Multiversion Concurrency Control

Sebastian Faller

30. Juni 2009

Introduction

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

What's the use of a good quotation if you can't change it
-Anonymous

Introduction

- theoretically attractive and practically relevant
- more than one copy of the same data item
  - keep old version until write operation commits successfully
  - distinct transactions could be given distinct versions of the same data item to read or to overwrite.
  - keep track of changes $\Rightarrow$ history
- assumption transparent versioning
  - versioning is transparent to outside world
  - users or applications are not aware of the fact that data items can appear in multiple versions.

Multiversion Schedules

- Multiversion Schedule is
  - essentially a schedule s in the sense used so far

Excursion: Schedule

- s consists of the union of the operations from the given transactions plus a termination operation
- for each transaction, there is either a commit or an abort in s
- all transaction orders are contained in the partial order given by s
- the commit or abort operation always appears as the last step of a transaction
- every pair of operations $p, q \in op(s)$ from distinct transaction that access the same data item and have at least one write operation among them is ordered in s such a way that either $p <_s q$ or $q <_s p$
- a schedule is a prefix of a history

  + a version function
- it contains
  - versioned read and write operations
  - whose ordering respects the individual transaction ordering

Multiversion Schedules

Version Function

Let s be a history with initialisation 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 which is the identity on write steps.

- write steps are translated into a version creation step
  - $w(x)$ (always) creates a new version of x
  - $h(w_i(x)) = w_i(x)$, and $w_i(x)$ writes x_j
- read steps are translated into a version read step
  - $r(x)$ reads an (existing) version of x
  - $h(r_i(x)) = w_j(x)$ for some $w_j(x) <_s r_i(x)$, and $r_i(x)$ reads x_j

Multiversion Schedules

Multiversion Schedule

Let $T = t_1, ..., t_n$ be a finite set of transactions.

1. A *multiversion history* (or complete multiversion schedule) for T is a pair $m = (op(m), <_m)$, where $<_m$ is an order on $op(m)$ and
   1. $op(m) = h(\cup_{i=1}^{n} op(t_i))$ for some version function h (here we assume that h has been canonical extended from single operations to sets of operations)
   2. for all $t \in T$ and all operations $p, q \in op(t)$ the following holds:
      $$p <_t q \Rightarrow h(p) <_m h(q)$$
   3. if $h(r_j(x)) = r_j(x_i)$, $i \neq j$, and c_j is in m, then c_i is in m and $c_i < c_j$.
2. A *multiversion schedule* is a prefix of a multiversion history

Multiverison Schedules

Monoversion Schedule

A multiversion schedule is called *monoversion schedule* if version function maps each read step to the last preceding write step on the same data item.

- $m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_2)w_1(z_1)c_1c_2$
- $s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_1(z)c_1c_2$

Reads-From Relation

Let m be a multiversion schedule, $t_i, t_j \in trans(m)$.
The *reads-from relation* of m is defined by

$$RF(m) := (t_i, x, t_j) | r_j(x_i) \in op(m)$$

View Equivalence

Let m be two multiversion schedules such that $trans(m') = trans(m)$.
m and m' are *view eqivalent*, abbreviated $m \approx_v m'$, if $RF(m) = RF(m')$

Multiversion View Serializability

Let m be a multiversion history. Then m is called *multiversion view serializable* if there exist a serial monoversion history m' for the same set of transactions such that $m \approx_v m'$
Let MVSR denote the class of all multiversion view-serializable histories

Example View Equivalence:

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

Example Multiversion View Serializability:

- Let $m = w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_2(x_1) r_3(x_0) w_3(x_3) c_3 w_2(y_2) c_2$
- and $m' = w_0(x_0) w_0(x_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) w_2(y_2) c_2$
- Then we have $m \approx_v m'$. Moreover, m' is view equivalent to history $s = w_0(x) w_0(y) c_0 r_3(x) w_3(x) c_3 w_1(x) w_2(y) c_2$

Example View Equivalence:

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

Example Multiversion View Serializability:

- Let $m = w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_2(x_1) r_3(x_0) w_3(x_3) c_3 w_2(y_2) c_2$
- and $m' = w_0(x_0) w_0(x_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) w_2(y_2) c_2$
- Then we have $m \approx_v m'$. Moreover, m' is view equivalent to history $s = w_0(x) w_0(y) c_0 r_3(x) w_3(x) c_3 w_1(x) w_2(y) c_2$

Example View Equivalence

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

Example Multiversion View Serializability:

- Let $m = w_0(x_0) w_0(y_0) c_0 w_1(x_1) c_1 r_2(x_1) r_3(x_0) w_3(x_3) c_3 w_2(y_2) c_2$
- and $m' = w_0(x_0) w_0(x_0) c_0 r_3(x_0) w_3(x_3) c_3 w_1(x_1) w_2(y_2) c_2$
- Then we have $m \approx_v m'$. Moreover, m' is view equivalent to history $s = w_0(x) w_0(y) c_0 r_3(x) w_3(x) c_3 w_1(x) w_2(y) c_2$

Example View Equivalence:

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

Example Multiversion View Serializability:

- Let $m = w_0(x_0)w_0(y_0)c_0 w_1(x_1)c_1 r_2(x_1)r_3(x_0)w_3(x_3)c_3 w_2(y_2)c_2$
- and $m' = w_0(x_0)w_0(x_0)c_0 r_3(x_0)w_3(x_3)c_3 w_1(x_1)w_2(y_2)c_2$
- Then we have $m \approx_v m'$. Moreover, m' is view equivalent to history $s = w_0(x)w_0(y)c_0 r_3(x)w_3(x)c_3 w_1(x)w_2(y)c_2$

Theorem

$VSR \subset MVSR$

Example:

- $m = w_0(x_0) w_0(y_0) c_0 r_1(x_0) w_2(x_2) w_2(y_2) c_2 r_1(y_0) c_1$

Theorem

The Problem of deciding weather a given multiversion history is in MVSR is NP complete

Multiversion Serializability
Testing Membership in MVSR

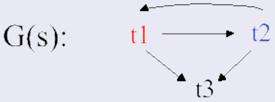
Theorem

For any two multiversion schedules m and m', $m \approx_v m'$ implies $G(m) = G(m')$

Excursion: Conflict Graph

Let s be a schedule. The conflict graph $G(s) = (V, E)$ is a directed graph with vertices $V := commit(s)$ and
edges $E := \{(t, t')|t \neq t'$ and there are steps
$p \in t, q \in t' with (p, q) \in conf(s)\}$.

$$s = r_1(y)\ r_3(w)\ r_2(y)\ w_1(y)\ w_1(x)\ w_2(x)\ w_2(z)\ w_3(x)\ c_1\ c_3\ c_2$$



Version Order

If x is a data item, a *version order* for x is any nonreflexive and total ordering of all vesrsions of x that are written by operations by 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 (MVSG)

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

$$\textit{if } x_i \ll x_j, \textit{ then } (t_i, t_j \in E), \textit{ otherwise } (t_k, t_i) \in E.$$

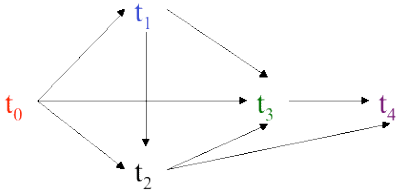
Multiversion Serializability
Testing Membership in MVSR

Example:

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

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

MVSG(m, $\ll$):



Notice: Testing whether appropriate $\ll$ exists for given m is not necessarily polynomial → NP-completeness result remains

Multiversion Conflict

A *multiversion conflict* in a multiversion schedule m is a pair of steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$

Multiversion Reduceability

A (totally ordered) multiversion history m is *multiversion reducible* if it can be transformed into a serial monoversion history by a infinite sequence of transformation steps, each of which exchanges the order of two adjacent steps, (i.e. steps p, q with $p < q$ such that $o < p$ or $q < o$ for all other steps o) but without reversing the ordering of a multiversion conflict (i.e. *rw* pair)

Multiversion Conflict Serializability

A multiversion 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. Let MCSR denote the class of all multiversion conflict-serializable histories.

Multiversion Conflict Graph

Let m be a multiversion schedule. The multiversion conflict graph of m is a graph that has the transactions of m as its nodes and an edge from t_i to t_k if there are steps $r_i(x_i)$ and $w_k(x_k)$ for the same data item x in m such that $r_i(x_i) <_m w_k(x_k)$.

Theorem

A multiversion history is multiversion reducible iff it is multiversion conflict serializable.

Theorem

A multiversion history is MCSR iff its multiversin conflict graph is acyclic.

Theorem

$MCSR \subset MVSR$

Multiversion Concurrency Control Protocols

The MTVO *(multiversion timestamp ordering)* Protocol

- scheduler processes operations in FIFO fashion
- transforms data operations into operations on versions of data items
- are then processed that the result appears as if produced by a serial monoversion schedule with transactions in the order of timestamps that are assigned at the beginning of a transaction.

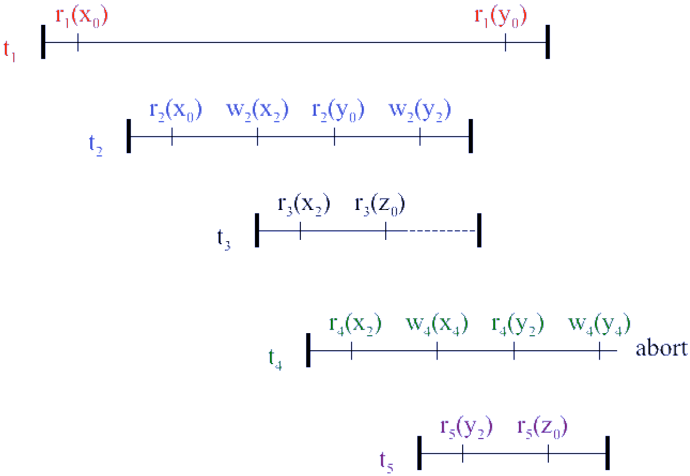
Protocol Rules:

1. A step $r_i(x)$ is transformed into a step $r_i(x_k)$. where x_k is the version of x that carries the largest timestamp $\leq ts(t_i)$ and was written by $t_k, k \neq i$.
2. A step $w_i(x)$ is processed as follows:
   1. If a step of the form $r_j(x_k)$ such that $ts(t_k) < ts(t_i) < ts(t_j)$ has already been scheduled, then $w_i(x)$ is rejected and t_i is aborted,
   2. otherwise, $w_i(x)$ is transformed into $w_i(x_i)$ and executed.
3. A commit c_i is delayed until the commit c_j of all transactions t_j that have written new versions of data items read by t_i have been processed. (This part of the protocol is optional and included in order to ensure correct transaction recovery)

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

- $x_i \ll x_j \iff ts(t_i) < ts(t_j)$

Example:


r1(x0)
r1(y0)
t1
r2(x0)
w2(x2)
r2(y0)
w2(y2)
t2
r3(x2)
r3(z0)
t3
r4(x2)
w4(x4)
r4(y2)
w4(y4)
abort
t4
r5(y2)
r5(z0)
t5

Protocol Rules:

1. If step is not final within the transaction:
   1. an $r(x)$ is executed right away, by assigning to it the current version of the requested data item, i.e. the most recently committed version (but not any other, previously committed one), or by assigning to it an uncommitted version of x;
   2. a $w(x)$ is executed only when the transaction that gas written x last is finished, so that there are no other uncommitted versions of x
2. If the step is final within the transaction t_i, it is delayed until the following types of transactions are committed:
   1. all those transactions t_j that have read the current version of a data item written by t_i,
   2. all those t_j from which t_i has read some version.

Example on how M2VPL works:

- Input: $s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_2(x)c_2w_1(y)c_1$
  1. $r_1(x)$ assigned to x_0 and executed: $r_1(x_0)$
  2. $w_1(x)$ executed since no other transaction is still active: $w_1(x_1)$
  3. let $r_2(x)$ be assigned to x_1 and executed: $r_2(x_1)$
  4. $w_2(y)$ is executed: $w_2(y_2)$
  5. let $r_1(y)$ be assigned to y_0 and execute: $r_1(y_0)$
  6. if $w_2(x)$ were not the last step of t_2, it would be delayed since t_1 is still active and has written x_1. However, as it is the final step of t_2, the final step rules need to be applied. It turns out that t_2 nonetheless has has to wait for the following reason:
     a) t_1 has read the current version of the data item y (y_0), and t_2 overwrites this version,
     b) t_2 has read x_1 from t_1
  7. $w_1(y)$, the final step of t_1, is executed since
     a) t_2 has not read a current version of a data item written by t_1 (current versions are x_0, y_0),
     b) t_1 has not read a version written by t_2 $w_1(y_1)$
  8. finally, $w_2(x)$ can be executed: $w_2(x_2)$
- Output:

  $m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_0)w_1(y_1)c_1w_2(x_2)c_2$

Protocol Rules:

1. $w_i(x)$
   - add edge from (t_j, t_i) to the graph
   - for each t_j for which an $r_j(x)$ has already been scheduled
   - if G becomes cyclic
     - $w_i(x)$ is not executed
   - else
     - $w_i(x)$ is executed the newly written version is kept
2. $r_i(x)$
   - is scheduled and receives version written by $w_j(x)$
   - add edge from (t_j, t_i) to G
   - since t_j is not late, this does not close a cycle
   - in additon
     - add edge from either (t_k, t_j) or (t_i, t_k) for each step $w_k(x)$ already scheduled
     - t_k is neither late or early

- those t_j on a path that originates from t_i, since these are supposed to follow t_i in an equivalent serial schedule; let us call them *late*;
- those t_j for which a path exists from t_j to another candidate t_k that writes x and from t_k to t_i, since in an equivalent serial schedule, t_k writes x after t_j; let us call them early.

Multiversion Concurrency Control Protocols

The ROMV *(read-only multiversion protocol)*Protocol

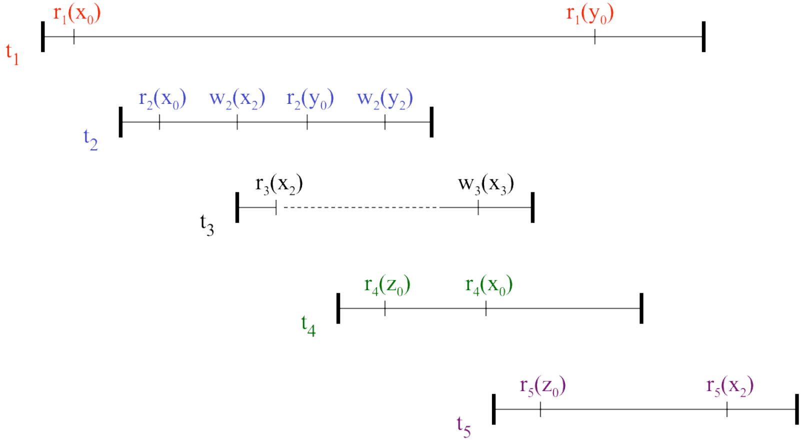
Protocol Rules:

- 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 (i.e $Gen(ROMV) \subseteq MVSR$)

- $x_i \ll x_j \Leftrightarrow c_i < c_j$

Exampel:



t_1
$r_1(x_0)$
$r_1(y_0)$
t_2
$r_2(x_0)$
$w_2(x_2)$
$r_2(y_0)$
$w_2(y_2)$
t_3
$r_3(x_2)$
$w_3(x_3)$
t_4
$r_4(z_0)$
$r_4(x_0)$
t_5
$r_5(z_0)$
$r_5(x_2)$

Lessons Learned

- A **Multiversion Schedule** is essentially a *schedule* in the sense used so far together with a *version function*.
- **Multiversion Serializability**
  - is more powerful
  - performance improvements achievable
- **Multiversion Protocols**
  - 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

Literature I

[AB87] M.L. Ahuja and J.C. Brown. Concurrency control by preordering entities in databases with multiversional entities. In *In Proc. 3rd IEEE International Conference on Data Engineering*, pages 312 – 321, 1987. srihdsthsdathdsfhtudfstdsth.

[AS93] R. Agrawal and S. Sengupta. Modular sychronization in distributed, multiverion databases: Version control and concurrency control. In *IEEE Transactions on Knwoledge and Data Engineering*, volume 5, pages 126 – 137, 1993.

[BC92a] P.M. Bober and M.J. Carey. Multiversion query locking. In *In Proc. 18th International Conference on Very Large Databases*, pages 497 – 510, 1992.

Literature II

[BC92b] P.M. Bober and M.J. Carey. On mixing queries and transactions via multiversion locking. In *In Proc. 8th IEEE International Conference on Data Engineering*, pages 535 – 545, 1992.

[Ber98] P.A. Bernstein. Repositories and object oriented databases. *SIGMOD Record*, 27(1), 1998.

[BG83] P.A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithems. In *ACM Transaction on Database Systems*, volume 8, pages 465 – 483, 1983.

[BG87] V. Hadzilacos Bernstein, P.A. and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison - Wesley, 1987.

Literature III

[BR80] H. Heller Bayer, R. and A. Reiser. Parallelism and recovery in database systems. *Transactions on Database Systems*, 5:139 – 156, 1980.

[BS83] G.N. Buckley and A. Silberschatz. Obtaining progessive protocols for a simple multiversion datatbase model. In *In Proc. 9th International Conference on Very Large Data Bases*, pages 74 – 80, 1983.

[CG85] A. Chan and R. Gray. Implementing distributed read-only transactions. In *IEEE Transaction on Software Engineering*, volume 11, pages 205 – 212, 1985.

Literature IV

[CJ90] W. Cellary and G. Jamier. Consistency of versions in object-oriented databases. In *In Proc. 16th International Conference on Very Large Data Bases*, pages 432 – 441, 1990.

[Cla92] B. Claybrook. *OLTP - Online Transaction Processing Systems*. New York: J. Wiley Sons, 1992.

[CM88] E. Gelenbe Cellary, W. and T. Morzy. *Concurrency Cointrol in Distributed Data Base Systems*. Amsterdam: North-Holland, 1988.

[CR82] S. Fox W.K. Lin A. Nori Chan, A. and D.R. Ries. The implementation of an integrated conccurrency control and recovery scheme. In *In Proc. ACM SGMOD International Conference on Management of Data*, pages 184 – 191, 1982.

Literature V

[DuB82] D. DuBourdieu. Impelementation of distributed transactions. In *In Proc. 6th Berkley Workshop on Distributed Data Management and Computer Networks*, pages 81 – 93, 1982.

[Had88] T. Hadzilacos. Serilization graph algorithems for muliversion concurrency control. In *In Proc. 7th ACM SIGACT-SIGMOD Symposium on Principals of Database Systems*, pages 135 – 141, 1988.

[HS00] W. Mahnke N. Ritter Haerder, T. and H.-P. Steiert. Generating versioning facilities for a design-data repository supporting cooperative applications. *International Journal of Cooperative Information Systems*, 9:117 – 146, 2000.

Literature VI

[IM90] T. Kameda Ibaraki, T. and T. Minoura. Multiversion cautious schedulers for database concurrency control. In *IEEE Transaction on Software Engineering*, volume 16, pages 302 – 315, 1990.

[Kat90] R.H. Katz. Toward a unified framework for version modelling in engineering databases. *ACM Computing Surveys*, 22:375 – 408, 1990.

[Lau83] G. Lausen. Formal aspects of optimistic concurrency control in a multipel version database system. *Information Systems*, 8:291 – 301, 1983.

[ML92] H. Pirahesh Mohan, C. and R. Loire. Efficent and flexible methods for transient versioning of record to avoid locking by read-only transactions, 1992.

Literature VII

[Mor93] T. Morzy. The correctness of concurrency control of multiversion database systems with limited number of versions. In *In Proc 9th IEEE International Conference on Data Engineering*, pages 565 – 604, 1993.

[Pap86] C.H. Papadimitriou. *The Theory of Database Concurrency Control*. MD: Computer Science Press, 1986.

[PK84] C.H. Papadimitriou and P. Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems*, 9:89 – 99, 1984.

Literature VIII

[Raz93] Y. Raz. Commitment ordering based distributed concurrency control for bridging single and multi version resources. In *In Proc. 3rd International Workshop on Research Issues in Data Engineering: Interoperability in Multi Database Systems*, pages 189 – 199, Vienna, Austria, 1993.

[Ree78] D.P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, MIT, Camebridge, MA, 1978.

[Ree83] D.P. Reed. Implementing atomic actions on dcentralized data. *ACM Transaction on Computer Systems*, 1:3 – 23, 1983.

Literature IX

[Sch77] B.-M Schueler. Update reconsidered. In *In Proc. IFIP Working Conference on Architecture and Models in DBMS*, pages 149–164. Amsterdam: North-Holland, 1977.

[SR81] Richard E. Stearns and Daniel J. Rosenkrantz. Distributed database concurrency controls using before-values. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 74–83, New York, NY, USA, 1981. ACM.

[Vid91] K. Vidyasankar. Unified theory of database serializability. *Fundamenta Informaticae*, 14:147 – 183, 1991.