# A Read-Only Transaction Anomaly Under Snapshot Isolation

**By Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil**

**fekete@it.usyd.edu.au, {eoneil, poneil}@cs.umb.edu**

Research of all authors was supported by NSF Grant IRI 97-11374.

## Abstract

Snapshot Isolation (SI), is a multi-version concurrency control algorithm introduced in [BBGMOO95] and later implemented by Oracle. SI avoids many concurrency errors, and it never delays read-only transactions. However it does not guarantee serializability. It has been widely assumed that, under SI, read-only transactions always execute serializably provided the concurrent update transactions are serializable. The reason for this is that all SI reads return values from a single instant of time when all committed transactions have completed their writes and no writes of non-committed transactions are visible. This seems to imply that read-only transactions will not read anomalous results so long as the update transactions with which they execute do not write such results. In the current note, however, we exhibit an example contradicting these assumptions: it is possible for an SI history to be non-serializable while the sub-history containing all update transactions is serializable.

## 1. Definition of Snapshot Isolation

In what follows, we assume *time* is measured by a counter that advances whenever any transaction starts or commits, and we designate the time when transaction $T_i$ starts as *start($T_i$)* and the time when $T_i$ commits as *commit($T_i$)*.

**Definition 1.1: Snapshot Isolation (SI)**. A transaction $T_i$ executing under SI conceptually reads data from the committed state of the database as of time start($T_i$) (the *snapshot*), and holds the results of its own writes in local memory store, so if it reads data it has written it will read its own output. Predicates evaluated by $T_i$ are also based on rows and index entry versions from the committed state of the database at time start($T_i$), adjusted to take $T_i$'s own writes into account. Snapshot Isolation also must obey a "First Committer (Updater) Wins" rule, explained below.

The interval in time from the start to the commit of a transaction, represented [Start($T_i$), Commit($T_i$)], is called its *transactional lifetime*. We say two transactions $T_1$ and $T_2$ are *concurrent* if their transactional lifetimes overlap, i.e., [start($T_1$), commit($T_1$)] $\cap$ [start($T_2$), commit($T_2$)] $\neq \emptyset$. Writes by transactions active after $T_i$ starts, i.e., writes by *concurrent transactions*, are not visible to $T_i$. When $T_i$ is ready to commit, it obeys the *First Committer Wins* rule, as follows: $T_i$ will successfully commit if and only if no concurrent transaction $T_k$ has already committed writes (updates) of rows or index entries that $T_i$ intends to write. See also the discussion of the variant *First Updater Wins* rule below. The First Committer Wins rule is reminiscent of certification in optimistic concurrency control, but only items written by $T_i$ are checked for concurrent modification, not items read.

In the Oracle implementation of Snapshot Isolation (referred to as the *SERIALIZABLE* Isolation Level in Oracle [JAC95]), an attempt by $T_i$ to read a row that has changed since start($T_i$) will cause the system to read an appropriate older version in the rollback segment. Indexes are also accessed in the appropriate snapshot state, so that predicate evaluation retrieves row versions current as of the snapshot. The First Committer Wins rule is enforced, not by a commit-time validation, but by checks done at the time of updating. If $T_i$ and $T_k$ are concurrent, and $T_i$ updates the data item X, then it will take a Write lock on X; if $T_k$ subsequently attempts to update X while $T_i$ is still active, $T_k$ will be prevented by the lock on X from making further progress. If $T_i$ then commits, $T_k$ will abort; $T_k$ will only be able to continue if $T_i$ drops its lock on X by aborting. If, on the other hand, $T_i$ and $T_k$ are concurrent, and $T_i$ updates X but then commits before $T_k$ attempts to update X, there will be no delay due to locking, but $T_k$ will abort immediately when it attempts to update X (the abort does not wait until $T_k$ attempts to commit). For Oracle we rename the *First Committer Wins* rule to *First Updater Wins*; the ultimate effect is the same – one of the concurrent transactions updating a data item aborts. Aborts by a transaction for this reason are known as serialization errors, ORA-08177 (Oracle Release 9.2).

Snapshot Isolation is an attractive isolation level. Reading from a snapshot means that a transaction never sees the partial results of other transactions: T sees all the changes made by transactions that commit before start(T), and it sees no changes made by transactions that commit after start(T). Also, the First Committer Wins rule allows SI to avoid the most common type of lost update error, as shown in Example 1.1.

## 2. Anomaly Behavior in SI

Most of the anomalies that occur in lower Locking Isolation Levels such as READ COMMITTED are absent in SI. Example 1.1 gives an example.

**Example 1.1. Lost Update.** If transaction $T_1$ tries to modify a data item X while a concurrent $T_2$ also tries to modify X, then SI's First Committer Wins rule will cause one of the transactions to abort, so the first update will not be lost. E.g., in example history H1 below, we display the values read and written in a versioned notation we use to specify SI histories; when $T_i$ writes a version of X, the version is named $X_i$.

H1: $R_1(X0,50)$ $R_2(X0,50)$ $W_2(X2,70)$ $C_2$ $W_1(X1,60)$ $A_1$

This history leaves X with the value 70 (version X2), since only $T_2$, attempting to add an increment of 20 to X, was able to complete. $T_1$ can now retry and hopefully add its increment of 10 to X without interference. Note that many database system products with locking-based concurrency default to the READ COMMITTED isolation level, which takes long-term write-locks but no long-term read locks (it only tests reads to make sure they do not read write-locked data); in that case, the history above without the versioned data items would succeed in both its writes, causing a Lost Update. •

Despite its attractions, SI does not ensure that all executed histories are serializable, as defined in classical transactional theory (e.g., in [BHG87, PAPA86, GR93]). Indeed it is possible for a set of transactions, each of which in isolation respects an integrity constraint, to execute under SI in such a way as to leave the database in a corrupted state. One such problem is called "Write Skew".

**Example 1.2. Write Skew**. Suppose X and Y are data items in different rows representing checking account balances of a married couple at a bank, with a constraint that $X+Y > 0$ (the bank permits either account to be overdrawn, as long as the sum of the account balances remains positive). Assume that initially $X0 = 70$ and $Y0 = 80$. Under SI, transaction $T_1$ reads X0 and Y0, then subtracts 100 from X, assuming it is safe because the two data items added up to 150. Transaction $T_2$ concurrently reads X0 and Y0, then subtracts 100 from Y, assuming it is safe for the same reason. Each update is safe by itself, but SI will result in the following history:

H2: $R_1(X0,70)$ $R_2(X0,70)$ $R_1(Y0,80)$ $R_2(Y0,80)$ $W_1(X1,-30)$ $C_1$ $W_2(Y2,-20)$ $C_2$

Here the final committed state (X1 and Y2) violates the constraint $X+Y > 0$. This problem was not detected by First Committer Wins because two different data items were updated, each under the assumption that the other remained stable. Hence the name "Write Skew". •

While Example 1.2 displays one of a number of anomalous situations that can arise in SI, the occurrence of such situations is actually rather rare in real-world applications. The TPC-C benchmark application [TPC-C], consisting of seven transactional programs, displays no such anomalies, and it is reasonably representative. We also point out that it is quite easy to modify application or database design to avoid the anomaly of Example 1.2. One way is to require in the transactional program that each Read of X and Y to update Y give the impression of a Write of X (this is possible in Oracle using the Select For Update statement). Now it seems that both X and Y are updated in H2 and collision will occur. Another approach requires that each constraint on the sum of two accounts X and Y be materialized in another row Z and insist that all updates of X and Y must keep Z up to date. Then the anomaly of history H will not arise, since collision on updates of Z will occur whenever X and Y are updated by two different transactions.

## 3. A Read-Only Transaction Anomaly in SI

Another type of anomaly can occur resulting from read-only transaction participation. As we explained in the Abstract, this is surprising. Starting with [BBGMOO95], it was assumed that read-only transactions always execute serializably, without ever needing to wait or abort because of concurrent update transactions. This seemed self-evident because all reads take place at an instant of time, when all committed transactions have completed their writes and no writes of non-committed transactions are visible. The implied guarantee is that read-only transactions will not read anomalous results so long as the update transactions do not write such results. But Example 1.3 shows this isn't true.

**Example 1.3. Read-Only Transaction Anomaly**. Suppose X and Y are data items in different rows representing a checking account balance and a savings account balance, and that initially $X0 = 0$ and $Y0 = 0$. In history H3 below, transaction $T_1$ deposits 20 to the savings account Y, $T_2$ subtracts 10 from the checking account X, considering the withdrawal covered as long as $X+Y > 0$, but accepting an overdraft with a penalty charge of 1 if X+Y goes negative; finally, $T_3$ is a read-only transaction that retrieves the values of X and Y and prints them out for the customer. For one sequence of operations, this can result in the following history under SI:

H3: $R_2(X0,0)$ $R_2(Y0,0)$ $R_1(Y0,0)$ $W_1(Y1,20)$ $C_1$ $R_3(X0,0)$ $R_3(Y1,20)$ $C_3$ $W_2(X2,-11)$ $C_2$

The anomaly that arises in this transaction is that read-only transaction $T_3$ prints out X = 0 and Y = 20, while final values are Y = 20 and X = -11. This can't happen in any serializable execution since if 20 was added to Y before 10 was subtracted from X, no charge of 1 would ever occur, and the final balance should be 10, not 9. A customer, knowing a deposit of 20 was due and worried that his check for 10 might have caused a penalty, would conclude he was safe based on the data read by $T_3$. Indeed, such a print-out by $T_3$ would be embarrassing for a bank should the SEC ask how the charge occurred. We also note that any execution of $T_1$ and $T_2$ (with arbitrary parameter values) without $T_3$ present will always act serializably. •

**Intuitive Explanation of Example 1.3.** In H3, $T_2$ reads X0 = 0 and Y0 = 0, then writes X2 = -11, while $T_1$ concurrently updates Y1 to hold 20, which would change the behavior of $T_2$ if $T_2$ started after $T_1$ committed. The only equivalent serial history ending with X = -11 and Y = 20 must have $T_2$ followed by $T_1$. But since concurrent transactions don't see each other's results in SI, $T_1$ can commit first (out of serial order). Now the read-only transaction $T_3$ can see the committed state of $T_1$ only: Y = 20, and X = 0, and conclude that the deposit came in before the charge, implying there would be no penalty charge when $T_2$ executes. The fact that SI allows commit order different than serial order is what causes the anomaly.

## 4. Conclusion

There is great practical value in understanding the properties of database histories under weak isolation (that is, with concurrency control that does not automatically guarantee serializable behavior). It is especially important to understand what can and can't happen when running under SI, in view of the wide commercial penetration of the Oracle DBMS, which implements SI. Several papers have identified sufficient conditions that can be checked for application programs to guarantee serializable behavior for those applications [F99, BLL00]. We are in the process of attempting to develop a broad theory which covers the anomalies of Example 1.2 and 1.3, along with numerous others. We hope our theory will guide the application designer to adjust the programs without changing their functionality, so that serializability is guaranteed.

## References

[BBGMOO95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. Proc. of the ACM SIGMOD International Conference on Management of Data, 1995. Pages 1-10.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison Wesley, 1987. (This text is now out of print but can be downloaded from http://research.microsoft.com/pubs/ccontrol/default.htm)

[BLL00] A. Bernstein, P. Lewis and S. Lu. Semantic Conditions for Correctness at Different Isolation Levels. In Proceedings of IEEE International Conference on Data Engineering, 2000. Pages 57-66.

[F99] A. Fekete. Serializability and Snapshot Isolation. Proceedings of the Australian Database Conference, Auckland, New Zealand, January 1999. Pages 201-210.

[GR93] J. N. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., 1993.

[JAC95] K. Jacobs, with contributors: R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, B. Quigley. Concurrency Control: Transaction Isolation and Serializability in SQL92 and Oracle7. Oracle White Paper, Part No. A33745, July, 1995.

[PAPA86] C. Papadimitriou. The Theory of Database Concurrency Control. Computer Science Press, 1986.

[TPC-C] TPC-C Benchmark Specification, available at http://www.tpc.org/