A Theoretical Study of 'Snapshot Isolation'

Ragnar Normann
Department of Informatics
University of Oslo
ragnarn@ifi.uio.no

Lene T. Østby
Department of Informatics
University of Oslo
lene.t.ostby@accenture.com

ABSTRACT

Snapshot Isolation is a popular and efficient protocol for concurrency control. In this paper we discuss Snapshot Isolation in view of the classical theory for transaction processing. In addition to summarizing previous research we prove that the set SI of histories that may be generated by Snapshot Isolation is incomparable to final state, view and conflict serializability, that SI is monotone, and that schedules generated by Snapshot Isolation are strict and thus have good properties with respect to recoverability.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Transaction processing*

General Terms

Theory

Keywords

Snapshot Isolation, concurrency, transaction, serializability, monotonicity, recoverability

1. INTRODUCTION AND MOTIVATION

Snapshot Isolation is a multiversion protocol for concurrency control that is quite popular and heavily used. To our knowledge the first commercial available implementation was in Borland's multiversion concurrency control database InterBase 4 released in 1994. Later, among others, Oracle, PostgreSQL and Microsoft SQL Server have implemented versions of it [5] (but not IBM's DB2). Snapshot Isolation is attractive because it offers an isolation level that both avoids most concurrency phenomena and anomalies and still allows a much higher degree of concurrency than strict 2-phase locking (the traditional conservative protocol for concurrency control).

The first presentation of Snapshot Isolation in the litterature was made by Berenson & al. in [2]. They discussed Snapshot Isolation in relation to concurrency phenomena and isolation levels. A later paper by Fekete & al. [5] focuses on identifying cases where Snapshot Isolation will not comply with the isolation level serializable, and how to avoid these (e.g. at the application level). Papers discussing other aspects of Snapshot Isolation are, at best, rare.

In this paper we relate Snapshot Isolation to the well-known theory of serializability and recoverability. Most of our results were first presented in the master thesis [13] of the second author.

The organization of the rest of the paper is as follows: Some useful background material is given in section 2 before the Snapshot Isolation protocol is presented in section 3. Sections 4 and 5 summarize the previous research relating Snapshot Isolation to concurrency phenomena and anomalies and to the SQL isolation levels, Sections 6 through 8 discuss Snapshot Isolation and serializability, monotonicity and recoverability. Some final remarks are given in section 9.

2. NOTATION AND THEORETICAL FRAMEWORK

We have chosen to use a terminology and notation very close to the one used by Weikum & Vossen in their book [11].

Definition 1. A transaction t_i is a finite set of operations op_i with a strict (linear) order $<_i$. Each operation in op_i is either a read operation, denoted $r_i(x)$, where x is the data item read by t_i, or a write operation, denoted $w_i(y)$, where y is the data item written by t_i.

The semantics of $<_i$ is ordering in time, i.e., if p and q are two operations in op_i with $p <_i q$ then p is executed prior to q. The operations are atomic, i.e., when an operation in op_i is executed, this execution is finished before the next operation in op_i can be executed. The execution of an operation is called a *step*. Thus the execution of a transaction is a finite sequence of steps.

Definition 2. ([11, definition 3.1]) Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of transactions, where each $t_i \in T$ has the form $t_i = (\mathrm{op}_i, <_i)$, with op_i denoting the set of operations in t_i and $<_i$ denoting their order, $1 \leq i \leq n$.

1. A history for T is a pair $s = (\mathrm{op}(s), <_s)$ such that:

(a) $\mathrm{op}(s) \subseteq \bigcup_{i=1}^{n} \mathrm{op}_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$ and $\bigcup_{i=1}^{n} \mathrm{op}_i \subseteq \mathrm{op}(s)$, i.e., s consists of the union of operations from the given transactions plus some (terminating) operations, each of which may either be a c_i (commit) or an a_i (abort);

(b) $(\forall i, 1 \leq i \leq n) c_i \in \mathrm{op}(s) \Leftrightarrow a_i \notin \mathrm{op}(s)$, i.e., for each transaction, there is either a commit or an abort in s, but not both;

(c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$, i.e., all transaction orders are contained in the partial order given by s;

(d) $(\forall i, 1 \leq i \leq n)(\forall p \in \mathrm{op}_i) p <_s a_i$ or $p <_s c_i$, i.e., the commit or abort operation always appears as the last step of a transaction;

(e) each pair of operations $p, q \in \mathrm{op}(s)$ from distinct transactions that accsess the same data item and have at least one write operation among them is ordered in s in such a way that either $p <_s q$ or $q <_s p$.

The transactions participating in a history s is denoted $\mathrm{trans}(s)$, i.e., $\mathrm{trans}(s) = T$.

2. A schedule is a prefix of a history.

In plain language, definition 2 states that a history h for T is a partial order of all steps in the transactions in T such that

- the internal order of each transaction's steps is preserved in h
- each transaction in T is terminated by either a commit or an abort in h
- all read-write or write-write conflicts between different transactions in T are ordered in h

When working with histories it is convenient to introduce two (fictious) transactions: one initializing transaction t_0 that writes the initial state of the database (and commits) before the execution of the history is started, and one final (read-only) transaction t_∞ that reads the whole database after the execution of the history is finished.

As mentioned in the introduction, Snapshot Isolation is a multiversion protocol. We therefore need the following two definitions:

Definition 3. ([11, definition 5.1]) Let s be a history with initializing transaction t_0 and final transaction t_∞. A version function for s is a function h, which associates whith each read step of s a previous write step on the same data item, and which is the identity on write steps.

Definition 4. ([11, definition 5.2]) Let $T = \{t_1, \ldots, t_n\}$ be a (finite) set of transactions.

1. A multiversion history for T is pair $m = (\mathrm{op}(m), <_m)$, where $<_m$ is an order on $\mathrm{op}(m)$ and

(a) $\mathrm{op}(m) = h(\bigcup_{i=1}^{n} \mathrm{op}(t_i))$ for some version function h

(b) for all $t \in T$ and all operations p, $q \in \mathrm{op}(t)$ the following holds:

$$p <_t q \Rightarrow h(p) <_m h(q)$$

(c) if $h(r_j(x)) = w_i(x_i), i \neq j$, and c_j is in m, then c_i is in m and $c_i <_m c_j$.

2. A multiversion schedule is a prefix of a multiversion history.

Finally, to be able to compare isolation levels we need the following definition from Berenson & al. [2]:

Definition 5. An isolation level L_1 is *weaker* than an isolation level L_2 (or L_2 is *stronger* than L_1), denoted $L_1 \ll L_2$, if all non-serializable histories that obey the citeria of level L_2 also satisfy level L_1, and there is at least one non-serializable history that can occur at level L_1 but not at level L_2. Two isolation levels are *incompareable*, denoted $L_1 \gg\ll L_2$, when each isolation level allows a non-serializable history that is disallowed by the other.

Note that this definition only takes the non-serializable histories into account. It is irrelevant which serializable histories that may be disallowed.

3. THE SNAPSHOT ISOLATION PROTOCOL

Definition 6. The Snapshot Isolation Protocol produces multiversion schedules by enforcing the following two rules:

1. When a transaction t reads a data item x, t reads the newest version of x written by a transaction that committed before t started.

2. The write sets of two concurrent transactions must be disjoint.

The set of histories that may be produced by the Snapshot Isolation protocol is denoted SI.

Rule 1 states that the version function maps each read action $r_i(x)$ to the most recent committed write action $w_j(x)$ at the time t_i started.

Rule 2 states that if t_1 and t_2 are two transactions where t_1 starts before t_2, and t_1 commits after t_2 has started, t_1 and t_2 are not permitted to write the same data item.

There are several methods to enforce rule 2. One is to compare write sets at commit. Oracle has implemented a more efficient algorithm which in Fekete et al. [6] is called 'first updater wins'. This algorithm works as follows:

Assume two transactions t_1 and t_2 are concurrent, t_1 writes x, and t_2 also wants to write x. Then t_2 cannot write x before t_1 has released its lock on x. Then there are three possibilities:

- If t_2 is queued to write x and t_1 commits, t_2 is immediately aborted.
- If t_1 commits before t_2 tries to write x, t_2 is aborted when it tries to perform the write.
- If t_1 drops its lock because it aborts, t_2 is allowed to write x.

Finally, here is the rule for when old versions are superfluous:

- A version x_i of a data item x may be deleted if, and only if, there is a newer version x_k of x, and all active transactions were started after t_k was committed.

4. CONCURRENCY PHENOMENA AND ANOMALIES

We base our discussion mainly on Berenson & al. [2], and throughout this section we make heavy use of their definitions and examples. In [2] they distinguish between concurrency phenomena (denoted by P) – which *may* result in errors – and concurrency anomalies (denoted by A) – which *always* result in errors. We follow their nomenclature in the list below:

P0 – Dirty Write
Example: $w_1(x)w_2(x)$ (c_1 or a_1)

P1 – Dirty Read
Example: $w_1(x)r_2(x)$ (c_1 or a_1)

P2 – Fuzzy or Non-Repeatable Read
Example: $r_1(x)w_2(x)$ (c_1 or a_1)

P3 – Phantom
Example: $r_1(P)w_2(y \text{ in } P)$ (c_1 or a_1)
The P in the example stands for 'Predicate' and is typically the result set of a SQL WHERE-clause. So if t_1 reads the data items that satisfy P (e.g. to evaluate some aggregate function on them) whereupon t_2 writes a new data item satisfying P before t_1 commits, t_1 will commit with an incorrect answer.

A stricter interpretation of P3 will transform it into an anomaly:

A3 – Phantom
Example: $r_1(P)w_2(\text{insert } y \text{ to } P)c_2r_1(P)c_1$
Here t_1 executes a query satisfying the predicate P twice, finding that the result has changed.

P4 – Lost Update
Example: $r_1(x)w_2(x)w_1(x)c_1$

A5A – Read Skew
Example: $r_1(x)w_2(x)w_2(y)c_2r_1(y)$ (c_1 or a_1)

A5B – Write Skew
Example: $r_1(x)r_2(y)w_1(y)w_2(x)$ (c_1 and c_2 in some order)

A6 – Read-Only Transaction Anomaly
Example: $r_2(x)w_1(y)c_1r_3(x)r_3(y)c_3w_2(x)c_2$
Snapshot Isolation was originally trusted to never present incorrect data from read-only transactions if no concurrent update transactions wrote incorrect values. But in [6] Fekete et al. presented the above example showing that this is not the case. In the example the commit order differs from the serial order $t_1t_2t_3$, and since t_1 and t_2 write different data items, Snapshot Isolation will not detect this Read-Only anomaly.

THEOREM 1. (Berenson & al.[2], Fekete & al.[6]) *Of the ten concurrency phenomena and anomalies listed above, only these three may occur in a schedule produced by Snapshot Isolation:*

P3: Phantom
A5B: Write skew
A6: Read-only transaction anomaly

5. SQL ISOLATION LEVELS

Isolation levels were introduced in the SQL-92 standard [1], [9, section 14.3]. This standard defines four levels of isolation which SQL-servers should support (the levels are ranked from the strongest to the weakest):

Serializable. A history generated at isolation level Serializable shall prohibit all concurrency phenomena and anomalies, in fact it shall produce the same result as some serial execution of the transactions in the history.

Repeatable Read. All phenomena and anomalies but phantoms are prohibited.

Read Committed. All phenomena and anomalies except dirty writes and reads may occur.

Read Uncommitted. Only dirty writes are prohibited.

THEOREM 2. (Berenson & al.[2])
ReadCommitted $\ll$ SI $\ll$ Serializable *while*
SI $\gg\ll$ RepeatableRead

6. SERIALIZABILITY

Chapter 3 in the book [11] of Weikum and Vossen is a good source for the theory of serialization. We use their definitions and refer to their book for a more comprehensive treatment of this subject.

Definition 7. Let s be a history of n transactions $t_1, \ldots, t_n$ with initializing transaction t_0 and final transaction t_∞, and let D be the set of data items read or written by some t_k. For each $x \in D$ we recursively define the Herbrand semantics of each read step $r_k(x)$ and each write step $w_k(x)$ in s by

1. $H_s(w_0(x)) = f_{0x}()$ where $f_{0x}()$ is a 0-ary function (a constant – the initial value of x)
2. $H_s(r_k(x)) = H_s(w_p(x))$ where $t_p (p \neq k)$ is the last transaction to write x prior to t_k reading x
3. $H_s(w_k(x)) = f_{kx}(H_s(r_k(y_1)), \ldots H_s(r_k(y_m)))$ where $r_k(y_1), \ldots, r_k(y_m)$ are all read steps performd by t_k prior to $w_k(x)$, and f_{kx} is an uninterpreted m-ary function symbol

The Herbrand semantics of the history s is the function $H[s]$ from D into the universe of Herbrand formulas defined by:

$$H[s](x) = H_s(r_\infty(x)), x \in D$$

Definition 8. Two histories s_1 and s_2 are final state equivalent if they contain the same transactions and $H[s_1] = H[s_2]$.

A history is final state serializable if it is final state equivalent to a serial history.

The set of all finite state serializable histories is denoted FSR.

Definition 9. Two histories s_1 and s_2 are view equivalent if they are final state equivalent and $H_{s_1}(p) = H_{s_2}(p)$ for all (read and write) steps p.

A history is view serializable if it is view equivalent to a serial history.

The set of all view serializable histories is denoted VSR.

Definition 10. The conflict relation of a schedule s, $\mathrm{conf}(s)$, is defined as the set of all pairs (p, q) where $p <_s q$ and the ordering is a consequence of rule (e) in definition 2, i.e., p and q are operations in two different transactions in s which are in either a read-write or a write-write conflict.

Two histories s_1 and s_2 are conflict equivalent if they contain the same transactions and $\mathrm{conf}(s_1) = \mathrm{conf}(s_2)$.

A history is conflict serializable if it is conflict equivalent to a serial history.

The set of all conflict serializable histories is denoted CSR.

The following result is considered well-known (see e.g. [11, corollary 3.3]):

THEOREM 3. CSR $\subset$ VSR $\subset$ FSR

THEOREM 4. SI *is neither included in, nor includes, any of the serializability classes* FSR, VSR *or* CSR.

PROOF. Due to theorem 3 it is sufficient to find two histories h_1 and h_2 such that $h_1 \in \mathrm{CRS} \setminus \mathrm{SI}$ and $h_2 \in \mathrm{SI} \setminus \mathrm{FSR}$.

For h_1 we may use the example of a dirty read (P1 in section 4):

$$h_1 = w_1(x)w_2(x)c_1c_2$$

h_1 is conflict equivalent to t_1t_2, and thus in CSR. But since t_1 and t_2 are two concurrent transactions that both write x, h_1 is not in SI.

For h_2 the example of a write skew (A5B in section 4) will do:

$$h_2 = r_1(x)r_2(y)w_1(y)w_2(x)c_1c_2$$

Both reads are of the initial value and the two write sets are disjoint, so $h_2 \in \mathrm{SI}$. To prove the theorem it remains to show that $h_2 \notin \mathrm{FSR}$, i.e. that h_2 is not final state equivalent to any of the two possible serial histories:

$$s_1 = t_1t_2 = r_1(x)w_1(y)c_1r_2(y)w_2(x)c_2$$

$$s_2 = t_2t_1 = r_2(y)w_2(x)c_2r_1(x)w_1(y)c_1$$

To prove this we calculate the Herbrand semantics:

$$\begin{aligned}
H[h_2](x) &= H_{h_2}(r_\infty(x)) = H_{h_2}(w_2(x)) = f_{2x}(H_{h_2}(r_2(y))) \\
&= f_{2x}(H_{h_2}(w_0(y))) = f_{2x}(f_{0y}()) \\
H[s_1](x) &= H_{s_1}(r_\infty(x)) = H_{s_1}(w_2(x)) = f_{2x}(H_{s_1}(r_2(y))) \\
&= f_{2x}(H_{s_1}(w_1(y))) = f_{2x}(f_{1y}(H_{s_1}(r_1(x))) \\
&= f_{2x}(f_{1y}(H_{s_1}(w_0(x))) = f_{2x}(f_{1y}(f_{0x}()) \\
H[s_2](x) &= H_{s_2}(r_\infty(x)) = H_{s_2}(w_2(x)) = f_{2x}(H_{s_2}(r_2(y))) \\
&= f_{2x}(H_{s_2}(w_0(y))) = f_{2x}(f_{0y}())
\end{aligned}$$

$$\begin{aligned}
H[h_2](y) &= H_{h_2}(r_\infty(y)) = H_{h_2}(w_1(y)) = f_{1y}(H_{h_2}(r_1(x))) \\
&= f_{1y}(H_{h_2}(w_0(x))) = f_{1y}(f_{0x}()) \\
H[s_1](y) &= H_{s_1}(r_\infty(y)) = H_{s_1}(w_1(y)) = f_{1y}(H_{s_1}(r_1(x))) \\
&= f_{1y}(H_{s_1}(w_0(x))) = f_{1y}(f_{0x}()) \\
H[s_2](y) &= H_{s_2}(r_\infty(y)) = H_{s_2}(w_1(y)) = f_{1y}(H_{s_2}(r_1(x))) \\
&= f_{1y}(H_{s_2}(w_2(x))) = f_{1y}(f_{2x}(H_{s_2}(r_2(y))) \\
&= f_{1y}(f_{2x}(H_{s_2}(w_0(y))) = f_{1y}(f_{2x}(f_{0y}())
\end{aligned}$$

Since $H[h_2](x) \neq H[s_1](x)$ and $H[h_2](y) \neq H[s_2](y)$ our proof is complete. $\square$

7. MONOTONICITY

The concept *monotone histories* was introduced by Yannakakis [12]:

'The greater the number of transactions that run concurrently in the system, the less the chances that a request of a transaction to read or write some data will be granted immediately. Or in other words, if a request is granted under a given load, it would also be granted if the load were lighter (i.e., if some of the other transactions were not present).'

Yannakis called histories adhering to this principle *monotone.* To give a more formal description of monotonicity Yannakakis introduced the concept *subschedule* which Weikum & Vossen [11, end of section 3.7]) call a *projection*:

Definition 11. Let s be a schedule for a set T of transactions, and let $U \subseteq T$. The projection of s on U, denoted $\Pi_U(s)$, is what we get if we from s remove all operations performed by all transactions in $T \setminus U$.

Schedulers use projections mainly to handle aborts: Whenever one or more transactions in a schedule abort, the schedule is replaced by the projection of the schedule on its non-aborted transactions. In fact, the same happens when a transaction commits: The transaction is added to the history of all committed transactions and removed from the schedule by a projection, but projections caused by commits can not create any problems.

Definition 12. A class E of histories (legal schedules) is said to be monotone if the following holds: If s is in E then all projections of s are in E.

The importance of monotonicity is that a monotone class is closed under aborts. The following example illustrates this and shows why it seems almost impossible to make a scheduler for a class which is not monotone:

Example Let E be the class of schedules a given scheduler $\mathcal{S}$ may produce, i.e. E is the class of legal schedules, and assume that E is not monotone. Then the following may happen:

$\mathcal{S}$ makes a schedule s for a set of transactions T.
A transaction $t \in T$ aborts.
$\Pi_{T\setminus\{t\}}(s) \notin E$ (construe an illegal schedule).

Another problem is that an illegal schedule s may become legal if a new transaction arrives to be interleaved with s.

We claim that this example proves the importance of monotonicity.

Both Yannakakis [12] and Weikum & Vossen [11] have tacitly assumed that their database is not multiversion when discussing monotonicity. In fact, as far as we can see, no discussions of monotonicity in multiversion databases exist, neither in the litterature, nor on the net. Therefore we include a short discussion of this topic:

The main problem is how to define projections. Consider the following multiversion history of three transactions t_1, t_2 and t_3:

$$s = r_1(x_0)r_1(y_0)r_2(y_0)w_2(y_2)c_2r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$$

An uncritical use of definition 11 to calculate the projection of s on $T = \{t_1, t_3\}$ yields

$$\Pi_T(s) = r_1(x_0)r_1(y_0)r_3(x_0)r_3(y_2)c_3w_1(x_1)c_1$$

Here t_3 reads a version of y written by t_2 which does not appear in $\Pi_T(s)$. In fact, if the projection is performed because t_2 aborts prior to writing y, the version y_2 does not exist when t_3 reads y. Thus, in the general case, the version function (see definition 3) has to be recalculated during the projection.

A more thorough discussion of this problem is beyond the scope of this paper, but, as we shall see, our next result is independent of the outcome of such a discussion.

Theorem 5. *The class* SI *is monotone.*

Proof. Let s be a schedule generated by Snapshot Isolation, let p be the projection of s on a subset of the transactions in s, i.e., p is a schedule for the non-aborted transactions in s, and let t be a transaction in p reading a data item x.

When the Snapshot Isolation scheduler constructed s it scheduled t to read the last version x_k of x written by a transaction t_k that committed before t started.

Since t_k is committed it can not be among the transactions in s that are not in p. Thus x_k is still the last version of x written by a transaction which committed before t started even if some of the transactions in s are aborted.

Finally we know that s does not contain any write-write conflicts. Removing transactions from s cannot create any new conflicts, so p does not contain any write-write conflicts either.

Thus both criteria of definition 6 are proved, and we conclude that p is in SI. It follows that SI is monotone. ☐

8. RECOVERABILITY

Again, we borrow the terminology of Weikum & Vossen [11] and cite their definitions:

Definition 13. ([11, definition 11.5]) A schedule s is *recoverable* if the following holds for all transactions $t_i, t_j \in \text{trans}(s), i \neq j$: if t_i reads from t_j in s and $c_i \in \text{op}(s)$, then $c_j <_s c_i$.
Let RC denote the class of all recoverable schedules.

Definition 14. ([11, definition 11.6]) A schedule s *avoids cascading aborts* if the following holds for all transactions $t_i, t_j \in \text{trans}(s), i \neq j$: if t_i reads X from t_j in s, then $c_j <_s r_i(x)$.
Let ACA denote the class of all schedules that avoid cascading aborts.

Definition 15. ([11, definition 11.7]) A schedule s is *strict* if the following holds for all transactions $t_i \in \text{trans}(s)$ and for all $p_i(x) \in \text{op}(t_i), p \in \{r, w\}$: if $w_j(x) <_s p_i(x), i \neq j$: then $a_j <_s p_i(x) \vee c_j <_s p_i(x)$.
Let ST denote the class of all strict schedules.

Definition 16. ([11, definition 11.8]) A schedule s is *rigorous* if it is strict and additionally satisfies the following condition: for all transactions $t_i, t_j \in \text{trans}(s)$, if $r_j(x) <_s w_i(x), i \neq j$, then $a_j <_s w_i(x) \vee c_j <_s w_i(x)$.
Let RG denote the class of all rigorous schedules.

Theorem 6. ([11, theorem 11.2])
RG ⊂ ST ⊂ ACA ⊂ RC

Theorem 7. SI ⊂ ST, SI $\not\subseteq$ RG *and* RG $\not\subseteq$ SI

Proof. First, assume that $s \in$ SI and that $t_i, t_j \in \text{trans}(s)$, $i \neq j$, have a conflict $w_j(x) <_s r_i(x)$. Since t_i only reads values that are committed before t_i starts, we must have $a_j <_s r_i(x) \vee c_j <_s r_i(x)$. Since s does not contain any write-write conflicts, this shows s to be strict, so SI ⊆ ST.

Next, consider the (Write Skew (A5B in section 4)) history

$$s_1 = r_1(x)r_1(y)r_2(x)r_2(y)w_1(y)c_1w_2(x)c_2$$

Here all reads are of the initial value and the write sets of t_1 and t_2 are disjoint so $s_1 \in \mathrm{SI}$ by definition 6. But since t_1 writes y after t_2 reads y but before t_2 commits, by definition 16 $s_1 \notin \mathrm{RG}$. Hence $s_1 \in \mathrm{SI} \setminus \mathrm{RG}$ which shows $\mathrm{SI} \nsubseteq \mathrm{RG}$.

To prove the last claim, consider the history

$$s_2 = r_1(x)r_2(y)w_1(x)c_1r_2(x)c_2$$

s_2 contains no write-write conflicts and all reads are of comitted values, so by definition 15 $s_2 \in \mathrm{ST}$. Neither does s_2 contain any read-write conflicts, so by definition 16 $s_2 \in \mathrm{RG}$. But since t_2 reads a value of x written after t_2 executed its first operation, $s_2 \notin \mathrm{SI}$. Hence $s_2 \in \mathrm{RG} \setminus \mathrm{SI}$, so $\mathrm{RG} \nsubseteq \mathrm{SI}$.

Finally, by theorem 6 we have $\mathrm{RG} \setminus \mathrm{SI} \subseteq \mathrm{ST} \setminus \mathrm{SI}$. Thus we have $s_2 \in \mathrm{ST} \setminus \mathrm{SI}$ which prove the inclusion $\mathrm{SI} \subseteq \mathrm{ST}$ to be strict. $\square$

9. CONCLUSIONS AND FUTURE WORK

Our main results are that SI neither includes CSR nor is included in FSR, that SI is monotone, and that all histories in SI are strict. In [2] Berenson & al. showed that SI is a stronger isolation level than read committed. Thus Snapshot Isolation guarantees a reasonably strong level of isolation and produces easily recoverable histories. Since SI is monotone the simplicity of the Snapshot Isolation Protocol makes it easy to implement an efficient scheduler for SI. So in spite of the fact that Snapshot Isolation may generate histories that are not final state serializable, we think it is a wise decision of the DBMS vendors to offer Snapshot Isolation as an isolation level.

Note: Snapshot Isolation vs. Serializable

Among others, both Oracle and PostgreSQL use Snapshot Isolation to implement their isolation level 'Serializable'. According to theorem 2 this is not correct. In fact, this is an explicit violation of the SQL-99 standard [8]. Of the other major DBMSes, Microsoft SQL Server offers both Serializable and Snapshot Isolation as isolation levels.

Note: The importance of monotonicity

As shown in the example in section 7 monotonicity is an important property for schedulers. We therefore find it strange that this topic is not covered in many (most?) popular textbooks discussing schedulers (among them [3, 4, 7, 10]). An exception is the book of Weikum & Vossen [11] where they in section 3.7 prove that neither FSR nor VSR are monotone. In our opinion this is the main reason why both final state and view equivalence are unfit as serializability criteria. Not being monotone is much more important than the fact that their schedulers have to be based on NP-complete algorithms.

Future Work

While the order of steps within a transaction (Definition 1) is linear the order of steps in a history (Definition 2) is partial as only conflicting steps are ordered. However, the definition of a snapshot requires that we can compare all steps to all commits. This works fine as long as all transactions run on the same computer using the same system clock. But in a distributed system with several autonomous nodes, it is not obvious how to define a snapshot. This question is of course closely related to the notion of global time, but we feel that there are still some open questions related to 'Distributed Snapshot Isolation'.

10. ACKNOWLEDGEMENTS

We want to thank the anonymous referees for their usefull comments.

11. REFERENCES

[1] American National Standards Institute. *ANSI X3. 135-1992 Information Systems – Database Language – SQL*, 1992.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[3] T. Connolly and C. Begg. *Database Systems. 5th Edition.* Addison Wesley, 2010.

[4] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems. 5th Edition.* Benjamin/Cummings, 2006.

[5] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making Snapshot Isolation Serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[6] A. Fekete, E. O'Neil, and P. O'Neil. A Read-Only Transaction Anomaly Under Snapshot Isolation. *SIGMOD Record*, 33(3):12–14, September 2004.

[7] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book. Second Edition.* Prentice-Hall, 2009.

[8] International Organization for Standardization. *ANSI/ISO SQL 99. (ISO/IEC9075-2:1999(E), page 83)*, 1999.

[9] J. Melton and A. R. Simon. *Understanding The New SQL: A Complete Guide.* Morgan Kaufmann, 1993.

[10] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts. 5th Edition.* McGraw-Hill, 2005.

[11] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[12] M. Yannakakis. Serializability by Locking. *Journal of the ACM*, 31(2):227–244, April 1984.

[13] L. T. Østby. En teoretisk studie av "Snapshot Isolation". Master's thesis, Dept. of Informatics, University of Oslo, 2008. (In Norwegian).