Concurrency Control

Instructor: Matei Zaharia

[link: cs245.stanford.edu]
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

\( T_1: \) Read(A)  \hspace{1cm} \( T_2: \) Read(A)
\[
\begin{align*}
A &\leftarrow A + 100 \\
\text{Write}(A) \\
\text{Read}(B) \\
B &\leftarrow B + 100 \\
\text{Write}(B)
\end{align*}
\]

\[
\begin{align*}
A &\leftarrow A \times 2 \\
\text{Write}(A) \\
\text{Read}(B) \\
B &\leftarrow B \times 2 \\
\text{Write}(B)
\end{align*}
\]

Constraint: A=B
## 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></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>Read(A); A ← A×2;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td>Write(A);</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>250</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td></td>
<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>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>T₂</td>
<td>125</td>
<td>250</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
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:

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

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

\[ T_1 \quad T_2 \]
However, for $S_D$:

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

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

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

Conflicting actions: $r_1(A)$ $w_1(A)$ $w_1(A)$ $r_2(A)$ $w_2(A)$ $w_2(A)$

pairs of actions that would change the result of a read or write if swapped
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

Schedules $S_1$, $S_2$ are conflict equivalent 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 implies that there exists at least one equivalent serial execution with the same 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. $p_i(A)$ and $q_j(A)$ are conflicting actions)
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 $S_2$

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

$S_2 = ...q_j(A)... p_i(A)...$

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

**Counter example:**

\[
S_1 = w_1(A) \text{ } r_2(A) \text{ } w_2(B) \text{ } r_1(B)
\]

\[
S_2 = r_2(A) \text{ } w_1(A) \text{ } r_1(B) \text{ } w_2(B)
\]
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(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 = ........ q_j(A) ........ p_1(A) ........

(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 \quad T_2 \quad \ldots \quad 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

$\mathbf{T}_i$: $\ldots \ l_i(A) \ \ldots \ r_i(A) \ \ldots \ u_i(A) \ \ldots$

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

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

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

Only one transaction can lock item at a time
Exercise

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

\[ S_1 = l_1(A) \ l_1(B) \ r_1(A) \ w_1(B) \ l_2(B) \ u_1(A) \ u_1(B) \]
\[ r_2(B) \ w_2(B) \ u_2(B) \ l_3(B) \ r_3(B) \ u_3(B) \]
\[ S_2 = l_1(A) \ r_1(A) \ w_1(B) \ u_1(A) \ u_1(B) \ l_2(B) \ r_2(B) \]
\[ w_2(B) \ l_3(B) \ r_3(B) \ u_3(B) \]
\[ S_3 = l_1(A) \ r_1(A) \ u_1(A) \ l_1(B) \ w_1(B) \ u_1(B) \ l_2(B) \]
\[ r_2(B) \ w_2(B) \ u_2(B) \ l_3(B) \ r_3(B) \ u_3(B) \]
Exercise

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

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

\[ S_2 = l_1(A) r_1(A) w_1(B) u_1(A) u_1(B) l_2(B) r_2(B) w_2(B) l_3(B) r_3(B) u_3(B) \text{ u}_2(B) \text{ missing} \]

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1(A); \text{Read}(A)$</td>
<td>$l_2(A); \text{Read}(A)$</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>$A \leftarrow A + 100; \text{Write}(A); u_1(A)$</td>
<td>$A \leftarrow A \times 2; \text{Write}(A); u_2(A)$</td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td>$l_1(B); \text{Read}(B)$</td>
<td>$l_2(B); \text{Read}(B)$</td>
<td>250</td>
<td>50</td>
</tr>
<tr>
<td>$B \leftarrow B + 100; \text{Write}(B); u_1(B)$</td>
<td>$B \leftarrow B \times 2; \text{Write}(B); u_2(B)$</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>
Rule #3: 2-Phase Locking (2PL)

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

- no unlocks
- no locks

Transactions must 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>$l_1(A)$; Read($A$)</td>
<td>$l_2(A)$; Read($A$)</td>
</tr>
<tr>
<td>$A \leftarrow A + 100$; Write($A$)</td>
<td>$A \leftarrow A \times 2$; Write($A$)</td>
</tr>
<tr>
<td>$l_1(B)$; $u_1(A)$</td>
<td>$l_2(B)$; $u_1(B)$</td>
</tr>
<tr>
<td>Read($B$); $B \leftarrow B + 100$</td>
<td>$l_2(B)$; delayed</td>
</tr>
<tr>
<td>Write($B$); $u_1(B)$</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)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td>l₂(B) ← delayed</td>
</tr>
<tr>
<td>Write(B); u₁(B)</td>
<td>l₂(B); u₂(A); Read(B)</td>
</tr>
<tr>
<td></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); \text{Read}(A) )</td>
<td>( l_2(B); \text{Read}(B) )</td>
</tr>
<tr>
<td>( A \leftarrow A + 100; \text{Write}(A) )</td>
<td>( B \leftarrow B \times 2; \text{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 the 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 \implies \text{conflict-serializable schedule (2PL)}

To help in proof:

**Definition:** Shrink(T_i) = SH(T_i) = first unlock action of T_i
Lemma

\[ T_i \rightarrow T_j \text{ in } S \Rightarrow SH(T_i) <_S SH(T_j) \]

Proof:

\( T_i \rightarrow T_j \) means that
\[ S = \ldots p_i(A) \ldots q_j(A) \ldots; \text{ } p, q \text{ conflict} \]

By rules 1, 2:
\[ S = \ldots p_i(A) \ldots u_i(A) \ldots I_j(A) \ldots q_j(A) \ldots \]

By rule 3:
\[ SH(T_i) \quad SH(T_j) \]

So, \( SH(T_i) <_S SH(T_j) \)
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 is a Subset of Serializable
$S_1$: $w_1(X) \ w_3(X) \ w_2(Y) \ w_1(Y)$

$S_1$ cannot be achieved via 2PL: The lock by $T_1$ for $Y$ must occur after $w_2(Y)$, so the unlock by $T_1$ for $X$ must occur after this point (and before $w_1(X)$). Thus, $w_3(X)$ cannot occur under 2PL where shown in $S_1$.

But $S_1$ is serializable: equivalent to $T_2$, $T_1$, $T_3$. 
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:

\[ S=\ldots \, l-S_1(A) \, r_1(A) \, l-S_2(A) \, r_2(A) \ldots \, u_1(A) \, u_2(A) \]
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:
u_i(A): unlock whatever modes T_i 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

\[ T_1 = \ldots l - X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots \]
Rule 1: Well-Formed Transactions

What about transactions that read and write same object?

Option 2: Upgrade lock to X on write

\[ T_1 = \ldots l-S_1(A)\ldots r_1(A)\ldots l-X_1(A)\ldots w_1(A)\ldots u_1(A)\ldots \]

(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>

compat =

Lock already held in

New request
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 lock (e.g., S→X)
     can be allowed in growing phase
Proof: similar to X locks case

Detail:

l-m_i(A), l-n_j(A) do not conflict if compat(m,n)
l-m_i(A), u-n_j(A) do not conflict if compat(m,n)
Lock Modes Beyond S/X

Examples:

(1) increment lock

(2) update lock
Example 1: Increment Lock

Atomic addition action: $\text{IN}_i(A)$

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

$\text{IN}_i(A)$, $\text{IN}_j(A)$ do not conflict, because addition is commutative!
## Compatibility Matrix

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

**Lock already held in**

**New request**

**compat**
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>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New request

- Lock already held in

**compat**

- Lock already held in
Compatibility Matrix

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

New request

Note: asymmetric table!
Which Objects Do We Lock?

Table A
- Table B
- ::
- ::
- DB

Tuple A
- Tuple B
- Tuple C
- ::
- ::
- DB

Disk block A
- Disk block B
- ::
- 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...

```
<table>
<thead>
<tr>
<th>Stall 1</th>
<th>Stall 2</th>
<th>Stall 3</th>
<th>Stall 4</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>restroom</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>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

hall
```
Example

\[ \text{R1} \quad \text{t}_1 \quad \text{t}_2 \quad \text{t}_3 \quad \text{t}_4 \]
Example

```
R1
  |--- t1
  |   |--- t2
  |   |   |--- t3
  |   |--- t4
  |--- T1(IS)
  |--- T1(S)
```
Example

\[ T_1(IS), T_2(S) \]
Example 2

Diagram:
- R1
  - t1
  - t2
  - t3
  - t4

Arrows:
- R1 -> t1
- R1 -> t2
- R1 -> t3
- R1 -> t4
- T₁(IS) from R1 to t1
- T₁(S) from t2 to R1
Example 2

\[ T_1(IS), T_2(IX) \]
Example 3

R1

\[ T_1(IS), T_2(S), T_3(IX)? \]
## Multiple Granularity Locks

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

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

*Notes:*
- **IS**: Shared lock on individual objects.
- **IX**: Shared lock on individual objects with exclusive access.
- **S**: Shared lock on individual objects.
- **SIX**: Shared lock on individual objects with exclusive access on different objects.
- **X**: Exclusive lock on individual objects.