REAL-TIME MULTIVERSION REPEATABLE READ ISOLATION LEVEL

Jan Lindström
solidDB, IBM Helsinki Lab, IBM Software Group,
P.O. Box 265, Fi-00101 Helsinki, Finland
jan.lindstrom@fi.ibm.com

ABSTRACT
Concurrency control is the activity of synchronizing database operations by concurrently executing transactions on a shared database. We examine the problem of concurrency control when the database supports multiple versions of the data. Multiversion concurrency control is used in order to improve the level of achievable concurrency. The goal is to produce an execution that has the same effect as a serial one. We use the multiversion concurrency control theory to analyze histories produced by multiversion concurrency control methods. We show that using traditional repeatable read isolation level is inadequate and provide a new isolation level definition for multiversion repeatable read. We show that the new isolation level captures the essence of repeatable read isolation level in multiversioning systems.

KEY WORDS
Concurrency Control, Real-Time Databases, Isolation levels, Multiversioning

1 Introduction

A *database system* (DBS) is a process that executes read and write operations on data items of a database. A *transaction* is a program that issues reads and writes to a DBS. If transactions execute concurrently, the interleaved execution of their reads and writes by the DBS can produce undesirable results. Thus, the basic problem in concurrency control is to maintain the consistency of a database updated by interleaved transactions [10]. Specifically, the goal of concurrency control is to produce an execution that has the same effect as a serial one. Although we hope it would have better performance time characteristics as a serial one. Such executions are called *serializable* [5]. A DBS secures a serializable execution by controlling the order in which reads and writes are executed [14]. The methods for securing serializability are diverse. Some methods monitor the execution requests in order to secure a meaningful subset of serializability [14] or insert lock-unlock steps in the transactions [7].

Traditional databases, hereafter referred to as databases, deal with persistent data. Transactions access this data while maintaining its consistency. The goal of transaction and query processing in databases is to get a good throughput or response time. In contrast, *real-time systems* can also deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal character of the data and the response-time requirements forced by the environment, tasks in real-time systems have time constraints, e.g. periods or deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks [20].

One of the most important points to remember here is that real-time does not just mean fast [20]. Additionally real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time means the need to manage *explicit* time constraints in a predictable fashion, that is, to use time-cognizant methods to deal with deadlines or periodicity constraints associated with tasks.

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement defined by serializability [5], most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [7]. In recent years, various real-time concurrency control methods have been proposed for the single-site RTDBS by modifying 2PL (e.g. [11, 13, 19]). However, 2PL has some inherent problems such as the possibility of deadlocks as well as long and unpredictable blocking times. These problems appear to be serious in real-time transaction processing since real-time transactions need to meet their timing constraints, in addition to consistency requirements [15].

In a *multiversion* DBS each write on a data item produces a new version of the data item. For each read of the data item the DBS selects one of the versions of data item to be read. Because writes do not overwrite each other and reads can read any version the DBS has more alternatives controlling the order of reads and writes. Multiversion concurrency control (MVCC) is described in some detail in sections 4.3 and 5.5 of in [4]. this paper cites a 1978 dissertation by D.P. Reed [17] which describes MVCC and claims it as an original work. Many interesting concurrency control algorithms using multiversioning have been proposed (e.g. [2, 6, 18, 9, 12, 16, 21]).

MVCC is particularly adept at implementing true snapshot isolation [3], something which other methods of concurrency control frequently do either incompletely or with high performance cost. Snapshot isolation has been adopted by several major database management systems, such as SQL Anywhere, InterBase, Firebird, Oracle, PostgreSQL, Solid and Microsoft SQL Server. The main reason

for its adoption is that it allows better performance than serializability, yet still avoids the kind of concurrency anomalies that cannot easily be worked around. Snapshot isolation has also been used [3] in criticism of the ANSI SQL-92 standard's definition of isolation levels, as it exhibits none of the "anomalies" that the SQL standard prohibited, yet snapshot isolation is not serializable (the anomaly-free isolation level defined by ANSI).

Unfortunately, the ANSI SQL-92 standard was written with a lock-based database in mind, and hence is rather vague when applied to MVCC systems [3, 8]. As a concrete example, imagine a bank storing two balances, X and Y, for accounts held by a single person, Phil. The bank will allow X or Y to run a deficit, provided that the total held in both is never negative (i.e., $X + Y \geq 0$ must hold). Suppose both X and Y start at \$100. Now imagine Phil initiates two transactions concurrently, T_1 withdrawing \$200 from X, and T_2 withdrawing \$200 from Y.

If the database guaranteed serializable transactions, the simplest way of coding T_1 is to deduct \$200 from X, and then verify that $X + Y \geq 0$ still holds, aborting if not. T_2 similarly deducts \$200 from Y and then verifies $X + Y \geq 0$. Since the transactions must serialize, either T_1 happens first, leaving $X = -\$100, Y = \$100$, and preventing T_2 from succeeding or T_2 happens first and similarly prevents T_1 from succeeding.

Rest of the paper is organized as follows. In section 4 present a new multiversion repeatable read isolation level to solve problems on traditional repeatable read isolation level. Finally, Section 5 presents the conclusions of this work.

2 Real-Time Data and Transactions

- *Absolute consistency*: Data is only valid between absolute points in time. This is due to the need to keep the database consistent with the environment.
- *Relative consistency*: Different data items that are used to derive new data must be temporally consistent with each other. This requires that a set of data items used to derive a new data item form a *relative consistency set* R.

Data item d is *temporally consistent* if and only if d is absolutely consistent and relatively consistent [15]. Every data item in the real-time database consists of the current state of the object (i.e. current value stored in that data item), and two timestamps. These timestamps represent the time when this data item was last accessed by the committed transaction. These timestamps are used in the concurrency control method to ensure that the transaction reads only from committed transactions and writes after the latest committed write.

In this section, transactions are characterized along three dimensions; the manner in which data is used by transactions, the nature of time constraints, and the significance of executing a transaction by its deadline, or more precisely, the consequence of missing specified time constraints [1].

To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally. For the simplicity of the exposition, it is assumed that each transaction reads and writes a data item at most once. From now on the abbreviations *r, w, a* and *c* are used for the read, write, abort, and commit operations, respectively.

Definition 2.1 *A transaction T_i is partial order with an ordering relation $\prec_i$ where [5]:*

1. $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ is a data item } \} \cup \{a_i, c_i\}$;
2. $a_i \in T_i$ *if and only if* $c_i \notin T_i$;
3. *if t is c_i or a_i, for any other operation $p \in T_i$, $p \prec_i t$; and*
4. *if $r_i[x], w_i[x] \in T_i$, then either $r_i[x] \prec_i w_i[x]$ or $w_i[x] \prec_i r_i[x]$.*

Informally, (1) a transaction is a subset of read, write and abort or commit operations. (2) If the transaction executes an abort operation, then the transaction is not executing a commit operation. (3) if a certain operation t is abort or commit then the ordering relation defines that for all other operations precede operation t in the execution of the transaction. (4) if both read and write operation are executed to the same data item, then the ordering relation defines the order between these operations.

A *real-time transaction* is a transaction with additional real-time attributes. We have added additional attributes for a *real-time transaction*. These attributes are used by the real-time scheduling algorithm and concurrency control method. Additional attributes are the following:

- Timing constraints - e.g. deadline is a timing constraint associated with the transaction.
- Criticalness - It measures how critical it is that a transaction meets its timing constraints. Different transactions have different criticalness. Furthermore, criticalness is a different concept from deadline because a transaction may have a very tight deadline but missing it may not cause great harm to the system.
- Value function - Value function is related to a transaction's criticalness. It measures how valuable it is to complete the transaction at some point in time after the transaction arrives.
- Resource requirements - Indicates the number of I/O operations to be executed, expected CPU usage, etc.
- Expected execution time. Generally very hard to predict but can be based on estimate or experimentally measured value of worst case execution time.

- Data requirements - Read sets and write sets of transactions.
- Periodicity - If a transaction is periodic, what its period is.
- Time of occurrence of events - In witch point in time a transaction issues a read or write request.
- Other semantics - Transaction type (read-only, write-only, etc.).

3 Real-Time Multiversion Concurrency Control Theory

In order to reason about the execution of a collection of transactions on a database using MVCC we need a formal representation of histories rich enough to describe multiple versions of data items. We will develop this in a similar fashion to [22].

A version function h translates each write step into a version creation step and each read step into a version read step.

Definition 3.1 *Multiversion schedule. Let $T = \{t_1, ..., t_n\}$ be a (finite) set of transactions. A multiversion history for T is a pair $L = (\Sigma, \prec_\Sigma)$ where $\prec_\Sigma$ is order on Σ and*

1. $\Sigma = h(\bigcup_{i=1}^{n} \Sigma_i)$ *for some function* h
2. $\forall t \in T$ *and* $\forall p, q \in \Sigma_t$ *the following holds:*
   $p \prec_t q \Rightarrow h(p) \prec_\Sigma h(q)$
3. *if* $h(r_j(x)) = r_j(x_i), i \neq j$ *and* $c_j \in \Sigma$ *then* $c_i \in \Sigma$ *and* $c_i \prec_\Sigma c_j$.

A multiversion schedule is a prefix of a multiversion history.

Condition (1) states that each transaction operation is translated into an appropriate multiversion operation. Condition (2) states that history function preserves all orderings defined by transactions. Condition (3) states that a transaction may not read a version until it has been produced.

Definition 3.2 *Reads-From Relation. Let Σ be a multiversion schedule, $t_, t_j \in \Sigma$ transactions. The reads-from relation of Σ is defined by $RF(\Sigma) = \{(t_i, x, t_j) | r_j(x_i) \in \Sigma\}$.*

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

Definition 3.3 *Multiversion conflict. A multiversion conflict in a multiversion schedule Σ is a pair of steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) \prec_\Sigma w_k(x_k)$.*

Definition 3.3 essentially states that in a multiversion schedule the only relevant conflict is read write operation pairs on the same data item, not necessarily on the same version. It is easy to see that write write pairs on the same data item no longer count as conflicts, as they create different versions. A multiversion schedule can read the same data item more than once and these reads can read different version.

Definition 3.4 *Multiversion Multiple Reads. If a transaction reads the same data item more than once these reads are ordered as condition (2) stated in Definition 3.1 and these reads are numbered. In other words, if $h(r_{j_k}(x)) \in \Sigma_j$ and $h(r_{j_{k+1}(x))} \in Sigma_j$ then $h(r_{j_k}(x)) \prec_\Sigma h(r_{j_{k+1}}(x))$. Note it may hold that two consecutive reads in the transaction do not read the same version of the data item i.e $h(r_{j_k}(x)) = r_{jk}(x_i)$ and $h(r_{j_{k+1}}(x)) = r_{jk+1}(x_l)$ where $i \neq l$.*

In this work we assume that the transaction writes a data item at most once. Similarly, the construction in Definition 3.4 can be used if multiple writes to the same data item are allowed. From above we obtain:

Definition 3.5 *Multiversion Conflict Serializability. A multiversion history Σ 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 Σ.*

It can be shown that membership of a history in class multiversion conflict serializable can be characterized in graph-theoretic terms using the following notion of a multiversion serialization graph.

Definition 3.6 *Multiversion Serialization Graph (MVSG). For a given schedule Σ and a version order $\prec$, the multiversion serialization graph $MVSG(\Sigma, \prec)$ of Σ then is the conflict graph $G(\Sigma) = (V, E)$ with the following edges added to each $r_k(x_j)$ and $w_i(x_i)$ in committed projection of Σ, where k, i and j are pairwise distinct: $if x_i \prec x_j$, then $(t_i, t_j) \in E$, otherwise $(t_k, t_i) \in E$.*

Not surprisingly, we obtain:

Theorem 3.1 *A multiversion history is multiversion conflict serializable (MVSC) iff its multiversion serialization graph is acyclic.*

4 Real-Time Multiversion Repeatable Read Isolation Level

In a traditional monoversion repeatable read isolation level no data records retrieved by a SELECT statement can be changed; however, if the SELECT statement contains any ranged WHERE clauses, phantom reads may occur. In this isolation level the transaction acquires read locks on all retrieved data, but does not acquire range locks.

In a multiversion concurrency control situation radically changes because no locks are acquired for retrieved data. Therefore, a new set of inconsistent retrievals are possible.

Definition 4.1 *Initialization transaction T_0. Lets assume that we have a initialization transaction performing the following steps:*

```
T0
create table total(s integer);
create table vals(i integer not null,
  val integer, primary key(i));
insert into total values (5);
insert into vals values (1,1),(2,1),(3,1),
  (4,1),(5,1);
commit;
```

Now executing first initialization transaction described in Definition 4.1 lets consider following execution history of transactions T_1 and T_2:

Example 4.1 *Read only transaction case.*

```
T1                          T2
begin;
select * from total;
select * from vals;
                            begin;
                            insert into vals values (6,1);
                            update total set s = s + 1;
                            commit;
select * from total;
select * from vals;
commit;
```

Naturally, transaction T_1 sees value (5) in both selects from the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in both selects from the table $vals$.

Using a traditional repeatable read isolation level transaction T_2 can't update the value on total table because transaction T_1 has taken a shared lock on the same record. However, on multiversion concurrency control no locks are taken for reads, and therefore transaction T_2 can update total table. Furthermore, transaction T_1 does not see the committed changes done by transaction T_1 and thus produces *consistent repeatable read* in other words, relations produced on the first set of selects are equivalent to second set of selects. Thus, the situation is simple when we we have a read only transaction.

The situation becomes more interesting when the transaction is not a read only transaction. Lets again assume that we have the same initialization transaction as in Definition 4.1. Now consider following execution history of transactions T_1 and T_2.

Example 4.2 *Repeatable read with own changes.*

```
T1                          T2
begin;
select * from total;
select * from vals;
                            begin;
                            select * from total;
                            select * from vals;
                            insert into vals values (6,1);
                            update total set s = s + 1;
                            select * from total;
                            select * from vals;
                            commit;
select * from total;
select * from vals;
commit;
```

Naturally, transaction T_1 sees value (5) in both selects from the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in both selects from the table $vals$. Similarly, transaction T_2 sees in the first selects value (5) in the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in the table $vals$. Moreover, transaction T_2 sees it's own changes in the second set of selects, i.e. (6) in the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1),(6,1)$ in the table $vals$.

Transaction T_1 again sees equivalent database state on both sets of the selects and does not see committed changes done by transaction T_2. However, transaction T_2 in the second set of selects sees its own changes. Furthermore. transaction T_2 is valid iff values read on the first set of selects do not change i.e no new versions of these data items are created before commit.

Finally, if both transactions are not read only transactions we have an inconsistent repeatable read. Lets again assume that we have the same initialization transaction as in Definition 4.1. Now consider the following execution history of transactions T_1 and T_2:

Example 4.3 *Inconsistent repeatable read.*

```
T1                          T2
begin;
select * from total;
select * from vals;
                            begin;
                            insert into vals values (6,1);
                            update total set s = s + 1;
                            commit;
insert into vals values (7,1);
update total set s = s + 1;
select * from total;
select * from vals;
commit;
```

Transaction T_1 sees value (5) in the first select from the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in the first select from the table $vals$. Furthermore, transaction T_2 sees (7) from the table $total$ because this is the

*value the transactions itself has created. However, transaction T_1 sees values $(1,1),(2,1),(3,1),(4,1),(5,1),(7,1)$ from the table $vals$. This is inconsistent because now sum(val) from $vals \not\equiv$ select * from $total$. Additionally, transaction T_2 sees in the first selects value (5) in the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in the table $vals$. Moreover, transaction T_2 sees it's own changes in the second set of selects, i.e. (6) in the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1),(6,1)$ in the table $vals$.*

This execution history is possible because transaction T_2 releases all locks on transaction commit. We could prevent the transaction T_1 from seeing its own changes and then result of both sets of selects would be equivalent and consistent. But this would look as a inconsistency from the users point of view. Additionally, if transaction code asserts that change has been successful, this assertion would fail. If we let transaction T_1 see it's own changes but not the committed changes of transaction T_2 we have a inconsistent retrieval assuming that select sum(val) from vals $\equiv$ select * from total. This is because we would see that total table has value 7 but sum of values in val table is 6.

This is why we need to define a new isolation level which we call *real-time multiversion repeatable read*.

Definition 4.2 *Real-time Multiversion Repeatable Read (MCC-RRR). Let T be a transaction and Σ_T be operations defined by transaction. Then transaction T obeys multiversion consistent repeatable read iff:*

1. *(Read only): If we have a read-only transaction and the transaction reads the same data item more than one then the version function always translates each read step to the same version of the data item.*
2. *(Own changes): If a transaction writes a data item all successive reads to the same data item read the version transaction has written.*
3. *(Version upgrade): If a transaction reads the same data item after it has created a new version of another data item the second read might read a newer version of the data item compared to first read before the write.*
4. *(Real-Time rule): If a transaction writes the data item and there is a concurrent transaction that has also read and written the same data item, the transaction may write this data item only if it has greater priority. Thus higher priority transaction has preference for creating a new version of the data item.*

Now we are ready to present algorithms to implement a multiversion real-time repeatable read isolation level. For presentation MVCC-RRR. We begin where transaction T will read the version valid at the start of the transaction $readlevel$ and if the transaction is doing a update, then the transaction will read the latest version of the data item. Identification and read version is stored to the read set RS of the transaction.

Definition 4.3 $read(T, X, readlevel, mode)$*:*

```
if X ∈ RS(T)
  update X in RS(T)
else add X to RS(T)
end if
if mode == read then
  read version(X, readlevel)
else
  read version(X, ∞)
end if
```

In a write a new version of the data item is created and this data item is stored to write set WS of the transaction. If we write a data item we will increase the $readlevel$ to the data item X new version. Thus if we have create a new version of the data item following reads to the same data item will read the created version.

Definition 4.4 $write(T, X, readlevel)$

```
create  new  version of  X;
add X to WS(T)
set readlevel to version(X)
```

Finally, for every data item read by the transaction we must see that there is no other transaction that has also written the same data item and if the data item version is different from the version read originally that only this transaction has created it. If there is active transaction that has read this data item then we may create a new version of the data item if and only if priority of validating transaction is greater than active transaction that has read the data item. This is the fact why we call this real-time method.

Definition 4.5 $validate\_write\_set(T, readlevel, active)$

```
for all X in  RS(T)
  if version(X) != current_version(X)
    if X in WS(T)
      if current_version(X) != version(WS(T))
        return FALSE;
      else
        if X in RS(active) and
          priority(T) < priority(active)
           return FALSE
        endif
      endif
    else
      return FALSE;
    end if
  end if
end for
return TRUE;
```

Lets again assume that we have the same initialization transaction as in Definition 4.1. Now consider the following execution history of transactions T_1 and T_2 and assume that $priority(T_2) > priority(T_1)$ using the MVCC-RRR method:

Example 4.4 *Real-time Multiversion repeatable read isolation level.*

```
T1
begin;
select * from total;
select * from vals;
        T2
        begin;
        insert into vals values (6,1);
        update total set s = s + 1;
        commit;
insert into vals values (7,1);
update total set s = s + 1;
select * from total;
select * from vals; commit;
```

*Now, transaction T_1 sees value (5) in the first select from the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in first select from the table $vals$. Furthermore, transaction T_2 sees (7) from the table $total$ this is because this value is the value the transaction itself has created. Additionally, transaction T_1 sees values $(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,1)$ from the table $vals$. This is consistent because now sum(val) from $vals \equiv$ select * from $total$. Additionally, transaction T_2 sees in the first selects value (5) in the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1)$ in the table $vals$. Moreover, transaction T_2 sees it's own changes in the second set of selects, i.e. (6) in the table $total$ and values $(1,1),(2,1),(3,1),(4,1),(5,1),(6,1)$ in the table $vals$.*

5 Conclusions

Multiversion concurrency control is an attractive choice for concurrency control method in database system because reads can read any version without locking and thus adds more alternatives in controlling the order of reads and writes producing more concurrency to transaction execution. Repeatable read isolation level is the most used isolation level on many applications. However, traditional repeatable read isolation level definition is written with a strict lock-based concurrency control in mind and hence is rather vague when applied to multiversion concurrency control systems. We have presented definition for consistent repeatable read isolation level for multiversion concurrency control and presented algorithms witch implement this definition.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, Mar. 1988.

[2] R. Bayer. On the integrity of data bases and resource locking. In *IBM Symposium: Data Base Systems*, pages 339–361, 1975.

[3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. O'Neil, and O. P. A critique of ansi sql isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[4] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), June 1981.

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

[6] A. Chan, S. Fox, W. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. In *SIGMOD Conference*, pages 184–191, 1982.

[7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[8] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2), 2005.

[9] R. Kataoka, T. Satoh, and U. Inoue. A multiversion concurrency control algorithm for concurrent execution of partial update and bulk retrieval transactions. In *Proceedings of the 10th Phoenix Conference on Computers and Communications*, pages 130–136, 1991.

[10] H. T. Kung and C. H. Papadimitriou. An optimality theory of database concurrency control. In *Proceedings of ACM SIGMOD 1979 Annual Conference*, pages 116–126, 1979.

[11] K.-Y. Lam, S.-L. Hung, and S. H. Son. On using real-time static locking protocols for distributed real-time databases. *The Journal of Real-Time Systems*, 13(2):141–166, Sept. 1997.

[12] X. Liu, J. A. Miller, and N. R. Parate. Transaction management for object-oriented databases: Performance advantages of using multiple versions. In *Proceedings of the 25th Annual Simulation Symposium*, pages 222–231, Los Alamitos, Calif., Apr. 1992. IEEE Computer Society Press.

[13] K. Marzullo. Concurrency control for transactions with priorities. Tech. Report TR 89-996, Department of Computer Science, Cornell University, Ithaca, NY, May 1989.

[14] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[15] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, Apr. 1993.

[16] Y. Raz. Commitment ordering based distributed concurrency control for bridging single and multi version resources. In *RIDE-IMS*, pages 189–198, 1993.

[17] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[18] A. Silberschatz. A multi-version concurrency scheme with no rollbacks. In *PODC*, pages 216–223, 1982.

[19] S. H. Son, S. Park, and Y. Lin. An integrated real-time locking protocol. In *Proceedings of the 8th International Conference on Data Engineering*, pages 527–534, Tempe, Arizona, USA, 1992. IEEE Computer Society Press.

[20] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.

[21] L. Wang and Z. Peng. Extension of multi-version concurrency control mechanisms for long-duration transaction based on nested transaction model. In *CIT*, pages 963–968, 2004.

[22] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann, 2002.