Transactional Information Systems:

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Gerhard Weikum and Gottfried Vossen

© 2002 Morgan Kaufmann
ISBN 1-55860-508-8

“Teamwork is essential. It allows you to blame someone else.” (Anonymous)
Part II: Concurrency Control

• 3 Concurrency Control: Notions of Correctness for the Page Model
• 4 Concurrency Control Algorithms
• 5 Multiversion Concurrency Control
• 6 Concurrency Control on Objects: Notions of Correctness
• 7 Concurrency Control Algorithms on Objects
• 8 Concurrency Control on Relational Databases
• 9 Concurrency Control on Search Structures
• 10 Implementation and Pragmatic Issues
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned

“The optimist believes we live in the best of all possible worlds. The pessimist fears this is true.” (Robert Oppenheimer)
Transaction Scheduler

Clients

Requests

Data Server

Transaction Manager (TM)

Data Manager (DM)

Database
Definition 4.1 (CSR Safety):
For a scheduler S, Gen(S) denotes the set of all schedules that S can generate. A scheduler is called **CSR safe** if Gen(S) ⊆ CSR.
Scheduler Classification

Concurrency control protocols

- Pessimistic
  - Non-locking
    - TO
    - SGT
  - Locking
    - Two-phase
      - AL
      - O2PL
    - Non-two-phase
      - WTL
      - RWTL
      - 2PL
        - C2PL
        - S2PL
        - SS2PL
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
  • 4.3.1 Introduction
  • 4.3.2 Two-Phase Locking (2PL)
  • 4.3.3 Deadlock Handling
  • 4.3.4 Variants of 2PL
  • 4.3.5 Ordered Sharing of Locks (O2PL)
  • 4.3.6 Altruistic Locking (AL)
  • 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  • 4.3.8 Geometry of Locking
• 4.4 Non-Locking Schedulers
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned
General Locking Rules

For each step the scheduler **requests a lock** on behalf of the step’s transaction. Each lock is requested in a specific **mode** (read or write).

If the data item is not yet locked in an **incompatible mode** the lock is granted; otherwise there is a **lock conflict** and the transaction becomes **blocked** (suffers a **lock wait**) until the current lock holder **releases the lock**.

### Compatibility of locks:

<table>
<thead>
<tr>
<th>lock holder</th>
<th>rl_j(x)</th>
<th>wl_j(x)</th>
<th>lock requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>rl_i(x)</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>wl_i(x)</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

### General locking rules:

**LR1**: Each data operation $o_i(x)$ must be preceded by $ol_i(x)$ and followed by $ou_i(x)$.  

**LR2**: For each $x$ and $t_i$ there is at most one $ol_i(x)$ and at most one $ou_i(x)$.  

**LR3**: No $ol_i(x)$ or $ou_i(x)$ is redundant.  

**LR4**: If $x$ is locked by both $t_i$ and $t_j$, then these locks are compatible.
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
  • 4.3.1 Introduction
  • 4.3.2 Two-Phase Locking (2PL)
    • 4.3.3 Deadlock Handling
    • 4.3.4 Variants of 2PL
    • 4.3.5 Ordered Sharing of Locks (O2PL)
    • 4.3.6 Altruistic Locking (AL)
    • 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
    • 4.3.8 Geometry of Locking
• 4.4 Non-Locking Schedulers
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned
**Definition 4.2 (2PL):**
A locking protocol is **two-phase (2PL)** if for every output schedule $s$ and every transaction $t_i \in \text{trans}(s)$ no $\text{ql}_i$ step follows the first $\text{ou}_i$ step ($q, o \in \{r, w\}$).

**Example 4.4:**

$s = w_1(x) \ r_2(x) \ w_1(y) \ w_1(z) \ r_3(z) \ c_1 \ w_2(y) \ w_3(y) \ c_2 \ w_3(z) \ c_3$

\[ t_1 \]
\[ w_1(x) \quad w_1(y) \quad w_1(z) \]
\[ t_2 \]
\[ r_2(x) \quad w_2(y) \]
\[ t_3 \]
\[ r_3(z) \quad w_3(y) \quad w_3(z) \]

\[ w_1(x) \quad w_1(x) \quad w_1(y) \quad w_1(y) \quad w_1(z) \quad w_1(z) \quad w_1(z) \quad w_1(z) \quad w_1(z) \quad w_1(z) \quad c_1 \]
\[ r_2(x) \quad r_2(x) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad w_2(y) \quad c_2 \]
\[ r_3(z) \quad r_3(z) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad w_3(y) \quad c_3 \]
Correctness and Properties of 2PL

Theorem 4.1:
Gen(2PL) ⊂ CSR (i.e., 2PL is CSR-safe).

Example 4.5:
\[ s = w_1(x) \, r_2(x) \, c_2 \, r_3(y) \, c_3 \, w_1(y) \, c_1 \in CSR \]
but \( \not\in \) Gen(2PL) for \( w_u1(x) < rl_2(x) \) and \( ru_3(y) < wl_1(y) \),
\[ rl_2(x) < r_2(x) \) and \( r_3(y) < ru_3(y) \), and \( r_2(x) < r_3(y) \)
would imply \( w_u1(x) < wl_1(y) \) which contradicts the two-phase property.

Theorem 4.2:
Gen(2PL) ⊂ OCSR

Example:
\[ w_1(x) \, r_2(x) \, r_3(y) \, r_2(z) \, w_1(y) \, c_3 \, c_1 \, c_2 \]
Theorem 4.2 \((\text{Gen}(2\text{PL}) \subset \text{OCSR})\)

- \text{Gen}(2\text{PL}) \text{ contains data and termination operations of committed transactions.}
- Strict inclusion by example. Why inclusion?
- \(s\) in \text{Gen}(2\text{PL}). Consider \(G(s)\).
- \text{An edge } T_i \rightarrow T_j \text{ implies conflicting operation } o_i \text{ and } o_j \text{ such that } T_i \text{ unlocked its 1}\text{st item prior to } T_j \text{ locking some item.}
- \text{Augment with new edges from } T_i \text{ to } T_j \text{ if all operations of } T_i \text{ precede in } s \text{ all operations of } T_j.
- \text{Such an edge means } T_i \text{ unlocked its 1}\text{st item prior to } T_j \text{ locking some item.}
- \text{We claim there’s no cycle in the augmented graph and its topological sorting provides } s'.
  - \text{Proof of claim: Suppose there is a cycle } T_1,\ldots,T_n,T_1.
  - \text{So } T_1 \text{ unlocked 1}\text{st item} < T_2 \text{ locked} < T_2 \text{ unlocked 1}\text{st item} < \ldots < T_{n-1} \text{ unlocked 1}\text{st item} < T_n \text{ locked an item} < T_n \text{ unlocked 1}\text{st item} < T_1 \text{ locked}
  - \text{So } T_1 \text{ unlocked an item and later on locked an item, not 2 phase!}
Proof of 2PL Correctness (covered already)

Let $s$ be the output of a 2PL scheduler, and let $G$ be the conflict graph of $\text{CP (DT}(s))$ where $\text{DT}$ is the projection onto data and termination operations and $\text{CP}$ is the committed projection.

The following holds (Lemma 4.2):

(i) If $(t_i, t_j)$ is an edge in $G$, then $p_{u_i}(x) < q_{l_j}(x)$ for some $x$ with conflicting $p, q$.

(ii) If $(t_1, t_2, ..., t_n)$ is a path in $G$, then $p_{u_1}(x) < q_{l_n}(y)$ for some $x, y$.

(iii) $G$ is acyclic.

This can be shown as follows:

(i) By locking rules LR1 through LR4.

(ii) By induction on $n$.

(iii) Assume $G$ has a cycle of the form $(t_1, t_2, ..., t_n, t_1)$.

   By (ii), $p_{u_1}(x) < q_{l_1}(y)$ for some $x, y$,
   which contradicts the two-phase property.
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
  - 4.3.1 Introduction
  - 4.3.2 Two-Phase Locking (2PL)
  - 4.3.3 Deadlock Handling
    - 4.3.4 Variants of 2PL
    - 4.3.5 Ordered Sharing of Locks (O2PL)
    - 4.3.6 Altruistic Locking (AL)
    - 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
    - 4.3.8 Geometry of Locking
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
Deadlock Detection

Deadlocks are caused by cyclic lock waits (e.g., in conjunction with lock conversions).

Example:

\[ \begin{align*}
  &t_1 \quad r_1(x) \quad w_1(y) \\
  &w_2(y) \quad w_2(x) \\
  &t_2
\end{align*} \]

**Deadlock detection:**

(i) Maintain dynamic **waits-for graph (WFG)** with active transactions as nodes and an edge from \( t_i \) to \( t_j \) if \( t_j \) waits for a lock held by \( t_i \).

(ii) Test WFG for cycles

- continuously (i.e., upon each lock wait) or
- periodically.
Deadlock Resolution

Choose a transaction on a WFG cycle as a **deadlock victim** and abort this transaction, and repeat until no more cycles.

**Possible victim selection strategies:**
1. Last blocked
2. Random
3. Youngest
4. Minimum locks
5. Minimum work
6. Most cycles
7. Most edges
Illustration of Victim Selection Strategies

Example WFG:

Most-cycles strategy would select $t_1$ (or $t_3$) to break all 5 cycles.

Example WFG:

Most-edges strategy would select $t_1$ to remove 4 edges.
Deadlock Prevention

Restrict lock waits to ensure **acyclic WFG** at all times.

### Reasonable deadlock prevention strategies:

1. **Wait-die**: old waits for young
   
   upon $t_i$ blocked by $t_j$:
   
   if $t_i$ started before $t_j$ then wait else abort $t_i$

2. **Wound-wait**: young waits for old
   
   upon $t_i$ blocked by $t_j$:
   
   if $t_i$ started before $t_j$ then abort $t_j$ else wait

3. **Immediate restart**:
   
   upon $t_i$ blocked by $t_j$: abort $t_i$

4. **Running priority**:
   
   upon $t_i$ blocked by $t_j$:
   
   if $t_j$ is itself blocked then abort $t_j$ else wait

5. **Timeout**:
   
   abort waiting transaction when a timer expires

Abort entails later restart.
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
  - 4.3.1 Introduction
  - 4.3.2 Two-Phase Locking (2PL)
  - 4.3.3 Deadlock Handling
- 4.3.4 Variants of 2PL
  - 4.3.5 Ordered Sharing of Locks (O2PL)
  - 4.3.6 Altruistic Locking (AL)
  - 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  - 4.3.8 Geometry of Locking
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
Definition 4.3 (Conservative 2PL):
Under static or conservative 2PL (C2PL) each transaction acquires all its locks before the first data operation (preclaiming).

Definition 4.4 (Strict 2PL):
Under strict 2PL (S2PL) each transaction holds all its write locks until the transaction terminates.

Definition 4.5 (Strong 2PL):
Under strong 2PL (SS2PL) each transaction holds all its locks (i.e., both r and w) until the transaction terminates.
Properties of S2PL and SS2PL

Theorem 4.3:
Gen(SS2PL) ⊂ Gen(S2PL) ⊂ Gen(2PL)

Theorem 4.4:
Gen(SS2PL) ⊂ COCSR
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
  • 4.3.1 Introduction
  • 4.3.2 Two-Phase Locking (2PL)
  • 4.3.3 Deadlock Handling
  • 4.3.4 Variants of 2PL
  • 4.3.5 Ordered Sharing of Locks (O2PL)
    • 4.3.6 Altruistic Locking (AL)
    • 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
    • 4.3.8 Geometry of Locking
• 4.4 Non-Locking Schedulers
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned
Ordered Sharing of Locks

**Motivation:**
Example 4.6:
\[ s_1 = w_1(x) \ r_2(x) \ r_3(y) \ c_3 \ w_1(y) \ c_1 \ w_2(z) \ c_2 \]
\[ \in \text{COCSR}, \text{ but } \notin \text{Gen(2PL)} \]

**Observation:**
the schedule were feasible if **write locks could be shared**
s.t. the order of lock acquisitions dictates the order of data operations

**Notation:**
\[ pl_i(x) \rightarrow ql_j(x) \text{ (with } i \neq j) \text{ for } pl_i(x) <_s ql_j(x) \land p_i(x) <_s q_j(x) \]

Example reconsidered with ordered sharing of locks:
\[ wl_1(x) \ w_1(x) \ r_2(x) \ r_2(x) \ r_3(y) \ r_3(y) \ ru_3(y) \ c_3 \]
\[ wl_1(y) \ w_1(y) \ wu_1(x) \ wu_1(y) \ c_1 \ wl_2(z) \ w_2(z) \ ru_2(x) \ wu_2(z) \ c_2 \]
### Lock Compatibility Tables With Ordered Sharing

<table>
<thead>
<tr>
<th>$LT_1$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>_</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_2$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>→</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_3$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>_</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>→</td>
<td>_</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_4$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>_</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>_</td>
<td>→</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_5$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>→</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>→</td>
<td>_</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_6$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>_</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>→</td>
<td>→</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_7$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>→</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>_</td>
<td>→</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$LT_8$</th>
<th>$rl_i(x)$</th>
<th>$wl_i(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rl_i(x)$</td>
<td>+</td>
<td>→</td>
</tr>
<tr>
<td>$wl_i(x)$</td>
<td>→</td>
<td>→</td>
</tr>
</tbody>
</table>
**Additional Locking Rules for O2PL**

**OS1 (lock acquisition):**
Assuming that $pl_i(x) \rightarrow ql_j(x)$ is permitted, if $pl_i(x) <_s ql_j(x)$ then $p_i(x) <_s q_j(x)$ must hold.

**Example:**

$wl_1(x) \ w_1(x) \ wl_2(x) \ w_2(x) \ wl_2(y) \ w_2(y) \ wu_2(x) \ wu_2(y) \ c_2$

$wl_1(y) \ w_1(y) \ wu_1(x) \ wu_1(y) \ c_1$

Satisfies OS1, LR1 – LR4, is two-phase, but $\not\in$ CSR

**OS2 (lock release):**
If $pl_i(x) \rightarrow ql_j(x)$ and $t_i$ has not yet released any lock, then $t_j$ is order-dependent on $t_i$. If such $t_i$ exists, then $t_j$ is on hold. While a transaction is on hold, it must not release any locks.

**O2PL:** locking with rules LR1 - LR4, two-phase property, rules OS1 - OS2, and lock table $LT_8$
Example 4.7:

\[ s = r_1(x) \; w_2(x) \; r_3(y) \; w_2(y) \; c_2 \; w_3(z) \; c_3 \; r_1(z) \; c_1 \]

\[ rl_1(x) \; r_1(x) \; w_2(x) \; w_2(x) \; rl_3(y) \; r_3(y) \; w_2(y) \; w_2(y) \]
\[ w_3(z) \; w_3(z) \; ru_3(y) \; wu_3(z) \; c_3 \; rl_1(z) \; r_1(z) \; ru_1(x) \; ru_1(z) \; wu_2(x) \; wu_2(y) \; c_2 \; c_1 \]
Theorem 4.5:
Let $LT_i$ denote the locking protocol with ordered sharing according to lock compatibility table $LT_i$.
For each $i$, $1 \leq i \leq 8$, $\text{Gen}(LT_i) \subseteq \text{CSR}$.

Theorem 4.6:
$\text{Gen}(O2PL) \subseteq \text{OCSR}$

Theorem 4.7:
$\text{OCSR} \subseteq \text{Gen}(O2PL)$

Corollary 4.1:
$\text{Gen}(O2PL) = \text{OCSR}$
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- **4.3 Locking Schedulers**
  - 4.3.1 Introduction
  - 4.3.2 Two-Phase Locking (2PL)
  - 4.3.3 Deadlock Handling
  - 4.3.4 Variants of 2PL
  - 4.3.5 Ordered Sharing of Locks (O2PL)
  - **4.3.6 Altruistic Locking (AL)**
  - 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  - 4.3.8 Geometry of Locking
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
**Altruistic Locking (AL)**

**Motivation:**

**Example 4.8:** concurrent executions of

\[ t_1 = w_1(a) \, w_1(b) \, w_1(c) \, w_1(d) \, w_1(e) \, w_1(f) \, w_1(g) \]

\[ t_2 = r_2(a) \, r_2(b) \]

\[ t_3 = r_3(c) \, r_3(e) \]

**Observations:**

- \( t_2 \) and \( t_3 \) access subsets of the data items accessed by \( t_1 \)
- \( t_1 \) knows when it is “finished” with a data item
- \( t_1 \) could “pass over” locks on specific data items to transactions that access only data items that \( t_1 \) is finished with
  (such transactions are “in the wake” of \( t_1 \))

**Notation:**

\( d_i(x) \) for \( t_i \) donating its lock on \( x \) to other transactions

**Example with donation of locks:**

\( w_{l_1}(a) \, w_1(a) \, d_1(a) \, r_{l_2}(a) \, r_2(a) \, w_{l_1}(b) \, w_1(b) \, d_1(b) \, r_{l_2}(b) \, r_2(b) \, w_{l_1}(c) \, w_1(c) \, \ldots \)

... \( r_{u_2}(a) \, r_{u_2}(b) \, \ldots \) \( w_{u_1}(a) \, w_{u_1}(b) \, w_{u_1}(c) \, \ldots \)
Additional Locking Rules for AL

**AL1:** Once $t_i$ has donated a lock on $x$, it can no longer access $x$.

**AL2:** After $t_i$ has donated a lock $x$, $t_i$ must eventually unlock $x$.

**AL3:** $t_i$ and $t_j$ can simultaneously hold conflicting locks only if $t_i$ has donated its lock on $x$.

**Definition 4.27:**

(i) $p_j(x)$ is *in the wake* of $t_i$ ($i \neq j$) in $s$ if $d_i(x) \prec_s p_j(x) \prec_s o_{ui}(x)$.

(ii) $t_j$ is in the wake of $t_i$ if some operation of $t_j$ is in the wake of $t_i$.

(iii) $t_j$ is completely in the wake of $t_i$ if all its operations are in the wake of $t_i$.

**AL4:** When $t_j$ is indebted to $t_i$,

$t_j$ must remain completely in the wake of $t_i$.

**AL:** locking with rules LR1 - LR4, two-phase property, donations, and rules AL1 - AL4.
Example:
rl₁(a) r₁(a) d₁(a) wl₃(a) w₃(a) wu₃(a) c₃
rl₂(a) r₂(a) wl₂(b) ru₂(a) w₂(b) wu₂(b) c₂ rl₁(b) r₁(b) ru₁(a) ru₁(b) c₁
→ disallowed by AL (even $\notin$ CSR)

Example corrected using rules AL1 - AL4:
rl₁(a) r₁(a) d₁(a) wl₃(a) w₃(a) wu₃(a) c₃
rl₂(a) r₂(a) rl₁(b) r₁(b) ru₁(a) ru₁(b) c₁ wl₂(b) ru₂(a) w₂(b) wu₂(b) c₂
→ admitted by AL ($t₂$ stays completely in the wake of $t₁$)
Correctness and Properties of AL

Theorem 4.8:
Gen(2PL) ⊂ Gen(AL).

Theorem 4.9:
Gen(AL) ⊂ CSR

Example:
s = r_1(x) r_2(z) r_3(z) w_2(x) c_2 w_3(y) c_3 r_1(y) r_1(z) c_1
→ ∈ CSR, but ∉ Gen(AL)
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
  • 4.3.1 Introduction
  • 4.3.2 Two-Phase Locking (2PL)
  • 4.3.3 Deadlock Handling
  • 4.3.4 Variants of 2PL
  • 4.3.5 Ordered Sharing of Locks (O2PL)
  • 4.3.6 Altruistic Locking (AL)
  • 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  • 4.3.8 Geometry of Locking
• 4.4 Non-Locking Schedulers
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned
(Write-only) Tree Locking

Motivating example:
concurrent executions of transactions with access patterns
that comply with organizing data items into a virtual tree

\[ t_1 = w_1(a) \ w_1(b) \ w_1(d) \ w_1(e) \ w_1(i) \ w_1(k) \]
\[ t_2 = w_2(a) \ w_2(b) \ w_2(c) \ w_2(d) \ w_2(h) \]

Definition (Write-only Tree Locking (WTL)):
Under the write-only tree locking protocol (WTL) lock requests and releases must obey LR1 - LR4 and the following additional rules:

**WTL1:** A lock on a node \( x \) other than the tree root can be acquired only if the transaction already holds a lock on the parent of \( x \).

**WTL2:** After a \( w_{ui}(x) \) no further \( w_{li}(x) \) is allowed (on the same \( x \)).

Example:
\[ w_{l1}(a) \ w_1(a) \ w_{l1}(b) \ w_{u1}(a) \ w_1(b) \ w_{l2}(a) \ w_2(a) \ w_{l1}(d) \ w_1(d) \ w_{u1}(d) \ w_{l1}(e) \ w_{u1}(b) \ w_1(e) \ w_{l2}(b) \ w_{u2}(a) \ w_2(b) \ldots \]
Correctness and Properties of WTL

**Lemma 4.6:**
If $t_i$ locks $x$ before $t_j$ does in schedule $s$, then for each successor $v$ of $x$ that is locked by both $t_i$ and $t_j$ the following holds: $w_{l_i}(v) \lessdot s w_{u_i}(v) \lessdot s w_{l_j}(v)$.

**Theorem 4.10:**
$\text{Gen}(\text{WTL}) \subseteq \text{CSR}$.

**Theorem 4.11:**
WTL is deadlock-free.

**Comment:** WTL is applicable even if a transaction‘s access patterns are not tree-compliant, but then locks must still be obtained along all relevant paths in the tree using the WTL rules.
Read-Write Tree Locking

**Problem:** $t_i$ locks root before $t_j$ does, but $t_j$ passes $t_i$ within a “read zone”

**Solution:** formalize “read zone” and enforce two-phase property on “read zones”

Example:

```plaintext
rl_1(a) rl_1(b) r_1(a) r_1(b) wl_1(a) w_1(a) wl_1(b) ul_1(a) rl_2(a) r_2(a)
w_1(b) rl_1(e) ul_1(b) rl_2(b) r_2(b) ul_2(a) rl_2(e) r_2(i) ul_2(b) r_2(e) r_1(e)
r_2(i) wl_2(i) w_2(i) wl_2(k) ul_2(e) ul_2(i) rl_1(i) ul_1(e) r_1(i) ...
```

→ appears to follow TL rules but $\not\in$ CSR
Locking Rules of RWTL

For transaction t with read set RS(t) and write set WS(t) let C_1, ..., C_m be the connected components of RS(t).

A **pitfall** of t is a set of the form 

\[ C_i \cup \{ x \in WS(t) \mid x \text{ is a child or parent of some } y \in C_i \}. \]

**Example:**

t with RS(t)={f, i, g} and WS(t)={c, l, j, k, o} has pitfalls pf_1={c, f, i, l, j} and pf_2={g, c, k}.

**Definition (read-write tree locking (RWTL)):**

Under the **read-write tree locking protocol (RWTL)** lock requests and releases must obey LR1 - LR4, WTL1, WTL2, and the two-phase property within each pitfall.
Correctness and Generalization of RWTL

Theorem 4.12:
Gen (RWTL) ⊆ CSR.

RWTL can be generalized for a DAG organization of data items into a **DAG locking** protocol with the following additional rule: 
t_i is allowed to lock data item x only if holds locks on a majority of the predecessors of x.
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
  - 4.4.1 Timestamp Ordering
  - 4.4.2 Serialization-Graph Testing
  - 4.4.3 Optimistic Protocols
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
(Basic) Timestamp Ordering

**Timestamp ordering rule (TO rule):**
Each transaction $t_i$ is assigned a unique timestamp $ts(t_i)$ (e.g., the time of $t_i$'s beginning).
If $p_i(x)$ and $q_j(x)$ are in conflict, then the following must hold:

$p_i(x) <_s q_j(x)$ iff $ts(t_i) < ts(t_j)$ for every schedule $s$.

**Theorem 4.15:**
$Gen\ (TO) \subseteq CSR$.

**Basic timestamp ordering protocol (BTO):**
- For each data item $x$ maintain $max-r(x) = \max \{ts(t_j) \mid r_j(x) \text{ has been scheduled}\}$ and $max-w(x) = \max \{ts(t_j) \mid w_j(x) \text{ has been scheduled}\}$.
- Operation $p_i(x)$ is compared to $max-q(x)$ for each conflicting $q$:
  - if $ts(t_i) < max-q(x)$ for some $q$ then abort $t_i$
  - else schedule $p_i(x)$ for execution and set $max-p(x)$ to $max w.r.t. ts(t_i)$
BTO Example

\[ s = r_1(x) \; w_2(x) \; r_3(y) \; w_2(y) \; c_2 \; w_3(z) \; c_3 \; r_1(z) \; c_1 \]

\[ r_1(x) \; w_2(x) \; r_3(y) \; a_2 \; w_3(z) \; c_3 \; a_1 \]
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
  - 4.4.1 Timestamp Ordering
  - 4.4.2 Serialization-Graph Testing
  - 4.4.3 Optimistic Protocols
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
Serialization Graph Testing (SGT)

**SGT protocol:**
- For $p_i(x)$ create a new node in the graph if it is the first operation of $t_i$.
- Insert edges $(t_j, t_i)$ for each $q_j(x) \preceq p_i(x)$ that is in conflict with $p_i(x)$ ($i \neq j$).
- If the graph has become cyclic then abort $t_i$ (and remove it from the graph) else schedule $p_i(x)$ for execution.

**Theorem 4.16:**
Gen (SGT) = CSR.

**Node deletion rule:**
A node $t_i$ in the graph (and its incident edges) can be removed when $t_i$ is terminated and is a source node (i.e., has no incoming edges).

**Example:**
$r_1(x) \ w_2(x) \ w_2(y) \ c_2 \ r_1(y) \ c_1$
removing node $t_2$ at the time of $c_2$
would make it impossible to detect the cycle.
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
  - 4.4.1 Timestamp Ordering
  - 4.4.2 Serialization-Graph Testing
- 4.4.3 Optimistic Protocols
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
Optimistic Protocols

Motivation: conflicts are infrequent

Approach:
divide each transaction $t$ into three phases:

read phase:
execute transaction with writes into private workspace

validation phase (certifier):
upon $t$‘s commit request
test if schedule remains CSR if $t$ is committed now
based on $t$‘s read set $RS(t)$ and write set $WS(t)$

write phase:
upon successful validation
transfer the workspace contents into the database
(deferred writes)
otherwise abort $t$ (i.e., discard workspace)
Backward-oriented Optimistic CC (BOCC)

Execute a transaction’s validation and write phase together as a critical section: while $t_i$ being in the val-write phase, no other $t_k$ can enter its val-write phase.

**BOCC validation** of $t_j$:
compare $t_j$ to all previously committed $t_i$
accept $t_j$ if one of the following holds
- $t_i$ has ended before $t_j$ has started, or
- $RS(t_j) \cap WS(t_i) = \emptyset$ and $t_i$ has validated before $t_j$

**Theorem 4.46:**
Gen (BOCC) $\subset$ CSR.

**Proof:**
Assume that $G(s)$ is acyclic. Adding a newly validated transaction can insert only edges into the new node, but no outgoing edges (i.e., the new node is last in the serialization order).
BOCC Example

\[\begin{align*}
\text{read phase} & \quad \text{write phase} \\
\text{val.} & \\
\text{abort} &
\end{align*}\]
Forward-oriented Optimistic CC (FOCC)

Execute a transaction’s val-write phase as a strong critical section: while \( t_i \) being in the val-write phase, no other \( t_k \) can perform any steps.

**FOCC validation** of \( t_j \):

compare \( t_j \) to all concurrently active \( t_i \) (which must be in their read phase)
accept \( t_j \) if \( \text{WS}(t_j) \cap \text{RS}^*(t_i) = \emptyset \) where \( \text{RS}^*(t_i) \) is the current read set of \( t_i \)

Remarks:

• FOCC is much more flexible than BOCC:
  upon unsuccessful validation of \( t_j \) it has three options:
  • abort \( t_j \)
  • abort one of the active \( t_i \) for which \( \text{RS}^*(t_i) \) and \( \text{WS}(t_j) \) intersect
  • wait and retry the validation of \( t_j \) later (after the commit of the intersecting \( t_i \))
• Read-only transactions do not need to validate at all.


Correctness of FOCC

**Theorem 4.18:**

Gen (FOCC) ⊂ CSR.

**Proof:**

Assume that G(s) has been acyclic and that validating \( t_j \) would create a cycle. So \( t_j \) would have to have an outgoing edge to an already committed \( t_k \).

However, for all previously committed \( t_k \) the following holds:

- If \( t_k \) was committed before \( t_j \) started, then no edge \( (t_j, t_k) \) is possible.
- If \( t_j \) was in its read phase when \( t_k \) validated, then WS(\( t_k \)) must be disjoint with RS*(\( t_j \)) and all later reads of \( t_j \) and all writes of \( t_j \) must follow \( t_k \) (because of the strong critical section); so neither a wr nor a ww/rw edge \( (t_j, t_k) \) is possible.
FOCC Example

**read phase**

- \( t_1 \)
  - \( r_1(x) \)
  - \( r_1(y) \)
  - \( \text{val.} w_1(x) \)

**write phase**

- \( t_2 \)
  - \( r_2(y) \)

- \( t_3 \)
  - \( r_3(z) \)
  - \( \text{abort} \)

- \( t_4 \)
  - \( r_4(x) \)
  - \( r_4(y) \)
  - \( \text{val.} w_4(y) \)

- \( t_5 \)
  - \( r_5(x) \)
  - \( r_5(y) \)
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
• 4.4 Non-Locking Schedulers
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned
Hybrid Protocols

**Idea:** Combine different protocols, each handling different types of conflicts (rw/wr vs. ww) or data partitions.

**Caveat:** The combination must guarantee that the union of the underlying “local” conflict graphs is acyclic.

**Example 4.15:**
use SS2PL for rw/wr synchronization and TO or TWR for ww with **TWR (Thomas’ write rule)** as follows:

for \( w_j(x) \): if \( ts(t_j) > \text{max-w}(x) \) then execute \( w_j(x) \) else do nothing

\[
\begin{align*}
s_1 &= w_1(x) \ r_2(y) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1 \\
s_2 &= w_1(x) \ r_2(y) \ w_2(x) \ w_2(y) \ c_2 \ r_1(y) \ w_1(y) \ c_1
\end{align*}
\]

both accepted by SS2PL/TWR with \( ts(t_1) < ts(t_2) \), but \( s_2 \) is not CSR.

**Problem with \( s_2 \):** needs synch among the two “local” serialization orders.

**Solution:** assign timestamps such that the serialization orders of SS2PL and TWR are in line

\[ ts(i) < ts(j) \iff c_i < c_j \]
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
• 4.4 Non-Locking Schedulers
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned
Lessons Learned

• S2PL is the most versatile and robust protocol and widely used in practice
• Knowledge about specifically restricted access patterns facilitates non-two-phase locking protocols (e.g., TL, AL)
• O2PL and SGT are more powerful but have more overhead
• FOCC can be attractive for specific workloads
• Hybrid protocols are conceivable but non-trivial