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
Which Objects Do We Lock?

Table A
Table B

Tuple A
Tuple B
Tuple C

Disk block A
Disk block B

DB
DB
DB

Idea: Multi-level locking
Example
Example

Diagram:

- **R_1**
  - **t_1**
  - **t_2**
  - **t_3**
  - **t_4**

- **T_1(IS)** from **R_1** to **t_1**
- **T_1(S)** from **t_2** to **t_1**
Example

$T_1(IS), T_2(S)$
Example 2

R₁

- t₁
- t₂
- t₃
- t₄

T₁(IS)

T₁(S)
Example 2

$R_1$ with $t_1$, $t_2$, $t_3$, $t_4$

$T_1(IS)$, $T_2(IX)$

$T_1(S)$, $T_2(X)$
Example 3

$R_1$

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

<table>
<thead>
<tr>
<th>compat</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holder</td>
<td>IS</td>
</tr>
<tr>
<td></td>
<td>IS</td>
</tr>
<tr>
<td></td>
<td>IX</td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>SIX</td>
</tr>
<tr>
<td></td>
<td>X</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><strong>IS</strong></td>
<td>IS, S</td>
</tr>
<tr>
<td><strong>IX</strong></td>
<td>IS, S, IX, X, SIX</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>SIX</strong></td>
<td>X, IX, SIX</td>
</tr>
<tr>
<td><strong>X</strong></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

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

\text{Insert}
Changes to Locking Rules:

1. Get exclusive lock on A before deleting 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

<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>
**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>I-(S₁(o₁))</td>
<td>I-(S₂(o₁))</td>
</tr>
<tr>
<td>I-(S₁(o₂))</td>
<td>I-(S₂(o₂))</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
<tr>
<td>Insert (o₃[12, Mary,..])</td>
<td>Insert (o₄[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

<table>
<thead>
<tr>
<th>$T_1$: Insert&lt;12,Mary&gt;</th>
<th>$T_2$: Insert&lt;12,Sam&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>I-$X_1$(R)</td>
<td>I-$X_2$(R)</td>
</tr>
</tbody>
</table>

Check constraint

Insert<12,Mary>
U$_1$(R)

I-$X_2$(R)

Check constraint

Oops! id=12 already in R!

---

CS 245
Instead of Locking R, Can Use Index Nodes for Ranges

Example:
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 the transaction commits
Sample Locking System

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

Also need good ways to block transactions until locks are available, and to find deadlocks
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 Approach

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 $T_1$, $T_2$, $T_3$, …, then resulting schedule will be conflict equivalent to $S_s = T_1$, $T_2$, $T_3$, …
Implementing Validation

System keeps track of two sets:

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

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

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

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

\text{time}
Example That Validation Must Allow:

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

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

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

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

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

finish \quad T_2

time
Another Thing Validation Must Prevent:

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

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

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

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

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

when $T_j$ starts phase 1:

\[ \text{ignore}(T_j) \leftarrow \text{FIN} \]

at $T_j$ Validation:

\[ \text{if Check}(T_j) \text{ then} \]

\[ \text{VAL} \leftarrow \text{VAL} \cup \{T_j\} \]

\[ \text{do write phase} \]

\[ \text{FIN} \leftarrow \text{FIN} \cup \{T_j\} \]
Check($T_j$)

for $T_i \in$ VAL – ignore($T_j$) do
  if ($WS(T_i) \cap RS(T_j) \neq \emptyset$ or $\left( T_i \notin FIN \text{ and } WS(T_i) \cap WS(T_j) \neq \emptyset \right)$)
    then return false

return true
Exercise

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

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

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

V: \( RS(V) = \{B\} \)
\( WS(V) = \{D,E\} \)
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
Concurrency Control & Recovery

Example:

\[ T_j \]
\[ w_j(A) \]
\[ \vdots \]
\[ \text{Abort } T_j \]

\[ T_i \]
\[ r_i(A) \]
\[ \vdots \]
\[ \text{Commit } T_i \]

Non-persistent commit (bad!)

avoided by recoverable schedules
Concurrent Control & Recovery

Example:

\[ T_j \]
\[ w_j(A) \]
\[ r_i(A) \]
\[ w_i(B) \]

Abort \( T_j \)

\[ T_i \]

[Commit \( T_i \)]

Cascading rollback (bad!)

Avoided by avoids-cascading -rollback (ACR) schedules
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

\[ T_j \quad T_i \quad T_i \]
\[ \vdots \quad \vdots \quad \vdots \]
\[ w_j(A) \quad r_i(A) \]
\[ \vdots \quad \vdots \quad \vdots \]
\[ C_i \quad \leftarrow \text{can we commit here?} \]
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. $a_j \not<_S r(A)$ \quad ($<_S$: 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

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

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

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

\[
\begin{array}{c}
T_j & T_i \\
W_j(A) & \vdots \\
\vdots & \vdots \\
C_j & \vdots \\
u_j(A) & \vdots \\
\vdots & r_i(A)
\end{array}
\]
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
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

**Repeatabile 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: $A+B \geq 0$

» $T_1$: read $A$, $B$; if $A+B \geq 1$, subtract 1 from $A$

» $T_2$: read $A$, $B$; if $A+B \geq 1$, 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 provide serializable

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