Serializable Isolation for Snapshot Databases

Michael J. Cahill[*]
mjc@it.usyd.edu.au

Uwe Röhm
roehm@it.usyd.edu.au

Alan D. Fekete
fekete@it.usyd.edu.au

School of Information Technologies
University of Sydney
NSW 2006 Australia

ABSTRACT

Many popular database management systems offer snapshot isolation rather than full serializability. There are well-known anomalies permitted by snapshot isolation that can lead to violations of data consistency by interleaving transactions that individually maintain consistency. Until now, the only way to prevent these anomalies was to modify the applications by introducing artificial locking or update conflicts, following careful analysis of conflicts between all pairs of transactions.

This paper describes a modification to the concurrency control algorithm of a database management system that automatically detects and prevents snapshot isolation anomalies at runtime for arbitrary applications, thus providing serializable isolation. The new algorithm preserves the properties that make snapshot isolation attractive, including that readers do not block writers and vice versa. An implementation and performance study of the algorithm are described, showing that the throughput approaches that of snapshot isolation in most cases.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Reliability

Keywords

Multiversion Concurrency Control, Serializability Theory, Snapshot Isolation

[*]The author is also an employee of Oracle Corporation. This work was done while at the University of Sydney.

©ACM, (2008). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution.
The definitive version was published in the Proceedings of the 2008 ACM SIGMOD international conference on management of data, pages 729-738
http://doi.acm.org/10.1145/1376616.1376690

1. INTRODUCTION

Serializability is an important property when transactions execute because it ensures that integrity constraints are maintained even if those constraints are not explicitly declared to the DBMS. If a DBMS enforces that all executions are serializable, then developers do not need to worry that inconsistencies in the data might appear as artifacts of concurrency or failure. It is well-known how to use strict two-phase locking (and various enhancements such as escrow locking and multigranularity locking) to control concurrency so that serializable executions are produced [11]. Some other concurrency control algorithms are known that ensure serializable execution, but these have not been adopted in practice, because they usually perform worse than a well-engineered implementation of strict two-phase locking (S2PL).

Snapshot isolation (SI) [3] is an alternative approach to concurrency control, taking advantage of multiple versions of each data item. In SI, a transaction T sees the database state as produced by all the transactions that committed before T starts, but no effects are seen from transactions that overlap with T. This means that SI never suffers from Inconsistent Reads. In a DBMS using SI for concurrency control, reads are never delayed because of concurrent transactions' writes, nor do reads cause delays in a writing transaction. In order to prevent Lost Update anomalies, SI does abort a transaction T when a concurrent transaction commits a modification to an item that T wishes to update. This is called the "First-Committer-Wins" rule.

Despite the nice properties of SI, it has been known since SI was formalized in [3] that SI allows non-serializable executions. In particular, it is possible for an SI-based concurrency control to interleave some transactions, where each transaction preserves an integrity constraint when run alone, but where the final state after the interleaved execution does not satisfy the constraint. This occurs when concurrent transactions modify different items that are related by a constraint, and it is called the Write Skew anomaly.

Example 1: Suppose that a table Duties(DoctorId, Shift, Status) represents the status ("on duty" or "reserve") for each doctor during each work shift. An undeclared invariant is that, in every shift, there must be at least one doctor on duty. A parametrized application program that changes a doctor D on shift S to "reserve" status, can be written as in Figure 1.

This program is consistent, that is, it takes the database from a state where the integrity constraint holds to another state where the integrity constraint holds. However, suppose there are exactly two doctors $D1$ and $D2$ who are on

```
BEGIN TRANSACTION

UPDATE Duties SET Status = 'reserve'
 WHERE DoctorId = :D
   AND Shift = :S
   AND Status = 'on duty'

SELECT COUNT(DISTINCT DoctorId) INTO tmp
  FROM Duties
 WHERE Shift = :S
   AND Status = 'on duty'

IF (tmp = 0) THEN ROLLBACK ELSE COMMIT
```

Figure 1: Example parameterized application exhibiting write skew

duty in shift S. If we run two concurrent transactions, which run this program for parameters $(D1, S)$ and $(D2, S)$ respectively, we see that using SI as concurrency control will allow both to commit (as each will see the other doctor's status for shift S as still unchanged, at "on duty"). However, the final database state has no doctor on duty in shift S, violating the integrity constraint.

Despite the possibility of corrupting the state of the database, SI has become popular with DBMS vendors. It often gives much higher throughput than strict two-phase locking, especially in read-heavy workloads, and it also provides users with transaction semantics that are easy to understand. Many popular and commercially important database engines provide SI, and some in fact use SI when serializable isolation is requested [13].

Because SI allows data corruption, and is so common, there has been a body of work on how to ensure serializable executions when running with SI as concurrency control. The main techniques proposed so far [9, 8, 14] depend on doing a design-time static analysis of the application code, and then modifying the application if necessary in order to avoid the SI anomalies. For example, [9] shows how one can introduce write-write conflicts into the application, so that all executions will be serializable even on SI.

Making SI serializable using static analysis has a number of limitations. It relies on an education campaign so that application developers are aware of SI anomalies, and it is unable to cope with ad-hoc transactions. In addition, this must be a continual activity as an application evolves: the analysis requires the global graph of transaction conflicts, so every minor change in the application requires renewed analysis, and perhaps additional changes (even in programs that were not altered). In this paper, we instead focus on guaranteeing serializability for every execution of arbitrary transactions, while still having the attractive properties of SI, in particular much better performance than is allowed by strict two-phase locking.

1.1 Contributions

We propose a new concurrency control algorithm, called Serializable Snapshot Isolation, with the following innovative combination of properties:

- The concurrency control algorithm ensures that every execution is serializable, no matter what application programs run.
- The algorithm never delays a read operation; nor do readers cause delays in concurrent writes.
- Under a range of conditions, the overall throughput is close to that allowed by SI, and much better than that of strict two-phase locking.
- The algorithm is easily implemented by small modifications to a system that provides SI.

We have made a prototype implementation of the algorithm in the open source data management product Oracle Berkeley DB [16], and we evaluate the performance of this implementation compared to the product's implementations of Strict Two-Phase Locking (S2PL) and SI.

The key idea of our algorithm is to detect, at runtime, distinctive conflict patterns that must occur in every non-serializable execution under SI, and abort one of the transactions involved. This is similar to the way serialization graph testing works, however our algorithm does not operate purely as a certification at commit-time, but rather aborts transactions as soon as the problem is discovered; also, our test does not require any cycle-tracing in a graph, but can be performed by considering conflicts between pairs of transactions, and a small amount of information which is kept for each of them. Our algorithm is also similar to optimistic concurrency control [15] but differs in that it only aborts a transaction when a *pair* of consecutive conflict edges are found, which is characteristic of SI anomalies. This should lead to significantly fewer aborts than optimistic techniques that abort when any single conflict edge is detected. Our detection is conservative, so it does prevent every non-serializable execution, but it may sometimes abort transactions unnecessarily.

The remainder of the paper is structured as follows: in Section 2 we give an introduction to snapshot isolation and current approaches to ensuring serializable isolation with SI; in Section 3 we describe the new Serializable SI algorithm; in Section 4 we describe the implementation in Oracle Berkeley DB and in Section 5 we evaluate its performance. Section 6 concludes.

2. BACKGROUND

2.1 Snapshot Isolation

SI is a concurrency control approach that uses multiple versions of data to provide non-blocking reads. When a transaction T starts executing, it gets a conceptual timestamp start-time(T); whenever T reads a data item x, it does not necessarily see the latest value written to T; instead T sees the version of x which was produced by the last to commit among the transactions that committed before T started and also modified x (there is one exception to this: if T has itself modified x, it sees its own version). Thus, T appears to execute against a snapshot of the database, which contains the last committed version of each item at the time when T starts.

SI also enforces an additional restriction on execution, called the "First-Committer-Wins" rule: it is not possible to have two concurrent transactions which both commit and both modify the same data item. In practice, implementations of SI usually prevent a transaction from modifying an item if a concurrent transaction has already modified it.

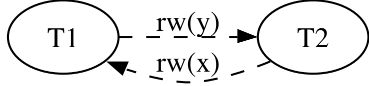


Figure 2: Serialization graph for transactions exhibiting write skew

SI was introduced in the research literature in [3], and it has been implemented by the Oracle RDBMS, PostgreSQL, SQL Server 2005, and Oracle Berkeley DB. It provides significant performance improvements over serializability implemented with two-phase locking (S2PL) and it avoids many of the well-known isolation anomalies such as Lost Update or Inconsistent Read. In some systems that do not implement S2PL, including the Oracle RDBMS and PostgreSQL, SI is provided when serializable isolation is requested.

2.2 Write Skew

As noted in [3], SI does not guarantee that all executions will be serializable, and it can allow corruption of the data through interleaving between concurrent transactions which individually preserve the consistency of the data. Here is an execution that can occur under SI:

$r_1(x{=}50,y{=}50)$ $r_2(x{=}50,y{=}50)$ $w_1(x{=}\text{-}20)$ $w_2(y{=}\text{-}30)$ c_1 c_2

This sequence of operations represents an interleaving of two transactions, T1 and T2, withdrawing money from bank accounts x and y, respectively. Each of the transactions begins when the accounts each contain $50, and each transaction in isolation maintains the constraint that $x + y > 0$. However, the interleaving results in $x + y = -50$, so consistency has been violated. This type of anomaly is called a *write skew*.

We can understand these situations using a multiversion serialization graph (MVSG). There are a number of definitions of this in the literature, because the general case is made complicated by uncertainty over the order of versions (which indeed renders it NP-Hard to check for serializability of a multiversion schedule). For example, there are definitions in [5, 12, 17, 1].

With snapshot isolation, the definitions of the serialization graph become much simpler, as versions of an item x are ordered according to the temporal sequence of the transactions that created those versions (note that First-Committer-Wins ensures that among two transactions that produce versions of x, one will commit before the other starts). In the MVSG, we put an edge from one committed transaction T1 to another committed transaction T2 in the following situations: T1 produces a version of x, and T2 produces a later version of x (this is a *ww*-dependency); T1 produces a version of x, and T2 reads this (or a later) version of x (this is a *wr*-dependency); T1 reads a version of x, and T2 produces a later version of x (this is a *rw*-dependency). In Figure 2 we show the MVSG for the history with write skew, discussed above. In drawing our MVSG, we will follow the notation introduced in [1], and use a dashed edge to indicate a *rw*-dependency.

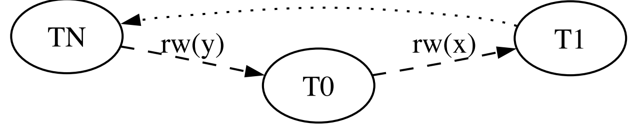


Figure 3: Generalized dangerous structure in the MVSG

As usual in transaction theory, the absence of a cycle in the MVSG proves that the history is serializable. Thus it becomes important to understand what sorts of MVSG can occur in histories of a system using SI for concurrency control. Adya [1] showed that any cycle produced by SI has two *rw*-dependency edges. This was extended by Fekete et al in [9], which showed that any cycle must have two *rw*-dependency edges that occur consecutively, and further, each of these edges is between two concurrent transactions.

We adopt some terminology from [9], and call an *rw*-dependency between concurrent transactions a *vulnerable edge*; we call the situation where two consecutive vulnerable edges occur in a cycle as a *dangerous structure*. It is illustrated in Fig 3. We refer to the transaction at the junction of the two consecutive vulnerable edges as a *pivot* transaction. The theory of [9] shows that there is a pivot in any non-serializable execution allowed by SI.

We take an interesting example from [10] to illustrate how a dangerous structure may occur at runtime. Consider the following three transactions:

T0: r(y) w(x)

T1: w(y) w(z)

TN: r(x) r(z)

These three transactions can interleave such that TN, a read-only transaction, sees a state that could never have existed in the database had T0 and T1 executed serially. If TN is omitted, T0 and T1 *are* serializable because there is only a single anti-dependency from T0 to T1.

Two of the possible non-serializable interleavings of these three transactions are illustrated in Figure 4. These diagrams should be read from left to right; the arrows indicate the *rw*-dependencies between transactions. In Figure 4(a), both reads occur after the writes. In Figure 4(b), TN reads x before it is written by T0.

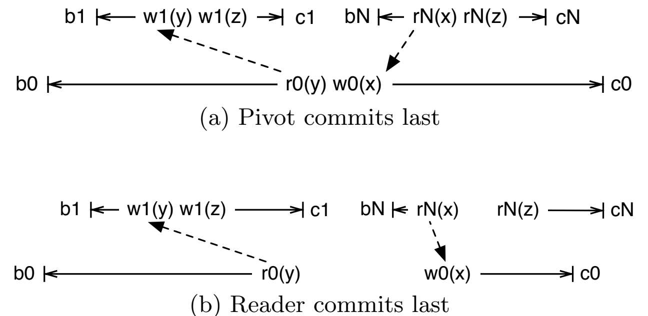


Figure 4: SI anomalies at runtime

Notice that there are no constraints on the commit order: when an anomaly occurs under SI, the transactions can go on to commit in any order. This observation is one of the challenges that an algorithm to detect SI anomalies at run-time must overcome: we cannot always know when a transaction commits whether it will have consecutive vulnerable edges.

2.3 Phantoms

Throughout the discussion so far, we have followed typical concurrency control theory and assumed that a transaction is a sequence of reads and writes on named data items. In general, a relational database engine must also deal with predicate operations (such as SQL "where" clauses). These mean that a concurrency control algorithm must also consider phantoms, where an item created or deleted in one transaction would change the result of a predicate operation in a concurrent transaction if the two transactions executed serially. The solution used in traditional two-phase locking is multigranularity locking [6], where a predicate operation takes intention locks on larger units, such as pages or tables, to prevent insertion of records that might match the predicate.

2.4 Related Work

An extensive line of research has considered, not the serializability of particular executions allowed by SI, but rather the question of whether a given set of application programs is guaranteed to generate serializable executions when run on a system with SI as the concurrency control mechanism. This problem was addressed in [7] and the techniques were refined in [9]. The key to this work is to consider a static analysis of the possible conflicts between application programs. Thus a static dependency graph, or SDG, is drawn, with a edge from program P1 to P2, if there can be an execution where P1 generates a transaction T1, P2 generates T2, and there is a dependency edge from T1 to T2. It was shown how a dangerous structure in the MVSG can be related to a similar structure in the SDG, and this justified the intuition of experts, who had previously decided that every execution of the TPC-C benchmark [20] is serializable on a platform using SI. As well as showing how to prove that certain programs generate only serializable executions, [9] proposed that one could modify transaction programs so that they fall into this class. The modifications typically involve introducing extra write-write conflicts, between transaction programs that might give rise to transactions joined by a vulnerable edge within a dangerous structure.

The theory of [9] was extended in [8] to the case where some transactions use SI and others use S2PL (as is possible with Microsoft SQL Server 2005 or Oracle Berkeley DB). Performance studies [2] indicate that modifying applications to ensure serializability under SI can be done without significant cost when the appropriate technique is used.

In [14], a system is described to automate the analysis of program conflicts using syntactic features of program texts, such as the names of the columns accessed in each statement. One important finding of their work is that snapshot isolation anomalies do exist in applications developed using tools and techniques that are common throughout the software industry.

An alternative approach to ensuring correctness when running on platforms with SI is in [4], where conditions are given to ensure all executions preserve given integrity constraints, without necessarily being serializable.

Others have previously suggested ways to alter the SI concurrency control in order to avoid non-serializable executions at run-time. Proposals related to certification through serialization graph testing are in [18] and [21]. These suggestions have not focused on feasibility of implementation within a DBMS. In particular, the space required to represent complete conflict graphs and the overhead required to maintain them may be prohibitive.

3. SERIALIZABLE SNAPSHOT ISOLATION

The essence of our new concurrency control algorithm is to allow standard SI to operate, but to add some book-keeping so we can dynamically detect cases where a non-serializable execution could occur, and then we abort one of the transactions involved. This makes the detection process a delicate balance: if we detect too few cases, some non-serializable execution may emerge (counter to our goal of having true serializability guarantees for the applications), but if we detect too many cases, then performance might suffer as unnecessary aborts waste resources. As a third factor in designing an algorithm, we also need to keep the overhead cost of detection low. One can imagine a concurrency control algorithm which aborts a transaction exactly when an operation will result in a non-serializable execution; this would be a serialization-graph-testing algorithm (using the appropriate multiversion serialization graph). Serialization-graph-testing however requires expensive cycle detection calculations on each operation, and would be very expensive. Thus we accept a small chance of unnecessary aborts, in order to keep the detection overhead low.

The key design decision in our new algorithm is thus the situations in which potential anomalies are detected. We do this based on the theory of [1] and its extension from [9], where some distinctive conflict patterns are shown to appear in every non-serializable execution of SI. The building block for this theory is the notion of a *rw*-dependency (also called an "anti-dependency"), which occurs from T1 to T2 if T1 reads a version of an item x, and T2 produces a version of x that is later in the version order than the version read by T1. In [1] it was shown that in any non-serializable SI execution, there are two *rw*-dependency edges in a cycle in the multiversion serialization graph. [9] extended this, to show that there were two *rw*-dependency edges which form consecutive edges in a cycle, and furthermore, each of these *rw*-edges involves two transactions that are active concurrently.

Our proposed serializable SI concurrency control algorithm detects a potential non-serializable execution whenever it finds two consecutive *rw*-dependency edges in the serialization graph, where each of the edges involves two transactions that are active concurrently. Whenever such a situation is detected, one of the transactions will be aborted. To support this algorithm, the DBMS maintains, for each transaction, two boolean flags: `T.inConflict` indicates whether there is an *rw*-dependency from another concurrent transaction to T, and `T.outConflict` indicates whether there is an *rw*-dependency from T to another concurrent transaction. Thus a potential non-serializability is detected when `T.inConflict` and `T.outConflict` are both true.

We note that our algorithm is conservative: if a non-serializable execution occurs, there will be a transaction with

T.inConflict and T.outConflict. However, we do sometimes make false positive detections; for example, an unnecessary detection may happen because we do not check whether the two *rw*-dependency edges occur within a cycle. It is also worth mentioning that we do not always abort the particular pivot transaction T for which T.inConflict and T.outConflict is true; this is often chosen as the victim, but sometimes the victim is the transaction that has an *rw*-dependency edge to T, or the one that is reached by an edge from T.

How can we keep track of situations where there is an *rw*-dependency between two concurrent transactions? There are two different ways in which we notice such a dependency. One situation arises when a transaction T reads a version of an item x, and the version that it reads (the one which was valid at T's start time) is not the most recent version of x. In this case the writer U of any more recent version of x was active after T started, and so there is a *rw*-dependency from T to U. When we see this, we set the flags T.outConflict and U.inConflict (and we check for consecutive edges and abort a transaction if needed). This allows us to find *rw*-dependency edges for which the read occurs in real-time after the write that is logically later. However, it does not account for edges where the read occurs first, and at a later real-time, a version is created by a concurrent transaction.

To notice these other *rw*-dependency cases, we use a lock management infrastructure. A normal WRITE lock is taken when a new version is created; note that many SI implementations do keep such write-locks anyway, as a way to enforce the First-Committer-Wins rule. We also introduce a new lock mode called SIREAD. This remembers the fact that an SI transaction has read a version of an item. However, obtaining the SIREAD lock does not cause any blocking, even if a WRITE lock is held already, and similarly an existing SIREAD lock does not delay granting of a WRITE lock; instead, the presence of both SIREAD and WRITE locks on an item is a sign of an *rw*-dependency, and so we set the appropriate inConflict and outConflict flags on the transactions which hold the locks. One difficulty, which we discuss later, is that we need to keep the SIREAD locks that T obtained, even after T is completed, until all transactions concurrent with T have completed.

3.1 The Algorithm

We now present in pseudocode the Serializable SI concurrency control algorithm.

The main data structure needed by the algorithm is two boolean flags in each transaction record: T.inConflict indicates whether or not there is a *rw*-dependency from a concurrent transaction to T, and T.outConflict indicates whether there is a *rw*-dependency from T to a concurrent transaction. As well, we need a lock manager that keeps both standard WRITE locks, and also special SIREAD locks.

In describing the algorithm, we make some simplifying assumptions:

1. For any data item x, we can efficiently get the list of locks held on x.
2. For any lock l in the system, we can efficiently get l.owner, the transaction object that requested the lock.
3. For any version xt of a data item in the system, we can efficiently get xt.creator, the transaction object that created that version.

```
modified begin(T):

    existing SI code for begin(T)
    set T.inConflict = T.outConflict = false
```

Figure 5: modified begin(T)

```
modified read(T, x):

    get lock(key=x, owner=T, mode=SIREAD)
    if there is a WRITE lock(wl) on x
        set wl.owner.inConflict = true
        set T.outConflict = true

    existing SI code for read(T, x)

    for each version (xNew) of x
    that is newer than what T read:
        if xNew.creator is committed
          and xNew.creator.outConflict:
            abort(T)
            return UNSAFE_ERROR
        set xNew.creator.inConflict = true
        set T.outConflict = true
```

Figure 6: modified read(T, x)

```
modified write(T, x, xNew):

    get lock(key=x, locker=T, mode=WRITE)

    if there is a SIREAD lock(rl) on x
      with rl.owner is running
      or commit(rl.owner) > begin(T):
        if rl.owner is committed
          and rl.owner.inConflict:
            abort(T)
            return UNSAFE_ERROR
        set rl.owner.outConflict = true
        set T.inConflict = true

    existing SI code for write(T, x, xNew)
    # do not get WRITE lock again
```

Figure 7: modified write(T, x, xNew)

```
modified commit(T):

    if T.inConflict and T.outConflict:
        abort(T)
        return UNSAFE_ERROR

    existing SI code for commit(T)
    # release WRITE locks held by T
    # but do not release SIREAD locks
```

Figure 8: modified commit(T)

4. When finding a version of item `x` valid at some given timestamp, we can efficiently get the list of other versions of `x` that have later timestamps.

These assumptions are true for the initial target system (Berkeley DB). We discuss in section 4.1 how to implement the algorithm if these assumptions do not hold.

The concurrency control layer processes each operation as shown in Figures 5 to 8. In each case, the processing includes the usual processing of the operation by the SI protocol as well as some extra steps. For simplicity, in this description we do not show all the cases where we could check whether to abort T because both `T.inConflict` and `T.outConflict` hold; we have written the check once, in the `commit(T)` operation, and beyond that we only show the extra cases where an abort is done for a transaction that is not the pivot (because the pivot has already committed). In the implementation, we actually abort an active transaction T as soon as any operation of T discovers that both `T.inConflict` and `T.outConflict` are true. Likewise, conflicts are not recorded against transactions that have already aborted or that will abort due to both flags being set.

When a conflict between two transactions leads to both conflict flags being set on either one, without loss of correctness either transaction could be aborted in order to break the cycle and ensure serializability. Our prototype implementation follows the algorithm as described above, and prefers to abort the pivot (the transaction with both incoming and outgoing edges) unless the pivot has already committed. If a cycle contains two pivots, whichever is detected first will be aborted. However, for some workloads, it may be preferable to apply some other policy to the selection of which transaction to abort, analogous to deadlock detection policies. For example, aborting the younger of the two transactions may increase the proportion of complex transactions running to completion. We intend to explore this idea in future work.

For the Serializable SI algorithm, it is important that the engine have access to information about transaction T (its transaction record, including inConflict and outConflict, as well as any SIREAD locks it obtained) even after T has completed. This information must be kept as long as any transaction U is active which overlaps T, that is, we can only remove information about T after the end of every transaction that had already started when T completed. In Section 4 we describe how this information is managed in the Berkeley DB implementation – in particular, how the space allocated to transaction objects is reclaimed.

3.2 Correctness

The Serializable SI algorithm ensures that every execution is serializable, and thus that data integrity is preserved (under the assumption that each transaction individually is coded to maintain integrity). This subsection gives the outline of the argument that this is so. By Theorem 2.1 from [9], which shows that in any non-serializable execution there is a dangerous structure, we are done provided that we can establish the following: whenever an execution contains a dangerous structure (transactions TN, a pivot T0, and T1, such that there is a *rw*-dependency from TN to T0 and TN is concurrent with T0, and also there is a *rw*-dependency from T0 to T1 and T0 is concurrent with T1), then one of the transactions is aborted. In this situation, we must consider the possibility that TN=T1, which is the classic example of Write Skew.

Our algorithm has an invariant, that whenever the execution has a *rw*-dependency from T to U, and the transaction record for both T and U exists, then `T.outConflict` and `U.inConflict` are both set to true. By definition, the *rw*-dependency comes from the existence of a read by T that sees some version of x, and a write by U which creates a version of `x` that is later in the version order than the version read by T.

One of these operations (`read(T, x)` and `write(U, x)`) will happen first because the database engine will perform some latching during their execution, and the other will happen later. The *rw*-dependency is present in the execution once the second of these operations occurs. If this second operation is `read(T, x)`, then at the time that operation is processed, there will already be the version of `x` created by U; the pseudocode in Figure 6 shows that we explicitly set both flags as required. On the other hand, if the `write(U, x)` occurs after `read(T, x)`, then at that time T will hold a SIREAD lock on x, and the pseudocode in Figure 7 shows that both flags are set.

Based on the invariant just described, we now must argue that one of the transactions in any dangerous structure is aborted. If both *rw*-dependencies exist at the time the pivot T0 completes, then the code in Figure 8 will notice that `T.inConflict` and `T.outConflict` are set (because of the invariant), and so T will be aborted when it requests to commit. If, however, one or both *rw*-dependencies appears after the pivot has committed, then we look at the first event in which both dependencies are true; in the pseudocode for this event, the flag for the other dependency will already be set in T2's transaction record, and so the transaction executing this event will be aborted.

In summary, the argument for correctness is as follows:

1. Non-serializable executions under SI consist of a cycle including two consecutive *rw*-dependencies.

2. Our algorithm detects every *rw*-dependency.

3. When two consecutive *rw*-dependencies are detected, at least one transaction is aborted which breaks the cycle.

The exhaustive testing of the implementation that we describe below in Section 4.2 further supports this argument for the algorithm's correctness.

3.3 False positives

Our algorithm uses a conservative approximation to cycle detection in the graph of transaction conflicts, and as such may cause some benign transactions to abort.

In particular, the interleaving of transactions in Figure 9 will set `outConflict` on T0 when it executes w0(x) and finds the SIREAD lock from TN. Then `inConflict` will be set on T0 when T1 executes w1(y) and finds T0's SIREAD lock. During the commit of T0, the two flags will be checked and since both are set, T0 will abort. However, this interleaving

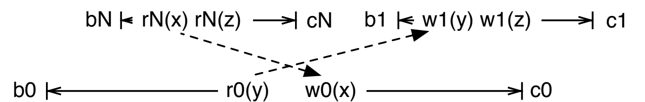


Figure 9: False positive: no path from T1 to TN

is equivalent to the serial history {TN, T0, T1} because TN precedes T1 and hence there is no path of dependencies from T1 to TN.

The issue is that the two flags we have added to each transaction cannot indicate anything about the relative ordering of the incoming and outgoing conflicts. It may be possible to reduce the number of false positives by keeping a reference to the conflicting transactions for each edge rather than a single bit, but in general a transaction may have multiple incoming and outgoing edges and it is not clear whether the overhead of maintaining a more complex data structure would justify the reduction in false positives.

3.4 Detecting phantoms

As described in section 2.3, a concurrency control algorithm for a relational DBMS must also consider phantoms, where an item is created in one transaction, and is incorrectly not seen in a predicate operation in a concurrent transaction. The solution used in traditional two-phase locking is multigranularity locking [6], where a predicate operation takes intention locks on larger units, such as pages or tables, to prevent insertion of records that might match the predicate.

To prevent phantoms in a system with row-level locking and versioning, the algorithm described here would need to be extended to take SIREAD locks on larger granules analogously to multigranularity intention locks in traditional two-phase locking systems. If a conflicting write operation occurs after the predicate read, this would detect the predicate-*rw*-conflict between transactions correctly.

In the case where a predicate read is interleaved *after* the conflicting write operation, such a system would find a data item with a creation or deletion timestamp greater than the read timestamp of the predicate read. In other words, a row will be skipped because no version of the row was visible when the transaction performing the predicate read began or a row used by the predicate read has since been deleted. Such rows can be used to detect this kind of conflict as described in Figure 6 without further modification.

We have not pursued the details in this paper because the phantom issue does not arise in our prototype implementation, since Oracle Berkeley DB does all locking and versioning at page granularity.

4. IMPLEMENTATION

The algorithm described above was implemented in Oracle Berkeley DB version 4.6.21 [16]. Berkeley DB is an embedded database that supports SI as well as serializable isolation with S2PL. Locking and multi-version concurrency control are performed at the granularity of database pages. This can introduce unnecessary conflicts between concurrent transactions, but means that straightforward read and write locking is sufficient to prevent phantoms.

We added the following to Berkeley DB:

1. New error returns `DB_SNAPSHOT_CONFLICT`, to distinguish between deadlocks and update conflicts, and `DB_SNAPSHOT_UNSAFE`, to indicate that committing a transaction at SI could lead to a non-serializable execution.

2. A new lock mode, `SIREAD` that does not conflict with any other lock modes. In particular, it does not introduce any blocking with conflicting `WRITE` locks. The code that previously avoiding locking for snapshot isolation reads was modified to instead get an `SIREAD` lock.

3. Code to clean old `SIREAD` locks from the lock table. This code finds the earliest read timestamp of all active transactions, and removes from the lock table any locks whose owning transactions have earlier commit timestamps.

   The cleanup code is executed if the lock table becomes full (a complete sweep of all lock objects), and also whenever a request for a `WRITE` lock finds an old `SIREAD` lock, in order to spread out the work of cleaning up old locks.

4. When a `WRITE` lock request for transaction T finds an `SIREAD` lock already held by T, the `SIREAD` lock is discarded before the `WRITE` lock is granted. This is done for two reasons: firstly, whenever possible we would prefer to return an error indicating an update conflict if that is the primary cause of the failure. Secondly, there is no need to hold those `SIREAD` locks after the transaction commits: the new version of the data item that T creates will cause an update conflict with concurrent writers.

Making these changes to Berkeley DB involved only modest changes to the source code. In total, only 692 lines of code (LOC) were modified out of a total of over 200,000 lines of code in Berkeley DB. Approximately 40% (276 LOC) of the changes related to detecting lock conflicts and a further 17% (119 LOC) related to cleaning obsolete locks from the lock table. Of the code comprising the locking subsystem of Berkeley DB, 3% of the existing code was modified and the total size increased by 10%.

4.1 Generalizing to other database engines

In Berkeley DB, it is reasonable for transaction objects and locks to remain in the system for some time after commit. In fact, transaction objects already have a reference count for the existing implementation of SI and transaction objects are deleted when the reference count goes to zero after the transaction commits.

If this were not the case, we would need to maintain a table containing the following information: for each transaction ID, the begin and commit timestamps together with an `inConflict` flag and an `outConflict` flag. For example:

txnID	beginTime	commitTime	inConf	outConf
100	1000	1100	N	Y
101	1000	1500	N	N
102	1200	N/A	Y	N

Rows can be removed from the table when the commit time of a transaction is earlier than the begin times of all active transactions. In this case, only transaction 102 is still running (since the commit timestamp is not set). The row for transaction 100 can be deleted, since it committed before the only running transaction, 102, began.

Likewise, a table can be constructed to track SIREAD locks, if the lock manager in the DBMS cannot easily be modified to keep locks for a transaction after it completes. Rows in the lock table become obsolete when the owning transaction becomes obsolete. That is, when all concurrent transactions have completed.

4.2 Testing

We have also done an exhaustive analysis of the implementation, by testing it with all possible interleavings of some set of transactions known to cause write skew anomalies. For example, one test set was as follows:

T1 : b1 r1(x) c1

T2 : b2 r2(y) w2(x) c2

T3 : b3 w3(y) c3

The implementation was tested by generating the test code, where each test case was a different interleaving of a given set of transactions. In all cases where the transactions were executed concurrently, one of the transactions aborted with the new "unsafe" error return. These results were manually checked to verify that no non-serializable executions were permitted although all interleavings committed without error at SI.

5. EVALUATION

To evaluate the effects of making SI serializable, we need a benchmark that is not already serializable under SI. The SmallBank benchmark [2] was designed to model a simple banking application involving checking and savings accounts, with transaction types for balance (Bal), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS) and amalgamate (Amg) operations. Each of the transaction types involves a small number of simple read and update operations. The static dependency graph for SmallBank is given in Figure 10, where the double arrows represent write-write conflicts and the dashed arrows represent read-write conflicts. It can be seen by inspection that there is a dangerous structure Balance → WriteCheck → TransactSavings → Balance, so the transaction WriteCheck is a pivot.

5.1 Evaluation Setup

Sullivan [19] designed and implemented a tool called `db_perf` that can execute and measure arbitrary workloads against Berkeley DB. Using `db_perf`, we simulated the access patterns of the SmallBank benchmark. We compared the built-in S2PL and SI isolation levels in Berkeley DB with our implementation of the new Serializable SI algorithm.

The five transaction types of SmallBank were executed use `db_perf` in a uniform random mix of the different transaction types. There was no sleep or think time: each thread executed transactions continually, as quickly as Berkeley DB could process them.

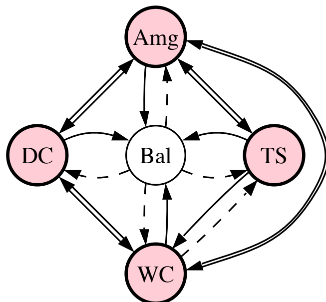
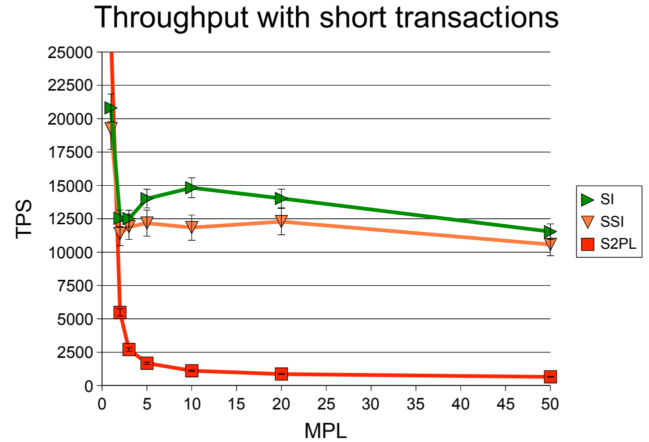
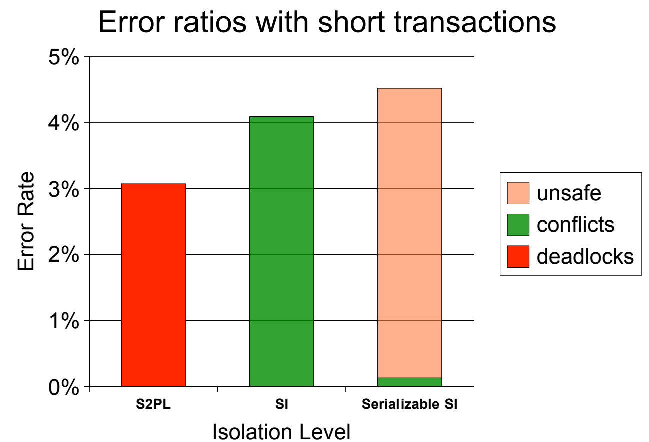


Figure 10: Static dependency graph for the SmallBank benchmark



(a) Relative throughput of SI, S2PL and Serializable SI without log flushes



(b) Relative error rates of SI, S2PL and Serializable SI at MPL 20 without log flushes

Figure 11: Results without flushing the log during commit

A single set of Berkeley DB binaries were used for all measurements, with parameters to control the isolation level, the number of threads (the multi-programming level, or MPL), the data volume, the duration of each run and various Berkeley DB parameters such as the page size and deadlock detection policy.

The experiments were run on an AMD Athlon64 3200+ CPU with 1GB RAM running openSUSE Linux 10.2 with kernel version 2.6.18, glibc version 2.5 and GCC version 4.1.2. All data was stored using the XFS filesystem on a set of four Western Digital Caviar SE 200 GB SATA hard disks using software RAID5. All graphs include 95% confidence intervals.

5.2 Performance with Short Transactions

Results are given in Figure 11 for measurements where system was configured so that commit operations do not wait for a physical disk write before completing. This configuration is common on high-end storage systems with redun-

dant power sources or in solid state drives based on Flash memory. A small data size was configured here to model moderate contention: the savings and checking tables both consisted of approximately 100 leaf pages. All data fitted in cache.

In this configuration, transaction durations are very short, with response times typically under 1ms. When there is no contention (such as in the MPL=1 case), the CPU was 100% busy throughout the tests.

It can be seen that in this configuration, Serializable SI performs significantly better that S2PL (by a factor of 10 at MPL 20). This is due to blocking in S2PL between read and write operations, and also because the conflicts in S2PL are found via deadlock detection, which introduces further delays.

Figure 11(b) shows that Serializable SI has a slightly higher total rate of aborts than either S2PL or SI at MPL 20. Interestingly, a high proportion of errors are reported as "unsafe" errors rather than update conflicts. This is because in this benchmark, several transactions execute a read operation followed by a write. In between those two operations, a *rw*-conflict can be detected leading to an unsafe error.

5.3 Performance with Long Transactions

Figure 12 shows results for the same set of experiments, changing only the commit operations to wait for a physical write to disk. This significantly increased the duration of transactions, increasing response times by at least an order of magnitude to 10-20ms. In this configuration, I/O rather than the CPU is the bottleneck at MPL 1, and throughput increases as MPL is increased for all isolation levels, because increasing numbers of transactions can be committed for each I/O operation due to group commit.

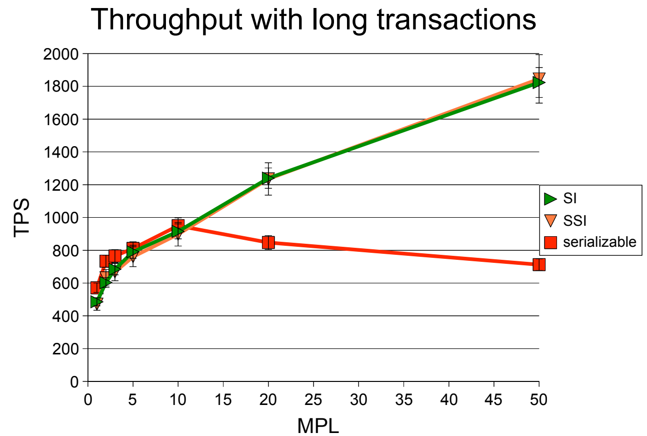
Up to MPL 5, there is little to separate the three concurrency control algorithms, but at MPL 10, the rate of deadlocks at S2PL begins to have an impact on its throughput.

The error rate at Serializable SI is significantly higher with longer duration transactions, due to increased number of *rw*-conflicts. However, since this test is primarily I/O bound and transactions are simple, the impact on throughput compared with SI is small.

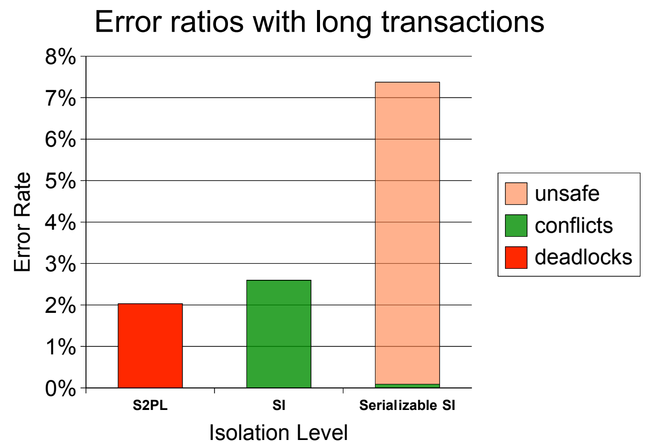
5.4 Overhead Evaluation

The results so far have concentrated on experiments with high contention in order to highlight the differences between the isolation levels. In Figure 13, we present results from running the benchmark including log flushes with a data set ten times larger. The increased volume of data results in far fewer conflicts between transactions but still fits in RAM so the only I/O operations are log writes. For S2PL, this produces less blocking and under SI and Serializable SI the rates of update conflicts are also lower than in the earlier experiments. In this case, the performance of S2PL and SI are almost identical and we can see the overhead in the Serializable SI implementation at between 10-15%.

This overhead is due in part to CPU overhead from managing the larger lock and transaction tables but primarily to a higher abort rate due to false positives with Serializable SI. One issue in particular contributed to the false positive rate. As mentioned earlier, Berkeley DB performs locking and versioning at page-level granularity. As a consequence, our Serializable SI implementation also detects *rw*-dependencies at page-level granularity. All of our exper-



(a) Relative throughput of SI, S2PL and Serializable SI with log flushes



(b) Relative error rates of SI, S2PL and Serializable SI at MPL 20 with log flushes

Figure 12: Results when the log is flushed during commit

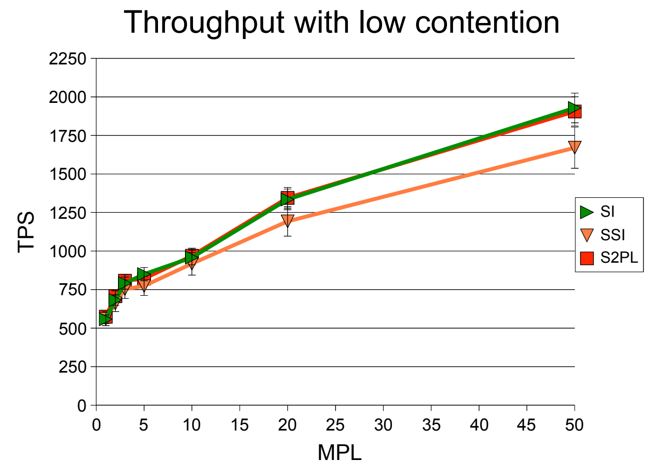


Figure 13: Throughput with low contention

iments use Berkeley DB's Btree access method, so whenever any transaction needs to update the Btree root page (for example, as a result of a page split), it will register a conflict with every concurrent transaction, as all transaction types need to read the root page. With standard SI, concurrent transactions simply read the older version of the root page, and with S2PL, concurrent transactions are blocked temporarily but not aborted unless a deadlock occurs. These conflicts and the resulting higher abort rate would not occur in a record-level implementation of Serializable SI.

6. CONCLUSIONS AND FUTURE WORK

This paper presents a new method for implementing serializable isolation based on a modification of snapshot isolation. A prototype of the algorithm has been implemented in Oracle Berkeley DB and shown to perform significantly better that two-phase locking in a variety of cases, and often comparably with snapshot isolation.

One property of Berkeley DB that simplified our implementation was working with page level locking and versioning. In databases that version and lock at row-level granularity (or finer), additional effort would be required to avoid phantoms, analogous to standard two phase locking approaches such as multigranularity locking.

The Serializable SI algorithm is conservative, and in some cases leads to significantly higher abort rates than SI. One observation about SI anomalies is that no cycle can exist if the transaction causing the incoming anomaly precedes the transaction causing the outgoing anomaly. We intend to investigate whether this observation can lead to a more efficient version of the algorithm that has fewer false positives.

7. REPEATABILITY ASSESSMENT RESULT

Figures 11a and 13 have been verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at `http://www.sigmod.org/codearchive/sigmod2008/`.

8. REFERENCES

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions (PhD thesis)*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1999.

[2] M. Alomari, M. Cahill, A. Fekete, and U. Röhm. The cost of serializability on platforms that use snapshot isolation. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, 2008.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, , and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM Press, June 1995.

[4] A. Bernstein, P. Lewis, and S. Lu. Semantic conditions for correctness at different isolation levels. In *Proceedings of IEEE International Conference on Data Engineering*, pages 57–66. IEEE, February 2000.

[5] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

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

[7] A. Fekete. Serializability and snapshot isolation. In *Proceedings of Australian Database Conference*, pages 201–210. Australian Computer Society, January 1999.

[8] A. Fekete. Allocating isolation levels to transactions. *PODS*, 2005.

[9] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, to appear.

[10] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, 2004.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] T. Hadzilacos. Serialization graph algorithms for multiversion concurrency control. In *PODS*, pages 135–141, 1988.

[13] K. Jacobs, R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, and B. Quigley. Concurrency control, transaction isolation and serializability in SQL92 and Oracle7. Oracle White Paper, Part No A33745, 1995.

[14] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *VLDB*, pages 1263–1274. ACM, 2007.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In A. L. Furtado and H. L. Morgan, editors, *VLDB*, page 351. IEEE Computer Society, 1979.

[16] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[17] Y. Raz. Commitment ordering based distributed concurrency control for bridging single and multi version resources. In *Proceedings of Third International Workshop or Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, pages 189–198. IEEE, June 1993.

[18] V. T.-S. Shi and W. Perrizo. A new method for concurrency control in centralized database systems. In R. E. Gantenbein and S. Y. Shin, editors, *Computers and Their Applications*, pages 184–187. ISCA, 2002.

[19] D. G. Sullivan. *Using probabilistic reasoning to automate software tuning*. PhD thesis, Harvard University, Cambridge, MA, USA, 2003. Adviser-Margo I. Seltzer.

[20] Transaction Processing Performance Council. TPC-C Benchmark Specification. http://www.tpc.org/tpcc, 2005.

[21] Y. Yang. The adaptive serializable snapshot isolation protocol for managing database transactions. Master's thesis, University of Wollongong, NSW Australia, 2007.