Proceedings of the Fourth IASTED International Conference
Advances in Computer Science and Technology (ACST 2008)
April 2-4, 2008 Langkawi, Malaysia
ISBN CD: 978-0-88986-730-7

MULTIVERSION REPEATABLE READ ISOLATION LEVEL – THEORY AND PRACTICE

Jan Lindström
Solid, an IBM Company
Itälahdenkatu 22 B
00210 Helsinki, Finland
jan.lindstrom@solidtech.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, Algorithms, Database theory.

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 [11]. 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* [6]. 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 [8, 23, 17].

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 [5]. this paper cites a 1978 dissertation by D.P. Reed [16] which describes MVCC and claims it as an original work. Many interesting concurrency control algorithms using multiversioning have been proposed (e.g. [3, 7, 18, 13, 1, 2, 10, 12, 15, 20]).

MVCC is particularly adept at implementing true snapshot isolation [4], 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 [4] 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 [4, 9]. 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.

Under snapshot isolation, however, both T_1 and T_2 can operate on private snapshots of the database: each deducts \$200 from an account, and then verifies that the

new total is zero, using the other account value that held when the snapshot was taken. Since neither update conflicts, both commit successfully, leaving $X = Y = -\$100, X+Y = -\$200$. This non-serializable anomaly is known as *write skew* [9]. ANSI's "REPEATABLE READ" isolation level allows phantom reads, but prevents write skew. In contrast, snapshot isolation allows write skew, but prevents phantom reads. Serializable transactions allow neither.

Rest of the paper is organized as follows. In section 2 we present a formal representation of histories for multiversion concurrency control. Section 3 present a new multiversion repeatable read isolation level to solve problems on traditional repeatable read isolation level. Section 4 presents a performance evaluation where the proposed method is compared with the traditional implementation of repeatable read isolation level on multiversion concurrency control. Finally, Section 5 presents the conclusions of this work.

2 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 [21].

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

Definition 2.1 *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 2.2 *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 2.3 *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 2.3 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 2.4 *Multiversion Multiple Reads. If a transaction reads the same data item more than once these reads are ordered as condition (2) stated in Definition 2.1 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 2.4 can be used if multiple writes to the same data item are allowed. From above we obtain:

Definition 2.5 *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 2.6 *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 2.1 *A multiversion history is multiversion conflict serializable (MVSC) iff its multiversion serialization graph is acyclic.*

Definition 2.7 *Initialization transaction T_0. Lets assume that we have a initialization transaction performing the following steps:*

T_0

```
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 2.7 lets consider following execution history of transactions T_1 and T_2:

Example 2.1 *Read only transaction case.*

```
T1
begin;
select * from total;
select * from vals;
                    T2
                    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 2.7. Now consider following execution history of transactions T_1 and T_2.

Example 2.2 *Repeatable read with own changes.*

```
T1
begin;
select * from total;
select * from vals;
                    T2
                    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 2.7. Now consider the following execution history of transactions T_1 and T_2:

Example 2.3 *Inconsistent repeatable read.*

```
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;
```

*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. Therefore, there is clear need for a new definition for consistent repeatable read isolation level in multiversion concurrency control. This new definition and algorithm is presented in the next section.

3 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 3.1 *Multiversion Repeatable Read. 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* $WS(T) = \emptyset \wedge h(r_{T_n}(x)), h(r_{T_m}(x)) \in \Sigma_T, n > m \Rightarrow h(r_{T_n}(x)) = r_{T_n}(x_j) \wedge h(r_{T_m}(x)) = r_{T_m}(x_j)$.
2. *(Own changes)* $\forall x \in WS(T) \bigcap RS(T) \wedge h(w_T(x)) = w_T(x_t) \prec_\Sigma h(r_T(x)) \Rightarrow h(r_T(x)) = r_T(x_T)$.
3. *(Version upgrade)* $\forall x \in WS(T)$ *if* $\exists y \in RS(T) : h(r_{T_i}[y]) \prec h(w_T[x]) \prec h(r_{T_j}[y]) \wedge j > i \Rightarrow h(r_{T_i}[y]) = r_{T_i}[y_k] \prec h(w_T[x]) = w_T[x_l] \prec h(r_{T_j}[y]) = r_{T_j}[y_m] \wedge k \leq m \leq l$.
4. *(No version changes)* $\forall x \in WS(T)$ *if* $x \in RS(T) \Rightarrow \nexists k : h(r_T[x]) \prec h(x_k[x]) =\prec c_k \prec now, k \neq T$

Condition (1) states that 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. Condition (2) states that if a transaction writes a data item all successive reads to the same data item read the version transaction has written. Condition (3) states that if the 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. Condition (4) states that if a transaction is not read-only transaction then versions read by this transaction must remain the same at the validation phase. Thus, no other transaction has created a new version of those data items.

Now we are ready to present algorithms to implement a multiversion repeatable read isolation level. For presentation MVCC-RR. 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 3.2 $read(T, X, readlevel, mode)$*:*

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

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 3.3 $write(T, X, readlevel)$

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

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.

Definition 3.4 $validate\_write\_set(T, readlevel)$

```
begin
  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
             return FALSE;
        end if
     end if
  end for
```

```
  return TRUE;
end
```

Theorem 3.1 *MVCS $\subset$ MVCC-RR. To show true subset property it is enough to give an example history h where $h \in$ MVCC-RR but $h \notin$ MVCS. Any example where h contains phantom rows is enough. Lets assume initialization transaction described in Definition 2.7. For example:*

```
T1
begin;
select * from vals where i between 1 and 5;
                          T2
                           begin;
                           insert into vals values(4,1);
                           commit;
update vals set val = 2 where i = 2;
select * from vals where i between 1 and 5;
commit;
```

Corollary 3.1 *MVCC-RR $\in P$. Thus MVCC-RR belongs to class P and can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input.*

Lets again assume that we have the same initialization transaction as in Definition 2.7. Now consider the following execution history of transactions T_1 and T_2 using the MVCC-RR method:

Example 3.1 *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 $vals$.*

4 Performance evaluation

We have carried out a set of experiments in order to examine the feasibility of our prototype implementation, specifically the concurrency control mechanism. All experiments were executed in the MySQL database running on a AMD Opteron Processor 146 processor containing 2 GB of main memory with the Linux operating system 2.6.20.

The test database represents a typical network database that mimics a Home Location Register (HLR) [22] which is used to store information about users of the network. Operators use HLR databases to store subscriber data, location data, network access data, and about network services data, for example call forwarding. To simplify presentations schema presented below does not contain all the operations of the HLR. Instead database and transactions are from The Telecom One (TM1) benchmark designed for telecommunication applications [19].

In the Figure 1 we have compared overall performance of different MVCC implementations in MySQL/InnoDB and MySQL/solidDB storage engines. MySQL/InnoDB implements traditional MVCC using locks while MySQL/solidDB uses MVCC-RR method presented in this paper. This experiment clearly shows that MVCC-RR provides a lot better overall performance because it allows more concurrency between different type of transactions.

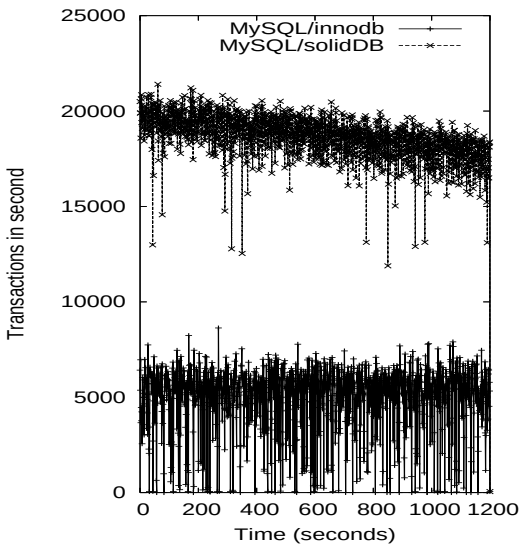

Figure 1. MVCC methods compared.

5 Conclusion

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. We have shown that these algorithms produce correct results.

References

[1] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed multi-version optimistic concurrency control for relational databases. In *COMPCON*, pages 416–421, 1986.

[2] M. Ahuja and J. C. Browne. Concurrency control by pre-ordering entities in databases with multiversioned entities. In *ICDE*, pages 312–321, 1987.

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

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

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

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

[7] 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.

[8] 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, November 1976.

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

[10] 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.

[11] 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.

[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., April 1992. IEEE Computer Society Press.

[13] S. Muro, T. Kameda, and T. Minoura. Multi-version concurrency control scheme for a database system. *J. Comput. Syst. Sci.*, 29(2):207–224, 1984.

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

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

[16] 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.

[17] D. R. Ries and M. Stonebraker. Locking granularity revisited. *ACM Trans. Database Syst.*, 4(2):210–227, 1979.

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

[19] T. Strandell. Open source database systems: Systems study, performance and scalability. Master's thesis, 2003.

[20] 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.

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

[22] Wikipedia. Network switching subsystem. Technical report, http://en.wikipedia.org/wiki/Home_Location_Register.

[23] M. Yannakakis, C. H. Papadimitriou, and H. T. Kung. Locking policies: Safety and freedom from deadlock. In *20th Annual Symposium on Foundations of Computer Science*, pages 286–297, 1979.