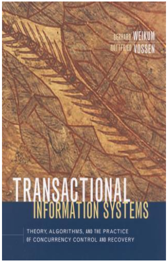
Transactional Information Systems:

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 5: Multiversion Concurrency Control

- **5.2 Multiversion Schedules**
- 5.3 Multiversion Serializability
- 5.4 Limiting the Number of Versions
- 5.5 Multiversion Concurrency Control Protocols
- 5.6 Lessons Learned

"A book is a version of the world. If you do not like it, ignore it; or offer your own version in return." (Salmon Rushdie)

Motivation

Example 5.1:

$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$ $\rightarrow \notin$ CSR

but: schedule would be tolerable
if $r_1(y)$ could read the **old version** y_0 of y
to be consistent with $r_1(x)$

$\rightarrow$ s would then be equivalent to serial $s' = t_1\ t_2$

Motivation

Example 5.1:

$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$ $\rightarrow \notin$ CSR

but: schedule would be tolerable

if $r_1(y)$ could read the **old version** y_0 of y

to be consistent with $r_1(x)$

$\rightarrow$ s would then be equivalent to serial $s' = t_1\ t_2$

Approach:

- each w step creates a new version
- each r step can choose which version it wants/needs to read
- versions are transparent to application and transient (i.e., subject to garbage collection)

Multiversion Schedules

Definition 5.1 (Version Function):
Let s be a history with initial transaction t_0 and final transaction t_∞.
A **version function** for s is a function h which associates with each read step of s a previous write step on the same data item, and the identity for writes.

Multiversion Schedules

Definition 5.1 (Version Function):
Let s be a history with initial transaction t_0 and final transaction t_∞.
A **version function** for s is a function h which associates with each read step of s a previous write step on the same data item, and the identity for writes.

Definition 5.2 (Multiversion Schedule):
A **multiversion (mv) history** for transactions $T = \{t_1, ..., t_n\}$ is a pair $m=(op(m), <_m)$ where $<_m$ is an order on op(m) and

(1) $op(m) = \cup_{i=1..n}\ h(op(t_i))$ for some version function h,
(2) for all $t \in T$ and all $p, q \in op(t_i)$: $p <_t q \Rightarrow h(p) <_m h(q)$,
(3) if $h(r_j(x)) = w_j(x_i)$, $i \neq j$, then c_i is in m and $c_i <_m c_j$.

A **multiversion (mv) schedule** is a prefix of a multiversion history.

Example 5.2: $r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(z_1)\ c_1\ c_2$ with $h(r_1(y)) = w_0(y_0)$

Multiversion Schedules

Definition 5.1 (Version Function):
Let s be a history with initial transaction t_0 and final transaction t_∞.
A **version function** for s is a function h which associates with each read step of s a previous write step on the same data item, and the identity for writes.

Definition 5.2 (Multiversion Schedule):
A **multiversion (mv) history** for transactions $T = \{t_1, ..., t_n\}$ is a pair $m = (op(m), <_m)$ where $<_m$ is an order on op(m) and

(1) $op(m) = \cup_{i=1..n}\; h(op(t_i))$ for some version function h,

(2) for all $t \in T$ and all $p, q \in op(t_i)$: $p <_t q \Rightarrow h(p) <_m h(q)$,

(3) if $h(r_j(x)) = w_j(x_i)$, $i \neq j$, then c_i is in m and $c_i <_m c_j$.

A **multiversion (mv) schedule** is a prefix of a multiversion history.

Example 5.2: $r_1(x_0)\; w_1(x_1)\; r_2(x_1)\; w_2(y_2)\; r_1(y_0)\; w_1(z_1)\; c_1\; c_2$ with $h(r_1(y)) = w_0(y_0)$

Definition 5.3 (Monoversion Schedule):
A multiversion schedule is a **monoversion schedule** if its version function maps each read to the last preceding write on the same data item.

Example: $r_1(x_0)\; w_1(x_1)\; r_2(x_1)\; w_2(y_2)\; r_1(y_2)\; w_1(z_1)\; c_1\; c_2$

Chapter 5: Multiversion Concurrency Control

- 5.2 Multiversion Schedules
- **5.3 Multiversion Serializability**
- 5.4 Limiting the Number of Versions
- 5.5 Multiversion Concurrency Control Protocols
- 5.6 Lessons Learned

Multiversion View Serializability

Definition 5.4 (Reads-from Relation):
For mv schedule m the reads-from relation of m is
RF(m) = $\{(t_i, x, t_j) \mid r_j(x_i) \in op(m)\}$.

Multiversion View Serializability

Definition 5.4 (Reads-from Relation):
For mv schedule m the reads-from relation of m is
RF(m) = $\{(t_i, x, t_j) \mid r_j(x_i) \in op(m)\}$.

Definition 5.5 (View Equivalence):
mv histories m and m' with trans(m)=trans(m') are **view equivalent**,
$m \approx_v m'$, if RF(m) = RF(m').

Multiversion View Serializability

Definition 5.4 (Reads-from Relation):
For mv schedule m the reads-from relation of m is
RF(m) = $\{(t_i, x, t_j) \mid r_j(x_i) \in op(m)\}$.

Definition 5.5 (View Equivalence):
mv histories m and m' with trans(m)=trans(m') are **view equivalent**,
$m \approx_v m'$, if RF(m) = RF(m').

Definition 5.6 (Multiversion View Serializability):
m is multiversion view serializable if there is a serial monoversion history m'
s.t. $m \approx_v m'$.
MVSR is the class of multiversion view serializable histories.

Multiversion View Serializability

Definition 5.4 (Reads-from Relation):
For mv schedule m the reads-from relation of m is
RF(m) = $\{(t_i, x, t_j) \mid r_j(x_i) \in op(m)\}$.

Definition 5.5 (View Equivalence):
mv histories m and m' with trans(m)=trans(m') are **view equivalent**,
$m \approx_v m'$, if RF(m) = RF(m').

Definition 5.6 (Multiversion View Serializability):
m is multiversion view serializable if there is a serial monoversion history m'
s.t. $m \approx_v m'$.
MVSR is the class of multiversion view serializable histories.

Example 5.5:
$m = w_0(x_0)\ w_0(y_0)\ c_0\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_0)\ r_2(y_1)\ c_2$ $\quad \notin$ MVSR

Example 5.6:
$m = w_0(x_0)\ w_0(y_0)\ c_0\ w_1(x_1)\ c_1\ r_2(x_1)\ r_3(x_0)\ w_3(x_3)\ c_3\ w_2(y_2)\ c_2$ $\quad \approx_v t_0\ t_3\ t_1\ t_2$

Properties of MVSR

Theorem 5.1: VSR $\subset$ MVSR

Example: $s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Properties of MVSR

Theorem 5.1: VSR $\subset$ MVSR

Example: s = $r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Theorem 5.2: Deciding if a mv history is in MVSR is NP-complete.

Properties of MVSR

Theorem 5.1: VSR $\subset$ MVSR

Example: $s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Theorem 5.2: Deciding if a mv history is in MVSR is NP-complete.

Theorem 5.3:
The conflict graph of an mv schedule m is a directed graph G(m) with transactions as nodes and an edge from t_i to t_j if $r_j(x_i) \in op(m)$.
For all mv schedules m, m': $m \approx_v m' \Rightarrow G(m) = G(m')$.

Example:

$$\left.\begin{array}{l} m = w_1(x_1)\ r_2(x_0)\ w_1(y_1)\ r_2(y_1)\ c_1\ c_2 \\ m' = w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_1)\ r_2(y_0)\ c_2 \end{array}\right\} \begin{array}{l} G(m) = G(m'), \\ \text{but not } m \approx_v m' \end{array}$$

Testing MVSR

Definition 5.8 (Multiversion Serialization Graph (MVSG)):
A version order for data item x, denoted $<<_x$, is a total order among all versions of x.
A **version order** for mv schedule m is the
union of version orders for items written in m.
The **mv serialization graph** for m and a given version order <<, **MVSG (m, <<),**
is a graph with transactions as nodes and the following edges:

(i) all edges of G(m) are in MVSG(m, <<)
(i.e., for $r_k(x_j)$ in op(m) there is an edge from t_j to t_k)

(ii) for $r_k(x_j)$, $w_i(x_i)$ in op(m): if $x_i << x_j$ then there is an edge from t_i to t_j

(iii) for $r_k(x_j)$, $w_i(x_i)$ in op(m): if $x_j << x_i$ then there is an edge from t_k to t_i

Testing MVSR

Definition 5.8 (Multiversion Serialization Graph (MVSG)):
A version order for data item x, denoted $<<_x$, is a total order among all versions of x.
A **version order** for mv schedule m is the
union of version orders for items written in m.
The **mv serialization graph** for m and a given version order <<, **MVSG (m, <<),**
is a graph with transactions as nodes and the following edges:

(i) all edges of G(m) are in MVSG(m, <<)
(i.e., for $r_k(x_j)$ in op(m) there is an edge from t_j to t_k)

(ii) for $r_k(x_j)$, $w_i(x_i)$ in op(m): if $x_i << x_j$ then there is an edge from t_i to t_j

(iii) for $r_k(x_j)$, $w_i(x_i)$ in op(m): if $x_j << x_i$ then there is an edge from t_k to t_i

Theorem 5.4:
m is in MVSR iff there exists a version order << s.t. MVSG(m, <<) is acyclic.

MVSG Example

Examples 5.7 and 5.8:

$m = w_0(x_0)\ w_0(y_0)\ w_0(z_0)\ c_0$
$\mathbf{r_1(x_0)}\ r_2(x_0)\ r_2(z_0)\ r_3(z_0)$
$w_1(y_1)\ \mathbf{w_2(x_2)}\ w_3(y_3)\ w_3(z_3)\ c_1\ c_2\ c_3$
$r_4(x_2)\ r_4(y_3)\ r_4(z_3)\ c_4$

with version order <<:
$x_0 << x_2$
$y_0 << y_1 << y_3$
$z_0 << z_3$

MVSG(m, <<):

$$\begin{tikzcd}
 & t_1 \arrow[d] \arrow[drr] & & \\
t_0 \arrow[ur] \arrow[rrr] \arrow[dr] & & & t_3 \arrow[r] & t_4 \\
 & t_2 \arrow[urr] \arrow[urrr] & & &
\end{tikzcd}$$

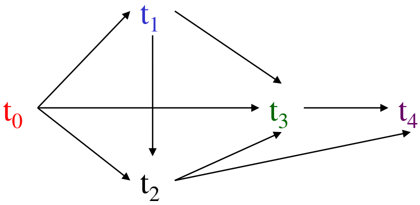
MVSG Example

Examples 5.7 and 5.8:

$m = w_0(x_0)\ w_0(y_0)\ w_0(z_0)\ c_0$
$\quad \mathbf{r_1(x_0)}\ r_2(x_0)\ r_2(z_0)\ r_3(z_0)$
$\quad w_1(y_1)\ \mathbf{w_2(x_2)}\ w_3(y_3)\ w_3(z_3)\ c_1\ c_2\ c_3$
$\quad r_4(x_2)\ r_4(y_3)\ r_4(z_3)\ c_4$

with version order <<:
$x_0 \ll x_2$
$y_0 \ll y_1 \ll y_3$
$z_0 \ll z_3$

MVSG(m, <<):



Notice: Testing whether appropriate << exists for given m is not necessarily polynomial ⇒ NP-completeness result remains

Multiversion Conflict Serializability

Definition 5.9 (Multiversion Conflict):
A **multiversion conflict** in m is a pair $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

Multiversion Conflict Serializability

Definition 5.9 (Multiversion Conflict):
A **multiversion conflict** in m is a pair $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

Definition 5.10 (Multiversion Reducibility):
An mv history is **multiversion reducible** if it can be transformed into a serial monoversion history by exchanging the order of adjacent steps other than multiversion conflict pairs.

Multiversion Conflict Serializability

Definition 5.9 (Multiversion Conflict):
A **multiversion conflict** in m is a pair $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

Definition 5.10 (Multiversion Reducibility):
An mv history is **multiversion reducible** if it can be transformed into a serial monoversion history by exchanging the order of adjacent steps other than multiversion conflict pairs.

Definition 5.11 (Multiversion Conflict Serializability):
An mv history is **multiversion conflict serializable** if there is a serial monoversion history with the same transactions and the same (ordering of) multiversion conflict pairs.
MCSR denotes the class of all multiversion conflict serializable histories.

Multiversion Conflict Serializability

Definition 5.9 (Multiversion Conflict):
A **multiversion conflict** in m is a pair $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

Definition 5.10 (Multiversion Reducibility):
An mv history is **multiversion reducible** if it can be transformed into a serial monoversion history by exchanging the order of adjacent steps other than multiversion conflict pairs.

Definition 5.11 (Multiversion Conflict Serializability):
An mv history is **multiversion conflict serializable** if there is a serial monoversion history with the same transactions and the same (ordering of) multiversion conflict pairs.
MCSR denotes the class of all multiversion conflict serializable histories.

Definition 5.12 (Multiversion Conflict Graph):
For an mv schedule m the **multiversion conflict graph** is a graph with transactions as nodes and an edge from t_i to t_k if there are steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

Properties of MCSR

Theorem:
m is in MCSR $\Leftrightarrow$ m is multiversion reducible $\Leftrightarrow$ m's mv conflict graph is acyclic

Properties of MCSR

Theorem:
m is in MCSR $\Leftrightarrow$ m is multiversion reducible $\Leftrightarrow$ m's mv conflict graph is acyclic

Theorem 5.6:
MCSR $\subset$ MVSR

Example:

$m = w_0(x_0)\ w_0(y_0)\ w_0(z_0)\ c_0\ \underline{r_2(y_0)}\ r_3(z_0)\ w_3(x_3)\ c_3\ \underline{r_1(x_3)}\ \underline{w_1(y_1)}\ c_1\ \underline{w_2(x_2)}\ c_2$
$\quad r_\infty(x_2)\ r_\infty(y_1)\ r_\infty(z_0)\ c_\infty$

$\rightarrow\ \notin$ MCSR

$\rightarrow\ \in$ MVSR

$m \approx_v t_0\ t_3\ t_2\ t_1\ t_\infty$

Chapter 5: Multiversion Concurrency Control

MVTO Protocol

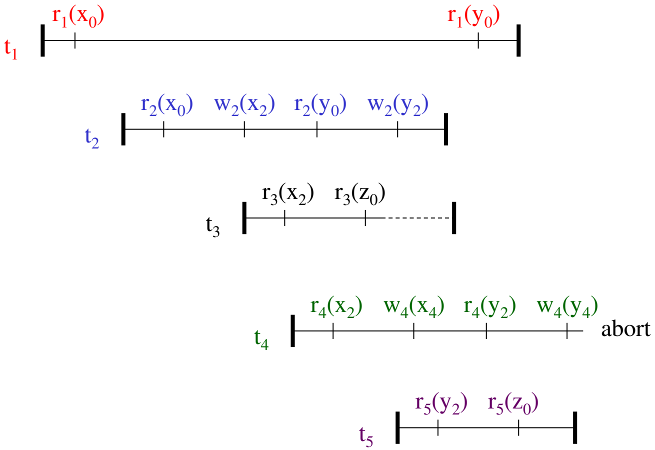
Multiversion timestamp ordering (MVTO):

- each transaction t_i is assigned a unique timestamp $ts(t_i)$
- $r_i(x)$ is mapped to $r_i(x_k)$ where x_k is the version that carries the largest timestamp $\leq ts(t_i)$
- $w_i(x)$ is
  - rejected if there is $r_j(x_k)$ with $ts(t_k) < ts(t_i) < ts(t_j)$
  - mapped into $w_i(x_i)$ otherwise
- c_i is delayed until c_j of all transactions t_j that have written versions read by t_i

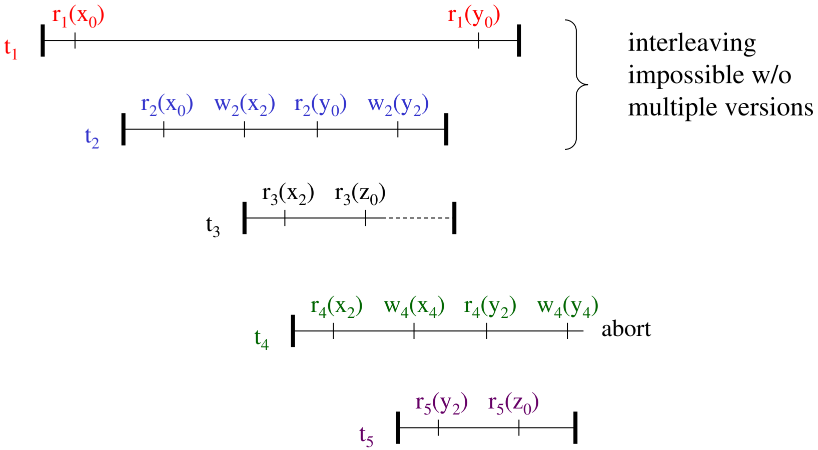
Correctness of MVTO (i.e., Gen(MVTO) $\subseteq$ MVSR):

$$x_i << x_j \Leftrightarrow ts(t_i) < ts(t_j)$$

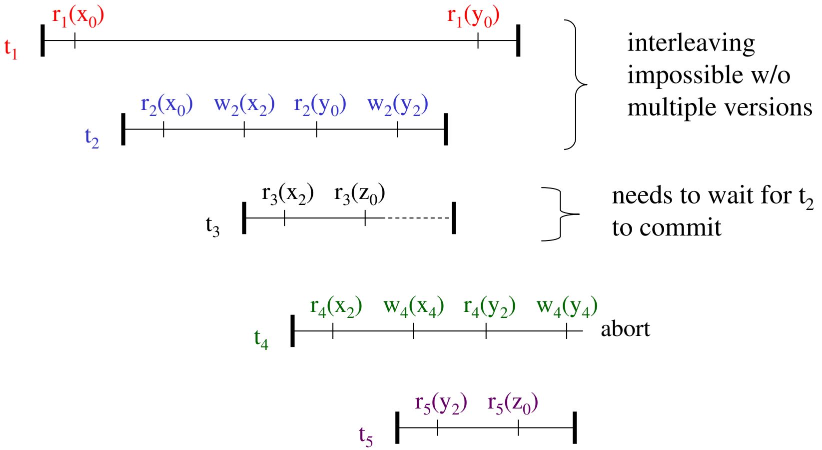
MVTO Example



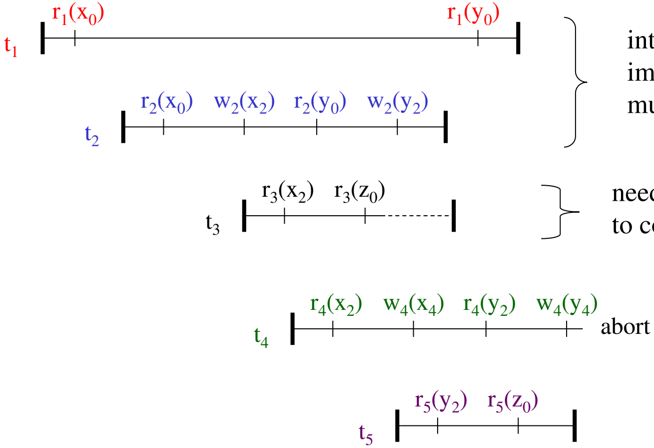
MVTO Example



MVTO Example



MVTO Example



interleaving impossible w/o multiple versions

needs to wait for t_2 to commit

since last write is too late (in the presence of t_5)

Multiversion 2PL (MV2PL) Protocol

General approach:

- use write locking to ensure that
  at each time there is at most one uncommitted version
- for t_i that is not yet issuing its final step:
  - $r_i(x)$ is mapped to "current version" (i.e., the most recent committed version)
    or an uncommitted version
  - $w_i(x)$ is executed only if x is not write-locked, otherwise it is blocked
- t_i's final step is delayed until after the commit of:
  - all t_j that have read from a current version of a data item that t_i has written
  - all t_j from which t_i has read

Multiversion 2PL (MV2PL) Protocol

General approach:

- use write locking to ensure that
  at each time there is at most one uncommitted version
- for t_i that is not yet issuing its final step:
  - $r_i(x)$ is mapped to "current version" (i.e., the most recent committed version)
    or an uncommitted version
  - $w_i(x)$ is executed only if x is not write-locked, otherwise it is blocked
- t_i's final step is delayed until after the commit of:
  - all t_j that have read from a current version of a data item that t_i has written
  - all t_j from which t_i has read

Example 5.9:

for input schedule

$$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$$

MV2PL produces the output schedule

$$r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(y_1)\ c_1\ w_2(x_2)\ c_2$$

Specialization: 2V2PL Protocol

2-Version (before/after image) 2PL:

- request **write lock $wl_i(x)$** for writing a new uncommitted version and ensuring that at most one such version exists at any time
- request **read lock $rl_i(x)$** for reading the current version (i.e., most recent committed version)
- request **certify lock $cl_i(x)$** for final step of t_i on all data items in t_i's write set

lock holder \ lock requestor	$rl_j(x)$	$wl_j(x)$	$cl_j(x)$
$rl_i(x)$	+	+	–
$wl_i(x)$	+	–	–
$cl_i(x)$	–	–	–

Correctness of 2V2PL (i.e., Gen(2V2PL) $\subseteq$ MVSR):

$$x_i \ll x_j \Leftrightarrow f_i < f_j \text{ (for final ``certify'' steps of } t_i, t_j)$$

2V2PL Example

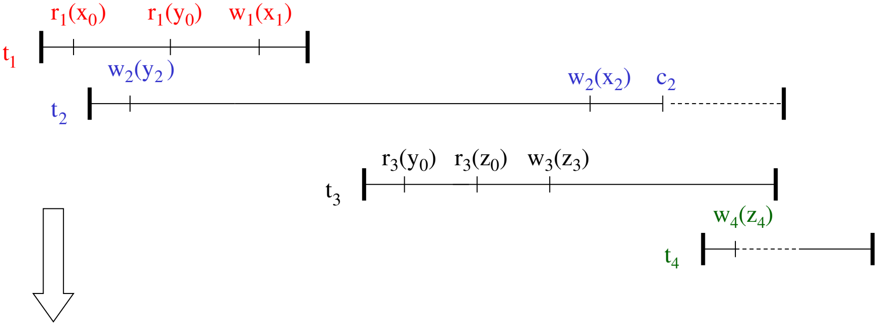
Example 5.10:

s = $r_1(x)$ $w_2(y)$ $r_1(y)$ $w_1(x)$ c_1 $r_3(y)$ $r_3(z)$ $w_3(z)$ $w_2(x)$ c_2 $w_4(z)$ c_4 c_3

2V2PL Example

Example 5.10:

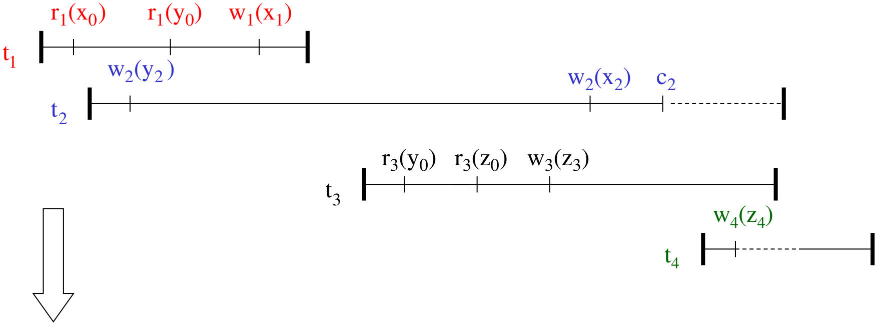
s = $r_1(x)$ $w_2(y)$ $r_1(y)$ $w_1(x)$ c_1 $r_3(y)$ $r_3(z)$ $w_3(z)$ $w_2(x)$ c_2 $w_4(z)$ c_4 c_3



2V2PL Example

Example 5.10:

s = $r_1(x)$ $w_2(y)$ $r_1(y)$ $w_1(x)$ c_1 $r_3(y)$ $r_3(z)$ $w_3(z)$ $w_2(x)$ c_2 $w_4(z)$ c_4 c_3



$rl_1(x)$ $r_1(x_0)$ $wl_2(y)$ $w_2(y_2)$ $rl_1(y)$ $r_1(y_0)$ $wl_1(x)$ $w_1(x_1)$ $cl_1(x)$ u_1 c_1
$rl_3(y)$ $r_3(y_0)$ $rl_3(z)$ $r_3(z_0)$ $wl_2(x)$ $cl_2(x)$ $wl_3(y)$ $w_3(z_3)$ $cl_3(y)$ u_3 c_3
$cl_2(y)$ u_2 c_2 $wl_4(z)$ $w_4(z_4)$ $cl_4(z)$ u_4 c_4

Multiversion Serialization Graph Testing (MVSGT)

Idea:

build version order and MVSG simultaneously (and incrementally)

Protocol rules:

- $r_i(x)$ is mapped to $r_i(x_j)$ such that
  - there is no path $t_j \rightarrow \ldots \rightarrow t_k \rightarrow \ldots \rightarrow t_i$ with previous $w_k(x_k)$ (eliminate "too old" transactions)
  - there is no path $t_i \rightarrow \ldots \rightarrow t_j$ (eliminate "too young" transactions)

  abort t_i if no such t_j exists
- upon $w_i(x_i)$
  add edges $t_j \rightarrow t_i$ for all t_j with previous $r_j(x_k)$
  abort t_i when detecting cycle
- upon $r_i(x_j)$
  add edge $t_j \rightarrow t_i$ and
  edges $t_k \rightarrow t_j$ or $t_i \rightarrow t_k$ for all t_k with previous $w_k(x_k)$

ROMV Protocol

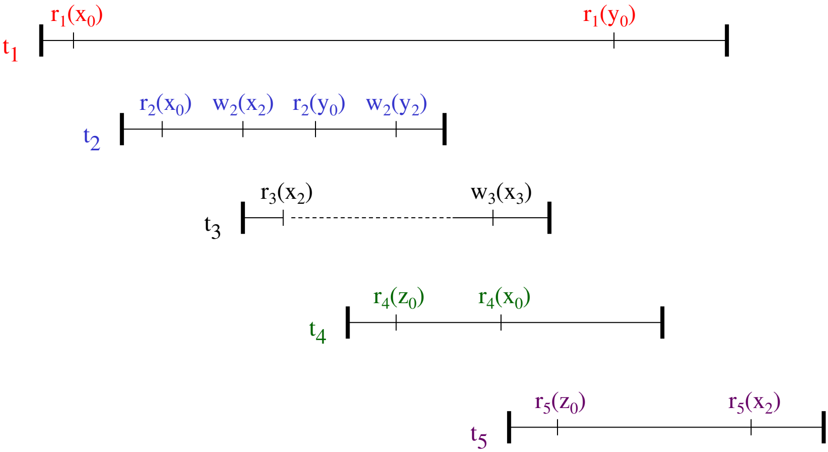
Read-only Multiversion Protocol (ROMV):

- each update transactions uses 2PL on both its read and write set but each write creates a new version and timestamps it with the transaction's commit time
- each read-only transaction t_i is timestamped with its begin time
- $r_i(x)$ is mapped to $r_i(x_k)$ where x_k is the version that carries the largest timestamp $\leq ts(t_i)$ (i.e., the most recent committed version as of the begin of t_i)

Correctness of ROMV (i.e., Gen(ROMV) $\subseteq$ MVSR):

$$x_i << x_j \;\Leftrightarrow\; c_i < c_j$$

ROMV Example



Chapter 5: Multiversion Concurrency Control

Lessons Learned

- Transient and transparent versioning adds a degree of freedom to concurrency control protocols, making MVSR considerably more powerful than VSR
- The most striking benefit is for long read transactions that execute concurrently with writers.
- This specific benefit is achieved with relatively simple protocols like ROMV.

Summary

- Concurrency control in the page model allows for many approaches, yet locking dominates
- Non-locking algorithms may be used in special situations
- Multiple versions can help making concurrency control more flexible