Concurrency Control

Instructor: Matei Zaharia

cs245.stanford.edu
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

Concurrency control + recovery

Beyond serializability
Recap: 2-Phase Locking (2PL)

# locks held by $T_i$

Growing Phase

Shrinking Phase

Time
How Is 2PL Implemented In Practice?

Every system is different, but we’ll show one simplified way
Sample Locking System

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

2. Hold all locks until a transaction commits
Sample Locking System

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

Also need ways to block transactions until locks are available, and to find deadlocks
Optimizing Performance

Beyond the base 2PL protocol, many ways to improve performance & 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:

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

Do not conflict

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

$T_i = \ldots I-S_1(A) \ldots r_1(A) \ldots u_1(A) \ldots$

$T_i = \ldots I-X_1(A) \ldots w_1(A) \ldots u_1(A) \ldots$

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

\[ T_1 = \ldots l \cdot X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots \]
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 replacing S lock with X lock.)
Rule 2: Legal Scheduler

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

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

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

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

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

Lock mode compatibility matrix

\[
\text{compat} = \begin{array}{c|cc}
\text{Lock already held in} & S & X \\
\hline
S & \text{true} & \text{false} \\
X & \text{false} & \text{false} \\
\end{array}
\]
Rule 3: 2PL Transactions

No change except for upgrades: allow upgrades from S to X only in growing phase
Rules 1, 2, 3 $\implies$ Conf. Serializable Schedules for S/X Locks

**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

(3) hierarchical locks
Increment Locks

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

A compatibility matrix is a table that shows compatibility between different types of locks. The table below represents the compatibility of lock types:

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

- **S** (Shared) is compatible with itself and incompatible with exclusive (X) and intent (I) locks.
- **X** (Exclusive) is compatible with itself and incompatible with shared (S) and intent (I) locks.
- **I** (Intent) is compatible with itself and incompatible with shared (S) and exclusive (X) locks.

The new request needs to check against the existing locks to determine if it can be granted.

- **New request**

  - Lock already held in

    The table above shows the compatibility between different lock types. New requests need to check against the existing locks to determine if they can be granted.
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: Lock already held in
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>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!

Lock already held in

Lock already held in
Which Objects Do We Lock?

Table A
Table B

Tuple A
Tuple B
Tuple C

Disk block A
Disk block B

DB
DB
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 |
+-------+-------+-------+-------+
        |       |       |       |
        |       |       |       |
        |       |       |       |
+-------+-------+-------+-------+
        |       |       |       |
        |       |       |       |
+-------+-------+-------+-------+
        |       |       |       |
        |       |       |       |
+-------+-------+-------+-------+

restroom

hall
```
Example
Example

Diagram:

- Root node R1
- Nodes t1, t2, t3, t4
- Arrows:
  - T₁(IS) from t1 to R1
  - T₁(S) from t2 to t1

Diagram shows a tree structure with R1 as the root, t1, t2, t3, and t4 as its children, with directed edges representing dependencies or relationships.
Example

R1

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

\[
\begin{array}{c}
\text{Example 2} \\
\text{CS 245} \\
32
\end{array}
\]
Example 2

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

Diagram:
- R1
- t1
- t2
- t3
- t4
- \( T_1(S) \)
- \( T_2(X) \)
Example 3

$T_1(IS), T_2(S), T_3(IX)$?
## Multiple Granularity Locks

<table>
<thead>
<tr>
<th>Holder</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>
# 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>
Multi-Granularity 2PL Rules

1. Follow multi-granularity compat function
2. Lock root of tree first, any mode
3. Node Q can be locked by $T_i$ in S or IS only if parent(Q) locked by $T_i$ in IX or IS
4. Node Q can be locked by $T_i$ in X, SIX, IX only if parent(Q) locked by $T_i$ in IX, SIX
5. $T_i$ is two-phase
6. $T_i$ can unlock node Q only if none of Q’s children are locked by $T_i$
Exercise:

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

Can $T_2$ access object $f_{2.2}$ 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

A
::
Z
α

→ Insert
Changes to Locking Rules:

1. Need exclusive lock on A to delete A

2. When $T_i$ inserts an object A, $T_i$ receives an exclusive lock on A
Still Have Problem: Phantoms

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

R

<table>
<thead>
<tr>
<th>id</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>
$T_1$: Insert $<12,\text{Mary},\ldots>$ into $R$

$T_2$: Insert $<12,\text{Sam},\ldots>$ into $R$

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l-S_1(o_1)$</td>
<td>$l-S_2(o_1)$</td>
</tr>
<tr>
<td>$l-S_1(o_2)$</td>
<td>$l-S_2(o_2)$</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert $o_3[12,\text{Mary},\ldots]$</td>
<td>Insert $o_4[12,\text{Sam},\ldots]$</td>
</tr>
</tbody>
</table>
Solution

Use multiple granularity tree

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

<table>
<thead>
<tr>
<th>T₁: Insert&lt;12,Mary&gt;</th>
<th>T₂: Insert&lt;12,Sam&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>T₂</td>
</tr>
<tr>
<td>I-X₁(R)</td>
<td>I-X₂(R) delayed</td>
</tr>
</tbody>
</table>

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

I-X₂(R)
Check constraint
Oops! id=12 already in R!
Instead of Locking All of R, Can Lock Ranges of Keys

Example:
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

Concurrency control + recovery

Beyond serializability
Validation Overview

Transactions have 3 phases:

1. Read
   » Read all DB values needed
   » Write to temporary storage
   » No locking

2. Validate
   » Check whether schedule so far is serializable

3. Write
   » If validate OK, write to DB
Key Idea

Make validation atomic

If the validation order is $T_1, T_2, T_3, \ldots$, then the resulting schedule will be conflict equivalent to $S_s = T_1, T_2, T_3, \ldots$.
Implementing Validation

System keeps track of two sets:

FIN = transactions that have finished phase 3 (write phase) and are fully done

VAL = transactions that have successfully finished phase 2 (validation)
Example That Validation Must Prevent:

\[
\begin{align*}
RS(T_2) &= \{B\} & RS(T_3) &= \{A, B\} \\
WS(T_2) &= \{B, D\} & WS(T_3) &= \{C\}
\end{align*}
\]

T_2 \quad start \\
\downarrow \\
T_3 \quad start \\
\downarrow \\
T_2 \quad validated \\
\downarrow \\
T_3 \quad validated \\
\downarrow \\
\text{time}
Example That Validation Must Allow:

\[\text{RS}(T_2) = \{B\} \quad \text{RS}(T_3) = \{A, B\} \neq \emptyset\]

\[\text{WS}(T_2) = \{B, D\} \quad \text{WS}(T_3) = \{C\}\]
Another Thing Validation Must Prevent:

\[
RS(T_2) = \{A\} \quad RS(T_3) = \{A, B\}
\]

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

\[
\text{time} \quad T_2 \quad \text{validated} \quad T_3 \quad \text{validated} \quad \text{finish} \quad T_2
\]
Another Thing Validation Must Prevent:

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

T_2 validated

T_3 validated

BAD: \( w_3(D) \) \( w_2(D) \)
Another Thing Validation Must Allow:

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

when $T_j$ starts phase 1:
  
  ignore($T_j$) $\leftarrow$ FIN

at $T_j$ Validation:
  
  if Check($T_j$) then
    VAL $\leftarrow$ VAL $\cup$ \{ $T_j$\}
    do write phase
    FIN $\leftarrow$ FIN $\cup$ \{ $T_j$\}
Check($T_j$)

for $T_i \in$ VAL – ignore($T_j$) do
    if ($WS(T_i) \cap RS(T_j) \neq \emptyset$ or
        ($T_i \notin$ FIN and $WS(T_i) \cap WS(T_j) \neq \emptyset$))
        then return false
    
return true
Exercise

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

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

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

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

\[\text{start} \quad \square \quad \text{validate} \quad \text{finish}\]
Is Validation $= 2PL$?
$S: w_2(y) \; w_1(x) \; w_2(x)$

Achievable with 2PL?

Achievable with validation?
S: $w_2(y) \ w_1(x) \ w_2(x)$

S can be achieved with 2PL:
$l_2(y) \ w_2(y) \ l_1(x) \ w_1(x) \ u_1(x) \ l_2(x) \ w_2(x) \ u_2(x) \ u_2(y) $

S cannot be achieved by validation:
The validation point of $T_2$, $val_2$, must occur before $w_2(y)$ since transactions do not write to the database until after validation. Because of the conflict on $x$, $val_1 < val_2$, so we must have something like:

S: $val_1 \ val_2 \ w_2(y) \ w_1(x) \ w_2(x)$

With the validation protocol, the writes of $T_2$ should not start until $T_1$ is all done with 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 slide)
Validation Subset of 2PL?

Say S’ not legal (due to w-r conflict):
S’: ... l1(x)  w2(x)  r1(x)  val1  u1(x)  ...
  » At val1: T2 not in Ignore(T1); T2 in VAL
  » T1 does not validate: WS(T2) ∩ RS(T1) ≠ ∅
  » contradiction!

Say S’ not legal (due to w-w conflict):
S’: ... val1  l1(x)  w2(x)  w1(x)  u1(x)  ...
  » Say T2 validates first (proof similar if T1 validates first)
  » At val1: T2 not in Ignore(T1); T2 in VAL
  » T1 does not validate:
    T2 ∉ FIN AND WS(T1) ∩ WS(T2) ≠ ∅
  » contradiction!
Is Validation = 2PL?
When to Use Validation?

Validation performs better than locking when:

» Conflicts are rare
» System resources are plentiful
» Have tight latency constraints
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

Concurrency control + recovery

Beyond serializability
Example:

\[
\begin{align*}
&\text{Commit } T_i \\
&\text{Abort } T_j
\end{align*}
\]

Non-persistent commit (bad!) avoided by recoverable schedules
Concurrency Control & Recovery

Example:

\[ T_j \]

\[ \vdots \]

\[ \text{w}_j(A) \]

\[ \vdots \]

\[ \text{Abort } T_j \]

\[ T_i \]

\[ \vdots \]

\[ \text{r}_i(A) \]

\[ \vdots \]

\[ \text{w}_i(B) \]

\[ \vdots \]

[Commit \( T_i \)]

avoided by avoids-cascading -rollback (ACR) schedules

Cascading rollback (bad!)
Core Problem

Schedule is conflict serializable

$T_j \rightarrow T_i$

But not recoverable
To Resolve This

Need to mark the “final” decision for each transaction in our schedules:

» **Commit decision:** system guarantees transaction will or has completed

» **Abort decision:** system guarantees transaction will or has been rolled back
Model This as 2 New Actions:

\[ c_i = \text{transaction } T_i \text{ commits} \]

\[ a_i = \text{transaction } T_i \text{ aborts} \]
Back to Example

\[ T_j \quad T_i \]
\[ \vdots \quad \vdots \]
\[ w_j(A) \quad r_i(A) \]
\[ \vdots \quad \vdots \]
\[ C_i \leftarrow \text{can we commit here?} \]
Definition

$T_i$ reads from $T_j$ in $S$ ($T_j \Rightarrow_S T_i$) if:

1. $w_j(A) <_S r_i(A)$

2. $a_j \not<_S r(A)$ ($<_S$: does not precede)

3. If $w_j(A) <_S w_k(A) <_S r_i(A)$ then $a_k <_S r_i(A)$
Definition

Schedule S is recoverable if

whenever $T_j \Rightarrow^S T_i$ and $j \neq i$ and $c_i \in S$

then $c_j <^S c_i$
Notes

In all transactions, reads and writes must precede commits or aborts

\[ \iff \text{If } c_i \in T_i, \text{ then } r_i(A) < a_i, w_i(A) < a_i \]

\[ \iff \text{If } a_i \in T_i, \text{ then } r_i(A) < a_i, w_i(A) < a_i \]

Also, just one of \( c_i, a_i \) per transaction
How to Achieve Recoverable Schedules?
With 2PL, Hold Write Locks Until Commit ("Strict 2PL")

$T_j \quad T_i$

$W_j(A) \quad :$

$\vdots \quad \vdots$

$C_j \quad :$

$u_j(A) \quad :$

$\vdots \quad r_i(A)$
With Validation, No Change!

Each transaction’s validation point is its commit point, and only write after
Definitions

S is **recoverable** if each transaction commits only after all transactions from which it read have committed.

S avoids **cascading rollback** if each transaction may read only those values written by committed transactions.

S is **strict** if each transaction may read and write only items previously written by committed transactions (≡ strict 2PL).
Relationship of Recoverable, ACR & Strict Schedules

- Recoverable
- ACR
  - Strict
    - Serial
Examples

Recoverable:

\[ w_1(A) \ w_1(B) \ w_2(A) \ r_2(B) \ c_1 \ c_2 \]

Avoids Cascading Rollback:

\[ w_1(A) \ w_1(B) \ w_2(A) \ c_1 \ r_2(B) \ c_2 \]

Strict:

\[ w_1(A) \ w_1(B) \ c_1 \ w_2(A) \ r_2(B) \ c_2 \]
Recoverability & Serializability

Every strict schedule is serializable

Proof: equivalent to serial schedule based on the order of commit points
  » Only read/write from previously committed transactions
Recoverability & Serializability

Serializable

Serial

Strict

ACR

Recoverable
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

Concurrency control + recovery

Beyond serializability
Weaker Isolation Levels

Dirty reads: Let transactions read values written by other uncommitted transactions
  » Equivalent to having long-duration write locks, but no read locks

Read committed: Can only read values from committed transactions, but they may change
  » Equivalent to having long-duration write locks (X) and short-duration read locks (S)
Weaker Isolation Levels

**Repeatable reads:** Can only read values from committed transactions, and each value will be the same if read again

» Equivalent to having long-duration read & write locks (X/S) but not table locks for insert

Remaining problem: phantoms!
Weaker Isolation Levels

**Snapshot isolation:** Each transaction sees a consistent snapshot of the whole DB (as if we saved all committed values when it began)

» Often implemented with multi-version concurrency control (MVCC)

Still has some anomalies! Example?
Weaker Isolation Levels

**Snapshot isolation:** Each transaction sees a consistent snapshot of the whole DB (as if we saved all committed values when it began)

» Often implemented with multi-version concurrency control (MVCC)

Write skew anomaly: txns write different values

» Constraint: A+B ≥ 0

» T₁: read A, B; if A+B ≥ 1, subtract 1 from A

» T₂: read A, B; if A+B ≥ 1, subtract 1 from B

» Problem: what if we started with A=1, B=0?
Interesting Fact

Oracle calls their snapshot isolation level “serializable”, and doesn’t implement true serializable

Many other systems provide snapshot isolation as an option
  » MySQL, Postgres, MongoDB, SQL Server