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

- Example: Escalate dynamically to coarse locks when memory usage for LCBs becomes critical
Lock Escalation: Microsoft SQL Server
Lock Hierarchy

- Always have a Shared Lock (S) on DB level.
- When your query is connected to a DB (USE MyDatabase), Shared Lock prevents the dropping of the DB, or that backups are restored over that database.
- You have locks on the table, on the pages, and the records when you are performing an operation.
SQL Server Lock Escalation

- In DML: Intent Exclusive or Update Lock (IX or IU) on the table and page level, and a Exclusive or Update Lock (X or U) on the changed records.

- SQL Server always acquires locks from top to bottom to prevent Race Conditions, when multiple threads trying to acquire locks concurrently within the locking hierarchy.
SQL Server Lock Escalation

- DELETE operation on a table against 20,000 rows.

- Let’s assume that a row is 400 bytes long, means that 20 records fit onto one page of 8kb:
SQL Server Lock Escalation

- One S Lock on the database, 1 IX Lock on the table, 1,000 IX locks on the pages (20,000 records are spread across 1,000 pages), and you have finally 20,000 X locks on the records itself.

- In sum you have acquired 21,002 locks for the DELETE operation.
- Every lock needs in SQL Server 96 bytes of memory, so we look at 1.9 MB of locks just for 1 simple query.

- This will not scale indefinitely when you run multiple queries in parallel.
- For that reason SQL Server implements now the so-called Lock Escalation.
SQL Server Lock Escalation

- For more than 5,000 locks on one level in your locking hierarchy, SQL Server escalates into a simple coarse-granularity lock.
- SQL Server will by default *always* escalate directly to the table level.
- An escalation policy to the page level just doesn’t exist
- One Exclusive Lock (X) on the table level. **Concurrency of your database in a very negative way**
- No other session is able any more to access that table – every other query will just block

![Diagram of Database and Table locks](image.png)
SQL Server Lock Escalation

- Since SQL Server 2008 you can also control how SQL Server performs the Lock Escalation – through the ALTER TABLE statement and the property LOCK_ESCALATION.

- 3 different options:
  - **TABLE**: Always performs the Lock Escalation to the table level
  - **AUTO**: Lock Escalation is performed to the partition level, if the table is partitioned, and otherwise to the table level.
  - **DISABLE**: Disable Lock Escalation for that specific table. Lock Manager of SQL Server can then consume a huge amount of memory. **Not Recommended !!!**

```sql
-- Controlling Lock Escalation
ALTER TABLE Person.Person
SET (LOCK_ESCALATION = AUTO -- or TABLE or DISABLE)
GO
```
Lock Escalation

• System is decreasing the granularity of your locks
• Ex: DB turning your 100 row-level locks against a table into a single table-level lock.

• **Oracle will never escalate a lock. Never.**

• The terms lock conversion and lock promotion are synonymous.
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
• 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
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

$\text{search (CityIndex, } \text{“Austin”)}$
$\text{fetch(x)}$
$\text{modify(x)}$
$\text{delete (CityIndex, } \text{“Austin”, } @x)\text{)}$
$\text{insert (CityIndex, } \text{“Dallas”, } @x)\text{)}$

Layer 1

$\text{search (CityIndex, } \text{“Boston”)}$
$\text{fetch(y)}$

$t_1$

$t_{11}$

$t_{12}$

$t_{13}$

$t_{14}$

$t_{15}$

$t_{21}$

$t_{22}$
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
Oracle Manual Locking and User-Defined Locks

• Manually lock data via a SQL statement.
  – SELECT...FOR UPDATE
  – LOCK TABLE statement

• Create our own locks via the DBMS_LOCK package.
SELECT...FOR UPDATE

- To avoid the lost update issue whereby one session would overwrite another session’s changes.
- A method to serialize access to detail records to enforce business rules.

```
SCOTT@ORA12CR1> select empno, ename, sal
2 from emp
3 where empno = :empno
4 and decode( ename, :ename, 1 ) = 1
5 and decode( sal, :sal, 1 ) = 1
6 for update nowait
7 /

EMPNO ENAME SAL
---------- ---------- ----------
7934 MILLER 130
```

```
SCOTT@ORA12CR1> update emp
2 set ename = :ename, sal = :sal
3 where empno = :empno;
1 row updated.
SCOTT@ORA12CR1> commit;
Commit complete.
```
LOCK TABLE statement

• Simply locks the table, not the rows in the table.

• If you start modifying the rows, they will be locked as normal. So, this is not a method to save on resources (as it might be in other RDBMSs).

• **Locking a Table: Example for a large batch update**
  
  – locks the employees table in exclusive mode but does not wait if another user already has locked the table:

  ```
  LOCK TABLE employees IN EXCLUSIVE MODE NOWAIT;
  ```
**DBMS_LOCK package**

- **Use this package to serialize access to some resource external to Oracle.**
  - Using the UTL_FILE routine that allows you to write to a file on the server’s file system.
- File is external, Oracle won’t coordinate the many users trying to modify it simultaneously.
- Before you open, write, and close the file, you will request a lock named after the file in exclusive mode, and after you close the file, you will manually release the lock.
- Only one person at a time will be able to write a message to this file. Everyone else will queue up.
- **DBMS_LOCK package allows you to manually release a lock when you are done with it, or to give it up automatically when you commit, or even to keep it as long as you are logged in.**
DBMS_LOCK package

• ALLOCATE_UNIQUE Procedure: Allocates a unique lock identifier
  • Syntax
  DBMS_LOCK.ALLOCATE_UNIQUE (lockname IN VARCHAR2, lockhandle OUT VARCHAR2, expiration_secs IN INTEGER DEFAULT 864000);
  • CONVERT Function: converts a lock from one mode to another.
  • Syntax
  DBMS_LOCK.CONVERT(id IN INTEGER || lockhandle IN VARCHAR2, lockmode IN INTEGER, timeout IN NUMBER DEFAULT MAXWAIT) RETURN INTEGER;
  • RELEASE Function: explicitly releases a lock previously acquired using the REQUEST function
  • Syntax
  DBMS_LOCK.RELEASE (id IN INTEGER) RETURN INTEGER;
  DBMS_LOCK.RELEASE (lockhandle IN VARCHAR2) RETURN INTEGER;
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:

\[ r_1(x_0) \quad r_1(y_0) \quad r_2(x_0) \quad r_2(y_0) \quad w_1(x_1) \quad c_1 \quad w_2(y_2) \quad c_2 \]

Possible interpretation:

constraint \( x + y \geq 0 \), \( x_0 = y_0 = 5 \),
\( t_1 \) subtracts 10 from \( x \), \( t_2 \) subtracts 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: 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
Optimistic Locking

- Defers all locking up to the point right before the update is performed.
- One popular implementation of optimistic locking is to keep the old and new values in the application, and upon updating the data, use an update like

```
Update table
Set column1 = :new_column1, column2 = :new_column2, ....
Where primary_key = :primary_key
And decode( column1, :old_column1, 1 ) = 1
And decode( column2, :old_column2, 1 ) = 1
```

Other Options:
- Optimistic Locking Using a Version Column (systimestamp column)
- Optimistic Locking Using a Checksum
Optimistic Locking Using a Version Column

EODA@ORA12CR1> create table dept
2 ( deptno number(2),
3 dname varchar2(14),
4 loc varchar2(13),
5 last_mod timestamp with time zone
6 default systimestamp
7 not null,
8 constraint dept_pk primary key(deptno)
9 )
10 /
Table created.

• Then we INSERT a copy of the DEPT data into this table:

EODA@ORA12CR1> insert into dept( deptno, dname, loc )
2 select deptno, dname, loc
3 from scott.dept;
4 rows created.
EODA@ORA12CR1> commit
Optimistic Locking Using a Version Column

• That code re-creates the DEPT table, but with an additional LAST_MOD column that uses the `TIMESTAMP WITH TIME ZONE` data type.

• We have defined this column to be NOT NULL so that it must be populated, and its default value is the current system time.

• `TIMESTAMP` data type has the highest precision available in Oracle, typically going down to the microsecond

• For an application that involves user think time, this level of precision on the `TIMESTAMP` is more than sufficient.

• The odds of two people reading and modifying the same row in the same fraction of a second are very small indeed.
Optimistic Locking Using a Version Column

• Maintain this value.

1. Application can maintain the LAST_MOD column by setting its value to SYSTIMESTAMP when it updates a record

2. A trigger/stored procedure
• Trigger will add additional processing on top of that already done by Oracle.
• Each application is responsible for maintaining this field
  – It needs to consistently verify that the LAST_MOD column was not changed and set the LAST_MOD column to the current SYSTIMESTAMP.
• The best way: encapsulating the update logic in a stored procedure and not allowing the application to update the table directly at all.
Optimistic Locking Using a Version Column

Example, if an application queries the row where DEPTNO=10:

```
EODA@ORA12CR1> variable deptno number
EODA@ORA12CR1> variable dname varchar2(14)
EODA@ORA12CR1> variable loc varchar2(13)
EODA@ORA12CR1> variable last_mod varchar2(50)
EODA@ORA12CR1>
EODA@ORA12CR1> begin
  2 :deptno := 10;
  3 select dname, loc, to_char( last_mod, 'DD-MON-YYYY HH.MI.SSXFF AM TZR' )
  4 into :dname,:loc,:last_mod
  5 from dept
  6 where deptno = :deptno;
  7 end;
  8 /
PL/SQL procedure successfully completed.
```

which we can see is currently

```
EODA@ORA12CR1> select :deptno dno, :dname dname, :loc loc, :last_mod lm
  2 from dual;
```

```
DNO DNAME LOC LM
------- ------------ ---------- ----------------------------------------
10 ACCOUNTING NEW YORK 15-APR-2014 07.04.01.147094 PM -06:00
```
Optimistic Locking Using a Version Column

• Update statement to modify the information.
• Last line very important
  – Make sure timestamp has not changed and uses the built-in function TO_TIMESTAMP_TZ (tz is short for time zone) to convert the string we saved in from the SELECT statement back into the proper data type.
• line 3 of the UPDATE statement updates the LAST_MOD column to be the current time if the row is found to be updated:

```
EODA@ORA12CR1> update dept
2   set dname = initcap(:dname),
3   last_mod = systimestamp
4   where deptno = :deptno
5   and last_mod = to_timestamp_tz(:last_mod, 'DD-MON-YYYY HH.MI.SSXFF AM TZR');
1 row updated.
```
Optimistic Locking Using a Version Column

- One row was updated, the row of interest.
- We updated the row by primary key (DEPTNO) and verified that the LAST.MOD column had not been modified by any other session between the time we read it first and the time we did the update.
- If we were to try to update that same record again, using the same logic but without retrieving the new LAST.MOD value:

  ```sql
  EODA@ORA12CR1> update dept
  2 set dname = upper(:dname),
  3 last_mod = systimestamp
  4 where deptno = :deptno
  5 and last_mod = to_timestamp_tz(:last_mod, 'DD-MON-YYYY HH.MI.SSXFF AM TZR');
  0 rows updated.
  ```
- 0 rows updated is reported this time because the predicate on LAST.MOD was not satisfied.
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

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

**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:

\[ \begin{array}{c}
t_{k1} \\
\vdots \\
t_{kn} \\
t_k \\
t_{i1} \\
\vdots \\
t_{in}
\end{array} \]

Case 2:

\[ \begin{array}{c}
t_{k1} \\
\vdots \\
t_{kn} \\
t_k \\
t_{i1} \\
\vdots \\
t_{in}
\end{array} \]
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