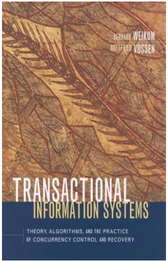
Transactional Information Systems:

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

- 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 8: Concurrency Control on Relational Databases

- **8.2 Predicate-Oriented Concurrency Control**
- 8.3 Relational Update Transactions
- 8.4 Exploiting Transaction-Program Knowledge
- 8.5 Lessons Learned

"Knowledge without wisdom is a load of books on the back of an ass."
(Japanese proverb)

Relational Databases

- Database consists of tables
- Operations on tables and databases are
  - Queries (select-from-where expressions)
  - Insertions
  - Deletions
  - Modifications
- Queries and updates use (single or sets of) predicates or conditions (where clause)
- Sets C of conditions span hyperplanes H(C) of tuples
- Hyperplanes can be subject to locking

Phantom Problem

Example 8.1

Emp	*Name*	*Department*	*Position*	*Salary*
	Jones	Service	Clerk	20000
	Meier	Service	Clerk	22000
	Paulus	Service	Manager	42000
	Smyth	Toys	Cashier	25000
	Brown	Sales	Clerk	28000
	Albert	Sales	Manager	38000

Update transaction t:

(a) Delete From Emp
Where Department = ‘Service’
And Position = ‘Manager’
(b) Insert Into Emp Values
(‘Smith’, ‘Service’, ‘Manager’, 40000)
(c) Update Emp Set Department = ‘Sales’
Where Department = ‘Service’
And Position <> ‘Manager’
(d) Insert Into Emp Values
(‘Stone’, ‘Service’, ‘Clerk’, 13000)

Retrieval transaction q:

Select Name, Position, Salary
From Emp
Where Department = ‘Service’

Retrieval transaction p:

Select Name, Position, Salary
From Emp
Where Department = ‘Sales’

Observations:

- *Interleaving q with t leads to inconsistent read known as “phantom problem”*
- *Locking existing records cannot prevent this problem*

Predicate Locking

- Associate with each operation on table $R(A_1, ..., A_n)$ a set C of conditions that covers a set H(C) of – existing or conceivable – tuples with $H(C) = \{\mu \in dom(A_1) \times ... \times dom(A_n) \mid \mu \text{ satisfies } C\}$
- Each operation locks its H(C)
  [Update operations need to lock pre- and postcondition H(C) and H(C‘)]

Example 8.2:

C_a: Department = ‘Service’ ∧ Position = ‘Manager’
C_b: Name=‘Smith’∧ Department=‘Service’∧ Position=‘Manager’∧ Salary=40000
C_c: Department = ‘Service’∧ Position ≠ ‘Manager’
C_c‘: Department = ‘Sales’∧ Position ≠ ‘Manager’
C_d: Name=‘Stone’∧ Department=‘Service’∧ Position=‘Clerk’∧ Salary=13000
C_q: Department = ‘Service’
C_p: Department = ‘Sales’

$H(C_a) \cap H(C_q) \neq \emptyset$, $H(C_b) \cap H(C_q) \neq \emptyset$, $H(C_c) \cap H(C_q) \neq \emptyset$, $H(C_d) \cap H(C_q) \neq \emptyset$
$H(C_c') \cap H(C_q) = \emptyset$
$H(C_a) \cap H(C_p) = H(C_b) \cap H(C_p) = H(C_c) \cap H(C_p) = H(C_d) \cap H(C_p) = \emptyset$
$H(C_c') \cap H(C_p) \neq \emptyset$

Precision Locking

- Predicate locks on predicates C_t and C_t‘ on behalf of transactions t and t‘ in modes m_t and m_t‘ are compatible if
  - t = t‘ or
  - both m_t and m_t‘ are read (shared) mode or
  - $H(C_t) \cap H(C_t‘) = \emptyset$

- Testing whether $H(C_t) \cap H(C_t‘) = \emptyset$ is NP-complete
- For preventing the phantom problem it is sufficient that
  - queries lock predicates and
  - insert, update, and delete operations lock individual records, and
  - compatibility is checked by testing that an update-affected record does not satisfy any of the query predicate locks

8 Concurrency Control on Relational Databases

- 8.2 Predicate-Oriented Concurrency Control
- 8.3 Relational Update Transactions
- 8.4 Exploiting Transaction-Program Knowledge
- 8.5 Lessons Learned

Idea

- Transactions are sequences of insert, delete, or modify operations (in the style of SQL updates)
- Define notions of serializability along the lines of the classical ones
- The semantic information available on transaction effects can be exploited to allow more concurrency
- Additional concurrency can be allowed by using dependency information, in particular FDs

Transaction Syntax and Semantics

Definition 8.1 (IDM Transaction):
An **IDM transaction** over a database schema D is a finite sequence of update operations (insertions, deletions, modifications) over D.

If $t = u_1 \ldots u_m$ is an IDM transaction over a given database, the effect of t, eff(t), is defined as

$$\mathrm{eff}(t) := \mathrm{eff}[u_1] \circ \ldots \circ \mathrm{eff}[u_m]$$

Insertion:	expression of the form $i_R(C)$, where C specifies a tuple over R
Deletion:	expression of the form $d_R(C)$, where C is a set of conditions
Modification:	expression of the form $m_R(C_1; C_2)$ (tuples satisfying C_1 are modified so that they satisfy C_2)

Transaction Equivalence

Definition 8.2 (Transaction Equivalence):
Two IDM transactions over the same database schema are equivalent, written t ≈ t‘, if eff(t) = eff(t‘), i.e., t and t‘ have the same effect.

Transaction equivalence can be decided in polynomial time:

- using a graphical illustration of transaction effects (“transition specs“)
- using a sound and complete axiomatization of “≈“

We look at the latter (but only at some of the relevant rules)

Commutativity Rules

Let C_1, C_2, C_3, C_4 be sets of conditions describing pairwise disjoint hyperplanes:

1. $i(C_1)\ i(C_2) \approx i(C_2)\ i(C_1)$
2. $d(C_1)\ d(C_2) \approx d(C_2)\ d(C_1)$
3. $d(C_1)\ i(C_2) \approx i(C_2)\ d(C_1)$ if $C_1 <> C_2$
4. $m(C_1; C_2)\ m(C_3; C_4) \approx m(C_3; C_4)\ m(C_1; C_2)$ if $C_3 <> C_1, C_2$ and $C_1 <> C_4$
5. $m(C_1; C_2)\ i(C_3) \approx i(C_3)\ m(C_1; C_2)$ if $C_1 <> C_3$
6. $m(C_1; C_2)\ d(C_3) \approx d(C_3)\ m(C_1; C_2)$ if $C_3 <> C_1, C_2$

Simplification Rules

Let C_1, C_2, C_3, be sets of conditions describing pairwise disjoint hyperplanes:

1. i(C_1) i(C_1) => i(C_1)
2. d(C_1) d(C_1) => d(C_1)
3. i(C_1) d(C_1) => d(C_1)
4. d(C_1) i(C_1) => i(C_1)
5. m(C_1; C_1) => e
6. m(C_1; C_2) i(C_2) => d(C_1) i(C_2)
7. i(C_1) m(C_1; C_2) => m(C_1; C_2) i(C_2)
8. m(C_1; C_2) d(C_1) => m(C_1; C_2)
9. m(C_1; C_2) d(C_2) => d(C_1) d(C_2)
10. d(C_1) m(C_1; C_2) => d(C_1)
11. m(C_1; C_2) m(C_1; C_3) => m(C_1; C_2)
    if C_1 <> C_2
12. m(C_1; C_2) m(C_2; C_3)
    => m(C_1; C_3) m(C_2; C_3)

These rules can be used for transaction optimization.

Final State Serializability

Definition 8.3 (Final State Serializability):
A history s for a set $T = \{ t_1 , \ldots t_n \}$ of IDM transactions is final state serializable if $s \approx s'$ for some serial history s' for T.
Let FSR_{IDM} denote the class of all final state serializable histories (for T).

Example 8.3/4: Let

$t_1 = d(3)\ m(1; 2)\ m(3; 4)$, $t_2 = d(3)\ m(2; 3)$

and consider $s = d_2(3)\ d_1(3)\ m_1(1; 2)\ m_2(2; 3)\ m_1(3; 4)$

s is neither equivalent to $t_1\ t_2$ nor to $t_2\ t_1$; thus, s is not in FSR_{IDM}

However, optimizing t_1 to $d(3)\ m(1; 2)$ yields

$$s' = d_2(3)\ d_1(3)\ m_1(1; 2)\ m_2(2; 3) \approx t_1\ t_2$$

Testing Membership in $\mathrm{FSR}_{\mathrm{IDM}}$

Theorem 8.1:
The problem of testing whether a given history is in $\mathrm{FSR}_{\mathrm{IDM}}$ is NP complete.

Thus, “exact“ testing is no easier than for page model transactions when semantic information is present.

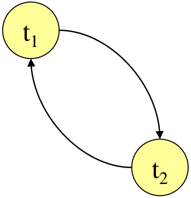
Conflict Serializability

Definition 8.4 (Conflict Serializability):
A history s for a set T of n transactions is conflict serializable if the equivalence of s to a serial history can be proven using the commutativity rules alone.
Let CSR_{IDM} denote the class of all conflict serializable histories (for T).

Definition 8.5 (Conflict Graph):
Let T be a set of IDM transactions and s a history for T. The conflict graph G(s) = (T, E) of s is defined by: (t_i, t_j) is in E if for transactions t_i and t_j in V, i <> j, there is an update u in t_i and an update u‘ in t_j s.t. $u <_s u‘$ and uu‘ is not equivalent to u‘u (i.e., uu‘≈ u‘u does not hold).

Theorem 8.2:
Let s be a history for a set T of transactions. Then s is in CSR_{IDM} iff G(s) is acyclic.

Example 8.6



Consider $s = m_2(1; 2)\ m_1(2; 3)\ m_2(3; 2)$

G(s) is cyclic, so s is **not** in CSR_{IDM}

On the other hand, $s \approx m_1(2; 3)\ m_2(1; 2)\ m_2(3; 2) \approx t_1\ t_2$

so s is in FSR_{IDM}

Consequence: CSR_{IDM} is a strict subset of FSR_{IDM}

Extended Conflict Serializability

Sometimes, the ***context*** in which a conflict occurs can make a difference:

Example: Let

$$s = d_1(0)\ m_1(0; 1)\ m_2(1; 2)\ m_1(2; 3)$$

G(s) is cyclic, but $s \approx m_2(1; 2)\ d_1(0)\ m_1(0; 1)\ m_1(2; 3) \approx t_2\ t_1$

Intutively, the conflict involving $m_1(0; 1)$ does not exist (due to $d_1(0)$) !

Definition 8.6 (Extended Conflict Graph / Serializability):

Let s be a history for a set $T = \{ t_1 , \ldots t_n \}$ of transactions.

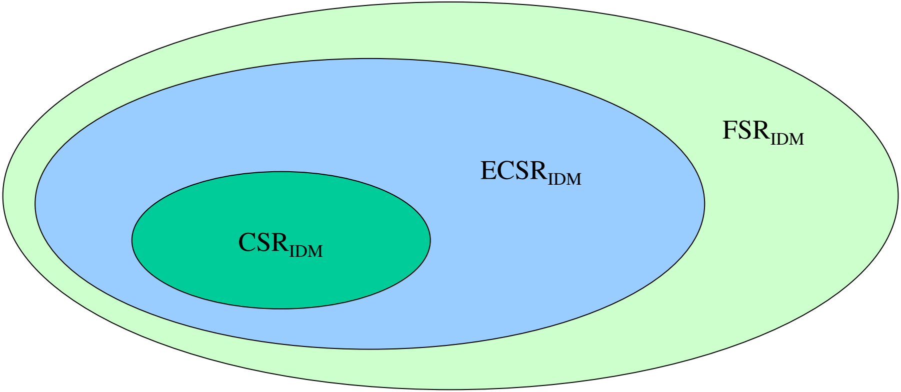
(i) The extended conflict graph EG(s) = (T, E) of s is defined by: (t_i, t_j) is in E if there is an update u in t_j s.t. s = s‘ u s‘‘ and u does not commute with the projection of s‘ onto t_i.

(ii) s is extended conflict serializable if EG(s) is acyclic.

Let $ECSR_{IDM}$ denote the class of all extended conflict serializable histories.

Relationship between the Classes

Theorem 8.3:
$\mathrm{CSR}_{\mathrm{IDM}} \subset \mathrm{ECSR}_{\mathrm{IDM}} \subset \mathrm{FSR}_{\mathrm{IDM}}$.



Serializability w/ Functional Dependencies

Consider a relation with attributes A and B s.t. A-> B holds, and the following history:

$s = m_1(A=0, B=0; A=0, B=2)\ m_2(A=0, B=0; A=0, B=3)$
$m_2(A=0, B=1; A=0, B=3)\ m_1(A=0, B=1; A=0, B=2)$

s is in neither of CSR_{IDM} , $ECSR_{IDM}$, FSR_{IDM} .

However, the first conflict affects (0,0), while the second affects (0,1), and ***these two tuples cannot occur simultaneously in a relation satisfying the given FD***! So depending on the state, $s \approx t_1\ t_2$ or $s \approx t_2\ t_1$.

8 Concurrency Control on Relational Databases

- 8.2 Predicate-Oriented Concurrency Control
- 8.3 Relational Update Transactions
- **8.4 Exploiting Transaction-Program Knowledge**
- 8.5 Lessons Learned

Motivation: Short Transactions Are Good

Example 8.12:

Debit/credit:

t_1: $r(A_1)w(A_1)r(B_1)w(B_1)$

t_2: $r(A_3)w(A_3)r(B_1)w(B_1)$

t_3: $r(A_4)w(A_4)r(B_2)w(B_2)$

Balance:

t_4: $r(A_2)$

t_5: $r(A_4)$

Audit:

t_6: $r(A_1)r(A_2)r(A_3)r(B_1)r(A_4)r(A_5)r(B_2)$

decompose
?

t_{11}: $r(A_1)w(A_1)$

t_{12}: $r(B_1)w(B_1)$

t_{21}: $r(A_3)w(A_3)$

t_{22}: $r(B_1)w(B_1)$

t_{31}: $r(A_4)w(A_4)$

t_{32}: $r(B_2)w(B_2)$

t_{61}: $r(A_1)r(A_2)r(A_3)r(B_1)$

t_{62}: $r(A_4)r(A_5)r(B_2)$

Transaction Chopping

Assumption:
all potentially concurrent app programs are known in advance and
their structure and resulting access patterns can be precisely analyzed

Definition 8.8 (Transaction Chopping):
A **chopping** of transaction t_i is a decomposition of t_i into pieces $t_{i1}, ..., t_{ik}$ s.t. every step of t_i is contained in exactly one piece and the step order is preserved.

Definition 8.10 (Correct Chopping):
A chopping of $T=\{t_1, ..., t_n\}$ is **correct** if every execution of the transaction pieces is conflict-equivalent to a serial history of T under a protocol with

- transaction pieces obey the execution precedences of the original programs.
- each piece is executed as a unit under a CSR scheduler.

Chopping Graph

Definition 8.9 (Chopping Graph):
For a chopping of transaction set T the **chopping graph C(T)** is an undirected graph s.t.

- the nodes of C(T) are the transaction pieces
- for two pieces p, q from different transactions C(T) contains a **c edge** between p and p‘ if p and q contain conflicting operations
- for two pieces p, q from the same transaction C(T) contains an **s edge**

Theorem 8.5:
A chopping is correct if the associated chopping graph does not contain an sc cycle (i.e., a cycle that involves at least one s edge and at least one c edge.

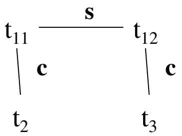
Example 8.13:

$t_1= r(x)w(x)\mathit{r(y)w(y)}$
$t_2= r(x)w(x)$
$t_3= r(y)w(y)$

⇒

$t_{11}= r(x)w(x)$
$t_{12}= \mathit{r(y)w(y)}$

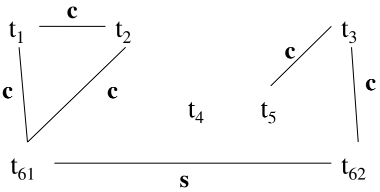
C(T):



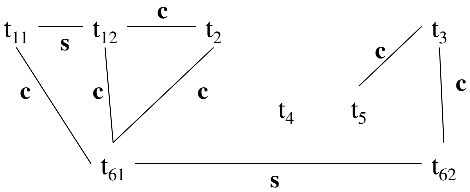
Chopping Example 8.14

t_1: $r(A_1)w(A_1)$*$r(B_1)w(B_1)$*
t_2: $r(A_3)w(A_3)r(B_1)w(B_1)$
t_3: $r(A_4)w(A_4)r(B_2)w(B_2)$
t_4: $r(A_2)$
t_5: $r(A_4)$
t_6: $r(A_1)r(A_2)r(A_3)r(B_1)$*$r(A_4)r(A_5)r(B_2)$*

t_{61}: $r(A_1)r(A_2)r(A_3)r(B_1)$
t_{62}: *$r(A_4)r(A_5)r(B_2)$*



t_{11}: $r(A_1)w(A_1)$
t_{12}: *$r(B_1)w(B_1)$*



Applicability of Chopping

Directly applicable to straight-line, parameter-less SQL programs with predicate locking

Needs to conservatively derive covering program for parameterized SQL, if-then-else and loops, and needs to be conservative about c edges

Example:

```
Select AccountNo From Accounts
Where AccountType=,savings‘ And City = :x;
if not found then
        Select AccountNo From Accounts
        Where AccountType=,checking‘ And City = :x
fi;
```

→

```
Select AccountNo From Accounts
Where AccountType=,savings‘;
Select AccountNo From Accounts
Where AccountType=,checking‘;
```

8 Concurrency Control on Relational Databases

- 8.2 Predicate-Oriented Concurrency Control
- 8.3 Relational Update Transactions
- 8.4 Exploiting Transaction-Program Knowledge
- **8.5 Lessons Learned**

Lessons Learned

- Predicate locking is an elegant method for concurrency control on relational databases, but has non-negligible overhead
  - → record locking (plus index key locking) for 2-level schedules remains the practical method of choice
- Concurrency control may exploit additional knowledge about limited operation types, integrity constraints, and program structure
- Transaction chopping is an interesting tuning technique that aims to exploit such knowledge