Precisely Serializable Snapshot Isolation (PSSI)

Stephen Revilak, Patrick O'Neil, Elizabeth O'Neil

University of Massachusetts at Boston
Boston, MA 02125, USA
srevilak@cs.umb.edu. poneil@cs.umb.edu, eoneil@cs.umb.edu

Abstract— **Many popular database management systems provide snapshot isolation (SI) for concurrency control, either in addition to or in place of full serializability based on locking. Snapshot isolation was introduced in 1995 [2], with noted anomalies that can lead to serializability violations. Full serializability was provided in 2008 [4] and improved in 2009 [5] by aborting transactions in dangerous structures, which had been shown in 2005 [9] to be precursors to potential SI anomalies. This approach resulted in a runtime environment guaranteeing a serializable form of snapshot isolation (which we call SSI [4] or ESSI [5]) for arbitrary applications. But transactions in a dangerous structure frequently do not cause true anomalies so, as the authors point out, their method is conservative: it can cause unnecessary aborts. In the current paper, we demonstrate our PSSI algorithm to detect cycles in a snapshot isolation dependency graph and abort transactions to break the cycle. This algorithm provides a much more precise criterion to perform aborts. We have implemented our algorithm in an open source production database system (MySQL/InnoDB), and our performance study shows that PSSI throughput improves on ESSI, with significantly fewer aborts.**

I. Introduction

Database serializability has been studied since the early 1970s [6]. Jim Gray proselytized its value in his mid-70's lectures, published later in [11]. "Serializable" is defined to mean "equivalent to serial execution" in terms of data access conflicts between concurrent transactions, so all constraints maintained by program logic will continue to hold when multiple transactions execute concurrently. Full serializability (with phantom avoidance) has traditionally been supplied by strict two-phase locking (S2PL), but this was often time-consuming because of Waits-For chains, so commercial products offered default lower isolation levels based on early IBM Research [10]. The default isolation level for most S2PL systems, including DB2 and Microsoft SQL Server, was Read Committed (RC), in which a lost update by two transactions reading and then writing the same data item concurrently is common.

Snapshot Isolation (SI) was introduced in [2] in 1995. It was shown there that none of the ANSI SQL Phenomena named in [1] to illustrate weaknesses in isolation levels below Serializable (SR) occur in SI transactions. This does not mean that SI is equivalent to SR however, since less obvious SI anomalies exist. Many commercial applications do not exhibit anomalies when running under SI because the anomalies often require conditions that experienced developers avoid, such as querying important aggregate quantities from a number of disparate elements, rather than materializing the aggregates and updating them as the elements are updated. For example, the TPC-C [17] application was shown in [9] to execute serializably in SI. Because of this behavior and because data reads are never delayed by Write locks in SI, many commercial database products now offer SI to their customers, including Oracle RDBMS, PostgreSQL, Microsoft SQL Server, and Oracle Berkeley DB.

Nevertheless, SI anomalies have been found in some applications developed in the software industry [15]. An early approach to providing serializability in snapshot isolation [9] showed how to detect anomalies that could arise in applications, along with a number of techniques to modify code to avoid these anomalies, and in [8], an approach was introduced to automate the analysis of [9]. A more recent pair of papers ([4][5]) provide Serializable Snapshot Isolation (SSI), which avoids such anomalies at runtime without any need to pre-examine the code, clearly a valuable advance. However, this runtime technique relies on aborting transactions that give rise to dangerous structures, and thus can cause unnecessary aborts. In [4] and most of [5], the algorithms eliminate all dangerous structures as soon as they can be detected during reads and writes and commits. In [5], an enhanced SSI system that we will call ESSI is described and tested, a system that eliminates only the crucial subset of dangerous structures to ensure serializability, at the cost of additional bookkeeping. We have implemented our own version of ESSI that does all testing at commit time, for comparison with PSSI. Note that among systems providing serializable executions, there is a strict progression in terms of increasing precision: SSI, ESSI and PSSI.

A. *Prior Work In Detail*

Definition 1.1. Snapshot Isolation (SI). A transaction T_1 executing under SI reads only data from a *snapshot* of committed data as of the time T_1 started, called its *Start-Timestamp:* $S(T_1)$. The SI system can set $S(T_1)$ to any time not greater than T_1's first data access. Snapshots are typically implemented by maintaining distinct versions of all updates to rows, along with time of update (including *dead versions* for deleted rows and *initial versions* for inserted rows). Then T_1, when it reads a row r it did not itself update, will see the version of r committed most recently prior to $S(T_1)$. T_1's writes (updates, inserts, and deletes) are also reflected in its private snapshot, to be accessed again if T_1 reads or updates such data a second time.

When the SI transaction T_1 is ready to commit, it gets a *Commit-Timestamp:* $C(T_1)$, which is larger than any prior assigned Timestamp. T_1 will commit successfully only if no committed transaction T_2 whose commit came after $S(T_1)$ wrote data that T_1 is also trying to write. If that happened then T_1 will abort. This feature, called *First-Committer-Wins*

(FCW), prevents lost updates [5], where two transactions read the same row version and write back different versions. A variant approach, called *First-Updater-Wins (FUW)*, is used in Oracle and in our own PSSI-System. FUW tests for conflicts as each row is updated: if T_2 updates row r and T_1 subsequently attempts to update r, then T_1 will Wait until T_2 either Commits (then T_1 will abort) or T_2 aborts (then T_1 will perform its update of r and continue). We will adopt FUW as the default rule in what follows. Only after a transaction T_1 successfully commits will its updates become visible, and then only to transactions whose Start-Timestamps come after T_1's Commit-Timestamp. Note that $C(T_k)$ can also be represented as C_k. Note too that papers [4] and [5] also use the FUW rule, but still refer to it as FCW.

Definition 1.2. Concurrent SI Transactions; Transactional Lifetime. Two SI transactions T_1 and T_2 are said to be *concurrent* when their *transactional lifetimes*, the intervals $[S(T_1), C(T_1)]$ and $[S(T_2), C(T_2)]$ have a non-empty intersection.

We note that a SI transaction T_1 need never wait to perform a read because of being blocked by a writer, since all row versions that T_1 reads are already committed; also, the set of row versions read by T_1 are always consistent, in a Read Committed sense. Predicate reads are also expected to be consistent with the version of data read, because index retrieval results are also versioned.

SI does not exhibit any of the anomalous phenomena, P0 through P4, defined in Section 3 of [2], which are associated with classical Isolation Levels of Strict Two-Phase Locking (S2PL) systems. However, SI does have some anomalies of its own.

Example 1.1. SI Write Skew. Suppose X and Y are data items representing checking account balances of a married couple. The bank permits either account to be overdrawn if the sum of balances remains positive: $X + Y > 0$. The versioned SI history H1 below begins with initial versions $X_0 = 70$ and $Y_0 = 80$, and an update to a data item X by a transaction T_k results in a version named X_k. In H1 W_1 creates X_1 and W_2 creates Y_2. Each update leaves a negative quantity for X and Y since it has read a total that exceeds 0, but concurrent changes leave both values negative.

H1: $R_1(X_0,70)$ $R_2(X_0,70)$ $R_1(Y_0,80)$ $R_2(Y_0,80)$ $W_1(X_1,-30)$ C_1 $W_2(Y_2,-20)$ C_2

This problem is not detected by the FUW rule because two different data items are updated, each assuming that the other remained stable. In [9] this anomaly is shown to be avoidable by using *select for update* for any data item X a transaction intends to remain stable, with FUW occurring if another transaction updates X. Thus T_1 could stabilize Y and T_2 stabilize X to avoid the anomaly in H1.

Example 1.2. Predicate Write Skew. Given an employees table with primary key eid and an assignments table with columns (eid, workdate, hours, projid), employees are assigned a number of hours of work on various projects on a given workdate by placing a row in the assignments table. The transactions doing this must maintain a constraint that no employee can be assigned more than eight hours work on any date. In the SI history H2, two different transactions concurrently assign an employee a new assignment for the same day. Predicate P says eid = 'e1234' and workdate = '09/22/10', and the query Q_k(P, result) has T_k retrieve the current number of hours assigned that eid on that date; the operation I_k(assignments,…) inserts defined rows into the assignments table.

H2: Q1(P, empty) I1(assignments, eid='e1234', projid=2, workdate='09/22/02', hours=5) Q2(P, empty) I2(assignments, eid='e1234', projid=3, workdate='09/22/02', hours=5) C1 C2

Both T_1 and T_2 evaluate predicate P and find no work assignment for the given eid and workdate. Each then inserts a 5-hour work assignment on that date. This results in 10 hours of work assigned to the employee, breaking the constraint of an eight-hour limit, although both transactions acted consistently. This is a form of Write Skew, but the anomaly cannot be avoided by using *select for update* on any set of data items. We can avoid the anomaly by creating a total_work_hours table with a unique row for each employee-day pair, inserting or updating that row as we add each new assignment. Given this, T_1 or T_2 would fail by FUW.

B. Our Contribution: PSSI

In papers [4][5], the first example given to demonstrate SI anomalies is a predicate write skew, where doctors sign themselves out from being on-duty when the count of doctors found by a "select count of all on-duty rows", is at least two. Of course if the last two doctors on duty sign out concurrently, we can end up with none on-duty. A solution would be to create a total-on-duty row, and update that row as each doctor goes off duty, leading to FUW for a later update in concurrent attempts. But such a solution, requiring changed application code, is not very useful in large old applications, which are difficult to modify. Papers [4][5] provide a runtime test that aborts transactions involved in a dangerous structure and thus guarantee serializable isolation for arbitrary applications using SI, a valuable advance. But transactions involved in a dangerous structure often do not cause anomalies, so the authors point out their method is *conservative* in that it can cause unnecessary aborts. We will define dangerous structures and illustrate this fact in Section II.

In the current paper we propose a new concurrency control algorithm, called Precisely Serializable Snapshot Isolation (PSSI), with the following properties.

- PSSI ensures that every execution of a valid transactional application is serializable.
- Write and Read operations never delay each other in PSSI; Writes are delayed only due to FUW.
- The throughput of PSSI is superior *our implementation* of (Enhanced) Serializable Snapshot Isolation (ESSI) introduced in [5] in measures we performed.
- Unlike ESSI, PSSI required numerous modifications to MySQL/InnoDB. We have prototyped and tested PSSI and the prototype is available for testing by others.

- PSSI is Precise in that it only aborts transactions that would otherwise cause a failure of serializability. There is a caveat to this property, explained below.

The caveat to the point that PSSI is Precise is that PSSI avoids phantom anomalies as InnoDB does, using a variant of IM index locking [13][16], but this is valid only up to predicate lock accuracy. If transaction T_1 performs a query of the form:

[1.2.1] select count(*) from Tbl where col1 between 20 and 30;

and a concurrent T_2 were to insert a row with col1 = 29, then we would expect a RW dependency (conflict) to point from T_1 to T_2. If T_1 were to repeat select statement [1.2.1], it would retrieve the same count, but a cycle of dependencies can cause non-serializable behavior, so each dependency is important. Now if the relevant values of col1 are 15, 25, and 35, then Query [1.2.1] under InnoDB IM locking would lock the row value 25, and gaps between 15 and 25 and between 25 and 35. This is because no form of IM locking has the ability to lock ONLY the range 20 to 30, which is all that needs to be locked. Thus if T_2 were to instead try to insert a row with col1 = 33 (which does not actually conflict with Query [1.2.1]), a RW dependency would still point from T_1 to T_2 because of the gap lock between 25 and 35, and this could eventually lead to a cycle of dependencies and an abort of T_1 or T_2. This is an unnecessary abort, so PSSI is not perfectly precise. However the fault lies with IM locking, and the same problem would arise with any S2PL system as well, including DB2 and Microsoft SQL Server. It might be possible to create a more precise version of predicate locking by naming specific range end-points, but this would require more heavyweight locks, and may not be worth the precision gained.

The rest of this paper is structured as follows: in Section II we explain SI Transaction Theory and describe PSSI design; InnoDB Modifications for PSSI are outlined in Section III; Section IV provides a PSSI Performance Analysis; Section V provides a suggestion for Future Work.

II. SI TRANSACTION THEORY

Transactional dependencies in snapshot isolation are ordered conflicts between transactions, catalogued by type. The type specifies whether the conflict involves only data items (rows), symbolized by -i-, or a predicate read and item write, symbolized by -pr-. The dependency T_m--i-wr$\rightarrow T_n$ means that T_m writes (inserts, updates, or deletes) a row version and T_n then reads the new version (noting absence in the case of a delete). Other item conflicts are T_m--i-ww$\rightarrow T_n$ and T_m--i-rw$\rightarrow T_n$, and in each case, the write by T_n updates the prior version installed by the T_m or read by T_m. For predicate conflicts, T_m--pr-wr$\rightarrow T_n$ and T_m--pr-rw$\rightarrow T_n$, the pr symbol involves a predicate read -r-, and the -w- symbol involves a write of a row (update, insert or delete) that would modify the predicate result. Note that the row write need not create a row retrieved by the predicate to modify the result; if a predicate retrieves the name of the horse in show (third) place in a race, then scratching the winner (delete) will change what the predicate retrieves, though the winner was never retrieved by the query. We can refer to more generic dependencies, such as T_m--wr$\rightarrow T_n$ (a *wr dependency*) for either T_m--i-wr$\rightarrow T_n$ or T_m--pr-wr$\rightarrow T_n$. Similarly, T_m--rw$\rightarrow T_n$ can represent either the -pr- or -i- type, and T_m--ww$\rightarrow T_n$ is unambiguously of -i- type. Note that all of these dependency arrows are oriented *from earlier to later in time* as we see below in SI-RW Diagrams.

Definition 2.1. DSG(H). A directed graph DSG(H), the *Dependency Serialization Graph* on the multi-version history H3, has *vertices* representing transactions that commit, and each distinctly labeled *edge* from T_m to T_n corresponding to a T_m--wr$\rightarrow T_n$, T_m--ww$\rightarrow T_n$ or T_m--rw$\rightarrow T_n$ dependency.

Example 2.1. Consider SI history H3: $W_1(X_1)$ $W_1(Y_1)$ $W_1(Z_1)$ C_1 $W_3(X_3)$ $R_2(X_1)$ $W_2(Y_2)$ C_2 $R_3(Z_1)$ C_3

See DSG(H3) in Figure 2.1. In H3, versions X_1, Y_1, and Z_1 are written at time C_1. Then version Y_2 replaces Y_1 as of C_2, so T_1--ww$\rightarrow T_2$, and Y_3 replaces Y_1 as of C_3, so T_1—ww$\rightarrow T_3$. The operation $R_2(X_1)$ means T_1--wr$\rightarrow T_2$. Note that $R_2(X_1)$ occurs after $W_3(X_3)$, but there is no T_3--wr$\rightarrow T_2$ because T_2 cannot read versions written by T_3. Indeed the reverse is true since T_2--rw$\rightarrow T_3$ because T_2 reads an earlier version (X_1) than the X_3 written later by T_3. Operation $R_3(Z_1)$ means T_1--wr$\rightarrow T_3$. Note the transactions in DSG(H3) are not in serial order since T_2 and T_3 are concurrent; however an examination of Figure 2.1 shows that a topological sort provides an equivalent serial order: T_1, T_2, T_3.

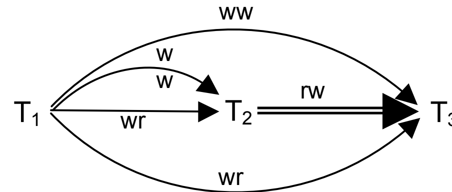

Figure 2.1 DSG(H3)

Note that the edge from T_2 to T_3 in Figure 2.1a is drawn as a "double line", a convention we use to represent a -rw- dependency (an anti-dependency) between two ***concurrent*** transactions. We see that in DSG(H3), it is not at all clear that T_2 and T_3 are concurrent transactions but we will shortly introduce a new diagram form that makes transaction duration more obvious.

Remark 2.1. Both T_m--wr$\rightarrow T_n$ and T_m--ww$\rightarrow T_n$ dependencies have the property that T_m must commit before T_n starts; in a -wr- case, T_n cannot read an item T_m has written unless $S(T_n)$ comes after C_m, and the same is true in a -ww- case, since T_m and T_n cannot both write the same item if they are concurrent (because of FUW). Thus a cycle cannot occur in a history having only transactions with -wr- and -ww- dependencies.

Remark 2.2. As we noted above, a T_m--rw$\rightarrow T_n$ dependency can occur between concurrent transactions and is then known as a *concurrent anti-dependency*, represented by a double line in diagrams. By Remark 2.1 at least one such anti-dependency must occur for a cycle to exist in the DSG, because all other dependency types go from left to right across commits of the earlier transaction.

The following definition of dangerous structure originally appeared in [9], and is used in [4][5]. The definition of essential dangerous structure is new here, but the importance of the concept figures in [5].

Definition 2.2 Dangerous Structures and Essential Dangerous Structures. Suppose in the serialization graph DSG(H) of some history H, there are three consecutive transactions T_1, T_2, T_3 (it might be possible that T_1 and T_3 are the same transaction), where T_1 and T_2 are concurrent, with T_1--rw→T_2 and T_2 and T_3 are concurrent with T_2--rw→T_3. Then this triple of transactions is called a *dangerous structure*. If the commit of T_3 is first of these three to commit, we call the structure an *essential dangerous structure.* See Figures 2.3 and 2.4 for illustrations or essential and non-essential dangerous structures.

The following theorem appeared in [9] with somewhat different wording, and we do not prove it here.

Theorem 2.1. Suppose H is a multi-version history arising in Snapshot Isolation that is not serializable. This can happen if and only if there is at least one cycle in the serialization graph DSG(H), and we claim that in every cycle there are three consecutive transactions, T_1, T_2, and T_3, (where T_1 and T_3 might be the same transaction) that qualifies as a Dangerous Structure. As part of the proof of this Theorem, it was shown that the commit of T_3 had to be first of the three transactions T_3, T_2, and T_1 for the cycle to exist, that is, the dangerous structure ensured by the cycle is in fact an essential dangerous structure.

In S2PL systems, two or more concurrent transactions can perform multiple operations interleaved in time, causing quite complex conflicts. But by Definition 1.1, a transaction T_k in SI only performs operations at two points in time, reads at $S(T_k)$, and writes at $C(T_k)$, and constructing a cycle for two or more transactions using dependencies that point forward in time seems nearly impossible. It is possible only because each SI transaction T_k has start time $S(T_k)$ and commit time $C(T_k)$ separated in time.

Recall from Example 1.1 the anomaly illustrated by SI Write Skew, with the history:

H1: $R_1(X_0,70)$ $R_2(X_0,70)$ $R_1(Y_0,80)$ $R_2(Y_0,80)$ $W_1(X_1,-30)$ C_1 $W_2(Y_2,-20)$ C_2

We illustrate this history with what we call an *SI-RW diagram* [9] for an SI based history: see Figure 2.2. In an SI-RW diagram, all transaction T_i Reads are assumed to occur at an instant of time at a vertex R_i, a pseudonym for $S(T_i)$, and all writes at an instant of time at a vertex W_i, a pseudonym for $C(T_i)$; vertex R_i appears to the left of vertex W_i (*time increases from left to right*), with the two connected by a horizontal dotted line segment, as we see in Figure 2.2. The SI-RW diagram has two types of edges: *Sibling edges* and *Conflict edges*; in our SI-RW diagrams, the sibling edges are the dotted line segment connecting the R_i and W_i vertices in any transaction T_i which has both Reads and Writes.

Note in Figure 2.2 that T_1 reads X and T_2 writes X (a -rw-dependency from T_1 to T_2), while T_2 reads Y and T_1 writes Y (another -rw- dependency from T_2 to T_1). The result is a dependency cycle for Write Skew.

Transaction T_1 is the upper dashed line in Figure 2.2, with R_1 on the left and W_1 on the right, and T_2 is the corresponding lower dashed line. We see that the two -rw- dependencies do indeed point forward in time. In terms of Theorem 2.1, T_1 is identified with T_3 (the two are identical). Notice that T_3 (i.e., T_1) does commit before T_2. We can now trace a cycle from R_2 through R_2--rw→W_1, then backwards in time through the duration of T_1 to R_1 and along R_1--rw→W_2, then complete the loop by going backwards in time from W_2 to R_2. It is the delay from R to W in T_1 and T_2 that allows these legs backward in time.

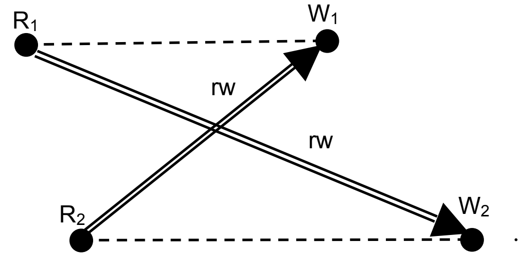

Figure 2.2. SI-RW Diagram of Write Skew History H1

Now if T_1 and T_3 are *not* identical as in Figure 2.2, then the three transactions of Theorem 2.1 make up a Dangerous Structure, with SI-RW Diagram pictured in Figure 2.3.

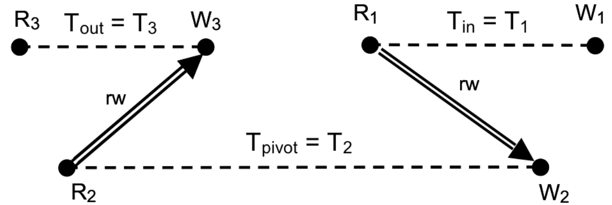

Figure 2.3. SI-RW Diagram of an (Essential) Dangerous Structure.

Note that in Figure 2.3, T_2 is also known as T_{pivot} in the terminology of [4][5], T_3 is also called T_{out} since the -rw- edge points out from T_{pivot} to T_3, and T_1 is also known as T_{in} since the -rw- edge points in from T_1 to T_{pivot}, and the order of commit of the three transactions is T_3, T_2, T_1, as required in our statement of Theorem 2.1. There is no guarantee that a Dangerous Structure results in a cycle: one might exist if there is a dependency path from T_3 to T_1, such as from W_3 to W_1 but such a path is by no means guaranteed. Thus a Dangerous Structure is only a possible precursor of a dependency cycle, which is why [4] and [5] said aborting transactions to break up dangerous structures was *conservative*, since it caused unneeded aborts. In Figure 2.4 we see a non-essential dangerous structure where T_3 commits later than T_2, which need not be aborted to avoid a cycle. In Figure 2.4, no path from W_3 to W_1 could complete a cycle unless it contained its own dangerous structure, since the two transactions are concurrent. Any dependency edge leaving R_3 to create a cycle (clearly an -rw- edge, say to W_4), would itself create another dangerous structure, with T_3 as the pivot, T_2 as T_{in}, and T_4 as T_{out}, so T_3 would be aborted.

Recall that the concurrency control algorithm in [4] that aborts all dangerous structures is called SSI, and the algorithm in [5], which aborts only essential dangerous structures, as

Enhanced SSI, or ESSI, a less conservative variant. Figure 2.4 is an example of a structure that would cause a transaction abort in SSI but not ESSI.

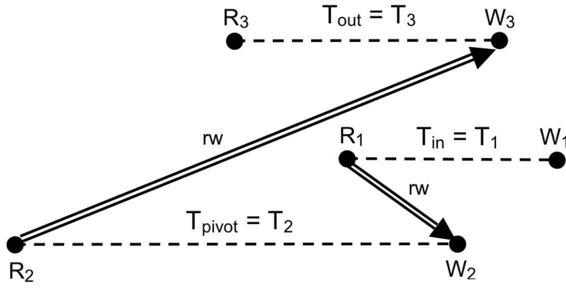

Figure 2.4. SI-RW Diagram of a (Non-Essential) Dangerous Structure.

Our PSSI must detect cycles at commit time before aborting transactions, making PSSI a Serial Graph Testing certifying scheduler [3]. Our measurements lead us to believe that PSSI is superior in terms of throughput to the ESSI of [5]. Certainly PSSI is subject to many fewer serializability aborts than ESSI. We later compare PSSI to our own implementation of ESSI, which is not identical to the ESSI implementation from [5].

III. PSSI IMPLEMENTATION

Native InnoDB provides two choices of Isolation Level: (1) strict two-phase locking (S2PL), using a variant of index-specific IM locking [16][13] to guard against phantom anomalies, and (2) Multi-version concurrency control (MVCC), where each transaction sees only versions of data created as of its start time [13]. MVCC is not SI because MVCC does not support FCW or FUW. We have modified InnoDB to provide SI, our own version of ESSI (based on design in [5], but using our own InnoDB modifications), and PSSI, a much more complex task.

A transaction T_i running at "traditional" SI does not set read locks; it simply reads rows based on the snapshot defined by its start time $S(T_i)$. However SI transactions do take write locks, to support FUW. If T_k attempts a write lock and must wait behind T_i, which already holds a write lock, then it waits until T_i's locks are released. When T_i commits, all transactions waiting behind its write locks are aborted, but if T_i aborts, the first waiting T_k waiting for each of T_i's write locks leaves lock wait and continues. (Others may still be waiting.) This is comparable to the approach used in Oracle.

For each transaction T_n running in PSSI we note read locks (similar to SIREAD locks of [4][5] in that read locks never cause waits) and write locks in a *lock table* we modified from InnoDB, as explained below in Section III.D. But only –ww- conflicts are detected and acted on while T_n is active, since these conflicts can cause FUW waits, and so must be detected immediately. We wait for T_n's commit time to enter -rw-, -wr-, and -ww- conflicts to our *Cycle Testing Graph (CTG)*, which we added to InnoDB to test for possible dependency cycles. We defer this test until C_n in PSSI for efficiency reasons, because that's the first time we will know all dependencies from T_n to earlier committed transactions, and a full test for cycles is simplified by the fact that we need only perform one depth-first search from T_n to find if any path forms a cycle in CTG. Thus in PSSI a committing transaction T_n can create dependency cycles only with transactions T_m that have already committed, so committed transactions must continue to exist as *zombie transactions* in the lock table and the CTG, in order to support tests for such cycles. See III.D for the test whereby a committed transaction eventually ceases to be a zombie and disappears from the lock table and CTG. Since the cycles we find from a committing T_n will contain only zombie transactions, the cycle can be broken only by aborting T_n. All other transactions in the cycle will have already committed. We provide details of the CTG in III.C.

A. Zombie Transactions

As each transaction starts, it is assigned a Transaction ID (TID). The TID is an integer n and the transaction is represented as T_n to symbolize this. As each active T_n performs read and write operations, we track these reads and writes in the lock table. On commit of T_n, we need to determine all dependencies from T_n to previously committed transactions T_m that might still be involved in a dependency cycle with a newly committed T_n, i.e., zombie transactions. We give some examples below of cycles between newly committed transactions and previously committed transactions. In order to determine what dependency edges exist between such transactions, read and write locks previously taken by zombie transactions must continue to exist in the lock table, and directed edges between all such transactions must continue in the CTG.

Figure 3.1 illustrates why a transaction cannot be ignored even after it has committed. In Figure 3.1a, transaction T_1 has

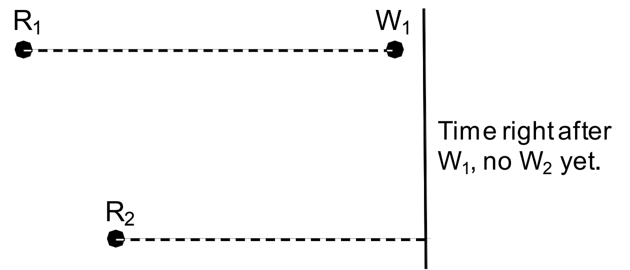

Figure 3.1a. Potential Cycle

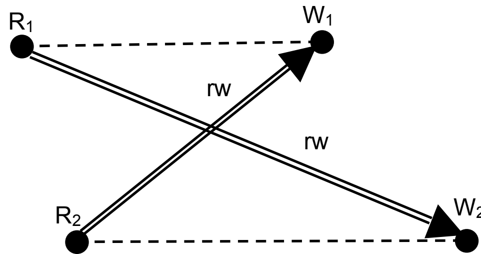

Figure 3.1b. SI-RW Diagram of Completed Cycle

committed, but transaction T_2 is concurrent with it and still active, so new dependency edges can still arise between T_1 and T_2, leading to a cycle. Note that after C_1, T_2 might read a row that was written by T_1 (causing the dependency edge T_2--rw→T_1 of Figure 3.1b) and T_2 might also update a new data item that was read but not written by T_1 (causing edge T_1--rw→T_2 of Figure 3.1b). The cycle of Figure 3.1b is the same

skew-write cycle we illustrated in Figure 2.2. Since T_1 had already committed but could still form a cycle with committing transaction T_2. Thus T_1 is a zombie transaction.

It is also possible for a transaction to be a zombie even if no active transaction is concurrent with it. In Figure 3.2, T_1 is a zombie because a cycle from the active T_4 can include a long-committed T_1. Note in Figure 3.2 that while T_4 remains active after T_3 commits, it is possible for T_4 to overwrite a row that T_1 read earlier, creating a dependency T_1--rw→T_4, and read a row that the concurrent T_3 wrote earlier, creating T_4--rw→T_3. We can now go backward in time from T_4 through T_4--rw→T_3 then back through the duration of T_3, and through T_3--rw→T_2 and back through the duration of T_2, then through T_2--rw→T_1 and complete the loop with T_1--rw→T_4. Note too that instead of T_4 writing a row that T_1 read earlier, T_4 could also write a row that T_1 wrote earlier -- thus a T_1--ww→T_4 (with T_1 and T_4 non-concurrent) could create a different cycle from T_1 to T_4 and back to T_1.

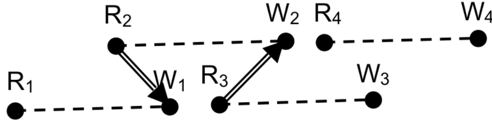

Figure 3.2. "Lightning Bolt" and a Potential Cycle

We define a "lightning bolt" of length k to be a sequence of transactions T_1-T_2-...-T_k, $k \geq 2$, where consecutive T_{i-1} and T_i are concurrent and have an anti-dependency between them, T_i--rw→T_{i-1} for each i, $1 < i \leq k$. In Figure 3.2, the sequence of transactions T_1-T_2-T_3 forms a lightning bolt of length 3. If we were to drop transaction T_1 from this sequence and keep only T_2-T_3, we would have a lightning bolt of length 2. Note the reason there cannot be a lightning bolt of length 1 is that two transactions are needed for an anti-dependency to exist.

Even if T_1 is deleted from Figure 3.2, an active T_4 can still cause a cycle with T_2-T_3 by creating a -rw- dependency from R_4 to W_3 and either a -rw- or -ww- dependency from T_2 to W_4. On the other hand, if we kept the lightning bolt T_1-T_2-T_3 and simply had T_4 create a -rw- dependency from R_4 to W_3, we would end up with a lightning bolt of length 4.

We explain how lightning bolts are pruned in Section III.E, after we have discussed the design of the Lock Table and CTG component.

B. Lock Table

The Lock Table of Figure 3.3 is descended from the Lock Manager diagram in Figure 8.8 of Gray and Reuter [12]. We note in Figure 3.3 that the *lock objects* represented by boxes R_i and W_i each sit on two doubly-linked lists. One list has a header at the InnoDB transaction object T_i, and contains a series of lock objects taken by that transaction. Our implementation adds new read locks at the beginning of the list and new write locks at the end of the list, a minor optimization for later processing. The second list has a header in the entries on the left reached by the hash value of a table page containing one or more row locks. New locks are added in temporal order from the header, read locks in order by T_i and Write locks with notional time of infinity (except in one unusual case of implicit lock conversion).

Note that although the lock headers on the left are for table pages, we do not normally lock entire pages. Individual rows and ranges of rows within a page are represented as bit maps within InnoDB's lock objects. This provides a particularly efficient representation for long ranges.

When a read lock is added to the lock table, no dependencies are determined at that time so there is no effect on the transaction performing the read or on other transactions until commit time. When a write lock is added, the only aborts and blocking that can occur are due to FUW. Thus reads and writes succeed or fail just as in pure SI under FUW as detailed above. In SSI and ESSI, reads and writes can fail due to detection of dangerous structures, i.e., for isolation reasons. In PSSI, a full cycle must occur to cause an abort, and we do not attempt to detect cycles until commit time.

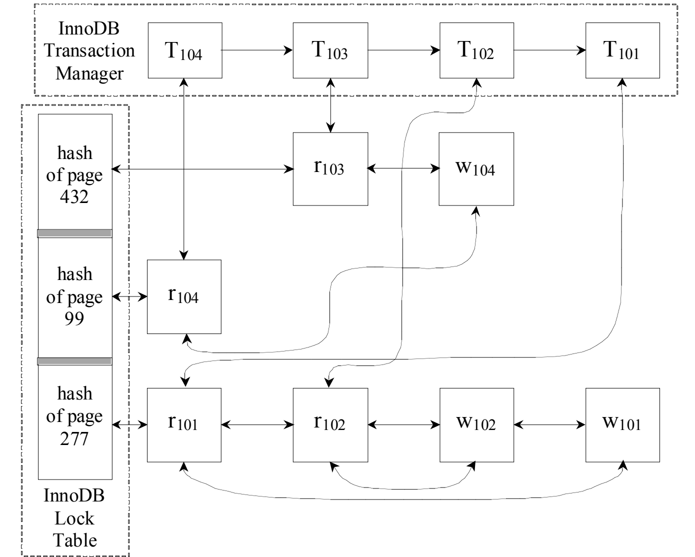

Figure 3.3. PSSI Lock Manager

C. Cycle Testing Graph (CTG) Component

As discussed above, a committing transaction T_k must be aborted if and only it creates a cycle of dependencies with previously committed transactions. PSSI checks this condition at commit time using the CTG, a graph with transactions as nodes and dependencies as edges, comparable to the DSG of Figure 2.1, except that if both -wr- and -ww- dependency edges exist in the same direction between two transactions, one of them is removed. These edges will end up without types in the CTG. The CTG contains previously committed transactions that can still figure in cycles (zombie transactions) plus the one now committing, if any. At each transaction T_k commits, the transaction is added to the CTG, and dependency edges to and from existing zombie transactions are determined from the Lock Table and added. The CTG is then cycle-tested with a depth-first search from T_k, and the T_k is aborted if a cycle is found. Finally, the CTG is "pruned" to remove transactions that cannot be a part of any future cycle.

The CTG contains the committed zombie transactions that are still represented in the lock manager by read locks and write locks. The edges of the CTG are directed and unlabeled. An edge exists from T_i to T_k if there is any type of dependency (read-write, write-read, or write-write) involving any data item, or multiple such dependencies. The edge link information is materialized as an array of edges in the source transaction object, along with a count of in-edges, to be used in pruning. Both the set of out-edges and the count of in-edges can change while a transaction resides in the CTG because newly committed transactions add to the edges and pruning removes edges as neighboring transactions are pruned away.

At commit time, it is straightforward to follow the chain of locks for T_k in the lock manager, and for each of T_k's locks, follow the chain of locks for a certain data item locked by T_k, to find all the locks held by T_m as well as T_k, and analyze these situations to find which ones in fact give rise to dependencies needed for the CTG. Each new dependency is entered in the source transaction's out-edge list and counted in the destination transaction's in-edge count.

D. Pruning Zombie Transactions

A zombie transaction T_k in the CTG can be pruned from the CTG and from the Lock table if it satisfies two conditions:

(C1) T_k's in-edge count is equal to zero.

(C2) T_k committed before the oldest active transaction started.

Condition C1 guarantees that there are no dependency edges leading into T_k from transactions still in the CTG, so no "lightning bolts" exist which include T_k. However Condition C1 allows T_k to be the first node in a lightning bolt. Condition C2 guarantees that no currently active transaction T_m can create a T_m--rw→T_k dependency edge, since that would require that R_m preceded W_k, which would mean T_m started before T_k committed, contradicting C2.

C1 and C2 are easy to check, and on finding such a transaction, its removal from CTG will remove out-edges and reduce the in-edge counts of other transactions, possibly making them eligible for removal as well. Note that the out-edges of the pruned transaction lead to the transactions with newly reduced counts, so a recursive search efficiently covers all possibilities. The algorithm for pruning zombie transactions has two stages:

1. Identify a starting set S of transactions in CTG that obey C1 and C2 and are therefore prunable.
2. Recursively prune the CTG starting from each transaction in S and following its out-edges to prune further as appropriate.

When the pruning algorithm completes, all transactions in the CTG that committed before the oldest transaction started have strictly positive in-edge count. At the next commit, the newly added transaction may be prunable, or this commit may move the start time of the oldest active transaction, so that a fresh batch of transactions satisfies condition C1.

We found no circumstances where lightning bolts grew arbitrarily long, but nevertheless implemented a method to restrict their length to a chosen limit L (such as 20 or 100), much as ESSI restricts their length to 2.

IV. Performance

We define a benchmark in this section named **SICYCLES**, intended for measuring performance in SI-like systems such as SI, ESSI and PSSI. **SICYCLES** is designed to run multiple identical concurrent transactions that read and update columns of distinct rows with the potential to generate anomaly cycles. The number of cycles measured may be expected to grow with the number of concurrent transactions supported, called the *MultiProgramming Level*, or *MPL*.

A. BENCH table for the SICYCLES benchmark

The **SICYCLES** benchmark runs transactions accessing a HOTSPOT on a single table named BENCH, which contains 1,000,000 rows. Columns of BENCH are defined as integer, not null, except for a kpad column of type char.

kpad has 1,000,000 distinct char values generated with length 20 to give all rows a 100-byte length

The integer columns for the 1,000,000-row table BENCH are described as follows.

kseq has values 1 to 1,000,000 in order on the rows stored and is the primary key for the table.

krandseq has values 1 to 1,000,000 distributed randomly on the rows and is unique on the table.

kval (the only column updated) has values between 10,000 and 99,999 distributed randomly.

A list of 17 different kn columns are defined below

Updates of kval in **SICYCLES** will add a small amount based on the values of the kval columns read, as described below; multiple updates of this kind will continue to leave kval values approximately the same.

The kn columns are listed as follows:

k4, k8, k16, k32, k64, k128, k256, k512, k1024, k2500, k5k, k10k, k25k, k50k, k100k, k250k, k500k.

Each kn column has n distinct values 1 to n randomly assigned to the different rows. Thus we will be able to generate predicates of the form "kn = m" with $1 \le m \le n$ to contain about 1,000,000/n rows independent of one another. All columns except kval (often modified) and kpad (never accessed) are indexed.

It is intended that the BENCH table always remain memory (buffer) resident for quick access.

If a HOTSPOT for queries and updates is chosen by a range on kseq or krandseq, and then the same index is used to select kval updates, there will be conflicts on the index page for kval that greatly slow the benchmark threads. Therefore we usually define a hotspot as a set of rows randomly chosen on the table and then remember the krandseq values to access those rows.

B. SICYCLES *Transaction Definition*

Recall that kval (the only column updated) has values between 10,000 and 99,999 distributed randomly. Each **SICYCLES** benchmark run has concurrent transactions with MultiProgramming Level (MPL) M, profiled by an integer pair (K, N), K ≥ 1, N ≥ 1. Each transaction in this run reads k randomly chosen rows (x_1, x_2, …, x_K) from a HOTSPOT of the BENCH table to find the average kval value v, then adds a small positive or negative fraction of the value v, c*v or -c*v (c = 0.001, positive or negative values chosen at random) to each of the kval values of rows (y_1, y_2, …, y_N), also in the HOTSPOT. No pair of rows from the union of x_i rows and y_j rows are identical.

For K = N = 1, we say that a transaction T_i that reads kval from the single row x and adds the positive or negative fraction c of x to y "copies a fraction of x to y leaving x unchanged", and that x is the source row and y is the sink row of T_k. We refer to this by saying for short: "T_k acts on (x, y)". Since the rows x and y are randomly chosen from a Hotspot described below, if T_k acts on (x, y) and a concurrent transaction T_m acts on (y, x), we end up with a skew write cycle, as in Figure 2.2. We can get a cycle of length three if the T_1 acts on (x, y), T_2 acts on (y, z), and T_3 acts on (z, x). Similarly we can create any larger length cycle by choosing x, y, …, z and copying by pairs from successive rows in the sequence to the next row, then copying from z to x at the end. Similar cycles can occur in many ways when K >1 and N > 1.

We note that the **SICYCLES** benchmark has an interesting property, that there is no reasonable set of aggregates to materialize that could be updated along with the row values themselves to avoid an anomaly, even in the case K = N = 1. In Write Skew Example 1.1, we have a constraint that X + Y > 0, and if we materialize a sum S = X + Y that must be updated with every update of X or Y this would avoid the write skew anomaly. But the only constraint that holds in **SICYCLES** when T_1 acts on (x, y) is that after running a concurrent set of transactions, x remains unchanged and y ends up somewhat larger. If there is any sequence x, y, …, z where successive pairs are acted on by concurrent transactions with no cycle, we still have y, …, z increasing in value and x remaining the same. Only if there is a cycle will no row retain its original value after running a concurrent set of transactions. The only set of aggregates we can add that would detect a cycle is the set of all row pairs (x, y) with a rule that at least one of the rows must remain the same after a concurrent set of transactions acts on them. A cycle will break this rule.

C. *Performance Measurements*

We implemented and tested the PSSI system, and our versions of SI and ESSI, using MySQL 5.1.31 with its InnoDB engine. Our tests used a two-machine client/server setup, the server being an HP Z400 workstation with a Quad-core 2.66 GHz Intel Xeon 3520 CPU, 4GB of RAM, running 64-bit OpenSUSE 11.2 (Linux kernel version 2.6.31) with a 7200 RPM Seagate SATA disk. Our client machine was a Lenovo ThinkPad with a 2.53 GHz Intel Core-2 Duo CPU, with 2GB of RAM, and a 32-bit version of OpenSUSE 11.2 (also kernel 2.6.31). Our client program was written in Java, and run with Sun's JVM version 1.6.0_18, and MySQL/ConnectorJ driver version 5.1.1. To minimize network latency, the client and server machines were connected via a dedicated physical LAN, with a gigabit Ethernet switch.

Each of our data points was chosen as the median from three 60-second measurement periods. Prior to each measurement period, we (a) start the MySQL server process, (b) run a series of select statements to bring needed pages into buffer (all row accesses during tests are buffer resident), and (c) execute a two-second warm-up, allowing the client to reach full MPL. At the end of each measurement period we bring down the MySQL server after a two-second cool down.

As noted in Section IV.B, we varied the number of reads, k, and the number of writes, n, in our tests, and we will denote this using the shorthand notation s*K*u*N*, k single-row selects, and n single-row updates. For example, the notation "s3u1" means "select three, update one".

Our test client inserts a randomized delay of 3ms ± 50% after each single row select or update (with the exception of the final statement before commit), and there is no think time between transactions. The randomized delays prevent our test system from being saturated prematurely, thereby allowing us to report measurements with a greater range of results. The lack of think time between transactions keeps the number M (for MPL) of concurrently active transactions consistent during the measurement period.

We found it rather simple to implement ESSI's criteria for aborting transactions, but our implementation differs in a number of ways from the one described in [5], so we cannot be fully confident that a throughput comparison between PSSI and our implementation of ESSI would be valid. That said, the comparison of serializable aborts and FUW aborts per transaction seems to indicate that the throughput differences between ESSI and PSSI are approximately correct. We hope to provide an apples-to-apples comparison between PSSI and ESSI at some future time, where the two implementations compete on the same hardware.

As a final point, we note that most of our measurements were run with one log flush to disk each second, whereas the ESSI measurements of [5] were run with a log flush following each commit. We show one measurement with a disk flush after each commit (Figure 4.9), but this approach slows performance markedly, since the frequent disk access lowers the effective MPL. However it is the more correct approach since InnoDB does not offer Group Commit, which allows multiple transactions to commit without acknowledging success until all logs in the group of transactions are flushed to disk. In this way transactions can run continuously, writing logs to a second buffer while the first one is being written to disk. All major DBMS systems provide Group Commit and there are comments about implementing Group Commit in the InnoDB source code. We decided to err on the side of better performance available on major systems and hopefully in InnoDB at some point in the future by performing infrequent log flushes in most of our tests

1) Successful Commits and Serialization Aborts s5u1

Figures 4.1, 4.2 and 4.3 show graphs of transactions with 5 selects and one update (s5u1) that commit successfully (CTPS for "Committed transactions per second"), which shows the most favorable results for PSSI vs. ESSI performance. The least favorable results occur when the number of selects and updates are the same (s3u3 and s1u1), as we will see.

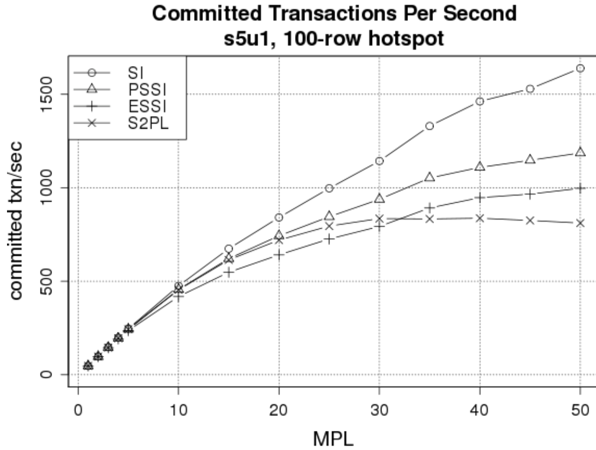

Figure 4.1 Committed Transactions Per Second, s5u1, 100 row hotspot

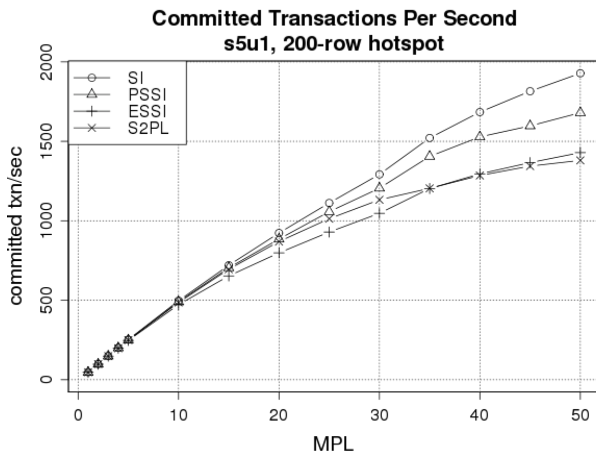

Figure 4.2 Committed Transactions Per Second, s5u1, 200 row hotspot

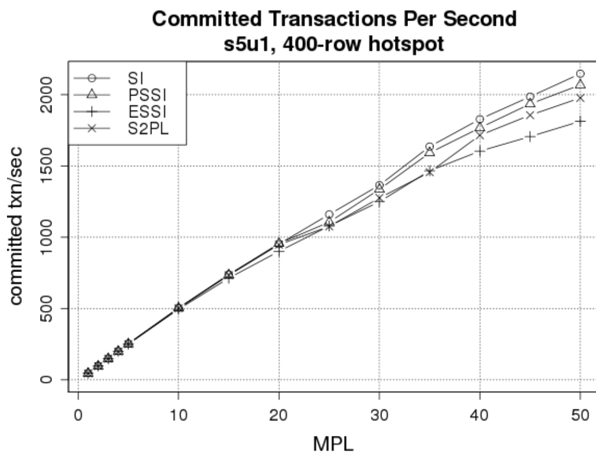

Figure 4.3 Committed Transactions Per Second, s5u1, 200 row hotspot

Note that the 100-row hotspot in Figure 4.1 seems too restrictive, since SI shows just over 1600 CTPS compared to just over 1900 CTPS in Figures 4.2 and over 2100 CTPS in 4.3, and S2PL shows a negative slope after 30 MPL. We will confine ourselves to 200 and 400-row hotspots in what follows. In Figures 4.2 and 4.3, SI has the most commits per second, but of course SI has no serializability aborts so it can commit anomalous transactions. PSSI, ESSI, and S2PL allow only serializable executions, but ESSI aborts essential dangerous structures while PSSI only aborts cycles. Thus PSSI can be expected to have a smaller number of serialization aborts than ESSI. And in fact we note that PSSI has higher CTPS in all our tests.

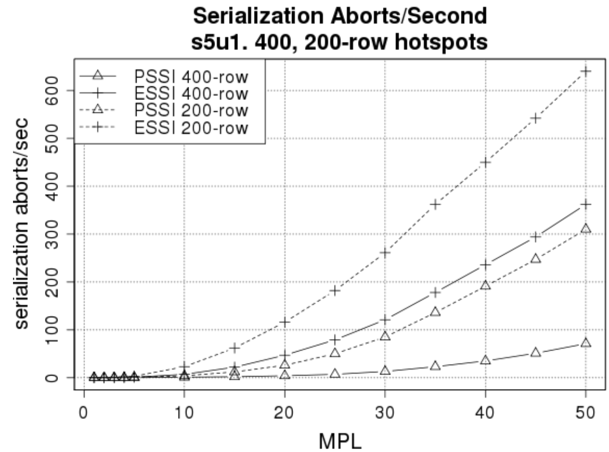

Figure 4.4 Serialization Aborts, s5u1, 400 $ 200 row hotspots

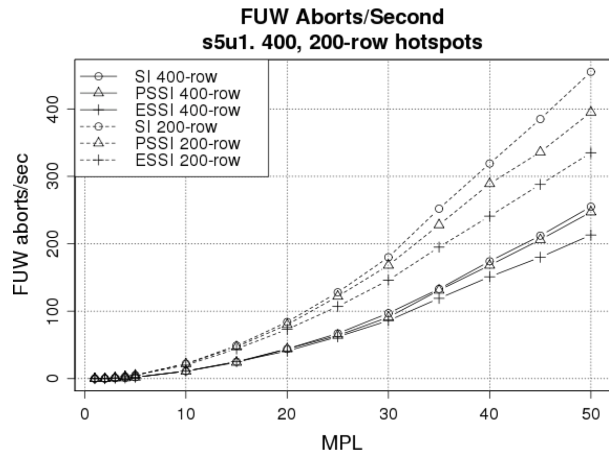

Figure 4.5 Serialization Aborts, s5u1, 400 $ 200 row hotspots

In Figure 4.4 the number of serialization aborts per second in the PSSI 200-row hotspot case at 50 MPL (using precise measurements underlying the Figure) is 310, compared to 640 in ESSI, a difference of 330 aborts per second. On the other hand, ESSI has 335 FUW aborts per second in the 200-row hotspot case at 50 MPL, fewer than the 395 FUW aborts per second for PSSI, a difference of 60 aborts per second. Thus the CTPS advantage that PSSI seems to have over ESSI Figure 4.2 at 50 MPL is 330 - 60 = 270 CTPS, but ESSI runs slightly more tranactions per second, so the actual CTPS advantage of PSSI over ESSI is 251. The reason for fewer

FUW aborts in ESSI is that a larger number of serialization aborts prevents some FUW aborts that might otherwise occur.

2) CTPS Difference for s3u1, s1u1 and s3u3

We repeat below the s5u1 CTPS measures for the 200-row hotspot, to compare it with successful commits per second for s3u1 and s1u1 with the same hotspot size.

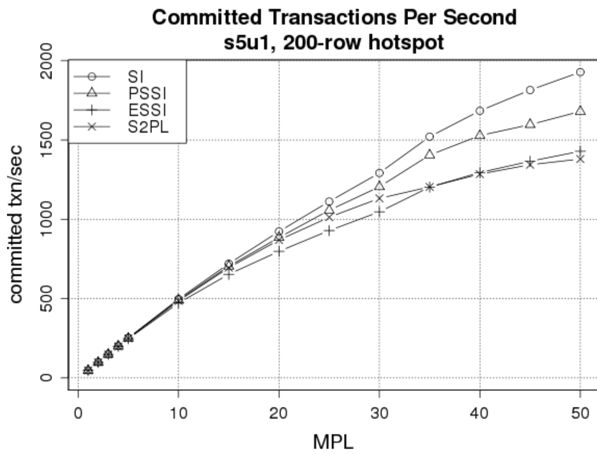

Figure 4.6 (=4.2) Committed Transactions Per Second, s5u1, 200 row hotspot

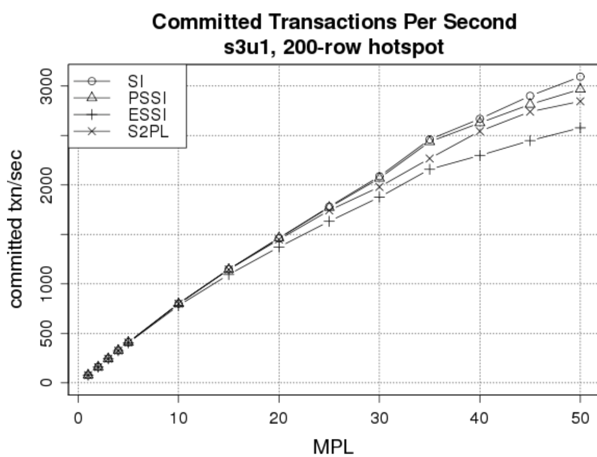

Figure 4.7 Committed Transactions Per Second, s3u1, 200 row hotspot

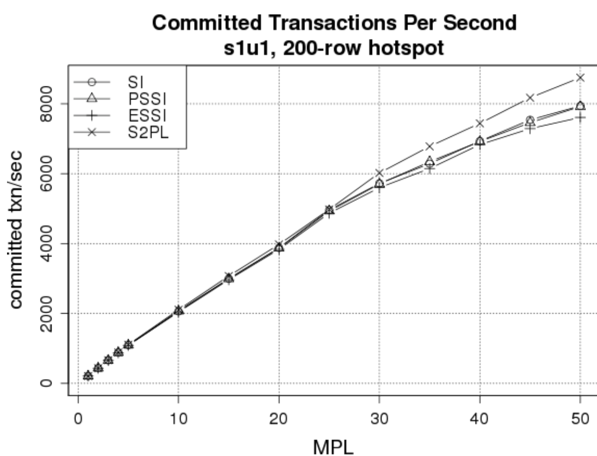

Figure 4.8 Committed Transactions Per Second, s1u1, 200 row hotspot

In Figure 4.6, the s5u1 case, the CTPS for PSSI at MPL 50 is 1680, while the CTPS for ESSI is 1429, a difference of 251 CTPS. In Figure 4.7, the s3u1 case, the CTPS for PSSI at MPL 50 is 2967, while the CTPS for ESSI is 2577, a difference of 390 CTPS. In Figure 4.8, the s1u1 case, the CTPS for PSSI at MPL 50 is 7921 while the CTPS for ESSI is 7610, a difference of 311 CTPS.

We also note that at s1u1, Figure 4.8, the performance of S2PL actually beats PSSI and ESSI at MPL 50. This occurs because with updates making up half of the data operations, PSSI and ESSI both have many more FUW aborts, but an FUW situation will only cause a lock wait in S2PL and both transactions might very well commit. We notice the same effect in Figure 4.9 measuring s3u3. The hotspot in this case is 1800 rows because the writes per transaction compared to s1u1 are tripled, meaning that the number of FUW aborts can be expected to rise by a factor of nine. Thus an 1800 row hotspot in this case corresponds to a 200-row hotspot in Figure 4.8 measuring CPTS for s1u1. Once again we note that S2PL beats PSSI (with SI overlaying it) and ESSI.

If we consider the Figures 4.8, 4.7 and 4.6 in increasing order by the number of reads per transaction, we note that with more read operations, the transactions have a larger number of inter-transactional dependencies with no increase in FUW aborts, causing a preponderance of serialization aborts where PSSI has a significant advantage over ESSI. The reason S2PL does more poorly in this sequence of Figures is that the number of read-write conflicts goes up with the larger number of reads, and thus there are more Waits.

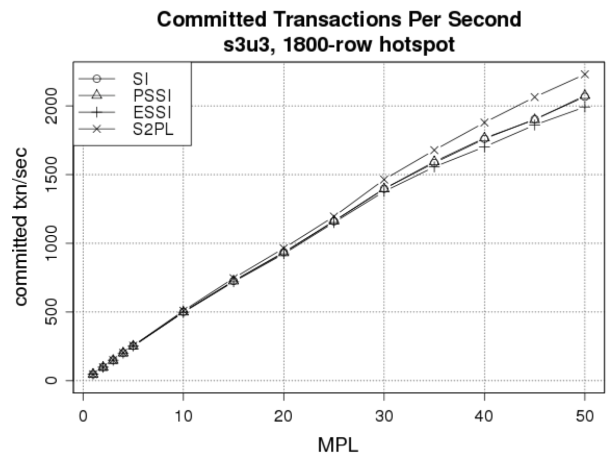

Figure 4.9 Committed Transactions Per Second, s3u3, 1800 row hotspot

We note that in the sibench benchmark of [5], S2PL was not competitive with SSI (our ESSI), but this benchmark had just two transactions, a read-only transaction that performed a full table scan and an update transaction that updated a single row. Thus there were no conflicts for ESSI, and in S2PL reads and writes took turns accessing rows, so throughput was dismal. ESSI throughput on the other hand was augmented by a design decision not to determine read snapshots until after the first data access of the transaction has occurred. This guaranteed that no FUW abort could occur even if two transactions performing only one update both updated the

same row. However we also note that in the TPC-C++ benchmark of [5], S2PL beat ESSI and SI as well, just as it does in our Figures 4.8 and 4.9.

In Figure 4.10, we provide the graph of committed transactions per second for s5u1 and a 200-row hotspot where a flush is performed at every commit.

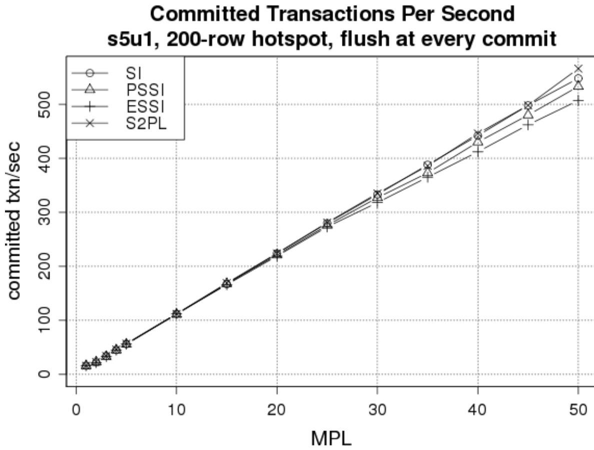

Figure 4.10 Commited Transactions per Second, s5u1, Flush on Commit

We note that the maximum CTPS measure for MPL 50 is a bit over 500 compared to the range 1400 to 2000 in Figure 4.2 (and equivalently, 4.6) where flushes were performed once a second instead of with every commit. We believe this demonstrates the approximate difference between a flush at every commit and a group commit where a flush occurs much less frequently and transactions can execute continuously. We also note that in this case S2PL runs slightly faster than SI and PSSI at MPL 50 instead of slowest in Figure 4.2 (and 4.6).

V. Future Work

Financial companies such as banks require an auditable record of account balances, i.e., a serial order derived from a transaction history. When an error is found in the total bank balance compared to total deposits and withdrawals at audit time, a bank must determine where the error occurred in the sequence of account changes. An error in a particular cash drawer might indicate a more serious problem. In S2PL schedulers, serializability guarantees that there is a strict serial order to row modifications, which corresponds to the order of commits by the transactions updating those rows.

Although we have shown that a serializable history is guaranteed in PSSI, the equivalent serial order of row updates often does not correspond to the transactional commit time. (This is true of ESSI and SSI as well.) This is because when a series of update transactions make up a lightning bolt in PSSI, the order of transaction commits is reversed in the serializably equivalent history once the lightning bolt becomes prunable. (See III.D for how this happens.) We show how this occurs with a lightning bolt of length 2 in Example 5.1.

Example 5.1. An obvious thought experiment that seems to determine the order of row updates is to consider a series of Read-Only transactions moving forward in time and reading all row values after each update transaction commits. But this approach fails because, as shown in [7], a Read-Only transaction can form a cycle with the update transactions of a lightning bolt of length two. Thus, the values shown by the read-only transaction are inconsistent. The history example given in [7], which we name H4 here, is this:

$$R_2(X_0)\,R_2(Y_0)\,R_1(Y_0)\,W_1(Y_1)\,C_1\,R_3(X_0)\,R_3(X_0)\,R_3(X_0)\,R_3(Y_1)\,C_3\,W_2(X_2)\,C_2$$

Figure 5.1. History H4, example given in Reference [7]

The anomalous cycle occurring in H4 is explained as follows: X and Y are data items in different rows representing a checking account balance X and a savings account balance Y, with $X_0 = 0$ and $Y_0 = 0$ initially. In what follows T_1 deposits 20 to Y and commits, and concurrently T_2 subtracts 10 from X, under the rule that a withdrawal is covered as long as X+Y > 0, but an overdraft with a penalty charge of 1 occurs if X+Y goes negative. Then T_3 is a read-only transaction that reads values X and Y and prints them for the customer. In H4, T_3 will show X = 0 and Y = 20, giving the impression that no penalty will take place. However when T_2 subtracts 10 from Y, the incremented value of X will not be seen, since the two transactions are concurrent, so we will end up with Y = 20 and X = -11. It was clearly wrong to allow T to show X = 0 and Y = 20, since the customer is misled. This is because of a cycle shown below in the SI-RW Diagram of H4 shown in Figure 5.2.

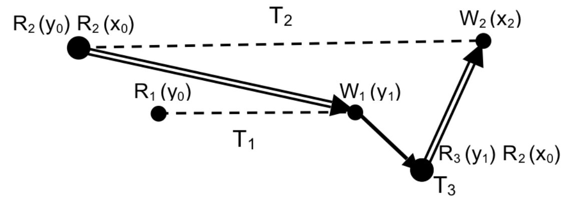

Figure 5.2. History H4: SI-RW Cycle with Read-Only Transaction T3

The example given in [7] seems rather contrived, but it is quite germane to considerations of the order in which updates occur in an SI-like history. We see that moving a read-only transaction to successive points in the history will not provide the proper values read for any equivalent serial history. In fact, the proper order of updates in Figures 5.1 and 5.2 is that $W_2(X_2)$ will occur first, setting X = -11, and then $W_1(Y_1)$ will occur, setting Y to 20. This becomes clear if we note that the anti-dependency from $R_2(Y_0)$ to $W_1(Y_1)$ means that T_2 must come before T_1 in an equivalent serial history. We believe we will be able to demonstrate in a future paper a commercially viable algorithm to provide an appropriate serial order for ESSI or PSSI while a transactional system is executing any challenging application.

References

[1] ANSI X3.135-1992, American National Standard for Information Systems — Database Language — SQL, November, 1992

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A Critique of ANSI SQL Isolation Levels". Proc. ACM SIGMOD 1995: pp. 1-10.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987. (ACM SIGMOD Anthology, Volume 4 Issue 1, or DVD2.)

[4] M. Cahill, U. Röhm and A. Fekete. "Serializable Isolation for Snapshot Databases". *Proc. ACM SIGMOD* 2008: 729-738.

[5] M. Cahill, U. Röhm and A. Fekete. "Serializable Isolation for Snapshot Databases". *ACM TODS* 2009: 20:1- 41.

[6] K.P. Eswaran, J. Gray, R. Lorie, I. Traiger, *"*The Notions of Consistency and Predicate Locks in a Database System*"* Res. Rep., RJ 1487, IBM Res. Lab., San Jose, Calif., 1974, later in CACM V19.11, pp. 624-633, Nov. 1974

[7] A. Fekete, E.J. O'Neil and P. E. O'Neil. "A Read-Only Transaction Anomaly Under Snapshot Isolation", *ACM SIGMOD Record*, Vol. 33, No. 3, Sept. 2004.

[8] A. Fekete, S. Goldrei, J. Asenjo, "Quantifying Isolation Anomalies", *Proc. VLDB* 2009.

[9] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, D. Shasha. "Making Snapshot Isolation Serializable". *ACM TODS*, 30(2), 492-528 (2005).

[10] J.N. Gray, R. A.Lorie, G.R. Putzolu and I.L. Traiger. "Granularity of Locks and Degrees of Consistency in a Shared Data Base". Presented 1975 in IFIP Working Conference. Accessible on http://research.microsoft.com/~Gray/

[11] Jim Gray. "Notes on Database Operating Systems. *an Advanced Course*," Bayer et. al. eds., Lecture notes in Computer Science 60, Springer-Verlag, 1978, pp. 393-481. Accessible on http://research.microsoft.com/~Gray/

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

[13] "InnoDB Transaction Model and Locking." http://dev.mysql.com/doc/refman/5.1/en/innodb-transaction-model.html Particular attention to 13.6.8.4. Record, Gap, and Next-Key Locks, and 13.6.9.5 Avoiding the Phantom Problem Using Next-Key Locking.

[14] 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.

[15] S. Jorwekar, A. Fekete, K. Ramamritham and S. Sudarshan. "Automating the Detection of Snapshot Isolation Anomalies". *Proc. VLDB* 2007: 1263–1274

[16] Mohan, C. 1995. "Concurrency control and recovery methods for B+-tree indexes: ARIES/KVL and ARIES/IM". In *Performance of Concurrency Control Mechanisms in Centralized Database System*s, V. Kumar, Ed. Prentice-Hall, 248-306.

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