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 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- 10.4 Transient Versioning
- 10.5 Nested Transactions for Intra-transaction parallelism
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned

“All theory, my friend, is grey; but the precious tree of life.”
(Johann Wolfgang von Goethe)
Organization of Lock Control Blocks

**Transaction Control Blocks (TCBs)**

- Transaction Id
- Update Flag
- Transaction Status
- Number of Locks
- LCB Chain

**Hash Table indexed by Resource Id**

**Resource Control Blocks (RCBs)**

- Resource Id
- Hash Chain
- FirstInQueue

**Lock Control Blocks (LCBs)**

- Transaction Id
- Resource Id
- Lock Mode
- Lock Status
- NextInQueue
- LCB Chain
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- **10.3 Multi-Granularity Locking and Lock Escalation**
- 10.4 Transient Versioning
- 10.5 Nested Transactions for Intra-transaction parallelism
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Reconciling Coarse- and Fine-grained Locking

**Problem:** For reduced overhead, table scans should use coarse locks
Detect conflict of page lock with tablespace lock

**Approach:** Set “intention locks” on coarser granules

**Multi-granularity locking protocol:**
- A transaction can lock any granule in S or X mode.
- Before a granule p can be locked in S or X mode, the transaction needs to hold an IS or IX lock on all coarser granules that contain p.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IS</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>IX</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>SIX</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Typical policy:**
- use coarse locks for table scans
- use fine locks otherwise
- escalate dynamically to coarse locks when memory usage for LCBs becomes critical
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- **10.4 Transient Versioning**
- 10.5 Nested Transactions for Intra-transaction parallelism
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Storage Organization for Transient Versioning

- update on current data moves old version to version pool
- read-only transactions follow version chains
- old versions are kept sorted by their successor timestamps
  → garbage collection simply advances begin pointer

Current data

- RID creation timestamp data fields pointer to prior version

Version pool

- deleted flag
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- 10.4 Transient Versioning
- **10.5 Nested Transactions for Intra-transaction parallelism**
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Multi-threaded Transactions

Example:

t_1: t_{11} t_{12} t_{13} t_{14} with t_{12} and t_{13} as parallel threads

t_{11}: r(t) r(p) w(p) /* store new incoming e-mail */

t_{12}: t_{121} t_{122} t_{123} t_{124} with t_{122}, t_{123}, t_{124} as parallel threads

t_{121}: r(t) r(s) w(s) /* update folder by subject */

t_{122}: r(r) r(n) r(l) w(l) /* update text index for descriptor_1 */

t_{123}: r(r) r(n) r(m) w(m) w(n) /* update text index for descriptor_2 */

t_{124}: r(r) r(n) r(l) w(l) /* update text index for descriptor_3 */

t_{13}: r(t) r(f) w(f) w(g) w(t) /* update folder by sender */

t_{14}: r(t) r(p) w(p) r(g) w(g) /* assign priority */
Locking for Nested Transactions

2PL protocol for nested transactions:
• Leaves of a transaction tree acquire locks as needed, based on 2PL for the duration of the transaction.
• Upon terminating a thread, all locks held by the thread are inherited by its parent.
• A lock request by a thread is granted if no conflicting lock on the same data item is currently held or the only conflicting locks are held by ancestors of the thread.

Theorem 10.1:
2PL for nested transactions generates only schedules that are equivalent to a serial execution of the transactions where each transaction executes all its sibling sets serially.
Layered Locking with Intra-transaction Parallelism

Layer 0

- search (CityIndex, “Austin”)
- delete (CityIndex, "Austin", @x)
- insert (CityIndex, “Dallas”, @x)

Layer 1

- search (CityIndex, “Boston”)
- fetch(y)

Transactions:

- t1
  - r(p) w(p)
  - r(r) r(n) r(l) w(l)
- t11
  - r(r) r(n) r(l)
  - r(p) w(p)
  - r(r)
- t12
  - r(p) w(p)
  - r(r)
  - r(n) r(l) w(l)
  - r(r) r(n) r(l) w(l)
  - r(p) w(p)
- t13
  - r(p) w(p)
  - r(r)
- t14
  - r(p) w(p)
  - r(r)
  - r(n) r(l) w(l)
- t15
  - r(r) r(n) r(l) w(l)
- t2
  - r(r) r(n) r(l) w(l)
  - r(p) w(p)
- t21
  - r(r) r(n) r(l) w(l)
- t22
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- 10.4 Transient Versioning
- 10.5 Nested Transactions for Intra-transaction parallelism
- **10.6 Tuning Options**
  - 10.7 Overload Control
  - 10.8 Lessons Learned
Tuning Repertoire

• Manual locking (or manual preclaiming)
• Choice of SQL isolation level(s)
• Application structuring towards short transactions
• MPL control
Definition 10.1 (Isolation Levels):

- A schedule s runs under isolation level **read uncommitted** (aka. dirty read or browse mode) if write locks are subject to S2PL.
- A schedule s runs under isolation **read committed** (aka. cursor stability) if write locks are subject to S2PL and read locks are held for the duration of an SQL operation.
- A schedule s runs under isolation level **serializability** if it can be generated by S2PL.
- A schedule s runs under isolation level **repeatable read** if all anomalies other than phantoms are prevented.

**Remark:** A scheduler can use different isolation levels for different transactions.

**Observation:** **read committed is susceptible to lost updates**

**Example:** \( r_1(x) \ r_2(x) \ w_2(x) \ c_2 \ w_1(x) \ c_1 \)
Definition 10.2 (Multiversion Read Committed and Snapshot Isolation Levels):

- A transaction runs under isolation level **multiversion read committed** if it reads the most recent committed versions as of the transaction’s begin and uses S2PL for writes.
- A transaction runs under **snapshot isolation** if it reads the most recent versions as of the transaction’s begin and its write set is disjoint with the write sets of all concurrent transactions.

**Observation:** **snapshot isolation does not guarantee MVSR**

**Example:**

\[
\begin{align*}
    r_1(x_0) & \quad r_1(y_0) & \quad r_2(x_0) & \quad r_2(y_0) & \quad w_1(x_1) & \quad c_1 & \quad w_2(y_2) & \quad c_2
\end{align*}
\]

Possible interpretation:

- constraint \(x + y \geq 0, x_0 = y_0 = 5\),
- \(t_1\) subtructs 10 from \(x\), \(t_2\) subtructs 10 from \(y\)
**Application-level “Optimistic Locking”**

Idea: strive for short transactions or short lock duration

**Approach:**
- aim at two-phase structure of transactions: read phase + short write phase
- run queries under relaxed isolation level (typically read committed)
- rewrite program to test for concurrent writes during write phase

**Example:**

```sql
Select Balance, Counter Into :b, :c
From Accounts Where AccountNo = :x
...
compute interests and fees, set b, ...
...
Update Accounts
Set Balance = :b, Counter = Counter + 1
Where AccountNo = :x And Counter = :c
```

avoids lost updates, but cannot guarantee consistency
Unrestricted multiprogramming level (MPL) can lead to performance disaster known as data-contention thrashing:
- additional transactions cause superlinear increase of lock waits
- throughput drops sharply
- response time approaches infinity
Benefit of MPL Limitation

system admin sets **MPL limit**: during load bursts excessive transactions wait in **transaction admission queue**

avoids thrashing, but poses a tricky tuning problem:
- overly low MPL limit causes long waits in admission queue
- overly high MPL limit opens up the danger of thrashing
  problem is even more difficult for highly heterogeneous workloads
Chapter 10: Implementation and Pragmatic Issues

• 10.2 Data Structures of a Lock Manager
• 10.3 Multi-Granularity Locking and Lock Escalation
• 10.4 Transient Versioning
• 10.5 Nested Transactions for Intra-transaction parallelism
• 10.6 Tuning Options

• 10.7 Overload Control

• 10.8 Lessons Learned
Conflict-ratio-driven Overload Control

\[
\text{conflict ratio} = \frac{\text{#locks held by all trans.}}{\text{#locks held by running trans.}}
\]

\[
\text{critical conflict ratio} \approx 1.3
\]
Conflict-ratio-driven Overload Control Algorithm

upon begin request of transaction t:
    if conflict ratio < critical conflict ratio
    then admit t else put t in admission queue fi

upon lock wait of transaction t:
    update conflict ratio
    while not (conflict ratio < critical conflict ratio)
        among trans. that are blocked and block other trans.
        choose trans. v with smallest product
        #locks held * #previous restarts
        abort v and put v in admission queue od

upon termination of transaction t:
    if conflict ratio < critical conflict ratio then
        for each transaction q in admission queue do
            if (q will be started the first time) or
                (q has been a rollback/cancellation victim and
                all trans. that q was waiting for are terminated)
            then admit q fi od fi
Wait-depth Limitation (WDL)

Wait depth of transaction $t =$

$$
\begin{cases}
0 & \text{if } t \text{ is running} \\
 i + 1 & \text{if } \max \{\text{wait depth of transactions that block } t\} = i
\end{cases}
$$

Policy: allow only wait depths $\leq 1$

Case 1:

$$
\begin{array}{c}
t_{k1} \\
\vdots \\
t_{kn}
\end{array} \rightarrow
\begin{array}{c}
t_k \\
\vdots \\
t_i
\end{array} \\
\text{...} \quad \text{...}
$$

Case 2:

$$
\begin{array}{c}
t_{k1} \\
\vdots \\
t_{kn}
\end{array} \rightarrow
\begin{array}{c}
t_k \\
\vdots \\
t_i
\end{array} \\
\text{...} \quad \text{...}
$$
Chapter 10: Implementation and Pragmatic Issues

- 10.2 Data Structures of a Lock Manager
- 10.3 Multi-Granularity Locking and Lock Escalation
- 10.4 Transient Versioning
- 10.5 Nested Transactions for Intra-transaction parallelism
- 10.6 Tuning Options
- 10.7 Overload Control
- 10.8 Lessons Learned
Lessons Learned

• Locking can be efficiently implemented, with flexible handling of memory overhead by means of multi-granularity locks
• Tuning options include
  • choice of isolation levels
  • application-level tricks
  • MPL limitation
• Tuning requires extreme caution to guarantee correctness: if in doubt, don‘t do it!
• Concurrency control is susceptible to data-contention thrashing and needs overload control