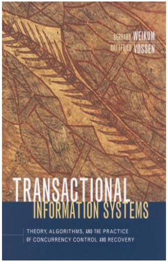
Transactional Information Systems:

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

Gerhard Weikum and Gottfried Vossen



ISBN 1-55860-508-8

"Teamwork is essential. It allows you to blame someone else."(Anonymous)

Part II: Concurrency Control

7 Concurrency Control Algorithms on Objects

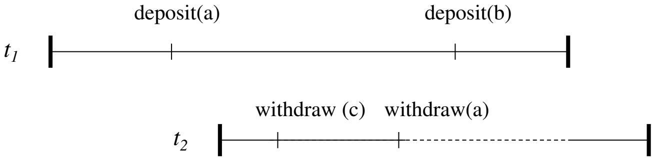
- **7.2 Locking for Flat Object Transactions**
- 7.3 Layered Locking
- 7.4 Locking on General Transaction Forests
- 7.5 Hybrid Algorithms
- 7.6 Locking for Return-value Commutativity
- 7.7 Lessons Learned

“A journey of thousand miles must begin with a single step.” (Lao-tse)

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:



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.

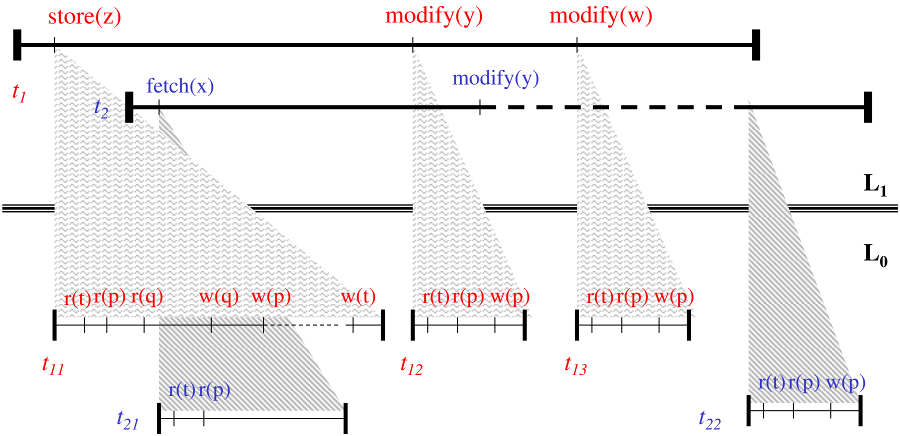
Theorem 7.1:
Layered 2PL generates only tree reducible schedules.

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

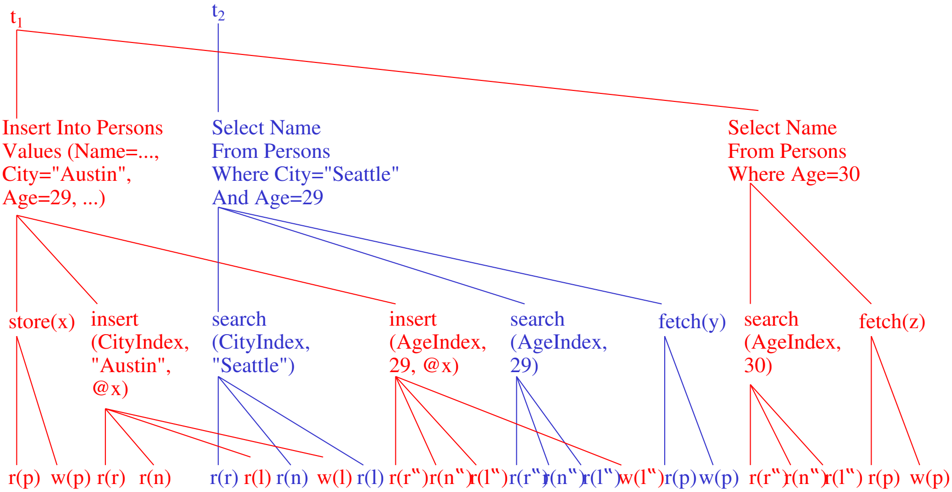
Special cases:

- single-page subtransactions merely need **latching**
- for all-commutative L_i operations, transactions are decomposed into sequences of independently isolated, **chained subtransactions**

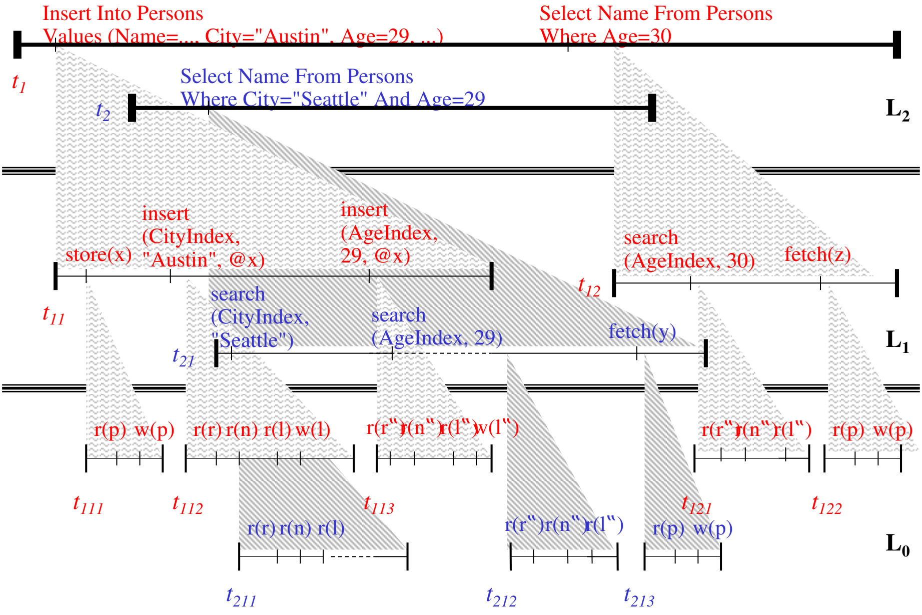
2-Level 2PL Example



3-Level Example



3-Level 2PL Example

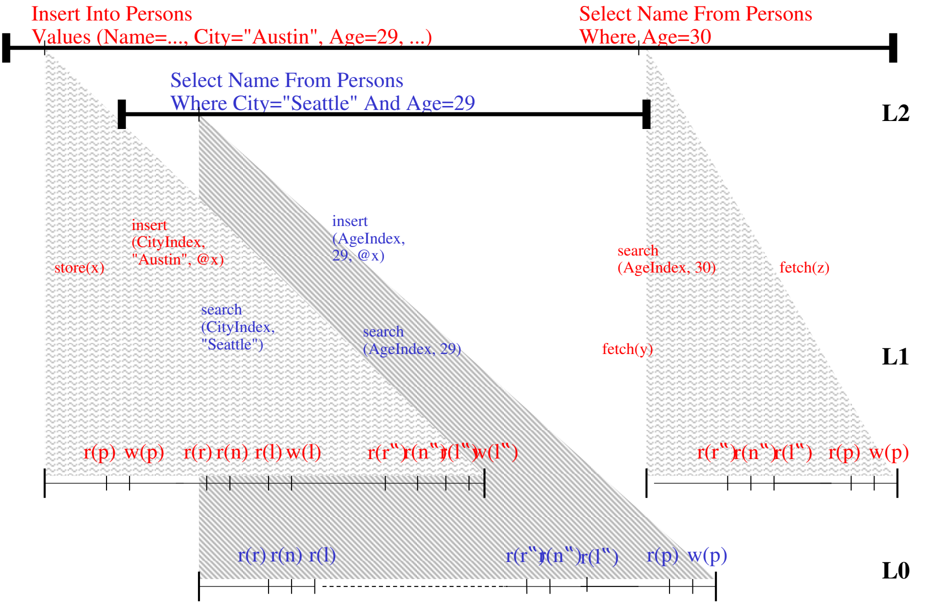


Selective Layered 2PL

For n-level schedule with layers L_n, ..., L_0
apply locking on selected layers Li_0, ... , Li_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:**
  Li_ν operation f(x) with $Li_{\nu-1}$ ancestor p, which is now a subtransaction, needs to lock x in f mode
- **Lock release rule:**
  Once an Li_ν lock of f(x) with $Li_{\nu-1}$ ancestor p is released, no other Li_ν descendant of p can acquire any locks.
- **Subtransaction rule:**
  At the termination of an Li_ν operation f(x), all $Li_{\nu+1}$ locks acquired for descendants of f(x) are released.

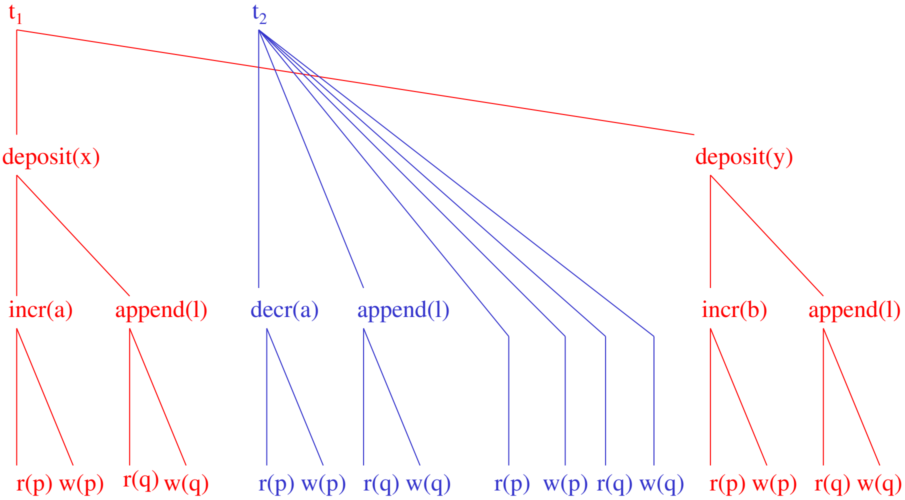
Selective Layered 2PL Example



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"

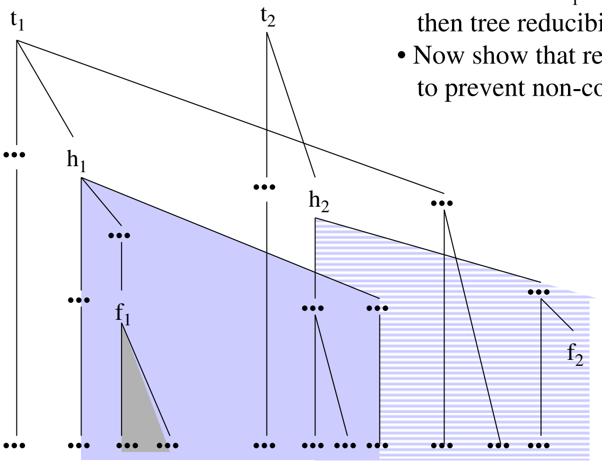
Solution: keep locks in "retained" mode

General Object-Model 2PL

- **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.

Proof Sketch for Theorem 7.2



- 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
  <operation type, return value>,
  Example: lock modes
  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)*

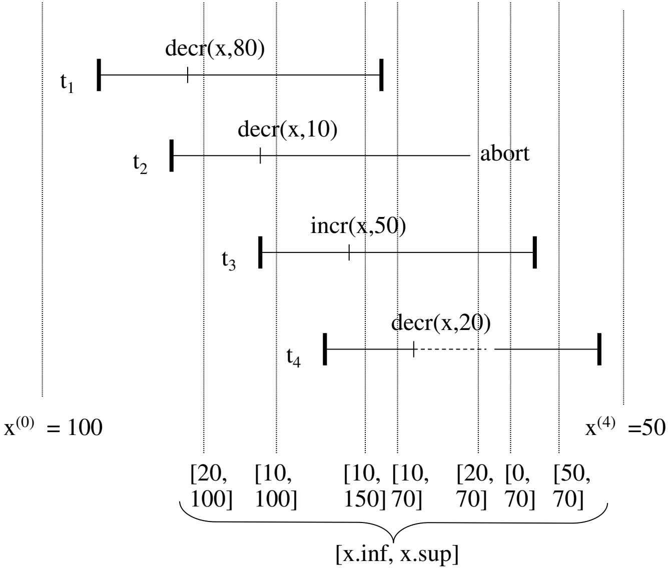
Approach:

- 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

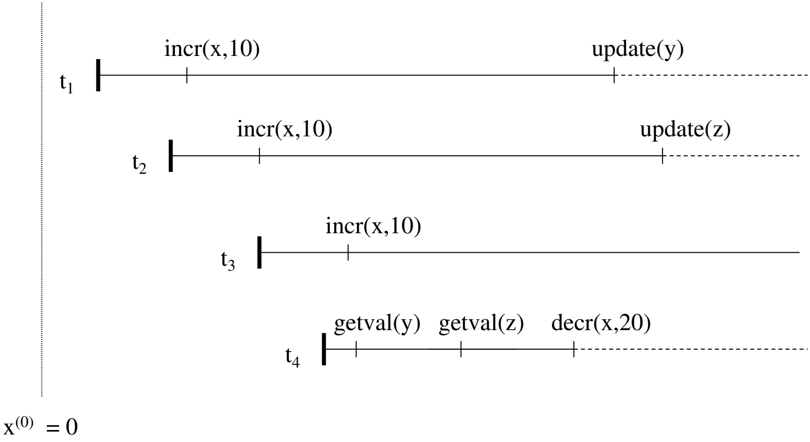
incr(x, Δ): if x.sup + Δ ≤ x.high then x.sup := x.sup + Δ; return ok else if x.inf + Δ > x.high then return no else wait fi fi;	*decr(x, Δ):* if x.low ≤ x.inf - Δ then x.inf := x.inf - Δ; return ok else if x.low > x.sup - Δ then return no else wait fi fi;
commit(t): for each op incr(x, Δ) by t do x.inf := x.inf + Δ od; for each op decr(x, Δ) by t do x.sup := x.sup - Δ od;	*abort(t):* for each op incr(x, Δ) by t do x.sup := x.sup - Δ od; for each op decr(x, Δ) by t do x.inf := x.inf + Δ od;

Escrow Locking Example



constraint:
$0 \leq x$

Escrow Deadlock Example



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.