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
7  Concurrency Control Algorithms on Objects

• 7.2 Locking for Flat Object Transactions
  • 7.3 Layered Locking
  • 7.4 Locking on General Transaction Forests
  • 7.5 Hybrid Algorithms
  • 7.6 Locking for Return-value Commutativity
  • 7.7 Lessons Learned

“A journey of thousand miles must begin with a single step.” (Lao-tse)
Reminder: Flat Object Schedule

(State-independent)
Commutativity table:

<table>
<thead>
<tr>
<th></th>
<th>withdraw (x,Δ₂)</th>
<th>deposit (x,Δ₂)</th>
<th>getbalance (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>withdraw (x,Δ₁)</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>deposit (x,Δ₁)</td>
<td>—</td>
<td>+</td>
<td>—</td>
</tr>
<tr>
<td>getbalance (x)</td>
<td>—</td>
<td>—</td>
<td>+</td>
</tr>
</tbody>
</table>
2PL for Flat Object Schedules

- introduce a special lock mode for each operation type
- derive lock compatibility from state-independent commutativity

- **Lock acquisition rule:**
  \( L_1 \) operation \( f(x) \) needs to lock \( x \) in \( f \) mode

- **Lock release rule:**
  Once an \( L_1 \) lock of \( f(x) \) is released, no other \( L_1 \) lock can be acquired.

**Example:**

- \( t_1 \): deposit(a)  deposit(b)
- \( t_2 \): withdraw(c)  withdraw(a)

---

Note that operations may be issued in parallel but the execution of a single operation is both sequential and uninterrupted, so as to fit the flat object definition.
7 Concurrency Control Algorithms on Objects

- 7.2 Locking for Flat Object Transactions
- **7.3 Layered Locking**
- 7.4 Locking on General Transaction Forests
- 7.5 Hybrid Algorithms
- 7.6 Locking for Return-value Commutativity
- 7.7 Lessons Learned
Layered 2PL

- **Lock acquisition rule:**
  \( L_i \) operation \( f(x) \) with parent \( p \), which is now a **subtransaction**, needs to lock \( x \) in \( f \) mode

- **Lock release rule:**
  Once an \( L_i \) lock of \( f(x) \) with parent \( p \) is released, no other child of \( p \) can acquire any locks.

- **Subtransaction rule:**
  At the termination of an \( L_i \) operation \( f(x) \), all \( L_{(i-1)} \) locks acquired for children of \( f(x) \) are released.

**Theorem 7.1:**
Layered 2PL generates only tree reducible schedules.

**Proof:** All level-to-level schedules are OCSR, hence the claim (by Theorem 6.2).

Special cases:
- single-page subtransactions merely need **latching**
- for all-commutative \( L_i \) operations, transactions are decomposed into sequences of independently isolated, **chained subtransactions**
Here modify(y) waits until the modify(y) lock is released, apparently SS2PL (otherwise could release after modify(w) is acquired)
3-Level Example

Insert Into Persons Values (Name=..., City="Austin", Age=29, ...)

Select Name From Persons Where City="Seattle" And Age=29

Select Name From Persons Where Age=30

store(x) insert (CityIndex, "Austin", @x)

search (CityIndex, "Seattle")

insert (AgeIndex, 29, @x)

search (AgeIndex, 29)

search (AgeIndex, 30)

fetch(z)

r(p) w(p) r(r) r(n)

r(r) r(l) r(n) w(l) r(r') r(n') r(l') w(l') r(p)

r(r') r(n') r(l') r(p)

Note erasure

Seems to be a “hypothetical execution”, not an actual one.

Note erasure

r(p)
3-Level 2PL Example

Insert Into Persons
Values (Name=..., City="Austin", Age=29, ...)

Select Name From Persons
Where City="Seattle" And Age=29

Select Name From Persons
Where Age=30

locks here are predicate locks
No level 2 predicate conflict

Locks here are predicate locks

store(x)
insert (CityIndex, "Austin", @x)
search (CityIndex, "Seattle")
insert (AgeIndex, 29, @x)
search (AgeIndex, 30) fetch(z)
search (AgeIndex, 29) fetch(y)

r(p) w(p)
(r) r(n) r(l) w(l)

Note erasure
Note erasure
Selective Layered 2PL

For n-level schedule with layers $L_n$, ..., $L_0$ apply locking on selected layers $L_{i_0}, \ldots, L_{i_k}$ with $1 \leq k \leq n$, $i_0 = n$, $i_k = 0$, $i_\nu > i_{\nu+1}$, skipping all other layers.

- **Lock acquisition rule:**
  $L_{i_\nu}$ operation $f(x)$ with $L_{i_{\nu-1}}$ ancestor $p$, which is now a subtransaction, needs to lock $x$ in $f$ mode.

- **Lock release rule:**
  Once an $L_{i_\nu}$ lock of $f(x)$ with $L_{i_{\nu-1}}$ ancestor $p$ is released, no other $L_{i_\nu}$ descendant of $p$ can acquire any locks.

- **Subtransaction rule:**
  At the termination of an $L_{i_\nu}$ operation $f(x)$, all $L_{i_{\nu+1}}$ locks acquired for descendants of $f(x)$ are released.
Selective Layered 2PL Example

Insert Into Persons
Values (Name=..., City="Austin", Age=29, ...)

Select Name From Persons
Where City="Seattle" And Age=29

Select Name From Persons
Where Age=30

store(x)
insert
(CityIndex, "Austin", @x)
search
(CityIndex, "Seattle")
insert
(AgeIndex, 29, @x)
search
(AgeIndex, 29)
fetch(y)

What about the ordering of operations in “hidden layers”?

A number of errors here
Intra Transaction Parallelism

Delete and Insert run in parallel by the same transaction
7 Concurrency Control
Algorithms on Objects

- 7.2 Locking for Flat Object Transactions
- 7.3 Layered Locking
- 7.4 Locking on General Transaction Forests
  - 7.5 Hybrid Algorithms
  - 7.6 Locking for Return-value Commutativity
  - 7.7 Lessons Learned
Problem Scenario

Problem: layers can be “bypassed”
Solution: keep locks in “retained” mode
• **Lock acquisition rule:**
  Operation $f(x)$ with parent $p$ needs to lock $x$ in $f$ mode

• **Lock conflict rule:**
  A lock requested by $r(x)$ is granted if
  - either no conflicting lock on $x$ is held
  - or when for every transaction that holds a conflicting lock, say $h(x)$,
    $h(x)$ is a retained lock and $r$ and $h$ have ancestors $r'$ and $h'$ such that
    $h'$ is terminated and commutes with $r'$

• **Lock release rule:**
  Once a lock of $f(x)$ with parent $p$ is released,
  no other child of $p$ can acquire any locks.

• **Subtransaction rule:**
  At the termination of $f(x)$,
  all locks acquired for children of $f(x)$ are converted into retained locks.

• **Transaction rule:**
  At the termination of a transaction, all locks are released.

**Theorem 7.2:**
The object-model 2PL generates only tree-reducible schedules.
• If all locks of \( t_1 \) were kept until commit, then tree reducibility were trivially guaranteed.
• Now show that retained \( f_1 \) lock by \( h_1 \) is sufficient to prevent non-commutative subtree:

Let \( f_2 \) be the first conflict with any lock under \( h_1 \);
\( f_2 \) is allowed to proceed only if \( h_1 \) is terminated and \( h_2 \) commutes with \( h_1 \):
- \( \rightarrow \) isolate \( h_2 \) from \( h_1 \)
- \( \rightarrow \) prune \( h_2 \) and \( h_1 \)
- \( \rightarrow \) commute \( h_2 \) with \( h_1 \) if necessary
7 Concurrency Control Algorithms on Objects

- 7.2 Locking for Flat Object Transactions
- 7.3 Layered Locking
- 7.4 Locking on General Transaction Forests
- 7.5 Hybrid Algorithms
  - 7.6 Locking for Return-value Commutativity
  - 7.7 Lessons Learned
Hybrid Algorithms

**Theorem 7.3:**
For 2-level schedules the combination of 2PL at $L_1$ and FOCC at $L_0$ generates only tree-reducible schedules.

**Theorem 7.4:**
For 2-level schedules the combination of 2PL at $L_1$ and ROMV at $L_0$ generates only tree-reducible schedules.

*These combinations are particularly attractive because subtransactions are short and there is a large fraction of read-only subtransactions.*
7 Concurrency Control Algorithms on Objects

• 7.2 Locking for Flat Object Transactions
• 7.3 Layered Locking
• 7.4 Locking on General Transaction Forests
• 7.5 Hybrid Algorithms
• 7.6 Locking for Return-value Commutativity
• 7.7 Lessons Learned
Locking for Return-value

Commutativity

- introduce a special lock mode for each pair
  \(<\text{operation type, return value}>\),
  
  Example: lock modes
  
  \(\text{withdraw-ok, withdraw-no, deposit-ok, getbalance-ok}\)

- defer lock conflict test until end of subtransaction

- rollback subtransaction if lock cannot be granted and retry
Escrow Locking

... on bounded counter object $x$ with lower bound $low(x)$ and upper bound $high(x)$

**Approach:**
- maintain infimum $inf(x)$ and supremum $sup(x)$ for the value of $x$ taking into account all possible outcomes of active transactions
- adjust $inf(x)$ and $sup(x)$ upon
  - operations $incr(x)$, $decr(x)$, and
  - commit or abort of transactions
# Escrow Locking Pseudocode

<table>
<thead>
<tr>
<th>incr($x, \Delta$):</th>
<th>decr($x, \Delta$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $x.\text{sup} + \Delta \leq x.\text{high}$ then</td>
<td>if $x.\text{low} \leq x.\text{inf} - \Delta$ then</td>
</tr>
<tr>
<td>$x.\text{sup} := x.\text{sup} + \Delta$; return ok</td>
<td>$x.\text{inf} := x.\text{inf} - \Delta$; return ok</td>
</tr>
<tr>
<td>else if $x.\text{inf} + \Delta &gt; x.\text{high}$ then</td>
<td>else if $x.\text{low} &gt; x.\text{sup} - \Delta$ then</td>
</tr>
<tr>
<td>return no</td>
<td>return no</td>
</tr>
<tr>
<td>else wait fi fi;</td>
<td>else wait fi fi;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>commit($t$):</th>
<th>abort($t$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each op incr($x, \Delta$) by $t$ do</td>
<td>for each op incr($x, \Delta$) by $t$ do</td>
</tr>
<tr>
<td>$x.\text{inf} := x.\text{inf} + \Delta$ od;</td>
<td>$x.\text{sup} := x.\text{sup} - \Delta$ od;</td>
</tr>
<tr>
<td>for each op decr($x, \Delta$) by $t$ do</td>
<td>for each op decr($x, \Delta$) by $t$ do</td>
</tr>
<tr>
<td>$x.\text{sup} := x.\text{sup} - \Delta$ od;</td>
<td>$x.\text{inf} := x.\text{inf} + \Delta$ od;</td>
</tr>
</tbody>
</table>
Escrow Locking Example

constraint:

\[ 0 \leq x \]

\[ x^{(0)} = 100 \]

\[ \begin{align*}
  & [20, 100] \\
  & [10, 100] \\
  & [10, 150] [70] \\
  & [20, 70] [0, 70] \\
  & [50, 70] \\
  & [x.\text{inf}, x.\text{sup}] 
\end{align*} \]

\[ x^{(4)} = 50 \]
Escrow Deadlock Example

$t_1$  incr(x,10)  update(y)

$t_2$  incr(x,10)  update(z)

$t_3$  incr(x,10)

$t_4$  getval(y)  getval(z)  decr(x,20)

$x^{(0)} = 0$
7 Concurrency Control Algorithms on Objects

- 7.2 Locking for Flat Object Transactions
- 7.3 Layered Locking
- 7.4 Locking on General Transaction Forests
- 7.5 Hybrid Algorithms
- 7.6 Locking for Return-value Commutativity
- 7.7 Lessons Learned
Lessons Learned

• Layered 2PL is the fundamental protocol for industrial-strength data servers with record granularity locking (it explains the trick of “long locking” and “short latching”).
• This works for all kinds of ADT operations within layers; decomposed transactions with chained subtransactions (aka. “Sagas”) are simply a special case.
• Non-layered schedules require additional, careful locking rules.
• Locking on some layers can be combined with other protocols (e.g., ROMV or FOCC) on other layers.
• Escrow locking on counter objects is an example for additional performance enhancements by exploiting rv commutativity.