PRECISELY SERIALIZABLE SNAPSHOT ISOLATION

A Dissertation Presented

by

STEPHEN A. REVILAK

Submitted to the Office of Graduate Studies,
University of Massachusetts Boston,
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2011

Computer Science Program

PRECISELY SERIALIZABLE SNAPSHOT ISOLATION

A Dissertation Presented

by

STEPHEN A. REVILAK

Approved as to style and content by:

Patrick O'Neil, Professor Emeritus
Chairperson of Committee

Elizabeth O'Neil, Professor
Member

Dan A. Simovici, Professor
Member

Samuel Madden, Associate Professor of EECS,
Massachusetts Institute of Technology
Member

Dan A. Simovici, Program Director
Computer Science Program

Peter A. Fejer, Chairperson
Computer Science Department

ABSTRACT

PRECISELY SERIALIZABLE SNAPSHOT ISOLATION

December 2011

Stephen A. Revilak, B.M., Berklee College of Music
M.S., University of Massachusetts Boston
Ph.D., University of Massachusetts Boston

Directed by Professor Emeritus Patrick O'Neil

Snapshot Isolation (SI) is a method of database concurrency control that uses timestamps and multiversioning, in preference to pessimistic locking. Since its introduction in 1995, SI has become a popular isolation level, and has been implemented in a variety of database systems: Oracle, Postgres, Microsoft SQL Server, and others. Despite the benefits that SI offers, one of the things it cannot provide is serializability. Past approaches for serializable SI have focused on avoiding dangerous structures (or essential dangerous structures). Dangerous structures are patterns of transaction dependencies that indicate the potential for a non-serializable execution; however, the presence of dangerous structures does not guarantee that a non-serializable execution will occur. Thus, avoiding dangerous structures is a conservative approach that may result in unnecessary transaction aborts. This dissertation presents Precisely Serializable Snapshot Isolation (PSSI), a set of serializability-providing enhancements to SI that utilize a more precise criterion for performing aborts. PSSI ensures serializability by detecting

dependency cycles among transactions, and aborting transactions to break such cycles. We have implemented PSSI in the open source MySQL/InnoDB database system, and our experimental tests demonstrate that PSSI's performance approaches that of SI, while minimizing the number of unnecessary transaction aborts.

ACKNOWLEDGEMENTS

To Pat and Betty O'Neil for their help and guidance in conducting this research. To Julie, for all of her patience and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

LIST OF ABBREVIATIONS

CTG	Cycle Testing Graph
CTPS	Committed Transactions per Second
DBMS	Database Management System
DSG	Dependency Serialization Graph
ESSI	Snapshot isolation with essential dangerous structure testing
FCW	First Committer Wins
FUW	First Updater Wins
iff	If and only if
LCB	Lock Control Block
LSN	Log Sequence Number
MPL	Multi-Programming Level (i.e., the number of concurrent transactions).
OCC	Optimisitic Concurrency Control
PSSI	Precisely Serializable Snapshot Isolation; an enhanced version of snapshot isolation that uses cycle testing to ensure serializability.
RID	Row identifier
SI	Snapshot Isolation
WAL	Write-ahead Logging

CHAPTER 1

INTRODUCTION

Snapshot isolation (SI) was presented as a new isolation level in [BBG95]. Rather than locking, SI relies on multiversioning and has the attractive property that readers do not block writers and writers do not block readers. The original definition of snapshot isolation provides a strong set of consistency guarantees, but falls short of ensuring full serializability. Several database systems offer snapshot isolation, including Oracle, Postgres, Microsoft SQL Server, and Oracle BerkleyDB.

The authors of [FLO05] provide a set of theoretical tools for reasoning about the correctness (i.e., serializability) of concurrent transactions running under SI. These tools model interactions between SI transactions as typed, ordered conflicts called *dependencies*. [FLO05] makes three more fundamental contributions: (a) it describes a characteristic structure that all non-serializable SI histories exhibit (called a *dangerous structure*), (b) it provides a static analysis technique that allows a DBA to determine whether a set of transactional programs can exhibit non-serializable behavior under SI, and (c) it provides methods to correct all programs where non-serializable behavior could occur.

The next significant advances came in [CRF08] and [CRF09]. These authors modified two open source database engines to detect dangerous structures [CRF08] and essential

dangerous structures [CRF09], and to abort transactions that caused them. Their contributions allow a DBMS using snapshot isolation to enforce serializability at runtime. However, the authors note that their approach can exhibit a significant number of false positives – transactions that will be aborted, even when no non-serializable behavior occurs. These false positives are the result of using dangerous structure testing as a criterion for correctness: all non-serializable SI histories contain at least one dangerous structure, but the presence of a dangerous structure does not necessarily mean that an SI history is non-serializable.

The work in this this dissertation is also rooted in [FLO05], but takes a different approach than [CRF08, CRF09]. Rather than relying on dangerous structures, we have focused on performing *cycle tests* over dependency graphs, and we have implemented this approach in the open-source MySQL/InnoDB database engine. Our implementation maintains a dependency graph over a suffix of an executing SI history, and aborts transactions when circular dependencies are found. Hence, our research provides a mechanism whereby a DBMS running snapshot isolation can guarantee serializability at runtime, with significantly fewer false positives than the approaches described in [CRF08, CRF09]. Experimental data in Chapter 5 shows that PSSI results in up to 50% fewer aborts than the strategy described in [CRF09] (see Figure 19, page 114), and PSSI's lower abort rate can translate into a 7–19% improvement in overall system throughput (see Figure 16, page 108).

1.1 Snapshot Isolation

In this section, we define snapshot isolation (SI) and give examples of common SI anomalies.

1.1.1 A Definition of Snapshot Isolation

Definition 1.1 (*Snapshot Isolation*): *Snapshot Isolation* associates two timestamps with each transaction T_i: start(T_i) and commit(T_i). These timestamps define T_i's *lifetime*, which can be represented by the interval $[\text{start}(T_i), \text{commit}(T_i)]$. T_i reads data from the committed state of the database as of start(T_i) (the *snapshot*), later adjusted to take T_i's own writes into account. T_i is allowed to commit if there is no committed transaction T_j such that (a) T_j was concurrent with T_i (i.e., their lifetimes overlap), and (b) T_j wrote data that T_i wishes to write. This feature is called the *first committer wins* (FCW) rule.

Database systems such as Oracle [Jac95] replace first committer wins with a validation test that occurs at the time of update, rather than at the time of commit. This strategy is called *first updater wins* (FUW), and explained in Definition 1.2.

Definition 1.2 (*First Updater Wins (FUW)*): Let T_i and T_j be two concurrent transactions, and let T_i update x. T_i's update takes a type of write lock on x, which is held for as long as T_i is active. T_i's write lock will not prevent concurrent transactions from reading x; it will only block concurrent writers. If T_j tries to update x while T_i is active, then T_j will be blocked until T_i commits or aborts. If T_i commits, then T_j will abort; if T_i aborts, then T_j can acquire its write lock on x and continue.

On the other hand, suppose T_i and T_j are concurrent, T_i updates x and commits, and T_j attempts to update x after commit(T_i). In this case, T_j will abort immediately, without entering lock wait.

Although the mechanics of FCW and FUW are different, both strategies serve the same purpose: they avoid *lost updates* by preventing concurrent transactions from writing the same data. Consider the classic lost update anomaly: $r_1(x)$, $r_2(x)$, $w_1(x)$, c_1, $w_2(x)$, c_2. FCW and FUW prevent this anomaly by aborting T_2, as T_2 wrote x, and x was written by a concurrent (and committed) transaction T_1. Note that this sequence of operations will produce deadlock under S2PL, causing one of the two transactions to abort. FCW and FUW also ensure that SI writes occur in a well-defined order.

Snapshot isolation has several attractive properties: readers and writers do not block each other (the only lock waits are those due to first updater wins). Additionally, snapshot isolation reads are "repeatable" in the sense that T_i's snapshot is defined by start(T_i), and this snapshot does not change during the course of T_i's lifetime (except, of course, to reflect T_i's own writes). Example 1.3 provides a simple illustration of these properties.

Example 1.3 (*A Simple SI History*): Consider history $H_{1.1}$:

$$H_{1.1}: \quad r_1(x_0, 10),\ w_2(x_2, 20),\ r_1(x_0, 10),\ c_1,\ c_2,\ r_3(x_2, 20),\ c_3.$$

In $H_{1.1}$, the notation $r_1(x_0, 10)$ means that T_i reads version x_0 and sees a value of 10; the notation $w_2(x_2, 20)$ means that T_2 writes version x_2, assigning a value of 20. When discussing multiversion histories, we use subscripts to denote different versions of a single data item, thus, x_2 refers to a version of x that was written by transaction T_2—the subscript indicates which transaction wrote that particular version of x. By convention, T_0 is a *progenitor* transaction which writes the initial values of all data items in the database.

In $H_{1.1}$, notice that T_1 read x and T_2 wrote x, but T_1 did not block T_2 and T_2 did not block T_1. Also note that while T_1 read x twice, it saw the same version (and the same value) each time: this behavior comes directly from the fact that T_1 sees a committed snapshot of the database as of start(T_1). Finally, note that T_3, which started after commit(T_2), read x_2, the version of x that T_2 wrote.

We should emphasize that snapshots are *not* created by copying data when a transaction T_i starts. Rather, the DBMS attaches timestamps to all data written, and uses those timestamps to compute snapshots. For example, if T_i writes x_i, then x's tuple header would typically contain the transaction number i. Later, if T_j tries to read x_i, then T_j can determine whether or not x_i is visible by comparing i to the list of transactions ids that were active when T_j started. This list of transaction ids is sometimes called an *invisibility list*, since T_i's modifications are invisible to T_j if i appears in T_j's invisibility list. When T_i commits, then i will not appear in invisibility lists for new transactions, and those new transactions will be able to read data that T_i wrote. Section 3.1 will discuss this process in detail.

1.1.2 Snapshot Isolation Anomalies

Snapshot isolation provides a strong set of consistency guarantees, and there are applications that do not exhibit anomalies when run under SI. For example, [FLO05] showed that the popular TPC-C benchmark [TPC10] will execute without anomaly when SI is chosen as the isolation level. Nonetheless, SI falls short of ensuring serializability as defined in [BHG87]. [JFR07] describes SI anomalies that were found in real-world applications running at IIT Bombay, which shows that SI's lack of serializability presents

more than just a theoretical problem. This section presents several common anomalies that can occur in SI. Perhaps the most common anomaly is *write skew*, which Example 1.4 illustrates.

Example 1.4 (*Write Skew*): Consider history $H_{1.2}$:

$$H_{1.2}: \quad r_1(x_0, 100),\ r_1(y_0, 100),\ r_2(x_0, 100),\ r_2(y_0, 100),\ w_1(x_1, -50),\ w_2(y_2, -50),\ c_1,\ c_2$$

We can interpret $H_{1.2}$ as follows: x and y represent two joint bank accounts, each with an initial balance of \$100; T_1 is withdrawing \$150 from account x while T_2 is withdrawing \$150 from account y. The bank provides overdraft protection, so a single account balance can become negative, as long as the sum of joint account balances does not fall below zero. T_1 and T_2 begin by computing the total balance of accounts x and y and find that balance to be \$200. $\$200 > \$150$, so each transaction believes that a \$150 withdrawal is permissible, but the end result is a negative total balance of $-\$100$.

Note that each transaction updates a different account balance, so the first committer (updater) wins rule does not apply here. Also note that if T_1, T_2 were executed serially (in any order), then the second transaction would detect the overdraft and no anomaly (i.e., non-serializable behavior) would occur.

A similar type of anomaly can occur with sets of records, where the sets are defined by where clause predicates. The set-based anomaly is referred to as *predicate write skew*, and illustrated in Example 1.5. (Example 1.5 is based on an example in [FLO05].)

Example 1.5 (*Predicate Write Skew*): Suppose we have an assignments table (date, employeeID, projectID, hours) which records assignments of employees to projects. In addition, we have a business rule stating that employees cannot be assigned more than

eight hours of projects during a single day. Snapshot isolation allows the following sequence to occur:

T_1: select sum(hours) from assignments where employeeID = 'e111' and date = '2010-09-01'; /* T_1 sees zero hours */

T_2: select sum(hours) from assignments where employeeID = 'e111' and date = '2010-09-01'; /* T_2 also sees zero hours */

T_1: insert into assignments values ('2010-09-01', 'e111', 'proj123', 5);

T_2: insert into assignments values ('2010-09-01', 'e111', 'proj456', 5);

T_1: commit;

T_2: commit;

In this example, each transaction issues a select statement (which involves a predicate read) to determine the number of hours assigned to employee e111 on 2010-09-01; each transaction sees no hours assigned, and assumes that it is okay to add a five-hour assignment. Both transactions succeed, giving employee e111 10 hours of assignments on 2010-09-01, violating the eight-hour limit. First committer (updater) wins offers no protection here, since the two inserts do not conflict on writes.

Write skew anomalies like the ones in Examples 1.4 and 1.5 can usually be avoided with constraint materialization and select for update [ACF08]. *Constraint materialization* involves using a single row to represent an aggregate constraint on a set of rows. For Example 1.5, we might create the table emp_hours (date, employeeID, hours), where each row represents the total number of hours assigned to a specific employee on a specific day. Any transaction that added an assignment would also have to update the corresponding

row in the emp_hours table, and first committer wins would prevent two concurrent transactions from changing the assignments for a particular employee on a particular day.

Select for update (called *promotion* in [ACF08]) treats reads as writes for the purpose of concurrency control. In Example 1.4, select for update would cause $r_1(x)$, $r_1(y)$ to take write locks on x and y. Depending on the DBMS and the ordering of conflicts, these write locks could either (a) cause a first updater wins error for a conflicting transaction, or (b) force a serial execution of the conflicting transactions.

Constraint materialization and select for update are effective in many situations. However, both techniques require changes to application code and/or the database schema, and such changes may be difficult to deploy in a running production system. Therefore, it would be preferable if a DBMS running at snapshot isolation could enforce serializability "out of the box", without having to rely on programmer workarounds. We believe that there is practical value in techniques that provide native support for serializable SI, thereby removing the burden from application developers.

1.2 Precisely Serializable Snapshot Isolation

This dissertation presents *Precisely Serializable Snapshot Isolation* (PSSI), an extension to snapshot isolation that performs cycle testing over dependency graphs. PSSI has the following characteristics:

- PSSI ensures that every execution of a set of valid transactional programs is serializable.

- PSSI is *precise* in the sense that it ensures serializability, while minimizing the number of unnecessary aborts. PSSI's precision is limited only the DBMS's ability to distinguish range intervals. (This limitation is discussed in Chapter 4.)

- PSSI only delays transactions in order to resolve first updater wins conflicts. If the DBMS implements first committer wins as the method for resolving concurrent write-write conflicts, then PSSI need not delay transactions at all.

- The throughput of PSSI, measured in committed transactions per second, generally exceeds the throughput of more conservative approaches. In some workloads, PSSI's throughput approaches that of ordinary snapshot isolation.

The remainder of this dissertation is organized as follows: Chapter 2 presents an in-depth look at the dependency theory introduced in [FLO05] and [Ady99]. In the process of doing so, we also explore the similarities and differences between dependency theory and the more traditional concepts of conflict serializability (e.g., as presented in [BHG87]). Chapter 2 presents a justification for using cycle testing as a criterion for correctness, and compares PSSI's cycle testing with two related approaches: dangerous structure testing [CRF09, CRF09] and optimistic concurrency control [KR81].

Chapter 3 presents a conceptual system design for PSSI; this chapter contains the fundamental data structures and algorithms upon which PSSI is built. Chapter 4 is a detailed case study in PSSI implementation; this chapter describes how we've taken (and in some cases, adapted) the algorithms from Chapter 3 to implement a PSSI prototype, using the InnoDB storage engine of MySQL 5.1.31 [ORA09].

Chapter 5 presents an experimental evaluation that compares the performance of four isolation levels: S2PL (strict two-phased locking), SI (snapshot isolation), ESSI (essential

dangerous structure testing, based our our implementation of [CRF09]), and PSSI. Chapter 6 describes areas for future work, and Chapter 7 concludes this dissertation.

CHAPTER 2

SNAPSHOT ISOLATION AND DEPENDENCY THEORY

Chapter 1 defined Snapshot Isolation and briefly discussed dependencies. Chapter 2 explores these topics in depth, and provides a justification for why dependency cycle testing is a valid basis for judging the serializability of SI histories.

An SI transaction T_i is defined by a lifetime $[\text{start}(T_i), \text{commit}(T_i)]$ where $\text{start}(T_i)$ represents T_i's start timestamp and $\text{commit}(T_i)$ represents T_i's commit timestamp. The simplest form of timestamp is a counter that increments each time a new $\text{start}(T_i)$ or $\text{commit}(T_i)$ is needed, so that timestamps form a strictly increasing sequence. Each SI transaction reads from a committed state of the database as of $\text{start}(T_i)$ (the snapshot, adjusted as necessary to take T_i's own writes into account) and T_i's writes becomes visible to new transactions that begin after $\text{commit}(T_i)$. This provides a particularly attractive model for reasoning about SI histories: we can treat all reads as occurring at $\text{start}(T_i)$ and all writes as occurring at $\text{commit}(T_i)$, regardless of the actual ordering of T_i's operations. For example, if T_1 performs the operations $r_1(x)$, $w_1(x)$, $r_1(y)$, $w_1(y)$, c_1, then we can treat T_1 as shown in Figure 1.

Figure 1 shows T_1 as two points on a timeline. $r_1(x)$, $r_1(y)$ appear on the left at $\text{start}(T_1)$, which effectively specifies that snapshot from which x and y are read. $w_1(x)$ and $w_1(y)$ appear on the right at $\text{commit}(T_1)$, which effectively specifies the point at which x_1

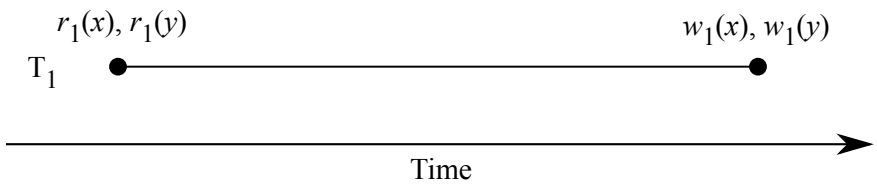

Figure 1: Reads and writes, relative to start(T_1) and commit(T_1)

and y_1 become visible to later transactions. T_i is permitted to read back its own writes, $w_i(x_i)$ followed by $r_i(x_i)$ for example, but in such cases, we assume that T_i simply remembers the value it wrote earlier.

Of course, transactions can interact with (and interfere with) one another. The next section will formalize dependencies [FLO05, Ady99]: the typed, ordered conflicts that model interactions among transactions in an SI history H.

2.1 Dependency Theory

This section presents a set of theoretical tools for reasoning about the correctness (i.e., serializability) of SI histories. It will be helpful to begin with a few definitions.

Definition 2.1 (*Concurrent Transactions*): Two SI transactions are *concurrent* if their lifetimes overlap, i.e., if

$$[\text{start}(T_i), \text{commit}(T_i)] \cap [\text{start}(T_j), \text{commit}(T_j)] \neq \emptyset$$

Definition 2.2 (T_i *installs* x_i): We say that T_i *installs* x_i if T_i writes x and commits. Once x_i is installed, x_i may be read by a transaction T_j where $\text{commit}(T_i) < \text{start}(T_j)$.

Definition 2.3 (*Immediate Successor*): We say that x_j is the *immediate successor* of x_i (or $x_j = \text{succ}(x_i)$), if T_i installs x_i, T_j installs x_j, $\text{commit}(T_i) < \text{commit}(T_j)$ and there is no transaction T_k such that T_k installs x_k and $\text{commit}(T_i) < \text{commit}(T_k) < \text{commit}(T_j)$.

Definition 2.4 (*Immediate Predecessor*): The definition of immediate predecessor is analogous to the definition of immediate successor. x_i is the *immediate predecessor* of x_j (or $x_i = \text{pred}(x_j)$), if and only if x_j is the immediate successor of x_i.

Definition 2.5 (*Dead Version*): If T_i deletes x_i and commits, then we say that x_i is a *dead version* of x. Dead versions are also referred to as *tombstones*.

Dependencies allow us to model the inter-transactional flow of data in an SI history H. There are five types of dependencies, which are listed in Definition 2.6.

Definition 2.6 (*Dependencies*):

1. We say that there is an *item write-read dependency* T_i-i-wr$\rightarrow T_j$ if T_i installs x_i and T_j later reads x_i.

2. We say that there is a *item write-write dependency* T_i-i-ww$\rightarrow T_j$ if T_i installs x_i and T_j installs x_i's immediate successor, x_j.

3. We say that there is an *item read-write dependency*, or *anti-dependency*, T_i-i-rw$\rightarrow T_j$ if T_i reads x_k and T_j installs x_k's immediate successor, x_j.

4. We say that there is a *predicate write-read dependency* T_i-p-wr$\rightarrow T_j$ if T_i installs x_i, and x_i changes the set of items retrieved by T_j's predicate. Here, $\text{commit}(T_i) < \text{start}(T_j)$ and T_j's predicate read takes into account x_i, or a later version of x.

5. We say that there is a *predicate read-write dependency*, or *predicate anti-dependency* T_i-p-rw$\rightarrow T_j$ if T_j installs T_j and T_j would change the set of rows retrieved by T_i. Here, start(T_i) $<$ commit(T_j) and T_i's predicate read takes into account some predecessor of x_j.

When it is clear from the context, or immaterial, we will omit the item ("i") or predicate ("p") designation and simply say T_i--rw$\rightarrow T_j$, T_i--wr$\rightarrow T_j$, or T_i--ww$\rightarrow T_j$.

Given the material in Definition 2.6 and the definition of snapshot isolation (Definition 1.1), we can make a number of immediate observations; these observations are given below as remarks.

Remark 2.7: Given T_i--wr$\rightarrow T_j$, T_j starts after T_i commits. Recall that T_j reads data from the committed state of the database as of start(T_j). Therefore, if T_j executes the operation $r_j(x_i)$, then we must have commit(T_i) $<$ start(T_j).

Remark 2.8: Given T_i--ww$\rightarrow T_j$, T_j starts after T_i commits. The first committer (updater) wins rule states that if two or more concurrent transactions write the same data, then only one of them may commit. Therefore, we must have commit(T_i) $<$ start(T_j); T_i and T_j cannot be concurrent. Note that this property creates a well-defined ordering of SI writes, and a well-defined ordering of data item versions.

Remark 2.9: Given T_i--rw$\rightarrow T_j$, T_i and T_j may, or may not be concurrent. If T_i reads x_k and T_j writes the immediate successor to x_k, then we can only conclude that start(T_i) $<$ commit(T_j). T_i--rw$\rightarrow T_j$ does not specify a relative ordering between commit(T_i) and start(T_j).

Remarks 2.7–2.9 show that dependencies are inherently tied to a temporal ordering of operations, where the arrow points from earlier to later in time. Thus, given T_i--rw$\rightarrow T_j$,

there is a read by T_i (which occurs at start(T_i)), a write of the same data item by T_j (which occurs at commit(T_j)), where the read precedes the write in serialization order. Similar statements can be made for write-read and write-write dependencies. This temporal ordering holds even if T_i and T_j are concurrent, as they might be with read-write dependencies. Dependencies lend themselves to two natural representations: directed graphs called dependency serialization graphs, and time-oriented graphs called SI-RW diagrams. Definitions 2.10 and 2.11 describe these representations.

Definition 2.10 (*Dependency Serialization Graph*): The *Dependency Serialization Graph* for a history H, or DSG(H), is a directed graph where each vertex is a committed transaction in H and there is a labeled edge from T_i to T_j iff there is a T_i--ww$\rightarrow T_j$, T_i--wr$\rightarrow T_j$, or T_i--rw$\rightarrow T_j$ dependency in H.

Definition 2.11 (*SI-RW Diagram*): SI-RW diagrams were introduced in [FLO05]. These diagrams present a time-oriented view of an SI history H, where each transaction T_i is represented by two vertices that are joined by a horizontal line, much like the timeline illustration shown in Figure 1. The left vertex shows start(T_i) (where all reads occur), and the right vertex shows commit(T_i) (where all writes occur). SI-RW diagrams depict T_i--ww$\rightarrow T_j$ and T_i--wr$\rightarrow T_j$ dependencies as solid diagonal arrows that run from commit(T_i) to commit(T_j) (ww dependencies), or from commit(T_i) to start(T_j) (wr dependencies). SI-RW diagrams depict T_i--rw$\rightarrow T_j$ dependencies as dashed arrows that run from start(T_i) to commit(T_j).

At this point, it would be useful to look at examples of dependency serialization graphs and SI-RW diagrams. For clarity, we will usually annotate edge labels with the

data item that caused the given dependency. For example, the edge label "ww (y)" means "a write-write dependency caused by a write of y followed by another write of y".

Example 2.12: Consider history $H_{2.1}$, which contains one instance of each dependency type. The dependency serialization graph for history $H_{2.1}$ is shown in Figure 2 and corresponding SI-RW diagram is shown in Figure 3.

$H_{2.1}$: $w_1(x_1), w_1(y_1), c_1, r_3(x_1), w_3(z_3), r_2(z_0), w_3(y_3), w_2(v_2), c_2, c_3$

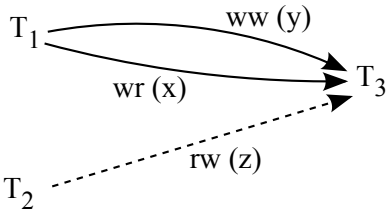

Figure 2: DSG for history $H_{2.1}$

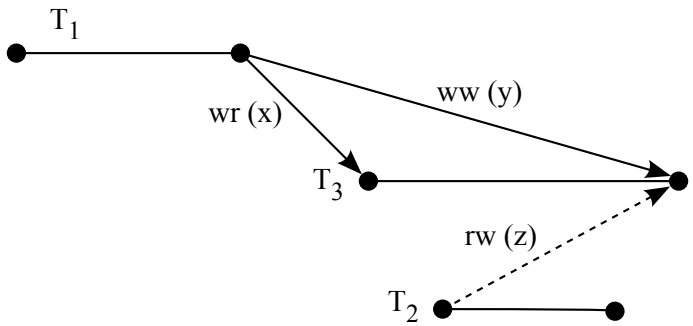

Figure 3: SI-RW diagram for history $H_{2.1}$

The DSG in Figure 2 shows the three transactions of history $H_{2.1}$, along with three labeled dependency edges: T_1--ww$\rightarrow T_3$, T_1--wr$\rightarrow T_3$, and T_2--rw$\rightarrow T_3$. We adopt [FLO05]'s convention of using separate edges when multiple dependencies occur between a pair of

transactions. An alternate convention is to use a single edge with multiple labels, for example, Figure 3.4 of [Ady99].

The T_1--wr$\rightarrow T_3$ dependency in Figure 2 comes from the operations $w_1(x_1)$ and $r_3(x_1)$ while T_1--ww$\rightarrow T_3$ comes from the operations $w_1(y_1)$ and $w_3(y_3)$. These dependencies tell us that T_3 must follow T_1 in any equivalent serial history. Similarly, the T_2--rw$\rightarrow T_3$ anti-dependency comes from the operations $r_2(z_0)$ and $w_3(z_3)$. (Recall from Example 1.3 that there is an assumed progenitor transaction T_0 which writes initial values for all data items in the database. $r_2(z_0)$ means that T_2 read the initial version of z.) Because T_2 read the immediate predecessor of z_3, we know that T_2 must come before T_3 in any equivalent serial history.

Dependency edges point from earlier to later in time, so DSG($H_{2.1}$) conveys a partial ordering where T_1 comes before T_3 and T_2 comes before T_3. $H_{2.1}$ is equivalent to two serial histories: $T_1 < T_2 < T_3$ and $T_2 < T_1 < T_3$.

The SI-RW diagram in Figure 3 shows the same transactions and dependencies as Figure 2, but Figure 3 conveys more temporal information. Like Figure 1, each transaction appears as two points on a timeline. Notice that the termination point for each dependency edge is consistent with the dependency type. For example, T_2--rw$\rightarrow T_3$ goes from start(T_2) to commit(T_3) and T_1--wr$\rightarrow T_2$ goes from commit(T_1) to start(T_2). This reflects the fact that reads happen at start(T_i) and writes happen at commit(T_i). Figure 3 also makes it easy to see that T_1 committed before T_2 started, and that T_2 and T_3 executed concurrently.

Example 2.13 (*SI-RW Diagram For Write Skew*): Chapter 1 contained an example of write skew (Example 1.4). For convenience, we reproduce the write skew history below as $H_{2.2}$, and Figure 4 shows the SI-RW diagram.

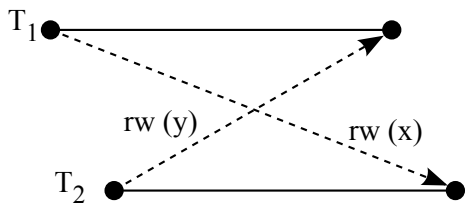

Figure 4: SI-RW diagram for history $H_{2.2}$, a write-skew anomaly

$$H_{2.2}: \quad r_1(x_0,100),\ r_1(y_0,100),\ r_2(x_0,100),\ r_2(y_0,100),\ w_1(x_1,-50),\ w_2(y_2,-50),\ c_1,\ c_2$$

Figure 4 illustrates a circular flow of data called a *dependency cycle*. Data flows forward in time along T_1--rw$\rightarrow T_2$, then backward in time along the duration of T_2, then forward in time along T_2--rw$\rightarrow T_1$. Intuitively, this tells us that $H_{2.2}$ is non-serializable because $T_1 < T_2 < T_1$: T_1 cannot follow itself in any serial history. The delay between start and commit allows the cycle to form, despite the fact that dependencies point from earlier to later in time.

Example 2.14 (*SI-RW Diagram for a Read-Only Anomaly*): This example comes from [FOO04], and demonstrates that a dependency cycle may contain read-only transactions. History $H_{2.3}$ represents a set of transactions on two bank accounts, x and y. Both accounts have an initial balance of zero when T_1 deposits \$20.00 into account y. T_2 withdraws \$10.00 from account x; from T_2's perspective, this withdrawal causes the total balance $(x+y)$ to fall below zero and T_2 imposes a one-dollar overdraft fee, giving account x a balance of \$-11. Finally, T_3 checks the total balance $(x+y)$, and finds that balance to be \$20.

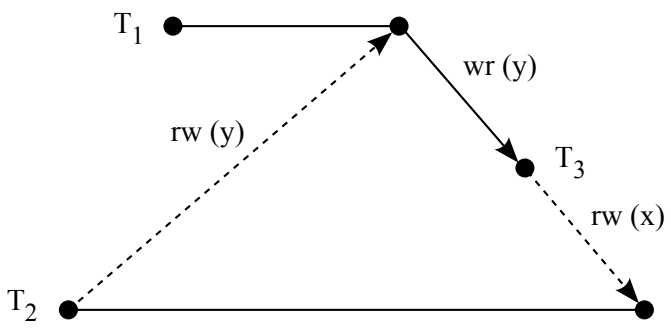

Figure 5: SI-RW diagram for history $H_{2.3}$, a read-only anomaly

$$H_{2.3}: \quad r_2(x_0,0),\ r_2(y_0,0),\ r_1(y_0,0),\ w_1(y_1,20),\ c_1,\ r_3(x_0,0),\ r_3(y_1,20),\ c_3,$$
$$w_2(x_2,-11),\ c_2$$

Why is $H_{2.3}$ not serializable? T_3 sees the effect of the twenty-dollar deposit, leading the account holder to believe that a subsequent ten-dollar withdrawal would bring the final balance to ten dollars, *without* incurring an overdraft fee. However, there was an overdraft fee and the final balance is nine dollars, not ten. Figure 5 shows the SI-RW diagram for history $H_{2.3}$.

T_3, the read-only transaction, appears as a "dot" in Figure 5, to emphasize the fact that commit(T_3) is contemporaneous with start(T_3), as far as the theoretical interpretation is concerned. T_3 sees the effects of T_1, but some of the data that T_3 reads is subsequently changed by T_2, and T_2 cannot observe the effects of T_1. Like our earlier write skew example, Figure 5 depicts a dependency cycle. This cycle runs from T_1 to T_3, then from T_3 to T_2, then backwards in time along the duration of T_2, and finally from T_2 to T_1. Note that if there would be no dependency cycle if T_3, the read-only transaction, were removed from $H_{2.3}$. Without T_3, $H_{2.3}$ would be a serializable history where $T_2 < T_1$.

Dependency cycles are a fundamental concept in the design of precisely serializable snapshot isolation. In the next section, we will compare dependencies and dependency serialization graphs to the more classic concepts of conflicts and conflict serialization graphs. We will also show that an SI history H is serializable if and only if its dependency serialization graph $\mathrm{DSG}(H)$ is acyclic.

2.2 Dependency Cycles and Serializability

Dependency theory has a great deal in common with classic serializability theory, but there are subtle differences between the two. The goal of this section is to explore these similarities and differences, and to justify the use of cycle testing as a method for determining whether an SI history H is serializable.

Conflict serializability states that two operations conflict if they operate on common data, and at least one of the two operations is a write. Two transactions T_i, T_j $(i \neq j)$ conflict of there is an operation $o_i \in \mathrm{T}_i$ and an operation $o_j \in \mathrm{T}_j$ such that o_i and o_j conflict. A history H can be represented by a *conflict serialization graph* $\mathrm{SG}(H)$, which is a directed graph where (1) T_i is a node in $\mathrm{SG}(H)$ iff T_i has committed in H and (2) there is an edge from $\mathrm{T}_i \rightarrow \mathrm{T}_j$ if one of T_i's operations o_i conflicts with one of T_j's operations o_j and o_i precedes o_j in the history H. The *serializability theorem* states that a history H is serializable iff its serialization graph $\mathrm{SG}(H)$ is acyclic. This material, along with proofs of the serializability theorem can be found in many database texts, such as [BHG87] and [OO01]. ([OO01] calls the structure a *precedence graph*, and denotes it as $\mathrm{PG}(H)$.)

Edge labels are the most obvious difference between $\mathrm{DSG}(H)$ and $\mathrm{SG}(H)$: $\mathrm{DSG}(H)$ edges are labeled with dependency types, while $\mathrm{SG}(H)$ edges are unlabeled. However, it

is not uncommon for DSG(H) and SG(H) to contain different *sets* of edges, depending on the specific operations involved. Example 2.15 illustrates this point.

Example 2.15 (*Comparison of* DSG(H) *and* SG(H)): Consider histories $H_{2.4}$ and $H_{2.5}$.

$H_{2.4}$: $w_1(x_1), c_1, w_2(x_2), c_2, w_3(x_3), c_3$

$H_{2.5}$: $w_1(v_1), w_1(x_1), c_1, w_2(x_2), w_2(y_2), c_2, w_3(y_3), w_3(z_3), c_3$

Histories $H_{2.4}$ and $H_{2.5}$ consist entirely of writes; therefore, all conflict serialization graph edges will come from write-write conflicts and all dependency serialization graph edges will come from write-write dependencies. SG($H_{2.4}$) contains three edges: $T_1 \rightarrow T_2$, $T_2 \rightarrow T_3$ and $T_1 \rightarrow T_3$. By contrast, DSG($H_{2.4}$) contains only two edges: T_1--ww$\rightarrow T_2$ and T_2--ww$\rightarrow T_3$. There is no T_1--ww$\rightarrow T_3$ edge in DSG($H_{2.4}$) because x_3 is not the immediate successor of x_1. Graphs SG($H_{2.4}$) and DSG($H_{2.4}$) have the same sets of nodes, but they are not isomorphic.

Now, let us consider $H_{2.5}$. In this case, SG($H_{2.5}$) and DSG($H_{2.5}$) are isomorphic. SG($H_{2.5}$) has edges $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$, while DSG($H_{2.5}$) has edges T_1--ww$\rightarrow T_2$ and T_2--ww$\rightarrow T_3$. Neither graph has an edge from T_1 to T_3, because T_1 and T_3 have disjoint write sets, and therefore do not conflict.

Although SG(H) and DSG(H) may not be isomorphic, we can show that the two graphs represent exactly the same set of paths.

Theorem 2.16: T_i is a node in SG(H) iff T_i is a node in DSG(H).

Proof: Theorem 2.16 follows from the definitions of conflict serialization graph and dependency serialization graph. T_i is a node in SG(H) iff T_i is committed in H, and T_i is a

node in DSG(H) iff T_i is committed in H. Therefore T_i is a node in SG(H) iff T_i is a node in DSG(H). □

Theorem 2.17: If there is a path from T_i to T_j in DSG(H), then there is a path from T_i to T_j in SG(H).

Proof: Much of Theorem 2.17 follows from Definition 2.6. There are three types of conflicts that can occur in H: read-write, write-read, and write-write. There are also three types of dependencies: T_i--rw→T_j is a read-write conflict where T_i's read precedes T_j's write in history H; T_i--wr→T_j is a write-read conflict where T_i's write precedes T_j's read in history H; and T_i--ww→T_j is a write-write conflict where T_i's write precedes T_j's write in H. Thus, each dependency represents a conflict between two transactions, T_i and T_j, where T_i's operation precedes T_j's operation in the history H. Therefore, each dependency edge from T_i to T_j in DSG(H) has a counterpart edge in SG(H). If every edge in DSG(H) has a counterpart edge in SG(H), then every path in DSG(H) has a counterpart path in SG(H). □

Theorem 2.18: If there is a path from T_i to T_j in SG(H), then there is a path from T_i to T_j in DSG(H).

Proof: Let $T_i \rightarrow T_j$ be an edge in SG(H). By definition, there is an operation $o_i \in T_i$ and an operation $o_j \in T_j$ such that o_i and o_j conflict and o_i precedes o_j in the history H. Since o_i and o_j conflict, we know that both operations affected a common data item x. There are two cases two consider: $T_i \rightarrow T_j$ is an edge in DSG(H) or $T_i \rightarrow T_j$ is not an edge in DSG(H).

Case 1. If $T_i \rightarrow T_j$ is an edge in DSG(H), then T_i--rw$\rightarrow T_j$, T_i--wr$\rightarrow T_j$, or T_i--ww$\rightarrow T_j$. Regardless, there is a path of length one between T_i and T_j in SG(H) and a path of length one between T_i and T_j in DSG(H).

Case 2. If $T_i \rightarrow T_j$ is not an edge in DSG(H), then one of the following three cases must hold:

1. T_i wrote x_i, T_j wrote x_j and x_j is not the immediate successor of x_i. This is a write-write conflict, but not a T_i--ww$\rightarrow T_j$ dependency.

2. T_i wrote x_i, T_j read x_k, and $x_k \neq x_i$. This is a write-read conflict, but not a T_i--wr$\rightarrow T_j$ dependency.

3. T_i read x_k, T_j wrote x_j and x_j is not the immediate successor of x_k. This is a read-write conflict, but not a T_i--rw$\rightarrow T_j$ anti-dependency.

All three cases imply the existence of one or more versions of x that were installed after $o_i(x)$ and before $o_j(x)$. The existence of these versions tells us that there is a sequence of transactions $T_{k_1} \ldots T_{k_n}$ where $T_{k_1} = T_i$, $T_{k_n} = T_j$, and there is a read-write, write-read, or write-write dependency between $T_{k_p} \rightarrow T_{k_{p+1}}$ for $1 \leq p \leq (n-1)$. In the terminology of [Ady99, Section 3.1.4] the edges in DSG(H) represent *direct* conflicts whereas the edges in SG(H) may represent *transitive* conflicts. Nonetheless, for every edge in SG(H), there is a path (of equal or greater length) in DSG(H). Therefore, if there is a path from T_i to T_j in SG(H), then there is also a path from T_i to T_j in DSG(H). □

Theorem 2.19: DSG(H) is acyclic iff SG(H) is acyclic.

Proof: The proof follows from Theorems 2.17 and 2.18. Theorem 2.17 tells us that any path in DSG(H) is also a path in SG(H), while Theorem 2.18 tells us that any path in

SG(H) is also a path in DSG(H). Therefore, if SG(H) contains a cycle then DSG(H) must also contain a cycle; if SG(H) does not contain a cycle, then DSG(H) cannot contain a cycle. □

Theorem 2.19 leads immediately to the desired conclusion, that H is serializable iff DSG(H) is acyclic.

Theorem 2.20 (*Serializability Theorem for Dependency Serialization Graphs*): An SI history H is serializable iff DSG(H) is acyclic.

Proof: The serialization theorem states that a history H is serializable iff SG(H) is acyclic. Theorem 2.19 states that SG(H) is acyclic iff DSG(H) is acyclic. Therefore, H is serializable iff DSG(H) is acyclic. □

Theorem 2.20 shows that, from the point of view of ensuring serializability, there is *no difference* between performing cycle tests over conflict serialization graphs and performing cycle tests over dependency serialization graphs. This justifies the use of cycle testing over dependency serialization graphs as a mechanism for detecting non-serializable SI histories.

2.3 Dangerous Structures

Section 2.2 showed that a snapshot isolation history H is serializable if and only if its dependency serialization graph DSG(H) is acyclic. However, one can make other characterizations about non-serializable SI histories. One particularly noteworthy characterization was proven in [FLO05]; we summarize their findings in Definition 2.21 and Theorem 2.22.

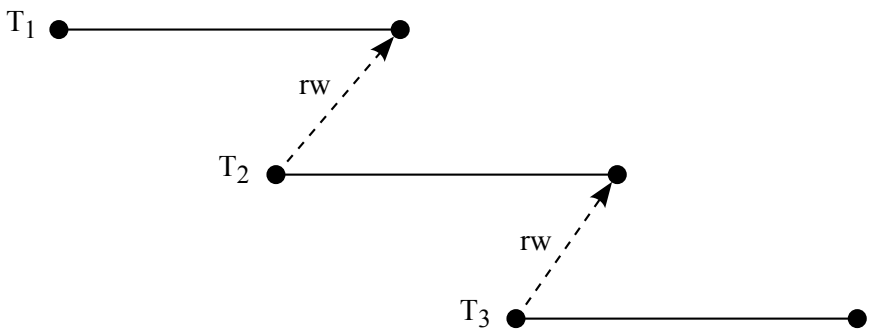

Figure 6: SI-RW diagram for a dangerous structure

Definition 2.21 (*Dangerous Structure*): A *dangerous structure* in a dependency serialization graph is a sequence of three transactions T_1, T_2, T_3 (perhaps with $T_1 = T_3$) such that T_1 and T_2 are concurrent, T_2 and T_3 are concurrent, and there are anti-dependency edges T_2--rw$\rightarrow T_1$ and T_3--rw$\rightarrow T_2$.

Theorem 2.22 (*Dangerous Structure Theorem*): Every non-serializable SI history *H* contains a dangerous structure.

In the case where $T_1 = T_3$, the dangerous structure will be cycle of length two. For example, Figure 4's write skew anomaly is a cycle of length two, and this cycle also happens to be a dangerous structure. In the more general case, T_1, T_2, and T_3 are three distinct transactions, and a dangerous structure represents a situation that *could* develop into a complete cycle. Let us examine Figure 6, which illustrates a dangerous structure with three distinct transactions.

The history represented by Figure 6 is serializable, and equivalent to the serial history T_3, T_2, T_1. Nonetheless, it is easy to see how this history could develop into a complete

cycle. For the sake of discussion, suppose that T_3 were still active, and that T_1, T_2 had committed. T_3 could cause a cycle of length three in any one of the following ways:

1. T_3 could write data that T_1 wrote, causing T_1--ww$\rightarrow T_3$. Note that T_1 and T_3 are not concurrent, so first committer wins offers no protection here.

2. T_3 could read data that T_1 wrote, causing T_1--wr$\rightarrow T_3$.

3. T_3 could write data that T_1 read, causing T_1--rw$\rightarrow T_3$.

Theorem 2.22's main contribution comes from showing us how cycles form – it is necessary to have two consecutive anti-dependencies between pairwise concurrent transactions. Theorem 2.22 also provides the basis another mechanism of ensuring serializability in SI: *dangerous structure testing*. Instead of disallowing dependency cycles, one could elect to disallow dangerous structures; this was the approach was taken by [CRF08, CRF09]. Ultimately, dangerous structure testing is more *conservative* than cycle testing, since dangerous structure testing can abort transactions which create the potential for a cycle, before the cycle has fully formed.

In the context of a DBMS implementation, [CRF09] makes a useful refinement to Theorem 2.22, by observing that it is only necessary to prevent dangerous structures where T_1 commits first. We call this an essential dangerous structure, as defined in Definition 2.23. (We also note that the term *essential dangerous structure* was introduced in [ROO11]; it was not used by the authors of [CRF09].)

Definition 2.23 (*Essential Dangerous Structure*): An *essential dangerous structure* is a dangerous structure (Definition 2.21) where T_1 commits first.

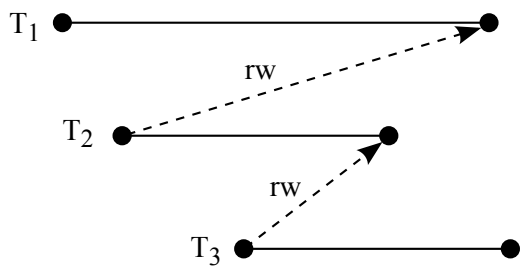

Figure 7: SI-RW diagram for a non-essential dangerous structure

The dangerous structure in Figure 6 is also an essential dangerous structure. By contract, Figure 7 shows a non-essential dangerous structure (where T_1 does not commit first).

Dangerous structure testing and essential dangerous structure testing can be used as the basis for two named isolation levels: SSI and ESSI, which are defined in Definition 2.24

Definition 2.24 (*SSI, ESSI*): *SSI* is an isolation level which aborts any transaction T_i that causes a dangerous structure to form. *ESSI* is an isolation level which aborts any transaction T_i that causes an essential dangerous structure to form.

[CRF09] reports that essential dangerous structure testing (ESSI) leads to fewer aborts than dangerous structure testing (SSI), which means that ESSI is a less conservative approach. However, ESSI is still more conservative than PSSI's cycle testing, and in fact, there is a strict progression from SSI to ESSI to PSSI: SSI is the most conservative and least precise, while PSSI is the least conservative and most precise.

We will return to essential dangerous structure testing in Chapter 5, which contains the results of our performance study.

2.4 Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) was first defined in [KR81]. OCC assumes that conflicts are rare, and seeks to improve concurrency by eliminating the use of locks. OCC and snapshot isolation share some common features, so it is worth discussing the similarities and differences of the two approaches.

Optimistic concurrency control treats transactions as having three phases: a read phase, a validation phase, and a write phase. The *read phase* corresponds to the user portion of a transaction (which may include modifications to local, per-transaction copies of data), while the *validation phase* is used to determine whether a given transaction is allowed to commit. Transactions use the *write phase* to copy their local modifications into the global database area, making those writes visible to other transactions. This three-phase approach implies a form of copy-on-write: if T_i wishes to write x, then T_i will make a local copy of x and modify the local copy; the local copy becomes globally visible after T_i's write phase, in which T_i's local modifications are copied back into the global database area. Of course, this also means that T_i's reads have to examine local modifications before considering data in the global database area (i.e., T_i must be able to observe the effects of its own writes).

Like OCC, PSSI can be viewed as dividing transactions into three phases. PSSI's read phase consists of the user portion of the transaction – everything between the begin and commit statements – while PSSI's validation phase is built around a cycle test. PSSI's write phase consists of removing the committing transaction from the transaction manager's active transaction list, as described at the end of Section 1.1.1.

The most significant differences between OCC and PSSI lie in the validation phase. PSSI's validation phase focuses on determining whether commit(T_i) would cause a dependency cycle, while OCC takes a more conservative approach, which is outlined below. Let readset(T_i) denote T_i's read set, let writeset(T_i) denote T_i's write set, and let T_j denote any transaction (concurrent or not) that committed prior to T_i's validation phase. According to [KR81], a committing T_i must satisfy one of the following three conditions in order to pass validation:

1. T_j finishes its write phase before T_i starts its read phase.

2. T_j completes its write phase before T_i starts its write phase, and writeset(T_j) $\cap$ readset(T_i) $= \emptyset$.

3. T_j completes its read phase before T_i completes its read phase and writeset(T_j) $\cap$ (readset(T_i) $\cup$ writeset(T_i)) $= \emptyset$.

From a database users' standpoint, the validation and write phases happen during commit; [KR81] treat them separately in order to explore mechanisms for allowing concurrent commits ([KR81] assumes that the write phase may take some time, and the authors wanted to avoid serializing commits around the write phase). For PSSI, we can think of the validation phase as starting when the database user issues a commit statement, and we can think of the write phase as completing when the database acknowledges the commit.

Condition 1 allows any serial execution of transactions. More specifically, condition 1 states that T_j's actions cannot prevent T_i from committing, if T_j commits before T_i starts. Condition 2 states that T_i cannot commit if T_i read data that was modified by a concurrent (but already committed) transaction T_j. Condition 2 effectively prohibits T_i--rw$\rightarrow T_j$

dependencies between concurrent transactions, which is something that PSSI allows. Like condition 2, condition 3 also prohibits T_i--rw$\rightarrow T_j$ dependencies. In addition, condition 3 provides an analog to the first committer wins rule: T_i is not allowed to commit if a concurrent (but committed) T_j wrote data that T_i is attempting to write.

Despite these differences, there are philosophical similarities between OCC and SI: both mechanisms attempt to improve concurrency through a reduction in locking. [KR81] noted that optimistic methods could be advantageous of disadvantageous, depending on the specifics of the database workload. The same characterization could also be made of PSSI.

2.5 Summary

This chapter presented dependencies, dependency serialization graphs, SI-RW diagrams, and a justification for using cycle tests do determine whether an SI history H is serializable. We have looked at examples of dependency serialization graphs, and we have seen that these structures have much in common with their counterparts from classic serializability theory, conflict serialization graphs. Finally, we have examined two approaches related to PSSI: dangerous structure testing and optimistic concurrency control.

Chapter 3 will present a conceptual design form implementing PSSI in a DBMS, but without focusing on any particular implementation. Chapter 4 will present our implementation of this design in MySQL's InnoDB storage engine.

CHAPTER 3

A CONCEPTUAL DESIGN FOR PSSI

This chapter presents a conceptual design for PSSI. The presentation focuses on PSSI's high-level features, saving the low-level details for an implementation case study in Chapter 4, where we describe the process of adding PSSI support to MySQL 5.1.31's InnoDB storage engine. This chapter discusses strategies for finding dependencies between transactions, explains how and when cycle testing is done, and provides a method for determining when a committed transaction T_i cannot be part of any future cycle. We begin the chapter with one of snapshot isolation's more germane issues: how to implement multiversioning. Multiversioning is not a new area of research, but it is a fundamental part of any SI system design.

This chapter frequently uses the term *data item*. Although the term is ambiguous, it has enjoyed common use in database literature (e.g., [Ady99], [FLO05], [CRF09], [FOO04], and others). We use the term "data item" to refer to rows and secondary (non-clustered) index entries, but not to units of larger granularity, such as pages, tables, or files. In particular, Section 3.3's lock table algorithms assume that the same treatment is applied to *both* rows and secondary indexes, and the term data item helps to capture this.

3.1 Timestamps and Versioning

When T_i reads x, T_i sees the most recent version of x that was committed prior to start(T_i) [ROO11]. This means that T_i observes a particular *state* of x (the *version*), which came into existence at a well-defined point in time, prior to start(T_i). SI allows several versions of x to exist simultaneously, and indeed, we have already seen an example of this in history $H_{1.1}$, which is repeated below for reference.

$H_{1.1}$ $r_1(x_0, 10)$, $w_2(x_2, 20)$, $r_1(x_0, 10)$, c_1, c_2, $r_3(x_2, 20)$, c_3.

In $H_{1.1}$, $w_2(x_2)$ is followed by $r_1(x_0)$, implying that x_2 and x_0 must co-exist, and the DBMS must make these versions available to transactions that require them. This example illustrates how SI imposes operational requirements on the way that versions are managed, so it is only natural to begin the discussion with versioning.

Section 1.1.1 described a technique for implementing SI, based on transactions ids and invisibility lists. (Recall that T_i's invisibility list is the list of transactions ids for T_j that were active at start(T_i); these T_j are concurrent with, and invisible to T_i.) This section presents the invisibility list technique in algorithmic form. Each SI transaction is defined by its lifetime $[\text{start}(T_i), \text{commit}(T_i)]$, and each T_i is guaranteed to receive a unique start timestamp, and (assuming T_i commits) a unique commit timestamp. The uniqueness of start timestamps makes it possible to use them as transaction ids.

Start-Transaction (Algorithm 1) illustrates the process of assigning a new transaction id (i.e., a new start timestamp), initializing T_i's invisibility list, and initializing T_i's commit timestamp. Algorithm 1's global_timestamp_counter is a global counter that is incremented each time a new timestamp is required. The construction of T_i's invisibility list makes the rudimentary assumption that the database transaction manager maintains

```
1: procedure Start-Transaction
2:     T_i = new transaction
3:     T_i.txid = ++global_timestamp_counter
4:     T_i.invisibility_list = { j | j is the transaction id of an active transaction }
5:     T_i.commit_ts = ∞
6:     Rest of transaction initialization steps
7: end procedure
```

Algorithm 1: Assignment of T_i's start timestamp and invisibility list

some form of active transaction list, so that T_i.invisibility_list is simply a copy of those active transaction ids, made at start(T_i). T_i.invisibility_list is fixed, and does not change during T_i's lifetime. If T_j starts while T_i is still active, then it is not necessary to add j to T_i.invisibility_list, because the timestamp relationship start(T_i) $<$ start(T_j) $<$ commit(T_j) prevents T_j's writes from being visible to T_i.

T_i.commit_ts is initialized to a placeholder value of ∞, and this placeholder value is used for as long as T_i is active. T_i's actual commit timestamp is assigned during the commit process, as shown in Commit-Transaction (Algorithm 2). T_i.commit_timestamp is drawn from the same global_timestamp_counter as T_i.txid. The last line of Algorithm 2 removes T_i from the list of active transactions, which prevents T_i's transaction id from appearing in future invisibility lists. Note that Algorithm 2 is only a skeletal outline of the commit process; line 3 consists of many steps, which will be discussed throughout this chapter.

Algorithms 1 and 2 show that timestamps are assigned during two events: when T_i starts and when T_i issues a commit statement. These are the only events that increment global_timestamp_counter. Note that the treatment of global_timestamp_counter is

```
1: procedure Commit-Transaction(transaction T_i)
2:     T_i.commit_timestamp = ++global_timestamp_counter
3:     Rest of commit process, including ensuring that T_i is serializable
4:     Remove T_i from the active transaction list
5: end procedure
```

Algorithm 2: Assignment of T_i's commit timestamp

consistent with SI's general semantics, namely, that reads occur at start(T_i) and writes becomes visible to transactions that begin after commit(T_i).

Our next objective is to show how T_i can use its invisibility list to choose the correct version of x during a read operation, after accessing x via primary key lookup, or after accessing x via table scan. In showing this, we make three assumptions: (1) the DBMS stores versions in a way that allows them to be accessed from newest to oldest, (2) for each version x_j of x, x_j's header contains the writer's transaction id as x_j.txid, and (3) for each version x_j of x, x_j's header contains a boolean field x_j.is_deleted that indicates whether x_j is a deleted ("dead") version.

Versioned-Read (Algorithm 3) shows how T_i determines which version of x is in its snapshot (if any). The input parameter "vlist" represents the list of versions of data item x, ordered from newest to oldest. The if-statements in lines 4 and 7 skip over x_j that are too new for T_i to see. The first x_j that fails to satisfy these conditions is the version of x in T_i's snapshot; line 10 stores this version in the variable "found" for further examination. It's possible that "found" – the x_j in T_i's snapshot – represents a deleted version, or that x is not part of T_i's snapshot at all. Line 14 tests these cases, returning a NOT_FOUND condition if appropriate. Otherwise, "found" is the x_j in T_i's snapshot, and this version is

```
1:  procedure Versioned-Read(Transaction T_i, Version List vlist)
2:      found = NULL                                   ▷ variable to hold the return value
3:      for x_j ∈ vlist do                  ▷ iterate from newest version to oldest version
4:          if x_j.txid > T_i.txid then
5:              continue                ▷ commit(T_j) > start(T_i). x_j is not visible to T_i
6:          end if
7:          if x_j.txid ∈ T_i.invisibility_list then
8:              continue                                ▷ T_j's writes are not visible to T_i
9:          end if
10:         found = x_j                                  ▷ This x_j is in T_i's snapshot
11:         break
12:     end for
13:                                                    ▷ examine candidate return value
14:     if (found == NULL) or (found.is_deleted) then
15:         return NOT_FOUND
16:     end if
17:     return found
18: end procedure
```

Algorithm 3: Versioned reads: how T_i selects the appropriate version of x

returned to T_i. Versioned-Read implies that PSSI employs special handling for deleted data items; we will discuss this topic in Section 3.2.

3.1.1 Version Storage

Algorithm 3 assumed a facility for iterating over versions in reverse-chronological order. This sections presents *chaining*, a technique for organizing versions in a way that supports reverse-chronological iteration.

Chaining places versions in a linked list: each version x_j contains a header field x_j.pred, and x_j.pred "points to" x_j's immediate predecessor. For the sake of discussion, let

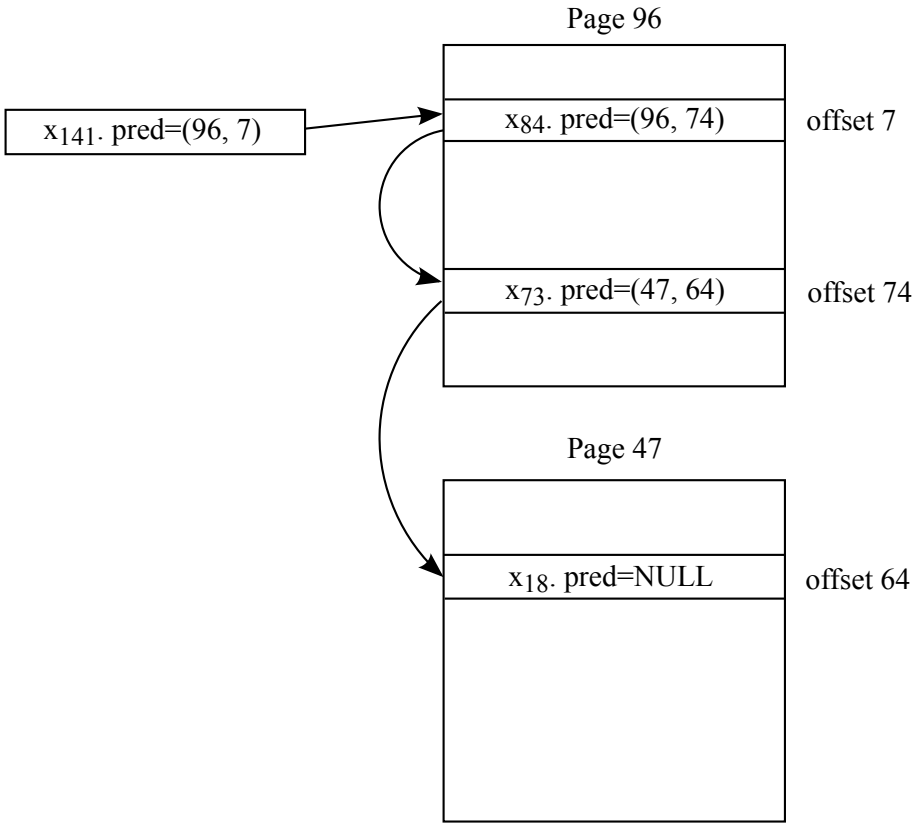

Figure 8: Four versions of x, chaining approach

us assume that x_j.pred consists of a page number and an offset. Figure 8 illustrates the idea behind chaining. In Figure 8, x_{141} is the newest version of x, stored by whatever means the DBMS uses for managing the physical storage of rows. $x_{141}.\text{pred} = (96, 7)$ indicates that x_{141}'s immediate predecessor is located on page 96, offset 7. Following x_{141}.pred brings us to x_{84}, whose immediate predecessor is located on page 96, offset 74. Following x_{84}.pred takes us to x_{73}, whose immediate predecessor is located on page 47, offset 64. Following x_{73}.pred takes us to x_{18}; x_{18}.pred is NULL, which means that there are no earlier versions of x.

Chaining gives the DBMS the flexibility to employ separate strategies for storing current and old versions of data items. For example, the DBMS could store pred(x_j) in a compressed format in which pred(x_j) contains only those column values where pred(x_j) differs from x_j.

3.1.2 Remarks on Version Management Strategies

Section 3.1 described a versioning strategy that is based on transaction ids and invisibility lists. Several real-world SI implementations utilize this strategy, including Postgres [PG10, Section 9.23] and Netezza [HHS07]. InnoDB's MVCC also uses transaction ids and invisibility lists.

Section 3.1.1 presented chaining as a method for physically organizing versions, and for allowing T_i to locate the version of x in its snapshot. This chaining presentation was based on InnoDB's MVCC implementation. Another technique stores two transaction ids in each version header: x_i.creator_txid, and x_i.deleter_txid. The creator_txid is the transaction id of the transaction that installed x_i, and the deleter_txid is the transaction id of the transaction that deleted x_i (perhaps by installing a newer version of x). For example, if transaction T_{35} inserts x, then this creates the version x_{35} where x_{35}.creator_txid = 35 and x_{35}.deleter_txid = NULL. If T_{47} later updates x, then T_{47} would set x_{35}.deleter_txid = 47 (i.e., T_{47} "deletes" x_{35} by creating a newer version of x), and T_{47} would also install x_{47}, with x_{47}.creator_txid = 47 and x_{47}.deleter_txid = NULL. The combination of these two transactions ids allows T_i to judge the visibility of individual versions, without having them organized in a linked list. We did not use the creator_txid, deleter_txid technique in our research, but we mention the technique because it is used by

several SI implementations, including Postgres [PG10, Section 54.5] and Netezza [HHS07].

3.2 Zombie Transactions

Chapter 2 showed how a committing transaction T_i can form dependency cycles with already-committed transactions. This highlights the need for retaining information about transactions after they commit, and we call special attention to this need in History $H_{3.1}$.

$$H_{3.1}: \quad r_1(x_0), r_2(z_0), w_1(y_1), c_1, r_2(y_0), w_2(x_2), c_2$$

History $H_{3.1}$ is an example of write skew. There is a T_1--rw$\rightarrow T_2$ dependency caused by $r_1(x_0)$ and $w_2(x_2)$, and there is an T_2--rw$\rightarrow T_1$ dependency caused by $r_2(y_0)$ and $w_1(y_1)$. Taken together, this pair of dependencies creates a cycle. The important point of history $H_{3.1}$ is the following: neither dependency existed when commit(T_1) occurred; it was only *after* commit(T_1) that T_2 read y and wrote x, causing the cycle to form.

To detect such cases, PSSI needs to "remember" information about a transaction T_i for some period of time after T_i commits. Specifically, T_i's actions must be remembered for as long as there is the potential for T_i to become part of a cycle. While this potential exists, T_i is referred to as a *zombie transaction*, as defined in Definition 3.1.

Definition 3.1 (*Zombie Transaction*): A *zombie transaction* T_i is a committed transaction that has the potential to become part of a future cycle. In order to detect cycles, PSSI must retain information about T_i's data accesses, and the dependency edges incident to T_i. When it is no longer possible for T_i to become part of a cycle, then T_i ceases to be a zombie transaction, and PSSI can discard information about T_i's data accesses, and the dependency edges incident to T_i.

In history $H_{3.1}$, T_1 becomes a zombie after commit(T_1), and remains a zombie until commit(T_2) completes. Section 3.4.1 provides algorithms for determining whether a transaction T_i is a zombie, and for discarding information about T_i when T_i is no longer a zombie.

Zombie transactions influence PSSI's handling of deleted data items. Suppose that T_d deletes x, creating the dead version x_d. PSSI must retain x_d's immediate predecessor pred(x_d) for as long as there is an active transaction T_i where pred(x_d) appears in T_i's snapshot; but how long should x_d itself remain in the database? x_d must remain in the database for as long as T_d is a zombie transaction. The reasoning behind this is as follows: if T_j "reads" x_d while T_d is a zombie, then T_j must make note of the operation $r_j(x_d)$. x_d does not affect the results of of T_j's query, but PSSI needs to know about the conflicting operations $w_d(x_d)$ (the delete) and $r_j(x_d)$ in order to detect the dependency T_d--wr$\rightarrow T_j$. Once T_d is no longer a zombie, then PSSI can garbage-collect the deleted data item x_d. Example 3.2 illustrates this idea.

Example 3.2 (*Treatment of Deleted Data Items*): Suppose we have the table accounts = (acctid, type, branch, balance) with the row x = (a1234, checking, Boston, 100.00). Now, consider the following set of transactions:

T_1: select count(*) from accounts where type = 'checking'; -- count includes x

T_2: delete from accounts where acctid = 'a1234'; -- delete x

T_2: commit;

T_3: select count(*) from accounts where branch = 'Boston'; -- does not include x

T_3: commit;

T_1: commit;

In this example, T_2 deletes x, and x matches the where clauses of both T_1 and T_3. T_1 and T_2 are concurrent; therefore, T_1 does not see the effect of T_2's delete, and T_1's count includes x. T_3 starts after T_2 commits; therefore, T_3 sees the effect of T_2's delete, and T_3's count does not include x. Reasoning about these facts implies that the serialization order is T_1 before T_2, and T_2 before T_3. From the standpoint of dependency theory, this example contains the dependencies T_1--rw$\rightarrow T_2$ and T_2--wr$\rightarrow T_3$, which imply the same serialization order: T_1, T_2, T_3. The focus of this example lies in showing how PSSI detects T_2--wr$\rightarrow T_3$.

T_2's delete does not immediately remove x from the database; instead, T_2's delete creates the dead version x_2 (i.e., x_2.is_deleted = TRUE). PSSI relies on T_3 "reading" x_2, and noting the operation $r_3(x_2)$. x_2 does not contribute to the result of T_3's count query, but the operations $w_2(x_2)$, $r_3(x_2)$ are necessary for detecting the dependency T_2--wr$\rightarrow T_3$. Furthermore, x_2 must remain in the database for as long as there is a T_j such that T_2--wr$\rightarrow T_j$ might contribute to a cycle (i.e., x_2 must remain in the database for as long as T_2 is a zombie transaction).

The term *zombie transaction* first appeared in [ROO11, Sec. 3], but the general problem of needing to "remember" information about T_i beyond commit(T_i) occurs in other contexts. This topic is further discussed in Section 3.4.2.

3.3 Lock Table

This section introduces the first of two components that form the cornerstone of PSSI's design: PSSI's lock table and cycle testing graph (CTG). The lock table and cycle testing graph work together in a cooperative fashion – the lock table is responsible for keeping

track of data accesses, finding dependencies caused by these data accesses, and communicating these dependencies to the CTG. The CTG is responsible for determining if commit(T_i) would cause a cycle, and for determining when a committed transaction T_j cannot be part of any future cycle (i.e., when T_j is no longer a zombie). The CTG informs the lock manager (and other database components, if necessary) when it determines that T_j is no longer a zombie.

Database locks are usually associated with the notion of "lock wait", so the name *lock table* is slightly misleading in our case. PSSI's lock table resembles the structure of a traditional pessimistic lock manager, but its behavior is significantly different. For example, conflicting reads and writes do not delay transactions, so one could think of PSSI's lock table as a mechanism for "non-blocking locking".

3.3.1 Lock Table Data Structures

The data structure behind PSSI's lock table is derivative of Gray and Reuter's lock manager design [GR92, Figure 8.8]. Many lock manager optimizations can be adapted to work with PSSI's lock table, and in Chapter 4 we will describe how InnoDB's lock manager was modified to function as a PSSI lock table.

The lock table's primary data structure is a multilist [AHU83, p. 147], where each list element is a *lock control block* (LCB). Figure 9 shows the layout of a lock control block. In Figure 9, LCB.txn is a pointer to the owning transaction, which provides access to information such as the transaction's start and commit timestamps. LCB.data_item_name is an identifier that describes the locked data. The data item name might be a row identifier (RID) that refers to a physical storage address, or it might be a hashed key that refers to a

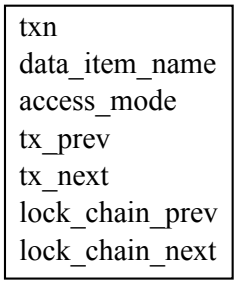

Figure 9: PSSI lock control block (LCB)

logical address. As noted in the introduction to this chapter, the term *data item* refers to rows and secondary index entries, but not to larger objects such as pages, tables, or files. LCB.access_mode distinguishes read accesses from write accesses, and the remaining four fields are linked list pointers. LCB.tx_prev and LCB.tx_next link the LCBs for a single transaction, while LCB.lock_chain_prev and LCB.lock_chain_next link the LCBs for a single data item. PSSI's lock table contains LCBs for both active and zombie transactions.

Figure 10 illustrates the lock table's multilist structure. Figure 10's vertical linked lists represent tx_prev and tx_next. These lists are rooted in the transaction objects themselves, so that T_i.tx_locks is the head of a linked list, and this linked list contains all of T_i's LCBs. Within T_i.tx_locks, reads are grouped together at the head of the list, and writes are grouped together at the tail of the list, a minor optimization for later processing. Figure 10's horizontal linked lists are *lock chains*, where each lock chain contains all accesses for a particular data item. Lock chains are accessed through a hash table, so that hash(x) produces the header cell containing x's lock chain. Within each lock chain, LCBs are kept in timeline order, so that reads appear at start(T_i) and writes appear at commit(T_i). We

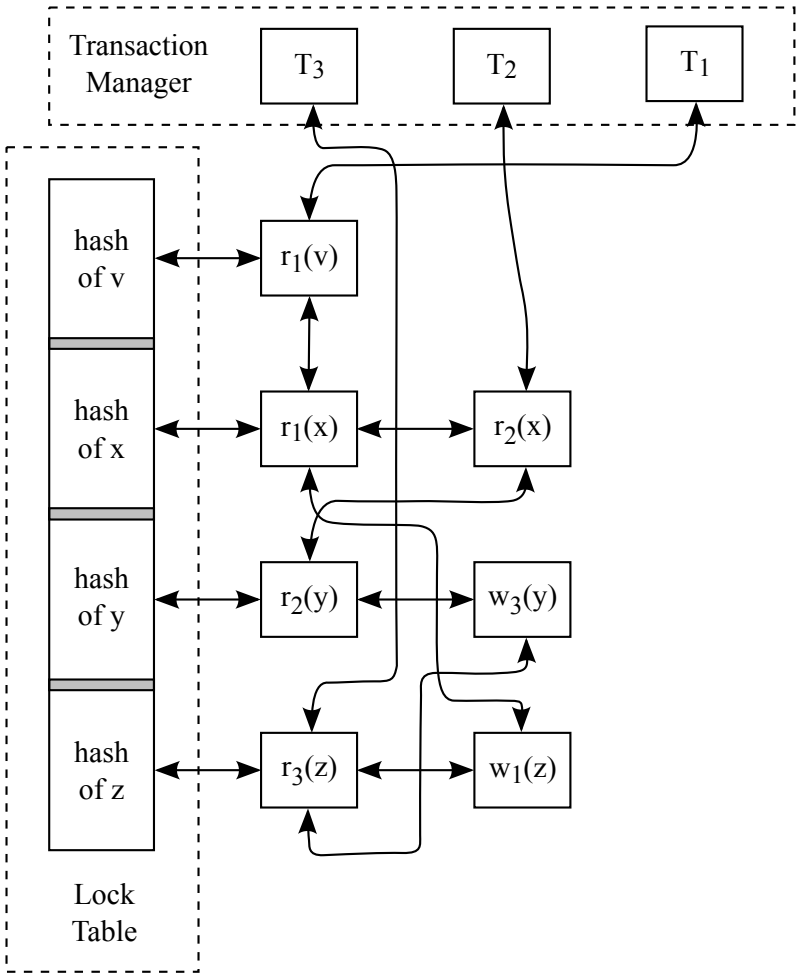

Figure 10: PSSI lock table

make the simplifying assumption that lock chains are homogeneous and contains LCBs for a single data item, but this simplifying assumption is not a requirement.

History $H_{3.2}$ corresponds to the lock table in Figure 10.

$H_{3.2}: \quad r_1(v), r_2(x), r_2(y), c_2, r_3(z), w_3(y), c_3, r_1(x), w_1(z), c_1$

By examining the lock chains in Figure 10, one can easily identify the dependencies that $H_{3.2}$ creates: there is T_2--rw→T_3 (from data item y), and T_3--rw→T_1 (from data item z).

There are obvious parallels between PSSI's lock table and the SI-RW diagrams introduced in Chapter 2. An SI-RW diagram presents a timeline view of an SI history, where reads occur at start(T_i) and writes occur at commit(T_i). The timeline ordering within lock chains is a direct manifestation of the SI-RW treatment.

3.3.2 Lock Table Algorithms for PSSI with FUW

This section presents a set of PSSI lock table algorithms where concurrent write-write conflicts are handled by first updater wins. As stated earlier, our goal is to present a general design that's independent of any particular DBMS implementation. Chapter 4 will discuss how these designs were implemented within the constraints of MySQL/InnoDB's lock manager.

Several themes underlie the algorithms presented in this section:

1. LCBs are added to the lock table as reads and writes occur.

2. The only lock waits that occur are those caused by FUW. Writes never wait for reads, and reads never wait at all.

3. Dependency testing is postponed until T_i issues a commit statement, since commit(T_i) is the earliest time that we can determine the full set of dependencies that T_i creates. This differs from earlier approaches (e.g., [CRF08, pg. 733]) which attempt to find dependencies as individual reads and writes occur.

```
1: procedure Add-Read-LCB(Transaction T_i, Data Item x)
2:     add r_i(x) to the head of T_i.tx_locks
3:     add r_i(x) to x's lock chain, positioned in timeline order, at start(T_i)
4: end procedure
```

Algorithm 4: Procedure to add a read LCB $r_i(x)$ for FUW

4. PSSI finds dependencies between a committing transaction T_i and committed zombie transactions T_j. Suppose our DBMS is executing the history H, and let $CH(H)$ be the committed projection of H containing all transactions that committed prior to commit(T_i). Given the assumption that $CH(H)$ is serializable, our goal is to ensure that $CH(H) \cup T_i$ is also serializable.

5. The algorithms assume that all conflicts (including predicate conflicts) can be observed through conflicts on data items. This assumption holds for ARIES-style range locking algorithms (e.g., [Moh95, Section 10.3.2]). Chapter 4 will describe how InnoDB's range locking (a variant of index-specific ARIES/IM) was modified for PSSI.

Coupled with cycle testing, these underlying themes make PSSI a serialization graph testing certifying scheduler [BHG87, Section 4.4]. The algorithmic time complexity of PSSI appears similar to the serialization graph testing designs presented in [BHG87], but unlike [BHG87]'s designs, PSSI does not require cascading aborts.

Add-Read-LCB (Algorithm 4) and Add-Write-LCB (Algorithm 5) show the steps involved in adding an LCB to PSSI's lock table, a process that is carried out each time a read or write occurs. Add-Read-LCB contains two steps: add $r_i(x)$'s LCB to the head of T_i.tx_locks, and place $r_i(x)$ in x's lock chain, positioned in timeline order at start(T_i). If $r_i(x)$ is positioned within x's lock chain at start(T_i), then all LCBs to the left of $r_i(x)$ will

```
1:  procedure Add-Write-LCB(Transaction T_i, Data Item x)
2:      add w_i(x) to the tail of T_i.tx_locks
3:      append w_i(x) to the tail of x's lock chain
4:                          ▷ iterate right-to-left, checking for concurrent write-write conflicts
5:      let lcb = w_i(x).lock_chain_prev
6:      let wait_for = NULL                          ▷ the lock T_i needs to wait for, if any
7:
8:      while lcb ≠ NULL do
9:          if lcb.txn committed before start(T_i) then
10:             break              ▷ concurrent write-write conflicts must be found by now
11:         end if
12:         if (lcb.access_mode == Write) and (lcb.txn is committed) then
13:             return FUW_VIOLATION          ▷ concurrent writer, already committed
14:         end if
15:         if (lcb.access_mode == Write) and (lcb.txn is active) then
16:             if wait_for == NULL then
17:                 let wait_for = lcb
18:             end if
19:         end if
20:         let lcb = lcb.lock_chain_prev          ▷ iterate left to previous LCB in timeline
21:     end while
22:
23:     if wait_for ≠ NULL then
24:         return LOCK_WAIT      ▷ T_i must wait until wait_for.txn commits or aborts
25:     end if
26:     return SUCCESS                                                    ▷ lock granted
27: end procedure
```

Algorithm 5: Procedure to add a write LCB $w_i(x)$ for FUW

be $r_j(x)$ where start(T_j) < start(T_i), or $w_k(x)$ where commit(T_k) < start(T_i). By a similar observation, all LCBs to the right of $r_i(x)$ will be $r_j(x)$ where start(T_i) < start(T_j), or $w_k(x)$ where start(T_i) < commit(T_k).

The procedure for adding a write LCB $w_i(x)$ is more complex, because it involves checking for FUW violations, and handling lock waits. Add-Write-LCB appends $w_i(x)$ to the tail of T_i.tx_locks, and to the tail of x's lock chain. While T_i is active, T_i.commit_ts $= \infty$, so placing $w_i(x)$'s LCB at the tail of x's lock chain is equivalent to positioning the LCB at commit(T_i). Lines 8–21 of Add-Write-LCB examine x's lock chain for concurrent write-write conflicts. The idea is to iterate over x's lock chain from right to left, looking for a $w_j(x)$ where T_j executed concurrently with T_i. There is a subtlety in this algorithm: the $w_j(x)$ that causes a FUW violation may not be the first $w_j(x)$ encountered during a right-to-left scan. Consider History $H_{3.3}$, where T_2 has just added $w_2(x)$ to the tail of x's lock chain, and needs to check for concurrent write-write conflicts.

$H_{3.3}$: $r_1(y), r_2(z), w_1(x), c_1, w_3(x), w_2(x)$ [$w_2(x)$ just added to x's lock chain]

In History $H_{3.3}$, T_1 and T_2 are concurrent, so the FUW violation between $w_1(x)$ and $w_2(x)$ should cause T_2 to abort immediately. For this to happen, T_2's right-to-left scan needs to look beyond the granted $w_3(x)$; in other words, T_2 should not enter lock wait because of $w_3(x)$.

Add-Write-LCB handles this case by using the variable "wait_for" to record the first (concurrent) write LCB found by the right-to-left scan. If no immediate FUW violation is found during lines 8–21 and wait_for $\neq$ NULL, then T_i must enter lock wait. By contrast, if no immediate FUW violation is found and wait_for is NULL, then there is no concurrent write-write conflict, and T_i acquires its write lock. Add-Write-LCB's LOCK_WAIT outcome means that deadlock is possible, so the DBMS must maintain a waits-for graph, and must perform deadlock detection each time a T_i enters lock wait.

The lock table's most important function is that of finding dependencies between a committing transaction T_i and each T_j in the set of zombie transactions.

```
1:  procedure Dependency-Check(Transaction T_i)
2:      for each l_i in T_i.tx_locks do
3:          let lock_chain = l_i's lock chain
4:          for each o_j in lock_chain do
5:              if Locks-Need-Dependency-Edge(l_i, o_j) then
6:                  Add-Dependency-Edge(l_i, o_j)
7:              end if
8:          end for
9:      end for
10: end procedure
```

Algorithm 6: Top-level procedure for finding dependencies during T_i commit

Dependency-Check (Algorithm 6) contains the top-level procedure for finding these dependencies. Dependency-Check is invoked once per transaction, after T_i.commit_ts is assigned, but before cycle testing is done. In order to find dependencies caused by T_i, one must find the dependencies caused by each $r_i(x)$ and $w_i(x)$; this is evident in the for-loop of lines 2–9, which iterates over the LCBs in T_i.tx_locks. Each of T_i's LCBs resides in a lock chain, and the inner loop of lines 4–8 examines the lock chain containing an individual T_i LCB, adding dependency edges when conflicts are found.

The procedure Locks-Need-Dependency-Edge (Algorithm 7) is responsible for determining whether two LCBs l_i and o_j conflict, which would warrant adding a dependency edge between their respective transactions, T_i and T_j. Note that Locks-Need-Dependency-Edge only considers LCBs belonging to zombie transactions, and not LCBs from active transactions. A dependency edge is warranted when the LCBs belong to different transactions, and at least one of them is a write.

Given a pair of conflicting LCBs, Add-Dependency-Edge (Algorithm 8) is responsible for adding the appropriate dependency edge to the CTG. Add-Dependency-Edge assumes

```
1:  procedure Locks-Need-Dependency-Edge(LCB l_i, LCB o_j)
2:      let T_i = l_i.txn                          ▷ T_i is the committing transaction
3:      let T_j = o_j.txn                          ▷ T_j is some other transaction
4:      if T_i == T_j then
5:          return FALSE                           ▷ T_i cannot conflict with itself
6:      end if
7:      if T_j is not committed then
8:          return FALSE          ▷ do not consider T_j because T_j is not (yet) a zombie
9:      end if
10:     if (l_i.access_mode == Write) or (o_j.access_mode == Write) then
11:         return TRUE                ▷ different transactions, conflicting access modes
12:     end if
13:     return FALSE
14: end procedure
```

Algorithm 7: Procedure to determine whether two LCBs warrant a dependency edge

that LCB l_i belongs to the committing transaction, and that LCB o_j belongs to a zombie transaction. There are three cases to consider, which are reflected in the three if-conditions of Algorithm 8. In the first case (lines 4–9), l_i is a read, which appears at start(T_i) in its lock chain, and o_j is a write, which may appear to the left or to the right of l_i. If o_j appears to the left of l_i, then T_j commits before T_i starts, and the appropriate edge is T_j--wr$\rightarrow T_j$. On the other hand, if o_j appears to the right of l_i, then T_j commits after T_i starts, and the appropriate edge is T_i--rw$\rightarrow T_j$.

Lines 10–11 and 12–13 of Add-Dependency-Edge handle the cases where l_i is a write. Write LCBs appear at commit(T_i), so the only LCBs that can appear "later" are those representing uncommitted writes, which are not considered by the dependency checking process (these write LCBs will be considered when their respective transactions commit). In these two cases, o_j must appear to the left of l_i. Lines 10–11 add T_j--rw$\rightarrow T_i$ when o_j is a read, and lines 12–13 add T_j--ww$\rightarrow T_i$ when o_j is a write.

```
1:  procedure Add-Dependency-Edge(LCB l_i, LCB o_j)
2:      let T_i = l_i.txn                          ▷ T_i is the committing transaction
3:      let T_j = o_j.txn                          ▷ T_j is a zombie transaction
4:      if (l_i.access_mode == Read) and (o_j.access_mode == Write) then
5:          if T_j committed before T_i started then
6:              add T_j--wr→T_i to CTG
7:          else
8:              add T_i--rw→T_j to CTG
9:          end if
10:     else if (l_i.access_mode == Write) and (o_j.access_mode == Read) then
11:         add T_j--rw→T_i to CTG
12:     else if (l_i.access_mode == Write) and (o_j.access_mode == Write) then
13:         add T_j--ww→T_i to CTG        ▷ FUW prevents T_i, T_j from being concurrent
14:     end if
15: end procedure
```

Algorithm 8: Procedure for determining which type of edge to add to the CTG

At this point, we should note a technicality regarding write LCBs and the timeline ordering within lock chains. Recall that T_i.commit_ts is initialized with a value of ∞, and the value of ∞ is used until T_i.commit_ts is assigned. Accordingly, write LCBs are always added to the tail of lock chains (see Algorithm 5, line 2). We have been making the simplifying assumption that lock chains are homogeneous, and contain LCBs for a single data item. Under this simplifying assumption, FUW guarantees that $w_i(x)$ for a committing T_i will be in the correct timeline-ordered position. However, if lock chains are heterogeneous (i.e., containing LCBs for different data items), then it will generally be necessary to reposition $w_i(x)$ after T_i.commit_ts is assigned, to maintain the timeline order. Reposition-Write-LCBs (Algorithm 9) gives the procedure for repositioning write LCBs during commit(T_i). Example 3.3 illustrates why repositioning is necessary for

```
1: procedure Reposition-Write-LCBs(Transaction T_i)
2:     let lcb = tail of T_i.tx_locks          ▷ writes are grouped at the tail of T_i.tx_locks
3:     while (lcb ≠ NULL) and (lcb.access_mode == Write) do
4:         reposition lcb in its lock chain, so that it appears in timeline order
5:           at T_i.commit_ts
6:         let lcb = lcb.tx_prev
7:     end while
8: end procedure
```

Algorithm 9: Procedure to reposition write LCBs, after T_i.commit_ts assigned

heterogeneous lock chains, and Example 3.4 illustrates why repositioning is not necessary for homogeneous lock chains.

Example 3.3 (*Repositioning and Heterogeneous Lock Chains*): Consider the history fragment $w_1(x)$, $w_2(y)$, c_2 with the assumption that $w_1(x)$ and $w_2(y)$ appear in the same (heterogeneous) lock chain. Prior to c_2, T_1 and T_2 are both active, and $w_1(x)$ and $w_2(y)$ are placed in timeline order, according to commit_ts $= \infty$. $w_2(y)$ appears to the right of $w_1(x)$, because $w_1(x)$ occurred before $w_2(y)$.

Once c_2 occurs, T_2 is assigned a commit timestamp (causing T_2.commit_ts $< \infty$) and T_1 is still active; this means that $w_1(x)$ and $w_2(y)$ are no longer in timeline order. To restore timeline order, $w_2(y)$ must be moved to the left of $w_1(x)$. Note that when T_1 commits, we will have commit(T_2) $<$ commit(T_1) which also requires $w_2(y)$ to appear before $w_1(x)$ in timeline order.

Example 3.4 (*Repositioning and Homogeneous Lock Chains*): Consider a homogeneous lock chain for x. While T_i is active, the lock table treats T_i.commit_ts as the value ∞, and all $w_i(x)$ are placed at the tail (rightmost end) of x's lock chain. If several active transactions have requested write locks on x, then the corresponding LCBs will appear at

the tail (right side) of x's lock chain; the first write lock requested (call it $w_i(x)$) is granted, and any $w_j(x)$ to the right of $w_i(x)$ represent transactions in lock wait.

If T_i commits, then all T_j having $w_j(x)$ to the right of $w_i(x)$ will be aborted (to enforce FUW), and $w_i(x)$ will become the right-most LCB in x's lock chain. Let $t_c = \text{commit}(T_i)$. We have $t_c < \infty$, but t_c is also the largest timestamp assigned so far. Therefore, a $w_i(x)$ that is positioned at timestamp t_c appears at the right-most end of x's lock chain, which is the position already occupied by $w_i(x)$. Thus, it is not necessary to change the position of $w_i(x)$ when T_i commits.

Finally, note that any $r_j(x)$ LCBs added to the lock table while T_i is active will be positioned in timestamp order according to $\text{start}(T_j)$. We always have $\text{start}(T_j) < \infty$, which means that such $r_j(x)$ will be placed to the left of a $w_i(x)$ held by an active transaction T_i.

Once PSSI determines the set of dependencies that $\text{commit}(T_i)$ would cause, PSSI performs a cycle test, as described in Section 3.4. If no cycle is found, then T_i's commit will succeed; on the other hand, if a cycle is found, then T_i will be forced to abort. In either case, a DBMS that employs FUW must be prepared to clean up any T_j that are waiting for T_i's write locks.

If T_i's commit succeeds, then Abort-FUW-Waiters (Algorithm 10) is applied to each of T_i's write locks. Recall that T_i's write locks are grouped together at the tail of T_i.tx_locks, a minor optimization for this step of the commit process. Abort-FUW-Waiters scans left to right from each $w_i(x)$, looking for waiting $w_j(x)$. Each waiting transaction T_j is added to the set "to_abort" and once the set of waiting transactions is found, they are aborted in one fell swoop.

```
1:  procedure Abort-FUW-Waiters(LCB lock)
2:      let T_i = lock.txn                          ▷ T_i commit succeeded, "lock" is w_i(x)
3:      to_abort = ∅
4:      let lcb = lock.lock_chain_next  ▷ waiting write locks must be to the right of w_i(x)
5:      while lcb ≠ NULL do
6:          if lcb is a waiting write lock then
7:              let T_j = lcb.txn
8:              to_abort = to_abort ∪ T_j
9:          end if
10:         let lcb = lcb.lock_chain_next          ▷ iterate right to next LCB in lock chain a
11:     end while
12:     abort all T_j ∈ to_abort
13: end procedure
```

Algorithm 10: Procedure for aborting transactions with waiting $w_j(x)$ locks

If T_i aborts (i.e., if T_i caused a cycle), then Unblock-FUW-Waiters (Algorithm 11) is applied to each of T_i's write locks. Unblock-FUW-Waiter unblocks the first T_j that is waiting behind $w_i(x)$ (other transactions may still be left waiting). One must take care to avoid race conditions when implementing this pair of algorithms. Consider history $H_{3.4}$, where three concurrent transactions attempt to write x.

$H_{3.4}$: $w_1(x)$, $w_2(x)$ (lock wait), $w_3(x)$ (lock wait)

If T_1 aborts, then Unblock-FUW-Waiters needs to unblock T_2 by granting $w_2(x)$, and leave T_3 in lock wait. However, if T_1 commits, then Abort-FUW-Waiters needs to abort both T_2 and T_3, but the abort of T_2 must not unblock T_3.

3.3.3 Optimizations to Dependency Checking

The timeline ordering of lock chains may make it possible to optimize Dependency-Check's inner loop (lines 4–8 of Algorithm 6). If we think of dependencies

```
1:  procedure Unblock-FUW-Waiters(LCB lock)
2:      let T_i = lock.txn                          ▷ T_i aborting, "lock" is w_i(x)
3:      let lcb = lock.lock_chain_next
4:      while lcb ≠ NULL do
5:          if lcb is a waiting write lock and T_j is not already aborted then
6:              let T_j = lcb.txn
7:              unblock T_j
8:              break                  ▷ locks to the right of w_j(x) continue to wait for T_j
9:          end if
10:         let lcb = lcb.lock_chain_next
11:     end while
12: end procedure
```

Algorithm 11: Procedure for unblocking a waiting $w_j(x)$ lock request

as relationships and consider the cardinalities involved, T_i--wr$\rightarrow T_j$ is a $1:N$ relationship between one writer and many readers; T_i--rw$\rightarrow T_j$ is an $N:1$ relationship between many readers and one writer; and T_i--ww$\rightarrow T_j$ is a $1:1$ relationship between two writers. In all three cases, each "w" side of the dependency is unique, and first updater wins guarantees a well-defined ordering of writes. This combination of circumstances may allow Dependency-Check to restrict the inner loop scan to a portion of a timeline-ordered lock chain.

Let $o_i(x)$ be any LCB in x's lock chain, where o may be a read or a write. Any T_i--rw$\rightarrow T_j$ or T_i--ww$\rightarrow T_j$ dependency that comes from $o_i(x)$ will be caused by the first committed $w_j(x)$ that appears to $o_i(x)$'s right; therefore, when examining the lock chain to the right of $o_i(x)$, it is not necessary to look beyond the first committed $w_j(x)$. Similarly any T_j--wr$\rightarrow T_i$ or T_j--ww$\rightarrow T_i$ dependency that comes from $o_i(x)$ is caused by the first $w_j(x)$ that appears to $o_i(x)$'s left; therefore, when examining the lock chain to the left of $o_i(x)$, it also not necessary to look beyond the first $w_j(x)$ encountered.

The usefulness of this optimization depends on the actual lock table implementation. This optimization will work perfectly well for lock chains having one LCB per data item; however, if the lock table uses LCBs to represent sets of data items, then this operation may not be feasible: if an LCB represents $o_i(\{x,y\})$, then lock chain scans cannot stop before finding *both* $w_j(x)$ and $w_k(y)$ to the right of $o_i(\{x,y\})$ and both $w_j(x)$, $w_k(y)$ to the left of $o_i(\{x,y\})$. InnoDB's lock table uses LCBs to represent sets of data items; this optimization was not beneficial for InnoDB, and we elected not to use it. We will return to the topic of InnoDB's lock table in Chapter 4.

3.4 Cycle Testing Graph

Chapter 2 introduced dependency serialization graphs. A dependency serialization graph (DSG) is a directed graph where nodes are committed transactions, and edges represent dependencies between pairs of committed transactions. DSGs represent complete histories, so it is impractical to use these graphs directly in database schedulers, just as it is impractical to use conflict serialization graphs over complete histories. This section introduces the cycle testing graph (CTG), a directed graph that contains only zombie transactions – committed transactions with the potential to become part of a future cycle – plus the transaction currently committing (if any). The CTG is capable of performing the same cycle testing functions as the DSG, but with a significantly smaller set of graph nodes. As an invariant, the CTG remains acyclic. Thus if C is a CTG and T_i is the currently committing transaction, then T_i is allowed to commit if $C \cup \mathrm{T}_i$ forms a new CTG C', and C' is acyclic. Unlike the DSG, CTG edges are *unlabeled*, and there is at most one edge from T_i to T_j. Therefore, there will be a CTG edge $\mathrm{T}_i \rightarrow \mathrm{T}_j$ if there are any dependencies from T_i to T_j.

```
1: procedure CTG-Add-Edge(Transaction T_i, Transaction T_j)
2:     if T_j ∉ T_i.out_edges then
3:         add T_j to T_i.out_edges
4:         increment T_j.in_edge_count
5:     end if
6: end procedure
```

Algorithm 12: Procedure for adding a $T_i \rightarrow T_j$ edge to the CTG

```
1: procedure CTG-Remove(Transaction T_i)
2:     for T_j ∈ T_i.out_edges do
3:         decrement T_j.in_edge_count
4:     end for
5:     remove T_i's node from the CTG
6: end procedure
```

Algorithm 13: Procedure for removing T_i from the CTG

Each zombie transaction T_i contains two CTG-specific fields: T_i.out_edges and T_i.in_edge_count. T_i.out_edges contains a list of T_i's out-edges (i.e., the transactions ids of T_j, such that there is an edge from $T_i \rightarrow T_j$), while T_i.in_edge_count contains the number of distinct transactions T_k that have an edge $T_k \rightarrow T_i$. PSSI's CTG uses out-edges for cycle testing, and in-edge counts for pruning transactions that are no longer zombies. CTG-Add-Edge (Algorithm 12) and CTG-Remove (Algorithm 13) are the chief manipulators of T_i.out_edges and T_i.in_edge_count. The former adds an edge from $T_i \rightarrow T_j$ and the latter removes T_i from the CTG. As these algorithms show, T_i.in_edge_count must be updated whenever an edge is added to, or removed from the CTG.

We have seen how PSSI uses Dependency-Scan to find dependency edges during T_i's commit, and Add-Dependency-Edge to add these edges to the CTG. Once these

dependencies are known, the CTG performs a cycle test. The cycle test is an ordinary depth-first search [CLR90, Section 23.3], starting from T_i. Using acyclicity as an invariant makes it possible to perform a single depth-first search per transaction. If the CTG C is acyclic but $C \cup T_i$ contains a cycle, then that cycle must pass through T_i, and that cycle can be found by a depth-first search that starts from T_i. Earlier papers [CRF09, Section 2.5] have postulated that cycle testing might be prohibitively expensive, but since only a single depth-first search per transaction is required, we believe the approach is quite economical.

3.4.1 CTG Pruning

In order to conserve memory, it's important to remove zombie transactions from the CTG, once we are certain that these transactions cannot be part of any future cycle. The process of removing transactions from the CTG is called *pruning*, and PSSI carries out this process each time there is a change in the set of active transactions. Theorem 3.5 gives the criteria for pruning zombie transactions from the CTG.

Theorem 3.5 (*Pruning Theorem*): A zombie transaction T_i cannot be part of any future cycle if T_i satisfies the following two conditions:

(1) T_i.in_edge_count = 0, and

(2) commit(T_i) $< t_0$, where t_0 is the start timestamp of the oldest active transaction.

Proof: In order to be part of a cycle, it is necessary for T_i to have an edge leading in, and an edge leading out. Condition (1) requires T_i to have no in-edges, therefore, T_i must acquire a new in-edge before it can become part of a cycle.

Condition (2) guarantees that T_i cannot acquire new in-edges. Without loss of generality, let T_k be any active transaction. If t_0 is the start timestamp of the oldest active

transaction, then we have $t_0 \leq \text{start}(T_k)$, and by condition (2), we have $\text{commit}(T_i) < t_0 \leq \text{start}(T_k)$. Because $\text{commit}(T_i) < \text{start}(T_k)$, T_k's commit could only cause the following types of dependency edges to become incident to T_i: T_i--wr$\rightarrow T_k$, T_i--ww$\rightarrow T_k$, or T_i--rw$\rightarrow T_k$. None of these adds a new in-edge to T_i, therefore, any new edges incident to T_i must be out-edges.

In summary, if T_i has no in-edges, and T_i cannot acquire new in-edges, then T_i cannot become part of a future cycle. If T_i cannot become part of a future cycle, then T_i is no longer a zombie, and T_i may be pruned from the CTG. □

Theorem 3.5 provides a set of conditions that are sufficient to ensure that T_i cannot be part of a future cycle, but we do not claim these conditions are necessary. (e.g., one could replace (1) with the condition "T_i.out_edges $= \emptyset$", and use that as the basis for an alternate version of the pruning theorem. However, such a set of conditions would likely be more difficult to check.)

The process for pruning the CTG consists of two phases, which are shown in Algorithm 14. The first phase finds a set of transaction S that already satisfy the pruning theorem and are, by definition, prunable. The second phase recursively examines each $T_i \in S$, since the removal of T_i may make it possible to prune additional transactions T_j, where $T_j \notin S$. Example 3.6 illustrates the pruning process.

Example 3.6: Suppose we have the CTG shown in Figure 11, where T_1 was the first transaction to start and the last transaction to commit, and where T_1's commit leaves no active transactions. (In this special case, we can take t_0 to be the next start timestamp that would be assigned.)

```
1:  procedure CTG-Prune
2:      let t_0 = start timestamp of the oldest active transaction
3:      let S = transactions with no in-edges, that committed before t_0
4:      for T_i ∈ S do
5:          CTG-Prune-Recursive(T_i, t_0)
6:      end for
7:  end procedure
8:
9:  procedure CTG-Prune-Recursive(Transaction T_i, timestamp t_0)
10:     if (T_i.commit_timestamp > t_0) or (T_i.in_edge_count > 0) then
11:         return
12:     end if
13:     let R = { T_j | T_j ∈ T_i.out_edges }
14:     CTG-Remove(T_i)
15:     Notify-Observers(T_i)
16:     for T_j ∈ R do
17:         CTG-Prune-Recursive(T_j, t_0)
18:     end for
19: end procedure
```

Algorithm 14: Procedures for pruning the CTG

The first phase of CTG-Prune identifies the set of transaction S that already satisfy the pruning theorem; this gives $S = \{ T_1, T_2 \}$. The second (recursive) phase consists of the following steps:

1. T_1 is removed from the CTG. T_1 has a single out-edge, $T_1 \rightarrow T_3$, so T_3 is examined next.

2. T_3.in_edge_count = 1 (from $T_2 \rightarrow T_3$). This makes T_3 ineligible for pruning, and the recursive examination of T_1 ends.

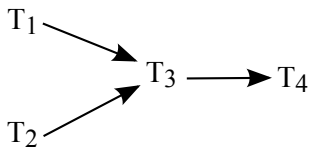

Figure 11: CTG for Example 3.6

3. T_2, the next member of *S*, is removed from the CTG. T_2 has a single out-edge, $T_2 \rightarrow T_3$, so T_3 is examined again.

4. Upon re-examination, T_3.in_edge_count = 0, which allows T_3 to be removed from the CTG. T_3 has a single out-edge, $T_3 \rightarrow T_4$, so T_4 is examined next.

5. T_4.in_edge_count = 0. T_4 is removed from the CTG, and this ends the recursive examination of T_2.

When CTG-Prune ends, the CTG in this example will be empty. *Note that the pruning order (*T_1*,* T_2*,* T_3*,* T_4*) is a topological sort, and provides an equivalent serial history over the set of zombie transactions*.

There is one remaining aspect of the pruning process that needs to be discussed: the Notify-Observers call that appears in CTG-Prune-Recursive. This is a straightforward application of the observer pattern. The idea is that several database components may have to retain information about T_i while it is a zombie transaction. For example, the lock table needs to retain T_i's read and write locks, and the transaction manager needs to retain T_i's basic data structure. Notify-Observers provides a way for the CTG to let other components know that T_i is no longer a zombie, thereby allowing those components to release any resources dedicated to T_i.

```
1: procedure Notify-Observers(Transaction T_i)
2:     for each CTG observer o do
3:         o.Clear-Transaction(T_i)
4:     end for
5: end procedure
```

Algorithm 15: Use of the observer pattern to remove T_i from DBMS internal components

The implementation of Notify-Observers appears in Algorithm 15. Each observer provides a Clear-Transaction callback, and Notify-Observers invokes these callbacks, passing the just-pruned transaction T_i. These callbacks would typically be established early in the database startup sequence, before any transactions are serviced.

3.4.2 Zombies and Pruning in Other Contexts

Section 3.2 stated that the problem of *zombie transactions* — the need to remember information about T_i for some time after T_i commits — occurs in contexts other than PSSI. This section describes two such contexts.

As noted in [CRF09, pg. 26], the strategy of essential dangerous structure testing (ESSI) requires the DBMS to keep track of an already-committed transaction T_i if T_i executed concurrently with a still-active transaction T_j. In other words, the DBMS cannot discard information about T_i until $\text{commit}(T_i) < t_0$, where t_0 is the start timestamp of the oldest active transaction. This matches condition (2) of the Pruning Theorem (Theorem 3.5).

Zombie transactions also appear in the discussion of basic serialization graph testing (basic SGT) from [BHG87, pg. 123] (although [BHG87] does not use the term "zombie transaction"). A basic SGT scheduler must retain information about a committed

transaction T_i for as long as T_i has an in-edge count greater than zero. This matches condition (1) of the Pruning Theorem.

Thus, PSSI's Pruning Theorem can be seen as the union of conditions imposed by two other approaches: a committed PSSI transaction T_i remains a zombie for as long as T_i has in-edges (as in basic SGT), or executed concurrently with a still-active transaction T_j (as in ESSI).

3.5 Summary

This chapter presented PSSI's design fundamentals. The chapter began with a discussion of timestamps, and techniques for version storage and management. Version management is not a new area of research, but it is an integral part of any SI implementation.

PSSI requires the database to keep track of already-committed zombie transactions, since zombie transactions ultimately play a role in the formation of dependency cycles, and influence the handling of deleted data items.

The lock table and cycle testing graph are the two central components in PSSI's design. The lock table is responsible for finding dependencies and reporting them to the CTG; the CTG is responsible for detecting cycles, and for determining when a zombie transaction T_i cannot be part of any future cycle.

Chapter 4 will describe how these designs were incorporated into the InnoDB storage engine that ships with MySQL 5.1.31, serving as a case study for the implementation of PSSI in a working database system.

CHAPTER 4

PSSI IMPLEMENTATION WITH MYSQL/INNODB

This chapter presents a case study where we describe our implementation of PSSI in MySQL 5.1.31's InnoDB storage engine. InnoDB was a good starting point for this project, given its support for multiversioning (but *not* snapshot isolation), and a variant of index-specific ARIES/IM locking [Moh95]. (ARIES is a family of algorithms for B-tree isolation and recovery, and the acronym "ARIES/IM" stands for "Algorithm for Recovery and Isolation Exploiting Semantics for Index Management".) Starting from a standard distribution MySQL/InnoDB 5.1.31, our prototype work required three phases of development:

1. adding support for first updater wins, thereby providing traditional snapshot isolation (i.e., as defined in [BBG95]),

2. implementing the algorithms presented in Chapter 3, slightly modified to fit the design of InnoDB's lock manager, and

3. adapting pessimistic index-specific IM to work with an optimistic multiversioned system.

Any database management *system* depends not only on individual components, but on the way those components work together in a cooperative fashion; InnoDB is no exception

to this rule. Consequently, in order to provide a comprehensive view of this system, we will have to describe a number of different parts. We begin with the implementation of SI in InnoDB, since that was a small and straightforward addition. From there, we move to InnoDB's physical structures: how rows and secondary index records are stored as a set of B-trees. Next, we present InnoDB's lock manager, and our modifications to it. Our lock manager coverage describes InnoDB's variant of index-specific IM, and how it is used to lock ranges and avoid phantom anomalies. Next, we describe our CTG implementation, and the issues we discovered in moving from conceptual design to a real-world system. The chapter concludes with a list of miscellaneous issues: durability, replication, and garbage collection of old versions.

4.1 Adding SI Support to InnoDB

As noted earlier, InnoDB provides MVCC but not snapshot isolation. Consequently, our first task was to add support for SI, by implementing first updater wins (see Definition 1.2, page 3).

Adding FUW involved little more than implementing Algorithms 5, 10, and 11 from Section 3.3.2. Let T_i be the transaction that has written x most recently, let T_j be a concurrent transaction that also wishes to write x, and recall that FUW must be prepared to handle two cases: (1) where $w_j(x)$ occurs before c_i, and (2) where $w_j(x)$ occurs after c_i. Case (1) is handled entirely within the lock manager. Case (2) is handled as follows: before T_j writes x, T_j must examine x.txid. If x.txid is visible to T_j (i.e., the last transaction that wrote x committed prior to start(T_j)), then T_j may write x and continue. Otherwise, x must have been written by a concurrent (and already-committed) transaction,

and T_j must abort. Ordinary SI does not have zombie transactions, so case (2) must be handled by checking timestamps in the row update code.

Each MySQL/InnoDB transaction is tied to a separate operating system thread, and one must be careful to avoid race conditions during FUW aborts (as alluded to in Section 3.3.2, during the discussion of Algorithm 10). We use the following strategy to void race conditions during FUW aborts: when T_i commits, T_i first identifies the set of waiting transactions T_j that have write-write conflicts with T_i; T_i marks such T_j by setting the field T_j.has_fuw to TRUE. Once this marking is done, T_i brings each T_j out of lock wait. When T_j comes out of lock wait, T_j notices that its has_fuw flag is set and rolls itself back. Thus, T_j is rolled back by its own operating system thread, and the rollback does not delay T_i's commit.

4.2 InnoDB Physical Structures

InnoDB uses B-trees exclusively for data storage. Every InnoDB table consists of (1) a clustered B-tree index called "PRIMARY" and (2) zero or more secondary B-tree indexes. InnoDB uses a logical addressing scheme where PRIMARY holds table rows, and secondary indexes refer to PRIMARY rows by primary key. This arrangement allows InnoDB to treat record locks and range locks uniformly: every InnoDB record lock is a range lock on a B-tree index. In this chapter, the term *record lock* means "a row lock" or "a lock on a secondary, non-clustered, index tuple".

InnoDB versions PRIMARY index rows. Each PRIMARY index row x_i contains two system header fields: the transaction id of the writing transaction T_i, and a pointer to x_i's immediate predecessor. This is the chaining scheme described in Section 3.1.1 and

illustrated in Figure 8. InnoDB stores old versions in a set of pages called the *rollback segment*, and like other data pages, these are managed by InnoDB's buffer pool. InnoDB stores old versions in a manner that is reminiscent of compressed value logging [GR92, Section 10.3.4]. If $x_j = \text{pred}(x_i)$, then x_j will contain (1) the transaction id j, (2) a pointer to $\text{pred}(x_j)$ (which may be a null pointer), and (3) the set of column values where x_j differs from x_i.

InnoDB does not version secondary index records. Instead, the header of each secondary index page p contains a field called PAGE_MAX_TRX_ID, which holds the largest transaction id i of the T_i that modified p. When T_j wishes to read from p, T_j begins by reading i from p.PAGE_MAX_TRX_ID; if T_i's writes are visible to T_j, then T_j may read p as-is. On the other hand, if T_i's writes are not visible to T_j, then T_j must dereference the PRIMARY row each time that T_j reads a secondary index record from p. Dereferencing the PRIMARY row allows T_j to locate the row version (if any) that is in T_j's snapshot. If one assumes that secondary index updates are infrequent, then this is useful space-saving compromise.

Every data page p contains a counter, PAGE_N_HEAP, that's used to assign *heap numbers* to individual records on p. Heap numbers uniquely identify records in individual pages, and the combination of (tablespace id, page number, heap number) acts as a row id (RID) and uniquely identifies any physical record in the database. Heap numbers denote the temporal order in which records were added to a leaf page, and these numbers are independent of dictionary ordering. Thus if u, v, y, and x are inserted into an empty page p, these records will be given the following heap numbers: $u = 2$, $v = 3$, $x = 5$, and $y = 4$. (Heap numbers zero and one have special meanings, which will be described shortly.) Once assigned, heap numbers do not change, unless the page splits, or undergoes

reorganization. This constancy is imperative, since InnoDB locks the physical location of records: the combination of tablespace id, page number, and heap number.

Heap numbers zero and one have special significance to InnoDB. Heap number zero is the page *infimum*; InnoDB temporarily locks this record during page reorganization. Heap number one is the page *supremum*; this record represents the gap at the far "right" side of the page, above the largest record in dictionary order. The supremum acts like an end-of-file marker for page p, so that if T_i has locked the supremum record on page p, then we know that T_i has scanned past the largest record on p. The infimum and supremum are physical records: the infimum record contains the string infimum offset 99 bytes from the start of the page, and the supremum record contains the string supremum offset 112 bytes from the start of the page.

Within B-tree leaf pages, records are organized as a linked list in dictionary order. The *page directory* is an array of entries that point to every n^{th} element of the linked list, for $n \leq 8$. This is an interesting design choice. This arrangement permits binary searches of the page directory, but the result of the binary search will be a sub-list of eight (or fewer) elements, and the sub-list must be scanned linearly. Consequently, binary searches on InnoDB page directories are slightly slower than a design that allocates separate directory entries to each record. On the other hand, inserts are generally very fast, since InnoDB only needs to manipulate a pair of linked list pointers, and if necessary, adjust the page directory so that there are no more than eight records between directory entries.

Figure 12 illustrates the u, v, y, x insertion scenario described earlier. The records are contained in a linked list in dictionary order, so that a range scan would traverse the heap numbers 2, 3, 5, 4, and 1. There are also two page directory entries: the first directory entry points to the sublist (u, v), and the second directory entry points to the sublist (x, y).

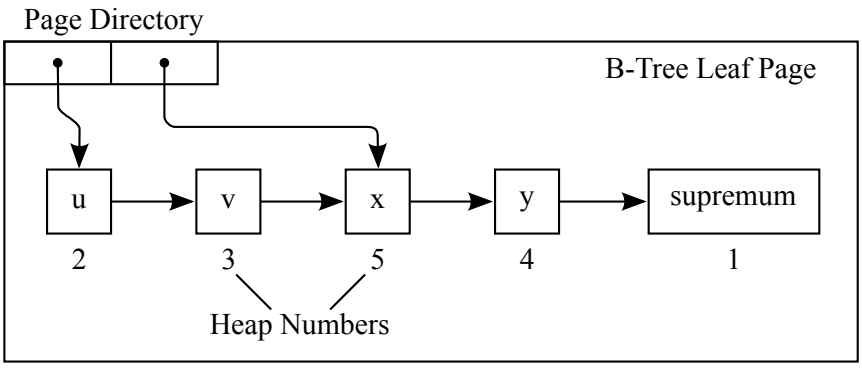

Figure 12: Structure of a B-tree leaf page, containing four records

A subsequent insert of *vv* would produce the page shown in Figure 13. In Figure 13, note that *vv* has been placed in the linked list between *v* and *x*, and assigned the heap number six.

The key point of these examples is the following: heap numbers are independent of dictionary order, and with the exception of page reorganizations, the mapping between heap numbers and records does not change. This fact is critically important to the working of InnoDB's lock manager, which locks physical row locations, based on the combination of tablespace id, page number, and heap number.

4.3 InnoDB's Lock Manager

The standard distribution InnoDB has a sophisticated lock manager, and a robust implementation of ARIES/IM index locking. InnoDB's lock manager provided a solid foundation for implementing PSSI's algorithms. This section explains how InnoDB's lock manager works, and how we modified it to support PSSI. A large portion of this work

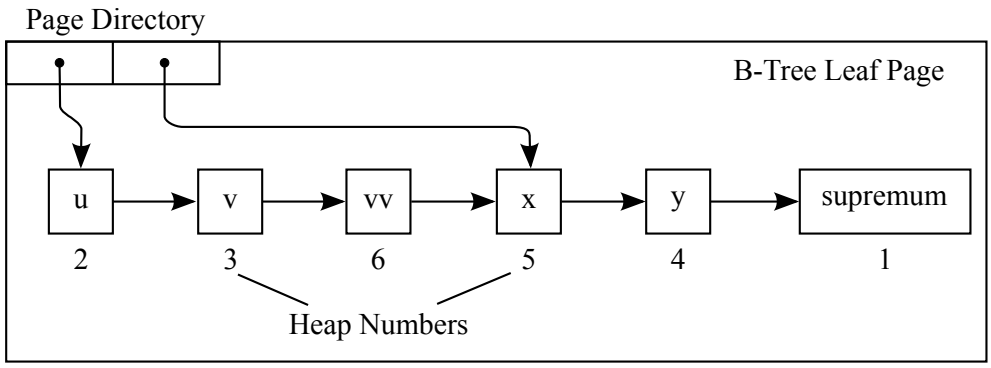

Figure 13: B-tree leaf page from Figure 12, after inserting *vv*

involved replacing the original lock manager code with the algorithms presented in Section 3.3. Changing the underlying lock manager semantics was one of the more challenging aspects of the project. InnoDB guarantees serializability through pessimistic S2PL, where the notions of *lock wait* and *conflict* are equivalent. PSSI treats lock wait and conflict as two different concepts (i.e., reads and writes may conflict, but do not cause lock waits), consequently, it was necessary to separate the notions of lock wait and conflict at the API layer within the lock manger.

InnoDB's lock manager has an interesting design. Rather than creating an LCB (or lock_t, as they're called the InnoDB source code) for each record accessed by T_i, InnoDB creates one LCB per page accessed by T_i. Along with standard bookkeeping information (transaction id, lock mode, and so forth), each LCB contains a bitmap denoting which records have been locked. Suppose that transaction T_i wishes to lock record x on page p in mode m. One of two things will happen: if T_i already has an LCB for page p in mode m, then T_i simply turns on the bit that corresponds to x's heap number. On the other hand, if T_i does not have an LCB for page p in mode m, then T_i creates a new LCB, and sets the

bit for x's heap number. This design requires slightly more memory when T_i locks few records on p, but it is very efficient for range scans (i.e., T_i can lock all records on p with a single LCB). The bitmaps work well for dependency testing: let l_i and l_j be two LCBs with conflicting modes, owned by distinct transactions T_i and T_j. We can determine whether there is a dependency between T_i and T_j with a simple (and efficient) bitmap intersection.

InnoDB organizes locks in a hashtable, similar to the one shown in Figure 10. A hash of p's page number determines a hashtable cell, and the hashtable cell contains all LCBs for page p. There are two structural differences between InnoDB's lock manager and Figure 10: the granularity of LCBs and the organization of lock chains. Figure 10 uses one LCB for each record that T_i accesses, but InnoDB uses one LCB for each page that T_i accesses. This was a minor obstacle to overcome, requiring only small changes to the algorithms presented in Chapter 3. The standard distribution InnoDB's lock chains are singly-linked lists, where new LCBs are appended to the right side of the chain. This is a more marked contrast from Figure 10, where lock chains are doubly-linked lists, and LCBs appear in timeline order. To permit bi-directional iteration, we replaced the singly-linked lock chains with doubly-linked ones, and implemented the requisite timeline ordering.

InnoDB utilizes multi-granular locking [BHG87, Sec. 3.9]. In the ordinary case of T_i accessing record x in table T, T_i must acquire an intention lock on T, and then a shared or exclusive lock on x. PSSI is concerned with finding dependency cycles based on record accesses; table locks are more of an implementation detail and really do not influence the working of PSSI.

4.3.1 Lock Modes and Lock Flags

Each InnoDB LCB has a 32-bit integer called type_mode, that contains the lock mode along with several flags. type_mode is organized as follows:

- **Access mode**. The low eight-bits of type_mode (type_mode & 0xf) represent the data access mode. The possible values are LOCK_S (4), LOCK_X (5), LOCK_IS (2), and LOCK_IX (3); these values correspond shared, exclusive, intention shared, and intention exclusive access modes. PSSI is strictly concerned with shared (LOCK_S) and exclusive (LOCK_X) record locks.

- **Table vs. record lock**. The next eight bits of type_mode (type_mode & 0xf0) contain one of LOCK_TABLE (0x10) or LOCK_REC (0x20). The former denotes a table lock, while the latter denotes a record lock.

- **Wait flag**. The LOCK_WAIT bit (type_mode & 0x100) denotes a waiting (not granted) lock request. This bit is turned on when T_i enters lock wait, and turned off when T_i leaves lock wait.

- **Gap bits**. InnoDB uses *gap bits* (type_mode & 0x600 – two bits) to distinguish between locks on records and locks on gaps between records.
  LOCK_ORDINARY (0) means that the lock applies to the record x, and to the gap immediately below x in dictionary order. LOCK_GAP (0x200) means that the lock applies to the gap below x in dictionary order, but not to x itself.
  LOCK_REC_NOT_GAP (0x400) means that the lock applies to x, but not the gap below. InnoDB's gap bits are a useful refinement to ARIES/IM, and can avoid some of the false positives inherent in the original ARIES algorithms.

- **Insert intention flag**. The LOCK_INSERT_INTENTION bit (type_mode & 0x800) is not an intention lock per-se. LOCK_INSERT_INTENTION denotes a newly-inserted record; the range locking algorithms (described in Section 4.3.2) use this flag to detect range conflicts.

Below, we provide several examples to illustrate the use of lock modes and the LCB's bitmap.

Example 4.1: Suppose we have a non-unique index with the following keys and heap numbers:

Key:	A	A	B	B	C	C	D
Heap Number:	2	3	5	4	7	6	8

and let T_i perform the scan "=B". T_i create the following two LCBs:

LCB_1:	bitmap:	5, 4
	type_mode:	LOCK_ORDINARY | LOCK_REC | LOCK_S
LCB_2:	bitmap:	7
	type_mode:	LOCK_GAP | LOCK_REC | LOCK_S

LCB_1 denotes the bitmap 0000 1100, and LCB_2, denotes the bitmap 0000 0010 (the left-most position is bit zero, the next position is bit 1, and so on). These two LCBs lock both "B" keys, the gaps below each "B" key, and the gap between "B" and "C" (but not "C" itself). Consequently, a concurrent transaction T_j could acquire an exclusive lock on bit 7 without conflicting with T_i.

Example 4.2: Given the index in Example 4.1, let T_i perform the range scan "between A and B". T_i creates the following LCB:

LCB_3: bitmap: 2, 3, 5, 4, 7

type_mode: LOCK_ORDINARY | LOCK_REC | LOCK_S

Note the difference in behavior here: when the scan is "=B", InnoDB locks the first non-matching key with LOCK_GAP, but when the scan is "between A and B", InnoDB locks the first non-matching key with LOCK_ORDINARY. In other words, an equality condition locks the gap below "C" but not "C" itself, while the "between" condition locks "C" and the gap below. The standard distribution InnoDB exhibits this behavior with unique and non-unique indexes. There appears to be an opportunity for improvement here (having "between" scans lock the first non-matching key with LOCK_GAP, instead of LOCK_ORDINARY), but we elected not to change this aspect of InnoDB's behavior.

Example 4.3: Suppose we have a unique index with the following keys and heap numbers:

Key: A C D

Heap Number: 2 4 3

and let T_i perform the range scan "=C". T_i creates the following lock:

LCB_4: bitmap: 4

type_mode LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S

In this case, T_i locks "C" but not the surrounding gaps. A concurrent transaction T_j could insert "B" without conflicting with T_i.

Example 4.4: Given the index from Example 4.3, let T_j perform the range scan "=B". This is a NOT FOUND condition, as "B" does not exist in the index. The NOT FOUND condition is represented with the following LCB:

LCB_5: bitmap: 4

type_mode: LOCK_GAP | LOCK_REC | LOCK_S

	OS	OX	RS	RX	GS	GX	GXI
OS	Y		Y		Y	Y	
OX					Y	Y	
RS	Y		Y		Y	Y	Y
RX					Y	Y	Y
GS	Y	Y	Y	Y	Y	Y	
GX	Y	Y	Y	Y	Y	Y	
GXI			Y	Y			Y

Table 1: InnoDB Lock Compatibility Matrix

This locks the gap between "A" and "C", but not "C" itself. In Section 4.3.2, we will see how LCB_5 will create a conflict, should a concurrent T_j try to insert "B".

As these examples show, InnoDB locks are a combination of orthogonal attributes: the data access mode (read/shared or write/exclusive), and the gap mode. Using an analysis technique from [Gra10, pg. 12], we can form a compound lock mode by concatenating the individual attributes. Two compound lock modes are compatible if all of the individual attribute pairs are compatible. Table 1 gives the lock compatibility matrix for InnoDB, where "Y" denotes compatible lock modes.

In Table 1, the first character represents the gap mode: (O)rdinary, (R)ecord-only, or (G)ap. The second character represents the data access mode: (S)hared or e(X)clusive. The third character represents the presence of the (I)nsert intention bit. It is worth pointing out several characteristics about Table 1. First, the compatibility matrix is symmetric. Second, the insert intention bit only appears in conjunction with LOCK_GAP and

LOCK_X. Third, gap locks only conflict with locks whose insert intention bit is set. The purpose of gap locks is to detect conflicts with inserts, and we will discuss how this is done in Section 4.3.2.

4.3.2 Range Locking and Phantom Avoidance

InnoDB prevents phantom anomalies with a variant of ARIES/IM, and this section focuses on how InnoDB's range locking algorithms were adapted for PSSI. The same code is used for SI, ESSI and PSSI, but our primary focus is the end product for PSSI. (The differences between the original S2PL implementation and the implementation we arrived at for PSSI are relatively minor, and mostly involve the handling of read/write conflicts; read/write conflicts cause lock waits for S2PL, but not for PSSI.) Before presenting these algorithms in detail, we will need to introduce the concepts of implicit and explicit locks.

Definition 4.5 (*Implicit Lock, Explict Lock*): An *implicit lock* is represented by x.txid, the transaction id in x's row header, while an *explicit lock* is represented by an LCB in the lock manager.

The idea behind implicit locks is as follows: suppose T_j inserts x and at the time of insert, T_j determines that the insert does not cause a (range) conflict. In this case, T_j does *not* need to add a $w_j(x)$ LCB to the lock manager; instead, the lock implied by x.txid is used to detect any subsequent conflicts. The term "implicit lock" appears frequently in the comments of InnoDB's source code (lock0lock.c in particular), but it is not a standard term in database literature.

Implicit locks require cooperation from accessing transactions. Let T_i be a transaction that wishes to access x; T_i must apply procedure Convert-Implicit-to-Explicit, given in

```
1:  procedure Convert-Implicit-to-Explicit(Transaction T_i, Data Item x)
2:      let j = Find-Implicit-Locker-Txid(x)
3:      if (j ≠ NULL) and (T_j is an active or zombie transaction) then
4:          if T_j does not have an explicit lock on x then
5:              create an explicit w_j(x) lock with
6:                  LOCK_REC_NOT_GAP | LOCK_REC | LOCK_X
7:          end if
8:      end if
9:      T_i enqueues its own o_i(x) lock here
10: end procedure
```

Algorithm 16: Conversion of Implicit to Explicit Locks

Algorithm 16. If the implicit locker (i.e., the inserting transaction) T_j is an active or zombie transaction, and there is no explicit $w_j(x)$ LCB, then T_i creates an explicit $w_j(x)$ on T_j's behalf, and then enqueues its own $o_i(x)$ LCB. Implicit-to-explicit conversions must be done before T_i enqueues its own LCB; if $o_i(x)$ is $w_i(x)$, then T_i may need to wait, and may need to undergo FUW resolution. (If T_j is a zombie transaction that committed prior to start(T_i), then FUW resolution is not necessary, and PSSI will only note the T_j--ww$\rightarrow T_i$ dependency.) Implicit-to-explicit conversion occurs regardless of whether x is visible to T_i, since the access must be noted for dependency testing.

Implicit locking has an obvious benefit for bulk inserts: T_i can insert a large number of rows, without having to record each one in the lock table. This can reduce the number of LCBs in the lock table's hashtable which, in turn, helps speed up lock table operations.

Algorithm 17, Find-Implicit-Locker-Txid shows the procedure used to find the transaction id j for the T_j that may hold an implicit lock on x; this procedure varies according to whether x is PRIMARY or secondary index record. If x is a PRIMARY index record, then x.txid identifies the implicit locker. The procedure is more complex when x is

```
1:  procedure Find-Implicit-Locker-Txid(Data Item x)
2:      if x is a PRIMARY index record then          ▷ PRIMARY (clustered) index case
3:          return x.txid
4:      end if
5:      let m = PAGE_MAX_TRX_ID for page containing x        ▷ secondary index case
6:      if T_m is neither active nor zombie then
7:          return NULL                                          ▷ no implicit lock
8:      end if
9:      let y = PRIMARY index row for x
10:     return y.txid
11: end procedure
```

Algorithm 17: Finding an Implicit Locker

a secondary index record, since secondary index records are not versioned, and have no txid header field. For a secondary index, the first step is to examine p.PAGE_MAX_TRX_ID for the page p containing x. Let T_m be the transaction where $m = p$.PAGE_MAX_TRX_ID; if T_m is neither an active nor a zombie transaction, then T_m cannot be part of any future cycle, and there is no implicit lock. On the other hand, if T_m is an active or zombie transaction, then Find-Implicit-Locker-Txid must locate y, the PRIMARY index row that x refers to, and use y.txid to find the implicit locker (which may, or may not be T_m). Lock manager mutexes are temporarily released in line 9 of Algorithm 17, since retrieving y may require a disk IO; the mutex is re-acquired when the function returns. (The temporary lock release was carried forward from the original S2PL implementation.)

Example 4.6 illustrates the process of implicit to explicit lock conversion with a PRIMARY index record. Example 4.8 (located at the end of this section) illustrates the use of implicit locks in a secondary index.

Example 4.6 (*Insert Before Range Scan*): Suppose we have a unique index with keys "A" and "F". T_1 will insert "C", T_2 will scan "between A and D", and both transactions will commit. The following sequence of events takes place:

1. T_1 positions a cursor over "A". ("C" will be inserted immediately after "A".)
2. T_1 examines NEXT(A) = F. There are no locks on "F", so inserting "C" does not cause a range conflict.
3. T_1 inserts "C", with an implicit lock. No LCB is added to the lock manager.
4. T_2 begins its scan "between A and D".
5. T_2 enqueues $r_2(A)$ with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S. (This is a unique index and "A" matches the lower boundary of the range scan, so, T_2 only needs to lock "A", and not the gap below.)
6. T_2 examines the transaction id in C's row header. This transaction id indicates that "C" is not visible to T_2, and there is no explicit lock on this record. T_2 converts T_1's implicit lock into an explicit one, and enqueues $w_1(C)$ on T_1's behalf. This lock has flags LOCK_REC_NOT_GAP | LOCK_REC | LOCK_X, which locks "C" but not the gap below.
7. T_2 enqueues $r_2(C)$. This lock has flags LOCK_ORDINARY | LOCK_REC | LOCK_S. "C" does not affect the results of T_2's query, but the lock is necessary in order to detect the T_2--rw$\rightarrow T_1$ dependency.
8. T_2 enqueues $r_2(F)$, also with LOCK_ORDINARY | LOCK_REC | LOCK_S. Assuming that "C" and "F" reside on the same page, T_2 can take the LCB created in

the prior step, and turn on the bit corresponding to F's heap number (both locks have the same type_mode).

9. T_1 and T_2 commit.

Example 4.6 illustrates the one case where PSSI generates false positives. When scanning "between A and D", T_2 locked "F", the first non-matching key encountered. If a T_3 were to insert "E", then PSSI would find a dependency between T_2 and T_3; although there is not a logical conflict between these two transactions, T_2 has locked "F" and the gap below, and T_3 is inserting a record into this gap. This is an inherent shortcoming of ARIES/IM, and would affect any system that uses ARIES/IM.

Next, we will look at the more interesting case for range locking: where T_i inserts x and a range conflict is evident at the time of insert. This procedure is outlined in Algorithm 18, Insert-Range-Check.

Let T_i be a transaction that inserts x. In preparation for the insert, T_i positions a cursor immediately over data item y, where y is the record immediately before x in dictionary order. Lines 4–5 collect the set of LCBs for z = NEXT(y), the data item immediately after y in dictionary order (i.e., T_i will insert x between y and z). In these steps, our goal is to identify LCBs that cover the gap that x is being inserted into; these are $o_j(z)$ LCBs with the LOCK_ORDINARY or LOCK_GAP bits set. (LOCK_REC_NOT_GAP LCBs are not included, since they do not cover the gap below z.) If there are no conflicting LCBs on z, then T_i may insert x (with an implicit lock) and return.

The rest of Algorithm 18 handles the case where a range conflict is apparent. Lines 11–16 examine the set of z LCBs, looking for a $w_k(z)$ that would warrant FUW handling (i.e., a write lock on the next key z, held by a concurrent transaction T_k). If such a $w_k(z)$

```
1:  procedure Insert-Range-Check(Transaction T_i, Data Item x)
2:      acquire x-latch on data page
3:      let y = data item such that x will be inserted immediately after y
4:      let z = NEXT(y)
5:      let S = set of LCBs on z with LOCK_ORDINARY or LOCK_GAP
6:      if S = ∅ then                                          ▷ no range conflict
7:          insert x_i                                         ▷ T_i has implicit lock on x
8:          release x-latch on data page
9:          return
10:     end if
11:     if there is an LCB ∈ S that would warrant FUW handling then
12:         enqueue w_i(x) with
13:             LOCK_INSERT_INTENTION | LOCK_GAP | LOCK_X
14:         release x-latch on data page
15:         return                                             ▷ caller waits, with FUW handling
16:     end if
17:     insert x, immediately after y                          ▷ range conflict, but no FUW handling
18:     enqueue w_i(x) with LOCK_REC_NOT_GAP | LOCK_X
19:     Insert-Inherit-Locks(T_i, x, z)
20:     release x-latch on data page
21: end procedure
```

Algorithm 18: Range Checking for Newly-Inserted Data Items

exists, then T_i will enqueue a $w_i(z)$ with mode LOCK_INSERT_INTENTION | LOCK_GAP | LOCK_X; this is the GXI lock in Table 1. Enqueueing this lock request will cause T_i to enter lock wait (T_k is still active), or abort immediately (T_i, T_k concurrent, and T_k has already committed). If the insert does not require FUW handling, then T_i proceeds to lines 17–20; T_i inserts x, enqueues a $w_i(x)$ lock with LOCK_REC_NOT_GAP and calls procedure Insert-Inherit-Locks (Algorithm 19). The lock manager needs to know x's heap before it can enqueue $w_i(x)$, but the heap number is not known until after x is inserted; thus, x must be locked after insert.

```
1: procedure Insert-Inherit-Locks(Transaction T_i, Data Item x, Data Item z)
2:     let S = set of o_j(z) LCBs with LOCK_ORDINARY or LOCK_GAP
3:     for each lcb ∈ S do
4:         let T_j = lcb.txn                          ▷ T_j is an active or zombie transaction
5:         enqueue o_j(x) with LOCK_ORDINARY          ▷ x inherits lock mode from z
6:     end for
7: end procedure
```

Algorithm 19: Lock Inheritance, Post Insert

Insert-Inherit-Locks takes two data items: x, the newly-inserted record, and z, the data item immediately above the gap where x was inserted. For each conflicting LCB on z, T_i creates an $o_j(x)$ with the LOCK_ORDINARY gap bit. Thus, if T_j had locked the gap between y and z (recall that y is the record immediately before x in dictionary order), then Insert-Inherit-Locks guarantees that T_j will have locks on the gap between x and z, a lock on x itself, and a lock on the gap between y and x.

Note that T_i acquires a page latch at the beginning of Algorithm 18, and holds the latch for the duration of the procedure. This latch prevents a concurrent T_j from modifying the page while T_i inserts x and checks for range conflicts. This latch is an essential part of range locking, since it ensures that the notion of "next record" does not change while the insert occurs. Example 4.7 illustrates the case of inserting a record which conflicts with an earlier range scan.

Example 4.7 (*Range Scan Before Insert*): Suppose we have a unique index with keys "A" and "F". T_1 will scan "between A and D", T_2 will insert "C", and both transactions will commit. The following sequence of events takes place.

1. T_1 enqueues $r_1(A)$ with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S. (As before, this is a unique index and "A" matches the beginning of the scan range; so, T_1 can lock "A" without locking the gap below.)

2. T_1 enqueues $r_1(F)$ with LOCK_ORDINARY | LOCK_REC | LOCK_S. "F" is the first non-matching key; this record does not affect the outcome of T_1's query, but since the record was accessed, T_1 must lock it.

3. T_2 positions a cursor above "A". ("C" will be inserted after "A".)

4. T_2 looks for locks on NEXT(A) = F, and finds (only) $r_1(F)$ with LOCK_ORDINARY | LOCK_REC | LOCK_S. This tells T_2 that there is no FUW violation, but the insert causes a range conflict.

5. T_2 inserts "C", immediately after "A".

6. T_2 enqueues $w_2(C)$ with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_X.

7. T_2 calls Insert-Inherit-Locks(T_2, C, F), which enqueues $r_1(C)$, on T_1's behalf.

We conclude this section with one final example, to illustrate the use of implicit locks in conjunction with secondary indexes.

Example 4.8 (*Implicit Locking, Secondary Index*): For this example, we will use a table TBL with three integer columns: PK (the primary key), C1 (with a non-unique secondary index, C1_IDX), and V1 (an un-indexed column). InnoDB uses logical addressing, where each secondary index record contains the concatenation of (1) the indexed columns and (2) the primary key; therefore, C1_IDX records have the form (C1, PK). Our example begins with the following table and index structure:

TBL, page p_1			
PK	C1	V1	*heap #*
7	4	0	*8*
8	4	0	*9*
9	5	0	*10*

C1_IDX, page p_2		
C1	PK	*heap #*
4	7	*21*
4	8	*22*
5	9	*23*

In this sample data, *heap #* does not denote a physical column; instead it denotes InnoDB's heap numbers (see Figure 12, page 68) for the individual records. During this example, we denote locked records in the form $(p : h)$ where p is a page number and h is a heap number; this is the same physical record specification that InnoDB uses. For example, $r_1(p_1 : 9)$ means T_1's read lock on page p_1, heap number 9, which is the TBL row (8, 4, 0); $r_2(p_2 : 21)$ means T_2's read lock on page p_2, heap number 21, which is the C1_IDX record (4, 7).

Let T_1 be a transaction that inserts the row (100, 4, 0) into TBL. This insert leaves TBL and C1_IDX in the following state:

TBL, page p_1			
PK	C1	V1	*heap #*
7	4	0	*8*
8	4	0	*9*
9	5	0	*10*
100	4	0	*111*

C1_IDX, page p_2		
C1	PK	*heap #*
4	7	*21*
4	8	*22*
4	100	*222*
5	9	*23*

Notice that two inserts have taken place: (100, 4, 0) in TBL (as heap number 111) and (4, 100) in C1_IDX (as heap number 222). When inserting these records, T_1 has to check

for conflicting range locks in *both* TBL and C1_IDX. For the sake of discussion, let us assume that T_1 observed no range conflicts with either record; both records have implicit locks coming from x.txid for $x = (100, 4, 0)$.

Now, let T_2 be a transaction (concurrent with T_1) that executes the query select count(*) from TBL where C1 = 4, which is an index-only query on C1_IDX. The following events take place.

1. T_2 reads the PAGE_MAX_TRX_ID field from p_2, and sees that T_1 was the last transaction to modify the page. (More specifically, T_1 is the transaction with the largest transaction id that modified p_2.) T_1 is an active transaction, and this tells T_2 two things: (1) T_2 will have to dereference the TBL row each time it accesses a C1_IDX record on p_2 (as secondary index records are not versioned, and may correspond to rows which are not visible to T_2), and (2) T_2 will have to check for implicit locks when accessing C1_IDX records.

2. T_2 fetches the first record (4, 7) from C1_IDX, and checks for an implicit lock. Using the primary key, T_2 fetches the TBL row $(7, 4, 0)$, and examines the transaction id in $(7, 4, 0)$'s row header. This row was written by a (non-zombie) transaction that committed prior to start(T_2), so there is no implicit lock.

3. T_2 enqueues a $r_2(p2 : 21)$ lock on (4, 7), with LOCK_ORDINARY | LOCK_REC | LOCK_S.

4. T_2 returns to the row (7, 4, 0), already determined to be visible to T_2. T_2 enqueues a $r_2(p1 : 8)$ lock on the row with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S. LOCK_REC_NOT_GAP is used because (7, 4, 0) was obtained via exact match on primary key.

5. T_2 reads C1 from the TBL row. In this row, C1 = 4, so T_2's "count(*)" becomes one.

6. T_2 fetches the next C1_IDX record, (4, 8). In doing so, T_2 performs actions similar to those in steps 2–5. T_2 will

   (a) Take a $r_2(p_2 : 22)$ lock on (4, 8), with LOCK_ORDINARY | LOCK_REC | LOCK_S.

   (b) Take a $r_2(p_1 : 9)$ lock on (8, 4, 0), with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S.

   (c) Read C1 = 4 from row (8, 4, 0), which increments T_2's "count(*)" to two.

7. T_2 fetches the next C1_IDX record, (4, 100) and checks for an implicit lock. T_2 uses the primary key to find the TBL row (100, 4, 0), examines the transaction id in (100, 4, 0)'s row header, and sees that (100, 4, 0) was written by a concurrent transaction T_1. T_2 checks for an explicit $w_1(p_2 : 222)$ lock on (4, 100), and finds none. T_2 enqueues two lock requests:

   (a) $w_1(p_2 : 222)$, converting T_1's implicit lock on (4, 100) to an explicit lock. This lock has LOCK_REC_NOT_GAP | LOCK_REC | LOCK_X.

   (b) $r_2(p_2 : 222)$, with LOCK_ORDINARY | LOCK_REC | LOCK_X.

8. T_2 returns to the TBL row (100, 4, 0). This row is not visible to T_2, and does not affect the result of T_2's "count (*)" query, but T_2 must lock (100, 4, 0) nonetheless, to ensure that the T_2--rw$\rightarrow T_1$ dependency can be noticed via conflicts on data items. Again, implicit-to-explicit lock conversion is necessary, and T_2 enqueues two more lock requests, one for T_1 and one for itself:

   (a) $w_1(p_1 : 111)$, with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_X.

(b) $r_2(p_1 : 111)$, with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S.

9. T_2 fetches the next record from C1_IDX, (5, 9). As before, T_2 dereferences the original row, (9, 5, 0), and examines the writer's transaction id. (9, 5, 0) was written by a (non-zombie) transaction that committed prior to start(T_2), so there is no implicit lock. T_2 enqueues $r_2(p_2 : 23)$ with LOCK_ORDINARY | LOCK_REC | LOCK_S.

10. T_2 returns to the original row, (9, 5, 0), and enqueues a $r_2(p_1 : 10)$ lock with LOCK_REC_NOT_GAP | LOCK_REC | LOCK_S.

11. T_2 reads C1 = 5 from (9, 5, 0). This is the first non-matching record T_2's range scan, so the scan ends with "count(*)" = 2.

As this example shows, a lot goes on when T_2 accesses a recently-modified secondary index page. If p_2.PAGE_MAX_TRX_ID denoted a transaction that was neither active nor a zombie, then T_2 would have been able to read p_2 as-is, without dereferencing TBL rows, or checking for implicit locks.

4.3.3 InnoDB's Kernel Mutex

Despite its many positive qualities, InnoDB's lock manager is not without drawbacks. One notable drawback is InnoDB's use of a single, monolithic mutex (called the *kernel mutex*) to protect the lock manager and transaction system. The kernel mutex serializes access to InnoDB's lock manager, which obviously affects the lock manager's scalability. For example, a design that used one mutex per lock table cell would allow T_i and T_j to set locks at the same time, as long as those locks reside in different hash table cells; by contrast, InnoDB's kernel mutex serializes these operations. (Note that locks in different

hash table cells affect different records, and therefore, cannot conflict.) InnoDB's kernel mutex limits scalability in other ways: for example, T_i cannot set a lock while InnoDB is assigning a start timestamp to T_j, since the kernel mutex covers both the transaction system and lock manager.

Comments in InnoDB's source code indicate that development started during the mid 1990's, when uni-core architectures were (arguably) the most common. In that context, a monolithic mutex shouldn't have much negative affect on system scalability; however, contemporary multi-core hardware would benefit from more granular mutexes. InnoDB's developers are aware of this situation, and plan to remove the kernel mutex in MySQL 5.6 [ORA11].

In our PSSI prototype (based on MySQL 5.1.31), the kernel mutex protects the lock manager, the transaction system, and the cycle testing graph. The lock manager and cycle testing graph are closely integrated, so this implementation decision was consistent with InnoDB's existing design. Unfortunately, protecting the CTG with the kernel mutex increases the probability of the kernel mutex becoming a bottleneck (we will see an example of this in Chapter 5).

4.4 Cycle Testing Graph

At an implementation level, PSSI's CTG is a typical directed graph. Graph nodes are C-language transaction structs, organized in a hashtable, and keyed by transaction id. Each transaction struct has an array of out-edges and a count of in-edges, as described in Section 3.4.

The challenging aspect of implementing the CTG came in changing InnoDB's transaction lifecycle model. MySQL has a data structure called a THD ("connection thread") which packages together (1) a client connection to the database server, (2) an operating system thread, and (3) a transaction struct to execute the client's SQL statements. Each connection is bound to a single server thread, a server thread is associated with a single transaction struct, and the transaction struct is re-used for each of the connection's transactions. This arrangement is a good fit for S2PL, where information about T_i can be discarded as soon as T_i commits, but unfortunately, this is not a suitable arrangement for PSSI. PSSI must keep track of zombie transactions for some period of time after they commit, which means that T_i's transaction struct must be dissociated from the connection thread after commit(T_i) (i.e., the connection thread may need to execute a new transaction T_j while T_i is still a zombie). To handle this situation, we modified InnoDB as follows:

1. We added a pool of transaction structs to InnoDB's transaction manager. These are inactive, non-zombie transaction structs waiting to be used. Having a pool avoids allocation overhead each time a transaction starts, and deallocation overhead each time a transaction ceases to be a zombie.

2. When a connection thread needs to start a new transaction, the connection thread acquires a transaction struct from the pool.

3. When T_i commits, T_i's transaction struct is dissociated from the connection thread; the struct becomes the property of the CTG, until the CTG determines that T_i is no longer a zombie. Pruning occurs after T_i's transaction struct is dissociated from its connection thread.

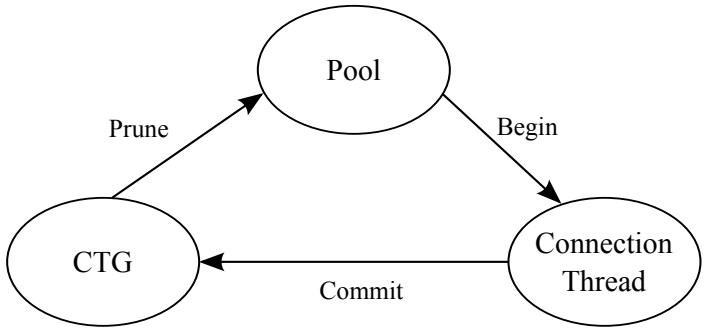

Figure 14: Lifecycle of Transaction Structs

4. When T_i is no longer a zombie (as determined by CTG-Prune), the CTG returns the struct to the pool (step 1) for later re-use.

A single transaction struct undergoes a cycle of re-use, as illustrated in Figure 14.

Our CTG contains two optimizations to speed up pruning (Algorithm 14, page 59). Pruning requires the CTG to identify a set of zombie transactions S where each $T_i \in S$ has no in-edges, and committed before the oldest active transaction started. From the transaction manager, we can easily obtain t_0, the start timestamp of the oldest active transaction, but identifying the members of S requires more care, as we wish to avoid a brute-force search of CTG nodes. Our CTG reduces this search space with an auxiliary linked list, called ctg.commit_ts_order; this a linked list of zombie transactions, ordered by commit timestamp.

CTG-Prune-Initial-Set (Algorithm 20) shows how the CTG uses commit_ts_order to identify the initial set S of prunable transactions. Given t_0 (the start timestamp of the oldest active transaction), CTG-Prune-Initial-Set walks down the list of transactions in commit_ts_order, collecting those that committed prior to t_0, and stopping at the first

```
1:  procedure CTG-Prune-Initial-Set(timestamp t_0)
2:      let S = ∅
3:      for T_i ∈ ctg.commit_ts_order do
4:          if commit(T_i) > t_0 then
5:              break
6:          end if
7:          if T_i.in_edge_count == 0 then
8:              S = S ∪ T_i
9:          end if
10:     end for
11:     return S
12: end procedure
```

Algorithm 20: Finding the Initial Set of Prunable Transactions

transaction that committed after t_0. In our experiments, the if-condition in lines 4–6 is the more important criterion. The test in lines 7–9 tends to be less selective; in low contention workloads, there may be few transactions with in-edges.

The second pruning optimization comes from the following observation: let p_1 and p_2 be two successive pruning operations where t_0 is the start timestamp of the oldest active transaction during p_1, and t_0' is the start timestamp of the oldest active transaction during p_2. If $t_0 = t_0'$, then the oldest active transaction has not changed, and p_2 will not remove any zombies from the CTG. This optimization is implemented by caching t_0 between successive calls to CTG-Prune; if the current t_0 matches the last-cached t_0, then CTG-Prune can return immediately, without doing any further work.

4.4.1 CTG Support for ESSI

As explained in Section 2.3, every non-serializable SI history contains a dangerous structure: three transactions T_1, T_2, T_3 (perhaps with $T_1 = T_3$), where T_1 and T_2 are concurrent, T_2 and T_3 are concurrent, and there are dependencies T_3--rw$\rightarrow T_2$ and T_2--rw$\rightarrow T_1$. This provides an alternate way of ensuring serializability for SI; instead of aborting transactions that cause dependency cycles, one could abort transactions that cause dangerous structures. Essential Dangerous Structure Testing (ESSI) is a refinement to this approach, which aborts transactions that create dangerous structures where T_1 commits first (see Definition 2.24, page 27). One of our goals for Chapter 5 will be to compare the performance of PSSI and ESSI, and in this section, we describe our implementation of ESSI. Our implementation of ESSI differs from that of [CRF09]; instead of testing for a dangerous structure with every record access, our implementation tests for dangerous structures at commit time, using the CTG. In other words, our prototype provides ESSI by replacing a commit-time test for cycles with a commit-time test for essential dangerous structures.

Figure 15 shows a diagram of an essential dangerous structure (this diagram appeared earlier, as Figure 6 in Chapter 2). In an essential dangerous structure, T_1 is the first transaction to commit. This leaves us to handle two cases: where the essential dangerous structure would formed by the commit of T_2, and where the essential dangerous structure would be formed by the commit of T_3.

The second case (T_3 commits last) is easy for the CTG to detect, as we can use T_3.out_edges to find T_2, and T_2.out_edges to find T_1. However, the first case (T_2 commits last) poses a problem, since the CTG (as presented so far), does not provide a way to

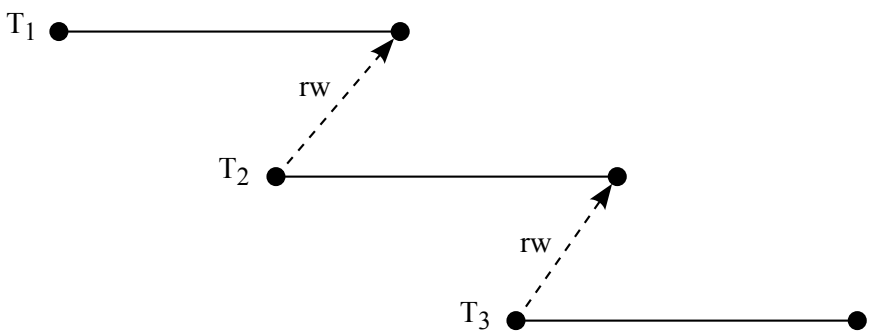

Figure 15: An Essential Dangerous Structure (Figure 6 repeated)

traverse backwards along directional edges. We solved this problem by having the CTG maintain both in-edge and out-edge sets when ESSI is in use, so that T_i.out_edges represents the set of T_j such that $T_i \rightarrow T_j$, and T_i.in_edges represents the set of T_k such that $T_k \rightarrow T_i$. Thus, we can detect essential dangerous structures where T_2 commits last by using T_2.out_edges to find T_1, and by using T_2.in_edges to find T_3. The dual edge sets are only used for ESSI; PSSI continues to use T_i.out_edges and T_i.in_edge_count.

The pruning criteria for ESSI and PSSI also differ. As shown in Theorem 3.5, PSSI can prune a zombie transaction T_i as long as (1) T_i has no in-edges, and (2) $\text{commit}(T_i) < t_0$, where t_0 is the start timestamp of the oldest active transaction. By contrast, ESSI can prune a zombie transaction T_i that meets only condition (2), having $\text{commit}(T_i) < t_0$ [CRF09, pg. 26]. Our PSSI prototype uses the same CTG pruning code for ESSI and PSSI; the code simply ignores in-edge counts when ESSI is used.

4.5 Durability, Replication, and Garbage Collection

This section addresses three miscellaneous implementation topics: durability, data replication, and garbage collection.

4.5.1 Durability

Durability is described by the property whereby a transaction T_i's state transformations are made durable and public, after T_i successfully commits [GR92, pg. 17]. Like many other systems, InnoDB supports durability through the write-ahead logging (WAL) protocol, in combination with log flush at commit. During commit, T_i is assigned a commit log sequence number, commit-LSN(T_i), and InnoDB's write-ahead log is flushed up to commit-LSN(T_i) before commit(T_i) completes. As distributed, InnoDB employs a variant of early lock release, whereby T_i releases its locks after commit-LSN(T_i) is assigned, but before T_i's logs are flushed to disk. Comments in the InnoDB source code rationalize this design choice as follows: if T_1 and T_2 are update transactions, and T_2 is waiting for T_1, the we are guaranteed to have commit-LSN(T_2) $>$ commit-LSN(T_1). Therefore, if a crash prevents T_1's logs from being flushed to disk, then T_2's logs will not be flushed to disk either. The monotonicity of commit LSNs ensures that the database cannot be corrupted by early lock release, but this design offers no protection for reader transactions (i.e., transactions that wrote no data). If a reader transaction T_3 was waiting for T_1, then it is possible for T_3 to read values written by T_1 (and subsequently for T_3 to commit) before T_1's logs are flushed to disk. If the database were to crash before T_1's logs were flushed, then recovery would undo T_1, and T_3 would have read values that never existed.

We surmise that InnoDB takes this calculated risk in order to improve performance. The standard distribution InnoDB achieves serializability through strict two-phased locking, so holding locks during a log flush will increase wait times and reduce system throughput. Holding locks during a log flush is less of an issue for PSSI (i.e., reads and writes do not block each other), and we modified InnoDB's commit sequence so that T_i's locks are not released (and T_i's commit timestamp is not assigned), until after T_i's logs have been flushed to disk. From the standpoint of durability, this is a safer approach. We also note that other researchers have made similar changes when using InnoDB for their research projects [CRF09, pg. 23]. In Chapter 6, we discuss a strategy that would allow early lock release, while preserving the durability guarantee for reader transactions.

4.5.2 Replication

Database replication is a popular feature of MySQL. Replication was not our main area of research (and we did not implement replication support for our PSSI prototype), but we feel that it is important to mention some of PSSI's implications for this area.

MySQL replication is implemented above the storage engine layer; MySQL can support database replication in any storage engine, so long as the storage engine provides the necessary API support. MySQL's database replication works as follows: when a transaction T_i commits, all of T_i's changes are written to a binary replication log, which is separate from the storage engine's (i.e., InnoDB's) transaction log. Once replication logs have been written on the master node, the logs are copied to one or more slave nodes, and replayed in serial order. The idea is that we start with two identical database nodes N_1 and N_2, apply a history H to N_1 (the master) and an equivalent serial history $S(H)$ to N_2 (the slave), and these transformations leave N_1 and N_2 in identical final states.

MySQL offers two forms of replication: statement-based, and row-based. *Statement-based* replication replicates entire insert, update and delete statements, while *row-based* replication replicates changes on a row-by-row basis. MySQL's row-based replication works well with PSSI's semantics, but statement-based replication does not.

Recall that MySQL writes its replication log during transaction commit; for statement-based replication, this assumes that the order of commits in a history H forms a conflict-equivalent serial history $S(H)$. This assumption holds for S2PL, but not for PSSI. (In S2PL's case, this assumption comes from a property called *external consistency*, which is discussed in Section 6.2.) In order to have MySQL's statement-based replication work with PSSI, one would need to write replication logs as transactions are pruned from the CTG, since CTG pruning order (and not commit order) gives an equivalent serial history.

MySQL's row-based replication is compatible with PSSI, even if rows are written to the binary replication log during transaction commit. For an individual row, FUW guarantees that changes are made in a well-defined order, by non-concurrent transactions. Therefore, the version ordering on a slave node will be identical to the version ordering on the master node.

4.5.3 Garbage Collection

Given InnoDB's support for MVCC, there are already mechanisms for garbage-collecting old versions of rows; InnoDB can remove an old version x_i from its rollback segment, provided that no active transaction T_j can read it. PSSI required a slight modification to InnoDB's garbage collection semantics: x_i can be garbage-collected, provided that there is

no active *or zombie* transaction T_j that can read it. The motivation for this change is best illustrated through an example.

Suppose that T_i deletes x and commits, creating a dead version x_i, and let T_j be a transaction where $\text{start}(T_j) > \text{commit}(T_i)$. A T_j query may depend on the deleted data item, and although T_j cannot "see" x_i, T_j may need to lock x in order to notice a T_j--rw$\rightarrow T_i$ dependency. As long as T_i is a zombie, any T_j--rw$\rightarrow T_i$ dependency can contribute to a future cycle; therefore, x_i must be lockable (and cannot be garbage collected) while T_i is still a zombie.

This is a conservative change to InnoDB's garbage collection algorithm, which ensures that dead x_i are not garbage-collected too early. However, old versions may be retained for longer than necessary, and there are opportunities for future optimizations here.

4.6 Summary

This chapter presented a case study of how PSSI can be implemented in a real-world database system, based on our experience with adding PSSI support to MySQL's InnoDB storage engine. This is not a trivial effort, and the work touches many areas of the DBMS. The goal of this chapter was to highlight some of the challenges faced, and draw attention to some of the subtle issues encountered.

CHAPTER 5

PSSI PERFORMANCE STUDIES

This chapter presents a performance evaluation of PSSI. We begin with a definition of SICycles, a benchmark that we developed for testing PSSI. Next, we provide a description of our testing environment, and the test parameters used. Finally, we present our test results.

5.1 The SICycles Benchmark

Database systems are often evaluated using TPC-C, which is a well-established industry-standard benchmark, developed by the Transaction Processing Council [TPC10]. Unfortunately, TPC-C is not a very good benchmark for evaluating PSSI. As shown in [FLO05], TPC-C executes serializably when run under (ordinary) snapshot isolation, which means that there are no dependency cycles for PSSI to avoid. This led us to develop a new benchmark called SICycles [ROO11]. SICycles is a benchmark that allows cycles to form in a variety of ways, tunable through workload parameters.

SICycles uses a one-million row bench table, which is modeled after the table used by the Set Query Benchmark [ON93]. The bench table has 20 integer columns (NOT NULL), and one character column (also NOT NULL), which are described below:

- kseq. kseq is a sequence of integers from 1 to 1,000,000, and the primary key of the table.

- krandseq. krandseq is kseq shuffled; integers from 1 to 1,000,000 in random order. krandseq has a unique index, and all rows are accessed through krandseq.

- kval. kval is an unindexed integer column, whose values are randomly chosen from the range 10,000 to 99,999. kval is the only column updated.

- kpad. kpad is an un-indexed character column, 20 bytes long. kpad is neither read nor written; it is padding to bring each record up to 100 bytes in length.

- kn columns: k4, k8, k16, k32, k64, k128, k256, k512, k1024, k2500, k5k, k10k, k25k, k50k, k100k, k250k, k500k. Each kn column contains values $[1,n]$, randomly assigned to rows. (In the knk columns, nk stands for $n \times 1000$ rows; for example, k5k contains values [1, 5000].) The test results presented in this chapter do not make use of the kn columns, but we document them here for completeness.

All columns are indexed, with the exception of kval (the only column updated) and kpad (which is never accessed).

An SICycles run is described by an integer pair (k,n) for $k \geq 1$, $n \geq 1$, and by a hotspot size h. Each transaction T_i reads k rows from the set $X = \{x_1, \ldots, x_k\}$, finds the average x.kval value v, and adds a fraction of that average $c \times v$ (or $-c \times v$, chosen at random) to each row in the set $Y = \{y_1, \ldots, y_n\}$. The set of rows $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_n\}$ are chosen from a randomly-distributed hotspot H (containing h rows), so that $X \cap Y = \emptyset$. The fraction c is a constant, $c = 0.001$. Roughly half of transactions add

$c \times v$ to rows in Y, while the other half adds $-c \times v$; therefore, we expect the set of kval values to stay approximately the same over the course of several runs.

The two procedures in Algorithm 21 provide pseudocode for the SICycles benchmark. Procedure Hotspot-Setup is a global routine which chooses the set of "hot" rows H. All reads and writes use rows in H, and the set of hot rows is common to all client threads. Procedure SICycles-Transaction gives pseudocode for a single SICycles transaction T_i. T_i begins by choosing its read set X, its write set Y, and determining whether to use $c = 0.001$ or $c = -0.001$. Next, T_i sums x.kval values in X, computes delta = $c \times \text{avg}(x.\text{kval})$, and adds delta to each y.kval value in Y. Upon completion of these steps, T_i issues a commit statement. The "random think-time" steps in Algorithm 21 denote random client-side delays of 3 ms $\pm$ 50%. These random delays prevent the system from being saturated prematurely, and allow us to report a wider range of results. Note that there is no think time between the final update statement and transaction commit.

For $k = n = 1$, T_i reads x.kval, and adds $c \times x$.kval (or $-c \times x$.kval) to y.kval, and we can say that "T_i copies a portion of x to y, leaving x unchanged". If T_i reads x and writes y while a concurrent T_j reads y and writes x, then we have a write skew cycle of length two, similar to the one shown in Figure 4 (page 18). If T_i reads x and writes y, T_j reads y and writes z, and T_k reads z and writes x, then we have a cycle of length three. For $k > 1$, $n = 1$ it is possible for cycles to form in many different ways as the multi-programming level (MPL) is increased. Under contention, higher read-to-write ratios tend to cause more T_i--rw$\rightarrow T_j$ anti-dependencies between pairs of concurrent transactions, and a large number of these anti-dependencies increases the likelihood of cycle formation. This topic is further discussed in Section 5.4.

```
1:  procedure Hotspot-Setup(hotspot size h)
2:      H = choose h distinct krandseq values, with uniform distribution
3:  end procedure
4:
5:  procedure SICycles-Transaction(reads k, writes n, random r)
6:      let X = select k distinct values from H
7:      let Y = select n distinct values from H, such that X ∩ Y = ∅
8:      let c = 0.001
9:      if r < 0.5 then                          ▷ r is a random value, r ∈ [0, 1)
10:         let c = −0.001
11:     end if
12:     let sum = 0
13:     for x ∈ X do
14:         sum += (select kval from bench where krandseq = :x)
15:         random think-time
16:     end for
17:     let delta = c × (sum/k)
18:     for y ∈ Y do
19:         update bench set kval = kval + :delta where krandseq = :y
20:         if y is not the last element in Y then
21:             random think-time
22:         end if
23:     end for
24:     commit
25: end procedure
```

Algorithm 21: Pseudocode for SICycles Benchmark

It is worth calling attention to two aspects of SICycles. First, all row access occurs through krandseq, a non-clustered index; this doubles the number of lock control blocks (LCBs) in InnoDB's lock manager. We intentionally chose krandseq, to put more stress on the lock manager. Consider a row having (kseq, krandseq, kval) values (32, 500, 21000), and a T_i that wishes to read kval where krandseq = 500. T_i uses the following data access pattern:

1. T_i locates the record in krandseq's index whose key is 500. InnoDB uses logical addressing, so the value of the index record is the primary key, 32.

2. T_i locates the PRIMARY index record with kseq = 32 and reads kval.

Each of these steps enqueues an LCB in the lock manager.

Second, the hotspot H is randomly distributed over the bench table, and not confined to a contiguous range of krandseq values. This gives a more even distribution of LCBs to hashtable cells. Recall from Section 4.3 that InnoDB's lock manager hashes LCBs on the basis of page number, so a contiguous hotspot would create artificially long lock chains in a small number of hashtable cells. Each krandseq index record is eight bytes long: four bytes for krandseq and four bytes for kseq, the primary key. With a 16kb page size, approximately 2000 krandseq index records fit on a single secondary index page. If we chose a hotspot of 1000 contiguous krandseq values, then all of those LCBs would lie on one or two secondary index pages, and hash to one or two cells in the lock manager's hash table.

5.2 SICycles Experimental Setup

We ran our tests using a two-machine client/server setup. The server is an HP Z400 workstation with a quad-core 2.66 GHz Intel 3520 CPU, 4GB of RAM, running OpenSUSE Linux 11.4, kernel version 2.6.37. The mysqld server comes from a production (non-debug) build of MySQL 5.1.31, compiled with gcc 4.5.1. Table data was stored on a Western Digital WDC-WD3200BEK 7200 RPM SATA disk, and logging was done to an Intel X25-E solid state disk (SSD). All filesystems were formatted as ext4, mounted with the noatime option, and the X25-E's write cache was disabled. With its write cache disabled, the X25-E can perform an fsync in approximately 2 ms, which provides good logging performance.

The client machine was a Lenovo ThinkPad with a dual-core 2.53 GHz Intel Core-2 Duo CPU, 2 GB of RAM, also running OpenSUSE Linux 11.4. The SICycles client is written in Java, and run with Sun JVM 1.6.0_26, and MySQL's Connector/J version 5.1.1. The client and server were directly connected to a gigabit ethernet switch, to minimize network latency.

All tests use the mysqld configuration parameters shown in Table 2. The most significant configurations parameters are described below, and full vendor documentation can be found at [ORA09, Sec. 13.6.3].

- **innodb_flush_log_at_trx_commit = 1**. This configuration parameter provides durability; InnoDB will not return an acknowledgment of T_i's commit until T_i's logs have been flushed to disk. In this configuration, InnoDB flushes logs with a group commit algorithm (i.e., [GR92, sec. 9.4.7]).

Configuration Parameter	value
innodb_additional_mem_pool_size	2M
innodb_flush_log_at_trx_commit	1
innodb_thread_concurrency	0
innodb_buffer_pool_size	1G
innodb_log_file_size	256M
innodb_log_buffer_size	16M
innodb_flush_method	fsync

Table 2: mysqld Configuration Parameters Used in Testing

- **innodb_thread_concurrency = 0**. InnoDB thread concurrency controls the number of threads concurrently executing inside the InnoDB storage engine. A value of zero means "no limit".

- **innodb_buffer_pool_size = 1G**. This configuration parameter specifies a one-gigabyte InnoDB buffer pool. One gigabyte provides enough room for table data to remain buffer-resident for the duration of the test.

- **innodb_flush_method = fsync**. This configuration parameter tells InnoDB to flush logs using the fsync system call.

SI, PSSI, and ESSI results come from a single mysqld binary: our PSSI prototype, which is a modified MySQL/InnoDB 5.1.31. S2PL results come from a standard distribution MySQL/InnoDB 5.1.31 (a separate binary), with modifications described in Remark 5.1.

Remark 5.1 (*Modifications to Standard Distribution InnoDB*): Section 4.5.1 stated that our PSSI prototype releases locks and assigns commit timestamps after T_i's logs are flushed to disk. For comparison with S2PL, we made the same modifications to the standard distribution InnoDB, so that S2PL also releases locks after log flush. This change was implemented by reordering steps in the InnoDB function trx_commit_off_kernel. No other modifications were made to the standard distribution InnoDB.

Each data point comes from the median of three test runs, and each test run consists of the following steps:

1. Start the mysqld server.
2. Perform a series of select statements to bring needed pages into buffer. Our objective was to have mysqld's working set in buffer for the duration of the test.
3. Perform a 70-second warmup, followed by a 60-second measurement period, followed by a five-second cool-down.
4. Stop the mysqld server.

Our test runs varied the number of reads *k*, the number of writes *n*, and the size of the hotspot *h*. We denote these parametrizations with the shorthand "s*k*u*n*-*h*" (or s*k*u*n*, when a specific hotspot size is not relevant). For example, s3u1-800 means "select 3, update 1, using an 800-row hotspot", while s5u1 means "select 5, update 1" (without referring to any particular hotspot size). Our tests use no think time between transactions, so that MPL is consistent throughout the measurement period. As stated in Section 5.1, the SICycles client introduces 3 ms $\pm$ 50% of think time between SQL statements; these think-time delays prevent the system from becoming saturated too soon, and allows us to report on a

wider range of MPL values. (Of course, these think times also create a lower bound on transaction duration; Section 5.3.3 discusses this further.)

This experimental setup is a departure from the one we used in [ROO11]. The differences are explained in Remark 5.2.

Remark 5.2: The test results we published in [ROO11] had InnoDB logging to a 7200 RPM hard disk drive, and configured to use (non-durable) asynchronous log flushes (innodb_flush_log_at_trx_commit = 0). We used this configuration to demonstrate good CTPS performance, and to compensate for (as we believed at the time) poor performance of InnoDB's group commit algorithm. The poor group commit performance we observed was the result of MySQL Bug 13669 [ORA05], *Group Commit is broken in 5.0*, which was reported in September 2005, but not fixed until after the release of MySQL 5.1.31.

At the time of writing [ROO11], we were not aware that certain MySQL configurations could avoid Bug 13669 entirely; specifically the effects of Bug 13669 disappear if one disables MySQL replication logging. (The [ROO11] tests were, in fact, run with replication logging disabled. We began our [ROO11] experiments both replication logging and group commit enabled; we switched to asynchronous logging to avoid the poor group commit performance, and then decided to eschew replication logging, since replication logging is not necessary for a single-node system. We did not realize that turning off replication logging would have the side effect of "fixing" the poor group commit performance we observed.)

Nothing here invalidates the experimental results obtained in [ROO11]. Rather, [ROO11]'s experimental results were obtained with a different system configuration than the experimental results presented here; the purpose of this Remark is to explain why the configurations are different. As stated earlier, the results in this chapter were obtained by

configuring InnoDB to write transaction logs to an enterprise-grade SSD, using durable group commit (and by disabling replication logging to prevent the effects of Bug 13669).

5.3 Test Results

Our performance tests used three different workload configurations (s5u1, s3u1, s1u1), four different hotspot sizes (200, 400, 800, and 1200 rows), with MPL varying from 1 to 100. We tested four isolation levels: S2PL, SI, ESSI, and PSSI. Of these four isolation levels, S2PL, ESSI, and PSSI are serializable, but SI is not. We have included SI measurements because they provide a useful point of reference, as SI is implemented in several popular database systems. Each of the following sections focuses on a different set of performance metrics: throughput, abort rate and abort type, transaction duration, and CTG characteristics.

5.3.1 CTPS Throughput

The first metric we examine is throughput, measured in committed transactions per second (CTPS). CTPS is influenced by two factors: transaction duration, and the frequency of transaction aborts.

5.3.1.1 s5u1 CTPS measurements

Figure 16 gives CTPS measurements for the s5u1 workload tests. Among the four hotspot sizes, we can observe several distinct trends. First, SI provides this highest throughput; we see this result because SI delays transactions only to prevent FUW violations, and SI does not prevent non-serializable histories from occurring (i.e., SI allows dependency cycles to

form). Second, as the hotspot size increases, PSSI's performance approaches that of SI. This occurs because larger hotspots produce less contention, and dependency cycles are less likely to form. For example, at 80 MPL, SI gives 3183 CTPS for s5u1-400 while PSSI gives 2879 ($\Delta = -11\%$); SI gives 3413 CTPS for s5u1-800 while PSSI gives 3370 ($\Delta = -1.3\%$); and SI gives 3511 CTPS for s5u1-1200 while PSSI gives 3487 ($\Delta = -0.7\%$).

In general, larger hotspots produce higher CTPS measurements for any given isolation level. Larger hotspots reduce the probability of conflict between pairs of concurrent transactions. Fewer conflicts between pairs of concurrent transactions mean fewer lock waits under S2PL, fewer FUW aborts under SI, ESSI, and PSSI, and fewer concurrent T_i--rw$\rightarrow T_j$ anti-dependencies. Of course, it is the concurrent T_i--rw$\rightarrow T_j$ anti-dependencies that make it possible for essential dangerous structures and cycles to form (i.e., a large number of concurrent T_i--rw$\rightarrow T_j$ anti-dependencies will, in general, increase the serialization abort rate under ESSI and PSSI).

In Figure 16, we also observe PSSI achieving a higher throughput than ESSI. At 80 MPL, PSSI has 2879 CTPS for s5u1-400 while ESSI has 2413 ($\Delta = -19\%$); PSSI has 3370 CTPS for s5u1-800 while ESSI has 2998 ($\Delta = -12\%$); and PSSI has 3487 CTPS for s5u1-1200 while ESSI has 3247 ($\Delta = -7\%$). As we will see in Section 5.3.2, these differences are mostly attributable to ESSI's higher abort rates. ESSI aborts transactions that form essential dangerous structures (which are cycles of length two, or precursors to cycles), while PSSI always waits for complete cycles to form. In other words, ESSI may abort transactions that do not cause non-serializable histories, and these extra aborts reduce ESSI's throughput.

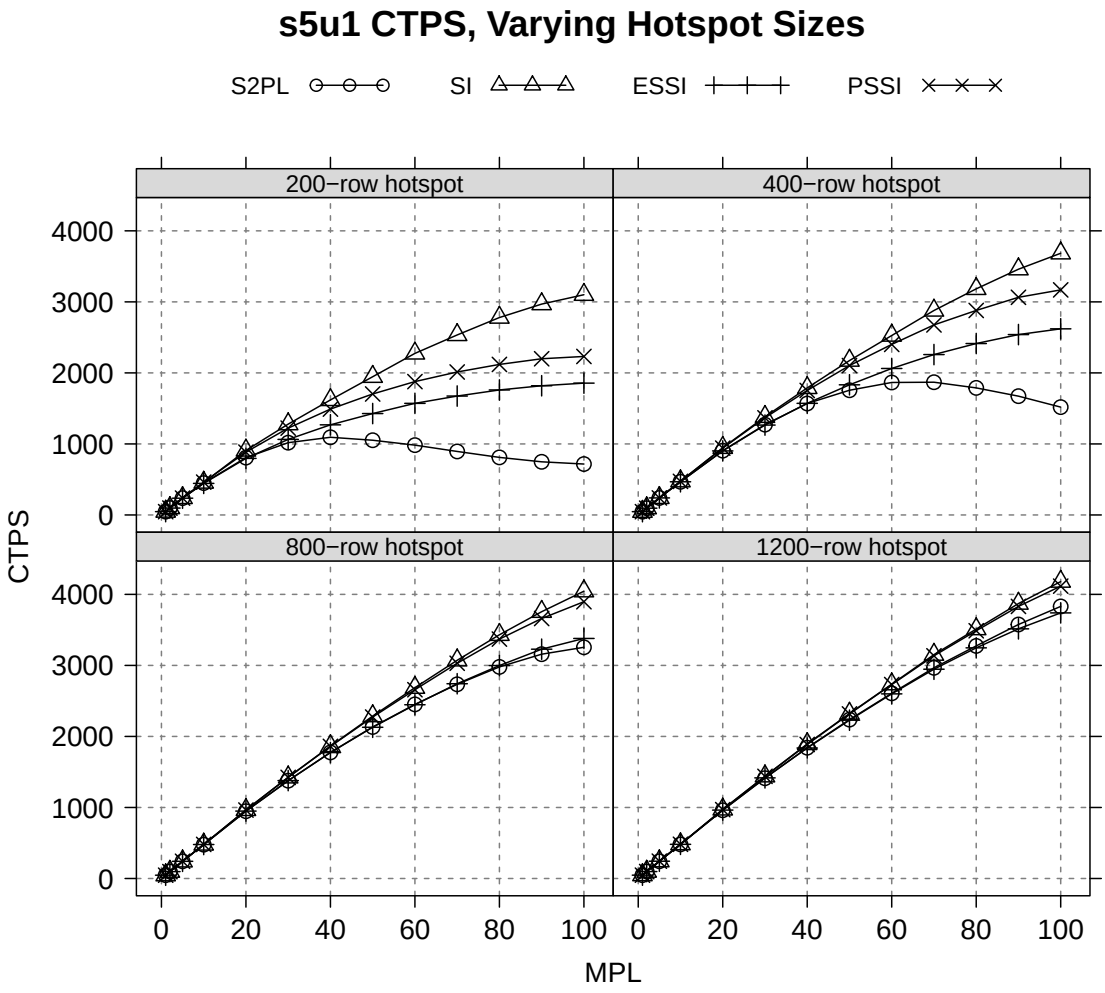

Figure 16: s5u1 CTPS, Varying Hotspot Sizes

The remaining isolation level is S2PL. S2PL's CTPS throughput is predominantly influenced by the frequency of lock waits, so S2PL throughput increases in workloads with lower data contention (i.e., larger hotspots). One notable aspect of S2PL's performance appears in the s5u1-1200 test, for MPL $\in [80, 100]$, where S2PL's performance exceeds that of ESSI. Here, the implication is that ESSI's unnecessary aborts

impede performance more than S2PL's lock waits. Another notable aspect of S2PL's performance is the negative slope in Figure 16's s5u1-200 and s5u1-400 measurements. These measurements show severe oversaturation, where the addition of client threads degrades, rather than improves performance. Here, many S2PL transactions are blocked, leading to long transaction durations, and a general inability of the system to move forward and make progress.

5.3.1.2 s3u1, s1u1 CTPS Measurements

Figure 17 shows s3u1 CTPS measurements. This first thing we observe is that s3u1 CTPS measurements are higher than their s5u1 counterparts; s3u1 transactions execute fewer SQL statements, and the per-transaction execution time is proportionally lower.

Our second observation is that the relative difference between isolation levels is smaller, which comes from reduced data contention: s3u1 has a lower read-to-write ratio than s5u1. This creates fewer T_i--rw$\rightarrow T_j$ dependencies between current transactions, and makes it less likely for cycles (and essential dangerous structures) to form. The likelihood of S2PL lock waits is similarly reduced.

Comparing SI and PSSI at 80 MPL, SI gives 4719 CTPS for s3u1-400 while PSSI gives 4571 ($\Delta = -3\%$); SI gives 5032 CTPS for s3u1-800 while PSSI gives 4996 ($\Delta = -0.7\%$); and SI gives 5143 CTPS for s3u1-1200 while PSSI gives 5122 ($\Delta = -0.4\%$).

Comparing PSSI and ESSI at 80 MPL, PSSI gives 4571 CTPS for s3u1-400 while ESSI gives 4024 ($\Delta = -14\%$); PSSI gives 4996 CTPS for s3u1-800 while ESSI gives 4718 ($\Delta = -6\%$); and PSSI gives 5122 CTPS for s3u1-1200 while ESSI gives 4959

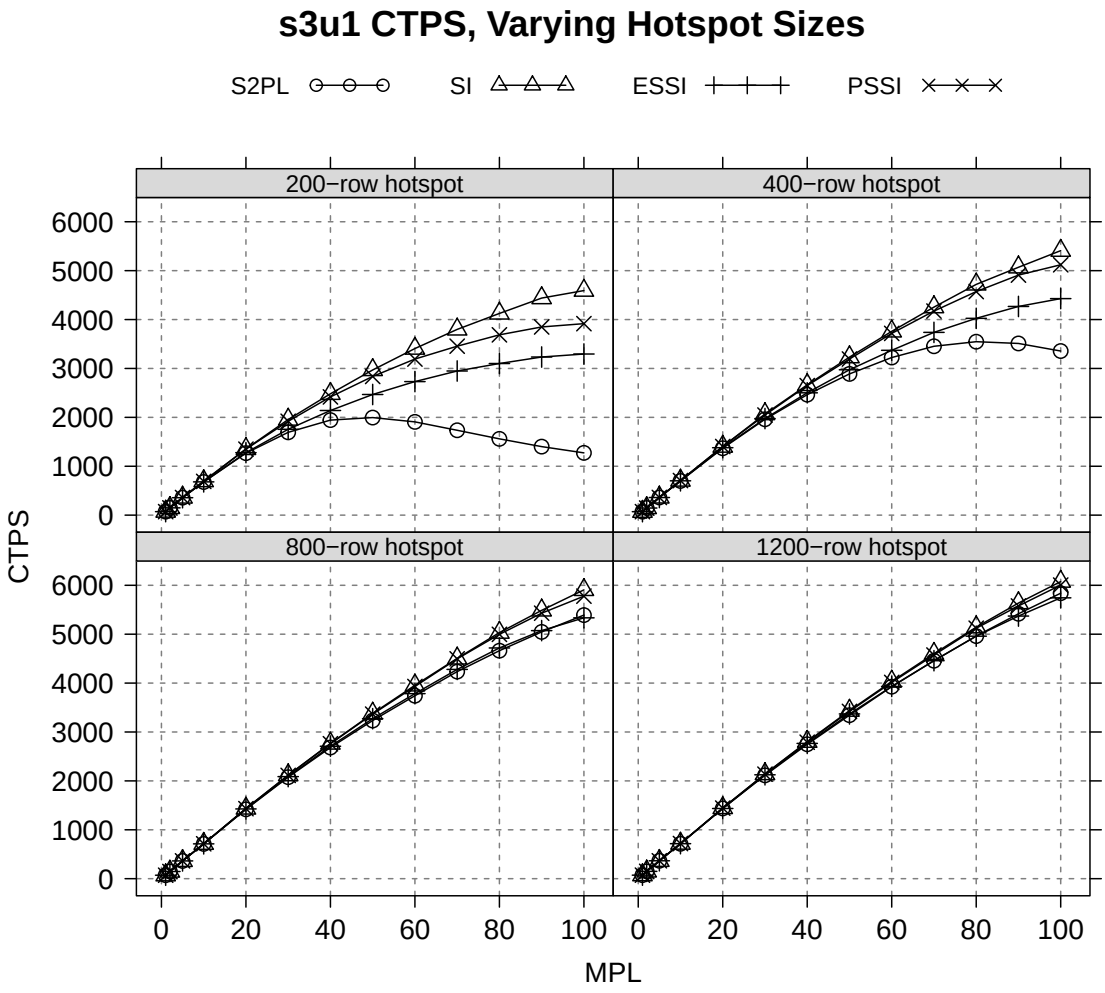

Figure 17: s3u1 CTPS, Varying Hotspot Sizes

($\Delta = -3\%$). In each case, the magnitude of s3u1 Δ's are smaller than their s5u1 counterparts.

Our final set of CTPS measurements appears in Figure 18, for an s1u1 workload. Of the s*k*u*n* workload configurations we tested, s1u1 has the lowest read-to-write ratio, and PSSI has the smallest advantage when compared to ESSI and S2PL. The most notable

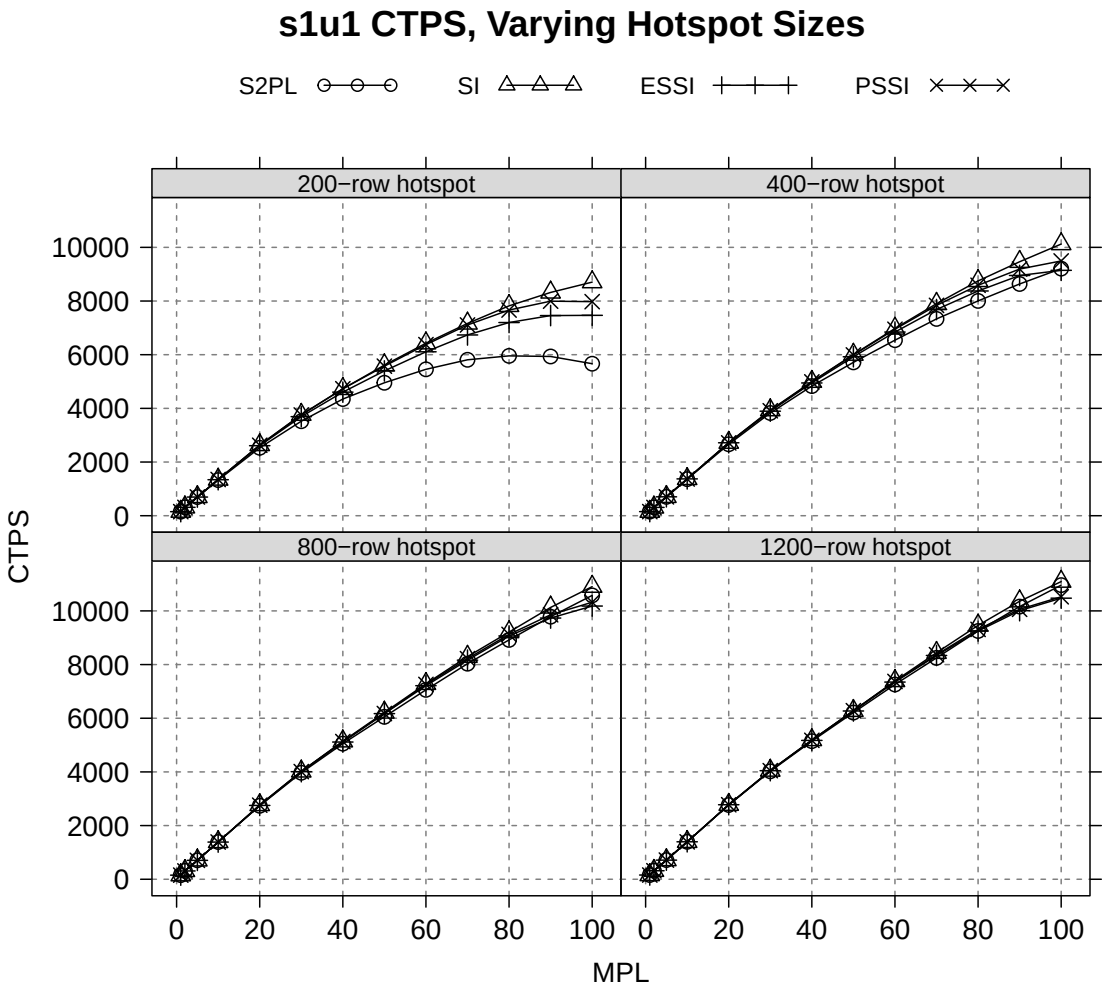

Figure 18: s1u1 CTPS, Varying Hotspot Sizes

findings from the s1u1 measurements appear with MPL $\in [90, 100]$ for ESSI and PSSI: at this point, ESSI and PSSI throughput begins to "flatline". We *believe* that this is caused by contention on InnoDB's kernel_mutex (see Section 4.3.3). At 100 MPL, PSSI and (our implementation of) ESSI need to find dependencies and prune the CTG over 10,000 times per second. This commit-time operation requires the committing transaction to hold the

kernel_mutex, which protects the lock manager, transaction system, and CTG. In this regard, PSSI is likely to benefit from a more granular set of mutexes. Note that this flatlining behavior does not occur with S2PL or SI, as S2PL and SI do not perform the extra commit-time steps that ESSI and PSSI perform.

5.3.2 Abort Rates and Abort Types

This section presents abort rates measured during our SICycles benchmark tests. Three sets of measurements are provided for each workload configuration: the overall abort rate, the first updater wins (FUW) abort rate, and the serialization abort rate. The *overall abort rate* shows the percentage of executed transactions T_i that were aborted, for any reason. These measurements provide a general comparison of abort rates for the different isolation levels. The *FUW abort rate* shows the percentage of transactions T_i that were aborted for violating the FUW rule (see Definition 1.2). The *serialization abort rate* shows the percentage of PSSI transactions T_i that were aborted to break dependency cycles, or the percentage of ESSI transactions T_j that were aborted to prevent the formation of dangerous structures.

SI transactions have only FUW aborts, while ESSI and PSSI transactions have both FUW and serialization aborts. This means that SI's abort rate is effectively a lower bounds on the overall abort rates for ESSI and PSSI. S2PL transactions abort only due to deadlock, and as we can see from Figures 19, 22, and 25, the deadlock rate in these tests is relatively low. Deadlocks do not occur in the SI, ESSI, or PSSI tests: SI, ESSI, and PSSI transactions enter lock wait only as necessary to support FUW, and each workload configuration has only a single write. Therefore, deadlock is not possible for SI, ESSI, or

PSSI in these workload configurations (SI, ESSI, PSSI deadlocks would require at least two writes per transaction).

In general, the discussions in this section are primarily focused on FUW and serialization aborts for SI, ESSI, and PSSI.

5.3.2.1 s5u1 Abort Rates and Abort Types

Figure 20 shows FUW abort rates for s5u1. In this figure, note that ESSI tends to have lower FUW abort rates than PSSI; this occurs because ESSI detects precursors to cycles and aborts some transactions unnecessarily. The unnecessary serialization aborts can have the side effect of reducing the number of FUW aborts. Example 5.3 illustrates this phenomenon.

Example 5.3: Consider the partial history $H_{5.1}$:

$$H_{5.1}: \quad r_1(x), w_1(y), r_2(y), c_1, w_2(z), r_3(z), c_2, w_3(v), w_4(v)$$

In $H_{5.1}$, T_1 and T_2 are committed, but T_3 and T_4 are still active (with T_4 in lock wait). There is a T_2--rw$\rightarrow T_1$ dependency from $r_2(y)$ and $w_1(y)$, and a T_3--rw$\rightarrow T_2$ dependency from $r_3(z)$ and $w_2(z)$. ESSI and PSSI treat $H_{5.1}$ in very different different ways. T_3 creates a dangerous structure, so ESSI aborts T_3, allowing T_4 to commit. By contrast, PSSI allows T_3 to commit (T_3 does not form a dependency cycle), causing an FUW abort for T_4. In this example, ESSI turns an FUW abort into a serialization abort.

A similar thing can happen between PSSI and SI. Let T_i, T_j be two concurrent transactions where T_i causes a dependency cycle, and T_i, T_j write common data. SI would allow T_i to commit, and abort T_j for FUW. PSSI would abort T_i, allowing T_j to commit.

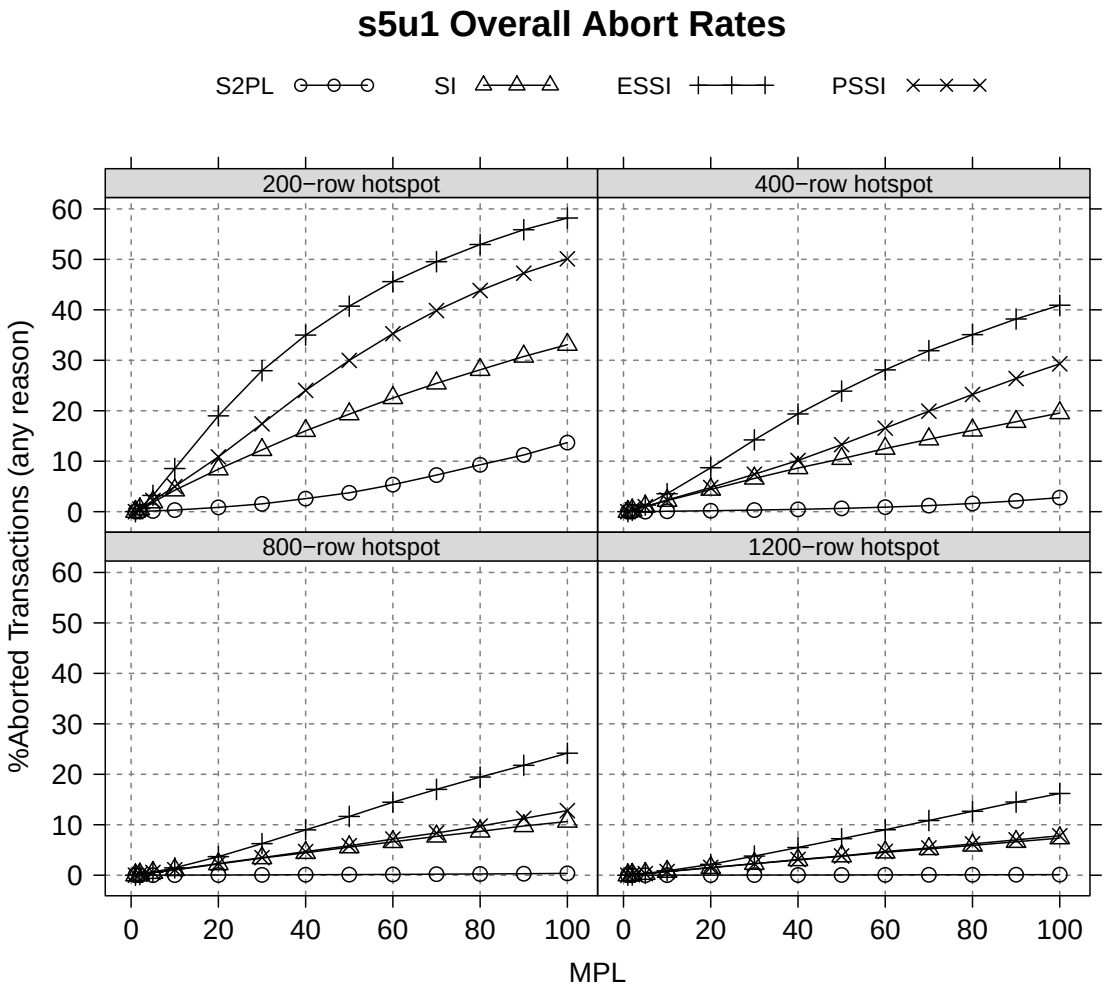

Figure 19: s5u1 Overall Abort Rates for Varying Hotspot Sizes

Figure 21 shows the s5u1 serialization abort rates for ESSI and PSSI. This data clearly demonstrates PSSI to have a lower serialization abort rate; the data also illustrates the false positive aborts that occur with ESSI.

To really understand the differences between PSSI and ESSI, we need to examine CTPS, FUW aborts, and serialization aborts together. Consider s5u1-800 at 80 MPL. In

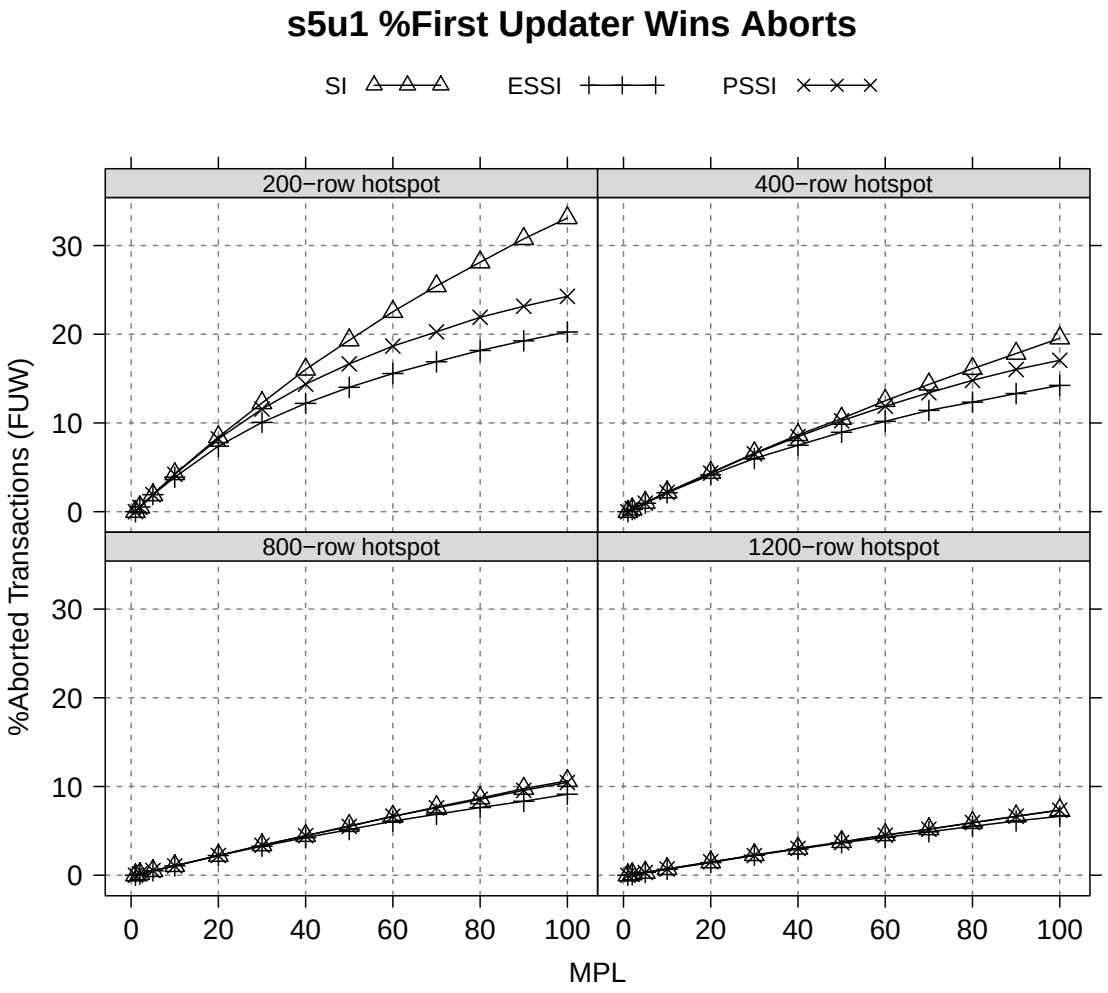

Figure 20: s5u1 FUW Aborts for Varying Hotspot Sizes

this configuration, ESSI has 2998 CTPS, 7.6% FUW aborts, and 11.8% serialization aborts. We can compute the total number of transactions executed per second t as $(1-0.076-0.118)t = 2998$, giving $t = 3719$ transactions executed per second. There were $0.076t = 283$ FUW aborts per second and $0.118t = 439$ serialization aborts per second. Compare this with PSSI: for s5u1-800 at 80 MPL, PSSI had 3370 CTPS, 8.6%

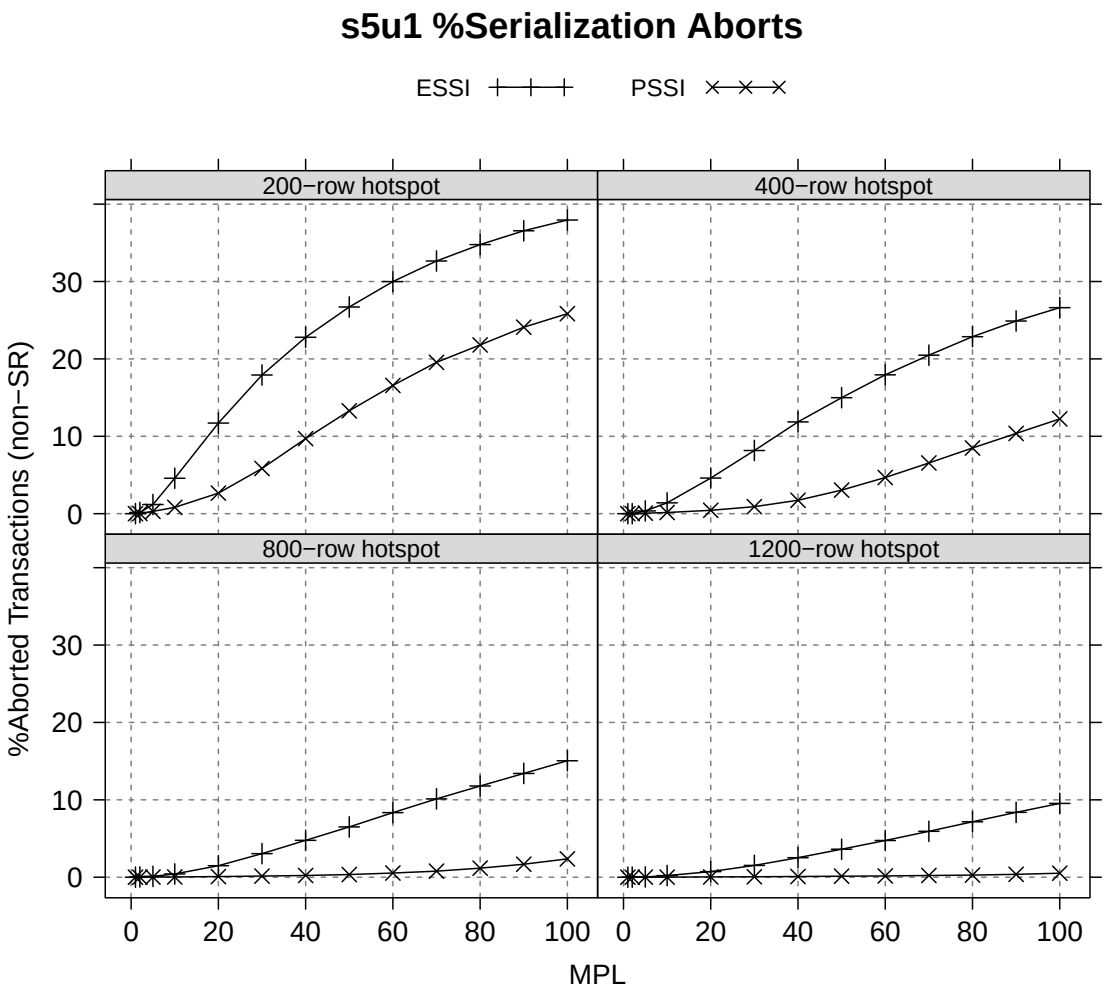

Figure 21: s5u1 Serialization Aborts for Varying Hotspot Sizes

FUW aborts, and 1.2% serialization aborts. We compute the number of transactions executed per second t as $(1-0.086-0.012)t = 3370$. This gives $t = 3736$ transactions executed per second, $0.086t = 321$ FUW aborts per second, and $0.012t = 45$ serialization aborts per second. ESSI and PSSI *execute* about the same number of transactions per second (3719 for ESSI, 3736 for PSSI), but a greater number of PSSI transactions

succeed. ESSI aborts $283+439=722$ transactions per second, while PSSI aborts $321+45=366$ transactions per second; a difference of 356. This is close to the difference in CTPS, $3370-2998=372$.

We repeat this analysis with another configuration, s5u1-400 at 80 MPL. At s5u1-400 80 MPL, ESSI had 2413 CTPS, 12.3% FUW aborts, and 22.9% serialization aborts. To find the number of transactions executed per second, we solve $(1-0.123-0.229)t=2413$ for t. This gives $t=3723$ transactions executed per second, $0.123t=458$ FUW aborts per second, and $0.229t=853$ serialization aborts per second. In the same configuration, PSSI has 2879 CTPS, 14.8% FUW aborts, and 8.5% serialization aborts. Solving $(1-0.148-0.085)t=2879$ for t gives $t=3754$ transactions executed per second, $0.148t=556$ FUW aborts per second, and $0.085t=319$ serialization aborts per second. ESSI aborts a total of $458+853=1311$ transactions per second, and PSSI aborts a total of $556+319=875$ transactions per second. The different in abort rates, $1311-875=436$ transactions per second, is close to to the difference in commits per second, $2879-2413=466$.

In conclusion, most ($>90\%$) of the CTPS difference between PSSI and ESSI can be attributed to ESSI's higher abort rates.

5.3.2.2 s3u1, s1u1 Abort Rates and Abort Types

Figures 23 and 24 show FUW and serialization abort rates for s3u1. In general, the number of serialization aborts decreases (due to reduced contention), and there is less difference among FUW abort rates for SI, ESSI, and PSSI. FUW violations are the only reason that SI aborts transactions; our three s*k*u*n* configurations contain the same number

of writes, so we expect SI's FUW abort rates to be similar for s5u1 and s3u1. Experimental data confirms this expectation: at 80 MPL, SI has 28.1% FUW aborts for s5u1-200 and 28.1% FUW aborts for s3u1-200; 16.1% FUW aborts for s5u1-400 and 15.9% FUW aborts for s3u1-400; 8.7% FUW aborts for s5u1-800 and 8.5% FUW aborts for s3u1-800; 5.9% FUW aborts for s5u1-1200 and 5.8% FUW aborts for s3u1-1200. ESSI and PSSI have higher FUW abort rates for s3u1 (as compared to s5u1) due to a reduced number of serialization aborts, and the phenomenon illustrated in Example 5.3.

For s3u1, we still see significant differences between ESSI and PSSI serialization abort rates. At 80 MPL, ESSI has 22.4% serialization aborts for s3u1-200 while PSSI has 9.1%; ESSI has 12.8% serialization aborts for s3u1-400 while PSSI has 1.7%; ESSI has 5.6% serialization aborts for s3u1-800 while PSSI has 0.2%; and, ESSI has 3.1% serialization aborts for s3u1-1200 while PSSI has 0.06%.

Figures 26 and 27 show FUW and serialization aborts for s1u1. Figure 26 shows an even greater degree of similarity between FUW abort rates, and plots of the three isolation levels nearly converge to single line in s1u1-800 and s1u1-1200. PSSI had very slow serialization abort rates in our s1u1 tests: at 80 MPL, PSSI had 0.2% serialization aborts for s1u1-200, 0.05% serialization aborts for s1u1-400, 0.01% serialization aborts for s1u1-800, and 0.005% serialization aborts for s1u1-1200.

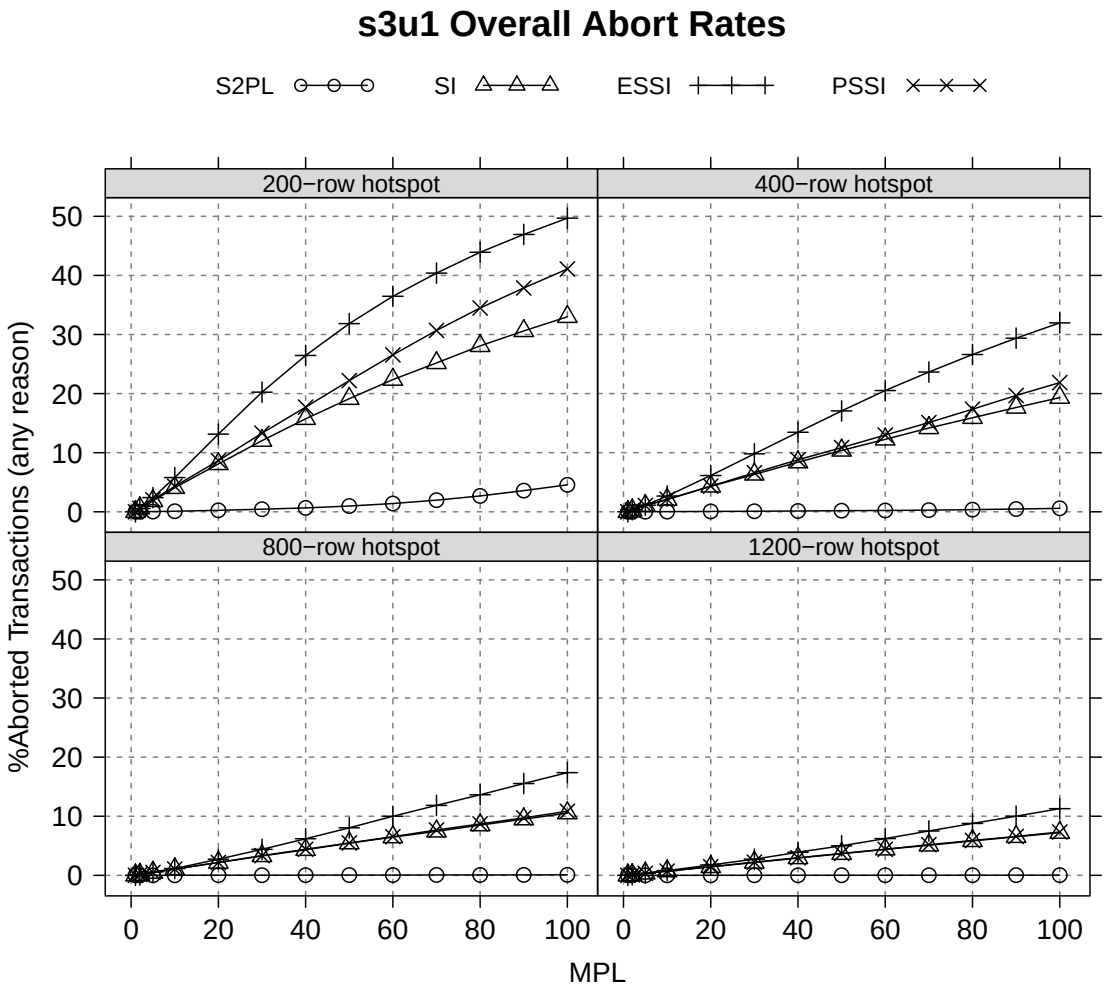

Figure 22: s3u1 Overall Abort Rates for Varying Hotspot Sizes

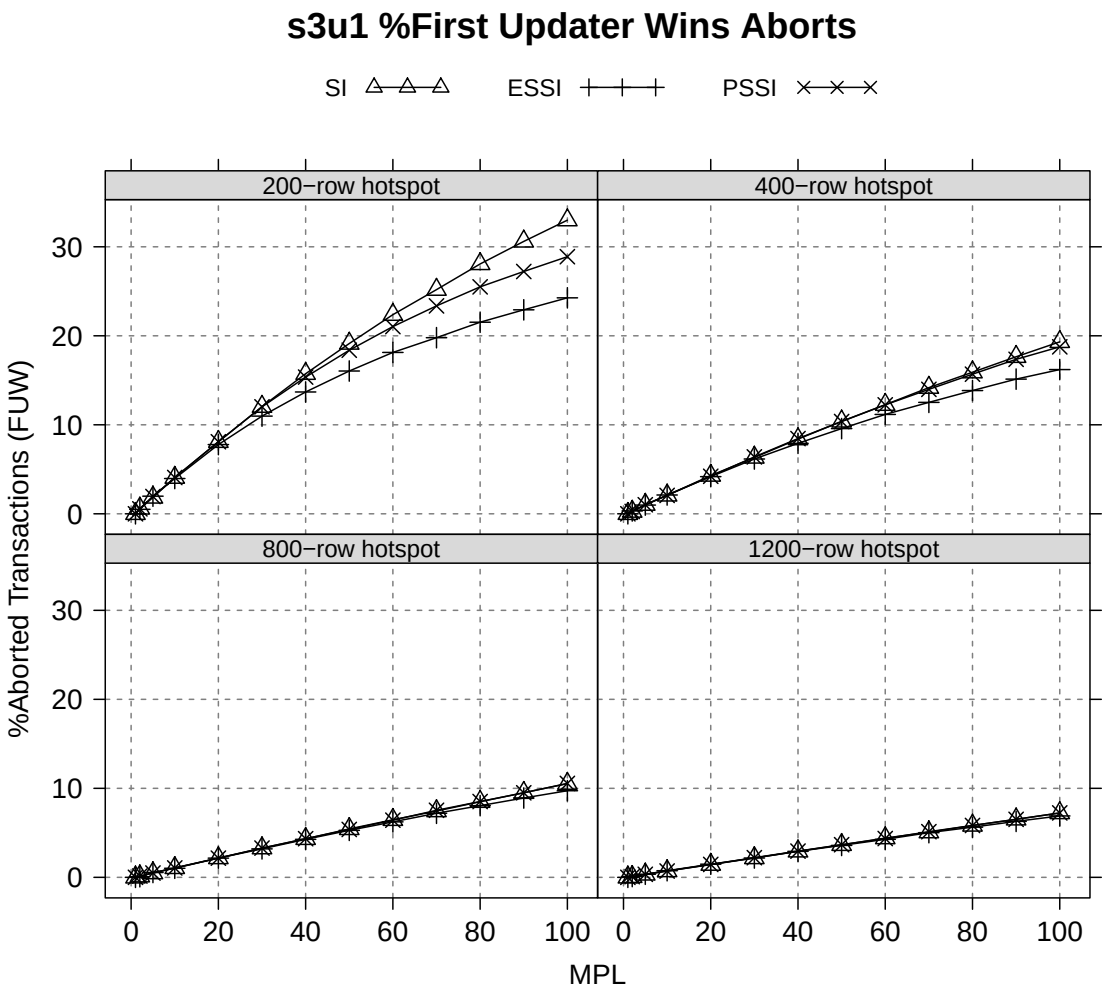

Figure 23: s3u1 FUW Aborts for Varying Hotspot Sizes

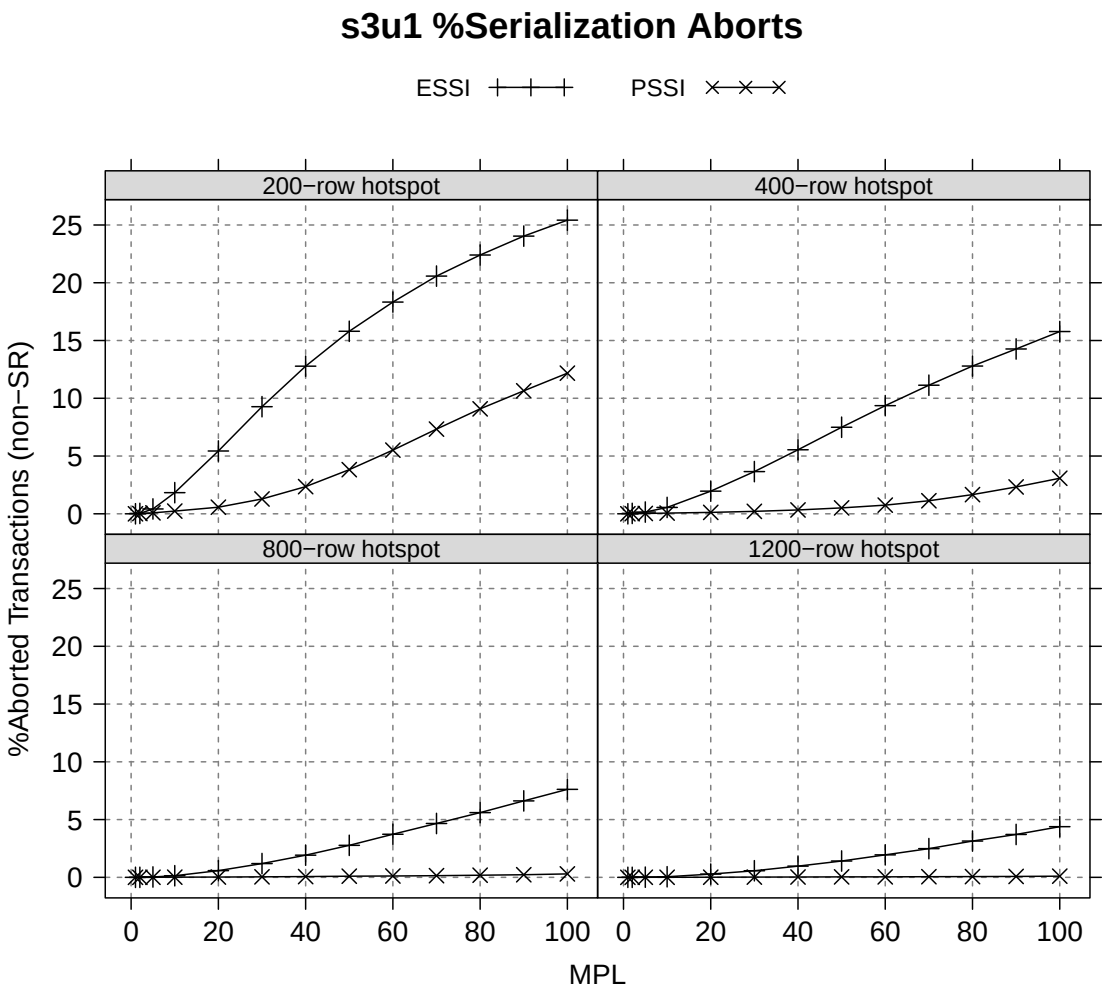

Figure 24: s3u1 Serialization Aborts for Varying Hotspot Sizes

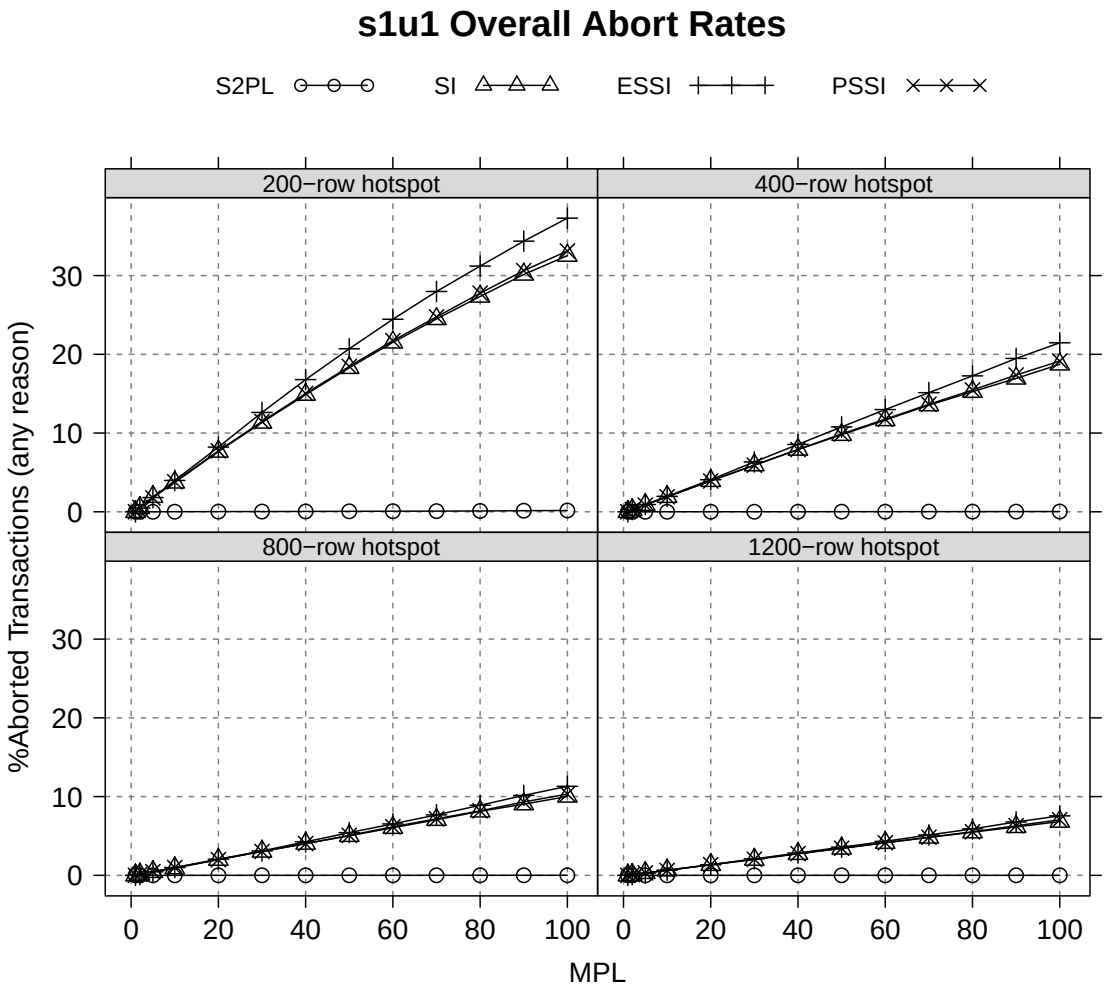

Figure 25: s1u1 Overall Abort Rates for Varying Hotspot Sizes

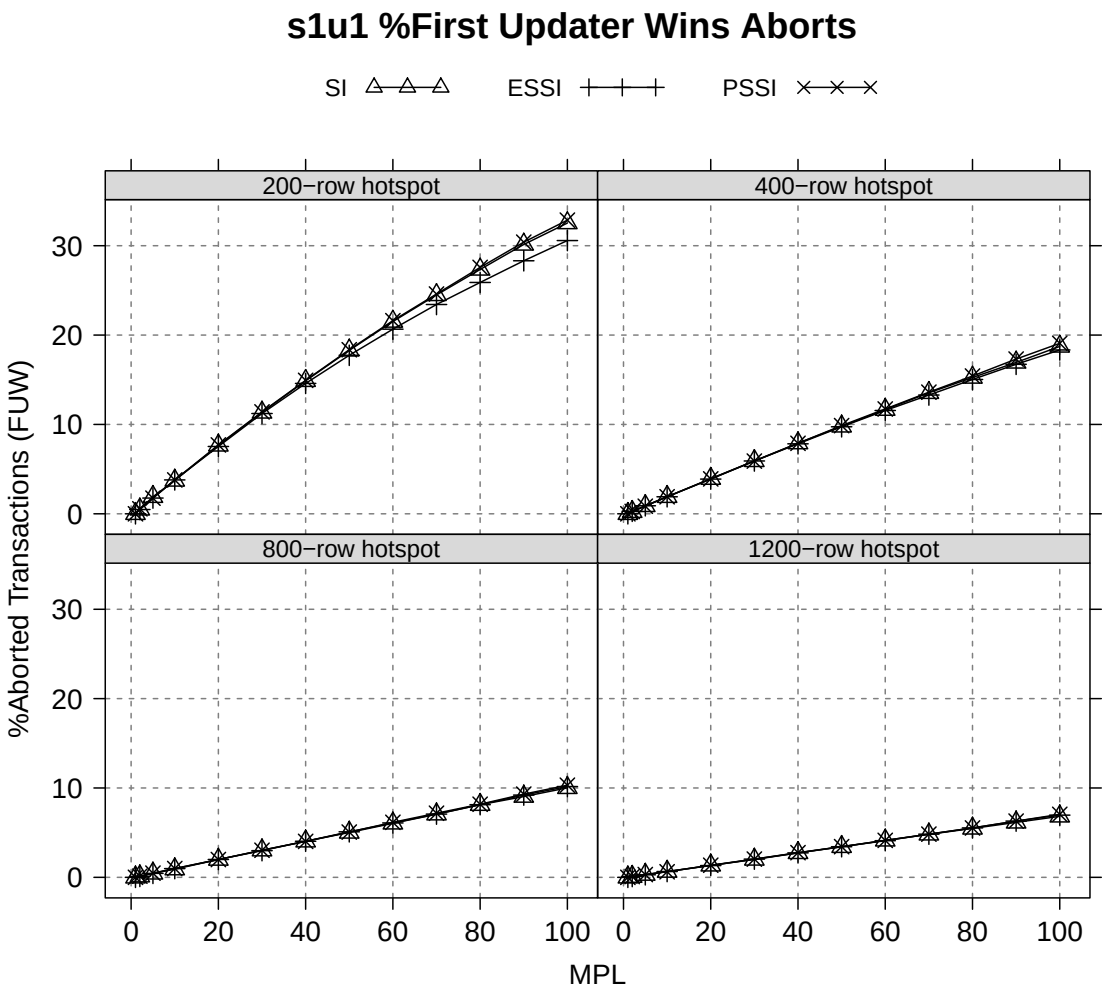

Figure 26: s1u1 FUW Aborts for Varying Hotspot Sizes

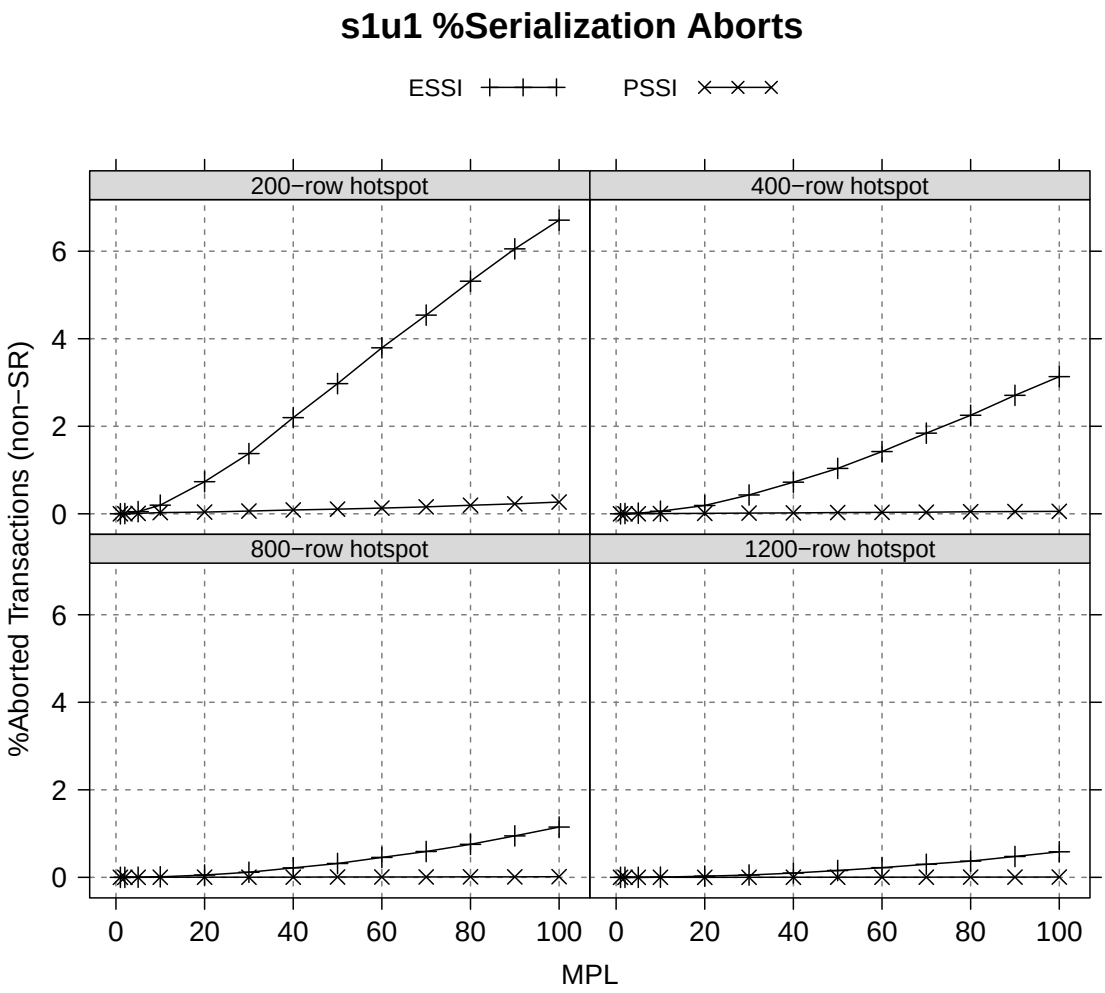

Figure 27: s1u1 Serialization Aborts for Varying Hotspot Sizes

5.3.3 Transaction Durations

This section examines the average duration of committed transactions (these measurements do not incorporate durations of transactions that abort). Transaction duration can be modeled as the sum of two distinct quantities: the time to execute SQL statements, and the time to commit. Our test configuration uses durable group commit, and the test client does not receive acknowledgment of commit(T_i) until after T_i's logs have been flushed to disk. As stated in Section 5.2, we configured InnoDB to write transaction logs to an Intel X25-E solid-state disk, with the disk write cache disabled. In this configuration, commit(T_i) took between 2.3 and 4.4 ms, with an average time of 3.4 ms. This section reports total durations (i.e., time to execute SQL statements plus time to commit), so the reader can assume that $\approx$ 2–4 ms of each duration measurement is dedicated to log flushes.

Also recall that our test program inserts a 3 ms $\pm$50% delay between SQL statements, but no delay before the final commit. For our workloads, this effectively becomes one 3 ms delay per select statement. The combination of log flush times and random delays allows us to establish a minimum lower bound on transaction duration: 17–19 ms for s5u1, 11–13 ms for s3u1, and 5–7 ms for s1u1. Of course, the measured durations are higher, since these lower bounds do not include time required to execute SQL statements, nor time required for the client to process the results.

Our analysis focuses on duration measurements for the 800 and 1200-row hotspot tests, as the S2PL durations for the 200 and 400-row hotspots were large enough to obscure measurements for the other isolation levels. As before, we present s5u1 first, then s3u1 and s1u1.

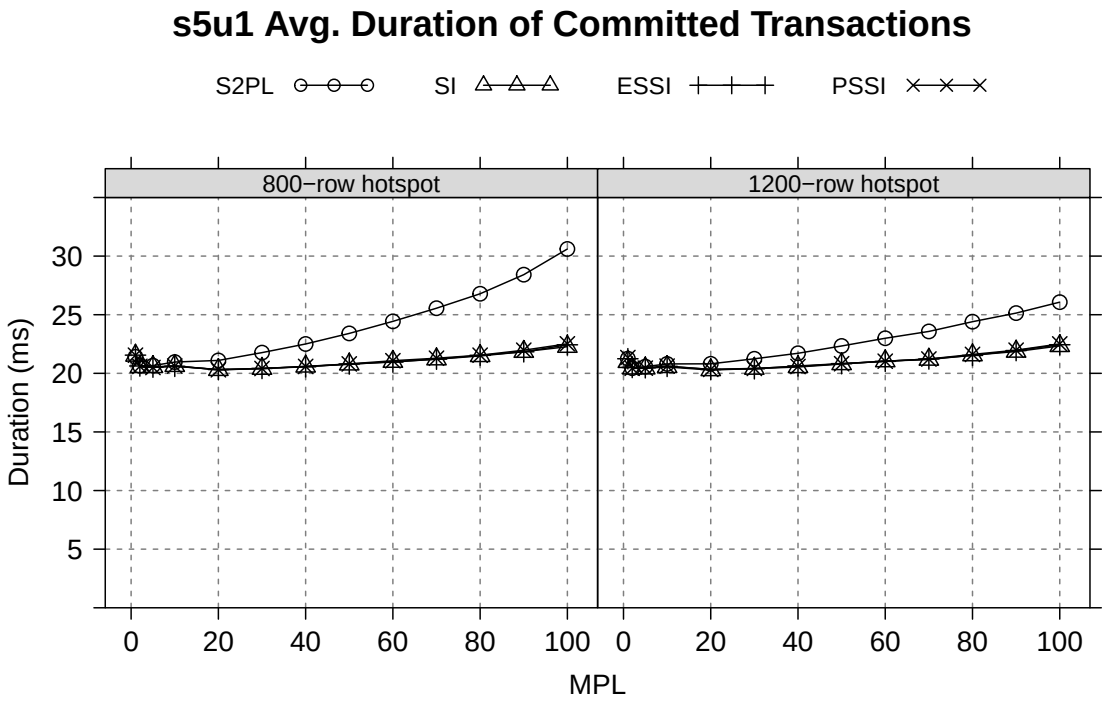

Figure 28: s5u1 Avg. Duration of Committed Transactions

The most obvious feature of Figure 28's s5u1 duration measurements is the difference between S2PL and the other three isolation levels (S2PL transactions take significantly longer). This occurs because all conflicts between concurrent S2PL transactions result in lock waits, whereas SI, ESSI, and PSSI delay transactions only to resolve FUW conflicts. SI, ESSI, and PSSI durations do increase with MPL, but these three isolation levels produce similar duration measurements. For example, at s5u1-800 80 MPL, SI, ESSI, and PSSI transactions all take an average of 21.5 ms. At s5u1-1200 80 MPL, SI and ESSI transactions take 21.5 ms while PSSI transaction take 21.6 ms.

The s3u1 results in Figure 29 paint a similar picture. For s3u1-800 80 MPL, S2PL transactions take 17.1 ms, SI and ESSI transactions take 14.7 ms, and PSSI transactions

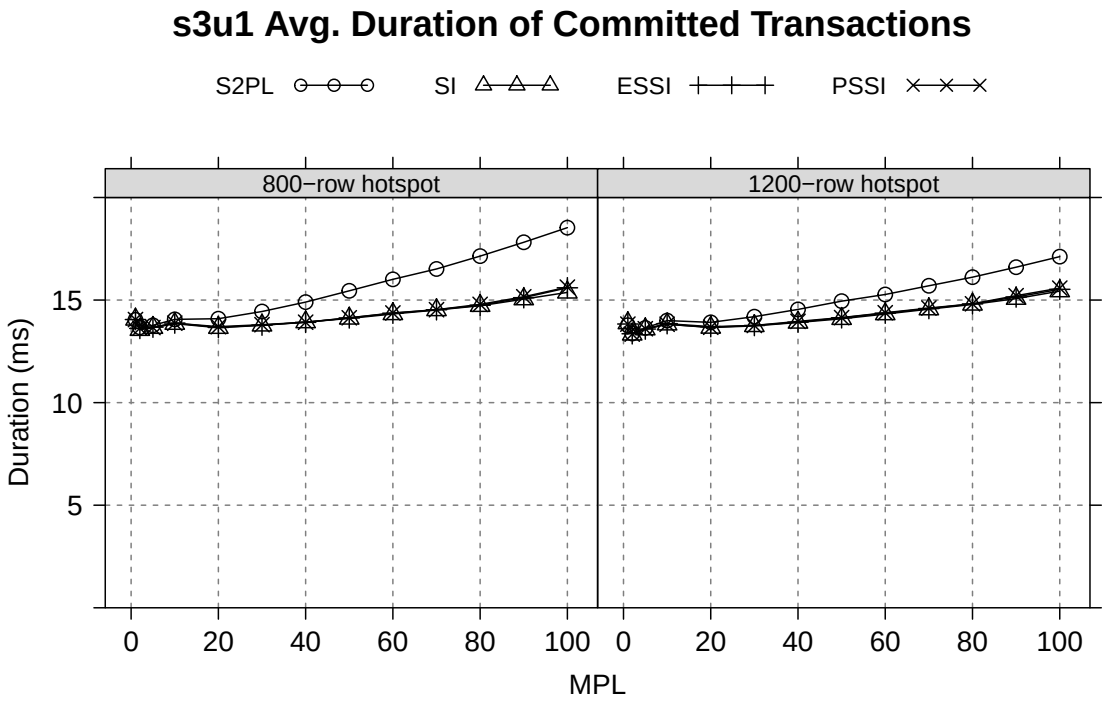

Figure 29: s3u1 Avg. Duration of Committed Transactions

take 14.8 ms. For s3u1-1200 80 MPL, S2PL transactions take 16.1 ms while SI, ESSI, and PSSI transactions take 14.8 ms.

The s1u1 results in Figure 30 show the smallest difference between S2PL and SI, ESSI, and PSSI. Also notice (in Figure 30) that ESSI and PSSI durations increase more rapidly as MPL approaches 100. This mirrors the CTPS "flatlining" shown in Figure 18, which we believed to be caused by kernel_mutex contention.

Taken together, the measurements in this section provide a good illustration of the fundamental difference between S2PL and PSSI. S2PL ensures serializability by delaying transactions (i.e., lock waits), while PSSI ensures serializability by aborting transactions to break dependency cycles. PSSI will perform better in any scenario where increased

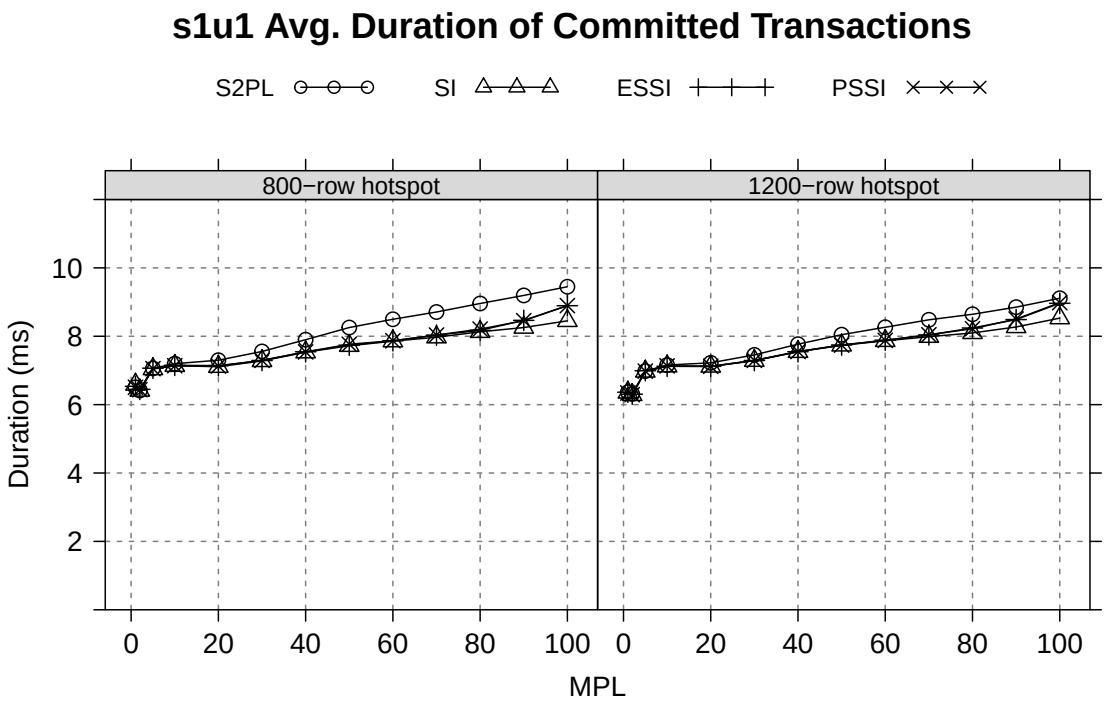

Figure 30: s1u1 Avg. Duration of Committed Transactions

transaction duration (due to lock waits) overshadows the percentage of aborted transactions.

5.3.4 Cycle Testing Graph Measurements

The cycle testing graph (CTG) is a central part of PSSI's design; consequently, we were interested in studying the CTG's behavior during our performance evaluation. This section presents several CTG-related metrics: the average CTG size for ESSI and PSSI, the number of edges traversed by PSSI's cycle tests, and the average length of cycles found.

5.3.4.1 CTG Size

Each CTG node represents a zombie transaction: a committed transaction T_i with the potential to become part of a future cycle (or, in the case of ESSI, a committed transaction T_i with the potential to become part of a future essential dangerous structure). Thus, CTG size gives the number of zombie transactions in the system. Figures 31, 32, and 33 show average CTG sizes for s5u1, s3u1, and s1u1 respectively. Among these figures, we can observe three specific trends.

First, high abort rates result in a smaller number of zombie transactions. For example, the 200-row hotspot measurements show the smallest number of zombies. This occurs for a fairly obvious reason: if many transactions abort, then fewer survive to become zombies, and the size of the CTG is smaller.

Second, PSSI tends to have a larger number of zombie transactions than ESSI. This is attributable to the different pruning criteria that ESSI and PSSI use (see Section 4.4.1). PSSI may prune a zombie transaction T_i if (1) T_i has no in-edges, and (2) T_i committed before the oldest active transaction started. By contrast, ESSI is not concerned with condition (1), and can prune any T_i that meets condition (2). PSSI's more restrictive

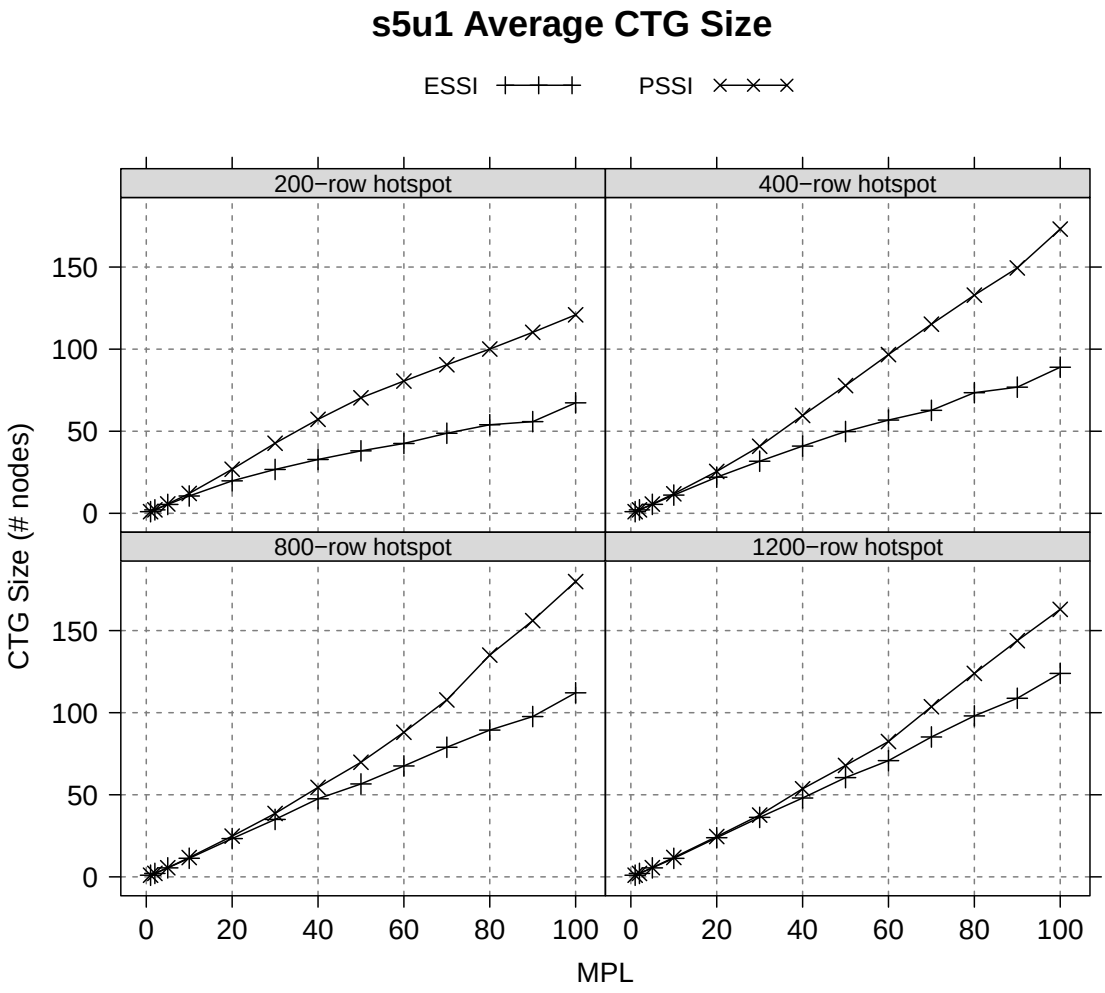

Figure 31: s5u1 Avg. CTG Size

pruning criteria cause transactions to remain zombies for a longer period of time, which increases CTG size.

Finally, as contention decreases, the number of zombie transactions for PSSI and ESSI gets closer together, and we say that the graphs *converge*; this convergence is most apparent in Figure 33's s1u1 measurements. In the absence of in-edges, PSSI's pruning

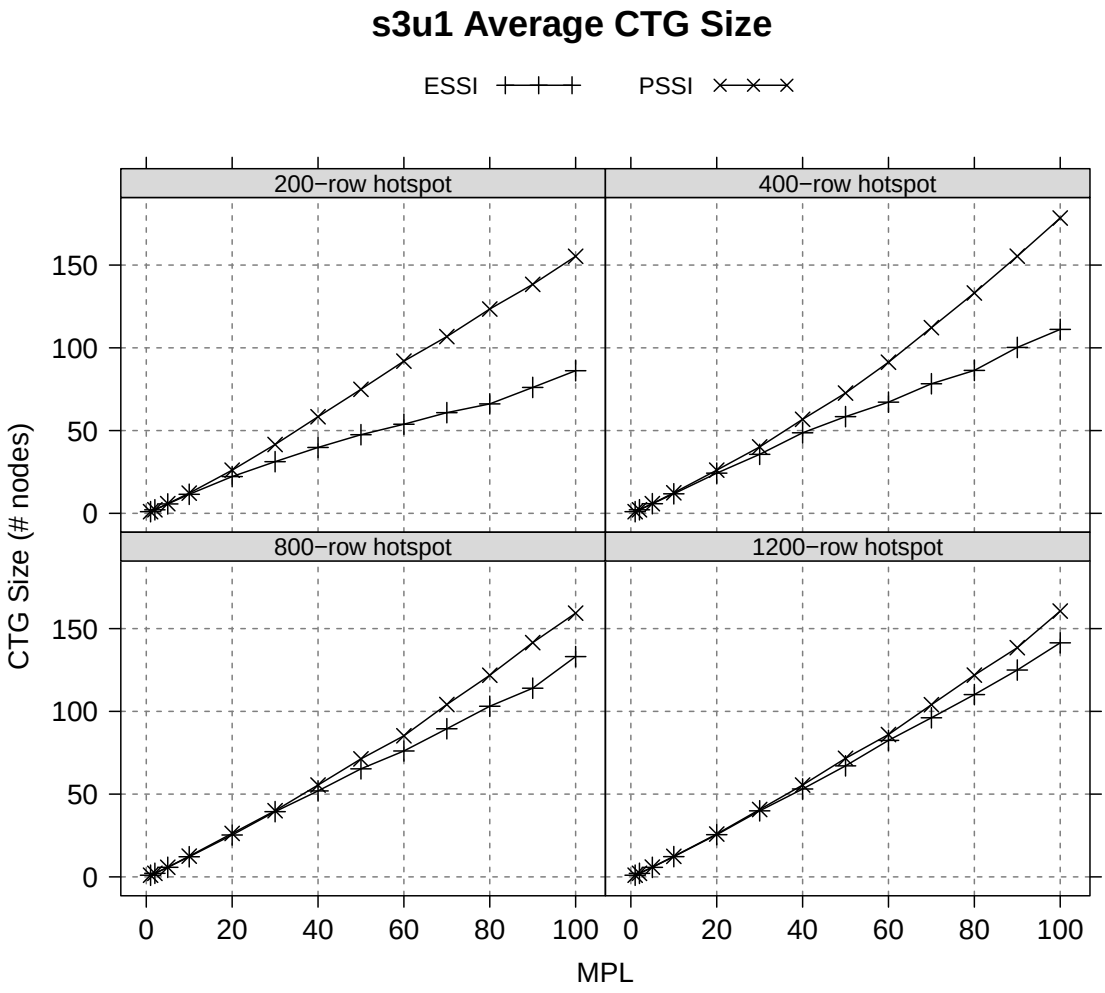

Figure 32: s3u1 Avg. CTG Size

criteria degenerates into ESSI's pruning criteria. Thus, if a workload creates few dependencies, then fewer PSSI transactions will have in-edges, and fewer PSSI transactions will remain zombies solely by virtue of having in-edge counts greater than zero.

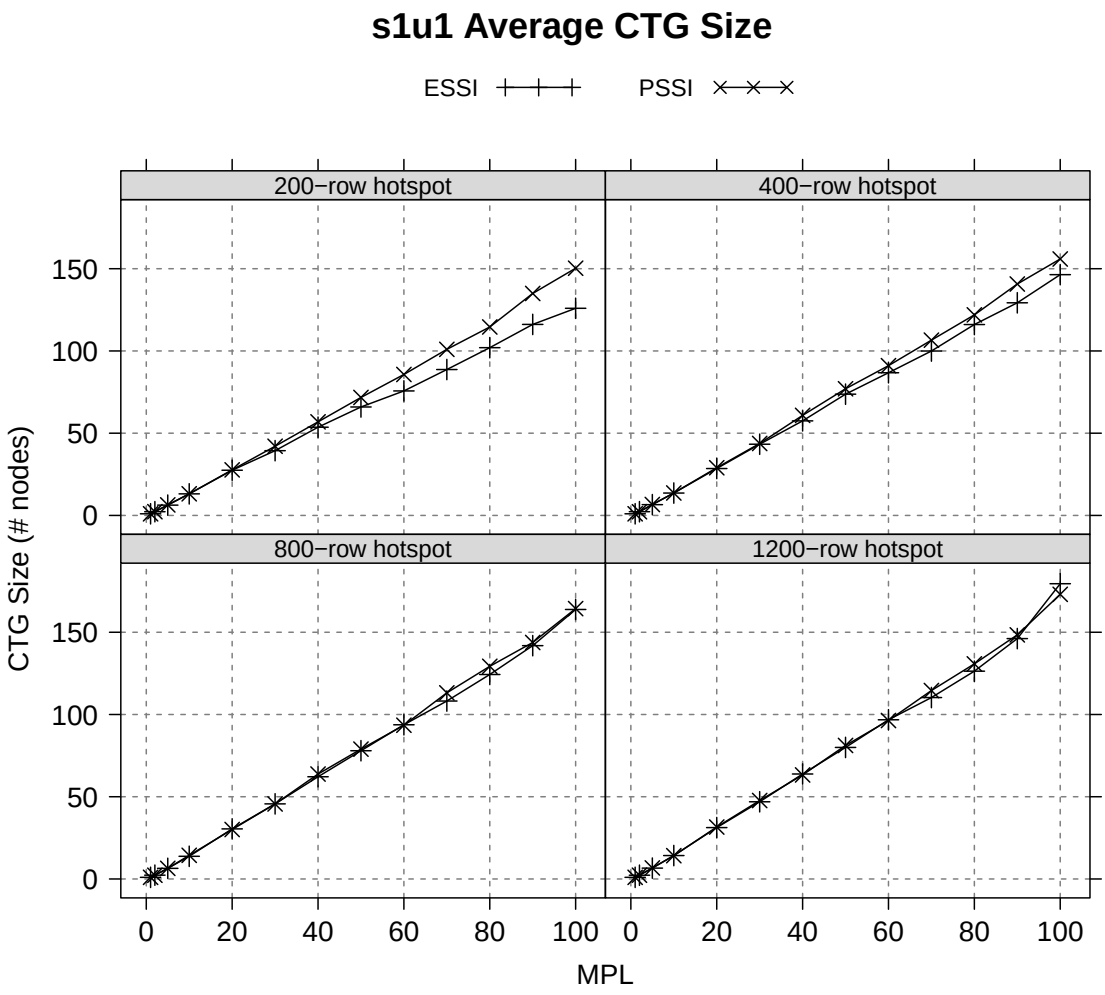

Figure 33: s1u1 Avg. CTG Size

5.3.4.2 Edges Traversed During Cycle Testing

During the design of PSSI, one of the questions that concerned us was "how expensive will cycle testing be?". To answer this question, we instrumented our PSSI prototype to measure the number of edges traversed during cycle tests. We can explain this measure as follows: consider a CTG C with edges $T_1 \rightarrow T_2$, $T_2 \rightarrow T_3$, and $T_2 \rightarrow T_4$. A cycle test for a newly-committed transaction T_1 is a depth-first search starting from T_1. This depth-first search traverses three edges: $T_1 \rightarrow T_2$, $T_2 \rightarrow T_3$, and $T_2 \rightarrow T_4$.

We use the number of CTG edges traversed as a proxy for measuring the cost of cycle testing: a large number of edge traversals indicates a high cost, while a low number of edge traversals indicates a low cost. Our CTG organizes nodes in a hashtable (see Section 4.4), so that an edge traversal involves little more than a hashtable lookup (on the "sink" transaction's transaction id).

Figures 34, 35, and 36 show the number of edges traversed during cycle tests for s5u1, s3u1, and s1u1 respectively. As one would expect, lower contention workloads require fewer edge traversals per cycle test.

In the s5u1 tests at 80 MPL, PSSI performed an average of 13.0 edge traversals per cycle test with a 200-row hotspot, 7.3 edge traversals with a 400-row hotspot, 1.36 edge traversals with an 800-row hotspot, and 0.37 edge traversals with a 1200-row hotspot. In the s3u1 tests at 80 MPL, PSSI performed an average of 8.1 edge traversals per cycle test with a 200-row hotspot, 1.99 edge traversals with a 400-row hotspot, 0.28 edge traversals with an 800-row hotspot, and 0.10 edge traversals with a 1200-row hotspot. Finally, in the s1u1 tests at 80 MPL, PSSI performed an average of 0.42, 0.10, 0.02, and 0.001 edge traversals per cycle test for 200, 400, 800, and 1200-row hotspots, respectively.

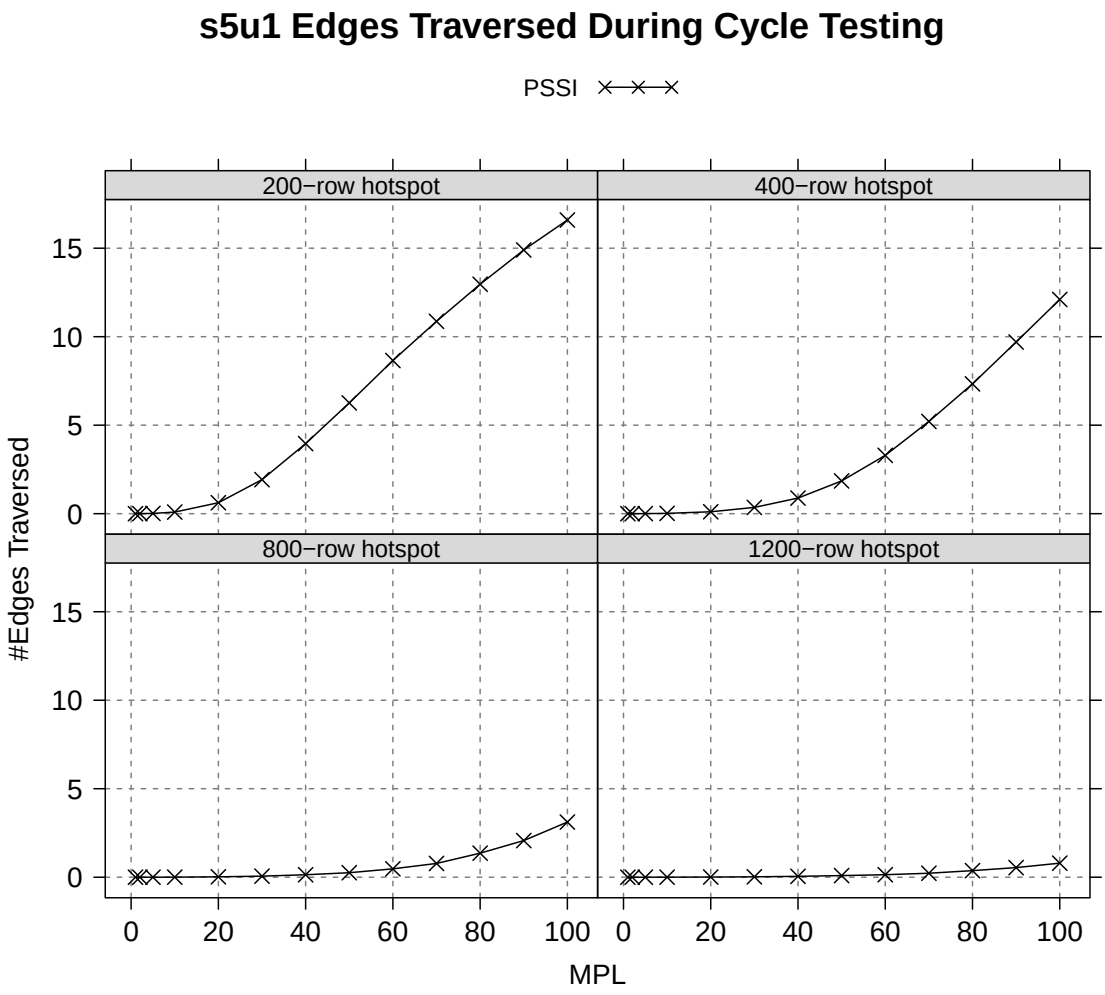

Figure 34: s5u1 Avg. Number of Edges Traversed During Cycle Tests

PSSI performs one commit-time cycle test per transaction, and each edge traversal can be done in near-constant time. Therefore, we believe that the cost of cycle testing is reasonable. This observation also influenced the design of Algorithm 20, CTG-Prune-Initial-Set, which treats in-edge counts as the less selective pruning criterion.

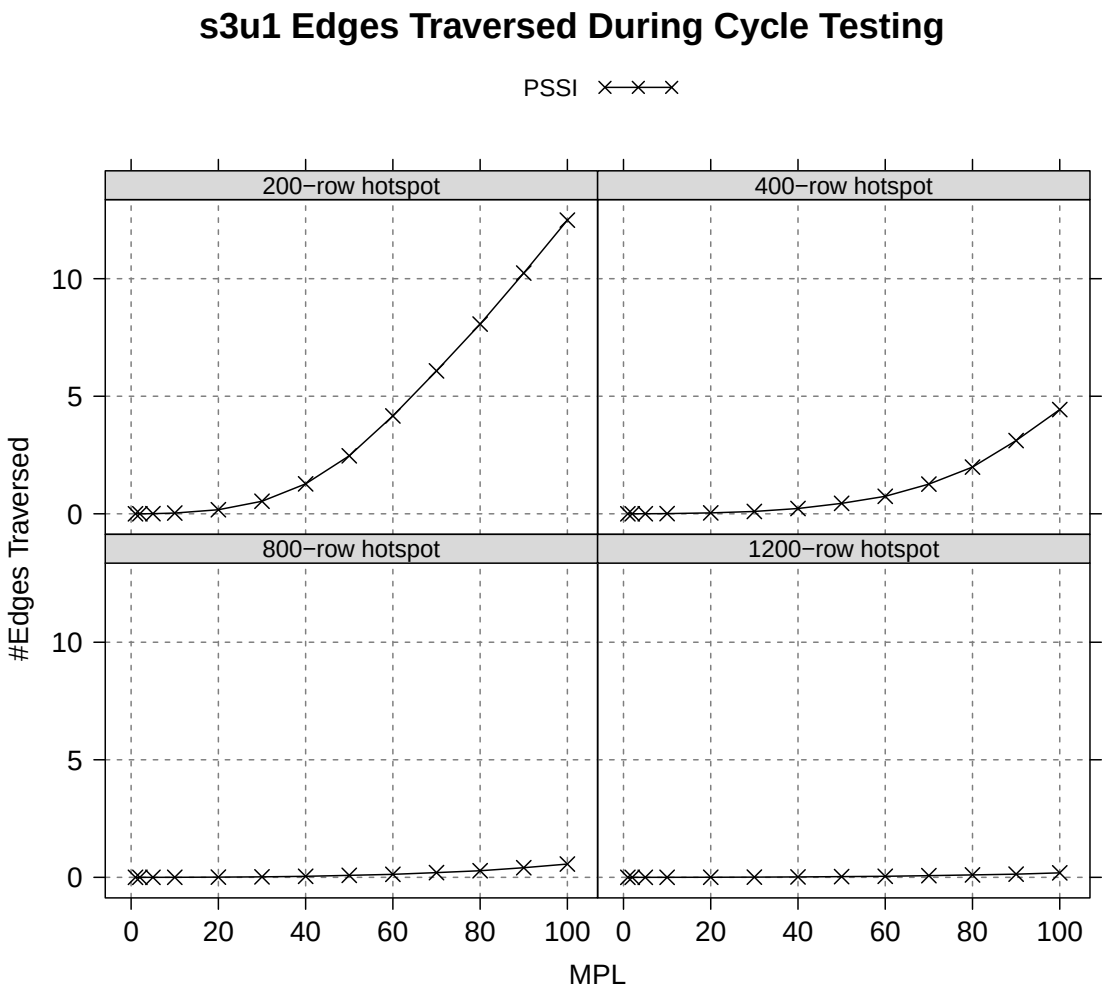

Figure 35: s3u1 Avg. Number of Edges Traversed During Cycle Tests

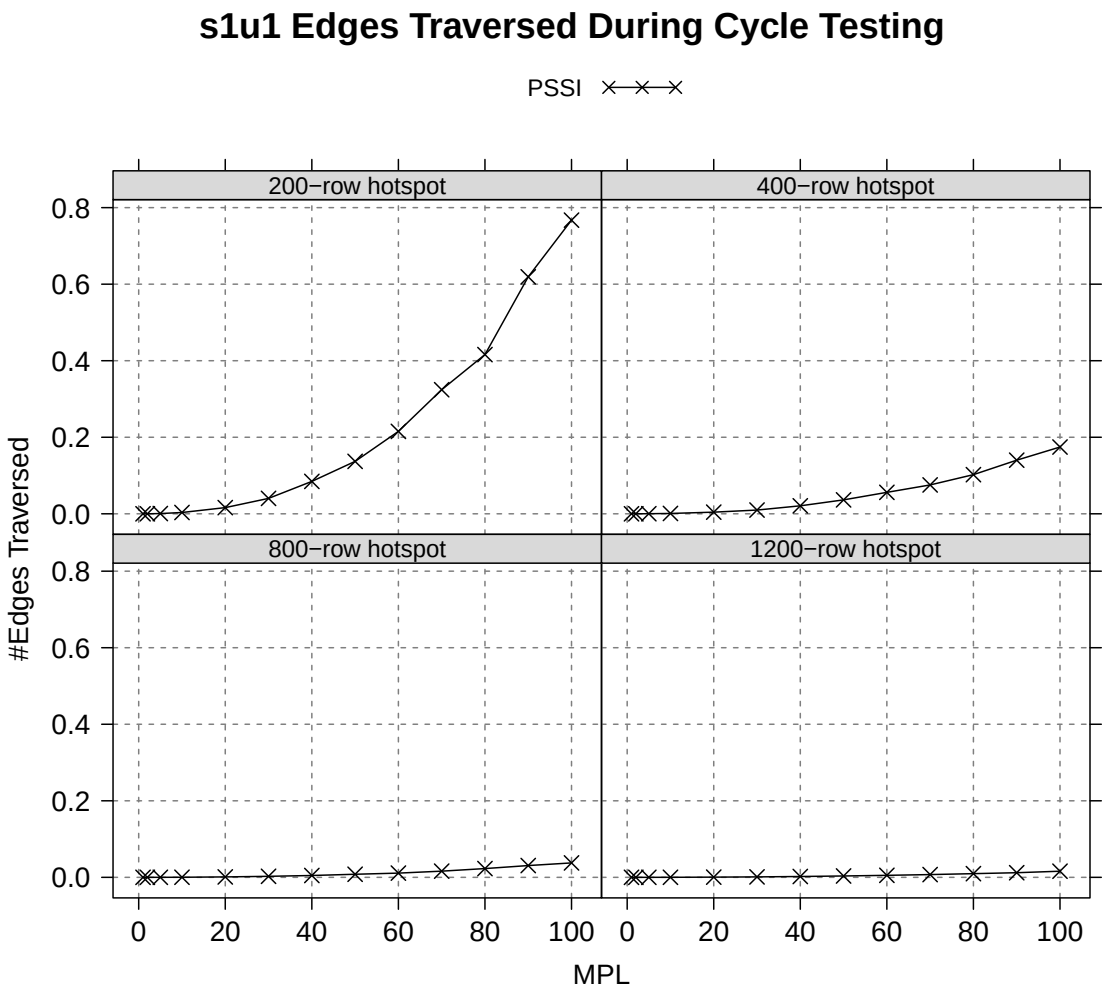

Figure 36: s1u1 Avg. Number of Edges Traversed During Cycle Tests

5.3.4.3 Average Cycle Lengths

The final CTG metric we examine is the average length of cycles found. In general, we expect many cycles to be short, having length 2–4. However, we also expect higher contention workloads to produce longer cycles, for the following reason: workloads with a mixture of reads and writes allow T_i--rw$\rightarrow T_j$ anti-dependencies to occur between pairs of concurrent transactions T_i, T_j, and higher levels of contention tend to create more of these anti-dependencies. Having many T_i--rw$\rightarrow T_j$ anti-dependencies between concurrent pairs of transactions makes it possible for long chains of anti-dependencies to form, leading backwards in time. These chains look like the essential dangerous structure shown in Figure 15 (pg. 92), but with the potential to be much longer. Consider the case of having T_i--rw$\rightarrow T_j$ (T_i, T_j concurrent), with a dependency path leading backwards in time from T_i to T_n, where commit$(T_n) <$ start(T_i). T_i could turn this path into a cycle (a) by overwriting data that T_n wrote (which creates T_n--ww$\rightarrow T_i$), (b) by reading data that T_n wrote (which creates T_n--wr$\rightarrow T_i$), or (c) by writing data that T_n read (which creates T_n--rw$\rightarrow T_i$). In each case, the (long) dependency cycle is $T_i \rightarrow T_j \rightarrow \ldots \rightarrow T_n \rightarrow T_i$.

Figures 37, 38, and 39 present the average cycle lengths observed during our s5u1, s3u1, and s1u1 tests. Smaller hotspots (high contention) produce the longest cycles, due to scenarios like the one described above. For example, s5u1-400 average cycle lengths varied from 2–9.0, s3u1-400 average cycle lengths varied from 2–8.3 and s1u1-400 average cycle lengths varied from 2–3.2. As the hotspot size h increased, shorter average cycle lengths were observed. s5u1-1200 average cycle lengths varied from 2–5.2, s3u1-1200 average cycle lengths varied from 2–3.3, and s1u1-1200 average cycle lengths varied from 2–2.2.

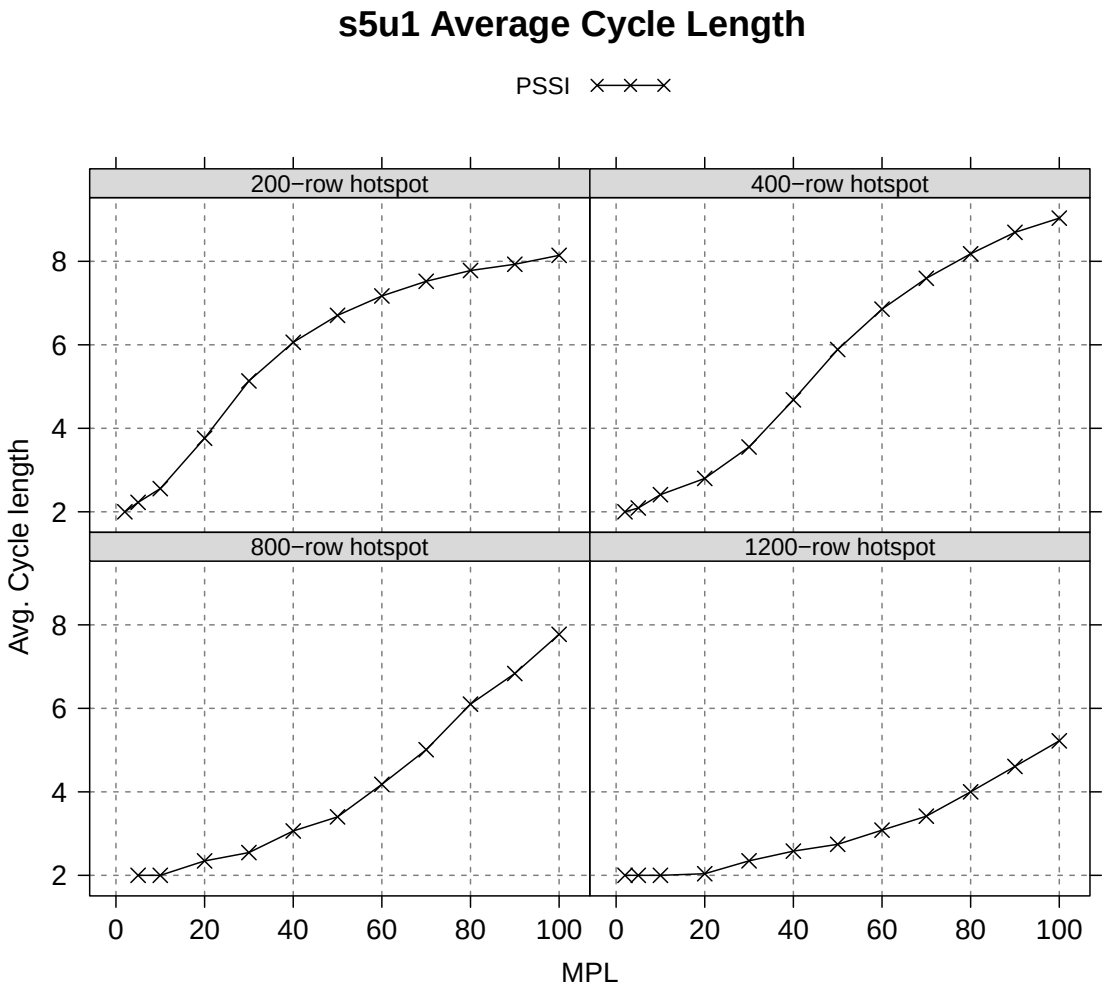

Figure 37: s5u1 Avg. Cycle Length

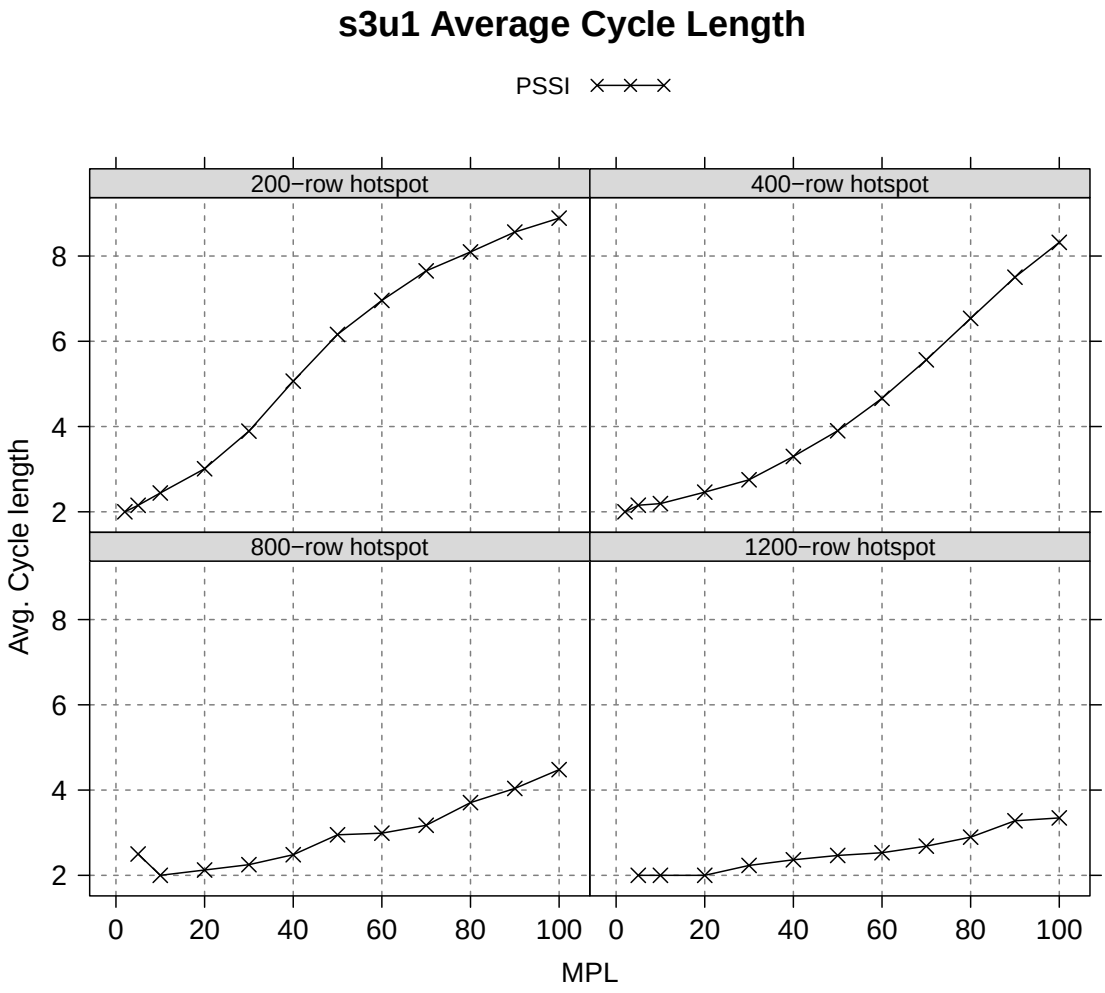

Figure 38: s3u1 Avg. Cycle Length

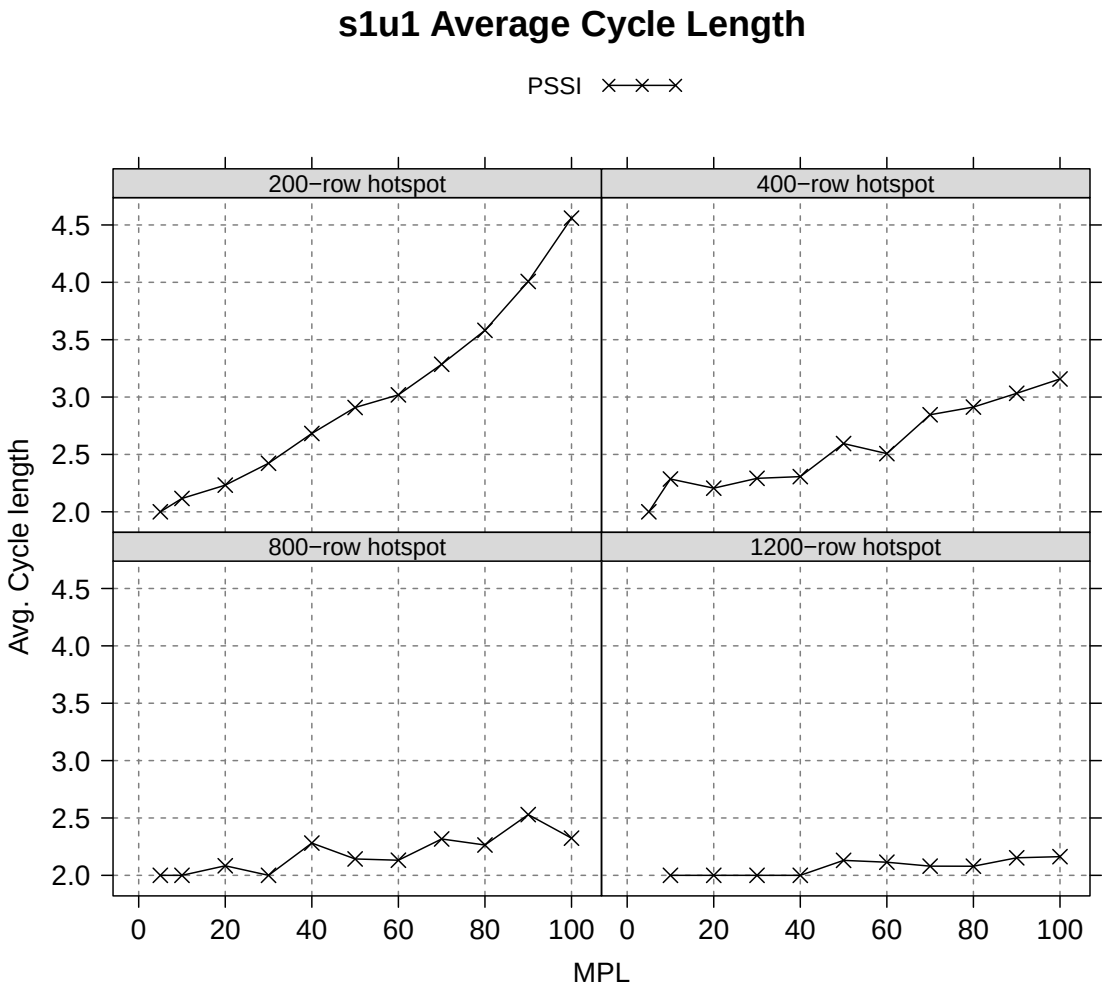

Figure 39: s1u1 Avg. Cycle Length

5.4 Generalizing PSSI's Performance Characteristics

Now that we have examined PSSI's behavior in the context of the SICycles workload, we would like to generalize aspects of PSSI's performance, in order to reason about how PSSI might perform in the context of other workloads.

We have measured PSSI's performance with three different SICycles workload configurations – s5u1, s3u1, and s1u1 – and each of these workload configurations can be characterized by its ratio of reads to writes; s5u1 has a 5:1 read-write ratio, s3u1 has a 3:1 read-write ratio, and s1u1 has a 1:1 read-write ratio. In terms of throughput, higher read-write ratios appear to give PSSI the biggest performance advantage. Why might this be the case? Higher read-write ratios tend to cause more read-write conflicts between pairs of concurrent transactions, and these conflicts do not cause blocking in PSSI. This gives PSSI an advantage when compared to S2PL, as S2PL resolves all conflicts by blocking, and forcing transactions to wait.

We have also seen that higher read-write ratios give PSSI a throughput advantage over ESSI. Again, this appears to be the result of frequent read-write conflicts: these conflicts tend to produce many T_i--rw$\rightarrow T_j$ anti-dependencies between pairs of concurrent transactions, which in turn makes it more likely for essential dangerous structures to form. Comparing serialization abort rates for PSSI and ESSI (e.g., Figure 21 for s5u1), we see evidence that s5u1's moderate read-write ratio produces many essential dangerous structures, but relatively few of these essential dangerous structures go on to become complete cycles.

We can think of read-write ratios as a continuum, varying between a pure write workload (with no reads), and a pure read workload (with no writes). In a pure write

workload, it's very likely that S2PL would provide the best performance. As far as isolation is concerned, the DBMS would have only one goal, namely, to avoid overwriting dirty data; S2PL's pessimism works perfectly well in this regard. For a pure read workload, abandoning isolation is arguably the best solution: if the database data is not changing, then there is no danger of anomaly, and the ideal solution is simply the one which imposes the least amount of overhead. Between these extremes lies a spectrum, and we believe that different isolation levels are best-suited to different regions of the spectrum. For PSSI, the "sweet spot" seems to lie in regions with a moderate-to-high read-write ratio. Towards the opposite end of the spectrum (low read-write ratio), there should be workloads where S2PL outperforms PSSI, and we expect this to happen in any workload where the penalty imposed by FUW aborts is greater than the penalty imposed by lock waits. Example 5.4 illustrates a workload that should create these conditions.

Example 5.4 (*Bank Accounts with Branch Balance*): Consider a banking application with two tables: accounts and branch_balance. The accounts table contains one row per bank account, and transactional programs update this table each time a debit or credit occurs. The branch_balance table contains one row, which represents the sum of all account balances in the entire branch. Thus, each time a transactional program changes an account balance, that transactional program must also apply a corresponding change to the summary row in branch_balance, to ensure that branch_balance accurately reflects the total balance of all accounts.

With this scenario in mind, consider history $H_{5.2}$, where x and y represent bank accounts, and z represents the summary row in branch_balance.

$H_{5.2}$: $w_1(x), w_2(y), w_1(z), w_2(z)$

In history $H_{5.2}$, T_1 is updating the balance of account x, T_2 is updating the balance of account y, and both transactions update the branch balance z. Strictly speaking, each row needs to be read before it is written. However, because each row is written immediately after being read, it would make sense for a transactional program to use select for update, whereby the history degenerates into a series of writes.

An S2PL scheduler would gladly accept $H_{5.2}$; $w_2(z)$ would be blocked until T_1 committed, but both T_1 and T_2 would eventually succeed. Note that T_1 and T_2 are accessing the accounts and branch_balance tables in the same order, so there is no danger of deadlock. By contrast, PSSI (and SI) would abort T_2 due to FUW. This is a very significant difference: S2PL allows concurrent updates to the accounts table, while serializing accesses to the "hot" branch_balance row. By contrast, the FUW rule prevents PSSI (and SI) from allowing concurrent updates to the accounts table; the need to update the "hot" row z has the effect of completely serializing the execution. Thus, $H_{5.2}$ is a history that S2PL allows, but PSSI and SI do not.

Example 5.4 illustrates a scenario where S2PL allows more concurrency than PSSI, and where S2PL should have better performance. Arguably, updates to the branch_balance row in Example 5.4 would be best handled by commutable increment and decrement operations, or by an approach like Escrow transactions [ON86].

5.5 Summary

This chapter presented SICycles, a parametrized benchmark that was designed to allow cycles of varying lengths to form in a variety of ways. We also presented experimental results obtained from running SICycles under four different isolation levels: S2PL, SI,

ESSI, and PSSI. PSSI performed well in these tests, coming close to SI's performance in several configurations. It should be noted that while PSSI's performance comes close to that of SI, we should always expect SI to give (at least slightly) better performance in workloads that allow dependency cycles to form. PSSI ensures serializability by aborting transactions to break dependency cycles, and these aborts reduce PSSI's throughput relative to SI; SI, by contrast, simply permits such dependency cycles.

Our presentation of experimental results gave the most attention to CTPS and abort rates, as these are likely the measures of greatest interest. We have also presented additional measures that help characterize PSSI's behavior: transaction duration, CTG size, cycle length, and the expense incurred in cycle testing.

The next chapter, Chapter 6, describes areas of future work.

CHAPTER 6

FUTURE WORK

This chapter describes two area that would benefit from future research. The first area involves a technique called *early lock release*; the second area deals with a property called *external consistency*.

6.1 Early Lock Release

Early lock release has the potential to improve database performance by allowing T_i's locks to be released (and T_i's commit timestamp to be assigned) before T_i's logs are flushed to stable storage. We did not implement early lock release in our PSSI prototype; nonetheless, we can reason about the effects that early lock release would have.

Section 4.5.1 noted that the standard distribution InnoDB uses a form of early lock release, whereby a committing transaction T_i releases its locks after T_i's commit log sequence number commit-LSN(T_i) is assigned, but before T_i's logs have been flushed to disk. Comments in InnoDB's source code justify this approach as follows: if T_1 and T_2 are update transactions, and T_2 is waiting for one of T_1's locks, then we are guaranteed to have commit-LSN(T_1) $<$ commit-LSN(T_2); therefore, if T_1's logs are not flushed to disk, then T_2's logs will not be flushed to disk either. If a system crash causes T_1 to roll back during recovery, then T_2 will be rolled back too.

This is a reasonable approach for transactions that write data, but it violates the durability guarantee for reader transactions. (Here, the term *reader transaction* refers to a transaction T_i that did not write data, regardless of whether T_i was declared to be read only via a set transaction statement. Similarly, the term *writer transaction* refers to transactions that wrote data.) InnoDB reader transactions do not generate logs, so if a reader transaction T_2 (running under S2PL) is waiting for a writer transaction T_1, and T_1 releases locks before its logs are flushed to disk, then it's possible for T_2 to read T_1's changes and commit, before T_1's changes become durable. If a system crash prevents T_1's logs from being flushed to disk, then T_1 will be rolled back during recovery, and T_2 will have read data that never existed.

For our research, we wanted a system that lived up to the ACID durability guarantee, and we arranged for PSSI to use *late lock release*, whereby T_i's locks are released (and T_i's commit timestamp is assigned) after T_i's logs have been flushed to disk (see Remark 5.1, page 103).

In [JPS10, Sec. 3.2], the authors measured significant performance improvements when early lock release was used with the TPC-B benchmark. Early lock release improved performance by a factor of two when logging to a fast disk (solid state disk, with a 100 μs flush time), and by a factor of 35 when logging to a slow disk (hard disk drive, with a 10,000 μs flush time). These measurements show the value of early lock release as a performance enhancement. For the case of InnoDB, the challenge comes in handling reader transactions properly.

For the discussion that follows, we will distinguish three separate phases of transaction commit: the commit request, the transaction end, and the commit acknowledgment. Definition 6.1 defines these three phases.

Definition 6.1 (*Phases of Transaction Commit*): A *commit request* occurs when a database client program sends a commit statement to the database server.

T_i's *transaction end* occurs when T_i's writes become visible to other transactions. For an S2PL transaction, T_i's write become visible when T_i releases its locks. For PSSI (and SI) transactions, T_i's writes become visible (to transactions T_j with $\text{start}(T_j) > \text{commit}(T_i)$) when the DBMS assigns T_i's commit timestamp. Some papers (like [JPS10]) use the term *pre-committed* to describe transactions that have ended, while InnoDB internals use the state *committed in memory*; we use the term "transaction end" to emphasize its place in the transaction lifecycle.

A *commit acknowledgment* occurs when the database sends an acknowledgment of $\text{commit}(T_i)$ back to the client. For example, in a JDBC program, T_i's commit acknowledgment occurs when connection.commit() returns.

In order to satisfy the ACID durability guarantee, a system that utilizes write-ahead logging (WAL) must ensure that (1) the logs of a writer transaction T_i are flushed to stable storage (i.e., disk) before T_i's commit acknowledgment occurs, and (2) for a reader transaction T_j with T_k--wr$\rightarrow T_j$ dependencies, all T_k logs are flushed to stable storage before T_j's commit acknowledgment occurs. Condition (1) is automatically handled by group commit and the WAL protocol; procedures in Algorithm 22 provide one method for satisfying condition (2).

The main goal of Algorithm 22 is to detect reader transactions that have read non-durable data, and to force these reader transactions to wait for the completion of the next group commit flush. Line 1 introduces a set F; at any moment in time, F represents the set of transactions that have reached transaction end, and are waiting for their logs to

```
1: let F = ∅                    ▷ Set of transaction ids for transactions currently awaiting log flush
2:
3: procedure Post-Write(Transaction T_i)
4:     set T_i.wrote_data = TRUE
5: end procedure
6:
7: procedure Post-Read(Transaction T_i, Data Item x_j)          ▷ x_j was written by T_j
8:     if (T_i.wrote_data == FALSE) and (j ∈ F) then            ▷ j is T_j's transaction id
9:         set T_i.non_durable_reader = TRUE
10:    end if
11: end procedure
12:
13: procedure Modified-Flush(Transaction T_i)
14:    if T_i.wrote_data == TRUE then
15:        add i to F
16:        normal group commit log flush for T_i
17:        remove i from F
18:        return
19:    end if
20:                                                  ▷ T_i is a reader transaction
21:    if T_i.non_durable_reader == TRUE then
22:        T_i waits for next group commit flush to complete
23:        return
24:    else                                          ▷ T_i did not read non-durable data
25:        return                                    ▷ No need to wait for a log flush
26:    end if
27: end procedure
```

Algorithm 22: Reader Transaction Safety for Early Lock Release

be flushed to disk. Algorithm 22 also introduces two per-transaction variables: T_i.wrote_data and T_i.has_read_non_durable; both variables are initialized with the value FALSE. T_i.wrote_data is used to distinguish reader transactions from writer transactions, and T_i.has_read_non_durable is used to identify transactions that have read non-durable data (i.e., writes whose log records have not yet been flushed to disk).

Now, let us examine how the procedures in Algorithm 22 work together. First, procedure Post-Write sets T_i.wrote_data to the value TRUE; the intention is for Post-Write to be called any time that T_i writes data (via insert, update, or delete). By the time that a database client requests T_i's commit, a reader transaction will have T_i.wrote_data set to FALSE, and a writer transaction will have T_i.wrote_data set to TRUE.

Procedure Post-Read is used whenever a transaction T_i reads data item x_j, where x_j is the version of x that appears in T_i's snapshot (x_j may be a dead version). Note that this read implies a T_j--wr$\rightarrow T_i$ dependency. If T_i is a reader transaction, then T_i checks for the presence of the transaction id j in the set F. If $j \in F$ then we know that commit(T_j) $<$ start(T_i), and T_j is waiting for its logs to be flushed to disk. This identifies T_i as a transaction that read non-durable data, and Post-Read sets T_i.has_read_non_durable to TRUE.

Finally, we have the procedure Modified-Flush, which can be thought of as a wrapper around the standard DBMS group commit function. Modified-Flush has two main code paths: one for writer transactions, and one for reader transactions. If T_i is a writer transaction, then T_i adds its transaction id i to the set F, performs a normal group commit log flush, and removes i from F. Thus, for any writer transaction T_i, we have $i \in F$ iff T_i is waiting for its logs to be flushed to disk (i.e., T_i is waiting to become durable).

If T_i is a reader transaction, then there are two sub-paths through Modified-Flush, dependent upon the value of T_i.has_read_non_durable. If T_i.has_read_non_durable has the value TRUE, then we know that (at some point) T_i read from T_j before T_j's logs were flushed to disk, and T_i is forced to wait for the next group commit flush to complete. Note that T_i can get into this state only if T_j was already waiting for a group commit flush; this means that T_i waits for a flush behind T_j, and T_j's logs will be flushed to disk before T_i's commit acknowledgment occurs. The second code path for reader transactions handles T_i that have not read non-durable data (T_i.has_read_non_durable == FALSE). If T_i has not read non-durable data, then Modified-Flush returns immediately, without T_i having to wait for a log flush. (It is not necessary to log commit records of reader transactions; during crash recovery, there is nothing to undo or redo for a reader transaction T_i).

For InnoDB, Algorithm 22 is orthogonal to isolation level, and will work for both PSSI and S2PL. These procedures can be implemented efficiently, with constant-time additional costs (i.e., if the set F is implemented with a hashtable).

Algorithm 22 is a conservative approach, which may cause a reader transaction T_i to wait unnecessarily. For example, T_i might read x_j before T_j's logs have been flushed to disk, but T_j's log flush might occur by the time that T_i's commit request occurs. Algorithm 22 could be made more precise through the use of additional bookkeeping information, but the additional overhead may not be worthwhile, particularly if non-durable reads occur infrequently.

6.1.1 Benefits of Early Lock Release for PSSI

For S2PL, the benefits of early lock release are obvious: early lock release reduces lock wait times, particularly when log flush occupies a significant portion of the total transaction duration. But how would early lock release benefit PSSI? As shown in Section 5.3, S2PL throughput is predominantly influenced by the number of lock waits, while PSSI throughput is predominantly influenced by the number of transaction aborts. Therefore, to argue that early lock release benefits PSSI is to argue that early lock release reduces the rate of PSSI transaction aborts. We can make this argument by showing how early lock release reduces the number of concurrent transactions executing within the DBMS.

Chapter 5 used the term multi-programming level (MPL) to refer to the number concurrently executing transactions. Chapter 5's SICycles workloads did not introduce think time between transaction executions, so the number of concurrent transactions was equal to the number of client threads; 80 client threads generated 80 MPL, and there would always be 80 active transactions. (In this context, an "active" transaction is one that is still holding locks, and has not been assigned a commit timestamp.) Now, let's consider an 80 MPL scenario that uses early lock release, where each group commit flushes the logs of ten transactions to disk (a group commit size of ten transactions was typical for our s3u1 workloads). Early lock release would have the effect of turning these 80 clients into ≈ 65 MPL inside the DBMS. The 65 MPL figure is derived as follows: if each group commit flushes the logs of n transactions to disk, then at any given time, there are n transactions flushing logs to disk, and an average of $(n/2)$ transactions waiting for the next group commit. With early lock release, these $1.5n$ transactions have reached *transaction end* (in the sense of Definition 6.1), and would not be considered active;

therefore, for 80 clients and $n = 10$, we have $80 - 1.5(10) = 65$ transactions active inside the DBMS. Test results in section 5.3.2 show that smaller numbers of concurrent transactions produce lower abort rates. For this scenario, we could expect early lock release to turn our 80 MPL abort rate into (approximately) a 65 MPL abort rate.

Early lock release also has the potential to improve the performance of PSSI's cycle testing graph (CTG) algorithms. Our PSSI prototype assumes that T_i does not end until its logs have been flushed to disk, which means that T_i cannot be pruned (see Section 3.4.1) until T_i's logs have been flushed to disk. Early lock release allows these assumptions to change: if T_i ends before it's logs are flushed to disk, then it should also be possible to prune T_i before its log flush completes. This would reduce the size of the CTG, which in turn, stands to speed up CTG pruning (i.e., Algorithm 20 on page 90).

6.2 External Consistency

The property of *external consistency* can be explained as follows: a DBMS provides external consistency if the serialization order of a history H is consistent with the partial order given by transaction commit and begin statements [JL06]. To put this another way, external consistency guarantees that T_i serializes before T_j whenever $\text{commit}(T_i) < \text{start}(T_j)$.

PSSI guarantees serializability, but not external consistency. This is easily illustrated with the essential dangerous structure diagram in Figure 40. Figure 40 shows a serializable history, whose dependencies T_3--rw$\rightarrow T_2$--rw$\rightarrow T_1$ yield the serialization order T_3, T_2, T_1. This history is not externally consistent because T_3 serializes before T_1, but $\text{commit}(T_1) < \text{start}(T_3)$.

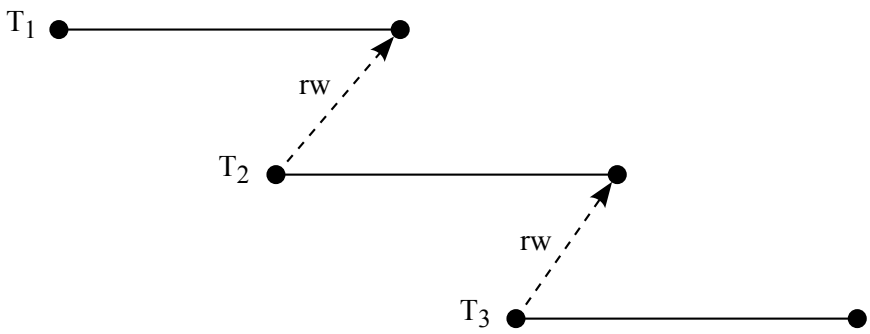

Figure 40: An Essential Dangerous Structure (Figure 6 repeated)

What are the implications of PSSI's lack of external consistency? This is a difficult question to answer. Returning to Figure 40, T_1 and T_3 might have been executed by the same user program. Since commit(T_1) $<$ start(T_3), the user program sees the appearance of T_1 serializing before T_3, while PSSI serializes these transactions in the opposite order. On the surface, there seems to be little trouble with this. In a history that produced Figure 15, we know that T_3 does not depend upon T_1, since such a dependency would create a $T_1 \rightarrow T_3$ dependency edge, forming a cycle, and causing T_3 to abort. However, the user program that executed T_1 and T_3 might have taken output from T_1, and provided this as input to T_3 (see the discussion of the Transaction Handshake Theorem in [BN09, pg. 144]). The transfer of data via user input effectively creates a T_1--wr$\rightarrow T_3$ dependency that is not visible to the DBMS. Of course, this issue is easily circumvented by having T_3 read the values in question directly from the database, as opposed to taking them from user input. There may be circumstances where PSSI's lack of external consistency produces results that are unintuitive (or unexpected) from the user's point of view, despite the fact that they are serializable from the database's point of view.

```
1: procedure Add-CS-Edges(Transaction T_i)
2:     for T_j ∈ ctg.commit_ts_order do
3:         if commit(T_j) > start(T_i) then
4:             break
5:         end if
6:         add T_j--cs→T_i to the CTG
7:     end for
8: end procedure
```

Algorithm 23: Procedure for Adding Commit-Start Dependencies to the CTG

With a few small modifications, PSSI could be extended to provide external consistency. To explain these modifications, we need to introduce a new dependency type, which is given in Definition 6.2.

Definition 6.2 (*Commit-Start Dependency*): We say that there is a *commit-start dependency* T_i--cs$\rightarrow T_j$ when T_i commits before T_j starts.

PSSI could provide external consistency by adding T_i--cs$\rightarrow T_j$ edges to the cycle testing graph (CTG), so that these edges are used during cycle tests. Algorithm 23 provides a way to compute the set of commit-start dependencies created by T_i's commit. Algorithm 23 takes advantage of one of the CTG's auxiliary data structures, ctg.commit_ts_order, which was introduced in Section 4.4. ctg.commit_ts_order is a list of ctg-resident zombie transactions, ordered by commit timestamp. ctg.commit_ts_order was originally added to improve the efficiency of CTG pruning, but it is also suitable for the task of finding T_j--cs$\rightarrow T_i$ dependencies. The procedure Add-CS-Edges iterates through each T_j in ctg.commit_ts_order, adding T_j--cs$\rightarrow T_i$ edges for each T_j with commit$(T_j) <$ start(T_i).

Providing external consistency would likely lead to a reduction in PSSI's CTPS throughput. Chapter 5 showed that PSSI's throughput benefits from low abort rates; adding commit-start dependencies would have the effect of turning some dangerous structures into cycles, thereby increasing aborts and reducing throughput. Consider the dangerous structure shown in Figure 40. Without commit-start dependencies, Figure 40 shows a serializable history, and PSSI would allow all three transaction to commit. Incorporating commit-start dependencies would add T_1--cs$\rightarrow T_3$ (because commit(T_1) $<$ start(T_3)), creating a cycle, and forcing T_3 to abort.

6.3 Summary

This chapter described early lock release, a technique that allows a transaction T_i to "end" before T_i's logs have been flushed to disk. Although the standard distribution InnoDB implements early lock release, it does so in a way that is unsafe for reader transactions. Consequently, we elected to use late lock release for our PSSI prototype, and for our performance testing.

This chapter presented a set of procedures that would make InnoDB's early lock release safe for reader transactions, regardless of the isolation level in use. Although we have not implemented these procedures, we believe that they, in conjunction with early lock release, could improve PSSI's performance by reducing the number of transaction aborts.

Finally, this chapter discussed the concept of external consistency. Our PSSI implementation does not provide external consistency, however, support for external consistency could be added with only a few small modifications. Further study is required

to assess the effects that external consistency would have on PSSI's throughput and abort rates.

CHAPTER 7

CONCLUSION

This dissertation began with a presentation of snapshot isolation (SI), a method of concurrency control that uses timestamps and multiversioning in preference to pessimistic two-phased locking. As originally proposed, SI provides a number of attractive properties (e.g., reads and writes do not block each other), but falls short of providing full serializability. This dissertation offers Precisely Serializable Snapshot Isolation (PSSI), a set of extensions to SI that provide serializability, while preserving the non-blocking qualities that makes SI attractive.

Any SI anomaly involves a dependency cycle. PSSI ensures serializability by keeping track of write-write, write-read, and read-write dependencies, and by aborting any transaction T_i that would cause a dependency cycle to form.

PSSI's heart and soul is comprised of two data structures: the lock table and the cycle testing graph (CTG). The lock table records transaction data accesses, and is responsible for finding all $T_i \rightarrow T_j$ dependencies. Chapter 3's lock table algorithms make the assumption that all dependencies (including predicate dependencies) can be detected via conflicts on individual data items. PSSI incorporates a version of index-specific ARIES/IM, which allows these assumptions to hold. ARIES's design principles treat

predicates as ranges, and achieve range locking via next-key locking [Moh95, pg. 261]. Next-key locking makes range conflicts detectable via conflicts on individual records.

We are only aware of one form of imprecision in PSSI (i.e., cases where PSSI might abort a transaction necessarily), and this imprecision comes as an inherent side-effect of ARIES-style locking algorithms. Given an index with keys "5" and "10" and a transaction T_1 that performs a range scan between "5" and "7", ARIES algorithms will lock $[5, 10]$, the smallest superset of T_i's range scan that can be represented by actual keys in the index. Thus, a T_2 that inserts "8" would be seen as conflicting with T_1, even though there is no logical conflict between the two transactions. Depending on the order of operations, the perceived conflict would produce a $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$ dependency; imprecision results if this dependency becomes part of a cycle, causing some transaction to (unnecessarily) abort. We note that this conservative error is not unique to PSSI, and will occur in any DBMS that uses ARIES-style locking algorithms. Consequently, we believe that PSSI ensures serializability as well as S2PL.

PSSI's cycle testing graph represents a partial suffix of a history H. This graph contains *zombie* transactions (committed transactions with the potential to become part of a future cycle), plus the currently committing transaction T_i. T_i is allowed to commit if the CTG $\cup$ T_i forms a new CTG C', where C' is acyclic.

The CTG cannot be left go grow without bounds, so PSSI incorporates a set of algorithms for *pruning*, a process that identifies zombie transactions that cannot be part of any future cycle, and removes them from the CTG. Pruning is equivalent to performing a topological sort of the CTG, therefore, for any history H accepted by PSSI, the CTG pruning order gives an equivalent serial history $S(H)$. This is a marked difference from

S2PL, where $S(H)$ is given by the commit order of transactions. (In general, the order of PSSI transaction commits does not give an equivalent serial history.)

Chapter 6 notes that S2PL possesses certain properties that PSSI does not. One of these properties is *external consistency*, which guarantees that T_i serializes before T_j any time that T_i commits before T_j. Further study is needed to fully understand the implications of PSSI's lack of external consistency. That said, with a few small changes (i.e., Algorithm 23, page 154), PSSI could be extended to provide external consistency. Further study is also needed to assess how an external consistency requirement would affect PSSI's performance. As explained in Section 4.5.2, external consistency is not a prerequisite for supporting database replication.

Chapter 2 stated that every non-serializable SI history contains a dependency cycle, and every dependency cycle contains at least one dangerous structure. The Dangerous Structure Theorem (Theorem 2.22, page 25) provides an alternate strategy for making SI serializable: instead of aborting any transaction T_i that causes a cycle to form, one could abort any transaction T_i that causes an essential dangerous structure to form. The drawback to the latter approach (called "ESSI" in Chapter 5) comes from the following: a dangerous structure is a necessary condition for a non-serializable SI execution, but in general, the existence of a dangerous structure does not guarantee that a non-serializable execution will occur. (Write skew cycles of length two are an exception to this statement, as these cycles are also dangerous structures.) Thus, preventing dangerous structures (ESSI) may cause some transactions to be aborted unnecessarily. Chapter 5 shows that the number of unnecessary aborts can be significant, and have a correspondingly adverse affect on system throughput. From our experimental tests, we conclude that PSSI

performs well because it minimizes the number of aborted transactions, and because it preserves the “reads and writes do not block” qualities of SI.

In conclusion, we recognize that some areas of PSSI would benefit from future research, but we believe that PSSI has the potential to become an attractive isolation algorithm for database management systems.

REFERENCE LIST

[ACF08] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. "The Cost of Serializability on Platforms That Use Snapshot Isolation." In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pp. 576–585, Washington, DC, USA, 2008. IEEE Computer Society.

[Ady99] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Mar 1999.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1983.

[BBG95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. "A Critique of ANSI SQL Isolation Levels." In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pp. 1–10, New York, NY, USA, 1995. ACM.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[BN09] Phil Bernstein and Eris Newcomer. *Principles of Transaction Processing, 2nd ed.* Morgan Kaufmann Publishers Inc., Burlington, MA, USA, 2009.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.

[CRF08] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. "Serializable Isolation for Snapshot Databases." In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 729–738, New York, NY, USA, 2008. ACM.

[CRF09] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. "Serializable isolation for snapshot databases." *ACM Trans. Database Syst.*, **34**(4):1–42, 2009.

[FLO05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. "Making Snapshot Isolation Serializable." *ACM Trans. Database Syst.*, **30**(2):492–528, 2005.

[FOO04] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. "A Read-Only Transaction Anomaly Under Snapshot Isolation." *SIGMOD Rec.*, **33**(3):12–14, 2004.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Gra10] Goetz Graefe. "A survey of B-tree locking techniques." *ACM Trans. Database Syst.*, **35**(3):1–26, 2010.

[HHS07] Foster Hinshaw, Craig Harris, and Sunil Sarin. "Controlling Visibility in Multi-Version Database Systems." US Patent 7,305,386, Dec 2007.

[Jac95] Ken Jacobs. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle 7*. Oracle Corporation, Redwood Shores, CA, USA, 1995. Part Number A33745.

[JFR07] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. "Automating the Detection of Snapshot Isolation Anomalies." In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pp. 1263–1274. VLDB Endowment, 2007.

[JL06] Daniel Jackson and Butler Lampson. "MIT 6.826 Principles of Computer Systems, Handout 20." http://web.mit.edu/6.826/www/notes/HO20.pdf, 2006. Last viewed September 29, 2011.

[JPS10] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. "Aether: a scalable approach to logging." *Proc. VLDB Endow.*, **3**:681–692, September 2010.

[KR81] H. T. Kung and John T. Robinson. "On optimistic methods for concurrency control." *ACM Trans. Database Syst.*, **6**(2):213–226, 1981.

[Moh95] C. Mohan. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, chapter *Concurrency Control and Recovery Methods for B+-tree Indexes: ARIES/KVL and ARIES/IM*, pp. 248–306. Prentice-Hall, Inc., 1995. Edited by Vijay Kumar.

[ON86] Patrick E. O'Neil. "The Escrow transactional method." *ACM Trans. Database Syst.*, **11**:405–430, December 1986.

[ON93] Patrick O'Neil. "The Set Query Benchmark." http://www.cs.umb.edu/ poneil/SetQBM.pdf, 1993. Last viewed Nov. 10, 2011.

[OO01] Patrick O'Neil and Elizabeth O'Neil. *Database Principles, Programming and Performance*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[ORA05] Oracle Corporation. "Group commit is broken in 5.0." http://bugs.mysql.com/bug.php?id=13669, Sep 2005. Last viewed Sep. 19, 2011.

[ORA09] Oracle Corporation. "MySQL 5.1 Reference Manual." http://dev.mysql.com/doc/refman/5.1/en/, Jan 2009. Last viewed Sep. 12, 2010.

[ORA11] Oracle Corporation. "MySQL 5.6: InnoDB scalability fix - Kernel mutex removed." http://blogs.innodb.com/wp/2011/04/mysql-5-6-innodb-scalability-fix-kernel-mutex-removed/, Jan 2011. Last viewed Aug. 28, 2011.

[PG10] The PostgreSQL Global Development Group. "PostgreSQL 9.0 Documentation." http://www.postgresql.org/docs/9.0/static/, Dec 2010. Last viewed Dec. 4, 2010.

[ROO11] Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil. "Precisely Serializable Snapshot Isolation." In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pp. 482–493, 2011.

[TPC10] Transaction Processing Performance Council. "TPC Benchmark C." http://tpc.org/tpcc/default.asp, Feb 2010. Last viewed Dec. 1, 2010.