The Problem

Different transactions may need to access data items at the same time, violating constraints.
The Problem

Even if each transaction maintains constraints by itself, interleaving their actions does not

Could try to run just one transaction at a time (serial schedule), but this has problems

» Too slow! Especially with external clients & IO
High-Level Approach

Define isolation levels: sets of guarantees about what transactions may experience

Strongest level: serializability (result is same as some serial schedule)

Many others possible: snapshot isolation, read committed, read uncommitted, …
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Example

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

T2: Read(A)
    A ← A×2
    Write(A)
    Read(B)
    B ← B×2
    Write(B)

Constraint: A = B
### Schedule C

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

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
## Schedule D

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

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>250</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>
Our Goal

Want schedules that are “good”, regardless of
  » initial state and
  » transaction semantics

Only look at order of read & write operations

Example:

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

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

\[ S'_C = r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B) \]
However, for $S_D$:

$$S_D = r_1(A)w_1(A)r_2(A)w_2(A) \quad r_2(B)w_2(B)r_1(B)w_1(B)$$

Another way to view this:

- $r_1(B)$ after $w_2(B)$ means $T_1$ should be after $T_2$ in an equivalent serial schedule ($T_2 \rightarrow T_1$)
- $r_2(A)$ after $w_1(A)$ means $T_2$ should be after $T_1$ in an equivalent serial schedule ($T_1 \rightarrow T_2$)
- Can’t have both of these!
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Concepts

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

**Conflicting actions:**

$$
\langle r_1(A), w_1(A), w_1(A) \rangle \quad \langle \quad \langle w_2(A), r_2(A), w_2(A) \rangle
$$

**Schedule:** a chronological order in which all the transactions’ actions are executed

**Serial schedule:** no interleaving of actions from different transactions
Question

Is it OK to model reads & writes as occurring at a single point in time in a schedule?

\[ S = \ldots \ r_1(x) \ \ldots \ w_2(b) \ \ldots \]
Question

What about conflicting, concurrent actions on same object?

Assume “atomic actions” that only occur at one point in time (e.g. implement using locking)
Definition

$S_1, S_2$ are **conflict equivalent** schedules if $S_1$ can be transformed into $S_2$ by a series of **swaps** of non-conflicting actions

(i.e., can reorder non-conflicting operations in $S_1$ to obtain $S_2$)
Definition

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule.

Key idea:

» Conflicts “change” result of reads and writes
» Conflict serializable means there exists some equivalent serial execution that does not change the effects

How can we compute whether a schedule is conflict serializable?
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Precedence Graph $P(S)$

Nodes: transactions in a schedule $S$

Edges: $T_i \rightarrow T_j$ whenever
- $p_i(A), q_j(A)$ are actions in $S$
- $p_i(A) <_S q_j(A)$ (occurs earlier in schedule)
- at least one of $p_i, q_j$ is a write (i.e. conflict)
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?
Another Exercise

What is P(S) for

\[ S = w_1(A) \ r_2(A) \ r_3(A) \ w_4(A) \]
Lemma

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

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

Proof:

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

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

$\Rightarrow S_1 = \ldots p_i(A) \ldots q_j(A) \ldots$

$S_2 = \ldots q_j(A) \ldots p_i(A) \ldots$

\[
\begin{cases} 
  \text{conflict} & \text{for } p_i, q_j \\
\end{cases}
\]

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

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

(\(\Leftarrow\)) Assume \(S_1\) is conflict serializable
\[ \Rightarrow \exists S_s (\text{serial}): S_s, S_1 \text{ conflict equivalent} \]
\[ \Rightarrow P(S_s) = P(S_1) \text{ (by previous lemma)} \]
\[ \Rightarrow P(S_1) \text{ acyclic since } P(S_s) \text{ is acyclic} \]
Theorem

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

P(S₁) acyclic ⇐⇒ S₁ conflict serializable

(⇒) Assume P(S₁) is acyclic

Transform S₁ as follows:

1) Take T₁ to be transaction with no inbound edges
2) Move all T₁ actions to the front
   \[ S₁ = \ldots \, q_j(A) \ldots \, p_1(A) \ldots \]

3) we now have S₁ = <T₁ actions><... rest ...>
4) repeat above steps to serialize rest!
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
How to Enforce Serializable Schedules?

Option 1: run system, recording P(S); at end of day, check for cycles in P(S) and declare whether execution was good
How to Enforce Serializable Schedules?

Option 2: prevent P(S) cycles from occurring

T_1  T_2  .....  T_n

Scheduler

DB
A Locking Protocol

Two new actions:

lock: $l_i (A)$  \(\leftarrow\)  Transaction $i$ locks object $A$

unlock: $u_i (A)$
Rule #1: Well-Formed Transactions

\[ T_i: \ldots l_i(A) \ldots r_i(A) \ldots u_i(A) \ldots \]

Transactions can only operate on locked items
Rule #2: Legal Scheduler

\[ S = \ldots \, l_i(A) \ldots \ldots \ldots u_i(A) \ldots \ldots \]

\[ \text{no } l_j(A) \]

Only one transaction can lock item at a time
Exercise

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

\[ S_1 = l_1(A) l_1(B) r_1(A) w_1(B) l_2(B) u_1(A) u_1(B) r_2(B) w_2(B) u_2(B) l_3(B) r_3(B) u_3(B) \]

\[ S_2 = l_1(A) r_1(A) w_1(B) u_1(A) u_1(B) l_2(B) r_2(B) w_2(B) l_3(B) r_3(B) u_3(B) \]

\[ S_3 = l_1(A) r_1(A) u_1(A) l_1(B) w_1(B) u_1(B) l_2(B) r_2(B) w_2(B) u_2(B) l_3(B) r_3(B) u_3(B) \]
Exercise

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

\[ S_1 = l_1(A) \ l_1(B) \ r_1(A) \ w_1(B) \ l_2(B) \ u_1(A) \ u_1(B) \]
\[ r_2(B) \ w_2(B) \ u_2(B) \ l_3(B) \ r_3(B) \ u_3(B) \]

\[ S_2 = l_1(A) \ r_1(A) \ w_1(B) \ u_1(A) \ u_1(B) \ l_2(B) \ r_2(B) \]
\[ w_2(B) \ l_3(B) \ r_3(B) \ u_3(B) \ u_2(B) \text{ missing} \]

\[ S_3 = l_1(A) \ r_1(A) \ u_1(A) \ l_1(B) \ w_1(B) \ u_1(B) \]
\[ l_2(B) \ r_2(B) \ w_2(B) \ u_2(B) \ l_3(B) \ r_3(B) \ u_3(B) \]
## Schedule F

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A);Read(A)</td>
<td>A←A+100;Write(A);u₁(A)</td>
<td>l₂(A);Read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A←Ax2;Write(A);u₂(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l₂(B);Read(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B←Bx2;Write(B);u₂(B)</td>
</tr>
<tr>
<td>l₁(B);Read(B)</td>
<td>B←B+100;Write(B);u₁(B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

CS 245
Rule #3: 2-Phase Locking (2PL)

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

Transactions first lock all items they need, then unlock them.
2-Phase Locking (2PL)

# locks held by $T_i$

Growing Phase

Shrinking Phase
## Schedule G

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

<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)</td>
<td>A ← A × 2; Write(A)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>l₂(B) delayed</td>
</tr>
</tbody>
</table>
## Schedule G

<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)</td>
<td>A ← A × 2; Write(A)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>l₂(B)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td>delayed</td>
</tr>
<tr>
<td>Write(B); u₁(B)</td>
<td></td>
</tr>
</tbody>
</table>
## Schedule G

<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)</td>
<td>A ← A × 2; Write(A)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>l₂(B) ← delayed</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td>l₂(B); u₂(A); Read(B)</td>
</tr>
<tr>
<td>Write(B); u₁(B)</td>
<td>B ← B × 2; Write(B); u₂(B)</td>
</tr>
</tbody>
</table>
Schedule H (T2 Ops 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 \leftarrow A + 100 ); Write(A)</td>
<td>( B \leftarrow B \times 2 ); Write(B)</td>
</tr>
<tr>
<td>( l_1(B) \leftarrow \text{delayed} ) (T2 holds B)</td>
<td>( l_2(A) \leftarrow \text{delayed} ) (T1 holds A)</td>
</tr>
</tbody>
</table>

Problem: Deadlock between transactions
Dealing with Deadlock

Option 1: Detect deadlocks and roll back one of the deadlocked transactions
  » The rolled back transaction no longer appears in our schedule

Option 2: Agree on an order to lock items in that prevents deadlocks
  » E.g. transactions acquire locks in key order
  » Must know which items $T_i$ will need up front!
Is 2PL Correct?

Yes! We can prove that following rules #1,2,3 gives conflict-serializable schedules.
Conflict Rules for Lock Ops

$l_i(A), l_j(A)$ 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-serializable schedule (2PL)

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:
$Ti \rightarrow Tj$ means that

$S = \ldots p_i(A) \ldots q_j(A) \ldots; \text{ } p,q \text{ conflict}$

By rules 1, 2:

$S = \ldots p_i(A) \ldots u_i(A) \ldots l_j(A) \ldots q_j(A) \ldots$

By rule 3: $SH(Ti)$ $SH(Tj)$

So, $SH(Ti) <_S SH(Tj)$
Theorem: Rules #1,2,3  \Rightarrow Conflict 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) acyclic

(4)  \Rightarrow S is conflict serializable
2PL Subset of Serializable
S1: $w_1(X) \ w_3(X) \ w_2(Y) \ w_1(Y)$

S1 cannot be achieved via 2PL: The lock by T1 for Y must occur after $w_2(Y)$, so the unlock by T1 for X must occur after this point (and before $w_1(X)$). Thus, $w_3(X)$ cannot occur under 2PL where shown in S1.

But S1 is serializable: equivalent to T2, T1, T3.
If You Need More Practice

Are our schedules $S_C$ and $S_D$ 2PL schedules?

$S_C: \ w_1(A) \ w_2(A) \ w_1(B) \ w_2(B)$

$S_D: \ w_1(A) \ w_2(A) \ w_2(B) \ w_1(B)$
Optimizing Performance

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:

\[ S = \ldots l_1(A) \ r_1(A) \ u_1(A) \ \ldots \ l_2(A) \ r_2(A) \ u_2(A) \ \ldots \]

Do not conflict
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:

\[ S = \ldots l_{s_1}(A) \; r_1(A) \; l_{s_2}(A) \; r_2(A) \; \ldots \; u_{s_1}(A) \; u_{s_2}(A) \]

Do not conflict
Multiple Lock Modes

Lock actions
l-m_i(A): lock A in mode m (m is S or X)

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

Shorthand:
ui(A): unlock whatever modes Ti has locked A
Rule 1: Well-Formed Transactions

\[ T_i = \ldots \text{i-S}_1(A) \ldots \text{r}_1(A) \ldots \text{u}_1(A) \ldots \]

\[ T_i = \ldots \text{i-X}_1(A) \ldots \text{w}_1(A) \ldots \text{u}_1(A) \ldots \]

Transactions must acquire the right lock type for their actions (S for read only, X for r/w).
Rule 1: Well-Formed Transactions

What about transactions that read and write same object?

Option 1: Request exclusive lock

\[ T1 = \ldots l-X_{1}(A) \ldots r_{1}(A) \ldots w_{1}(A) \ldots u(A) \ldots \]
Rule 1: Well-Formed Transactions

What about transactions that read and write same object?

Option 2: Upgrade lock to X on write

\[ T1 = \ldots l-S_1(A) \ldots r_1(A) \ldots l-X_1(A) \ldots w_1(A) \ldots u_1(A) \ldots \]

(Think of this as getting a 2\textsuperscript{nd} lock, or dropping S to get X.)
Rule 2: Legal Scheduler

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

\[ \text{no} \ l-X_j(A) \]

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

\[ \text{no} \ l-X_j(A) \]

\[ \text{no} \ l-S_j(A) \]
A Way to Summarize Rule #2

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

(II) If upgrade releases read lock (e.g., $S \rightarrow X$)

can be allowed in growing phase
Rules 1,2,3 $\Rightarrow$ Conf. Serializable Schedules for S/X Locks

Proof: similar to X locks case

Detail:

$l\cdot m_i(A), l\cdot n_j(A)$ do not conflict if $\text{compat}(m,n)$

$l\cdot m_i(A), u\cdot n_j(A)$ do not conflict if $\text{compat}(m,n)$
Lock Modes Beyond S/X

Examples:

(1) increment lock

(2) update lock
Example 1: Increment Lock

Atomic addition action: IN_i(A)

\{Read(A); A \leftarrow A+k; Write(A)\}

IN_i(A), IN_j(A) do not conflict, because addition is commutative!
Compatibility Matrix

<table>
<thead>
<tr>
<th>compat</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>
### Compatibility Matrix

<table>
<thead>
<tr>
<th>compat</th>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td><strong>T</strong></td>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
</tr>
<tr>
<td><strong>X</strong></td>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
</tr>
<tr>
<td><strong>I</strong></td>
<td><strong>F</strong></td>
<td><strong>F</strong></td>
<td><strong>T</strong></td>
</tr>
</tbody>
</table>
# Update Locks

A common deadlock problem with upgrades:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-S₁(A)</td>
<td>I-S₂(A)</td>
</tr>
<tr>
<td>I-X₁(A)</td>
<td>I-X₂(A)</td>
</tr>
</tbody>
</table>

--- Deadlock ---
Solution

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

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

New request
# Compatibility Matrix

Note: asymmetric table!

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

New request

compat

Lock already held in

---

CS 245

75
How Is Locking Implemented In Practice?

Every system is different (e.g., may not even provide conflict serializable schedules)

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

1. Don’t ask transactions to request/release locks: just get the weakest lock for each action they perform

2. Hold all locks until transaction commits

![Graph showing increase in number of locks over time](image-url)
Sample Locking System

Under the hood: lock manager that keeps track of which objects are locked
  » E.g. hash table

Also need a good way to block transactions until locks are available, and find deadlocks
Which Objects Do We Lock?

Table A
- Table B
  - Tuple A
    - Tuple B
      - Tuple C
  - Disk block A
    - Disk block B
  - DB

Tuple A
- DB

Tuple B
- DB

Tuple C
- DB

Disk block A
- DB
Which Objects Do We Lock?

Locking works in any case, but should we choose small or large objects?
Which Objects Do 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...

```
Stall 1  Stall 2  Stall 3  Stall 4
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
r

restroom

hall
```
Example
Example

R1

\[ T_1(IS) \]

\[ T_1(S) \]
Example

R1

$t_1$

$t_2$  \(\xrightarrow{T_1(S)}\)

$t_3$  \(\xrightarrow{T_1(IS), T_2(S)}\)

$t_4$
Example 2

R1

\text{T}_1(\text{IS})

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

t_1 \quad t_2 \quad t_3 \quad t_4
Example 2

R1

\[ T_1(IS), T_2(IX) \]

\[ T_1(S) \]

\[ T_2(IX) \]
### Multiple Granularity Locks

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IS</strong></td>
<td>IS</td>
<td>IS</td>
<td>IS</td>
<td>IS</td>
<td>IS</td>
</tr>
<tr>
<td><strong>IX</strong></td>
<td>IX</td>
<td>IX</td>
<td>IX</td>
<td>IX</td>
<td>IX</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td><strong>SIX</strong></td>
<td>SIX</td>
<td>SIX</td>
<td>SIX</td>
<td>SIX</td>
<td>SIX</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Multiple Granularity Locks

compat

<table>
<thead>
<tr>
<th></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>F</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>

Requestor

Holder
## Rules Within A Transaction

<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked by same transaction 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>none</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 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:

- R1
- T1(IX)
- T1(X)
- t1
- t2
- t3
- t4
- f2.1
- f2.2
- f3.1
- f3.2
Exercise:

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

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

Can T2 access object f2.2 in X mode? What locks will T2 get?
Insert + delete operations

Insert
Changes 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 Problem: Phantoms

Example: relation $R$ (id, name, …)
constraint: id is unique key
use tuple locking

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>id</td>
<td>Name</td>
<td>…..</td>
</tr>
<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 <12, Mary, …> into R
T2: Insert <12, Sam, …> into R

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1(o1)$</td>
<td>$S_2(o1)$</td>
</tr>
<tr>
<td>$S_1(o2)$</td>
<td>$S_2(o2)$</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Insert $o_3[12, Mary, ..]$</td>
<td>Insert $o_4[12, Sam, ..]$</td>
</tr>
</tbody>
</table>
Solution

Use multiple granularity tree

Before insert of node N, lock parent(N) in X mode
Back to example

T₁: Insert<12, Mary>

T₂: Insert<12, Sam>

X₁(R)

Check constraint
Insert<12, Mary>
U₁(R)

X₂(R)

Check constraint
Oops! e# = 12 already in R!

X₂(R) delayed
Instead of Using R, Can Use Index Nodes for Ranges

Example:

```
R
  /  \
Index 0<E#<100 Index 100<E#<200
   \       /  \
  E#=2    E#=107  E#=109
    \      / \
     E#=5  ...
```

...
Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking
  » Shared and exclusive locks
  » Lock tables and multi-level locking

Optimistic concurrency with validation
Next Class

Guest talk by **Reynold Xin** from Databricks:

Delta Lake: Making Cloud Data Lakes Transactional and Scalable

The same concurrency issues we saw happen in large data lakes with billions of files… how to offer transactions there?