Data-Intensive Computing Systems

Concurrency Control (II)

Shivnath Babu
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
How to enforce serializable schedules?

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

\[ \text{T}_1 \quad \text{T}_2 \quad \ldots \quad \text{T}_n \]

Scheduler

DB
A locking protocol

Two new actions:

lock (exclusive): $l_i (A)$
unlock: $u_i (A)$
Rule #1: Well-formed transactions

Ti: ... li(A) ... pi(A) ... ui(A) ...
Rule #2  Legal scheduler

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

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

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

S1 = l_1(A)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)

S2 = 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)

S3 = 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:

• What schedules are legal?
  What transactions are well-formed?

S1 = l₁(A)l₁(B)r₁(A)w₁(B)l₂(B)u₁(A)u₁(B)
    r₂(B)w₂(B)u₂(B)l₃(B)r₃(B)u₃(B)

S2 = l₁(A)r₁(A)w₁(B)u₁(A)u₁(B)
    l₂(B)r₂(B)w₂(B)l₃(B)r₃(B)u₃(B)

S3 = l₁(A)r₁(A)u₁(A)l₁(B)w₁(B)u₁(B)
    l₂(B)r₂(B)w₂(B)u₂(B)l₃(B)r₃(B)u₃(B)
## Schedule F

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_1(A); \text{Read}(A) )</td>
<td>( l_2(A); \text{Read}(A) )</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>
</tr>
<tr>
<td>( l_1(B); \text{Read}(B) )</td>
<td>( l_2(B); \text{Read}(B) )</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>
</tr>
</tbody>
</table>
## 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 ← Ax₂;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 ← Bx₂;Write(B);u₂(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td>150</td>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>
Rule #3 Two phase locking (2PL) for transactions

$T_i = \ldots \text{l}_i(A) \ldots \text{u}_i(A) \ldots$

| no unlocks | no locks |
# locks held by Ti

![Graph showing the number of locks held by Ti over time, with phases labeled as Growing and Shrinking.](image-url)
## 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 ← Ax₂; Write(A); l₂(B)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>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 x 2; Write(A) delayed</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td></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>$l_1(A)$; Read(A)</td>
<td>$l_1(B) u_1(A)$</td>
</tr>
<tr>
<td>$A \leftarrow A+100; \text{Write}(A)$</td>
<td>$l_2(A); \text{Read}(A)$</td>
</tr>
<tr>
<td>$l_1(B); u_1(A)$</td>
<td>$A \leftarrow Ax2; \text{Write}(A); l_2(B)$</td>
</tr>
<tr>
<td>Read(B); $B \leftarrow B+100$</td>
<td>delayed</td>
</tr>
<tr>
<td>Write(B); $u_1(B)$</td>
<td>$l_2(B); u_2(A); \text{Read}(B)$</td>
</tr>
<tr>
<td></td>
<td>B $\leftarrow Bx2; \text{Write}(B); u_2(B)$</td>
</tr>
</tbody>
</table>
Schedule H (T2 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 ← Bx2; Write(B)</td>
</tr>
</tbody>
</table>

$\downarrow l_1(B)$ delayed

$\downarrow l_2(A)$ delayed
• 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 $l_i(A), u_i(A)$:

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

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

$Ti \rightarrow Tj$ in $S \Rightarrow SH(Ti) <_S SH(Tj)$

Proof of lemma:

$Ti \rightarrow Tj$ means that

$S = \ldots p_i(A) \ldots q_j(A) \ldots$; $p,q$ 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  \implies \text{ 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) acyclic
(4) \implies S \text{ 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:

\[ S = \ldots l_1(A) \text{ r}_1(A) \text{ u}_1(A) \ldots \text{ l}_2(A) \text{ r}_2(A) \text{ u}_2(A) \ldots \]

Instead:

\[ S=\ldots l_{s1}(A) \text{ r}_1(A) l_{s2}(A) \text{ r}_2(A) \ldots \text{ u}_{s1}(A) \text{ u}_{s2}(A) \]

Do not conflict
**Lock actions**

l-\textit{t}_i(A): lock A in t mode (t is S or X)

u-\textit{t}_i(A): unlock t mode (t is S or X)

**Shorthand:**

u_\textit{i}(A): unlock whatever modes

T_i has locked A
Rule #1  Well formed transactions

\[ T_i = \ldots l-S_1(A) \ldots r_1(A) \ldots u_1(A) \ldots \]
\[ T_i = \ldots l-X_1(A) \ldots w_1(A) \ldots u_1(A) \ldots \]
• What about transactions that read and write same object?

Option 1: Request exclusive lock

\[ T_i = \ldots l-X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots \]
• What about transactions that read and write same object?

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

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

- Get 2nd lock on A, or
- Drop S, get X lock

Think of
Rule #2  Legal scheduler

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

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

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

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

\[ \text{no } l-S_j(A) \]
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 goes to more dominant lock atomically (e.g., $S \rightarrow X$)
    - has to be atomic
**Theorem**  Rules 1,2,3 ⇒ 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: \(\text{IN}_i(A)\)
  \[\{\text{Read}(A); \ A \leftarrow A+k; \ \text{Write}(A)\}\]
- \(\text{IN}_i(A), \ \text{IN}_j(A)\) do not conflict!

\[
\begin{align*}
A=5 & \quad \text{IN}_i(A) \rightarrow A=7 & \quad \text{IN}_j(A) \rightarrow A=17 \\
& \quad +2 \quad \quad \quad \quad \quad \quad +10 & \quad +2 \\
& \quad \quad \text{IN}_j(A) \rightarrow A=15 & \quad \text{IN}_i(A)
\end{align*}
\]
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>
<tr>
<td>Comp</td>
<td></td>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>I</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
Update locks

A common deadlock problem with upgrades:

T1

<table>
<thead>
<tr>
<th>l-S₁(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>l-X₁(A)</td>
</tr>
</tbody>
</table>

T2

<table>
<thead>
<tr>
<th>l-S₂(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>l-X₂(A)</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)
<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>

New request

Lock already held in
### New request

<table>
<thead>
<tr>
<th>Comp</th>
<th>Lock already held in</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>X</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>TorF</td>
</tr>
</tbody>
</table>

$\rightarrow$ symmetric table?
Note: object A may be locked in different modes at the same time...

\[ S_1 = \ldots \underline{\text{l-S}_1(A)} \ldots \underline{\text{l-S}_2(A)} \ldots \underline{\text{l-U}_3(A)} \ldots \]

\[ \begin{aligned}
& \underline{\text{l-S}_4(A)} \ldots ? \\
& \underline{\text{l-U}_4(A)} \ldots ? \\
\end{aligned} \]

• 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 ...
Sample Locking System:

(1) Don’t trust transactions to request/release locks
(2) Hold all locks until transaction commits
Scheduler, part I

Scheduler, part II

DB

Read(A), Write(B)

l(A), Read(A), l(B), Write(B)...

Ti

lock table
Lock table

Conceptually

Every possible object

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

If null, object is unlocked

Lock info for B

Lock info for C
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</th>
<th>Group</th>
<th>Wait</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>

tran mode wait? Nxt T_link

To other T3 records
What are the objects we lock?

<table>
<thead>
<tr>
<th>Relation A</th>
<th>Tuple A</th>
<th>Disk block A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation B</td>
<td>Tuple B</td>
<td>Disk block B</td>
</tr>
<tr>
<td></td>
<td>Tuple C</td>
<td></td>
</tr>
</tbody>
</table>

DB

DB

DB
• 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>restroom</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

hall
Example

R1

$T_1(\text{IS}), T_2(\text{S})$
Example

Example

$R1 \rightarrow t_1,t_2,t_3,t_4 \rightarrow T_1(IS), T_2(IX)$
## Multiple granularity

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

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

<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked in</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>

Diagram:

- P
- C
### By same transaction

<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>
Exercise:
• Can T₂ access object f₂.2 in X mode? What locks will T₂ get?
Exercise:

• Can $T_2$ access object $f_{2.2}$ in X mode? What locks will $T_2$ get?
Exercise:

- Can $T_2$ access object $f_{3.1}$ in $X$ mode? What locks will $T_2$ get?
Exercise:

- Can $T_2$ access object $f_{2.2}$ in S mode? What locks will $T_2$ get?
Exercise:

• Can $T_2$ access object $f_{2.2}$ in X mode? What locks will $T_2$ get?
Insert + delete operations

\[
\begin{array}{c}
A \\
\vdots \\
Z \\
\alpha
\end{array}
\]

\[
\text{Insert}
\]
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

Are these enough?
Still have a problem: **Phantoms**

Example: relation $R (E\#, \text{name}, \ldots)$
constraint: $E\#$ is key
use tuple locking

<table>
<thead>
<tr>
<th>R</th>
<th>E#</th>
<th>Name</th>
<th>....</th>
</tr>
</thead>
<tbody>
<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>
$T_1$: Insert $<04,Kerry,...>$ into $R$

$T_2$: Insert $<04,Bush,...>$ into $R$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1(o_1)$</td>
<td>$S_2(o_1)$</td>
</tr>
<tr>
<td>$S_1(o_2)$</td>
<td>$S_2(o_2)$</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>

Insert $o_3[04,Kerry,..]$  
Insert $o_4[04,Bush,..]$
Solution

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

<table>
<thead>
<tr>
<th>$T_1$: Insert&lt;04, Kerry&gt;</th>
<th>$T_2$: Insert&lt;04, Bush&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>$X_1(R)$</td>
<td>$X_2(R)$ delayed</td>
</tr>
</tbody>
</table>

Check constraint
Insert<04, Kerry>
U(R)

$X_2(R)$
Check constraint
Oops! e# = 04 already in R!
Instead of using R, can use index on R:

Example:

```plaintext
Index
0<\text{E\#} \leq 100

E\#=2 \quad E\#=5

...  

Index
100<\text{E\#} \leq 200

E\#=107 \quad E\#=109

...  
```
• 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]

- can we release A lock if we no longer need A??
Idea: traverse like “Monkey Bars”
Let us see why this works?

- Assume all $T_i$ start at root; exclusive lock
- $T_i \rightarrow T_j \Rightarrow T_i$ locks root before $T_j$

- Actually works if we don’t always start at root
Rules: tree protocol (exclusive locks)

(1) First lock by $T_i$ may be on any item
(2) After that, item $Q$ can be locked by $T_i$ only if parent($Q$) locked by $T_i$
(3) Items may be unlocked at any time
(4) After $T_i$ 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 $S_s = T_1 \ T_2 \ T_3 \ldots$
To implement validation, system keeps two sets:

- **FIN** = transactions that have finished phase 3 (and are all done)
- **VAL** = transactions that have successfully finished phase 2 (validation)
Example of what validation must prevent:

\[ RS(T_2) = \{B\} \quad RS(T_3) = \{A,B\} \neq \emptyset \]
\[ WS(T_2) = \{B,D\} \quad WS(T_3) = \{C\} \]
Example of what validation must prevent:

\[ RS(T_2) = \{B\} \]
\[ WS(T_2) = \{B,D\} \]
\[ RS(T_3) = \{A,B\} \neq \emptyset \]
\[ WS(T_3) = \{C\} \]

\[ T_2 \text{ start} \]
\[ T_3 \text{ start} \]
\[ T_2 \text{ validated} \]
\[ T_3 \text{ validated} \]

Time

Phase 3
Another thing validation must prevent:

\[ \text{RS}(T_2) = \{A\} \quad \text{RS}(T_3) = \{A, B\} \]
\[ \text{WS}(T_2) = \{D, E\} \quad \text{WS}(T_3) = \{C, D\} \]

BAD: \( w_3(D) \quad w_2(D) \)
Another thing validation must prevent:

RS(T_2) = \{A\}  \quad RS(T_3) = \{A, B\}
WS(T_2) = \{D, E\}  \quad WS(T_3) = \{C, D\}
Validation rules for $T_j$:

(1) When $T_j$ starts phase 1:
    \[ \text{ignore}(T_j) \leftarrow \text{FIN} \]

(2) at $T_j$ Validation:
    \[
    \text{if check } (T_j) \text{ then }
    \begin{align*}
    & \text{VAL} \leftarrow \text{VAL} \cup \{T_j\}; \\
    & \text{do write phase;} \\
    & \text{FIN} \leftarrow \text{FIN} \cup \{T_j\}
    \end{align*}
    \]
Check (T_j):

For Ti ∈ VAL - IGNORE (T_j) DO

    IF [ WS(T_i) ∩ RS(T_j) ≠ ∅ OR Ti ∉ FIN ] THEN RETURN false;

RETURN true;

Is this check too restrictive?
Improving Check($T_j$)

For $T_i \in \text{VAL - IGNORE (T}_j\text{)}$ DO

IF $[\ WS(T_i) \cap RS(T_j) \neq \emptyset \ OR$

$$\left( T_i \notin \text{FIN} \ \text{AND} \ WS(T_i) \cap WS(T_j) \neq \emptyset \right)$$

THEN RETURN false;

RETURN true;
Exercise:

T: $RS(T) = \{A, B\}$
$WS(T) = \{A, C\}$

U: $RS(U) = \{B\}$
$WS(U) = \{D\}$

W: $RS(W) = \{A, D\}$
$WS(W) = \{A, C\}$

V: $RS(V) = \{B\}$
$WS(V) = \{D, E\}$
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