Transactions and Locking

April 16, 2018
Schedule
An ordering of read and write operations.

Serial Schedule
No interleaving between transactions at all

Serializable Schedule
Guaranteed to produce equivalent output to a serial schedule
Conflict Equivalence

Possible Solution: Look at read/write, etc… conflicts!

Allow operations to be reordered as long as conflicts are ordered the same way

Conflict Equivalence: Can reorder one schedule into another without reordering conflicts.

Conflict Serializability: Conflict Equivalent to a serial schedule.
Example

Time

T1  T2  T3

R(A)

W(A)

W(A)

W(A)
Example

Time

T1  T2  T3

R(A)  W(A)  W(A)

Write order irrelevant (T3 overwrites either way)
View Serializability

**Possible Solution**: Look at data flow!

**View Equivalence**: All reads read from the same writer
Final write in a batch comes from the same writer

**View Serializability**: View Equivalent to a serial schedule.
How to detect conflict serializable schedule?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W(a)</td>
<td>R(b)</td>
<td>W(d)</td>
</tr>
<tr>
<td></td>
<td>W(b)</td>
<td>R(d)</td>
<td>W(d)</td>
</tr>
</tbody>
</table>

Precedence Graph

T₁ → T₂ → T₃

Cycle!
Not Conflict serializable

It is not conflict serializable because the precedence graph has a cycle. It can not be strict 2PL because T₂ will have to unlock(B) at the very end and hence it will be impossible for T₁ to write(B). Therefore, it is not serializable because of a cycle in the precedence graph.
Not conflict serializable but view serializable

Satisfies 3 conditions of view serializability

Every view serializable schedule which is not conflict serializable has blind writes.
How can conflicts be avoided?
How can conflicts be avoided?

- Optimistic Concurrency Control
- Conservative Concurrency Control
How can conflicts be avoided?

- Optimistic Concurrency Control
- Conservative Concurrency Control
Conservative Concurrency Control

- How can bad schedules be detected?

- What problems does each approach introduce?

- How do we resolve these problems?
Two-Phase Locking

- Phase 1: Acquire (do not release) locks.
- Phase 2: Release (do not acquire) locks.
  Why?
Two-Phase Locking

• Phase 1: Acquire (do not release) locks.

• Phase 2: Release (do not acquire) locks.

Why?

Can we do even better?
Example

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Acyclic - Conflict Serializable
2PL exists
## Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(a)</td>
<td>L(b)</td>
<td>L(d)</td>
</tr>
<tr>
<td>W(a)</td>
<td>R(b)</td>
<td>R(d)</td>
</tr>
<tr>
<td></td>
<td>W(d)</td>
<td>L(d)</td>
</tr>
<tr>
<td></td>
<td>R-L(b)</td>
<td>R-L(d)</td>
</tr>
<tr>
<td>L(b)</td>
<td>L(d)</td>
<td>R(d)</td>
</tr>
<tr>
<td>R-L(a)</td>
<td>R-L(b)</td>
<td>R-L(d)</td>
</tr>
<tr>
<td>L(b)</td>
<td>W(b)</td>
<td>R(d)</td>
</tr>
<tr>
<td>R-L(b)</td>
<td></td>
<td>R-L(d)</td>
</tr>
</tbody>
</table>
Need for shared and exclusive locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(a)</td>
<td>L(b)</td>
<td>L(d)</td>
</tr>
<tr>
<td>W(a)</td>
<td>L(b)</td>
<td>R(d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W(d)</td>
</tr>
</tbody>
</table>

Precedence Graph

It is conflict Serializable but requires granular control of locks
# Need for shared and exclusive locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SL(d)</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R(d)</td>
<td>X</td>
</tr>
<tr>
<td>XL(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SL(b)</td>
<td>SL(d)</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>R(b)</td>
<td>R-SL(b)</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>XL(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(b)</td>
<td>R-XL(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-SL(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XL(d)</td>
<td>W(d)</td>
<td>R-XL(d)</td>
<td></td>
</tr>
</tbody>
</table>
Reader/Writer (S/X)

- When accessing a DB Entity...
  - Table, Row, Column, Cell, etc...

- Before reading: Acquire a Shared (S) lock.
  - Any number of transactions can hold S.

- Before writing: Acquire an Exclusive (X) lock.
  - If a transaction holds an X, no other transaction can hold an S or X.
What do we lock?

Is it safe to allow some transactions to lock tables while other transactions to lock tuples?
New Lock Modes

Even within the same application, there may be a need for locks at multiple levels of granularity. Database elements are organized in a hierarchy:

- Relations
- Blocks
- Tuples

```
relations
blocks
  B1  B2  B3  B4
  t1  t2  t3  t4  t5
```

contained in
Hierarchical Locks

• Lock Objects Top-Down

• Before acquiring a lock on an object, an xact must have at least an intention lock on its parent!

• For example:

• To acquire a S on an object, an xact must have an IS, IX on the object’s parent (why not S, SIX, or X?)

• To acquire an X (or SIX) on an object, an xact must have a SIX, or IX on the object’s parent.
## New Lock Modes

**Lock Mode(s) Currently Held By Other Xacts**

<table>
<thead>
<tr>
<th>Lock Mode Desired</th>
<th>None</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
</tr>
<tr>
<td>IS</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
</tr>
<tr>
<td>IX</td>
<td>valid</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>S</td>
<td>valid</td>
<td>valid</td>
<td>fail</td>
<td>valid</td>
<td>fail</td>
</tr>
<tr>
<td>X</td>
<td>valid</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
<td>fail</td>
</tr>
</tbody>
</table>
Example

• An I lock for a super-element constrains the locks that the same transaction can obtain at a subelement.

• If Ti has locked the parent element P in IS, then Ti can lock child element C in IS, S.

• If Ti has locked the parent element P in IX, then Ti can lock child element C in IS, S, IX, X.
Example

• T1 wants exclusive lock on tuple t2
Example

• T2 wants to request an X lock on tuple t3
Example

T2 wants to request an S lock on block B2.

T1(IX)  \(\rightarrow\)  T2(IS)

\(\rightarrow\)

B1

T1(IX)

T1(X)

\(t_2\)

\(t_3\)

\(t_4\)

\(t_5\)

B2

T2(S)

not granted!

B3

B4
Deadlocks

• Deadlock: A cycle of transactions waiting on each other’s locks
  • Problem in 2PL; xact can’t release a lock until it completes
• How do we handle deadlocks?
  • **Anticipate**: Prevent deadlocks before they happen.
  • **Detect**: Identify deadlock situations and abort one of the deadlocked xacts.
Deadlock Detection

- **Baseline**: If a lock request cannot be satisfied, the transaction is blocked and must wait until the resource is available.

- Create a waits-for graph:
  - Nodes are transactions
  - Edge from $T_i$ to $T_k$ if $T_i$ is waiting for $T_k$ to release a lock.
  - Periodically check for cycles in the graph.
Example

\[ T_1: \ l_1(A); \ r_1(A); \ A := A+100; \ w_1(A); \ l_1(B); \ u_1(A); \ r_1(B); \ B := B+100; \ w_1(B); \ u_1(B); \]

\[ T_2: \ l_2(B); \ r_2(B); \ B := B*2; \ w_2(B); \ l_2(A); \ u_2(B); \ r_2(A); \ A := A*2; \ w_2(A); \ u_2(A); \]

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( A )</th>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_1(A); \ r_1(A); )</td>
<td>( l_2(B); \ r_2(B); )</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>( A := A+100; )</td>
<td>( B := B*2; )</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>( w_1(A); )</td>
<td>( w_2(B); )</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>( l_1(B) ) Denied</td>
<td>( l_2(A) ) Denied</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram

- **Time**
- **T1**: 
  - S(A)
  - R(A)
- **T2**: 
  - X(B)
  - W(B)
- **T3**: 
  - S(C)
  - R(C)
- **T4**: 
  - X(B)
  - X(A)
Example

Time T1 | T2 | T3 | T4
---|---|---|---
S(A) | R(A) | X(B) | W(B)
S(B) | X(C) | S(C) | R(C)
X(A) | X(B) | X(C)
Example

Time T1  T2  T3  T4

- S(A)  R(A)
- X(B)  W(B)
- S(B)
- X(C)  S(C)  R(C)
- X(B)  X(A)
Example

Time T1       T2       T3       T4

S(A)          R(A)      X(B)      W(B)
S(B)          S(C)      R(C)      X(C)

T1 --> T2
T4 --> T3
Example

Time $T_1$  $T_2$  $T_3$  $T_4$

$S(A)$
$R(A)$

$X(B)$
$W(B)$

$S(B)$

$X(C)$
$S(C)$
$R(C)$

$X(C)$

$X(B)$

$X(A)$

$T_1 ightarrow T_2$
$T_4 ightarrow T_3$
Example

Time T1  T2  T3  T4

S(A)  R(A)

X(B)  W(B)

S(B)

X(C)  S(C)  R(C)

T1 → T2

T4 → T3

X(A) → X(B) → X(A)
Example

Time \( T_1 \)  \( T_2 \)  \( T_3 \)  \( T_4 \)

- \( S(A) \)
- \( R(A) \)
- \( X(B) \)
- \( W(B) \)
- \( S(B) \)
- \( S(C) \)
- \( R(C) \)
- \( X(C) \)

\( X(B) \)
\( X(A) \)
Example

Time T1 T2 T3 T4

S(A) R(A)  
X(B) W(B)  
S(B)     
X(C)     

S(C) R(C)  
X(B)      
X(A)
Handling Deadlocks

Approach 1
Avoid getting into deadlocks

Approach 2
Detect (and fix) deadlocks after they occur
Avoiding Deadlocks

**Approach:** Require transactions to follow an invariant that is guaranteed to be deadlock free.
Avoiding Deadlocks

Example: Give each Lock an ID #. Only allow locks to be acquired in order of their ID.
Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td>X(B)</td>
<td>S(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(B)</td>
<td></td>
<td></td>
<td>X(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X(A)</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td>X(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(B)</td>
<td>X(C)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Out of Order (T3 is not valid)

- X(A)
- X(B)
Avoiding Deadlock

**Alternative:** Acquire all locks at the start.
Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td>X(B)</td>
<td>S(C)</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>X(B)</td>
<td>W(B)</td>
<td>R(C)</td>
</tr>
<tr>
<td></td>
<td>S(B)</td>
<td>S(B)</td>
<td>S(C)</td>
<td>X(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Avoiding Deadlocks

**Pro:** No Deadlocks… Ever

**Con:** Not all transactions are supported.

or

**Con:** Transactions need to maintain all locks that might possibly ever be required at all times.
Handling Deadlocks

Approach 1
Avoid getting into deadlocks

Approach 2
Detect (and fix) deadlocks after they occur
Deadlock Detection

- **Baseline**: If a lock request cannot be satisfied, the transaction is blocked and must wait until the resource is available.

- Create a waits-for graph:
  - Nodes are transactions
  - Edge from $T_i$ to $T_k$ if $T_i$ is waiting for $T_k$ to release a lock.
  - Periodically check for cycles in the graph.
Example

Time T1  T2  T3  T4

S(A)
R(A)

X(B)
W(B)

S(B)

S(C)
R(C)

X(C)

X(B)

X(A)
Example

Time T1 T2 T3 T4

S(A) R(A)
X(B) W(B)
S(B)
X(C) R(C)
X(A) X(B)

T1 → T2
T4 → T3
Example
Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(B)</td>
<td>W(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- S(A)
- R(A)
- X(B)
- W(B)
- S(B)
- X(C)
- S(C)
- R(C)
- X(C)
- X(A)
- X(B)

Diagram:

- T1 → T2
- T4 → T3
- T1 → T4
Example

Time $T_1$ $T_2$ $T_3$ $T_4$

$S(A)$
$R(A)$

$X(B)$
$W(B)$

$S(B)$

$S(C)$
$R(C)$

$X(C)$

$T_1$ $T_2$ $T_3$ $T_4$

$X(A)$

$X(B)$
Example

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(C)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deadlock Detection

What happens when a deadlock is detected?
GAME OF DEADLOCKS
YOU WIN OR YOU DIE
(and get restarted)
Deadlock Detection

**Default:** Kill as many deadlocked transactions as needed. (killed transactions may be restarted or “replayed”)

**Optional:** App-specific recovery logic
Detecting Deadlocks

**Pro:** No limitations on transactions

**Pro:** Best-case is faster than upfront acquisition

**Con:** Worst-case is much much much slower.

**Con:** Cycle detection is slow and expensive
Detecting Deadlocks

**Pro:** No limitations on transactions
**Pro:** Best-case is faster than upfront acquisition

**Con:** Worst-case is much much much slower.
**Con:** Cycle detection is slow and expensive
Simpler Detection Schemes

**Approach:** Accept false positives for faster deadlock detection
Simpler Detection Schemes

- **Trivial Solution**: Time-outs.

- **Invariant-Based Solution**: Enforce monotonicity property about which transactions are allowed to block which transactions.
Simpler Detection Scheme 1

Intuition: Never block on an ‘younger’ transaction
Simpler Detection Scheme 1

**Intuition**: Never block on an ‘younger’ transaction
Simpler Detection Scheme 1

Intuition: Never block on an ‘younger’ transaction

T2 → T1
Simpler Detection Scheme 1

Intuition: Never block on an ‘younger’ transaction
Simpler Detection Scheme 1

Intuition: Never block on an ‘younger’ transaction
Simpler Detection Scheme 1

Intuition: Never block on an ‘younger’ transaction
Simpler Detection Scheme 1

**Intuition**: Never block on an ‘younger’ transaction
Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)
Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)

the invariant is preserved

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)
Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)
the invariant is preserved

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)
avoid deadlock by killing T1
Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)
the invariant is preserved

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)
avoid deadlock by killing T1

“Wait-Die”
Simpler Detection Scheme 2

**Intuition**: Never block on a ‘younger’ transaction
Simpler Detection Scheme 2

T1

**Intuition**: Never block on a ‘younger’ transaction
Simpler Detection Scheme 2

Intuition: Never block on a ‘younger’ transaction
Simpler Detection Scheme 2

Intuition: Never block on a ‘younger’ transaction
Simpler Detection Scheme 2

Intuition: Never block on a ‘younger’ transaction
Simpler Detection Scheme 2

Intuition: Never block on a ‘younger’ transaction
Simpler Detection Scheme 2

Intuition: Never block on a ‘younger’ transaction
Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)
Simpler Detection Scheme 1

\begin{itemize}
\item $T_1$ holds a lock on $A$
\item $T_2$ tries to acquire the lock on $A$ (and would block)
\item the invariant is preserved
\item $T_2$ holds a lock on $A$
\item $T_1$ tries to acquire the lock on $A$ (and would block)
\end{itemize}
Simpler Detection Scheme 1

*T1 holds a lock on A*
*T2 tries to acquire the lock on A (and would block)*

the invariant is preserved

*T2 holds a lock on A*
*T1 tries to acquire the lock on A (and would block)*

avoid deadlock by killing T2 and giving T1 the lock
Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)
the invariant is preserved

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)
avoid deadlock by killing T2 and giving T1 the lock

“Wound-Wait”
Which transaction?

**Policy 1**: Wait-Die

“Those in power stay in power”

Blocking Xact Dies

**Policy 2**: Wound-Wait

“Take everything you can”

Blocking Xact Kills Other
Simpler Detection Schemes

**Preserve fairness**: A killed transaction is restarted with the same timestamp
Managing Deadlocks

- Approach 1: Avoidance
  - Invariant on lock acquisition order.
  - Acquire all locks upfront.

- Approach 2: Recovery
  - Detect cycles (or conditions that indicate cycles)
  - Kill/Restart transactions until there are no cycles.