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 [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, T₁ withdrawing $200 from X, and T₂ withdrawing $200 from Y.

If the database guaranteed serializable transactions, the simplest way of coding T₁ is to deduct $200 from X, and then verify that $X + Y \geq 0$ still holds, aborting if not. T₂ similarly deducts $200 from Y and then verifies $X + Y \geq 0$. Since the transactions must serialize, either T₁ happens first, leaving $X = -$100, $Y = 100$, and preventing T₂ from succeeding or T₂ happens first and similarly prevents T₁ from succeeding.

Under snapshot isolation, however, both T₁ and T₂ 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 con-
flicts, 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 al-
low 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 prob-
lems 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 \( \Sigma \) and

1. \( \Sigma = h(\bigcup_{i=1}^{n} \Sigma_i) \) for some function \( h \)
2. \( \forall t \in T \) and \( \forall p, q \in \Sigma \) the following holds:
   \( p \prec_{\Sigma} q \Rightarrow h(p) \prec_{\Sigma} h(q) \)
3. If \( h(r_j(x)) = r_j(x_i), i \neq j \) and \( c_i \in \Sigma \) 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. Con-
dition (2) states that history function preserves all orderings
defined by transactions. Condition (3) states that a transac-
tion may not read a version until it has been produced.

**Definition 2.2** Reads-From Relation. Let \( \Sigma \) be a multiver-
sion schedule, \( t_j \in \Sigma \) transactions. The reads-from rela-
tion of \( \Sigma \) 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 con-
flict in a multiversion schedule \( \Sigma \) 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 dif-
f erent 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 trans-
action 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
\( h(r_jk(x)) \in \Sigma_j \) and \( h(r_jk+1(x)) \in \Sigma_{j+1} \) then
\( h(r_jk(x)) \prec_{\Sigma} h(r_jk+1(x)) \). Note it may hold that two con-
secutive reads in the transaction do not read the same
version of the data item i.e \( h(r_jk(x)) = r_jk(x_i) \) and
\( h(r_jk+1(x)) = r_jk+1(x_l) \) where \( i \neq l \).

In this work we assume that the transaction writes a
data item at most once. Similarly, the construction in De-
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 \( \Sigma \) is multiversion conflict serializable
if there is a serial monoversion history for the same set of
transactions in which all pairs of operations in multiver-
sion conflict occur in the same order as in \( \Sigma \).

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 mul-
tiversion serialization graph.

**Definition 2.6** Multiversion Serialization Graph (MVSG).
For a given schedule \( \Sigma \) and a version order \( \prec \), the
multiversion serialization graph MVSG(\( \Sigma, \prec \)) of \( \Sigma \) then is
the conflict graph \( G(\Sigma) = (V, E) \) with the following edges
added to each \( r_k(x_j) \) and \( w_i(x_t) \) in committed projection
of \( \Sigma \), where \( k, i \) and \( j \) are pairwise distinct: if \( i \prec x_j \),
then \( (t_i, t_j) \in E \), otherwise \( (t_k, t_l) \in E \).

Not surprisingly, we obtain:

**Theorem 2.1** A multiversion history is multiversion con-
flict serializable (MVSC) iff its multiversion serialization
graph is acyclic.
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.**

\[
\begin{align*}
T_1 &\begin{align*}
&\text{begin;} \\
&\text{select * from total;} \\
&\text{select * from vals;} \\
&\text{commit;}
\end{align*} \\
T_2 &\begin{align*}
&\text{begin;} \\
&\text{insert into vals values (6,1);} \\
&\text{update total set s = s + 1;} \\
&\text{commit;}
\end{align*}
\]

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 \( T_2 \) 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 have a read only transaction.

The situation becomes more interesting when the transaction is not a read only transaction. Let’s 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.**

\[
\begin{align*}
T_1 &\begin{align*}
&\text{begin;} \\
&\text{select * from total;} \\
&\text{select * from vals;} \\
&\text{commit;}
\end{align*} \\
T_2 &\begin{align*}
&\text{begin;} \\
&\text{select * from total;} \\
&\text{select * from vals;} \\
&\text{insert into vals values (6,1);} \\
&\text{update total set s = s + 1;} \\
&\text{commit;}
\end{align*}
\]

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. Let’s 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.**

\[
\begin{align*}
T_1 &\begin{align*}
&\text{begin;} \\
&\text{select * from total;} \\
&\text{select * from vals;} \\
&\text{commit;}
\end{align*} \\
T_2 &\begin{align*}
&\text{begin;} \\
&\text{insert into vals values (6,1);} \\
&\text{update total set s = s + 1;} \\
&\text{commit;}
\end{align*}
\]

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 ≠ 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 \( \Sigma_T \) be operations defined by transaction. Then transaction \( T \) obeys multiversion consistent repeatable read iff:

1. (Read only) if \( W S(T) = \emptyset \) and \( h(r_{T_n}(x)) \in \Sigma_T, n > m \Rightarrow h(r_{T_n}(x)) = r_{T_m}(x_j) \) then \( r_{T_n}(x_j) \).
2. (Own changes) \( \forall x \in W S(T) \cap RS(T) \land h(w_T(x)) = w_T(x_i) \Leftrightarrow h(r_T(x)) \Rightarrow h(r_T(x)) = r_T(x_T). \)
3. (Version upgrade) \( \forall x \in W S(T) \) if \( \exists y \in RS(T) : h(r_T[y]) \prec h(w_T[x]) \prec h(r_T[y]) \land j > i \Rightarrow h(r_T[y]) = r_T[y] \prec h(w_T[x]) = w_T[x] \prec h(r_T[y]) = r_T[y] \land k \leq l. \)
4. (No version changes) \( \forall x \in W S(T) \) 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 once 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 \( \text{read}(T, X, \text{readlevel}, \text{mode}) : \)

begin
if \( X \in RS(T) \)
update \( X \) in \( RS(T) \)
else
add \( X \) to \( RS(T) \)
end if
if \( \text{mode} = \text{read} \)
read version\((X, \text{readlevel})\)
else
read version\((X, \infty)\)
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 \( \text{readlevel} \) 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 \( \text{write}(T, X, \text{readlevel}) \)

begin
create new version of \( X \);
add \( X \) to \( WS(T) \);
set \( \text{readlevel} \) 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 \( \text{validate_write_set}(T, \text{readlevel}) \)

begin
for all \( X \) in \( RS(T) \)
if \( \text{version}(X) \neq \text{current_version}(X) \)
if \( X \) is in \( WS(T) \)
if \( \text{current_version}(X) \neq \text{version}(WS(T)) \)
return FALSE;
else
return FALSE;
end if
end if
end for
Theorem 3.1 \( \text{MVCS} \subseteq \text{MVCC-RR} \). To show true subset property it is enough to give an example history \( h \) where \( h \in \text{MVCC-RR} \) but \( h \notin \text{MVCS} \). Any example where \( h \) contains phantom rows is enough. Let assume initialization transaction described in Definition 2.7. For example:

\[
T_1 \\
\begin{align*}
&\text{begin;}
&\text{select * from vals where } i \text{ between 1 and 5;}
&T_2 \\
&\begin{align*}
&\text{begin;}
&\text{select * from total;}
&\text{select * from vals;}
&T_1 \\
&\text{insert into vals values(6,1);}
&\text{update vals set val = 2 where } i = 2;
&\text{select * from vals where } i \text{ between 1 and 5;}
&\text{commit;}
&\text{insert into vals values(7,1);}
&\text{update total set } s = s + 1;
&\text{commit;}
&\text{commit;}
&\quad \text{select * from vals;}
&\quad \text{commit;}
&\end{align*}
&\text{commit;}
&\end{align*}
\]

Corollary 3.1 \( \text{MVCC-RR} \in \text{P} \). Thus \( \text{MVCC-RR} \) 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.

Let 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.

\[
T_1 \\
\begin{align*}
&\text{begin;}
&\text{select * from vals where } i \text{ between 1 and 5;}
&T_2 \\
&\begin{align*}
&\text{begin;}
&\text{select * from total;}
&\text{select * from vals;}
&T_1 \\
&\text{insert into vals values(6,1);}
&\text{update vals set val = 2 where } i = 2;
&\text{select * from vals where } i \text{ between 1 and 5;}
&\text{commit;}
&\text{insert into vals values(7,1);}
&\text{update total set } s = s + 1;
&\text{commit;}
&\quad \text{select * from total;}
&\quad \text{select * from vals;}
&\quad \text{commit;}
&\end{align*}
&\end{align*}
\]

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 \( \text{sum(val)} \) from vals \( \equiv \text{select * from total} \). 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 an 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.

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 which implement this definition. We have shown that these algorithms produce correct results.

References


