

## Chapter 7

---

# Synchronization: Multiversion Concurrency Control

# Multiversion serializability

# Motivation (1)

## Example 7.1:

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

non-repeatable (inconsistent) read

$s = r_1(x) w_1(x) r_1(y) w_1(z) c_1 r_2(x) w_2(y) c_2 \rightarrow \in \text{CSR (following 2PL)}$

no concurrency  $\Rightarrow$  inefficient!

$s = r_1(x) w_1(x) r_2(x) r_1(y) w_2(y) c_2 w_1(z) c_1 \rightarrow \in \text{CSR}$

repeatable (consistent) read

## Idea for better concurrency:

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

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

Obtain this effect by reading state(y) at beginning of  $t_1$

# Motivation (2)

## Basic solution:

Each write produces a new version of a data element.

- Extend construction of *CSR* histories by maintaining several „versions“ of  $x$  and  $y$ .

## Which version to read?

- Example 1: Read the value valid at the beginning of the transaction:

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

$$G(h') = t_1 t_2 \Rightarrow h' \in CSR$$

- Example 2: Read the currently valid value:

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

delay!

non-repeatable read

# Motivation (3)

- Increased concurrency because multiversion schedulers can generate (additional) schedules that are impossible if only a single version is available.
- Multiversion schedulers are part of many commercial database management systems.
- *Two-version* schedules versions even come for free:
  - ◆ To achieve rollback under recovery, database must register the value of a data element before it is subjected to change  
⇒ at the minimum there are always two versions of an element.

# Multiversion histories (1)

We start with the **general case** of an unlimited number of versions:

- Each write produces a new version of a data element.
  - ◆ The older versions are not overwritten but are kept in the database.
  - ◆ Given data element  $x$ , then  $x_i, x_j, \dots$  denote versions of  $x$  where the index refers to the transaction that wrote the version.
  - Each write of  $x$  in a multiversion schedule is of the form  $w_i(x_j)$ .
- Each read has a free choice of the version it wishes to access.
  - ◆ Read is denoted by  $r_i(x_j)$ .
  - We need initial versions  $\Rightarrow$  transaction  $t_0$ .

# Multiversion histories (2)

## Definition 7.2 (Version function):

Let  $h$  be a (version free) history with initialization transaction  $t_0$  (write of initial elements).

A **version function** for  $h$  is a function  $f$  which translates

1.  $w_i(x)$  to  $w_i(x_i)$ ,
2.  $r_i(x)$  to  $r_i(x_j)$  for some  $j$ ,
3.  $c_i$  to  $c_i$  and
4.  $a_i$  to  $a_i$

We leave open which one  
 $\Rightarrow$  leaves some latitude for  
the ordering of operations!

# Multiversion histories (3)

## Definition 7.3:

A **multiversion (mv) history** for transactions  $T = \{t_1, \dots, t_n\}$  is a history  $m = (op(m), <_m)$ ,  $<_m$  an order on  $op(m)$ , with the (additional) properties

(1)  $op(m) = \bigcup_{i=1}^n f(op(t_i))$  for some version function  $f$ ,

(2) for all  $t_i$  and all  $p, q \in op(t_i)$ :  $p <_{t_i} q \Rightarrow f(p) <_m f(q)$ ,

(3)  $f(r_j(x)) = r_j(x_i) \succ w_i(x_i) <_m r_j(x_i)$

(4)  $f(r_j(x)) = r_j(x_i), i \neq j, c_j \in m \succ c_i \in m \wedge c_i <_m c_j$

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

For each operation of a transaction there is a versioned operation in the mv history.

The version function must maintain the order of operations within a transaction.

Values read by a successful transaction must have been valid.

A bit weaker than before!

# Multiversion histories (4)

Remember

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

Rewritten ( $t_0$  ignored):

$$m = r_1(x_0) w_1(x_1) r_2(x_0) w_2(y_2) r_1(y_0) c_2 w_1(z_1) c_1$$

where  $f(w_i(e)) = w_i(e|_j)$

$$f(r_1(y)) = r_1(y_0)$$

$$f(r_2(x)) = r_2(x_0)$$

$$t_1 \quad t_2$$

Take instead:

$$m = r_1(x_0) w_1(x_1) r_2(x_1) w_2(y_2) r_1(y_0) w_1(z_1) c_1 c_2$$

where  $f(w_i(e)) = w_i(e|_j)$

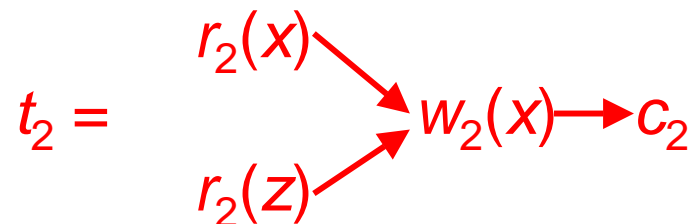
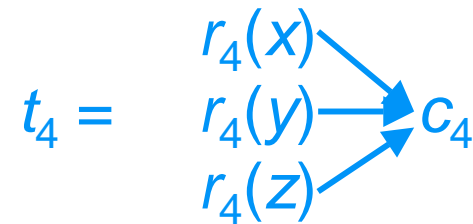
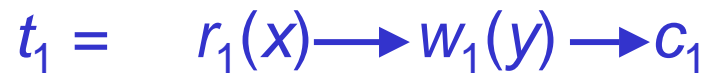
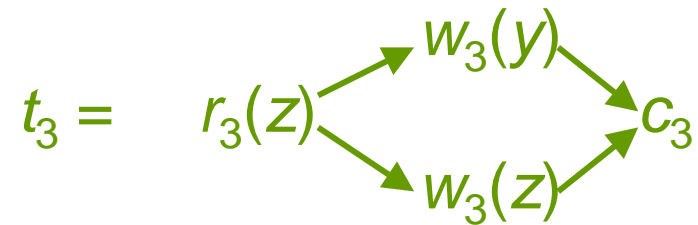
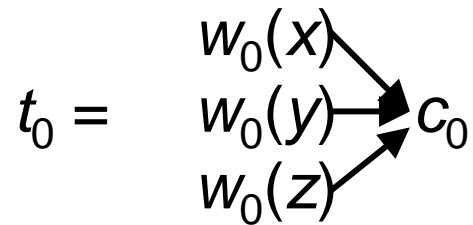
$$f(r_1(y)) = r_1(y_0)$$

$$f(r_2(x)) = r_2(x_1)$$

$$t_1 \rightarrow t_2$$

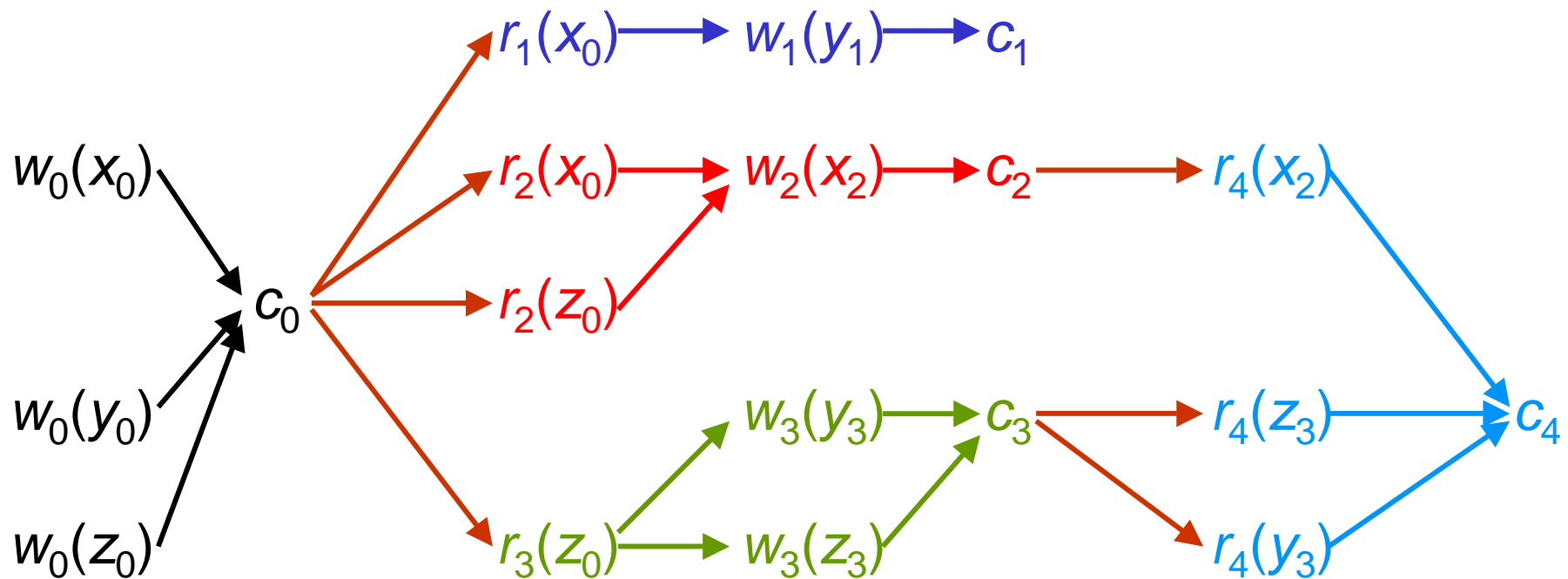
# Multiversion histories (5)

Take transactions:



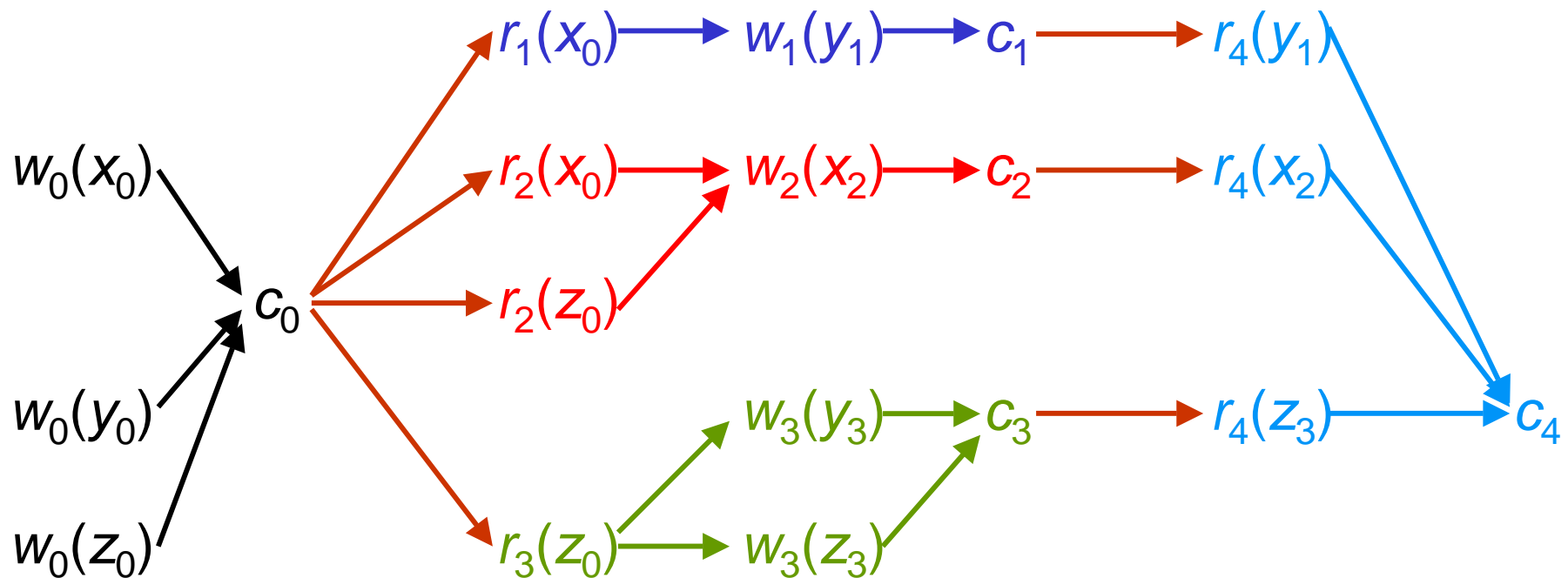
# Multiversion histories (6)

**Example 7.4:**  $h_6$  is a mv history over  $\{t_0, \dots, t_4\}$ :



# Multiversion histories (7)

**Example 7.5:**  $h_7$  is a mv history over  $\{t_0, \dots, t_4\}$  with  $r_4(y_3)$  changed to  $r_4(y_1)$ :



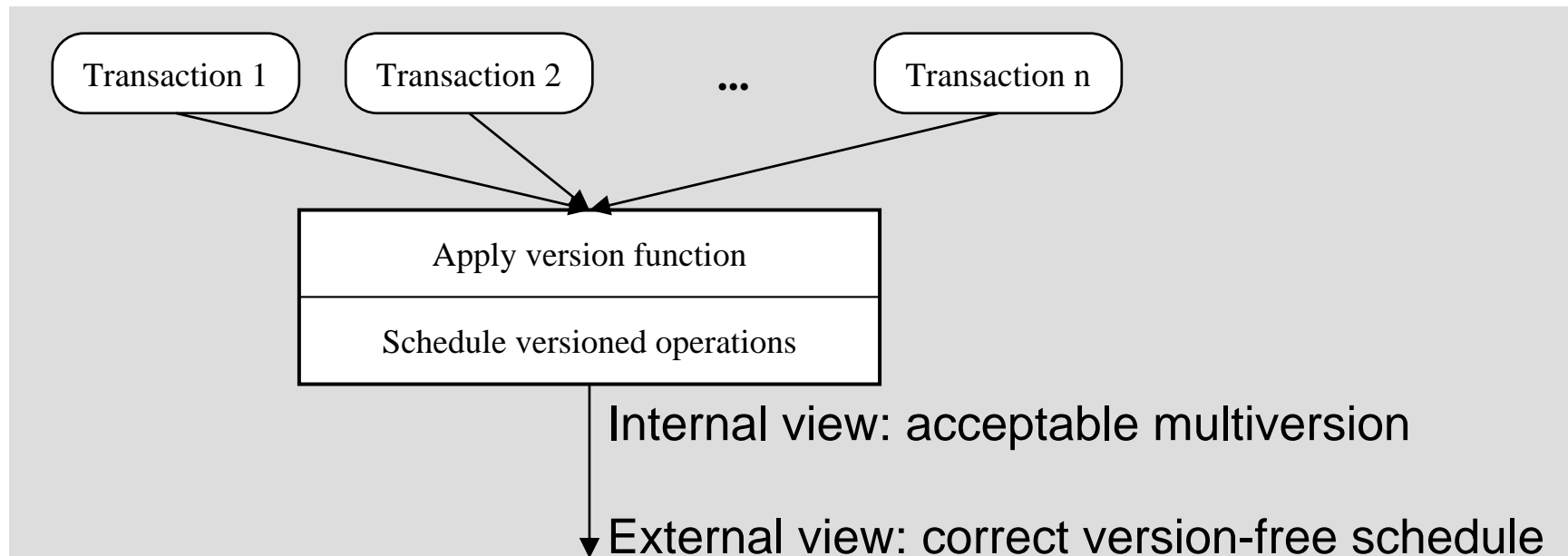
# Multiversion view serializability (1)

13

Which one is “correct”,  $h_6$  or  $h_7$ ?

What do we mean by “correct”?

- Internal view: Versioning is strictly internal to the database system to allow for better concurrency.
- External view: Versioning must remain entirely transparent to the application  $\Rightarrow$  versioning is externally not visible.



# Multiversion view serializability (2)

## Intuitive interpretation of a version-free history:

### ■ Definition 7.6: Monoversion history

A multiversion history is called a **monoversion history** if its version function maps each read step to the last preceding mv compliant write step on the same data item.

$$h = r_1(x) w_1(x) r_2(x) w_2(y) r_1(y) c_2 w_1(z) c_1$$

Monoversion:  $m = r_1(x_0) w_1(x_1) r_2(x_0) w_2(y_2) r_1(y_2) c_2 w_1(z_1) c_1$

$$h = r_1(x) w_1(x) r_2(x) w_2(y) r_1(y) w_1(z) c_1 c_2$$

Monoversion:  $m = r_1(x_0) w_1(x_1) r_2(x_1) w_2(y_2) r_1(y_0) w_1(z_1) c_1 c_2$

# Multiversion view serializability (3)

- Conversely, a monoversion can easily be translated to a version-free history: Just omit the indices.

Monoversion:  $m = r_1(x_0) w_1(x_1) r_2(x_0) w_2(y_2) r_1(y_2) c_2 w_1(z_1) c_1$   
Version-free:  $h = r_1(x) w_1(x) r_2(x) w_2(y) r_1(y) c_2 w_1(z) c_1$

Monoversion:  $m = r_1(x_0) w_1(x_1) r_2(x_1) w_2(y_2) r_1(y_0) w_1(z_1) c_1 c_2$   
Version-free:  $h = r_1(x) w_1(x) r_2(x) w_2(y) r_1(y) w_1(z) c_1 c_2$

# Multiversion view serializability (4)

“Acceptable” multiversion history:

**Definition 7.7 (Reads-from Relation):**

For mv schedule  $m$  the reads-from relation of  $m$ ,  $RF(m)$ ,  
 $t_j \triangleright_m(x) t_i \Leftarrow t_j \triangleright_m(x_i) t_i$

**Definition 7.8 (View Equivalence):**

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

If we do not limit the number of versions,  $t_\infty$  is unnecessary!

**Example 7.9:**  $(m \approx_v m')$

$m = w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_2(x_1) r_3(x_0) w_2(y_2) w_3(x_3) c_3 c_2$   
 $m' = w_0(x_0) w_0(y_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) c_1 r_2(x_1) w_2(y_2) c_2$

# Multiversion view serializability (5)

## Example 7.10:

Note: is not a monoversion!

Multiversion:

$$m = w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_2(x_1) r_3(x_0) w_2(y_2) w_3(x_3) c_3 c_2$$

Equivalent serial multiversion:

$$m' = w_0(x_0) w_0(y_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) c_1 r_2(x_1) w_2(y_2) c_2$$

Equivalent serial monoversion:

$$m'' = w_0(x_0) w_0(y_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) c_1 r_2(x_1) w_2(y_2) c_2$$

Version-free history:

$$h'' = w_0(x) w_0(y) c_0 r_3(x) w_3(x) c_3 w_1(x) c_1 r_2(x) w_2(y) c_2$$

# Multiversion view serializability (6)

## Example 7.11:

Is even a monoversion!

Multiversion:

$m = w_0(x_0) w_0(y_0) c_0 r_1(x_0) r_1(y_0) r_2(x_0) w_1(x_1) w_1(y_1) c_1 r_2(y_1) c_2$

Is no longer a monoversion!

Serial multiversion

$m' = w_0(x_0) w_0(y_0) c_0 r_1(x_0) r_1(y_0) w_1(x_1) w_1(y_1) c_1 r_2(x_0) r_2(y_1) c_2$

But no equivalent serial monoversion can be found (neither/nor):

$m'' = w_0(x_0) w_0(y_0) c_0 r_1(x_0) r_1(y_0) w_1(x_1) w_1(y_1) c_1 r_2(x_1) r_2(y_1) c_2$

$m'' = w_0(x_0) w_0(y_0) c_0 r_2(x_0) r_2(y_0) c_2 r_1(x_0) r_1(y_0) w_1(x_1) w_1(y_1) c_1$

⇒ no transparent versioning!

Existence of a serial multiversion is not a shortcut!

# Multiversion view serializability (7)

## Definition 7.12 (Multiversion View Serializability):

$m$  is **multiversion view serializable** if there is a serial monoversion history  $m'$  for the same set of transactions s.t.  $m \approx_v m'$ .

**MVSR** is the class of multiversion view serializable histories.

# Multiversion view serializability (8)

## Theorem 7.13:

$VSR \subset MVSR$

$m \in MVSR$  because  $t_1 \rightarrow t_2$

Monoversion:  $m = r_1(x_0) w_1(x_1) r_2(x_0) w_2(y_2) r_1(y_2) c_2 w_1(z_1) c_1$

Version-free:  $h = r_1(x) w_1(x) r_2(x) w_2(y) r_1(y) c_2 w_1(z) c_1$

## Theorem 7.14:

Hence  $h \in MVSR$  but  
 $h \notin CSR$  and  $h \notin VSR$

The problem of deciding whether a given multiversion history is in  $MVSR$  is NP-complete.

# Multiversion serialization graph (1)

## Graph-theoretic proof:

We know the order in which versions were produced.

⇒ Check whether the order imposed on the write operations by the scheduler with the effect of a specific version order defines an equivalent monoversion history.

## Definition 7.15 (Version order)

If  $x$  is a data item, a **version order for  $x$**  is any nonreflexive and total ordering of all versions of  $x$  that are written by operations in mv schedule  $m$ . A **version order  $\ll$  for  $m$**  is the union of all version orders of data items written by operations in  $m$ .

## Multiversion serialization graph (2)

22

The only conflicts possible in a multiversion history are *wr*.  
 $\Rightarrow$  A conventional conflict graph  $G(m)$  has an edge from  $t_i$  to  $t_j$  simply if  $r_j(x_j)$  is in  $m$ .

### Theorem 7.16:

For any two mv schedules  $m, m', m \approx_v m' \succ G(m) = G(m')$ .

# Multiversion serialization graph (2)

## Definition 7.17 (MVSG)

For a given mv schedule  $m$  and a version order  $\ll$ , the **multiversion serialization graph MVSG( $m, \ll$ )** is the conflict graph  $G(m) = (V, E)$  with the following edges added for each  $r_k(x_j)$  and  $w_i(x_j)$  in  $CP(m)$  where  $i \neq j \neq k$ :

if  $x_i \ll x_j$ , then add  $t_i \rightarrow t_j$ , else  $t_k \rightarrow t_i$ .

$\Rightarrow t_i < t_j < t_k$   
( $t_j \rightarrow t_k$  already in  $G(m)$ )

monoversion assumption  
 $\Rightarrow t_j < t_k < t_i$   
( $t_j \rightarrow t_k$  already in  $G(m)$ )

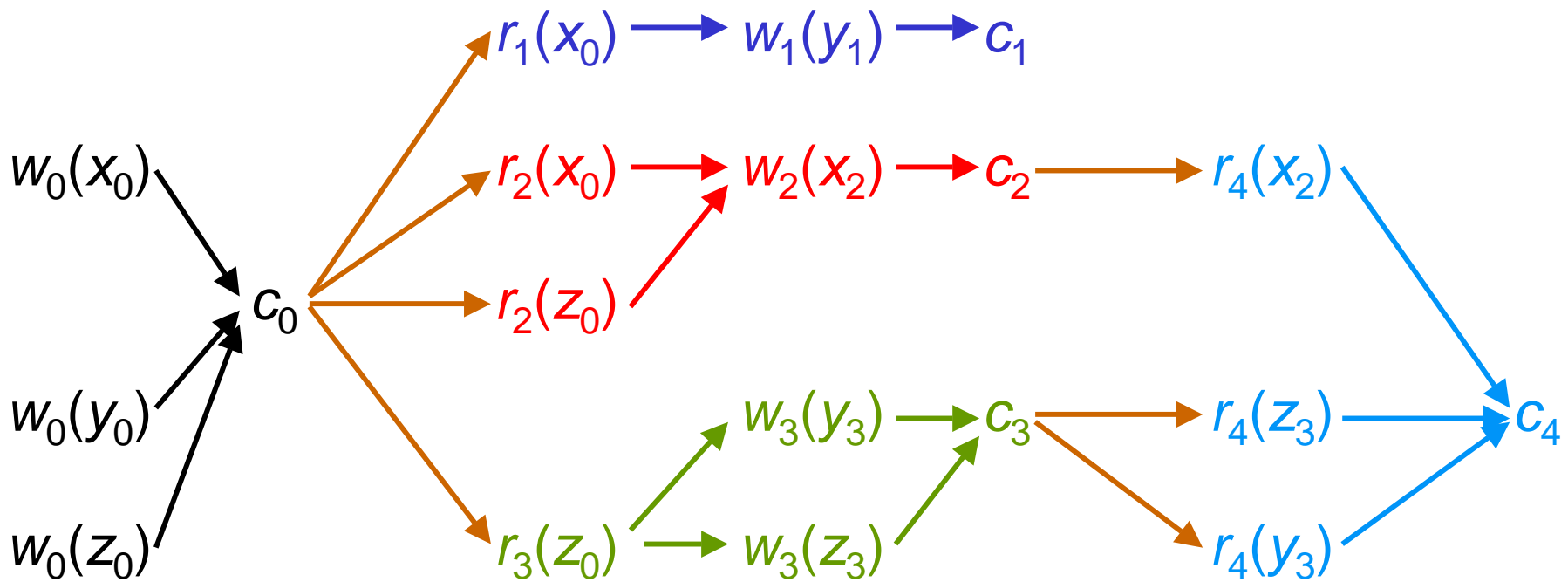
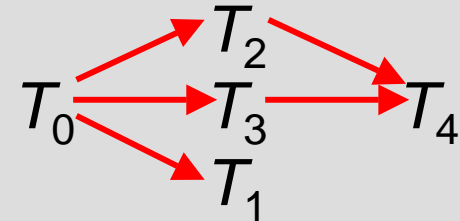
# Multiversion serialization graph (3)

$x_0 \ll x_2, y_0 \ll y_1 \ll y_3, z_0 \ll z_3$

or

$x_0 \ll x_2, y_0 \ll y_3 \ll y_1, z_0 \ll z_3$

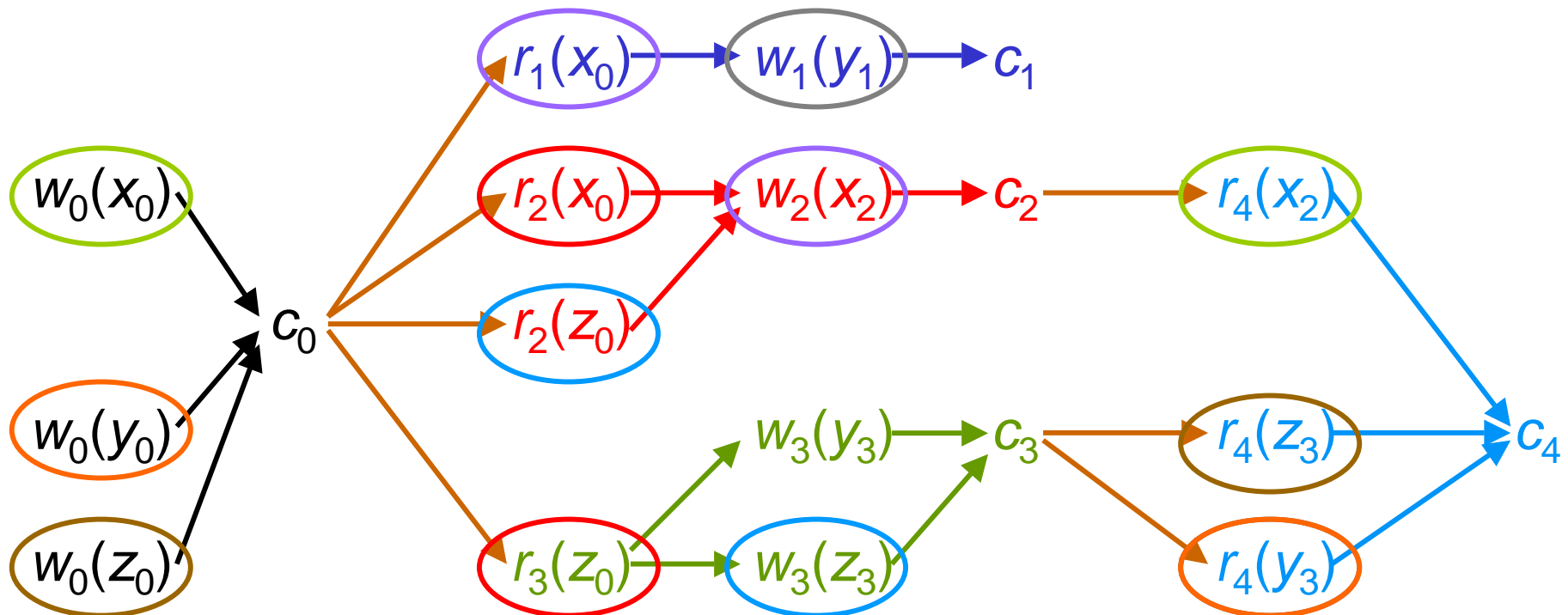
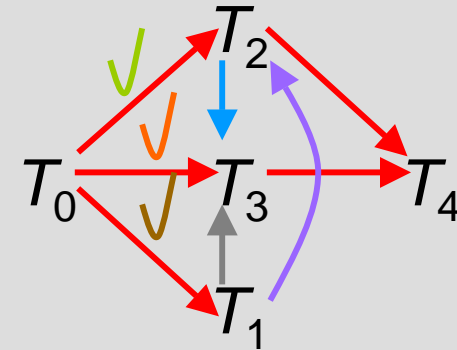
$G(m) =$



# Multiversion serialization graph (4)



$G(m) =$

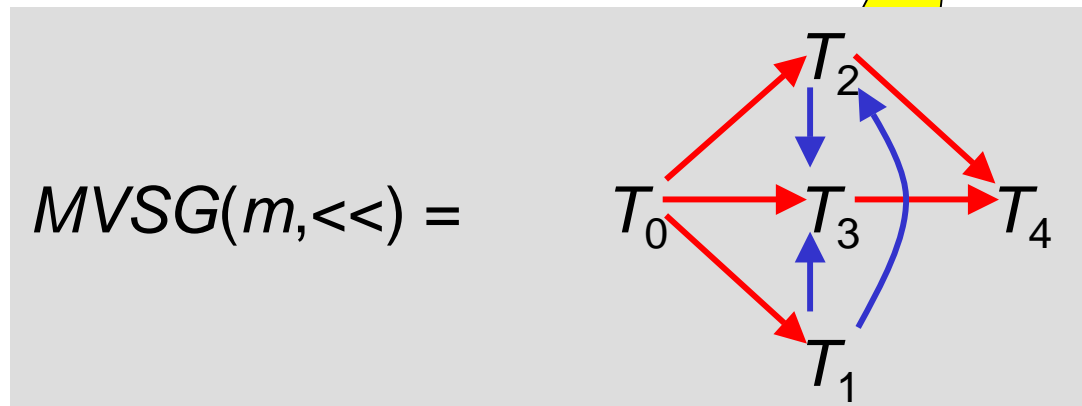


# Multiversion serialization graph (5)

## Theorem 7.18 (MVSR)

mv schedule  $m$  is in *MVSR*  $\Leftrightarrow$  there exists a version order  $\ll$  s.t.  $MVSG(m, \ll)$  is acyclic.

**acyclic!**



Similar to VSR: It cannot necessarily be tested in polynomial time whether a version order of the desired form exists.

Is there some chance to extend *CSR*?

# Multiversion conflict serializability (1)

- **Approach:** Which types of conflict are relevant to mv schedules, i.e., for which pairs of operations do we have to watch out for the same order in the original schedule and in the serial schedule?
  - ◆ *ww*: not a conflict because a new version is written.
  - ◆ *wr*: can be exchanged except where *r* accesses the version written by *w*.
  - ◆ *rw*: critical because an exchange opens up to *r* a larger choice among versions.

## Definition 7.19 (Multiversion Conflict):

A **multiversion conflict** in  $m$  is a pair  $r_i(x_j)$  and  $w_k(x_k)$  such that  $r_i(x_j) <_m w_k(x_k)$  ( $i \neq j \neq k$ ).

# Multiversion conflict serializability (2)

- **Remember:** *CSR* was (also) defined on the basis of reordering, using commutativity-based reducibility.
- We take the same approach (here for the special case of a total order).

## **Definition 7.20 (Multiversion Reducibility):**

A (totally ordered) 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 7.21 (Multiversion Conflict Serializability):**

A mv history  $m$  is **multiversion conflict serializable** if there is a serial monoversion history for the same set of transactions in which all pairs of operations in multiversion conflict occur in the same order as in  $m$ .

# Multiversion conflict serializability (3)

## Graph-theoretic characterization:

### Definition 7.22 (Multiversion Conflict Graph):

Let  $m$  be a mv history. The **multiversion conflict graph**  $G(m) = (V, E)$  is a directed graph with vertices  $V = \text{commit}(m)$  and edges  $E = \{(t_i, t_k) \mid i \neq j\}$  if there are steps  $r_i(x_j)$  and  $w_k(x_k)$  such that  $r_i(x_j) <_m w_k(x_k)$ .

### Theorem 7.23:

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

tested in polynomial time.

# Multiversion conflict serializability (3)

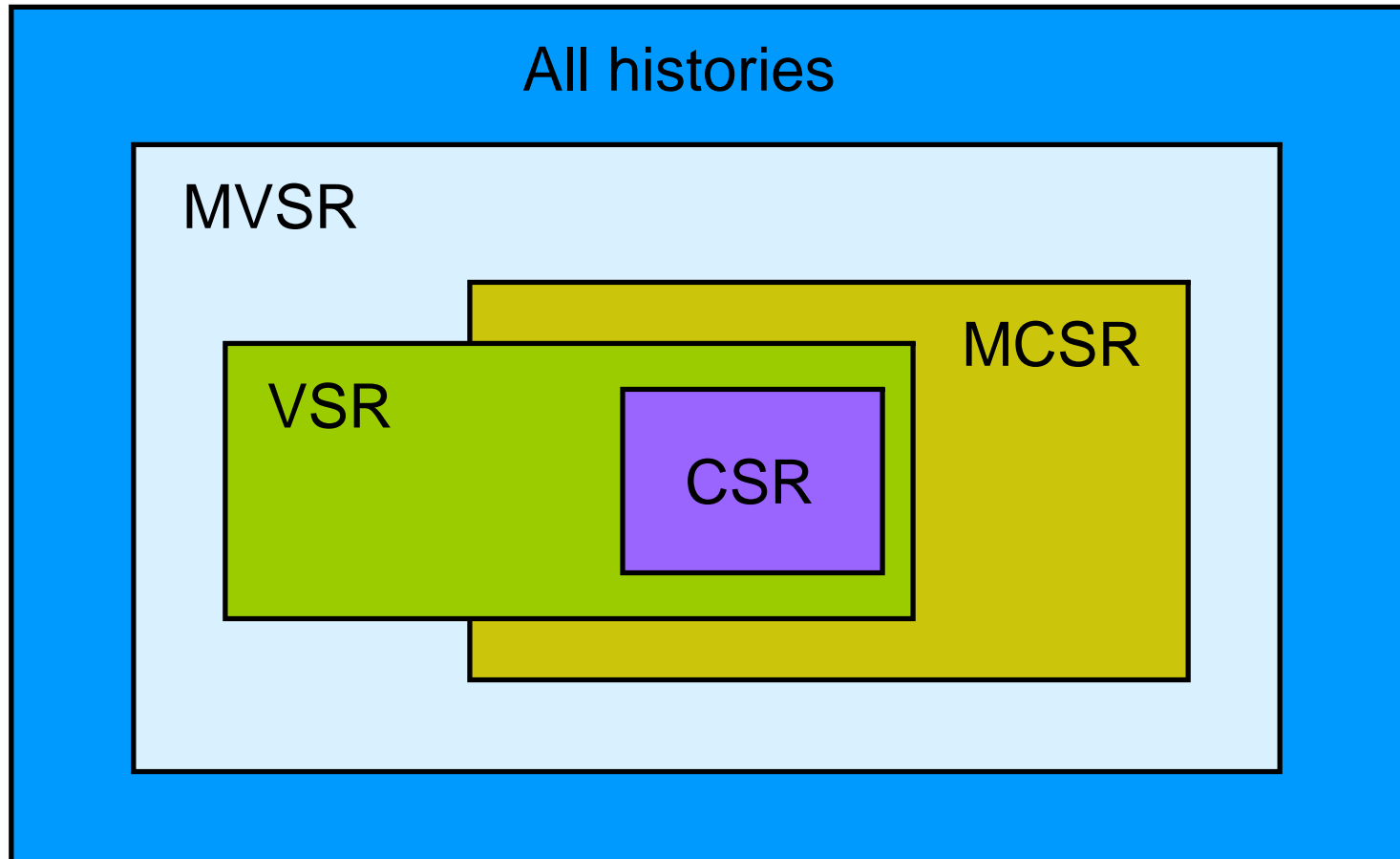
## Remark 7.24:

- *MCSR*: class of all multiversion conflict serializable histories.
- $MCSR \subset MVSR$
- *MCSR* has a polynomial membership test.
- The equivalent serial monoversion history cannot simply be derived by topologically sorting the graph. Further information is needed: version order.



Proofs still rely on *MVSG*.

# Summary



# Multiversion schedulers

# Multiversion 2PL (MV2PL)

- Protocol uses SS2PL.
- Modifications:
  - ◆ a simpler conflict matrix,
  - ◆ but version order must be tracked in addition.
- We restrict ourselves to the particularly simple case of 2 versions (2V2PL).

# 2-version 2PL (1)

## Adjustments to 2PL:

- 2PL: A w/ lock is exclusive. In particular, it prohibits concurrent reads. By using two versions we wish to relax the condition.
- If  $t_i$  writes  $x$ :
  - ◆ a new version  $x_i$  is created;
  - ◆ a lock must be found for  $x$  that inhibits reads on  $x_i$  and the creation of further versions for  $x$  (2-version 2PL!), and allows other transactions to read the single old  $x$ .
- On commit of  $t_i$   $x_i$  becomes the current version and the old version of  $x$  becomes inaccessible.
- Conclusion: Use 2PL for ww synchronization, and version selection für rw synchronization.

# 2-version 2PL (2)

## Adjustments to the locks:

2V2PL uses 3 kinds of locks: read, write and certify (or commit) locks.

Lock control:

- As before for *rl* und *wl*, controlled by the compatibility matrix.
- On commit: all *wl* locks turn into certify locks (*cl*).

		lock holder		
		read	write	certify
lock request	read	y	y	n
	write	y	n	n
	certify	n	n	n



# 2-version 2PL (4)

Only disadvantage: Wait for the end of concurrent readers on commit of a write transaction.

Certify locks behave like write locks in 2PL, however the time over which they are held is much shorter  $\Rightarrow$  advantage over 2PL. Before certify: write + concurrent reads.

		lock	write	certify
lock	read	y	y	n
request	write	y	n	n
	certify	n	n	n

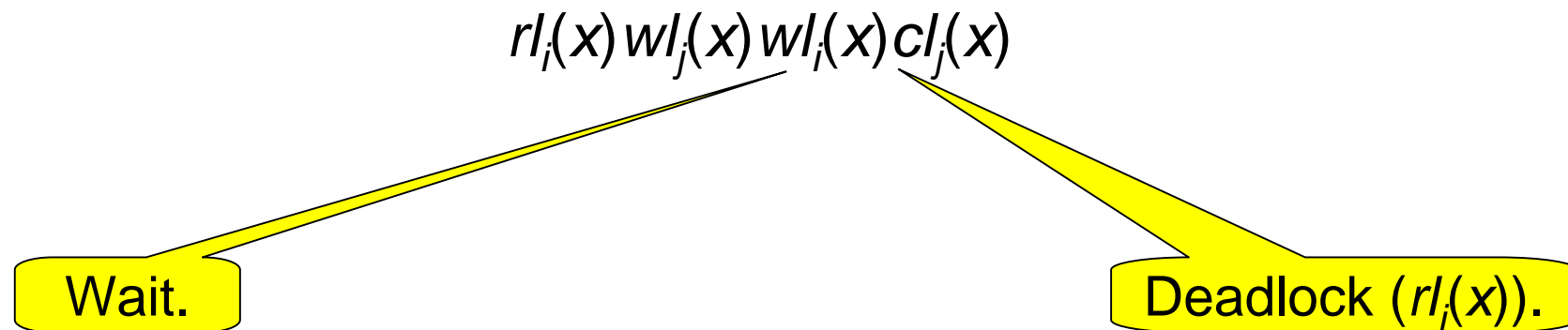
$C_i$

Wait until all readers have finished.

Change all write locks to certify locks. (There can be no write locks or certify locks held by other transactions.) After changing all locks and making the new versions persistent: Release locks.

## 2-version 2PL (5)

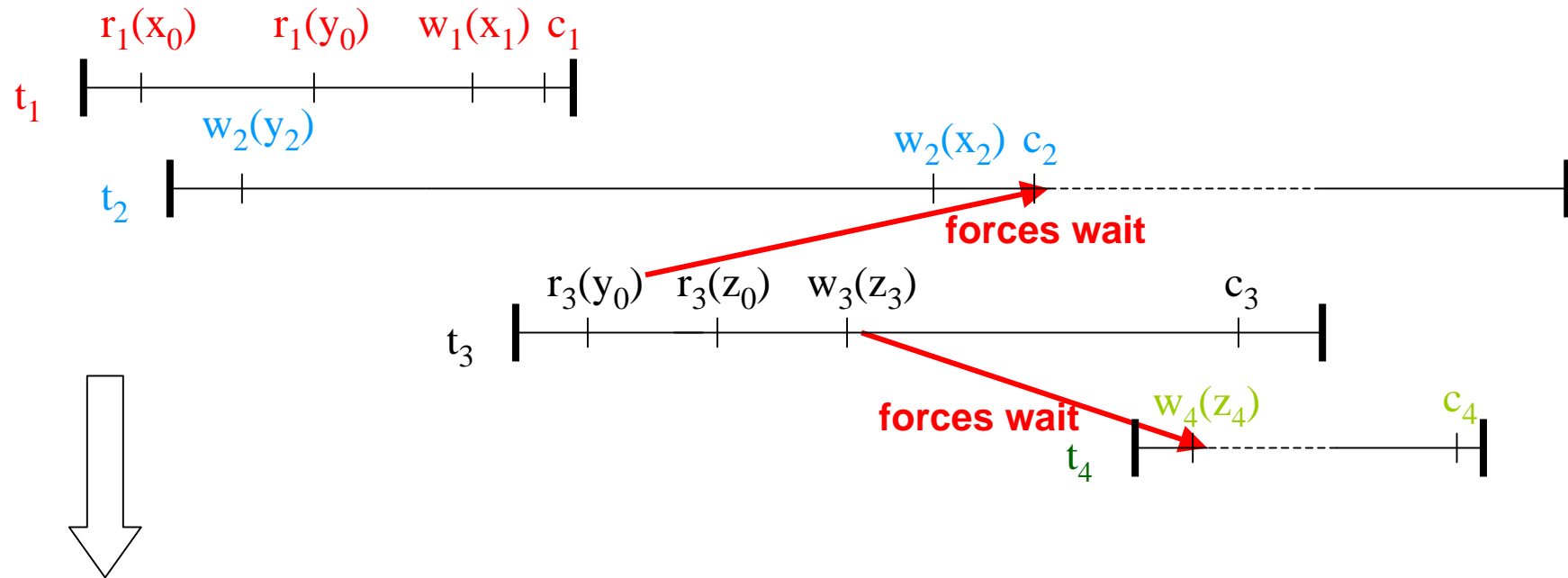
- Lock escalation  $rl \rightarrow wl$  is possible and may be a major source for deadlocks.



# 2V2PL Example

## Example 7.25:

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



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

# Correctness of 2V2PL (1)

40

## Theorem 7.26

$Gen(2V2PL) \subset MCSR$

# Correctness of 2V2PL (2)

41

$f_i$  : certify

2V2PL1:  $\forall t_i: r_i(x_j), w_i(x_j) <_m f_i \wedge f_i <_m C_i$

2V2PL2:  $\forall r_k(x_j) \in CP(m), j \neq k: C_j <_m r_k(x_j)$

Es werden nur freigegebene Versionen gelesen.

2V2PL3:  $\forall r_k(x_j), w_i(x_j) \in CP(m), j \neq k: f_i <_m r_k(x_j) \vee r_k(x_j) <_m f_i$

Die Zertifizierungen aller  $x$  schreibenden Transaktionen müssen mit den  $x$  lesenden Operationen geordnet sein.

2V2PL4:  $\forall r_k(x_j), w_i(x_j) \in CP(m), i \neq j \neq k: f_i <_m r_k(x_j) \succ f_i <_m f_j$

Zusammen mit 2V2PL2: jedes Lesen  $r_k(x_j)$  liest die letzte zertifizierte Version von  $x$ .

2V2PL5:  $\forall r_k(x_j), w_i(x_j) \in CP(m), i \neq j, i \neq k: r_k(x_j) <_m f_i \succ f_k <_m f_i$

Eine  $x$  schreibende  $t_j$  kann nicht zertifiziert werden, solange nicht alle Transaktionen, die  $x$  vorher lasen, zertifiziert wurden.

2V2PL6:  $\forall w_i(x_j), w_j(x_j) \in CP(m): f_i <_m f_j \vee f_j <_m f_i$

Zertifizierungen von Transaktionen, die beide ein  $x$  schreiben, sind wechselseitig atomar.

# Correctness of 2V2PL (2)

42

## Beweisskizze:

Definiere eine Versionsordnung  $\ll$  mit  $x_i \ll x_j \Leftrightarrow f_i <_m f_j$ .

2VPL6  $\succ \ll$  ist eine Versionsordnung.

Falls wir beweisen:  $t_i \rightarrow t_j \in MVSG(m, \ll) \succ f_i <_m f_j$ :

Da  $f_i <_m f_j$  eine Teilordnung von  $m$  und per Definition azyklisch ist, gilt:  $MVSG(m, \ll)$  ist azyklisch.

$\succ$  Behauptung.

# Correctness of 2V2PL (3)

Conflict graph  $G(m)$ :

Betrachte  $t_i \rightarrow t_j \in G(m)$

$\succ \exists x \ t_j \triangleright_m(x) t_i$

$\succ [2V2PL2] \ c_i <_m r_j(x_i)$

mit  $[2V2PL1] \ r_j(x_i) <_m f_j$

$\succ [2V2PL1, \text{Transitivität}] \ f_i <_m f_j$

# Correctness of 2V2PL (4)

MV (version edges):

Betrachte eine Versionsordnungskante, die durch  $w_i(x_i)$  und  $r_k(x_j)$  ( $i \neq j \neq k$ ) induziert wurde. Zwei Fälle:

- $x_i \ll x_j$   $\succ$  Kante  $t_i \rightarrow t_j$   $\succ$  [def  $\ll$ ]  $f_i \prec_m f_j$
- $x_j \ll x_i$   $\succ$  Kante  $t_k \rightarrow t_i$

$$x_j \ll x_i \succ f_j \prec_m f_i$$

$$2V2PL3 \succ f_i \prec_m r_k(x_j) \vee r_k(x_j) \prec_m f_i$$

Erstes Disjunkt: (2V2PL4)  $\succ f_i \prec_m f_j$ : Widerspruch!

$$\text{Also } r_k(x_j) \prec_m f_i$$

$$2V2PL5 \succ f_k \prec_m f_i$$

# Mehrversionen-Zeitstempelordnung (1)

45

Annahme: Zahl der aufbewahrten Versionen ist unbegrenzt.

- Ein **Mehrversionen-TO-Scheduler** (MVTO) bearbeitet Operationen in Ankunftsreihenfolge.
- Er übersetzt Operationen auf Datenelemente in solche auf Versionen:

$r_i(x)$  wird in  $r_i(x_k)$  übersetzt, wobei  $x_k$  die Version mit dem größten Zeitstempel kleiner  $ts(t_j)$  ist.  $r_i(x_k)$  wird dann zum DV gesendet.

Für  $w_i(x)$  werden zwei Fälle unterschieden:

1. Falls bereits ein  $r_j(x_k)$  mit  $ts(t_k) < ts(t_j) < ts(t_j)$  ausgeführt wurde, so wird  $w_i(x)$  zurückgewiesen.
  2. Sonst wird  $w_i(x)$  in  $w_i(x_j)$  übersetzt und zum DV gesendet.
- Wegen Def. 7.3 (4)  $f(r_j(x)) = r_j(x_j)$ ,  $i \neq j$ ,  $c_j \in m \succ c_i <_m c_j$  wird  $c_j$  solange verzögert, bis  $c_i$  für alle  $t_j$  mit  $t_j \triangleright t_i$  ausgeführt wurde.

# Mehrversionen-Zeitstempelordnung (2)

46

Leser werden nie zurückgewiesen.  
Schreiber werden nur zurückgewiesen,  
falls ihnen ein Leser folgt, der eine  
frühere Version gesehen hat.

$r_i(x)$  wird in  $r_i(x_k)$  übersetzt, wobei  $x_k$   
die Version mit dem größten  
Zeitstempel kleiner  $ts(t_i)$  ist.  $r_i(x_k)$  wird  
dann zum DV gesendet.

Für  $w_i(x)$  werden zwei Fälle  
unterschieden:

1. Falls bereits ein  $r_j(x_k)$  mit  $ts(t_k) < ts(t_i) < ts(t_j)$  ausgeführt wurde, so wird  $w_i(x)$  zurückgewiesen.
2. Sonst wird  $w_i(x)$  in  $w_i(x_i)$  übersetzt und zum DV gesendet.

1VTO weist Operationen, die  
zu spät kommen, zurück.

Dabei ist eine Operation  $p_i(x)$   
zu spät, falls gilt:

$$\exists q_j(x) \in h \quad p_i(x) \not\ll q_j(x) \in h \wedge q_j(x) <_h p_i(x) \wedge ts(t_j) > ts(t_i)$$

Da  $q_j(x)$  in einem solchen Fall  
schon ausgeführt ist, muss  
 $p_i(x)$  zurückgewiesen werden.

Leser und Schreiber, die zu  
spät kommen, werden  
zurückgewiesen.

# Mehrversionen-Zeitstempelordnung (3)

47

## Unterschiedliches Verhalten MVTO und 1VTO:

- Nur zulässig unter MVTO:

$$w_0(x_0) < w_2(x_2) < w_1(x_1)$$

$$w_1(x_1) < r_2(x_1) < w_0(x_0)$$

- Nicht zulässig unter MVTO:

$$w_0(x_0) < r_2(x_0) < w_1(x_1)$$



$w_1(x_1)$  **invalidiert**  $r_2(x_0)$

# MV-Zeitstempelordnung-Protokolle (1)

## MVTO Versionsauswahl:

Zur Auswahl der zu lesenden Versionen und um Invalidierungen zu vermeiden, verwaltet der Scheduler Zeitstempelintervalle. Für jede Version  $x_i$  wird ein Zeitstempelintervall  $interval(x_i) = [wts, rts]$  geführt, mit

- $wts$  ist der Zeitstempel von  $x_i$  und
- $rts$  ist der größte Zeitstempel eines Lesens von  $x_i$ . Falls kein solches Lesen existiert, so gilt  $wts = rts$ .

Sei  $intervals(x) = \{interval(x_i) \mid x_i \text{ ist eine Version von } x\}$ .

Zur Bearbeitung von  $r_i(x)$  untersucht der Scheduler  $intervals(x)$ , um diejenige Version  $x_j$  zu finden, deren Intervall  $[wts, rts]$  das größte  $wts$  kleiner  $ts(t_i)$  hat. Falls  $rts < ts(t_i)$ , so wird  $rts$  auf  $ts(t_i)$  gesetzt.

# MV-Zeitstempelordnung-Protokolle (2)

## MVTO Versionsauswahl:

Zur Auswahl der zu lesenden Versionen und um Invalidierungen zu vermeiden, verwaltet der Scheduler Zeitstempelintervalle. Für jede Version  $x_i$  wird ein Zeitstempelintervall  $interval(x_i) = [wts, rts]$  geführt, mit

- $wts$  ist der Zeitstempel von  $x_i$  und
- $rts$  ist der größte Zeitstempel eines Lesens von  $x_i$ . Falls kein solches Lesen existiert, so gilt  $wts = rts$ .

Sei  $intervals(x) = \{interval(x_i) \mid x_i \text{ ist eine Version von } x\}$ .

Zur Bearbeitung von  $w_i(x)$  untersucht der Scheduler  $intervals(x)$ , um diejenige Version  $x_j$  zu finden, deren Intervall  $[wts, rts]$  das größte  $wts$  kleiner als  $ts(t_i)$  hat. Falls  $rts > ts(t_i)$ , wird  $w_i(x)$  zurückgewiesen, sonst wird  $w_i(x_j)$  an den DV gesendet und ein neues Intervall  $interval(x_j) = [ts(t_i), ts(t_i)]$  erzeugt.

# MVTO-Korrektheit (1)

50

**Eigenschaften einer MVTO-Historie** über  $\{t_0, \dots, t_n\}$ .

**MVTO1** (Eigenschaft von Zeitstempeln):  $ts(t_i) = ts(t_j) \Leftrightarrow i = j$

**MVTO2** (Übersetzen von Lesen):

$$\forall r_k(x_j) \in CP(m) \quad w_j(x_j) <_{CP(m)} r_k(x_j) \wedge ts(t_j) < ts(t_k)$$

**MVTO3** (Übersetzen von Schreiben):

$$\forall r_k(x_j), w_i(x_i) \in CP(m), \quad i \neq j$$

(a)  $ts(t_i) < ts(t_j)$  XOR

(b)  $ts(t_k) < ts(t_j)$  XOR

(c)  $i = k \wedge r_k(x_j) <_{CP(m)} w_i(x_i)$

**MVTO4** (Vollständigkeit):

$$r_j(x_i) \in CP(m), \quad i \neq j \quad c_j \in CP(m) \succ c_i <_m c_j$$

# MVTO-Korrektheit (2)

## Satz 7.27

$Gen(MVTO) \subset MCSR$

### Beweisskizze:

Wir definieren Versionsordnung  $\ll$  wie folgt:  $x_i \ll x_j \Leftrightarrow ts(t_i) < ts(t_j)$ .

**Zeige:**  $MVSG(CP(m), \ll)$  ist azyklisch. **Zeige** dazu, dass

$$\forall t_i \rightarrow t_j \in MVSG(CP(m), \ll): ts(t_i) < ts(t_j).$$

$G(m)$ :

$t_i \rightarrow t_j \in G(CP(m)) \succ t_j \triangleright_{CP(m)}(x) t_i$  für ein  $x$ .

$MVTO2 \succ ts(t_i) < ts(t_j)$ .

# MVTO-Korrektheit (3)

## MV:

Seien  $r_k(x_j), w_i(x_j) \in CP(m)$ ,  $i \neq j \neq k$ . Führt zu einer der folgenden Versionsordnungskanten

1.  $x_i \ll x_j \succ t_i \rightarrow t_j \in MVSG(CP(m), \ll)$ :  
Nach Def. von  $\ll$ :  $ts(t_i) < ts(t_j)$ .
2.  $x_j \ll x_i \succ t_k \rightarrow t_i \in MVSG(CP(m), \ll)$ :  
Nach Def. von  $\ll$ :  $ts(t_j) < ts(t_i)$ .

MVTO3  $\succ$

(a)  $ts(t_i) < ts(t_j)$  XOR

(b)  $ts(t_k) < ts(t_i)$

(a) Widerspruch zu Vorauss.:  $\succ ts(t_k) < ts(t_j)$ .

Da alle Kanten in  $MVSG(CP(m), \ll)$  in Zeitstempelordnung sind, ist  $MVSG(CP(m), \ll)$  azyklisch.

# MVTO-Speicherbereinigung

53

Versionen und deren Intervalle können nur in Zeitstempelordnung gelöscht werden. Betrachte:

$$m_{11} = w_0(x_0) c_0 r_2(x_0) w_2(x_2) c_2 r_4(x_2) w_4(x_4)$$

mit  $ts(t_j)=i$ . Annahme:  $x_2$  wird gelöscht, aber nicht  $x_0$ .  $r_3(x)$  erreiche den Scheduler. Der übersetzt es fälschlicherweise in  $r_3(x_0)$  statt  $r_3(x_2)$ .

Korrekte Reihenfolge vermeidet falsches Übersetzen, nicht aber unnötiges Invalidieren.

Annahme:  $x_0$  wird gelöscht.  $r_1(x)$  erreiche den Scheduler. Dieser findet keine Version mit  $wts < ts(t_1)$ . Diese Bedingung zeigt an, dass die zu lesende Version bereits gelöscht wurde.  $r_1(x)$  wird zurückgewiesen.

# Read-only Multiversion Protocol

54

## Read-only Multiversion Protocol (ROMV):

- For long read transactions that execute concurrently with writers.

- Each update transaction 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  $t_i$ )

$Gen(ROMV) \subseteq MVSR$ :

Sketch of proof: all edges  $t_i \rightarrow t_j \Leftrightarrow c_i < c_j$

Indeed:  $Gen(ROMV) \subseteq MCSR$

# Read-only Multiversion Protocol

