad l2(\text{B}) \quad \text{delayed} \]

\[ \text{Read(B)}; B \leftarrow B + 100 \quad \text{Write(B)}; \ u1(\text{B}) \]
• 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 li(A), ui(A):
  • li(A), lj(A) conflict
  • li(A), uj(A) conflict

Note: no conflict <ui(A), uj(A)>, <li(A), rj(A)>, ...

Theorem  Rules #1,2,3 ⇒ conflict serializable schedule

To help in proof:
Definition Shrink(Ti) = SH(Ti) = first unlock action of Ti
Lemma

\[ Ti \rightarrow Tj \text{ in } S \Rightarrow SH(Ti) <_S SH(Tj) \]

**Proof of lemma:**

\[ Ti \rightarrow Tj \text{ means that} \]

\[ S = \ldots pi(A) \ldots qj(A) \ldots; \quad p,q \text{ conflict} \]

By rules 1,2:

\[ S = \ldots pi(A) \ldots ul(A) \ldots lj(A) \ldots qj(A) \ldots \]

By rule 3:

\[ SH(Ti) \ldots SH(Tj) \]

So, \[ SH(Ti) <_S SH(Tj) \]

---

**Theorem**

\[ \text{Rules #1,2,3 } \Rightarrow \text{ conflict (2PL) serializable schedule} \]

**Proof:**

1. Assume \( P(S) \) has cycle
   \[ T1 \rightarrow T2 \rightarrow \ldots \rightarrow Tn \rightarrow T1 \]
2. By lemma: \( SH(T1) < SH(T2) < \ldots < SH(T1) \)
3. Impossible, so \( P(S) \) acyclic
4. \( \Rightarrow S \) is conflict serializable

---

**2PL Subset of Serializable**

- \( S1: w1(x) w3(x) w2(y) w1(y) \)
  - \( S1 \) cannot be achieved via 2PL:
    The lock by \( T1 \) for \( y \) must occur after \( w2(y) \), so the unlock by \( T1 \) for \( x \) must occur after this point (and before \( w3(x) \)). Thus, \( w3(x) \) cannot occur under 2PL where shown in \( S1 \) because \( T1 \) holds the \( x \) lock at that point.
  - However, \( S1 \) is serializable (equivalent to \( T2, T1, T3 \)).
• 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:**

\[ S = ... l_s_1(A) r_1(A) l_s_2(A) r_2(A) ... u_s_1(A) u_s_2(A) \]

**Rule #1: Well Formed Transactions**

\[ Ti = ... l-S_1(A) ... r_1(A) ... u_1(A) ... \]
\[ Ti = ... l-X_1(A) ... w_1(A) ... u_1(A) ... \]

**Lock actions**

\[ l-ti(A): \text{lock } A \text{ in } t \text{ mode (} t \text{ is } S \text{ or } X) \]
\[ u-ti(A): \text{unlock } t \text{ mode (} t \text{ is } S \text{ or } X) \]

**Shorthand:**

\[ u_i(A): \text{unlock whatever modes } Ti \text{ has locked } A \]
• What about transactions that read and write same object?

Option 1: Request exclusive lock
Ti = ...l-X1(A) ... r1(A) ... w1(A) ... u(A) ...

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

Ti=...l-S1(A)...r1(A)...l-X1(A)...w1(A)...u(A)...

Rule #2: Legal Scheduler

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

no \( l-X_j(A) \)

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

no \( l-X_j(A) \)

no \( l-S_j(A) \)

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

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

Proof: similar to X locks case

Detail:
l-ti(A), l-rj(A) do not conflict if \text{comp}(t,r)
l-ti(A), u-rj(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), INj(A) do not conflict!

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

\[
\begin{align*}
A &= 15 \quad \text{INj(A)} \\
& \quad +10 \quad A = 17 \\
& \quad +2 \quad INi(A)
\end{align*}
\]
**Update Locks**

A common deadlock problem with upgrades:

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

--- Deadlock ---

**Solution**

If Ti wants to read A and knows it may later want to write A, it requests **update** lock (not shared)
Note: object A may be locked in different modes at the same time...

S1=...l-S1(A)...l-S2(A)...l-U3(A)...l-S4(A)...? 
  l-U4(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
  (E.g., may not even provide CONFLICT-serializable schedules)
- But here is one (simplified) way...
(1) Don’t trust transactions to request/release locks
(2) Hold all locks until transaction commits

Sample Locking System

Lock Table: Conceptually

But Use Hash Table:

If object not found in hash table, it is unlocked
Lock Info for A: Example

- Object:A
- Group mode:U
- Waiting:yes

List:

<table>
<thead>
<tr>
<th>Tran mode</th>
<th>Nxt</th>
<th>T_link</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>S</td>
<td>no</td>
</tr>
<tr>
<td>T2</td>
<td>U</td>
<td>no</td>
</tr>
<tr>
<td>T3</td>
<td>X</td>
<td>yes</td>
</tr>
</tbody>
</table>

To other T3 records

What Are The Objects We Lock?

- Relation A
- Relation B
- Tuple A
- Tuple B
- Tuple C
- Disk block A
- Disk block B

We Can Have It Both Ways!!

- 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

Ask any janitor to give you the solution...
Example

T1(IS), T2(S)

T1(S)

R1

t1

t2

t3

t4

Example

T1(IS), T2(IX)

R1

t1

t2

t3

t4

Multiple Granularity

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

Multiple Granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>T T T T F</td>
</tr>
<tr>
<td>IX</td>
<td>T T F F F</td>
</tr>
<tr>
<td>S</td>
<td>T F T F F</td>
</tr>
<tr>
<td>SIX</td>
<td>T F F F F</td>
</tr>
<tr>
<td>X</td>
<td>F F F F F</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 f2.2 in X mode? What locks will T2 get?
Exercise:

- Can T2 access object f2.2 in X mode? What locks will T2 get?

![Diagram 1]

Exercise:

- Can T2 access object f3.1 in X mode? What locks will T2 get?

![Diagram 2]

Exercise:

- Can T2 access object f2.2 in S mode? What locks will T2 get?

![Diagram 3]

Exercise:

- Can T2 access object f2.2 in X mode? What locks will T2 get?

![Diagram 4]
Insert + Delete Operations

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

T1: Insert <04,Kerry,...> into R
T2: Insert <04,Bush,...> into R

T1 | T2
---|---
S1(o1) | S2(o1)
S1(o2) | S2(o2)
Check Constraint | Check Constraint
Insert o3[04,Kerry,...] | Insert o4[04,Bush,...]
**Solution**

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

**Back To Example**

- **T1**: Insert<04,Kerry>  
  X1(R)
  Check Constraint  
  Insert<04,Kerry>  
  U(R)
  X2(R)  
  Check Constraint  
  Oops! e# = 04 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

Idea: Traverse Like “Monkey Bars”

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

---

**Tree Protocol with Shared Locks**

- Rules for shared & exclusive locks?

  - T1 S lock (released)
  - T1 X lock (released)
  - T1 S lock (held)
  - T1 X lock (will get)

  **T1 reads:**
  - B modified by T1
  - F not yet modified by T1

- **T1 S lock (released)**
- **T1 X lock (released)**
- **T1 S lock (held)**
- **T1 X lock (will get)**
Tree Protocol with Shared Locks

- Need more restrictive protocol
- Will this work??
  - Once $T_1$ locks one object in X mode, all further locks down the tree must be in X mode

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 $T_1$, $T_2$, $T_3$,… is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1 T_2 T_3$...

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:

RS(T2)={B} \not\subseteq RS(T3)={A,B} 
WS(T2)={B,D} \not\subseteq WS(T3)={C}

T_2 \text{ start} \quad T_3 \text{ start} \quad T_2 \text{ validated} \quad T_3 \text{ validated}

\cap \neq \emptyset

Another Thing Validation Must Prevent:

RS(T2)={A} \quad RS(T3)={A,B}
WS(T2)={D,E} \quad WS(T3)={C,D}

T_2 \text{ validated} \quad T_3 \text{ validated}

BAD: w_3(D) \quad w_2(D)
**Validation Rules For Tj:**

1. When Tj starts phase 1:
   \[ \text{ignore}(Tj) \leftarrow \text{FIN} \]
2. At Tj Validation:
   \[ \text{if check}(Tj) \text{ then} \]
   \[ \quad [ \text{VAL} \leftarrow \text{VAL} \cup \{Tj\}; \]
   \[ \quad \quad \text{do write phase;} \]
   \[ \quad \quad \text{FIN} \leftarrow \text{FIN} \cup \{Tj\} ] \]

**Check (Tj):**

For Ti ∈ VAL – IGNORE (Tj) DO

IF \[ \text{WS}(Ti) \cap \text{RS}(Tj) \neq \emptyset \text{ OR Ti } \notin \text{FIN} \]
RETURN false;

RETURN true;

Is this check too restrictive?

**Improving Check(Tj)**

For Ti ∈ VAL – IGNORE (Tj) DO

IF \[ \text{WS}(Ti) \cap \text{RS}(Tj) \neq \emptyset \text{ OR} \]
\[ (\text{Ti} \notin \text{FIN} \text{ AND WS}(Ti) \cap \text{WS}(Tj) \neq \emptyset) ] \]
RETURN false;

RETURN true;

**Exercise:**

- **U:** RS(U)={B}  \quad WS(U)={D}
- **W:** RS(W)={A,D}  \quad WS(W)={A,C}
- **T:** RS(T)={A,B}  \quad WS(T)={A,C}
- **V:** RS(V)={B}  \quad WS(V)={D,E}
Is Validation = 2PL?

S2: w2(y) w1(x) w2(x)

- S2 can be achieved with 2PL:
  l2(y) w2(y) l1(x) w1(x) l2(x) w2(x) u2(y) u2(x)

- S2 cannot be achieved by validation:
  The validation point of T2, val2 must occur before w2(y) since transactions do not write to the database until after validation. Because of the conflict on x, val1 < val2, so we must have something like
  S2: val1 val2 w2(y) w1(x) w2(x)
  With the validation protocol, the writes of T2 should not start until T1 is all done with its writes, which is not the case.

Validation Subset of 2PL?

- Possible proof (Check!):
  - Let S be validation schedule
  - For each T in S insert lock/unlocks, get S':
    - At T start: request read locks for all of RS(T)
    - At T validation: request write locks for WS(T);
      release read locks for read-only objects
    - At T end: release all write locks
  - Clearly transactions well-formed and 2PL
  - Must show S' is legal (next page)

- Say S' not legal:
  S': ... l1(x) w2(x) r1(x) val1 u2(x) ...
  - At val1: T2 not in Ignore(T1); T2 in VAL
  - T1 does not validate: WS(T2) ∩ RS(T1) ≠ ∅
  - contradiction!

- Say S' not legal:
  S': ... val1 l1(x) w2(x) w1(x) u2(x) ...
  - Say T2 validates first (proof similar in other case)
  - At val1: T2 not in Ignore(T1); T2 in VAL
  - T1 does not validate:
    T2 ∈ FIN AND WS(T1) ∩ WS(T2) ≠ ∅
  - contradiction!
Validation (also called optimistic concurrency control) is useful in some cases:
- Conflicts rare
- System resources plentiful
- Have real time constraints

Summary
Have studied concurrency control mechanisms used in practice
- 2PL
- Multiple granularity
- Tree (index) protocols
- Validation