Serializable Isolation for Snapshot Databases

This thesis is submitted in fulfillment of the requirements for the
degree of Doctor of Philosophy in the School of Information Technologies at
The University of Sydney

MICHAEL JAMES CAHILL

August 2009

Abstract

Many popular database management systems implement a multiversion concurrency control algorithm called snapshot isolation rather than providing full serializability based on locking. There are well-known anomalies permitted by snapshot isolation that can lead to violations of data consistency by interleaving transactions that would maintain consistency if run serially. Until now, the only way to prevent these anomalies was to modify the applications by introducing explicit locking or artificial update conflicts, following careful analysis of conflicts between all pairs of transactions.

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

Acknowledgements

Firstly, I would like to thank my supervisors, Dr Alan Fekete and Dr Uwe Röhm, without whose guidance I would not have reached this point.

Many friends and colleagues, both at the University of Sydney and at Oracle, have helped me to develop my ideas and improve their expression. Thank you to Alex, Ashok, Danny, James, Keith, Kevin, Margo, Mat, Mike, Mohammad, Raymes, Tim, Tara and Vincent, among many others. Of course, any errors, inaccuracies, blemishes or bugs that remain are entirely my own.

Lastly, I am profoundly grateful to my wife Rachel and to my children Patrick and Evelyn. Your support and patience while I have been busy working has been invaluable.

CONTENTS

List of Figures

CHAPTER 1

Introduction

1.1 Why does serializable execution matter?

Database management systems provide an abstraction called *transactions*, which group together a set of operations that read and write data. The application can choose to *commit* a transaction, in which case the DBMS guarantees that the changes will persist, even after system or application failures. Transactions may *abort*, either as the result of an application request or because of some error detected by the DBMS, in which case any changes it made are undone (rolled back) and discarded.

An application can execute multiple transactions concurrently, and the DBMS schedules the operations in order to maintain data consistency. If the schedule is equivalent to executing the committed transactions one after another in some order, it is called *serializable*.

If a DBMS enforces that all executions are serializable, application developers do not need to worry that inconsistencies in the data might appear as a result of interleaving operations in concurrent transactions. If non-serializable executions are permitted, so-called *anomalies* may occur such as Inconsistent Reads or Lost Updates. The problem of scheduling operations is called *concurrency control* and has been an active database research topic for over four decades.

It is well-known how to use strict two-phase locking (and various enhancements such as escrow locking and multigranularity locking) to control concurrency so that serializable executions are produced (Gray and Reuter, 1993). Some other concurrency control algorithms are known that ensure serializable execution, but these have not been widely adopted in practice, because they usually perform worse than a well-engineered implementation of strict two-phase locking (S2PL).

1.2 What makes snapshot isolation an attractive alternative?

Snapshot isolation (SI) (Berenson *et al.*, 1995) is an alternative approach to concurrency control, taking advantage of multiple versions of each data item. In SI, a transaction T sees the database state as produced by all the transactions that committed before T starts, but no effects are seen from transactions that overlap with T. This means that SI never suffers from Inconsistent Reads. In a DBMS using SI for concurrency control, reads are never delayed because of concurrent transactions' writes, nor do reads cause delays in a writing transaction. In order to prevent Lost Update anomalies, SI does abort a transaction T when a concurrent transaction commits a modification to an item that T wishes to update. This is called the "First-Committer-Wins" rule.

Despite the attractive properties of SI, it has been known since SI was formalized in (Berenson *et al.*, 1995) that SI permits non-serializable executions. In particular, it is possible for an SI-based concurrency control to interleave some transactions, where each transaction preserves an integrity constraint when run alone, but where the final state after the interleaved execution does not satisfy the constraint. This occurs when concurrent transactions modify different items that are related by a constraint, and it is called the "Write Skew" anomaly.

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

```
BEGIN TRANSACTION

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

SELECT COUNT(DISTINCT DoctorId) INTO tmp
  FROM Duties
 WHERE Shift = :S
```

```
    AND Status = 'on duty'

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

This program is consistent, that is, it takes the database from a state where the integrity constraint holds to another state where the integrity constraint holds. However, suppose there are exactly two doctors $D1$ and $D2$ who are on duty in shift S. If we run two concurrent transactions, which run this program for parameters $(D1, S)$ and $(D2, S)$ respectively, we see that using SI as concurrency control will allow both to commit (as each will see the other doctor's status for shift S as still unchanged, at "on duty"). However, the final database state has no doctor on duty in shift S, violating the integrity constraint (Cahill et al.*, 2008).*

Notice that the integrity constraint is violated even though the transaction checks it explicitly before committing, because the constraint is checked on the snapshot of data visible to the transaction rather than the final state resulting from interleaved execution.

SI has become popular with DBMS vendors in spite of the fact that it permits executions that corrupt the state of the database. It often gives much higher throughput than strict two-phase locking, especially in read-heavy workloads, and it provides application developers with transaction semantics that are easy to understand. Many popular and commercially important database engines provide SI, and some in fact provide SI when serializable isolation is requested (including Oracle (Jacobs *et al.*, 1995) and PostgreSQL (PostgreSQL, 2009)). As a result, the documentation for these systems warns users that their products can produce inconsistent results even at what they call serializable isolation:

> *Because Oracle Database does not use read locks, even in SERIALIZABLE transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level should not assume that the data they read will not change during the execution of the transaction (even though such changes are not visible to the transaction). Database inconsistencies can result unless such application-level consistency checks are coded carefully, even when using SERIALIZABLE transactions.* (Ashdown and others, 2005, page 2-17)

1.3 Why is a new approach needed?

Because SI permits data corruption, and is so common, there has been a body of work on how to ensure serializable executions when running with SI as concurrency control. The main techniques proposed so far (Fekete *et al.*, 2005; Fekete, 2005; Jorwekar *et al.*, 2007) depend on performing a design-time static analysis of the application code, and then modifying the application if necessary in order to avoid the SI anomalies. For example, (Fekete *et al.*, 2005) shows how one can modify an application to introduce conflicts, so that all executions will be serializable even when the application uses SI.

Using static analysis to make applications serializable when run at SI has a number of limitations. It relies on an education campaign so that application developers are aware of SI anomalies, and it is unable to cope with ad-hoc transactions. In addition, this must be a continual activity as an application evolves: the analysis requires the global graph of transaction conflicts, so every minor change in the application requires renewed analysis, and perhaps additional changes (even in programs that were not altered).

In this thesis, we instead focus changing the DBMS internals to guarantee that every execution is serializable regardless of the semantics of the transactions, while still having the attractive properties of SI, in particular much better performance than is allowed by strict two-phase locking.

1.4 Contributions

1.4.1 Serializable Snapshot Isolation

We describe a new concurrency control algorithm, called Serializable Snapshot Isolation (Serializable SI), with the following innovative combination of properties:

- The concurrency control algorithm ensures that every execution is serializable, no matter what application programs run.
- The algorithm never causes a read operation to block; nor do reads block concurrent writes.
- Under a range of conditions, the overall throughput is close to that allowed by SI, and better than that of strict two-phase row-level locking.

- The algorithm is easily implemented by small modifications to a data management platform that provides SI: less than 1% of the DBMS code was modified in either of our prototype implementations.

The key idea of our algorithm is to detect, at runtime, distinctive conflict patterns that must occur in every non-serializable execution under SI, and abort one of the transactions involved. This is similar to the way serialization graph testing (SGT) works, however our algorithm does not operate purely as a certification at commit-time, but rather aborts transactions as soon as the problem is discovered; also, our test does not require any cycle-tracing in a graph, but can be performed by considering conflicts between pairs of transactions, and a small amount of information which is kept for each of them. Our algorithm is also similar to optimistic concurrency control (Kung and Robinson, 1981) but differs in that it only aborts a transaction when a *pair* of consecutive conflict edges are found, which is characteristic of SI anomalies. This leads to significantly fewer aborts than optimistic techniques that abort when any single conflict edge is detected. Our detection is conservative, so it does prevent every non-serializable execution, but it may sometimes abort transactions unnecessarily.

1.4.2 Implementation Experience

We describe prototype implementations of the algorithm in two open source database management systems: Oracle Berkeley DB (Olson *et al.*, 1999) and the InnoDB transactional storage engine for MySQL (MySQL AB, 2006). Our primary finding relating to implementation experience was that it is feasible to add serializable isolation with non-blocking reads to systems that had previously offered snapshot isolation.

One important principle that guided the design of the Serializable SI algorithm was to use existing DBMS mechanisms such as the locking subsystem and the multiversion visibility checks. Our experience of implementing the algorithm in two systems demonstrated the soundness of this principle: only small changes were required, in a few isolated parts of the DBMS code.

We describe some implementation choices that had a major impact on the system's usability and performance. In particular, we delay the selection of read snapshot for transactions that start with a locking operation until after the lock is acquired. This design, described in detail in Section 4.5, ensures that single-statement transactions (including auto-commit transactions), never abort due to the first-committer-wins rule.

The biggest structural change to the DBMS required by Serializable SI is that some locks and transactions stay active in a suspended state after commit. The implementation needs to make a tradeoff relating to managing suspended transactions. Aggressively cleaning up suspended transactions will keep the quantity of additional information smaller at the expense of doing more work in every transaction commit.

1.4.3 Anomaly-aware Performance Evaluation

The traditional approach to DBMS performance evaluation focuses on two measurable quantities: *throughput*, which is the number of transactions committed per unit of time, and *response time*, which is the total elapsed time to execute a particular transaction. Benchmarks that discuss isolation generally specify a correctness criterion and vendors can choose the weakest isolation level that gives correct results. Popular benchmarks such as TPC-C (Transaction Processing Performance Council, 2005) execute correctly at weak isolation levels including snapshot isolation (Fekete *et al.*, 2005).

We evaluate the performance of our implementation of Serializable SI compared to each product's implementations of Strict Two-Phase Locking (S2PL), Berkeley DB's implementation of SI and a straightforward implementation of Snapshot Isolation that we added to InnoDB. To do so, we need a benchmark that is not serializable when run at SI so that we can measure the cost of providing serializable executions. We describe three benchmarks ranging from a very simple microbenchmark, `sibench` through to a basic banking workload, `SmallBank`, and concluding with a modification to TPC-C that we call `TPC-C++`, which introduces a new transaction type to the TPC-C mix that makes the modified workload non-serializable when run at SI.

The performance evaluation demonstrates that the cost of ensuring serializable isolation is generally small compared to the performance of SI. In the Berkeley DB evaluation, we demonstrate the shortcomings of page-level locking and versioning, including the high rate of false positives detected by our algorithm in this environment. In the InnoDB evaluation, we show that performance is comparable with SI in the majority of cases we investigate (less than 10%). In some carefully constructed, extreme cases, we show that the additional lock manager activity required by Serializable SI can cause the single-threaded design of the InnoDB lock manager to limit throughput. Our results illustrate that Serializable SI generally has a small execution cost versus SI and can scale to support significant workloads.

1.4.4 Summary

A preliminary version of this work appeared as (Cahill *et al.*, 2008), which described the addition of a new algorithm for serializable isolation to Oracle Berkeley DB. This thesis extends the approach to a relational DBMS using row-level locking and versioning, and deals with the associated problem of phantoms. We describe the implementation of a prototype based on our algorithm in the InnoDB storage manager for MySQL. The detailed evaluation we give here includes a new benchmark based on a small modification to TPC-C (Transaction Processing Performance Council, 2005), which is more complex and representative of typical OLTP workloads than the SmallBank benchmark presented in (Alomari *et al.*, 2008a) and used in (Cahill *et al.*, 2008). Our results indicate that Serializable SI provides serializable isolation with performance close to snapshot isolation and generally much better than strict two-phase locking.

1.5 Outline of the Thesis

This thesis is structured as follows.

Chapter 2 gives an introduction to database concurrency control with a focus on snapshot isolation. It describes current approaches to ensuring serializable isolation with SI.

Chapter 3 presents the new Serializable SI algorithm and outlines a proof of its correctness. Some enhancements and optimizations over the initial, basic description are also given.

Chapter 4 describes some details of two prototype implementations of Serializable SI, one in Berkeley DB and the other in InnoDB.

Chapter 5 deals with the difficulty of performance evaluation for weak isolation, and presents three benchmarks that are sensitive to the difference between SI and S2PL. That is, the benchmarks either highlight a situation in which SI performs better than S2PL, or give incorrect results when run at snapshot isolation, allowing the performance impact of serializable execution to be measured.

Chapter 6 presents the results of an extensive performance evaluation of the two prototype systems using the benchmarks of Chapter 5.

Chapter 7 concludes with a summary of our work and ideas for the direction of future work.

CHAPTER 2

Background

2.1 Transactions

The programming abstraction provided by database management systems for managing concurrency is called a *transaction*. A transaction is a list of operations that are executed by the DBMS as a group. We usually think of the operations as *reads* or *writes* to named data items in the database, where a data item may be a row in a relational table, but it can be any object in the database including a physical page or an entire table. Operations can also be more complex, including *inserts* and *deletes* that create or delete data items, and *predicate reads* that return a collection of data items matching some condition. We call a transaction consisting solely of read operations a *query*, and transactions containing any operation that modifies data an *update*.

The application makes a request to *begin* a transaction, then performs the operations, then requests that the transaction *commit* or *abort* (also known as a *rollback*). The list of operations for a transaction, including the begin and the commit or abort, and called the transaction's *log*. Transactions T_1 and T_2 are *concurrent* if there is an intersection between the interval [begin-time(T_1), commit-time$(T_1))$ and the interval [begin-time(T_2), commit-time$(T_2))$.

Transactions can be understood in terms of the so-called "ACID" properties:

atomicity: all of the results of a transaction should be visible in the database, or none of them should be. It should never be possible to see the results of some operations in a transaction without the others.

consistency: transactions should transform the database from one consistent state to another, in the sense that rules, constraints and referential integrity conditions required by the application should be maintained by each transaction.

isolation: each transaction should execute as if it is running alone in the system. The result of concurrent execution of transactions should be equivalent to some serial execution.

durability: once a transaction successfully commits, the DBMS guarantees that its results will be seen by subsequent transactions regardless of process or system failures.

Among these properties, atomicity, isolation and durability are provided by the DBMS. Consistency is sometimes enforced by the DBMS, through declared constraints such as referential integrity; however, not all consistency properties are declared, and so it becomes a responsibility of application developers to ensure that no transaction breaks any of the assumptions made by the application. However, assuming that each transaction program has been designed to maintain the consistency of the database, the concurrent execution of the transactions by a DBMS should also maintain consistency due to the isolation property.

Implementing transactions in a DBMS to provide the ACID properties involves a combination of the following techniques (Hellerstein *et al.*, 2007):

pessimism (locking): operations have to first place entries into a lock table before the operation can proceed. If there is a conflicting operation in progress, attempting to acquire a lock will *block*, delaying the operation until the conflicting lock is released.

logging: writing records sequentially that describe the operations being carried out by each transaction. Log records are used both to undo a transaction in the case of an abort, and also to recover after a crash and bring databases to a consistent state before processing restarts.

optimism: performing operations without blocking or verifying whether they conflict, then when a commit is requested, checking some conditions and aborting the transaction if certain patterns of conflicts has occurred.

managing multiple versions of data items: locking can be avoided for read operations by potentially reading obsolete data.

Each DBMS implementation combines these techniques in different ways, making tradeoffs that determine how they perform given different workloads.

2.2 Serializable Execution

Intuitively, a concurrent execution of transactions should give results equivalent to running the transactions one after another in some order. Formally, this is captured by a definition of *serializable execution*. This term is used in discussing an *execution history*, which is the complete list of operations executed by the transactions, as they are interleaved by the DBMS. An execution history is *serializable* if there is an equivalence if there is an equivalence between the database states generated by the execution history and the database states that would be created by executing the transactions in some serial order. A system provides *serializable isolation* if all of the execution histories it permits are serializable. There are several different definitions of 'equivalent' used, which lead to different notions of serializable execution.

An important form of equivalence is *conflict equivalence*, leading to a definition of *conflict serializability*. An execution history is conflict equivalent to a serial execution, if both contain the same operations, and all pair of conflicting operations (as defined below) are ordered the same way in both cases. Conflict serializability is a sufficient but not necessary condition for general serializability. The importance of conflict serializability is that it is easy to determine whether a given execution history is conflict serializable, but checking whether it meets the conditions for more general *view serializability* requires much more complex and time-consuming algorithms.

Verifying that a history is conflict serializable is equivalent to showing that a particular graph is free of cycles. The graph that must be cycle-free contains a node for each transaction in the history, and an edge between each pair of *conflicting* transactions. Transactions T_1 and T_2 are said to conflict (or equivalently, to have a *dependency*) whenever they perform operations whose results reveal something about the order of the transactions; in particular when T_1 performs an operation, and later T_2 performs a conflicting operation. Operations O_1 and O_2 are said to conflict if swapping the order of their execution would produce different results (either a query producing a different answer, or updates producing different database state). A cycle in this graph implies that there is a set of transactions that cannot be executed serially in some order that gives the same results as in the original history.

This is formalized in Theorem 1, which models each transaction as a *log* of operations, which is a list of read or write operations on named data items. The execution history is then an interleaving of the different logs; each log is a subsequence of the execution history involving the ops of a single transaction.

Theorem 1 (Conflict serializability, (Stearns *et al.*, 1976))**.** *zi Let* $\mathbf{T} = \{T_1, \ldots, T_m\}$ *be a set of transactions and let* E *be an execution of these transactions modeled by logs* $\{L_1, \ldots, L_m\}$. E *is serializable if there exists a total ordering of* $\mathbf{T}$ *such that for each pair of conflicting operations* O_i *and* O_j *from distinct transactions* T_i*, and* T_j *(respectively),* O_i *precedes* O_j *in any log* $L_1, \ldots L_m$ *if and only if* T_i *precedes* T_j *in the total ordering.*

Serializable isolation is the ideal for a DBMS because it allows transactions to be analyzed individually to ensure that each transaction maintains data consistency. That is, if each transaction maintains all of the (possibly undeclared) constraints required by an application when considered separately, any serializable execution of the transactions will also maintain consistency. Serializable isolation enables the development of a complex system by composing modules, after verifying that each module maintains consistency in isolation.

2.2.1 Strict Two-Phase Locking

Serializable isolation can be implemented with strict two-phase locking (S2PL). "Strict" 2PL is sufficient but not necessary to implement serializable isolation, but is a common choice because it simplifies processing of rollbacks (in particular, it avoids Cascading Aborts).

Implementing S2PL amount to building a *2PL scheduler*, a lock manager that processes lock requests in accordance with the 2PL specification (Bernstein and Goodman, 1981). This involves acquiring a shared lock for data item x before reading x, and acquiring an exclusive lock on x before modifying x. The locks must be held for the full duration of the transaction: they can only be released after the transaction commits or aborts. A request for an exclusive lock on x must wait until any locks on x already held by other transactions are released. Similarly, if another transaction already holds an exclusive lock on x, any requests for a lock on x must wait until the exclusive lock is released. If a set of transactions become blocked waiting for locks, such that each of the transaction is waiting for a lock held by one of the others, a *deadlock* is detected by the DBMS and at least one of the transactions is aborted.

The intuition for how S2PL guarantees serializable executions is that it forces the order of conflicts between operations to correspond with the order of transaction commits. If T_1 performs an operation, and later T_2 performs a conflicting operation, then the lock obtained by T_1 must be released before T_2

can do its operation. As T_1's lock is only released after T_1 commits, and the conflicting operation in T_2 must occur before T_2 commits, we see that T_1 must commit before T_2 commits. Thus there can be no dependency from T_2 to T_1 if T_1 commits before T_2, so the graph of dependencies is cycle-free and committed transactions can be serialized in order of their commit-times.

2.3 Weak Isolation

Serializable isolation implemented with S2PL is widely considered too slow for applications that require high-performance transaction processing because shared locks must be held for the whole duration of the transaction, leading to many lock conflicts and high rates of deadlocks for many applications. For this reason, DBMS platforms have long offered application developers a choice of weaker isolation levels, where serializable executions are not guaranteed, and the interleaving of operations in concurrent transactions can cause incorrect results, called *anomalies*, that would not be possible were the transactions executed serially.

The proposal to provide application developers with a choice of isolation levels weaker than fully serializable comes from (Gray *et al.*, 1975). Each weak isolation level is implemented by early release of some of the locks acquired by S2PL, or by avoiding taking some locks at all. The proposal in that paper gives specific locking algorithms that provide each level; the levels are named “Degree 0”, “Degree 1”, and so on. Degrees 1 and 2 were later called *Read Uncommitted* and *Read Committed*, respectively, in the ANSI SQL standard.

When using weak isolation, verifying the correctness of a system composed of many different types of transactions involves complex analysis of the interactions between all pairs of transactions. Further, there are complete theoretical frameworks for determining the correctness for applications only for the cases where the transactions use snapshot isolation, or mixture of SI and two-phase locking (Fekete, 2005; Fekete *et al.*, 2005). Performing updates at any isolation level other than serializable requires careful analysis to avoid data corruption. In summary, weak isolation levels give application developers a mechanism that trades performance for correctness. While measuring the effect on throughput may not be difficult, it is not easy to measure or predict how much effect the choice of a weak isolation level will have on correctness.

2.4 Multiversion Concurrency Control

A different approach to improving the performance of serializable isolation is to design algorithms that allow data items to be read without acquiring shared locks. Instead, the DBMS maintains multiple *versions* of each data item, and modifying a data item creates a new version rather than overwriting the existing data (Reed, 1978). Each version of a data item has an associated timestamp, create(x_i), assigned when the version x_i is committed, which defines the order of versions of each item. Read operations also have an associated timestamp, read-time, and access the version x_i with the largest create$(x_i) \leq$ read-time.

If queries are separated from updates, and queries are assigned a single read timestamp for all of the data items they read, it is clear that this approach will guarantee that queries see a consistent snapshot of the database, since all versions of data updated by a transaction become visible at a single timestamp, so a query will never see part of an update. The detailed performance study of MVCC algorithms in (Carey and Muhanna, 1986) showed that MVCC is a viable alternative to S2PL in many cases.

In much of the work on MVCC prior to Snapshot Isolation, it was assumed that update transactions would continue to use S2PL (Chan *et al.*, 1983; Bober and Carey, 1992; Mohan *et al.*, 1992), although more complex schemes were proposed for distributed databases (Bernstein and Goodman, 1981).

2.5 Snapshot Isolation

Snapshot Isolation (or *SI*) is a multiversion concurrency control approach that provides lock-free reads, while avoiding many of the anomalies associated with other weak isolation levels. Unlike most other MVCC algorithms, update transactions can also avoid locking for their reads. When a transaction T starts executing at SI, it gets a conceptual timestamp begin-time(T); whenever T reads a data item x, it does not necessarily see the latest value written to x; instead T sees the version of x which was produced by the last to commit among the transactions that committed before T started and also modified x. There is one exception to this: if T has itself modified x, it sees its own version. Thus, T appears to execute against a snapshot of the database, which contains the last committed version of each item at the time when T starts plus T's own changes. When T commits, it gets another timestamp commit-time(T); conceptually, all of T's changes are installed instantaneously as new versions that are visible to transactions that begin at or after commit-time(T).

SI also enforces an additional restriction on execution, called the "First-Committer-Wins" (FCW) rule: two concurrent transactions that both modify the same data item cannot both commit. In other words, when the interval [begin-time(T_1), commit-time(T_1)) has a non-empty intersection with the interval [begin-time(T_2), commit-time(T_2)) and both have written to the same item, at least one of them will abort. The FCW rule prevents the traditional Lost Update anomaly: the history of each data item is a totally ordered list of versions, and any update must be based on the most recent version.

In practice, most implementations of SI use locking during updates to prevent a transaction from modifying an item if a concurrent transaction has already modified it. The first transaction to acquire the lock for an item is permitted to update the item; concurrent transactions attempting to update the same item will block waiting to acquire the lock. If the first transaction goes on to commit, any blocked transactions will detect that a new version has been created and abort after the lock is released. On the other hand, if the first transaction aborts, then no new version will have been installed and one of the blocked transactions will acquire the lock and be permitted to install *its* version. This is sometimes referred to as "First-Updater-Wins", but the effect on permitted transaction histories is the same as the more abstract First-Committer-Wins rule given above.

SI was introduced in the research literature in (Berenson *et al.*, 1995), and it has been implemented by the Oracle RDBMS, PostgreSQL, Microsoft SQL Server 2005, and Oracle Berkeley DB, among others. It provides significant performance improvements over serializability implemented with two-phase locking (orders of magnitude for benchmarks containing a high proportion of reads) and it avoids many of the well-known isolation anomalies such as Lost Update or Inconsistent Read. In some systems that do not implement S2PL, including the Oracle RDBMS and PostgreSQL, snapshot isolation is provided when the application requests serializable isolation, despite the fact that (Berenson *et al.*, 1995) proved that they are not equivalent.

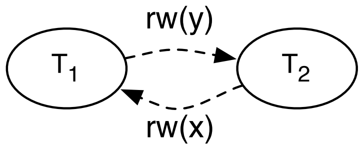


FIGURE 2.1: Serialization graph for transactions exhibiting write skew.

2.5.1 Write Skew Anomalies

As noted in (Berenson *et al.*, 1995), SI does not guarantee that all executions will be serializable, and it can allow corruption of the data through interleaving between concurrent transactions which individually preserve the consistency of the data.

Example 2. *Consider the interleaving of two transactions, T_1 and T_2, withdrawing money from bank accounts x and y, respectively, subject to the constraint that $x + y > 0$. Here is an execution that can occur under SI:*

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

Each of the transactions begins when the accounts each contain \$50, and each transaction in isolation maintains the constraint that $x + y > 0$: when T_1 commits, it calculates $x + y = -20 + 50 = 30$, and when T_2 commits, it calculates $x + y = 50 + -30 = 20$. However, the interleaving results in $x + y = -50$, so the constraint has been violated. This type of anomaly is called a write skew.

We can understand these situations using a multiversion serialization graph (MVSG). There are a number of definitions of general MVSG structures in the literature, because the general case is made complicated by uncertainty over the order of versions, which indeed renders it NP-Hard to check for serializability of a multiversion schedule. For example, there are definitions in (Bernstein and Goodman, 1983; Hadzilacos, 1988; Raz, 1993; Adya, 1999).

With snapshot isolation, the definitions of the serialization graph become much simpler, as versions of an item x are ordered according to the temporal sequence of the transactions that created those versions (note that First-Committer-Wins ensures that among two transactions that produce versions of x, one will commit before the other starts). In the MVSG, we put an edge from one committed transaction T_1 to another committed transaction T_2 in the following situations:

- T_1 produces a version of x, and T_2 produces a later version of x (this is a ww-dependency);
- T_1 produces a version of x, and T_2 reads this (or a later) version of x (this is a wr-dependency);
- T_1 reads a version of x, and T_2 produces a later version of x (this is a rw-dependency, also known as an *anti-dependency*, and is the only case where T_1 and T_2 can run concurrently).

In Figure 2.1 we show the MVSG for the history with write skew, discussed above. In drawing our MVSG, we will follow the notation introduced in (Adya, 1999), and use a dashed edge to indicate a rw-dependency.

As usual in transaction theory, the absence of a cycle in the MVSG proves that the history is serializable. Thus it becomes important to understand what sorts of MVSG can occur in histories of a system using SI for concurrency control. (Adya, 1999) showed that any cycle produced by SI has two rw-dependency edges. This was extended in (Fekete *et al.*, 2005), which showed that any cycle must have two rw-dependency edges that occur consecutively, and further, each of these edges is between a pair of transactions that are concurrent with each other. The formal statement of this is in Theorem 2

Theorem 2 ((Fekete *et al.*, 2005))**.** *Suppose H is a multi-version history produced under Snapshot Isolation that is not serializable. Then there is at least one cycle in the serialization graph MVSG(H), and in every cycle there are three consecutive transactions T_{in}, T_{pivot}, T_{out} (where it is possible that T_{in} and T_{out} are the same transaction) such that T_{in} and T_{pivot} are concurrent, with an edge $T_{in} \rightarrow T_{pivot}$, and T_{pivot} and T_{out} are concurrent with an edge $T_{pivot} \rightarrow T_{out}$. Further, among these three transactions, T_{out} is the first to commit.*

The intuition for this theorem is that under SI, a ww- or wr-dependency from T_i to T_j implies that T_i commits before begin(T_j); so if we consider a cycle in MVSG, and look at the transaction that commits first (call it T_{out}), the edge into T_{out} must be a rw-dependency between concurrent transactions. Furthermore, the transaction T_{pivot} from which this edge goes must commit after begin(T_{out}), or there would be no rw-dependency between them. If we then consider the edge of the cycle that leads to T_{pivot} (call the transaction at the start of this edge T_{in}), we see that T_{in} can't commit before begin(T_{pivot}), as that would be before commit(T_{out}) which was chosen to be the earliest commit in the cycle. Similarly, the edge $T_{in} \rightarrow T_{pivot}$ must be a rw-dependency between concurrent transactions.

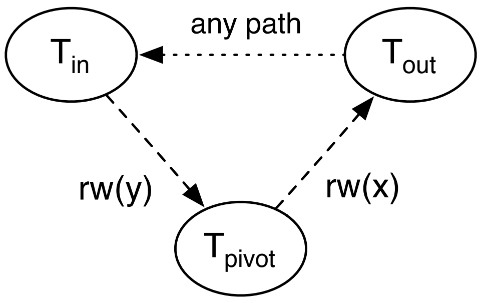


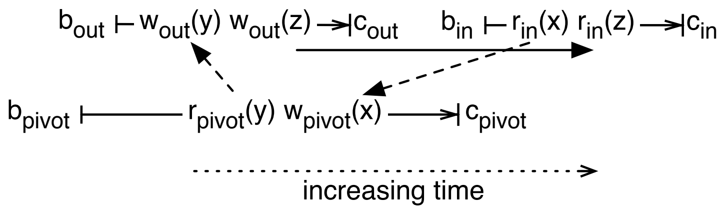
FIGURE 2.2: Generalized dangerous structure in the MVSG.

We adopt some terminology from (Fekete *et al.*, 2005), and call an rw-dependency between concurrent transactions a *vulnerable edge*; we call the situation where two consecutive vulnerable edges occur in a cycle as a *dangerous structure*. It is illustrated in Fig 2.2. We refer to the transaction at the junction of the two consecutive vulnerable edges as a *pivot* transaction, T_{pivot} in the figure. We call the transaction that is the source of the edge into the pivot the *incoming* transaction (T_{in}), and the remaining transaction the *outgoing* transaction, T_{out}. The theory of (Fekete *et al.*, 2005) shows that in any non-serializable execution allowed by SI there is a pivot transaction and the outgoing transaction commits first of all transactions in the cycle.

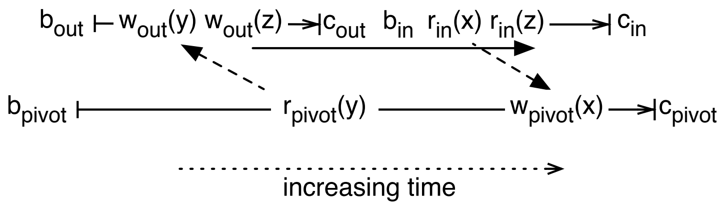
Example 3. *We take an interesting example from (Fekete* et al.*, 2004) to illustrate how a dangerous structure may occur at runtime. Consider the following three transactions:*

T_{pivot}*: r(y) w(x)*
T_{out}*: w(y) w(z)*
T_{in}*: r(x) r(z)*

These three transactions can interleave such that T_{in}*, a read-only transaction, sees a state that could never have existed in the database had* T_{pivot} *and* T_{out} *executed serially. The issue is that if* T_{pivot} *and* T_{out} *run concurrently, then in a serial schedule,* T_{pivot} *would have to be serialized before* T_{out} *because* T_{pivot}*'s read of* y *will not see the update from* T_{out}*. However, the read-only transaction* T_{in} *can be interleaved such that it reads a snapshot containing updates committed by* T_{out} *but not* T_{pivot}*. If* T_{in} *is omitted,* T_{pivot} *and* T_{out} are *serializable because there is only a single anti-dependency from* T_{pivot} *to* T_{out}.


b_out ⊢ w_out(y) w_out(z) →| c_out
b_in ⊢ r_in(x) r_in(z) →| c_in
b_pivot ⊢ r_pivot(y) w_pivot(x) →| c_pivot
increasing time

(a) Reading old values, where the reader commits last.


b_out ⊢ w_out(y) w_out(z) →| c_out
b_in r_in(x) r_in(z) →| c_in
b_pivot ⊢ r_pivot(y) w_pivot(x) →| c_pivot
increasing time

(b) A write after a read, where the pivot commits last.

FIGURE 2.3: Some SI anomalies at runtime.

Two of the possible non-serializable interleavings of these three transactions are illustrated in Figure 2.3. These diagrams should be read from left to right; the dashed arrows indicate the rw-dependencies between transactions. In Figure 2.3(a), both reads occur after the writes. In Figure 2.3(b), T_{in} reads x before it is written by T_{pivot}.

Also note in Figure 2.3(a) that $r_{in}(x)$ occurs after c_{pivot}. That is, the read-write conflict on data item x is not detected until after the pivot has committed. One of the challenges that an algorithm to detect SI anomalies at runtime must overcome is that we cannot always know when a transaction commits whether it will have consecutive vulnerable edges.

We allow for the possibility of *blind writes* in our formalism where there is no read of a data item before it is written, such as the $w(x)$ in T_{pivot} of the example above. This increases the generality of the approach without introducing complexity into either the discussion or the algorithm.

2.5.2 Phantoms

Throughout the discussion so far, we have followed typical concurrency control theory and assumed that a transaction is a sequence of reads and writes on named data items. In general, a relational database engine must also deal with predicate operations (such as SQL "where" clauses). These mean that a concurrency control algorithm must also consider phantoms, where an item created or deleted in one transaction would change the result of a predicate operation in a concurrent transaction if the two transactions executed serially.

The problem was identified in (Eswaran *et al.*, 1976), but the general purpose "predicate locking" solution suggested there has not been widely adopted because of the difficulty in testing mutual satisfiability of predicates. Instead, locking DBMS implementations commonly use algorithms based on "next-key locking". In these, a range of key space is protected against concurrent insertion or deletion by acquiring a shared lock on the next row in order, as a scan is made to check whether rows match a predicate. The scan might be through the data records or through an index. Inserts and deletes follow the same protocol, obtaining an exclusive lock on the row after the one being inserted or deleted. The result of this locking protocol is that a range scan prevents concurrent inserts or delete within the range of the scan, and vice versa.

The state-of-the-art of concurrency control algorithms for DBMS engines is presented with great clarity in (Hellerstein *et al.*, 2007). For locking approaches, the widely-used techniques include an efficient lock manager (Gray and Reuter, 1993), with phantoms prevented by some form of next-key or index lock (Mohan, 1990; Mohan and Levine, 1992; Lomet, 1993).

InnoDB uses a modification of next-key locking involving "gap" locks (MySQL AB, 2006). The motivation for "gap" locks is that a read of the range `[a, c)` should not conflict with `update(c)`. Similarly, `insert(b)` should not prevent a concurrent `update(c)`. A gap lock on `x` is conceptually a lock on the gap just before `x`: it conflicts with other gap locks on `x`, but does not conflict with locks on item `x` itself. Logically, a gap lock is equivalent to using a different key in the lock table for the same data item `x`. Scans, inserts and deletes acquire gap locks on the next key (or the special supremum key for the last key in a table) to prevent phantoms without introducing conflicts with ordinary reads and updates on the next key.

2.6 Making Applications Serializable with Snapshot Isolation

An extensive line of research has considered, not the serializability of particular executions allowed by SI, but rather the question of whether a given set of application programs is guaranteed to generate serializable executions when run on a system with SI as the concurrency control mechanism. This problem was addressed in (Fekete, 1999) and the techniques were refined in (Fekete *et al.*, 2005).

The key to this work is to consider a static analysis of the possible conflicts between application programs. Thus a *Static Dependency Graph* (SDG), is drawn, which has nodes which represent the transaction programs that run in the system. There is a edge in the SDG from program P_1 to P_2, if there can be an execution where P_1 generates a transaction T_1, P_2 generates T_2, and there is a dependency edge from T_1 to T_2, for example, T_1 reads item x and T_2 writes item x. Certain of the edges are distinguished from the others: we say that the edge from P_1 to P_2 is *vulnerable* if P_1 can give rise to transaction T_1, and P_2 can give rise to T_2, and T_1 and T_2 can execute concurrently with a read-write conflict (also called an anti-dependency); that is, where T_1 reads a version of item x which is earlier than the version of x which is produced by T_2. Vulnerable edges are indicated in graphs by a dashed line.

It was shown how a non-serializable cycle in the MVSG can be related to a similar structure in the SDG, and this justified the intuition of experts, who had previously decided that every execution of the TPC-C benchmark (Transaction Processing Performance Council, 2005) is serializable on a platform using SI. As stated in Definition 1, an SDG is said to contain a *dangerous structure* when there are two vulnerable edges in a row as part of a cycle (the other edges of the cycle may be vulnerable, or not).

Definition 1 ((Fekete *et al.*, 2005))**.** *The static dependency graph SDG(A) has a* dangerous structure *if it contains nodes P, Q and R (some of which may not be distinct) such that:*

(a) There is a vulnerable anti-dependency edge from R to P
(b) There is a vulnerable anti-dependency edge from P to Q
(c) Either Q=R or else there is a path in the graph from Q to R; that is, (Q,R) is in the reflexive transitive closure of the edge relationship.

The main theorem of (Fekete *et al.*, 2005), given here as Theorem 3, is that if a SDG is free of dangerous structures, then every execution of the programs is serializable on a DBMS using SI for concurrency control.

Theorem 3 ((Fekete *et al.*, 2005))**.** *If an application A has a static dependency graph SDG(A) with no dangerous structure, then A is serializable under SI; that is every history resulting from execution of the programs of A running on a database using SI is serializable.*

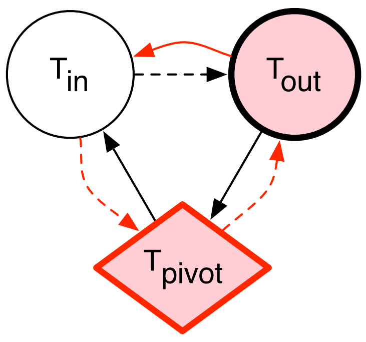


FIGURE 2.4: SDG example for three transactions exhibiting a read-only anomaly.

The papers described above give theorems which state that, under certain conditions on the programs making up an application mix, all executions of these programs will be serializable. What is the DBA to do, however, when s/he is faced with a set of programs that do not meet these conditions, and indeed may have non-serializable executions? A natural idea is to modify the programs so that the modified forms do satisfy the conditions; of course the modifications should not alter the essential functionality of the programs. Several such modifications were proposed in (Fekete *et al.*, 2005). The simplest idea to describe, and the most widely applicable, is called *materializing the conflict*. In this approach, a new table is introduced into the database, and certain programs are modified to include an additional statement which updates a row in this table. Another approach is *promotion*; this can be used in many, but not all, situations. We give more detail of these approaches below.

The idea behind both techniques is that we choose one edge of the two successive vulnerable edges that define a dangerous structure, and modify the programs joined by that edge, so that the edge is no longer vulnerable. We can ensure that an edge is not vulnerable, by making sure that some data item is written in both transactions (to be more precise, we make sure that some item is written in both, in all cases where a read-write conflict exists). Clearly we need to do this for one edge out of each pair that makes up a dangerous structure. If there are many dangerous structures, there are many choices of which edges to make non-vulnerable. (Jorwekar *et al.*, 2007) showed that choosing a minimal set of appropriate edges is NP-hard.

Example 4. *Figure 2.4 shows the static dependency graph for the example transaction programs given above in Example 3. Update transactions are shown shaded and have heavy outlines. The dangerous cycle of edges is highlighted, and the pivot is drawn as a diamond.*

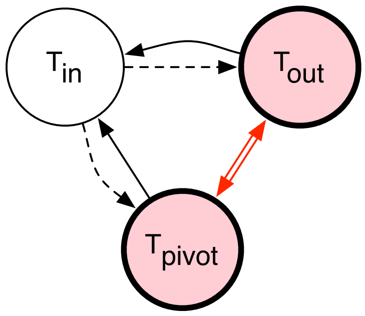


FIGURE 2.5: SDG example after introducing a conflict on the outgoing edge.

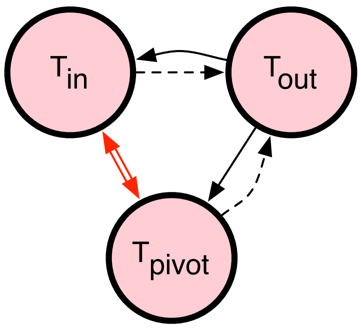


FIGURE 2.6: SDG example after introducing a conflict on the incoming edge.

Figures 2.5 and 2.6 show the effect on the SDG of making either the outgoing or incoming edge a write-write conflict in order to break the dangerous structure. In both cases, T_{pivot} is no longer a pivot because one of the consecutive vulnerable edges has been eliminated. Note, however, that the situations are not symmetrical: T_{out} is already an update transaction, so introducing an additional update on that edge does not change the basic nature of T_{out}. On the other hand, T_{in} is read-only in Example 3, so choosing to break the cycle on the incoming edge of T_{pivot} necessarily changes T_{in} into an update. Depending on the details of the technique chosen (promotion or materialization), in a more complex application the choice of edge and technique may have further implications for conflicts with other transactions in the application. This issue is explored in more detail in Section 2.8.5 when we discuss techniques for making the `SmallBank` benchmark serializable.

2.6.1 Materialization

To eliminate a vulnerable edge by materialization, we introduce a new table called `Conflict`, and an update statement into each program involved in the edge. The update statement modifies a row of the special `Conflict` table, which is not used elsewhere in the application. In the simplest approach, each program modifies a fixed row of `Conflict`; this will ensure that one of the programs aborts whenever they are running concurrently as a consequence of the the First-Committer-Wins. However, is is preferable to introduce contention only if it is needed to avoid rw-conflicts. Thus if we have programs P_1 and P_2 which have a read-write conflict when they share the same value for some parameter x, then we can place into each a statement:

```
UPDATE Conflict
   SET val = val + 1
 WHERE id = :x
```

This introduces a ww-conflict only when the programs share the same parameter x, which is precisely the case needed to prevent both concurrent transactions from committing.

2.6.2 Promotion

To use promotion to eliminate a vulnerable edge from P_1 to P_2, an update statement is added to P_1 called an *identity write*, which does not change the value of the item on which the read-write conflict occurs; P_2 is not altered at all. Suppose that for some parameter values, P_2 modifies some item in table `T` where a condition `C` holds, and P_1 contains:

```
SELECT ...
  FROM T
 WHERE C
```

An extra statement is added to P_1 as follows:

```
UPDATE T
   SET col = col
 WHERE C
```

Once again, the FCW rule will ensure that P_1 and P_2 do not run concurrently in situations where the parameter values mean that there is a read-write conflict. Promotion is less general than materialization, since it does not work for conflicts where one transaction changes the set of items returned in a predicate evaluation in another transaction.

In some platforms, there is another approach to promotion by replacing the SELECT statement (that is in a vulnerable read-write conflict) by `SELECT ...FOR UPDATE`. On some platforms, including the Oracle RDBMS, this is treated for concurrency control exactly like an update, and a transaction containing a `SELECT ...FOR UPDATE` cannot commit if it is concurrent with another that modifies the item. In other platforms, such as PostgreSQL, this statement prevents some but not all of the interleavings that give a vulnerable edge.

In particular, in PostgreSQL the following interleaving is permitted, even though it gives a vulnerable rw-conflict from T to U:

T	U
begin(T)	
$\vdots$	begin(U)
read-sfu(T, x)	$\vdots$
commit(T)	$\vdots$
	write(U, x)
	commit(U)

As a consequence, promotion using `SELECT ...FOR UPDATE` cannot be used on PostgreSQL (or other platforms with the same behavior).:w

2.6.3 Mixing Isolation Levels

All popular DBMS platforms allow applications to switch isolation levels for each transaction that is submitted to the system. This has implications when designing and implementing new concurrency control schemes: each transaction must get the requested isolation level regardless of operations that are executed in concurrent transactions at other isolation levels.

This is one of the reasons that snapshot isolation is often implemented using write locks: a transaction in the same system running at S2PL will acquire shared locks for the data items it reads, blocking a concurrent transaction running at SI from writing any of those items. If SI were implemented without write locking, by checking for conflicts optimistically before each SI transaction commits, a concurrent transaction that requested serializable isolation running at S2PL would not be guaranteed repeatable reads.

Mixing isolation levels is also an important technique that application can use to balance correctness with performance. In particular, the theory of (Fekete *et al.*, 2005) was extended in (Fekete, 2005) to the case where some transactions use SI and others use S2PL (as is possible with Microsoft SQL Server 2005 or Oracle Berkeley DB). It is shown that executing the *pivot* transactions at S2PL is sufficient to ensure that an application's execution will be serializable, but this requires analyzing the application in order to determine the static dependency graph so that the pivots can be identified.

Detailed performance studies (Alomari *et al.*, 2008b) analyzed the implications of mixing isolation levels to achieve serializable isolation. They indicate that with the optimal choice of technique, the impact of guaranteeing serializable isolation on throughput can be small. However, it is not always clear which technique will be optimal for a given mix of transaction programs on a particular platform.

2.6.4 Automating Static Analysis of Applications

(Jorwekar *et al.*, 2007) describes a system that automates the detection of transaction program conflicts using syntactic features of program texts, such as the names of the columns accessed in each statement. Working purely at the level of SQL syntax results in coarse granularity of conflict detection, resulting in a large proportion of false positives, where benign transaction programs were flagged by the tool for further (manual) analysis. When dangerous access patterns were manually confirmed, anomalies were avoided by modifying the transaction programs by hand. One important finding of their work is that snapshot isolation anomalies do exist in applications developed using tools and techniques that are common throughout the software industry.

An alternative approach to ensuring correctness when running on platforms with SI is in (Bernstein *et al.*, 2000), where conditions are given to ensure all executions preserve given integrity constraints, without necessarily being serializable.

2.7 Serialization Graph Testing

While locking techniques delay operations in order to guarantee serializable executions, another approach, called Serialization Graph Testing (SGT) schedulers, operate without locking by preventing any operation that would lead to a non-serializable execution. An SGT scheduler accepts requests from transactions that want to perform an operation, and determines whether each request would create a cycle in the serialization graph.

To decide whether to allow each operation, the SGT scheduler maintains a data structure representing the complete graph of conflicts between transactions. Nodes are added to the graph when a transaction first makes a request of the scheduler, and associated with each node is a representation of the set of data items read and written by the corresponding transaction. Edges are added to the graph when operations in concurrent transactions conflict. If a requested operation would create a cycle in the graph, the SGT scheduler will disallow the request, and the requesting transaction will usually rollback, as it would to handle a deadlock in a locking scheduler.

This description suggests that the data structure maintained by the SGT grows without bound. In fact, nodes and edges can be removed from the graph eventually, once there is no possibility that they will be involved in a cycle. Unfortunately, this may not be the case until some time after the transaction commits, so an SGT scheduler must maintain the graph for a sliding window into the complete history of execution (larger than the set of currently running transactions).

According to (Weikum and Vossen, 2002, page 170),

> *note that an SGT scheduler, although attractive from a theoretical point of view … is not practical to implement. For example, the (worst case)* space *required to maintain the serialization graph grows with the square of the number of relevant transactions, some of which are not even active anymore. We even need to keep around the information about the read sets and write sets of such already-committed transactions. In addition, and even more importantly, the testing of cycles in the graph, albeit only linear in the number of edges (i.e., quadratic in the number of nodes in the worst case), may be unacceptable at run time simply because such a test would be invoked*

very frequently. After all, concurrency control measures are part of a data server's innermost loops, and their run-time overhead is thus an extremely critical issue.

Previously, several researchers have proposed altering multiversion concurrency control algorithms to avoid non-serializable executions at run-time, by using a form of SGT. Examples are given in (Tada *et al.*, 1997; Shi and Perrizo, 2002; Yang, 2007). There is at least one commercial system purported to be based on SGT (Sarin, 2009), but the details are proprietary. Our thesis also suggests avoiding non-serializable executions at run-time, but we do not need to the cost of full cycle detection.

2.8 Database Performance Measurement

The measurement of database performance has been a topic of interest since the field began, and is the subject of entire textbooks (Kumar, 1995). Generally, benchmarks generate a workload representing some number of concurrent clients executing some mix of transactions in the DBMS. The number of clients is called the *multiprogramming level*, or MPL, and this is usually the independent variable in graphs of performance.

There are several different ways in which the database community has quantified the performance of different systems, including investigating the impact of different concurrency control algorithms. One approach has focused on measuring the performance under controlled conditions (Gray, 1993). Jim Gray was responsible for much of (Anonymous, 1985), which introduced a small banking set of programs as a benchmark, to allow fair comparison of performance, especially peak and sustainable throughput. The industry has formed a consortium, the Transaction Processing Performance Council (TPC), which has standardized many benchmarks, fixing precisely what is to be measured, and under what conditions (Transaction Processing Performance Council, 2005).

The two aspects of performance that are most commonly measured by benchmarks are the number of transactions that can be processed in a given amount of time, known as the *throughput*; and the total elapsed time required to execute a particular transaction, called the *response time*. Some benchmarks constrain one of these and measure the other: for example, setting a maximum response time and reporting the sustained throughput while requiring that 95% of measured response times are below the specified maximum. Another approach is to fix the throughput for a given data volume, so that in order to demonstrate higher throughput, the DBMS must be able to process a greater volume of data while maintaining response times.

Another way to investigate performance is by detailed simulations, where a software representation of a whole system is evolved through many steps. An early simulation paper was (Carey and Stonebraker, 1984). (Agrawal *et al.*, 1987a) compares the throughput obtained at different MPL, under different assumptions about the level of disk and CPU resources. A similar simulation study has particularly focused on multiversion concurrency control algorithms (Carey and Muhanna, 1986).

2.8.1 The TPC-C benchmark

One of the standard benchmarks developed by the Transaction Processing Performance Council for measuring the performance of the core transaction processing functionality of a DBMS is the TPC-C benchmark (Transaction Processing Performance Council, 2005). TPC-C models the database requirements of a business selling items from a number of warehouses to customers in some geographic districts associated with each warehouse.

TPC-C uses a mix of five concurrent transaction types of varying complexity. These transactions include creating and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. TPC-C exercises most of the components at the core of a DBMS storage engine by including:

- The simultaneous execution of multiple transaction types of varying complexity.
- Tests for transaction integrity (ACID properties).
- Moderate system and application execution time.
- Significant disk input/output, controlled by a scaling parameter.
- Non-uniform distribution of data access through primary and secondary keys.
- Databases consisting of tables with a variety of sizes, attributes, and relationships.
- Contention on data access and update.

TPC-C performance is measured in new-order transactions per minute. The primary metrics are the transaction rate (tpmC) and the associated price per transaction ($/tpmC). The table of top TPC-C results

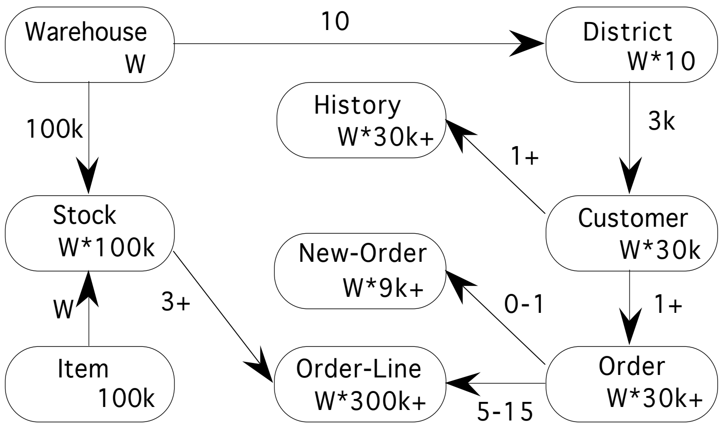


FIGURE 2.7: The tables and cardinality of TPC-C (Transaction Processing Performance Council, 2005).

has been an important driver of competition between the commercial DBMS vendors, who are locked in an ongoing race to each claim that they have the best OLTP system.

The schema and cardinality of TPC-C are given in Figure 2.7, and the benchmark consists of the following transactions:

- approximately 43% **New Order** transactions, abbreviated as NEWO, where a customer places an order for a list of different items;
- at least 43% **Payment** transactions, abbreviated as PAY, where a customer pays for an order placed earlier;
- 4% **Delivery** transactions, abbreviated as DLVY, which marks an order in each of 10 districts delivered. Following (Fekete *et al.*, 2005), we split the Delivery transaction into two parts: DLVY1, where there are no new orders, so there is no work for the transaction to do; and DLVY2, where there is at least one new order awaiting delivery;
- 4% **Order Status** transactions, abbreviated as OSTAT, where a customer checks the status of an order; and
- 4% **Stock Level** transactions, abbreviated as SLEVEL, where the company checks whether a warehouse is running low on any recently-ordered items.

It has long been known that TPC-C is serializable when run at snapshot isolation, and this was formally proved in (Fekete *et al.*, 2005). We adopt their abbreviations and analysis, and Figure 2.8 gives the static dependency graph they derived for TPC-C. One issue that arose in deriving the SDG for TPC-C was

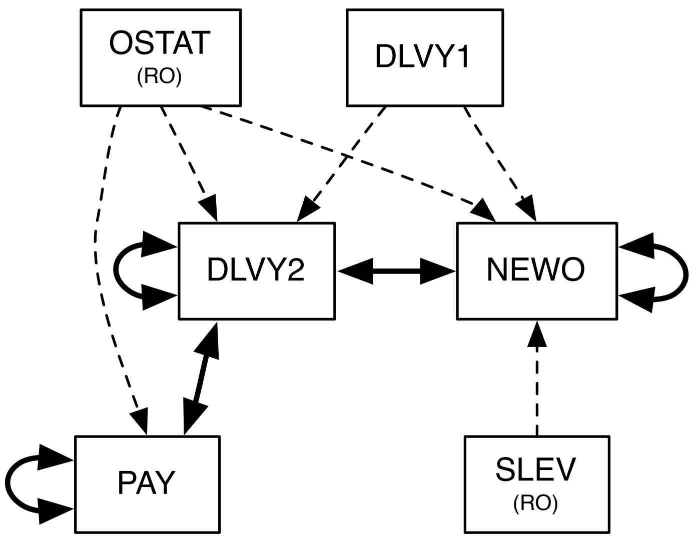


FIGURE 2.8: Static Dependency Graph for standard TPC-C, with rw-conflicts shown as dashed edges and ww-conflicts shown in bold (Fekete *et al.*, 2005).

that the description of the transactions in the TPC-C specification includes a mixture of SQL with flow control. Some of the TPC-C transactions behave quite differently depending on the state of the database that they encounter. For this reason, the TPC-C delivery transaction was split into two cases, DLVY1 and DLVY2, in order to remove ambiguity in the SDG and eliminate a superficial dangerous structure.

Once the SDG has been derived, however, it is easy to see by inspection of Figure 2.8 that it is free of cycles, and thus all executions will be serializable when TPC-C is executed with SI.

2.8.2 The `SmallBank` Benchmark

Standard benchmarks such as TPC-C (Transaction Processing Performance Council, 2005) are carefully designed to exercise a range of features of a system and would be of use in measuring the pure overhead of making SI serializable. However, we cannot use TPC-C directly to compare different ways of making applications serializable, since TPC-C itself generates only serializable executions on SI-based platforms, as described above. Thus in this thesis we have used a benchmark called `SmallBank`, published in (Alomari *et al.*, 2008a), which was designed to offer a diverse choice among modifications that will ensure serializable execution on SI. The `SmallBank` transaction mix is based on the example of an SI anomaly from (Fekete *et al.*, 2004) and provide some functionality reflecting a simple banking system.

The SmallBank benchmark models a simple banking application. The schema consists of three tables: Account(Name, CustomerID), Saving(CustomerID, Balance), Checking(CustomerID, Balance). The Account table represents the customers, with the name as primary key, and maps the names to unique customer IDs. The CustomerID is a primary key for both Saving and Checking tables. The Balance columns are numeric types, representing the balance in the corresponding account for one customer. The transaction types for balance (Bal), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS) and amalgamate (Amg) operations. Each of the transaction types involves a small number of simple read and update operations.

2.8.3 SmallBank Transaction Mix

The SmallBank benchmark runs instances of five transaction programs. All transactions start by looking up the customer's name in the Account table to retrieve the CustomerID.

Balance or Bal(N): a parameterized transaction that represents calculating the total balance for a customer with name N. It returns the sum of savings and checking balances for the specified customer.

DepositChecking, or DC(N, V): is a parameterized transaction that represents making a deposit into the checking account of a customer. Its operation is to increase the checking balance by V for the given customer. If the value V is negative or if the name N is not found in the table, the transaction will rollback.

TransactSaving, or TS(N, V): represents making a deposit or withdrawal on the savings account. It increases or decreases the savings balance by V for the specified customer. If the name N is not found in the table or if the transaction would result in a negative savings balance for the customer, the transaction will rollback.

Amalgamate, or Amg(N1, N2): represents moving all the funds from one customer to another. It reads the balances for both accounts of customer N1, then sets both to zero, and finally increases the checking balance for N2 by the sum of N1's previous balances.

WriteCheck, or WC(N, V): represents writing a check against an account. Its operation is evaluate the sum of savings and checking balances for the given customer. If the sum is less than V, it decreases the checking balance by V + 1 (reflecting a penalty of $1 for overdrawing), otherwise it decreases the checking balance by V.

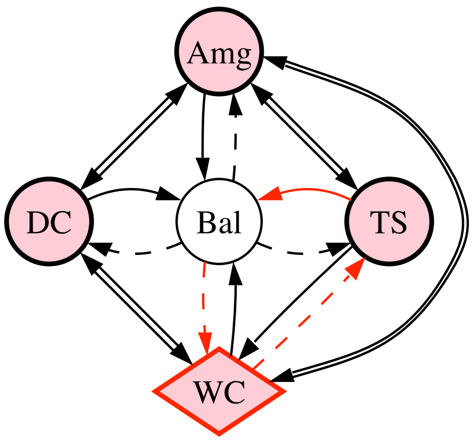


FIGURE 2.9: Static dependency graph for the `SmallBank` benchmark (Alomari *et al.*, 2008a).

2.8.4 `SmallBank` Static Dependency Graph

The static dependency graph for `SmallBank` is given in Figure 2.9, where the double arrows represent write-write conflicts and the dashed arrows represent read-write conflicts. Most of the analysis is quite simple, since TS, Amg and DC all read an item only if they will then modify it; from such a program, any read-write conflict is also a write-write conflict and thus not vulnerable. The edges from Bal are clearly vulnerable, since Bal has no writes at all, and thus a read-write conflict can happen when executing Bal concurrently with another program having the same parameter. The only subtle cases are the edge from WC (which reads the appropriate row in both Checking and Saving, and only updates the row in Checking). Since TS writes Saving but not Checking, the edge from WC to TS is vulnerable. In contrast, whenever Amg writes a row in Saving it also writes the corresponding row in Checking; thus if there is a read-write conflict from WC to Amg on Saving, there is also a write-write conflict on Checking (and so this cannot happen between concurrently executing transactions). That is, the edge from WC to Amg is not vulnerable.

It can be seen by inspection that there is a dangerous structure Balance (Bal) $\rightarrow$ WriteCheck (WC) $\rightarrow$ TransactSavings (TS) $\rightarrow$ Balance, so the transaction WriteCheck is a pivot. The other vulnerable edges run from Bal to programs which are not in turn the source of any vulnerable edge. The non-serializable executions possible are like the one in (Fekete *et al.*, 2004), in which Bal sees a total balance value which implies that a overdraw penalty would not be charged, but the final state shows such a penalty because WC and TS executed concurrently on the same snapshot.

2.8.5 Making `SmallBank` Serializable

Using the techniques we described in Section 2.6, there is a choice of which of the vulnerable edges to eliminate: either the edge from WriteCheck to TransactSaving can be made non-vulnerable (option *WT*), or the edge from Balance to WriteCheck can be made not vulnerable (option *BW*). And as discussed in Section 2.6, there are two main alternatives on how to make each option non vulnerable, giving the following options:

MaterializeWT: eliminate the vulnerable edge from WriteCheck to TransactSaving by materializing the conflict. A table `Conflict` is created, and not accessed elsewhere in the application, with schema Conflict(CustomerID, Value). In order to introduce write-write conflicts only when the transactions actually have a read-write conflict (that is, when both deal with the same customer), only the row in the Conflict table with the matching CustomerID is updated.

PromoteWT: eliminates the vulnerable WT edge by promotion, which involves adding an identity update to the `Saving` table in the WriteCheck transaction (or `SELECT FOR UPDATE` on some systems).

MaterializeBW: eliminate the vulnerable edge from Balance to WriteCheck by materializing the conflict. This can use the same `Conflict` table introduced for MaterializeWT, since the same key, CustomerID is involved in both cases.

PromoteBW: eliminate the vulnerable BW edge by promotion, which involves adding an update to the Checking table in the Balance transaction.

Apart from MaterializeWT, all of these options introduce updates into the Balance transactions, which was originally read-only. We give the modified SDG for the PromoteBW case in Figure 2.10. Note that in this case, the conflict edges from Balance to other transactions (not part of the vulnerable edge) have changed to write-write conflicts. These additional conflicts were found in (Alomari *et al.*, 2008a) to have a substantial negative impact on throughput. However, this is exactly the technique recommended by DBMS documentation (Ashdown and others, 2005) for ensuring serializable execution on databases that only offer SI!

In evaluating the performance impact of each of these options, (Alomari *et al.*, 2008a) found that:

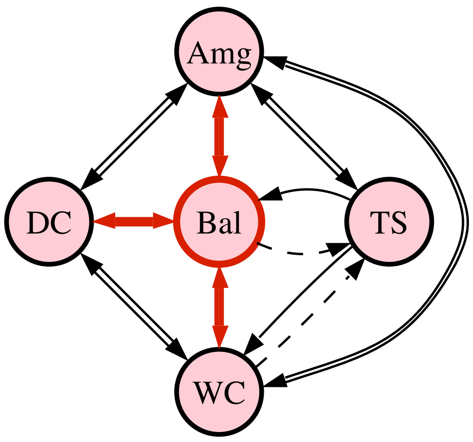


FIGURE 2.10: `SmallBank` SDG after promoting the Balance–WriteCheck conflict (Alomari *et al.*, 2008a).

- Simple approaches to ensuring serializable executions (without analyzing the SDG) should be avoided.
- One should avoid modifications that turn a query into an update, if possible.
- If a transaction type is executed with higher frequency in the application, or has stronger performance requirements, one should prefer techniques that avoid modifications to the high-frequency transaction type.
- Promotion is faster than materialization in PostgreSQL, but the opposite is true for the commercial platform.

In summary, complex analysis and platform-specific evaluation are required in order to make the most efficient use of the techniques described in Section 2.6, and the straightforward approaches to ensuring serializable executions can have a significant impact on performance. On the other hand, the existence of techniques that guarantee serializable executions with minimal impact on performance was an important motivation for our work.

2.9 Summary

In this chapter, we have introduced the concepts that underpin our work. In particular, we have described:

- the notion of a transaction;

- serializable executions of a set of transactions;
- conflict graphs and conditions under which a conflict graph corresponds to a serializable execution;
- weak isolation levels and anomalies;
- multi-version concurrency control and a particular instance, snapshot isolation;
- anomalies permitted by snapshot isolation: write skew and phantoms;
- conditions on application under which snapshot isolation gives serializable execution and techniques for modifying applications to make the serializable;
- serialization graph testing, an elegant but impractical approach to ensuring serializable execution; and
- a brief overview of the field of database performance evaluation, in particular describing the TPC-C and `SmallBank` benchmarks, and the choices involved in making `SmallBank` serializable under SI.

All of these ingredients are combined and contrasted in subsequent chapters as we describe our new algorithm for serializable isolation and evaluate its performance.

CHAPTER 3

Serializable Snapshot Isolation

The essence of our new concurrency control algorithm is to allow the standard snapshot isolation algorithm to operate, but to add some book-keeping so we can dynamically detect cases where a non-serializable execution could occur, and then we abort one of the transactions involved. This makes the detection process a delicate balance: if there are any cases that are not detected, some non-serializable execution may emerge (counter to our goal of having true serializability guarantees for the applications), but if we detect too many cases, then performance will suffer as unnecessary aborts waste resources. In addition, when designing an algorithm, we also need to keep the overhead cost of detection low. One can imagine a concurrency control algorithm which aborts a transaction exactly when an operation will result in a non-serializable execution; this would be a serialization graph testing (SGT) scheduler using the appropriate multiversion serialization graph. SGT schedulers, however, require expensive cycle detection calculations on each operation and would be very expensive, as discussed in Section 2.7. Thus we accept a small chance of unnecessary aborts, in order to keep the detection overhead low.

The key design decision in our new algorithm is thus carefully defining the situations in which potential anomalies are detected. We use the theory of (Adya, 1999) and its extension from (Fekete *et al.*, 2005), presented in Section 2.5.1, which proves that a distinctive pattern of conflicts appears in every non-serializable execution of SI. The building block for this theory is the notion of a rw-dependency (also called an "anti-dependency"), which occurs from transactions T_1 to T_2 if T_1 reads a version of an item x that is older than a version of x produced by T_2. In (Adya, 1999) it was shown that in any non-serializable SI execution, there are two rw-dependency edges in a cycle in the multiversion serialization graph. (Fekete *et al.*, 2005) extended this, to show that there are two rw-dependency edges which form consecutive edges in a cycle. Furthermore, each of these rw-edges involves two transactions that are active concurrently and the outgoing transaction from the pivot, T_{out} in Figure 2.2, must commit first.

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

We note that our algorithm is conservative: it prevents every non-serializable execution, because a transaction is aborted whenever there is a transaction T for which both `T.inConflict` and `T.outConflict` are set. However, we do sometimes make false positive detections; for example, an unnecessary detection may happen because we do not check whether the two rw-dependency edges occur within a cycle. It is also worth mentioning that we do not always abort the particular pivot transaction T for which `T.inConflict` and `T.outConflict` is true; this is often chosen as the victim, but depending on the interleaving of operations, the victim may be the transaction that has an rw-dependency edge to T, or the one that is reached by an edge from T.

It is possible with Serializable SI that a pattern of unsafe conflicts is detected, causing a transaction to be aborted and then restarted, and then the restarted transaction again causes an unsafe pattern of conflicts and is forced to abort. However, the impact of this issue is ameliorated by two facts:

- the rate of detection of unsafe conflict patterns is very low in all the cases we have measured (in the most extreme case, unsafe pattern detection increased the overall abort rate by 10%, but in the majority of cases the increase was less than 1%); and
- it is very rare for our algorithm to abort pure queries, and updates involved in conflicts usually wait for at least one write lock before they are aborted. By the time the write lock is granted, the conflicting operation has completed, so if the update is forced to retry, it will not have the same pattern of conflicts.

Our performance evaluation in Section 6 supports our belief that cascading aborts do not occur often enough in practice with Serializable SI to cause a measurable impact on throughput.

The remainder of this chapter is structured as follows. Section 3.1 discusses some of the design alternatives we considered while developing our approach. We then describe Serializable SI in detail. For ease of exposition, we start by describing a basic version of the algorithm in Section 3.2, corresponding closely to what we implemented in our Berkeley DB prototype. Then, in Section 3.3, we describe the implications for the lifecycle of transactions, and Section 3.4 presents an argument for the correctness of the basic algorithm. In Section 3.5, we extend the basic algorithm to detect phantoms in systems using fine-grained locking. Section 3.6 extends the basic algorithm to improve its precision and in Section 3.7 we describe some further optimizations, implemented in our InnoDB prototype.

3.1 Design alternatives

In developing the algorithm described later in this chapter, we considered various alternative approaches to preventing anomalies. These alternatives are discussed here in order to clarify how we arrived at the final design we implemented and evaluated.

3.1.1 After-the-fact analysis tool

We first explored whether we could help application developers make their applications serializable under SI with a tool that performed after-the-fact analysis of the transaction logs from a DBMS to determine whether a particular execution was serializable. This involved logging more information than a DBMS would usually log, since some information about the set of items read by a transaction is required in order to build a conflict graph and check whether it contains cycles. At a high level, this approach is similar to that of (Jorwekar *et al.*, 2007), though our focus was execution traces rather than static analysis.

If successful, this approach would have led to a testing tool for developers. However, as non-serializable executions can be difficult to generate repeatably with either unit tests or randomized testing, the tool would have been of limited value: all a developer could be confident about if the tool did not find any anomalies is that they are uncommon in the tested scenario. There is no way that such a tool could be used to prove that an application is serializable. We did consider creating a tool that would generate all possible interleavings of the execution extracted from the DBMS logs, but this opens the door to false positives, and is still not sufficient to prove that the application is serializable in general.

3.1.2 Approaches to runtime conflict detection

When considering the dynamic detection of rw-conflicts, our first idea was to use the existence of newer, committed versions of a data item that are ignored during a read as the sole mechanism for detecting read-write conflicts. That is, while reading a data item x, if a newer committed version of x is ignored, we would flag that a rw-conflict had occurred. However, we soon realized that in concurrent transactions T_1 and T_2, the read of a data item x in T_1 can happen before an update to x in T_2 (as in Figure 2.3(a)). In other words, the newer version may not be created until after the read occurs, so relying solely on the existence of newer versions during reads is not sufficient to detect all rw-conflicts.

We then thought about using the lock manager to detect all conflicts, inspired by the way S2PL guarantees serializability. The idea was not to delay transactions, but instead to use the existing DBMS system for detecting that two transactions had accessed the same data item. This raised several issues:

- in order to avoid blocking any transactions, a new lock mode would be required.
- the normal scoping of locks (releasing locks at commit time) is not sufficient to detect all conflicts, as conflicts may occur after one of the transactions involved has committed.
- holding exclusive locks for any longer than the scope of the transaction would lead to a substantial reduction in throughput due to increased blocking and potentially higher deadlock rates.
- we only want to flag a conflict if the two transactions involved overlap. If locks persist (temporarily) after a transaction commits, the lock manager needs to check that the transaction requesting a lock overlaps with the transaction holding any conflicting locks before indicating that a conflict has occurred.
- this approach increases the size of the lock table and the number of requests made to the lock manager.

Next we considered having a list of conflicts for each transaction, tracking the whole conflict graph as done for SGT schedulers. However, we quickly discovered that maintaining this graph introduces its own problems of concurrency control in addition to the space overhead required to represent it.

Another approach we pursued was to associate a read timestamp with each active data item, indicating the time of the most recent read to that item. An updating transaction could use the read timestamp to detect whether it was causing a read-write conflict by comparing the read timestamp of each data item it read with its begin timestamp. A read timestamp more recent than the transaction's begin timestamp would indicate a rw-conflict similar to the way that the existence of a version created after the begin timestamp indicates that a ww-conflict occurred. However, a single timestamp cannot represent multiple concurrent readers, and if a conflict is detected, we could mark an incoming conflict in the updating transaction, but this approach gave us no way to find the reader(s) and mark their outgoing conflict.

Finally, we combined the checks for ignored versions of data items with the use of the lock manager to arrive at the algorithm we present in this chapter. The advantage of this combination is primarily that all rw-conflicts are detected. Further, while we do place some additional load on the lock manager, the only locks that need to be held past a transaction's commit are the new non-blocking entries in the lock table. No scope changes are required for exclusive locks.

3.2 The Basic Algorithm

How can we keep track of situations where there is an rw-dependency between two concurrent transactions? As we remarked in Section 3.1.2, we found that there are two different ways in which we notice a rw-dependency between two concurrent transactions. One situation arises when a transaction T reads a version of an item x, and the version that it reads (the one which was valid at T's start time) is not the most recent version of x. In this case the writer U of any more recent version of x was active after T started, and so there is a rw-dependency from T to U. When our algorithm detects this, we set the flags `T.outConflict` and `U.inConflict` (and we check for consecutive edges and abort a transaction if needed). This allows us to detect rw-dependency edges for which the read operation is interleaved after a write, even though the write operation would be later in a serial execution. However, it does not account for edges where the read is interleaved first, and subsequently a new version is created by a concurrent transaction.

To notice these other rw-dependency cases, we rely on the lock management infrastructure of the DBMS. Most SI implementations acquire a normal exclusive lock anyway when a new version is created, as a way to enforce the First-Committer-Wins rule, and avoid cascading aborts. Since the update is going to make a request of the lock manager anyway, we want to introduce special entries to the lock table that the lock manager can use to detect these read-before-write conflicts when the write occurs.

We introduce a new lock mode called SIREAD, which is used to record the fact that an SI transaction has read a version of an item. However, obtaining the SIREAD lock does not cause any blocking, even if an exclusive lock is held already, and similarly an existing SIREAD lock does not delay granting of an exclusive lock; instead, the presence of both SIREAD and exclusive locks on an item is a sign of an rw-dependency, and so we set the appropriate `inConflict` and `outConflict` flags on the transactions which hold the locks. One difficulty, which we discuss later, is that we need to keep the SIREAD locks that T obtained, even after T is completed, until all transactions concurrent with T have completed.

Note, however, that our approach does *not* require that exclusive locks are held for any longer than they are normally held in order to implement SI. Exclusive locks can be dropped when a transaction commits: if a concurrent transaction subsequently reads one of the locked items, it will detect the conflict because of the presence of the new version of the item.

We now present the Serializable SI concurrency control algorithm. In describing the algorithm, we make some simplifying assumptions:

(1) For any data item `x`, we can efficiently get the list of locks held on `x`.
(2) For any lock `l`, we can efficiently get `l.owner`, the transaction object that requested the lock.
(3) For any version `xt` of a data item `x`, we can efficiently get `xt.creator`, the transaction object that created that version.
(4) When finding a version of item `x` valid at some given timestamp, we can efficiently get the list of other versions of `x` that have later timestamps.
(5) There is latching or other concurrency control in the DBMS so that read and write operations involving a data item `x` have an unambiguous order.

These assumptions are true for both of our prototype target systems (Berkeley DB and InnoDB). In particular, both lock tables are structured for efficient lookup of locks, there is a direct pointer from locks to the owning transaction and there is a table lookup required to find the transaction that created a given version of a data item. We discuss in Section 4.8 how to implement the algorithm if these assumptions do not hold.

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

FIGURE 3.1: Modifications to `begin(T)`, called to start a Serializable SI transaction.

Our algorithm makes the following changes to data structures:

- two boolean flags are added to each transaction object: `T.inConflict` indicates whether or not there is a rw-dependency from a concurrent transaction to T, and `T.outConflict` indicates whether there is a rw-dependency from T to a concurrent transaction; and
- the new SIREAD mode for locks, assuming a lock manager that already keeps standard exclusive locks.
- there is a new state for transactions that have committed but are kept suspended in order to correctly detect conflict that occur later; this may require a new flag kept with each transaction and/or a new queue for suspended transactions, depending on the details of the DBMS internals.

The Serializable SI algorithm modifies the operations of standard snapshot isolation as shown in Figures 3.1 to 3.5. In each case, the processing includes the usual processing of the operation by the SI protocol plus some extra steps.

The modification to the code to begin a transaction is given in Figure 3.1. This establishes the two new boolean flags used by the Serializable SI algorithm, both initially false.

The modification to transaction commit is given in Figure 3.2. First, the modified code must mark that the transaction is no longer running. The definition of whether a transaction "has committed" deserves some discussion. The intent is to ensure that the conflict bits will cause a running transaction to abort, or else if the transaction *would have* aborted but has already gone on to commit, the conflicting transaction must abort instead. So this test for whether a transaction is running must be atomic with respect to the check of the conflict flags at the beginning of `commit(T)`. After the flags have been checked during commit, a transaction can no longer abort due to the conflict flags being set, and should be considered committed when checking for conflicts. These operations were made atomic in the prototype implementations described later by performing them while holding appropriate system latches (for example, the latch protecting the log that is acquired during transaction commit).

```
1  modified commit(T):
2      atomic begin:
3          if T.inConflict and T.outConflict:
4              abort(T)
5              return UNSAFE_ERROR
6          mark T as committed
7      atomic end
8
9      existing SI code for commit(T), do not release SIREAD locks
10
11     if T holds any SIREAD locks:
12         suspend T, keeping SIREAD locks active
13     if T was the oldest active transaction:
14         clean up any suspended transactions older than active transactions
```

FIGURE 3.2: Modifications to `commit(T)`, called to commit a Serializable SI transaction, checking that no unsafe pattern of conflicts has been detected.

Then in line 4 of Figure 3.2, we check the two new flags we have added to the transaction handle. If both flags are set, that indicates that both an incoming and outgoing conflict were detected while the transaction was running, so it may be the *pivot* of a cycle in the conflict graph. If two consecutive edges are detected here, the transaction is aborted rather than continuing with the commit, and an error is returned to the application to indicate that an error was detected.

If no unsafe pattern of conflicts is detected, we continue with the existing code to commit the transaction. Then if the transaction holds any SIREAD locks, it is temporarily suspended until all concurrent transactions have completed. In addition, we check whether the committing transaction was the oldest active transaction in the system, and if so, clean up any suspended transactions that could no longer participate in a conflict. This is discussed in more detail below in Section 3.3.

When a conflict is detected between a reading transaction and a writing transaction, the processing is somewhat complicated by the fact that either the reader or the writer may have already committed. In addition, as described above in the discussion of Figure 3.2, the check of whether the transaction is running must be atomic with respect to the corresponding flags check during commit. We put this processing into a separate utility function called `markConflict` and give its pseudocode in Figure 3.3.

When acquiring locks in Figures 3.4 and 3.5, the notation `key=x` indicates that a lock is acquired on the data item `x`, not any particular version or value that `x` holds. Locks acquired with `owner=T` are acquired on behalf of transaction `T`, and do not conflict with other locks held by `T`.

```
markConflict(reader, writer):
    atomic begin:
        if writer has committed and writer.outConflict:
            abort(reader)
            return UNSAFE_ERROR
        if reader has committed and reader.inConflict:
            abort(writer)
            return UNSAFE_ERROR

        set reader.outConflict = true
        set writer.inConflict = true
    atomic end
```

FIGURE 3.3: A utility function, `markConflict(reader, writer)`, called when a conflict is detected.

```
modified read(T, x):
    get lock(key=x, owner=T, mode=SIREAD)
    if there is a conflicting EXCLUSIVE lock(wl) on x:
        markConflict(T, wl.owner)

    existing SI code for read(T, x)

    for each version (xNew) of x that is newer than what T read:
        markConflict(T, xNew.creator)
```

FIGURE 3.4: Modifications to `read(T, x)`, called to read a data item in a Serializable SI transaction.

```
modified write(T, x, xNew):
    get lock(key=x, owner=T, mode=EXCLUSIVE)

    for each conflicting SIREAD lock(rl)
      where rl.owner has not committed or commit(rl.owner) > begin(T):
        markConflict(rl.owner, T)

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

FIGURE 3.5: Modifications to `write(T, x, xNew)`, called to modify a data item in a Serializable SI transaction.

The modified `read(T, x)` operation cannot omit either the check for a conflicting exclusive lock or the check for newer versions of x. If either check were omitted, a race with a write operation on x would be possible, since we make no assumption about mutual exclusion between operations outside the of the lock subsystem, including obtaining a version of a particular item. If the read operation did not check for conflicting exclusive locks, the following interleaving of threads would be possible:

T	U
	get lock(x, U, EXCLUSIVE)
get lock(x, T, SIREAD)	
read(T, x)	
	write(U, x, $xNew$)

The result would be that neither thread detects the conflict. To avoid this race condition and ensure that all conflicts are detected, our modification to the `read` operation must check both for lock conflicts and for new versions of data items.

3.3 Transaction lifecycle changes

For the Serializable SI algorithm, it is important that the engine have access to information about a transaction T – its transaction record, including `T.inConflict` and `T.outConflict`, as well as any SIREAD locks it obtained – even after T has completed. The transaction record for T and any SIREAD locks acquired by T must be kept as long as there is any active transaction that overlaps with T. That is, we can only remove information about T after the end of every transaction that started before T completed. For this reason, transactions that commit holding an SIREAD lock are suspended temporarily rather than being immediately cleaned up in Figure 3.2. This suspend operation does not delay the client: the commit returns immediately, but the transaction record is kept so that future conflicts can be detected correctly.

Suspending transactions during commit implies that the suspended transactions are cleaned up at some later point. Information about a transaction T can be discarded once there is no transaction still running that was concurrent with T. It is important for efficiency that the set of suspended transactions and the lock table do not grow unnecessarily large. Details of how to ensure prompt clean-up are quite different between systems. In Section 4.3.1, we describe the approach we took in the Berkeley DB prototype, and in Section 4.6.1 we describe how this information is managed in the InnoDB implementation – in particular, how the space allocated to transaction objects is reclaimed.

3.4 Correctness

The Serializable SI algorithm ensures that every execution is serializable, and thus that data integrity is preserved (under the assumption that each transaction individually is coded to maintain integrity). Here we outline the argument that this is so. Note that here we only consider transactions consisting of read and write operations on individual data items. Predicate reads and the associated problem of phantoms are discussed in Section 3.5.

By Theorem 2 of Section 2.5.1 (originally from (Fekete *et al.*, 2005)), which shows that in any non-serializable execution there is a dangerous structure, we are done provided that we can establish the following: whenever an execution contains a dangerous structure (transactions T_{in}, a pivot T_{pivot}, and T_{out}, such that there is a rw-dependency from T_{in} to T_{pivot} and T_{in} is concurrent with T_{pivot}, and also there is a rw-dependency from T_{pivot} to T_{out} and T_{pivot} is concurrent with T_{out}), then one of the transactions is aborted. In this situation, we must consider the possibility that T_{in}=T_{out}, which is the classic example of write skew.

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

One of these operations (`read(T, x)` and `write(U, x)`) will happen first, because we assume that the database engine will perform some latching during their execution, and the other will happen later. The rw-dependency is present in the execution once the second of these operations occurs. If this second operation is `read(T, x)`, then at the time that operation is processed, there will already be the version of `x` created by U; lines 8-9 of the pseudocode in Figure 3.4 shows that we explicitly set both flags as required. On the other hand, if the `write(U, x)` occurs after `read(T, x)`, then there are two cases:

(1) T gets an SIREAD lock on x before U requests its EXCLUSIVE lock, and the pseudocode in Figure 3.5 will detect the conflict; or

(2) U acquires its EXCLUSIVE lock on x first, then T gets an SIREAD lock, and the conflict is detected by lines 2-4 of Figure 3.4.

In all cases, regardless of the interleaving of the lock requests or the read and write operations, a rw-conflict will be marked between T and U.

Based on the invariant just described, we now must argue that one of the transactions in any dangerous structure is aborted. If both rw-dependencies exist at the time T_{pivot} completes, then the code in Figure 3.2 will notice that both `inConflict` and `outConflict` are set for T_{pivot} (because of the invariant), and so T_{pivot} will be aborted when it requests to commit.

If, however, T_{in} reads a data item x written by T_{pivot} after T_{pivot} has committed, then a new version $xNew$ will be detected at line 8 of Figure 3.4. If T_{pivot} committed without holding an SIREAD lock, it would not have been suspended, so it is possible that the creator of $xNew$ is no longer available. However, in this case, T_{pivot} was a pure update, so it could never have a read-write conflict with a transaction, and thus could not be part of a cycle.

Assuming that T_{pivot} is the creator of $xNew$ and is available in the call to `markConflict`, if an outgoing conflict has already been detected for T_{pivot}, we must abort the reader, T_{in}, in order to prevent a cycle. Otherwise, conflicts are marked as usual in both transactions.

Similarly, if T_{out} writes a data item x that T_{pivot} read, then T_{pivot} must have taken an SIREAD lock on x and so would have been suspended during the commit. Then when T_{out} performs the write operation on x, the code in Figure 3.5 will find T_{pivot}'s SIREAD lock and detect the conflict if the two transactions were concurrent. If T_{pivot} already has an outgoing edge, we must abort T_{out} in order to prevent a cycle. Otherwise, the conflicts are marked on both transactions as before.

Therefore, even if both rw-dependencies occur after T_{pivot} commits, the algorithm will attempt to set both flags in T_{pivot}'s transaction structure, and the transaction responsible for the last detected dependency will be aborted.

In summary, the argument for correctness is as follows:

(1) Non-serializable executions under SI consist of a cycle including two consecutive rw-dependencies.
(2) Our algorithm detects every rw-dependency.
(3) When two consecutive rw-dependencies are detected, at least one transaction is aborted which breaks the potential cycle.

The exhaustive testing of the implementation that we describe below in Section 4.7 further supports this argument for the algorithm's correctness.

3.5 Detecting phantoms

A concurrency control algorithm for a relational DBMS must consider phantoms, where an item is created or deleted in one transaction, and the result is incorrectly not seen in a predicate operation in a concurrent transaction. Pure record-level locking, where readers simply lock the records they read and writers only lock the records they insert or delete, is not sufficient to prevent phantoms, because a record can be inserted that is logically within a range read by a concurrent query with no lock conflict occurring. Standard snapshot isolation also permits a form of write skew due to phantoms that must be prevented in order to guarantee serializable execution.

The target system for our first prototype implementation, Berkeley DB, acquires locks at the granularity of database pages rather than on individual records. Reading a record on page P with Berkeley DB's implementation of S2PL prevents any concurrent transaction from modifying page P, which includes updating or deleting any existing record on page P or inserting a new record onto page P. This coarser granularity of locking, combined with the fact that whenever a new page is inserted, some existing page is updated to link to the new page, means that the SIREAD locking protocol we presented in Section 3.2 is sufficient to prevent phantoms in Berkeley DB for the same reason that the existing Berkeley DB S2PL implementation is serializable. The enhancement we describe here is required for systems that use record-level locking.

The approach to preventing phantoms commonly used in platforms based on record-level S2PL (including InnoDB) are based on "next-key locking", including its "gap locking" refinement, as described in Section 2.5.2. In these schemes, a range of key space is protected against concurrent inserts or deletes by acquiring a shared lock on the gap after each row in order, as a scan is made to check whether rows

```
modified scanRead(T, x):
    get gap lock(key=next(x), owner=T, mode=SIREAD)
    if there is a conflicting EXCLUSIVE gap lock(wl) on next(x):
        markConflict(T, wl.owner)

    modified Serializable SI read(T, x)
```

FIGURE 3.6: Modifications to `scanRead(T, x)`, called when `x` is read during a scan to evaluate a predicate in a Serializable SI transaction.

```
modified insert(T, x) and delete(T, x):
    get gap lock(key=next(x), owner=T, mode=EXCLUSIVE)

    for each conflicting SIREAD gap lock(rl) on next(x)
      where rl.owner has not committed or commit(rl.owner) > begin(T):
        markConflict(rl.owner, T)

    insert: modified Serializable SI code for write(T, x, x)
    delete: modified Serializable SI code for write(T, x, NULL)
```

FIGURE 3.7: Algorithms for `insert(T, x)` and `delete(T, x)` in Serializable SI transactions.

match a predicate. Inserts and deletes follow the same protocol, obtaining an exclusive lock on the gap after the row being inserted or deleted.

Our solution for detecting phantoms in snapshot isolation builds on gap locking by adding the SIREAD lock mode to gap locks for scans and using these locks to detect predicate read-write conflicts between transactions. In Figures 3.6 and 3.7, we provide an extension to the basic Serializable SI algorithm to prevent phantoms.

The notation `next(x)` refers to the key in the table that sorts after `x`, or a special supremum key if `x` is the last key in the table.

If a predicate read operation occurs *before* the conflicting write operation, the enhanced Serializable SI algorithm given here detects the predicate-rw-conflict between transactions correctly.

In the case where a predicate read is interleaved *after* the conflicting write operation, the read would find a data item with a creation or deletion timestamp greater than the read timestamp of the predicate read. One situation in where a row is skipped because it has been inserted after the transaction began: no version of this row was visible when the reading transaction started. Another situation is where a

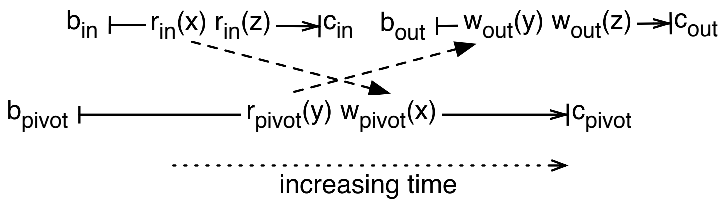

b_in r_in(x) r_in(z) c_in b_out w_out(y) w_out(z) c_out
b_pivot r_pivot(y) w_pivot(x) c_pivot
increasing time

FIGURE 3.8: False positive: no path from T_{out} to T_{in}.

row is accessed by a predicate read, but that row has been deleted since the transaction began. A special *tombstone version* is installed to indicate when this has occurred, and this is still a "newer version" for our purposes. In either case, the newer version of the row will trigger the detection of a conflict as described in Figure 3.4 without further modification. Note that tombstone versions can be reclaimed whenever no transaction could read the last valid version: at that point, all versions of the data item are gone so the tombstone has no further function.

This is implemented in our InnoDB prototype by building on the existing gap locking protocol built into InnoDB to prevent phantoms for the locking serializable isolation level. Insert and delete operations already acquire EXCLUSIVE locks on the gap, so all that was necessary was to have SIREAD locks for predicate reads follow the same protocol, and detect conflicts in exactly the same way as with ordinary row locks.

3.6 False positives

Our algorithm uses a conservative approximation to cycle detection in the graph of transaction conflicts, and as such may cause some benign transactions to abort. The problem is that the two flags we have added to each transaction cannot indicate whether there is a complete cycle in the conflict graph. On the other hand, we expect that general cycle detection in the full conflict graph would require a prohibitive amount of additional memory and CPU time.

In particular, the interleaving of transactions in Figure 3.8 will set `inConflict` on T_{pivot} when it executes w_{pivot}(x) and finds the SIREAD lock from T_{in}. Then `outConflict` will be set on T_{pivot} when T_{out} executes w_{out}(y) and finds T_{pivot}'s SIREAD lock. During the commit of T_{pivot}, the two flags will be checked and since both are set, T_{pivot} will abort. However, this interleaving is equivalent to the serial history $\{T_{in}, T_{pivot}, T_{out}\}$ because T_{in} precedes T_{out} and hence there is no path of dependencies from T_{out} to T_{in}.

To address this, we have enhanced the algorithm described above by adding incoming and outgoing *transaction references* rather than single bits. These references are initially set to `NULL`, then set to refer to the transaction that first causes each conflict. If there are multiple conflicts of either type, the reference is set to point to the transaction itself. We use a self-reference as a convenience: it corresponds to a self-loop in the MVSG, and has the advantage that the referenced transaction will be valid exactly when needed. Setting the reference to either `NULL` or to the transaction itself takes the place of a single bit set to 0 or 1 in the basic algorithm. We present pseudocode for these modifications in Figure 3.9 and 3.10.

The theory of (Fekete *et al.*, 2005) shows that in any non-serializable execution, there is a dangerous structure where the outgoing transaction, T_{out} in Figure 2.2 commits before all others in the cycle. Thus our algorithm still ensures serializability, even if we allow a pair of consecutive rw-dependencies, as long as we prevent situations where T_{out} commits first. Our algorithm does not track all transactions involved in the cycle, but if T_{out} commits before all transactions in the cycle, it must at least commit before either T_{pivot} and T_{in}. By using transaction references rather than single bits, we can check whether T_{in} or T_{pivot} has committed before T_{out}, and if so, avoid an abort in that case without the possibility of causing a non-serializable execution. In other words, using transaction references rather than single bits makes the detection of non-serializable executions more precise, with fewer false positives than the basic scheme described in Section 3.2.

Since transactions are only suspended temporarily and then deleted, references between transactions cannot be kept indefinitely. To deal with this, at commit time if either the incoming or outgoing transaction references are set to already-committed transactions, they are replaced with self-references to the committing transaction before it is suspended, as shown in Figure 3.10. In other words, our modified algorithm has the invariant that suspended transactions only ever reference transactions with an equal or later commit. Since we clean up suspended transactions in order of commit, this invariant guarantees that no suspended transaction will ever have a reference to a deleted transaction.

In Figure 3.9, we don't need to check if the reader has committed: in that case, the writer is the outgoing transaction, and is still running, so cannot have committed first.

```
markConflict(reader, writer):
    atomic begin:
        if writer has committed
          and writer.outConflict is not NULL
          and commit-time(writer.outConflict) <= commit-time(writer):
            abort(reader)
            return UNSAFE_ERROR

        if reader.outConflict is NULL:
            set reader.outConflict = writer
        else:
            set reader.outConflict = reader

        if writer.inConflict is NULL:
            set writer.inConflict = reader
        else:
            set writer.inConflict = writer
    atomic end
```

FIGURE 3.9: A modified `markConflict(reader, writer)` function that is more sensitive to false positives.

```
modified commit(T):
    atomic begin:
        if T.inConflict is not NULL and T.outConflict is not NULL
          and commit-time(T.outConflict) <= commit-time(T.inConflict):
            abort(T)
            return UNSAFE_ERROR
        mark T as committed

        if T.inConflict is not NULL and T.inConflict has committed:
            set T.inConflict = T
        if T.outConflict is not NULL and T.outConflict has committed:
            set T.outConflict = T
    atomic end

    existing SI code for commit(T), do not release SIREAD locks

    if T holds any SIREAD locks:
        suspend T, keeping SIREAD locks active
    if T was the oldest active transaction:
        clean up any suspended transactions older than active transactions
```

FIGURE 3.10: Enhanced version of `commit` that is more sensitive to false positives

3.7 Enhancements and optimizations

3.7.1 Abort early

For simplicity, in the description above, we did not show all the cases where we could check whether to abort T because both `T.inConflict` and `T.outConflict` are set; we have written the check once, in the `commit(T)` operation, and beyond that we only show the extra cases where an abort is done for a transaction that is not the pivot (because the pivot has already committed). In the prototype implementations, we actually abort an active transaction T as soon as any operation of T would cause both `T.inConflict` and `T.outConflict` to be set.

Likewise, in the `markConflict` function we use in our prototypes, conflicts are not recorded against transactions that have already aborted or that will abort due to both conflicts being detected.

3.7.2 Victim selection

When a conflict between two transactions leads to both conflict flags being set on either one, without loss of correctness either transaction could be aborted in order to break the cycle and ensure serializability. Our prototype implementation follows the algorithm as described above, and prefers to abort the pivot (the transaction with both incoming and outgoing edges) unless the pivot has already committed. If a cycle contains two pivots, whichever is detected first will be aborted.

However, for some workloads, it may be preferable to apply some other policy to the selection of which transaction to abort, analogous to deadlock detection policies (Agrawal *et al.*, 1987b). For example, aborting the younger of the two transactions may increase the proportion of complex transactions running to completion. Similarly, policies for victim selection can prioritize particular classes of transactions, reducing the probability of starvation. We intend to explore this idea in future work.

3.7.3 Upgrading SIREAD locks

A common pattern in database applications is a sequence of read-modify-write operations. That is, a data item `x` is read by a transaction T, then some modification to `x` is performed, and T updates the database with the new value. In the basic algorithm of Section 3.2, this would lead to T holding *both* an SIREAD lock and an EXCLUSIVE lock on x. Instead, the SIREAD lock can be upgraded to an EXCLUSIVE lock in this case without loss of correctness. That is, when the EXCLUSIVE lock is requested, the lock manager can replace the SIREAD lock with the EXCLUSIVE lock once any conflicts caused by the EXCLUSIVE lock are resolved.

Without this optimization, SIREAD locks would be kept after the transaction commits for all items that are modified, as well as all of the items that are simply read, leading to unnecessary conflicts and growth of the lock table. For simple transactions, this optimization can avoid keeping transactions suspended if they have no vulnerable reads, and hence no chance of an SI anomaly.

Note, however, that this optimization changes the conditions under which transactions must be suspended when they commit. Instead of checking only whether the transaction holds SIREAD locks, we now need to suspend a transaction if it holds SIREAD locks *or* if it has detected an outgoing conflict.

3.8 Mixing queries at SI with updates at Serializable SI

It is worth noting that read-only transactions, also called *queries*, can be run at unmodified SI and mixed in the same system with updates run at Serializable SI. In this configuration, there is no overhead of acquiring SIREAD locks for the queries, and no possibility of queries aborting due to the new “unsafe” error. The update transactions are serializable when considered separately from the queries, since write skew anomalies are prevented. Thus updates cannot introduce inconsistencies into the data as a result of interleaving and constraints implicit in the application will be maintained.

However, this configuration of transactions taken as a whole is *not* guaranteed to be serializable. As shown in (Fekete *et al.*, 2004), it possible for a query to read a database state that could not have existed in a serializable execution of the updates. An example from that work is analyzed in Example 3 of Section 2.5.1.

Regardless, we expect that the possibility of non-serializable queries will be acceptable in many applications and that this configuration will be popular in practice. It may be particularly attractive when making replicated snapshot isolation serializable, so that queries can be satisfied entirely locally at a single replicated node without requiring any communication between nodes.

3.9 Summary

This chapter described our new algorithm for serializable isolation, called Serializable Snapshot Isolation. We describe it as a set of modifications to an existing implementation of snapshot isolation in a DBMS. It modifies existing logic relating to finding the version of an item visible to a transaction during a read operation, and adds a new SIREAD lock mode that does not cause any blocking, but allows conflicts to be detected as part of the normal exclusive locking that is commonly used to by write operations in the implementation of snapshot isolation.

We presented the Serializable SI algorithm starting from a simple description, based on reads and writes of single data items, which also corresponds to the first prototype of the algorithm we implemented, in Berkeley DB. Our argument for the correctness of the algorithm is that it detects all rw-dependencies in an execution, and always aborts a transaction if two consecutive edges are found. To the simple algorithm, we added features to detect phantoms in systems using fine-grained locking, and described optimizations to reduce the rate of false positives flagged by Serializable SI, which in turn reduces the rate of transaction aborts.

CHAPTER 4

Implementation Details

The algorithm described in Chapter 3 was implemented in both Oracle Berkeley DB version 4.6.21 (Olson *et al.*, 1999) and the InnoDB transactional storage plugin version 1.0.1 for MySQL version 5.1.26. This chapter describes the two implementations, giving some details of how the algorithm was adapted to each engine.

4.1 Berkeley DB Introduction

The algorithm described in Section 3 was implemented in Oracle Berkeley DB version 4.6.21 (Olson *et al.*, 1999). Berkeley DB is an embedded database that supports SI as well as serializable isolation with S2PL. Locking and multi-version concurrency control are performed at the granularity of database pages. This can introduce unnecessary conflicts between concurrent transactions, but means that straightforward read and write locking is sufficient to prevent phantoms.

The architecture of Berkeley DB is described in Figure 4.1. It shows the subsystems that interact to implement transactional access methods: locking, logging and the buffer cache (also known as the memory pool in Berkeley DB).

4.2 Adding Snapshot Isolation to Berkeley DB

The author's implementation of Snapshot Isolation was incorporated into Berkeley DB release 4.5.20, in 2006. Prior to this version, Berkeley DB provided the standard locking isolation levels: serializable, read committed and read uncommitted. The design adopted for Berkeley DB was to manage versions of

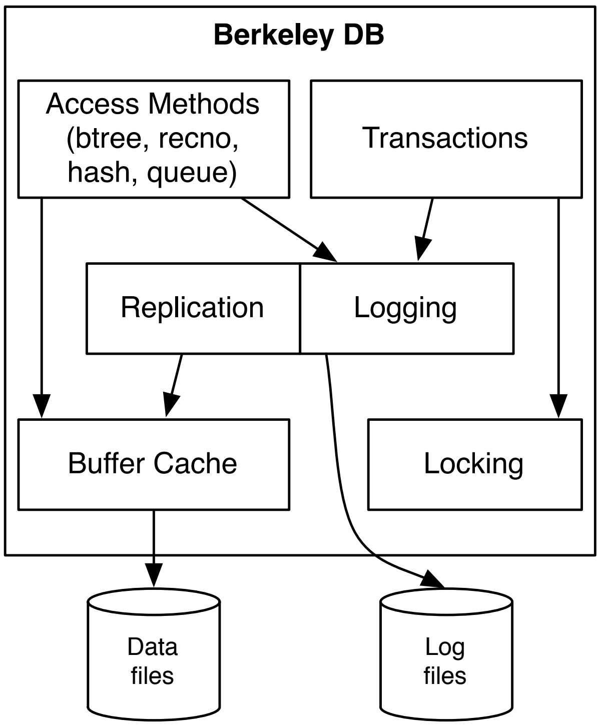


FIGURE 4.1: Architecture of Berkeley DB.

whole pages. That is, a transaction running at SI reads the versions of the database pages as they were when the transaction started, plus any changes it has made.

Implementing this page-versioning design primarily involved changes to the Berkeley DB cache manager, as described in Figure 4.2. Two new links were added to the buffer header of each page in cache: one to the creating transaction, and one to a buffer containing the previous version of the page. Further, transactions running at SI were modified to skip all requests for shared locks, and instead, when a page is requested, the cache manager returns the most recent version of that page committed before the requesting transaction started (or the current version, if the requesting transaction has already modified that page).

If a transaction requests a "dirty" page, which indicates an intent to modify the page, the cache manager checks whether a new version of the page has been committed since the requesting transaction started, and if so, indicates that an update conflict has occurred. This is how the First-Committer-Wins rule is implemented in Berkeley DB.

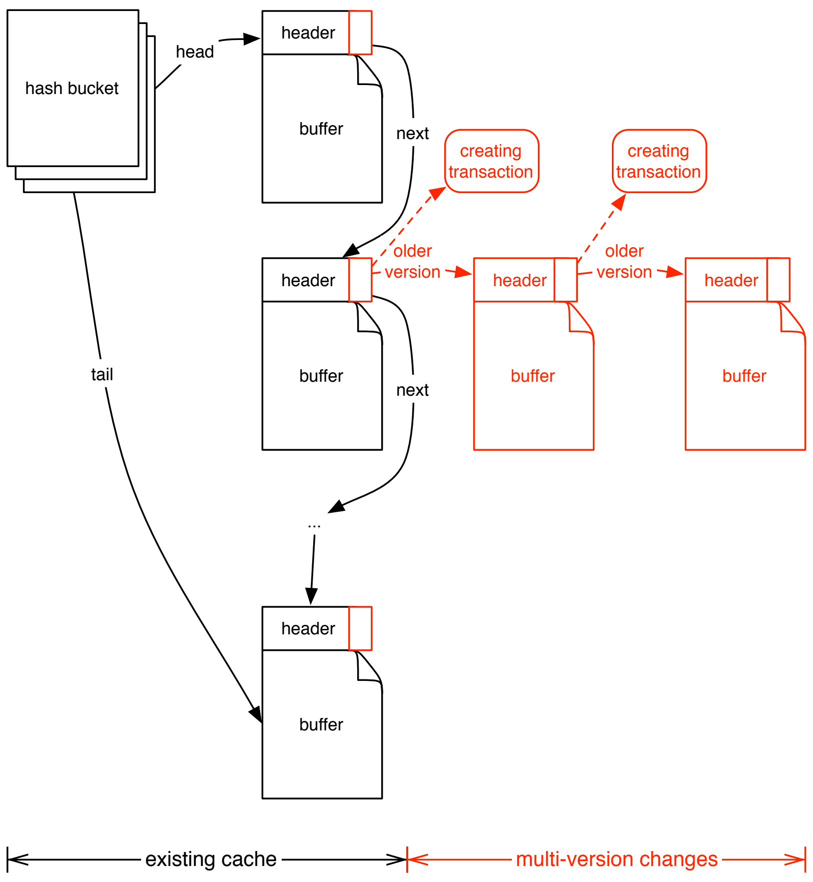


FIGURE 4.2: Changes to the Berkeley DB cache for MVCC.

This design has the following features:

- multiversion concurrency control can be turned on and off dynamically;
- there are no changes to the on-disk format of the database, and the technique is independent of the access method (B+Tree, hash table, etc.);
- the operation of checkpoints and recovery are unchanged: both can simply ignore versioning;
- keeping copies of whole pages has high overhead for small updates, both in terms of the CPU time to make copies and in the memory overhead in the cache; and
- it works best when the active set of versions fits in cache – otherwise, additional I/O is performed to manage old versions.

4.3 Adding Serializable SI to Berkeley DB

The basic algorithm of Section 3.2 is sufficient to guarantee serializable executions in Berkeley DB, because of the page-level locking and versioning design described in Section 4.2. That is what was implemented in our Berkeley DB prototype: the later enhancements from Sections 3.5-3.6 were not implemented. The more general optimizations in Section 3.7 *were* implemented in the Berkeley DB prototype: a transaction is aborted when an unsafe pattern is first detected (it is not deferred until commit), and SIREAD locks are upgraded when an EXCLUSIVE lock is acquired for the same data item by the same transaction.

The following changes were made to Berkeley DB in order to implement the Serializable SI algorithm:

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

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

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

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

4.3.1 Transaction Cleanup in Berkeley DB

In the Berkeley DB implementation, we did not aggressively clean up the suspended transactions and their locks during commit. Instead, we allowed Berkeley DB's transaction and lock tables to grow until some resource was exhausted (typically, the fixed amount of memory is fully allocated). Normally in this situation, Berkeley DB would indicate an error, but instead we first tried to clean up obsolete transactions and their locks.

The cleanup code is executed if the lock table becomes full (a complete sweep of all lock objects), and also whenever a request for a `WRITE` lock finds an out-of-date `SIREAD` lock. In this latter case, only the locks for one data item are scanned, in order to share the work of cleaning up old locks among all threads.

4.3.2 Analysis of the code changes

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

4.4 InnoDB Introduction

MySQL is a popular open source relational DBMS (MySQL AB, 2006). One of the architectural features of MySQL is that storage engines are pluggable: new storage engines with different concurrency control or I/O characteristics can be dynamically added to MySQL without changing the query processing “front end”.

The most stable and popular storage engine for transactional data management in MySQL is InnoDB (Innobase Oy, 2008). Our prototype implementation was created by modifying the InnoDB Plugin for MySQL version 1.0.1 using MySQL version 5.1.26.

InnoDB provides multiversion concurrency control by managing multiple versions of rows, and providing an older version of a row if required to produce a consistent read of a database. Unlike Berkeley DB, where old versions are kept until no running transaction could request them, in InnoDB, when an old version of a row is required, *undo* records from the log are used to reconstruct the version of the row that was current when the reading transaction began.

The default isolation level in InnoDB is called REPEATABLE READ, and at this isolation level, ordinary SQL `SELECT` statements use a consistent read view that does not require locking. The read view is a structure that is allocated when the transaction starts, and includes a list of active transactions (whose changes should not be visible), so that InnoDB can determine which transactions committed before the transaction began. This structure is used by InnoDB to ensure that the correct versions of rows are seen by the transaction.

The REPEATABLE READ isolation level has the interesting property that reads requiring locking, such as reads that are part of a SQL `SELECT FOR UPDATE` operation, always read the most recently committed values. Subsequent non-locking reads of the same item will again see the version from the consistent read view (assuming that it was not modified during the transaction). So within a single REPEATABLE READ transaction, the values returned are a mix of snapshot isolation and read committed semantics, depending on whether the operations acquire locks.

InnoDB also provides SERIALIZABLE isolation, which is conceptually equivalent to REPEATABLE READ but with all reads acquiring shared locks. That is, the most recently committed values are always read, and shared locks are held until the transaction commits. Further, InnoDB's SERIALIZABLE isolation level acquires shared locks on the gaps between rows during predicate reads and exclusive locks on the gaps during inserts and deletes in order to prevent phantoms as described in Section 2.5.2. In summary, InnoDB's SERIALIZABLE isolation is a traditional two-phase locking implementation that does not take advantage of multiple versions of data items that are available to reads at other isolation levels.

The existing InnoDB semantics are difficult to reason about if updates are performed at any isolation level other than SERIALIZABLE. However, the locking implementation of SERIALIZABLE isolation in InnoDB negates the concurrency benefits of multiversion concurrency control for reads that form part of an update transaction.

The InnoDB lock manager provides five lock modes: shared, exclusive, intention shared, intention exclusive and a special lock mode for auto-increment columns. The intention modes are used only at table-level granularity. Locks are represented efficiently in the lock manager as a bitmap associated with a page in the buffer manager, indicating which rows on the page are covered by the lock. In this way, locking a row generally only requires a few bits of memory in the lock manager.

Updates to the InnoDB lock table are protected by a global “kernel mutex”, which keeps the code simple but limits scalability on multi-core systems.

InnoDB holds a transaction’s locks until the point in transaction commit just *before* the write-ahead log is flushed. In other words, locks are released and changes become visible before log records are guaranteed to be on stable storage. This early release of locks creates a window of vulnerability in which a query could commit after seeing data values that would be lost after a crash. For short transactions that perform no other I/O, this optimization in InnoDB reduces the duration for which locks are held by 1-2 orders of magnitude. The reasoning given in the InnoDB source code for this design is that as the log is flushed in order, any subsequent update based on the not-yet-flushed data could not be flushed before the data it depends on. However, this reasoning does not apply to queries, which can read data that has not yet been made durable. There are various configuration parameters that can reduce the cost of flushing the log, including group commit and skipping the flush entirely, so releasing locks early is an interesting design choice.

In order to amplify the effects of locking and ensure that data visibility is fully transactional, we changed the order of operations during commit in InnoDB so that locks are not released until after the log has been flushed. Group commit was enabled (as it is by default in InnoDB) so that multiple transactions commits could be written to the log in a single flush, reducing the performance impact of this code change.

4.5 Adding Snapshot Isolation to InnoDB

As part of our work, we added an implementation of standard Snapshot Isolation (SI) to InnoDB. This involved using the existing REPEATABLE READ code to allow SI reads, including those that are part of an update, to use a consistent read view without acquiring shared locks. Records that are updated are locked with exclusive locks, as in REPEATABLE READ, and no gap locking is performed.

The first-committer-wins (FCW) rule was implemented by adding a check in the InnoDB code that determines whether old versions are required for a consistent read. If an SI transaction has an exclusive lock on the row and finds that a newer version has been committed since its read view was allocated, a new error `DB_UPDATE_CONFLICT` is returned, indicating that committing the transaction would violate the FCW rule.

Our implementation of SI has the useful property that the read view for an SI transaction is allocated only *after* any initial lock is acquired for the first statement in the transaction. For example, if the first statement is a SQL `UPDATE` or `SELECT FOR UPDATE`, an exclusive lock is acquired first, and only after the lock is granted is the transaction snapshot for reading determined . This simple implementation detail can dramatically reduce the number of aborts due to the FCW rule by reordering operations in a transaction or using `SELECT FOR UPDATE`. In particular, it ensures that transactions consisting of one statement that update a single data item will always read the latest version of the item, and thus will never abort due to the first-committer-wins rule.

For example, if two instances of a simple transaction that just increments a counter x are started at the same time, our implementation will ensure that one acquires the lock on x and commits before the second one chooses its snapshot. In other words, the second transaction will always read the results of the one that committed first, so there is no possibility of a update after the chosen snapshot, and thus no need to invoke the FCW rule. This issue did not arise in InnoDB previously because the existing REPEATABLE READ isolation level has no FCW rule, and in this simple example, since the data item is locked during the update, the most recent version is used anyway.

4.6 Adding Serializable SI to InnoDB

The first challenge in adding Serializable Snapshot Isolation to InnoDB was representing the SIREAD lock mode. InnoDB already has five lock modes, and packs the 5×5 lock conflict matrix into a 32-bit integer, so that checking for lock compatibility can be performed by bitwise integer operations. Adding a sixth lock mode for SIREADs would have increased the conflict matrix to 6×6, which cannot fit in the existing representation. Instead, we noted that SIREAD locks are only acquired for rows (or gaps between rows), but never at table granularity. InnoDB includes an “intention shared” lock mode (IS) that is only ever acquired for tables, never rows. The IS locks conflict only with EXCLUSIVE locks, which is exactly what we need for SIREAD locks. Our implementation uses IS locks on rows to represent SIREAD locks in order to avoid adding a new lock mode.

Phantoms are prevented in InnoDB at SERIALIZABLE isolation by gap locking as described in Section 2.5.2. We adapted InnoDB's gap locking to detect and prevent phantoms under Serializable SI as described in Section 3.5. This was quite straightforward to implement because SERIALIABLE transactions set a field in the transaction structure indicating that SELECT statements should acquire shared locks for rows and gaps. We use the same field to indicate that SELECT statements in Serializable SI transactions should acquire `IS` locks instead.

There are two places where read-write conflicts are detected: when reading an old version of a row, and when acquiring locks. In the lock manager, when deciding whether a row lock conflict should cause the transaction to block, we added code to check whether either of the locks involved in the conflict have `IS` mode. If so, we mark a read-write conflict, where the transaction making the `IS` lock request is the reader. In the code to retrieve the version of a row required by a consistent read, we mark a read-write conflict with the owner of any newer version.

In marking a read-write conflict, we first check that the two transactions do, in fact, overlap and that neither is already destined to abort. We then check whether this conflict will cause one of the transactions to abort and if so, only abort that transaction. Lastly, if neither transaction will abort as a result of the conflict, we mark the incoming and outgoing flags of the two transaction objects as described in Section 3.

When a Serializable SI transaction attempts to commit, we first check whether both incoming and outgoing flags are set, and if so, we roll back the transaction instead and return an error `DB_UNSAFE_-TRANSACTION`. We implemented the optimizations described in Section 3.6, where the incoming and outgoing edges are represented by references to two distinct transaction objects, and we roll back only when T_{out} commits before T_{in}.

If any SIREAD locks are held at commit time, the transaction structure is suspended rather than immediately recycled. In the InnoDB implementation, the transaction structure will be reused by the next operation, so a new "dummy" transaction object is allocated and initialized with the committing transaction's ID, SIREAD locks and conflicts, then the dummy transaction is added to a list of committed Serializable SI transactions that overlap with some running transaction.

4.6.1 Transaction Cleanup in InnoDB

If the committing transaction has the oldest read view of all active transactions, the list of committed Serializable SI transactions is scanned and any dummy transactions that no longer overlap with an active transaction are cleaned up and their SIREAD locks released. This eager cleanup of suspended transactions maintains a tight window of active transactions and minimizes the number of additional locks in the lock manager.

The prototype implementation for Berkeley DB, described in (Cahill *et al.*, 2008), used a reference count in the transaction object to perform garbage collection when a suspended transaction was no longer overlapping with any running transaction. Our technique has the advantage that aggressive cleanup ensure that only the minimal number of transaction or lock objects will be kept active.

4.6.2 Analysis of the code changes

The changes to InnoDB to implement both SI and Serializable SI were modest: around 250 lines of code were modified and 450 lines of code were inserted. There are over 180,000 lines of code in total in InnoDB, so this represents less than 0.4% of the InnoDB source code More than half of our changes were in the lock manager (modifying 6% of InnoDB's locking code), to allow IS locks on rows and to clean up locks when suspended Serializable SI transactions are freed. The remaining changes were in the InnoDB transaction code and row visibility checking, including code to detect when a read ignores newer versions of a row. In addition, the MySQL front-end query processing code was modified slightly to support setting the new isolation levels and to propagate the new error codes back to clients.

4.7 InnoDB Testing

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

T_1: b_1 r_1(x) c_1
T_2: b_2 r_2(y) w_2(x) c_2
T_3: b_3 w_3(y) c_3

The implementation was tested by writing a program that generated test cases from such a set of transactions, where each test case was a different interleaving of the transactions. Each interleaving was executed by starting a separate client connection for each transaction and synchronizing between the client connections in order to generate the specified interleaving on the server.

During development, this testing regime was invaluable in establishing that conflicts were detected in all relevant code paths through the DBMS. For example, there are different code paths for queries that involve scans depending on whether a relevant index is available. By testing different sets of transactions against various configurations of the database, we gained confidence that we had modified InnoDB in all relevant places.

In all cases where the transactions were executed concurrently against the final version of the code (evaluated below), one of the transactions aborted with the new "unsafe" error return. These results were manually checked to verify that no non-serializable executions were permitted although all interleavings committed without error at SI.

4.8 Generalizing to other database engines

A key issue for an implementation is how to keep the SIREAD locks and transaction information after the transaction commits, and then clean up the transactions and locks as efficiently as possible. In InnoDB, it is reasonable for transaction and locks to remain in the system for some time after commit, and we describe above (in Section 4.6) the clean up procedure during transaction commit. This is also true for the other open source DBMS engines that we are familiar with, including Berkeley DB and PostgreSQL.

In a different system where it is not feasible to keep transaction objects suspended after commit, a table would need to be maintained containing the following information: for each transaction ID, the begin and commit timestamps together with an `inConflict` flag and an `outConflict` flag. For example:

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

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

Another issue is how to store SIREAD locks if the existing lock manager cannot maintain them, for example if a new lock mode cannot be added or the lock manager cannot be modified to keep SIREAD locks after the owning transaction commits. In this case, a table could be constructed to represent SIREAD locks. Since SIREAD locks never cause blocking, the table would only be used to determine whether a read-write conflict has occurred. Rows in the SIREAD lock table become obsolete and can be deleted when the owning transaction becomes obsolete, that is, when all overlapping transactions have completed.

CHAPTER 5

Benchmarks

Standard database benchmarks such as TPC-C (Transaction Processing Performance Council, 2005) are carefully designed to exercise a range of features of a system. However, we cannot use TPC-C directly to compare different ways of making applications serializable, since TPC-C itself generates only serializable executions on SI-based platforms, as described in Section 2.8.1. To evaluate the effects of making SI serializable, we need a benchmark that is not already serializable under SI.

In this thesis we have used three different benchmarks to evaluate our prototypes. Firstly, the `SmallBank` benchmark from (Alomari *et al.*, 2008a) was adapted to run on Berkeley DB. Secondly, a new microbenchmark that we call `sibench` was designed and implemented for InnoDB to highlight the performance of its concurrency control algorithms including the SI and Serializable SI levels that we added. Finally, we modified TPC-C to introduce an anomaly when run under SI, and we call the resulting benchmark TPC-C++. We evaluated our InnoDB prototype with TPC-C++ to get a clearer picture of the impact of concurrency control algorithms on performance of the overall system.

This chapter describes the three benchmarks. Section 5.1 describes how we adapted `SmallBank` to run on Berkeley DB, then Section 5.2 describes our new `sibench` microbenchmark and Section 5.3 describes the TPC-C++ modifications to TPC-C.

5.1 Adapting `SmallBank` for Berkeley DB

Recall from Section 2.8.2 that the `SmallBank` benchmark models a simple banking application involving checking and savings accounts, with transaction types for balance (Bal), deposit-checking (DC), withdraw-from-checking (WC), transfer-to-savings (TS) and amalgamate (Amg) operations. Each of the transaction types involves a small number of simple read and update operations to two records per customer: a savings account balance, and a checking account balance. The static dependency graph for `SmallBank` was given in Figure 2.9 of Section 2.8.4, and it can be seen by inspection that there is a dangerous structure Balance → WriteCheck → TransactSavings → Balance, so the transaction WriteCheck is a pivot.

The original `SmallBank` paper, (Alomari *et al.*, 2008a), implemented the benchmark in a relational DBMS. Here we describe how we implemented `SmallBank` for Berkeley DB, which provides a general purpose storage manager, but no query processing or SQL layer. Luckily, the `SmallBank` schema is straightforward to implement on Berkeley DB. It consists of three tables: Account(Name, CustomerID), Saving(CustomerID, Balance), Checking(CustomerID, Balance). The `Account` table represents the customers; it maps names to customer IDs. The CustomerID is a primary key for both `Saving` and `Checking` tables. The `Balance` columns are numeric types, representing the balance in the corresponding account for one customer. Since each table has only two columns, a primary key and a value, that maps directly onto the key/data pair API provided by Berkeley DB.

5.1.1 Transaction Mix

The `SmallBank` benchmark runs instances of five transaction programs, chosen randomly with equal probabilities (20%). All transactions start by looking up the customer's name in the `Account` table to retrieve the CustomerID. Here, we translate the SQL descriptions of the transaction programs given in (Alomari *et al.*, 2008a) into read and write operations that can be processed by Berkeley DB. This assumes that there are Berkeley DB handles called `account`, `savings` and `checking` referring to the three tables in the schema.

Balance or Bal(N): calculates a total balance for a customer:

```
ID = account.get(N)
return savings.get(ID) + checking.get(ID)
```

DepositChecking, or DC(N, V): makes a deposit into the checking account of a customer:

```
ID = account.get(N)
checking.put(checking.get(ID) + V)
```

TransactSaving, or TS(N, V): makes a deposit or withdrawal on the savings account:

```
ID = account.get(N)
savings.put(savings.get(ID) + V)
```

Amalgamate, or Amg(N1, N2): moves all the funds from one customer to another:

```
ID1 = account.get(N1)
ID2 = account.get(N2)
checking.put(ID2, savings.get(ID1) + checking.get(ID1))
savings.put(ID1, 0)
checking.put(ID1, 0)
```

WriteCheck, or WC(N, V): writes a check against an account, checking for overdraft:

```
ID = account.get(N)
if savings.get(ID) + checking.get(ID) < V:
    checking.put(ID, checking.get(ID) - V - 1)
else:
    checking.put(ID, checking.get(ID) - V)
```

Once translated into this form, the transaction mix can be directly implemented on Berkeley DB.

5.2 `sibench`: a new microbenchmark for Snapshot Isolation

To explore the performance impact of avoiding blocking for read-write conflicts, we first illustrate the trade-off between bottlenecks including CPU time, lock contention and write-ahead log flushes during commits. We designed and implemented a very simple synthetic microbenchmark called `sibench` based on a read-write conflict between an query transaction and an update transaction. The benchmark uses a single table called `sibench` with two non-null integer columns: `id`, which is a primary key, and `value`.

The query transaction returns the `id` with the smallest `value` in the table. This transaction was designed so that the DBMS must scan all rows in the table, and so that there is some CPU cost associated with processing the rows, but the final result is small, so that communication cost is constant, and in particular, independent of the number of rows in the table. The query transaction is implemented by a single line of SQL:

```
SELECT id FROM sitest ORDER BY value ASC LIMIT 1
```

The update transaction simply increments the `value` of a row chosen from a uniform random distribution of the identifiers. It is also implemented by a single line of SQL:

```
UPDATE sitest SET value = value + 1 WHERE id = :id
```

Note that as a consequence of the optimization described in Section 4.5, where the read snapshot is not determined until after any lock required for the first statement has been acquired, conflicts between concurrent updates in `sibench` will result in blocking but not aborts for all three concurrency control algorithms.

Since there is only a single edge in the static dependency graph, there is no possibility of deadlocks or write skew, and so no transactions are expected to roll back during the execution of the benchmark. We verified during the evaluation that no transactions deadlocked or experienced write skew.

We concentrate on the primary parameter to `sibench`: I, the number of rows in the table. The value of I controls the number of rows read by the query, and hence the number of lock manager requests when the query runs at S2PL or Serializable SI. However, the query also involves a sort, so increasing I also increases the amount of CPU required to execute the benchmark. Further, for small values of I, the rate of write-write conflicts between simultaneous updates is high, so increasing I reduces the proportion of write-write versus read-write conflicts.

5.3 The TPC-C++ benchmark

When proposing changes to the core of a database system, the natural choice for an evaluation is to use one of the standard benchmarks of the Transaction Processing Council, such as the TPC-C benchmark (Transaction Processing Performance Council, 2005) introduced in Section 2.8.1. TPC-C models the database requirements of a business selling items from a number of warehouses to customers in some geographic districts associated with each warehouse.

TPC-C uses a mix of five concurrent transaction types of varying complexity. These transactions include creating and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. TPC-C exercises most of the components at the core of a DBMS storage engine, but it has long been known that TPC-C is serializable when run at snapshot isolation, and this was formally proved in (Fekete *et al.*, 2005).

Here we describe how we modified the standard TPC-C in order to measure the cost of providing serializable execution for arbitrary applications. Our main change is to add a new transaction type, called *Credit Check*, which we describe in Section 5.3.2.

5.3.1 Simplifications from full TPC-C

Our use of TPC-C as a basis for the evaluation is intended only to provide a widely-understood database workload. We do not attempt to get certifiable TPC-C results. In particular, we deviate from the TPC-C specification as follows:

- ignore terminal emulation entirely: only the database transactions are considered;
- omit the storage of historical data, as it has little bearing on concurrency control;
- leave out waits between transactions to increase the load on the database for a given number of clients;
- report total transaction throughput (in units of transactions per second, or *TPS*), rather than only the number of New Order transactions that are included in tpmC;
- avoid unnecessary conflicts without the complexity of partitioning the tables by assuming that the constant `w_tax` field in the Warehouse table can be cached by clients;
- in some cases, omit the updates to the year-to-date field in the Warehouse and District tables, which generate write-write conflicts every pair of Payment transactions for the same warehouse, or between Payment and Delivery tr. This value can be calculated when required with an aggregation query over the districts rather than having all Payment transaction update the same field.

```
EXEC SQL SELECT c_balance, c_credit_lim
           INTO :c_balance, :c_credit_lim
           FROM Customer
          WHERE c_id = :c_id AND c_d_id = :d_id AND c_w_id = :w_id

EXEC SQL SELECT SUM(ol_amount) INTO :neworder_balance
           FROM OrderLine, Orders, NewOrder
          WHERE ol_o_id = o_id AND ol_d_id = :d_id
            AND ol_w_id = :w_id AND o_d_id = :d_id
            AND o_w_id = :w_id AND o_c_id = :c_id
            AND no_o_id = o_id AND no_d_id = :d_id
            AND no_w_id = :w_id

if (c_balance + neworder_balance > c_credit_lim)
    c_credit = "BC";
else
    c_credit = "GC";

EXEC SQL UPDATE Customer SET c_credit = :c_credit
          WHERE c_id = :c_id AND c_d_id = :d_id AND c_w_id = :w_id

EXEC SQL COMMIT WORK;
```

FIGURE 5.1: Sample code for the Credit Check Transaction as SQL embedded in C.

5.3.2 The Credit Check Transaction

TPC-C++ keeps the same schema as TPC-C (see Figure 2.7 in Section 2.8.1), but adds a new transaction type called *Credit Check*. The purpose of the Credit Check transaction is to check whether a customer has a total balance of unpaid orders exceeding their credit limit, and to update the customer's credit status accordingly. Such a transaction could execute either as part of a batch job, updating the status for all customers meeting some criteria, or as part of a workflow during an interaction between the customer and the company.

A customer's total outstanding balance combines both the delivered, unpaid orders (tracked in the `c_balance` field of the Customer table) and total value of new orders that are not yet delivered. Sample code for the Credit Check transaction is given in Figure 5.1 in the style of appendix A of the TPC-C specification (Transaction Processing Performance Council, 2005).

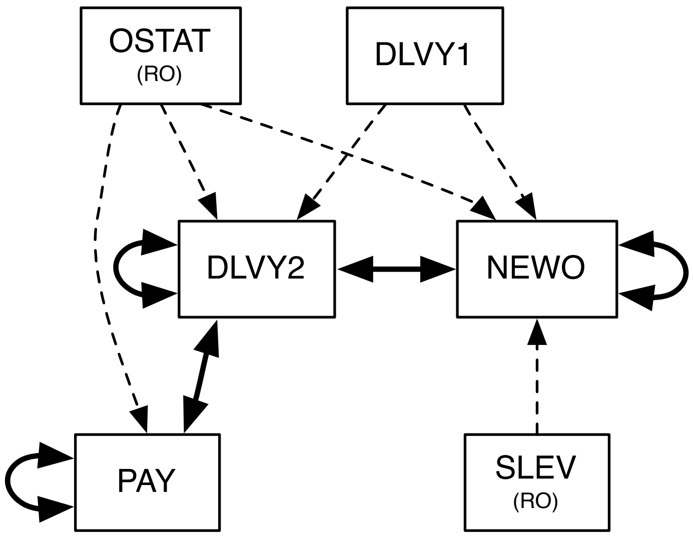


FIGURE 5.2: Static Dependency Graph for standard TPC-C (repeated) from (Fekete *et al.*, 2005).

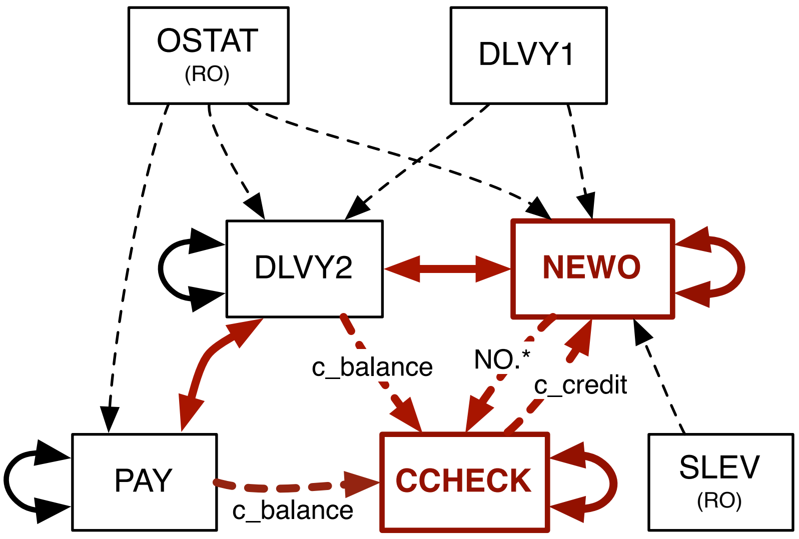


FIGURE 5.3: Static dependency graph for TPC-C++.

5.3.3 TPC-C++ Anomalies

As shown in Figure 5.3, although fairly simple, the Credit Check transaction complicates the static dependency graph significantly. A Credit Check transaction reads the `NewOrder` table, into which the New Order transaction inserts, and also reads the `c_balance` field of a customer record, which is updated by both the Delivery and Payment transactions. It updates the customer's `c_credit` field, which is read by the New Order transaction. If two Credit Check transactions were run concurrently for the same customer, each would attempt to update the same customer record, hence the ww-conflict with itself.

In a DBMS that performs locking and versioning at row-level granularity with the standard physical layout of the Customer table, the conflicts between the Credit Check transaction and either Delivery or Payment would in fact be write-write conflicts, even though the transactions update different fields in the Customer table. It is common practice in such situations to partition the table in order to improve concurrency, and the TPC-C specification explicitly permits partitioning and discusses some possible partitioning schemes for the Customer table. If `c_balance` and `c_credit` were stored in different partitions, the conflicts would be as shown even in a DBMS with row-level locking and versioning.

In the modified SDG of Figure 5.3, there are two pivots: New Order and Credit Check. There are several cycles: the simplest is from Credit Check to New Order and back again. This simple cycle corresponds to a credit check that runs concurrently with the placement of a new order. As a result, the Credit Check transaction does not see the effect of the New Order, and the New Order transaction does not see the result of the Credit Check. This is a straightforward write skew.

More complex cycles are also possible. For example, the cycle from Credit Check through New Order, Delivery and Payment back to Credit Check. That cycle can give rise to the following example of a non-serializable execution.

Example 5. *Assume that the customer is informed during a New Order transaction if they have a bad credit rating. The status is displayed on the terminal, so an operator might indicate to the customer that the order will be delayed or that additional charges will apply. Further, assume that some customer has a credit limit of \$1000 and that* `c_balance` *is initially \$900.*

The following interleaving of transactions is possible if TPC-C++ is executed at SI, where some of the transactions are initiated by the customer and a Credit Check is started simultaneously by a background process at the company:

Customer		*System*
Operation	*Unpaid total*	
New Order ($200)	*$1100*	
		begin Credit Check
Payment ($500)	*$600*	
New Order ($100)	*$700*	*calculate total balance = $1100*
		⇒ `c_credit = "BC"`
		commit
New Order ($150)	*$850 + bad credit*	

The customer's first New Order transaction exceeds their credit limit. The next transaction is a Payment, which reduces the unpaid total back under the limit. If the next order were marked as "bad credit", the customer would recognize that their overdraft had been detected before the payment was processed.

However, in this execution, a Credit Check transaction begins concurrently with the Payment transaction and commits concurrently with the following New Order transaction. If the credit check runs at Snapshot Isolation, it will calculate an unpaid total of $1100, as it does not see any changes from either of the concurrent transactions. It therefore commits a "bad credit" status, but the customer does not see the result until after *successfully placing an order with a "good credit" status.*

There is no way that this could have happened in a serializable execution. If the Credit Check had been serialized before the Payment, the $100 order would have been marked as bad credit. If the Credit Check were serialized after the payment, or after the $100 order, it would have calculated an unpaid balance below the $1000 credit limit and would have set `c_credit = "GC"`.

5.3.4 TPC-C++ Transaction Mix

In most of our evaluation, we keep the relative mix of transaction types from TPC-C and execute Credit Check transactions with the same probability as Delivery transactions. The numbers in the original specification were chosen so that a single random integer can be generated and used to select between transaction types.

Adding Credit Check transactions to the mix with the same frequency as Delivery transactions gives the following adjusted proportions: approximately 41% New Order transactions, at least 41% Payment transactions, 4% Credit Check transactions, 4% Delivery transactions, 4% Order Status transactions, and 4% Stock Level transactions.

The intent of this mix is primarily to stay close to the spirit of TPC-C. In addition, our goal is not to skew the benchmark such that anomalies overwhelm the normal workings of TPC-C when run at SI. We want to introduce the possibility of anomalies, and measure the cost of preventing them, not to create a mix of transactions where anomalies are occurring all the time.

5.3.5 Stock Level Mix

We also want to configure TPC-C++ to highlight the effects of concurrency control. Multiversion concurrency control provides most benefit in situations where read-write conflicts are the primary bottleneck, which correspond to dashed lines in the static dependency graph in Figure 5.3. Here we focus on the edge between the New Order transaction and the Stock Level transaction, which is due to the Stock Level transaction reading stock levels that are updated by the New Order transaction as orders are entered into the database.

We define the *Stock Level Mix* of TPC-C++ as a mix consisting of just the New Order and Stock Level transactions, where 10 Stock Level transactions are executed for each New Order transaction. The Stock Level transaction examines the 20 most recent orders, and there are an average of 10 order lines per order. The New Order transaction modifies approximately 20 rows on average. The result is that this mix is skewed, with approximately 100 rows read for each row updated. Thus there are some similarities with the `sibench` configuration where the table contains 100 rows, although the transactions, schema and data volumes are much more substantial with TPC-C++.

5.3.6 Data Scaling

TPC-C++ does not make any changes to the TPC-C schema, described in Section 2.8.1. The total volume of data generated depends on a single parameter, W, the number of warehouses. The cardinality of each table in the schema is defined a some multiple of W. In our evaluation, we present measurements from two cases, $W = 1$ and $W = 10$.

Varying W, the number of warehouses, has two effects:

(1) it changes the data volume, which determines whether the data fits in cache, or alternatively, whether read and write operations have to wait for disk I/O; and
(2) it varies the contention between threads, because the `w_ytd` column of the Warehouse table is updated by every Payment transaction for that warehouse.

To separate these two effects, we also measured the workload against a data set where the scaling factors of some of the tables was reduced, which we call the *tiny scale*. This allowed us to increase the number of warehouses (decreasing contention) while keeping data within memory so that operations were not delayed by disk I/O. In the tiny scale experiments, the number of customers is divided by 30 (there are only 100 customers per district), and the number of items is divided by 100 (there are 1000 rows in the Item table).

This gives the following approximate data volumes for each combination of scaling parameters:

	$W = 1$	$W = 10$
standard scale	120MB	1.2GB
tiny scale	2MB	20MB

5.4 Summary

This chapter has introduced the three benchmarks we have used to evaluate our prototype implementations of Serializable SI. We adapted the `SmallBank` benchmark from (Alomari *et al.*, 2008a) to run on Berkeley DB, which provides a lower-level, non-relational programming interface. We described a new microbenchmark that we call `sibench` consisting of a simple pair of transaction types, a query and an update, designed to highlight the effect of non-blocking reads in SI and Serializable SI. Lastly, we presented TPC-C++, a modification to TPC-C that introduces a new transaction type called Credit

Check, which creates the potential for non-serializable executions if TPC-C++ is run with SI. TPC-C++ enables the measurement of the cost of ensuring serializable executions within the framework of a comprehensive transaction processing workload.

CHAPTER 6

Performance Evaluation

In this chapter, we present the performance measurements of our two prototype implementations of Serializable SI: one in Berkeley DB, evaluated with the `SmallBank` benchmark, and another in InnoDB, evaluated with the `sibench` microbenchmark and with TPC-C++, our new modification to TPC-C to make it sensitive to snapshot isolation anomalies.

We compare each platform's SI and S2PL implementations with our prototype implementations of Serializable SI. Results for `SmallBank` and TPC-C++ at S2PL and Serializable SI reflect the throughput of executions that maintain consistency, so while the throughput of SI is higher in some cases, those executions may have left the database in an inconsistent state.

Our goals are to:

(1) Illuminate the fundamental tradeoff: how much does it cost to guarantee serializable executions regardless of the semantics of the application?
(2) Explore the parameter space and determine which factors determine performance in various regions. In particular, we would like to discover the conditions under which concurrency control is the primary factor determining performance. How do the various concurrency control implementations behave in extreme conditions? When are they I/O bound, CPU bound, constrained by logical locking, or by the scalability of the DBMS internals?
(3) Demonstrate that the prototype implementations operate correctly (providing serializable isolation) under concurrent loads. We performed manual analysis of the tables resulting from running each benchmark under Serializable SI to verify that the database state had not been corrupted.

(4) Provide some insight and recommendations regarding when Serializable SI is effective and when it is preferable to S2PL.

This chapter is structured as follows. In Section 6.1, we measure the performance of the Berkeley DB prototype using the `SmallBank` benchmark. Next, in Section 6.2 we begin the evaluation of our more complex InnoDB prototype using the simple `sibench` microbenchmark to highlight the differences between the available isolation levels and explore the behavior under extreme conditions. Lastly, we evaluate the impact of our InnoDB prototype on the performance of the DBMS as a whole using TPC-C++, which is based on the industry-standard TPC-C.

6.1 Berkeley DB: `SmallBank` evaluation

Section 5.1 describes how we adapted the `SmallBank` benchmark from (Alomari *et al.*, 2008a) to run on Berkeley DB, which provides a low-level, primary key interface to accessing and modifying data. We used a tool called `db_perf`, designed and implemented by Sullivan (Sullivan, 2003), that executes and measures a workload defined in a configuration file against Berkeley DB. The `db_perf` tool can configure the many options Berkeley DB makes available for performance tuning (most of which were left at their default values), and has built-in support for warming the cache before it starts measuring.

We extracted the access patterns of the `SmallBank` benchmark, and created a configuration file for `db_perf` that encodes the distribution of key and data items along with the five transaction types, based on the code in Section 5.1.1.

We configured `db_perf` to execute the five transaction types in a uniform random mix. There was no sleep or think time: each thread executed transactions continually, as quickly as Berkeley DB could process them. In all cases, the data volume was chosen so that all data fitted in cache. The only I/O during the benchmark was the write and flush operations to put log records into the database logs when each transaction committed. We measured the built-in S2PL and SI isolation levels in Berkeley DB with our implementation of the new Serializable SI algorithm.

The measurements were controlled by parameters that configured:

- the isolation level,
- the number of threads (the multi-programming level, or MPL),
- whether the transaction logs were flushed to disk during each commit, which is the difference between executing transactions in approximately 100μs without flushes, or around 10ms when waiting for the disk,
- the data volume (the number and size of records, together with the database page size, which together determine the number of pages in the database),
- the complexity of the transactions, controlled a number N of SmallBank operations to perform in each database transaction. Initially, $N = 1$, but in later measurements, we explore the case where $N = 10$, where each transaction is doing ten times more work.

We used Berkeley DB's Btree access method for all tables.

6.1.1 Berkeley DB Evaluation Setup

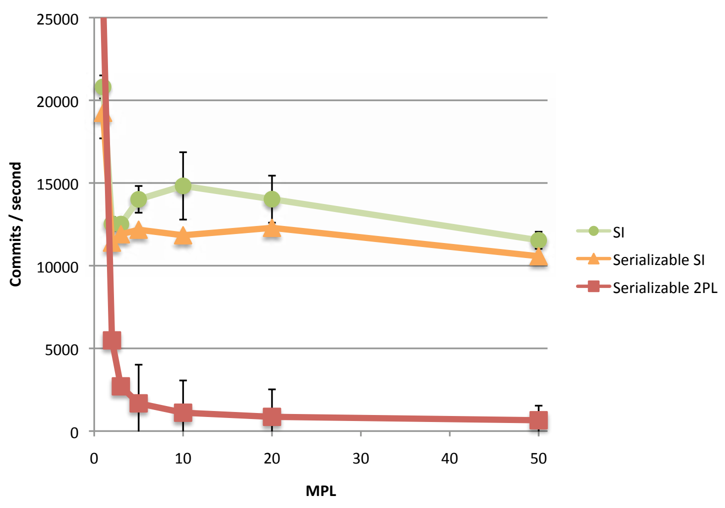
The experiments were run on an AMD Athlon64 3200+ CPU with 1GB RAM running OpenSUSE Linux 10.2 with kernel version 2.6.18, glibc version 2.5 and GCC version 4.1.2. A single set of Berkeley DB binaries were used for all measurements. All data was stored using the XFS filesystem on a set of four Western Digital Caviar SE 200 GB SATA hard disks using software RAID5.

We present our results as pairs of graphs. The first shows throughput in commits per second on the vertical axis versus multi-programming level (MPL) on the horizontal axis. The second graph for each set of experiments shows the number of transactions that were aborted at each isolation level, grouped by the type of error that caused the abort:

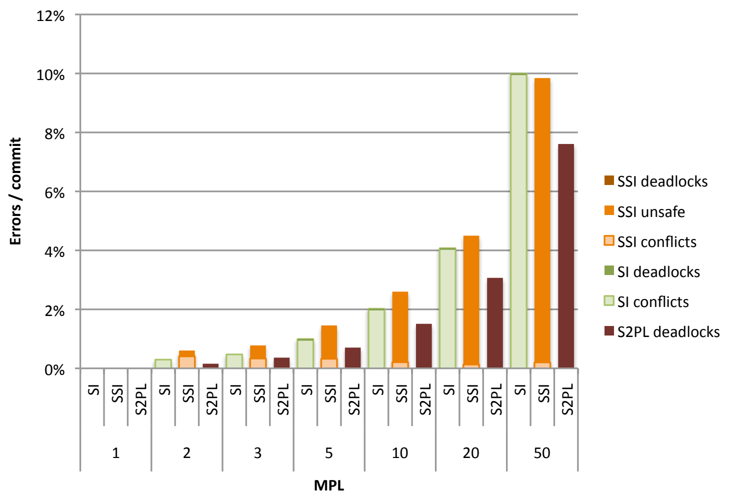
deadlocks: traditional cycles among locks of concurrent transactions

conflicts: errors are concurrent updates to a common data item (the "First-Committer-Wins" rule).

unsafe: errors are the new aborts introduced by potential cycles in the conflict graph detected by Serializable SI.



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



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

FIGURE 6.1: Berkeley DB `SmallBank` results without flushing the log during commit.

All graphs include 95% confidence intervals (assuming a normal distribution of results), although in many cases the confidence intervals are too small to be visible.

6.1.2 Short Transactions Workload

Results are given in Figure 6.1 for measurements where system was configured so that commit operations do not wait for a physical disk write before completing. This configuration is common on high-end storage systems with redundant power sources or in solid state drives based on Flash memory. A small data size was configured here to model moderate to high contention: the savings and checking tables both consisted of approximately 100 leaf pages. In other words, since Berkeley DB locks whole pages for each operation, there is approximately a 1% probability of conflict between each pair of concurrent transactions. In fact, the real probability is higher, due to rw-conflicts on internal pages in the B+Tree that are updated as part of a split or merge.

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

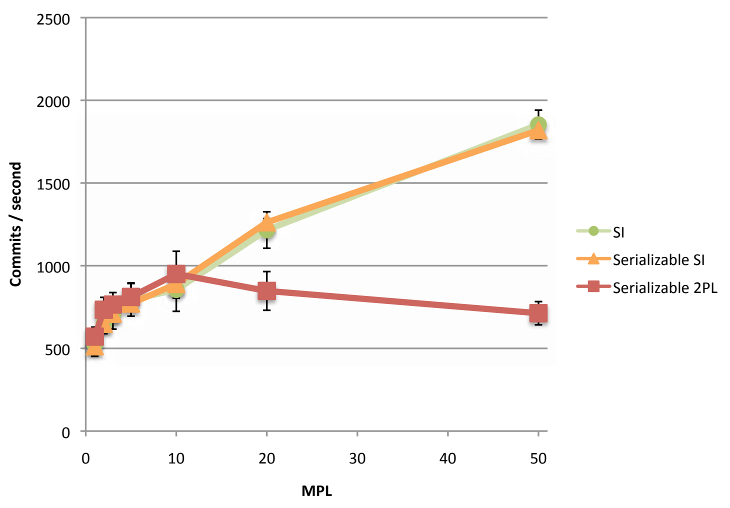
It can be seen that in this configuration, in Figure 6.1, Serializable SI performs significantly better that S2PL (by a factor of 10 at MPL 20). This is due to blocking in S2PL between read and write operations, and also because the aborts at S2PL are caused by deadlocks, and deadlock detection introduces further delays.

Figure 6.1(b) shows that Serializable SI has a slightly higher total rate of aborts than SI in most cases, and approximately 25% higher than S2PL. Interestingly, a high proportion of errors (almost all) are reported as "unsafe" errors rather than update conflicts. This is because in this benchmark, several of the transactions execute a read operation followed by a write. If a conflict occurs in between those two operations, a rw-conflict will be detected, leading to an unsafe error and an early abort. If the transaction had continued instead of aborting early, its write operation would trigger the FCW rule and a "conflict" error instead.

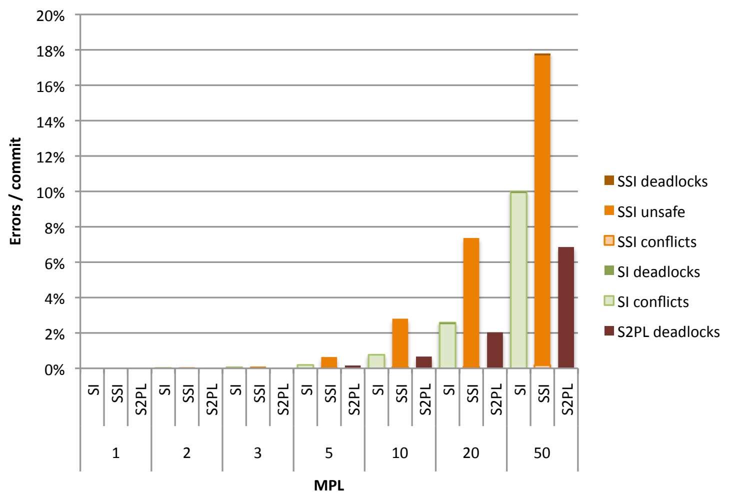
6.1.3 Long Transactions Workload

Figure 6.2 shows results for the same set of experiments, changing only the commit operations to wait for a physical write to disk. This significantly increased the duration of transactions, increasing response times by at least an order of magnitude to around 10ms. In this configuration, I/O rather than the CPU is the bottleneck at MPL 1, and throughput increases as MPL is increased for all isolation levels, because increasing numbers of transactions can be committed for each I/O operation due to group commit.

Up to MPL 10, there is little to separate the three concurrency control algorithms, but at MPL 20 the rate of deadlocks at S2PL begins to have an impact on its throughput. Deadlocks are more expensive



(a) Throughput

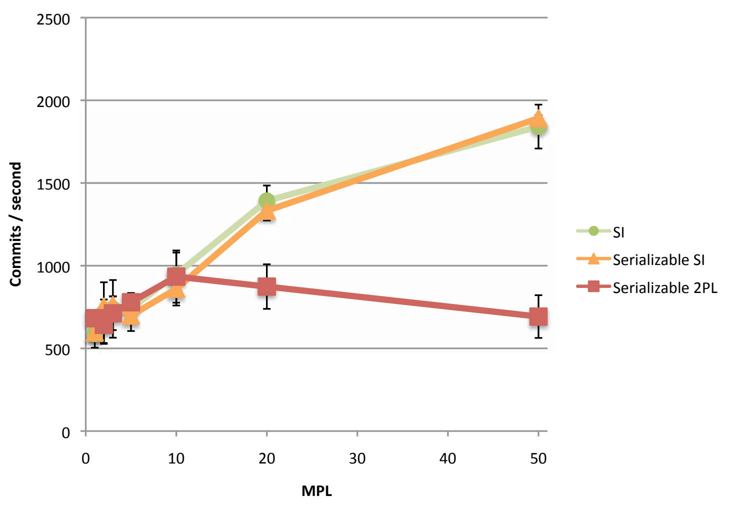


(b) Error rates

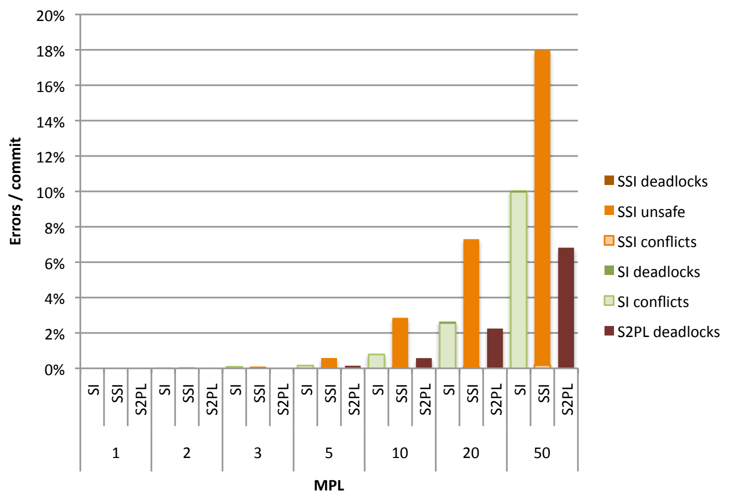
FIGURE 6.2: Berkeley DB `SmallBank` results when the log is flushed during commit.

to detect and resolve than SI conflicts or Serializable SI unsafe errors, because the built-in behavior of `db_perf` (which we did not change) is to detect deadlocks in a separate thread that runs only twice per second. The delay between deadlock detection contributes to the poor throughput of S2PL as some number of threads are blocked during that period.

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



(a) Throughput



(b) Error rates

FIGURE 6.3: Berkeley DB `SmallBank` results when the log is flushed during commit and each transaction performs 10 times more operations.

6.1.4 More Complex Transaction Workload

In the workloads we have presented so far, each transaction performs only 2-3 read operations followed by 1-2 writes. Next, we explore what happens if the transactions are made more complex. Instead of performing a single banking transfer or balance check per transaction, Figure 6.3 shows the effect of executing ten operations in each database transaction. The operations are chosen randomly from the SmallBank transaction mix.

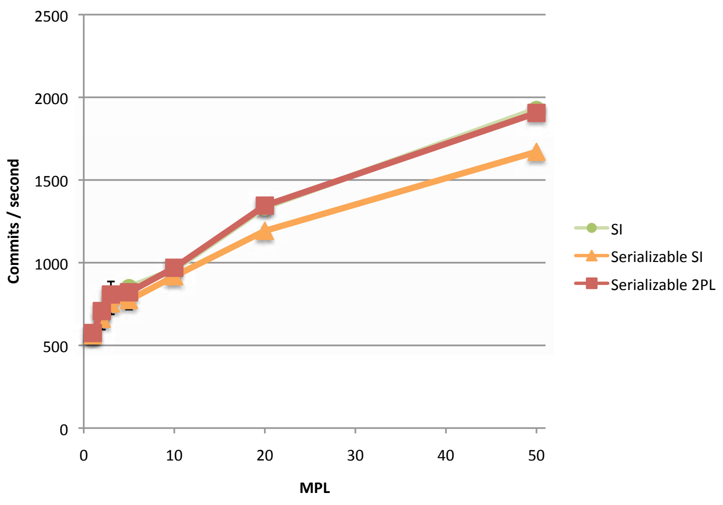
This workload is primarily I/O bound: transactions spend most of their execution time waiting for the log to be flushed so that their commit can be processed. More complex transactions write more log records, but still usually only need to flush the log once, when they commit. For this reason, the results in Figure 6.3 are very similar to those of Figure 6.2, even though ten times more work is being completed by each transaction.

6.1.5 Overhead Evaluation

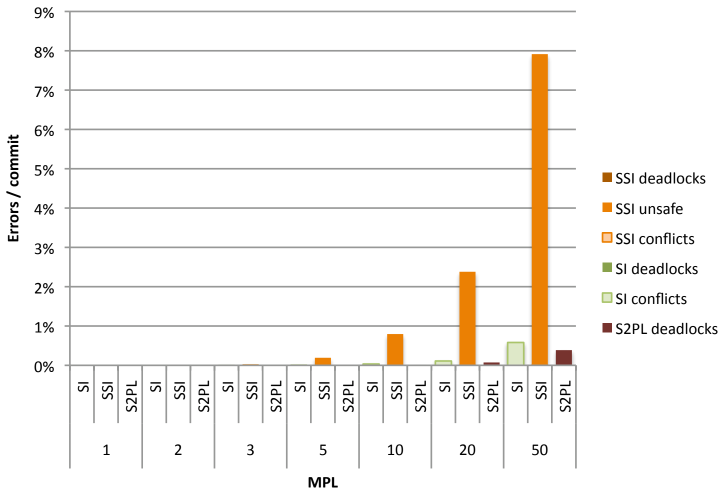
The results so far have concentrated on experiments with high contention in order to highlight the differences between the isolation levels. In Figure 6.4, we present results from running the benchmark including log flushes with a data set ten times larger. The increased volume of data results in far fewer conflicts between transactions but still fits in RAM so the only I/O operations are log writes. For S2PL, this produces less blocking and under SI and Serializable SI the rates of update conflicts are also lower than in the earlier experiments. In this case, the performance of S2PL and SI are almost identical and we can see the overhead in the Serializable SI implementation at between 10-15%.

This overhead is due in part to CPU overhead from managing the larger lock and transaction tables but primarily to a higher abort rate due to false positives with Serializable SI, as is evident in Figure 6.4(b). One issue in particular contributed to the false positive rate. It is clear that these errors are false positives because probability of true conflicts between transactions is low in this workload by the choice of data scaling.

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



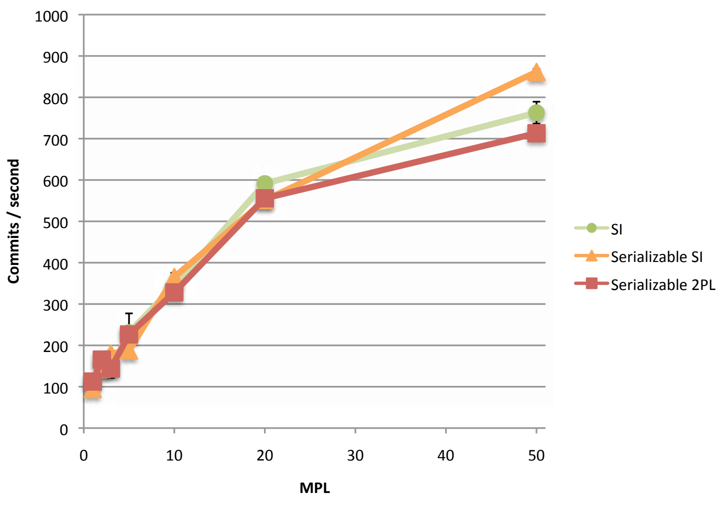
(a) Throughput



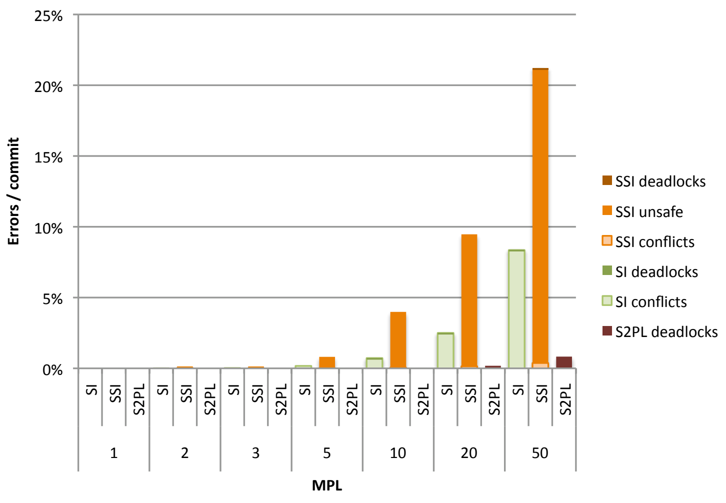
(b) Error rates

FIGURE 6.4: Berkeley DB `SmallBank` results when the log is flushed during commit and there is 1/10th of the contention.

In Figure 6.5, we measure the effect of combining the approach in Section 6.1.4 of using more complex transactions with low contention. The result is that there is very little to separate the throughput of the three isolation levels: all end up waiting for log flushes, and the contention for locks is low enough that S2PL does not suffer the degradation seen in the earlier results.



(a) Throughput



(b) Error rates

FIGURE 6.5: Berkeley DB `SmallBank` results when the log is flushed during commit, transactions perform 10 times more operations and there is 1/10th of the contention.

6.1.6 Berkeley DB Evaluation Summary

Berkeley DB implements an update as a delete of the old value followed by an insert of the new value, so B+Tree split and merge operations were occurring during the benchmark runs.

The high rates of false positives under our Berkeley DB prototype of Serializable SI are a concern, and were the motivation behind the improvements and optimizations to the basic algorithm described in Sections 3.6-3.7.

6.2 InnoDB Evaluation Setup

The MySQL server was run on a Linux system with an AMD Athlon XP 2600+ CPU with 2GB RAM running OpenSUSE Linux 11.0 with kernel version 2.6.25, glibc version 2.8 and GCC version 4.3.1. All data was stored using the EXT3 filesystem on a Maxtor 6Y060L0 PATA 7,200 RPM hard disk with write cache disabled. MySQL was started with only the following parameters modifying the default configuration (chosen after experimentation to give best throughput on our server without sacrificing ACID properties):

```
--max-connections=300 --table-cache=1000
--innodb_buffer_pool_size=1G --innodb_log_file_size=256M
--innodb_log_buffer_size=16M --innodb_flush_method=fsync
```

It is worth noting that these parameters force the log file to be flushed on every commit. We did run experiments where the log was not flushed, but do not include results here because in almost all cases, there was no difference between the throughput of the three concurrency control algorithms we want to measure.

The benchmark clients were run on a FreeBSD 7.0 system with an Intel Core(TM)2 Quad Q9300 at 2.50GHz and 4GB RAM using the Java HotSpot(TM) 64-Bit Server VM (build 1.6.0_03-p4) with the MySQL connector JDBC driver version 5.1.6. The systems were connected via a 100MBit Ethernet switch and were otherwise idle while the experiments ran.

We deliberately ran the database server on a slower machine so that it would reach CPU, memory and I/O saturation under different conditions during testing.

Each configuration was measured between 3 and 5 times, and the database cache was pre-warmed by loading all of the data into the database before measuring. Tests were run for 1 minute without measuring to allow the workload to reach a steady state, and results were measured for a further 2 minutes. All graphs include 95% confidence intervals (assuming a normal distribution of results), although in many cases the confidence intervals are too small to be visible.

6.3 InnoDB: `sibench` evaluation

To explore the performance impact of avoiding blocking for read-write conflicts, we first illustrate the trade-off between bottlenecks including CPU time, lock contention and write-ahead log flushes during commits. As described in Section 5.2, we designed and implemented a very simple synthetic microbenchmark called `sibench` based on a read-write conflict between an query transaction and an update transaction. Each of the two transactions consists of a single SQL statement, and no transaction is expected to abort when the benchmark is running.

The measurements were controlled by parameters that configured:

- the isolation level,
- the number of threads (the multi-programming level, or MPL),
- the number of items in the table, and
- the ratio of query transactions to update transactions, initially 1:1, and later 10 times more queries than updates.

In all of the results presented here, each client thread performs an equal number of query and update transactions. In these experiments, the error rates are omitted because there are effectively zero errors regardless of which concurrency control algorithm is selected.

6.3.1 Mixed workload

We begin by analyzing cases where there are equal numbers of query and update transactions submitted. That is, there is a 1:1 ratio of query transactions to update transactions.

We explore cases where $I \in \{10, 100, 1000\}$. When $I = 10$ and the number of query and update transactions are equal, there are 10 times more rows read than updated. In the extreme case where $I = 1000$, there are 1,000 rows read for each row updated.

Figure 6.6 shows the in case for $I = 10$, where the workload has a moderate ratios of reads to updates, SI scales to more than double the throughput of S2PL, since SI eliminates blocking for read-write conflicts.

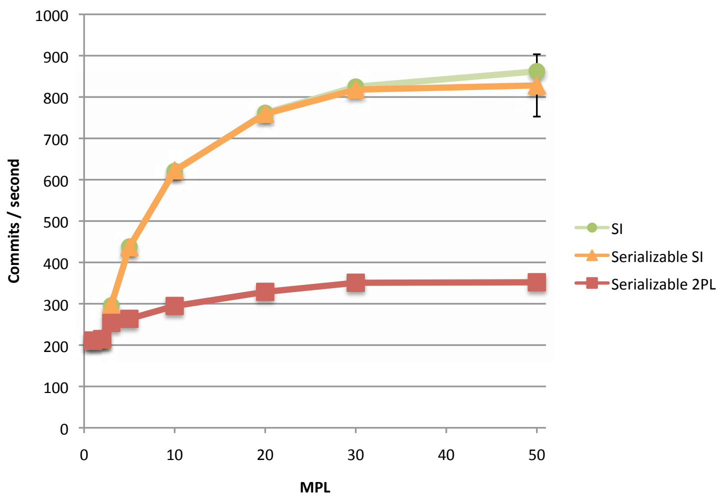


FIGURE 6.6: InnoDB `sibench` throughput with 10 items.

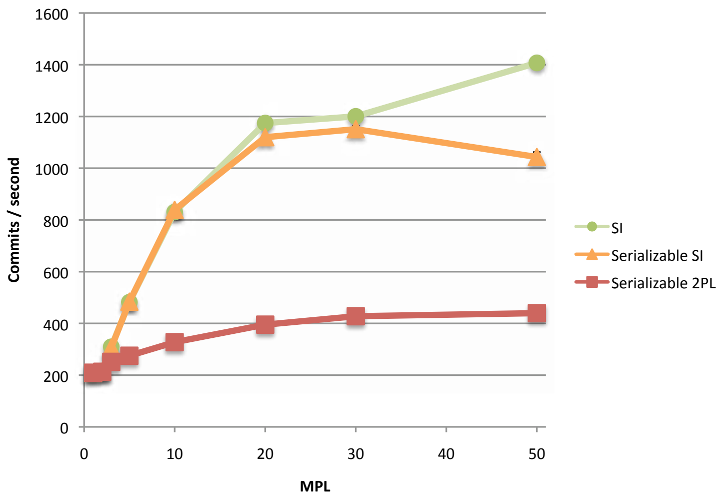


FIGURE 6.7: InnoDB `sibench` throughput with 100 items.

There is virtually no overhead for Serializable SI compared with SI. With such a small set of rows, the rate of conflicts between concurrent updates is high, particularly when MPL ≥ 10, and since updates block queries at S2PL, the throughput at S2PL is fairly flat after MPL 10.

When the number of rows is increased to 100, the workload is characterized by a high rate of reads per update (100 times more reads than updates). Figure 6.7 shows the results for this case, where SI scales to more than triple the throughput of S2PL, and with the larger set of rows, the rate of conflicts between concurrent updates is moderate, so that overall throughput is higher than when $I = 10$. There is virtually

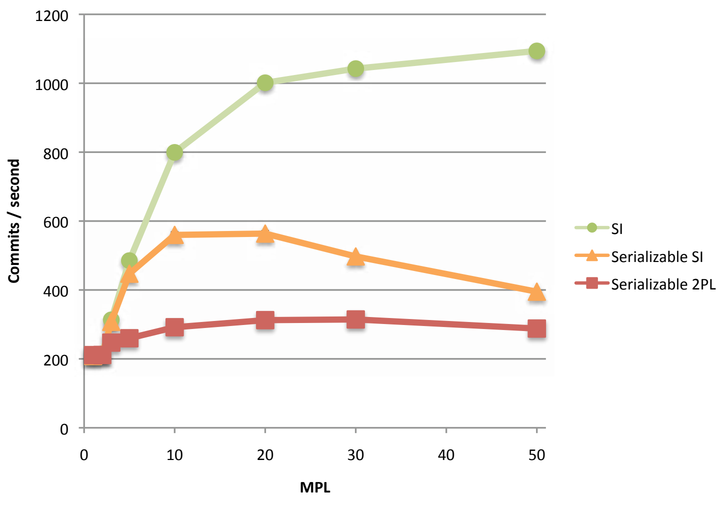


FIGURE 6.8: InnoDB `sibench` throughput with 1000 items.

no overhead for Serializable SI compared with SI until MPL 50, at which point the CPU becomes 100% utilized at Serializable SI due to the additional work of managing the larger lock table, while at SI the CPU is not saturated. For this reason, we begin to see a divergence between the throughput of SI and Serializable SI at MPL 50.

When I is increased further, to 1000, in Figure 6.8, more CPU time is required to sort the rows during each query and there are 10 times more requests to the lock manager than in Figure 6.7. Here we see that the gap between SI and Serializable SI is more pronounced, for two reasons:

- the CPU becomes saturated earlier under Serializable SI, at around MPL 20 rather than at MPL 30 under SI, due to the additional work involved in managing SIREAD locks in the lock table; and
- there is now significant contention for the mutex protecting the lock table. As mentioned in Section 4.4, the InnoDB lock table is protected by a single, global mutex, so concurrent lock requests contend for this mutex even when there is no chance that the requested locks would conflict. A more sophisticated lock manager should scale better under this Serializable SI workload, whereas the S2PL throughput is constrained by blocking caused by logical lock conflicts rather than a limitation of the lock manager implementation.

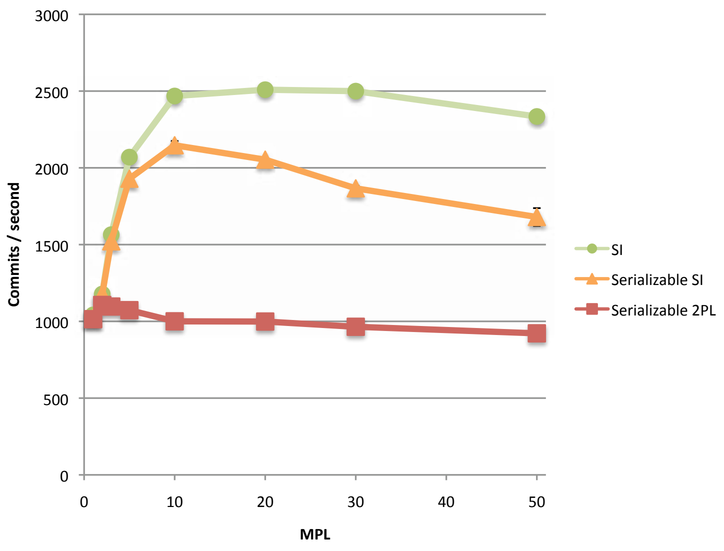


FIGURE 6.9: InnoDB `sibench` throughput with 10 items and 10 queries per update.

6.3.2 Query-mostly workloads

We now consider cases where there are ten times more query transactions than update transactions submitted. That is, there is a 10:1 ratio of query transactions to update transactions.

Note that we run the queries at the same isolation level as the update, even though the results would be serializable with the query run at various weak isolation levels. The intent here is to measure the performance of the concurrency control algorithms under extreme conditions, not to tune the system for the best throughput for a given workload.

When there are 10 items, in Figure 6.9, we see that the absolute throughput rates are higher with the query-mostly workload than with the mixed workload with the same data volume in Figure 6.6, simply because queries do not need to flush the transaction log when they commit and so they can processed more quickly.

Further, the throughput at S2PL is virtually constant, at 1000 tps, regardless of MPL. This is because the disk subsystem can support approximately 100 flushes per second, and there are only 10 rows in the table, so only 10 updates can commit simultaneously. Since an update conflicts with any concurrent read in `sibench`, the rate at which updates can commit constrains the total throughput.

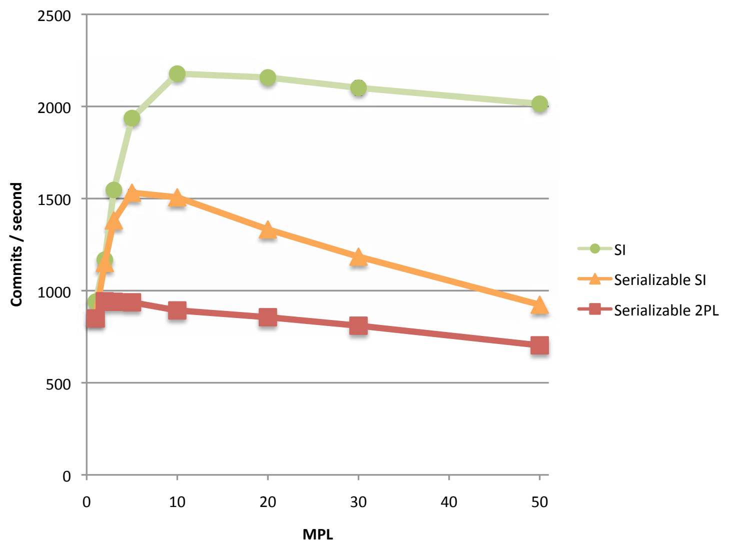


FIGURE 6.10: InnoDB `sibench` throughput with 100 items and 10 queries per updates.

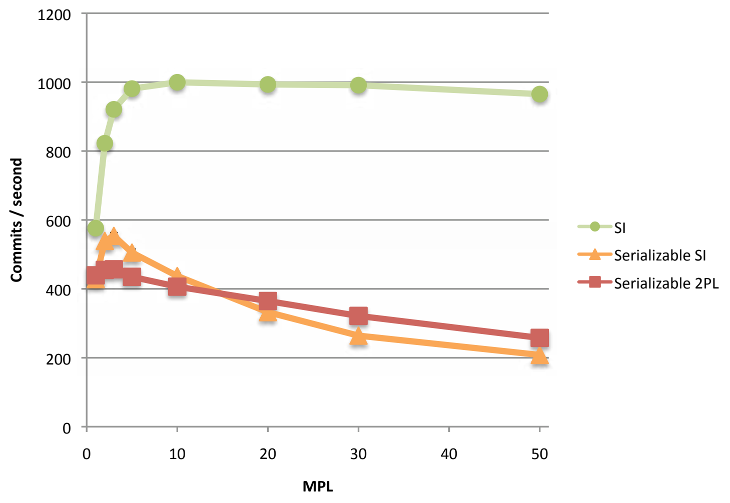


FIGURE 6.11: InnoDB `sibench` throughput with 1000 items and 10 queries per update.

In this workload there are 100 times more read operations than writes, as there are 10 queries per update and each query reads 10 rows. So the difference between the throughput of SI and Serializable SI can be explained by the 100 times more requests to the the InnoDB lock manager, comparable to what we saw in Figure 6.7.

With 100 items, the query-mostly workload run at Serializable SI generates 1000 times more lock requests than at SI, and in Figure 6.10 we see overall trends similar to Figure 6.8.

In the extreme case of Figure 6.11, $I = 1000$ and there are 10 queries executed for each update transaction. As a result, there are 10,000 times more reads than writes. Profiling showed that both the Serializable SI and S2PL algorithms end up spending virtually all of the time waiting for the lock manager mutex. We believe that with a more scalable lock manager implementation, Serializable SI would outperform S2PL in cases such as these because no blocking is required inside the lock manager

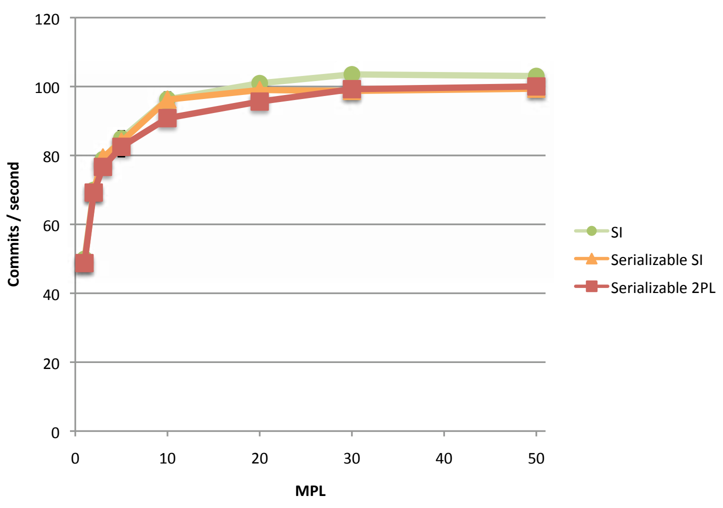
6.3.3 InnoDB: `sibench` summary

This section has measured the throughput of three concurrency control algorithms in InnoDB using our new `sibench` microbenchmark, which has two main parameters: table size and ratio of reads to writes. In spite of its simplicity, `sibench` enabled us to explore the parameter space, and to discover regions where either the CPU, logical lock contention or constraints inherent in the design of InnoDB itself were the primary factors determining throughput. Our prototype implementation of Serializable SI in InnoDB seems to be performing well: only in extreme cases is there significant departure from the performance obtained by SI.

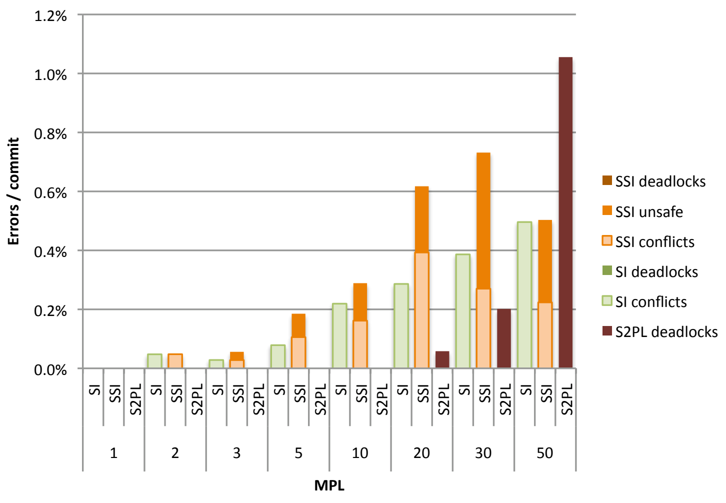
6.4 InnoDB: TPC-C++ evaluation

We now present our findings from running the TPC-C++ benchmark, described in Section 5.3, with our InnoDB prototype. This benchmark is a modification of TPC-C that introduces a new transaction type, Credit Check, and with it, the potential for non-serializable executions when TPC-C++ is run with SI.

Many more factors are at play in a benchmark of the complexity of TPC-C++, compared with the `sibench` microbenchmark. TPC-C++ can be configured to generate cache misses from data sets too large to fit in cache and involves more complex transaction logic that includes predicates and joins. Consequently, we do not expect that concurrency control will always be the main factor determining the results. On the other hand, a more complex benchmark is more representative of real applications than the workloads we have considered so far in this chapter, and gives a better sense of the impact of our work on the performance of the system as a whole.



(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.



(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.12: Throughput and error rates for InnoDB running TPC-C++ with 1 warehouse, skipping year-to-date updates.

Figure 6.12 shows the most common case that we saw when running TPC-C++ with a wide variety of parameters. It shows the throughput and error rates for TPC-C++ with a single warehouse, skipping the updates to the contentious year-to-date summary field `w_ytd` in the Warehouse table as described in Section 5.3.1. Here, concurrency control is not the primary factor determining the throughput, and all three choices of isolation level give comparable performance. Another way of putting this is that the application might as well use Serializable SI or S2PL over SI, since serializable executions are guaranteed with no measurable performance impact.

The rates of errors are shown in Figure 6.12(b). The rates are generally low, with less than 1% of transactions aborting. The general trends is that the rate of errors increases as the concurrent load is increased. That is because the probability of a pair of transactions conflicting is higher when there are more transactions running concurrently. There are somewhat higher rates of errors with Serializable SI than with SI, but since the overall rates are low, this difference has little impact on their relative throughputs.

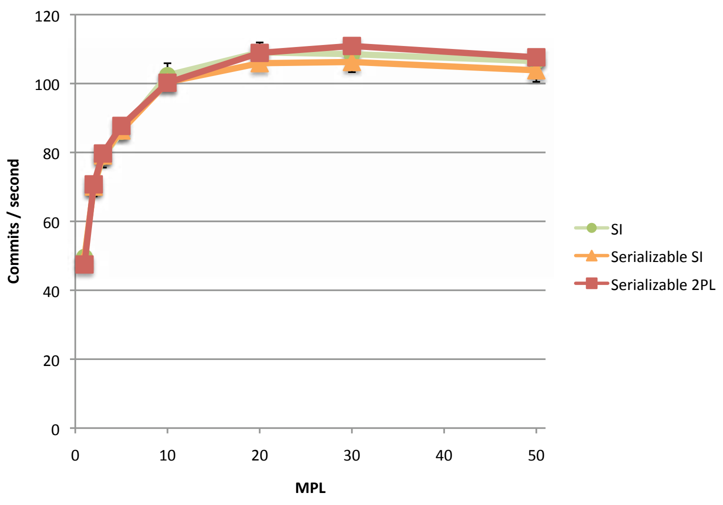
6.4.1 Larger data volume

The next situation we investigate is the impact of a larger data volume. As explained in Section 5.3.6, increasing the number of warehouses has two effects: it reduces the contention on the Warehouse year-to-date field and it can make the data too large to fit in cache. In this section, we explore the case where there are 10 warehouses, which corresponds to around 1.2GB of data, too much to fit in the 1GB of RAM in our test server.

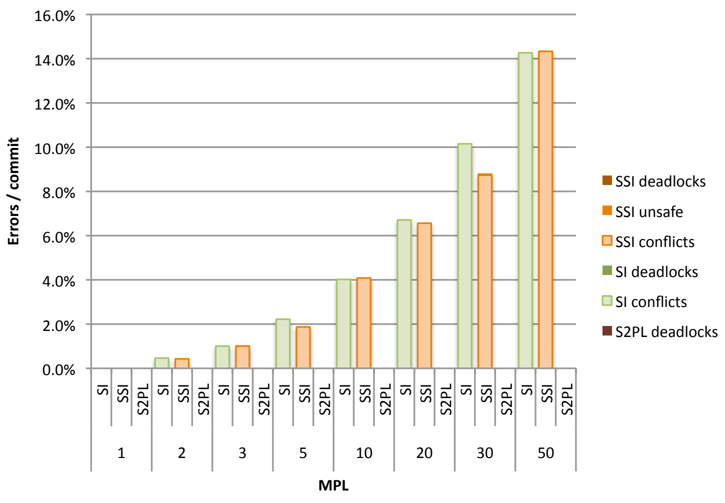
Figure 6.13 shows the throughput and error rates for TPC-C++ with $W = 10$. In Figure 6.13(a), we see that the throughput for all three isolation levels are virtually identical. Again, this was by far the most common case when we ran TPC-C++ with a variety of parameters. The TPC-C schema that we are using is designed so that conflicts become rarer as the data volume is scaled, so it is not surprising to see that the choice of isolation level is not the main contributor to performance.

However, recall that when TPC-C++ is run with ordinary SI, the execution is *not* serializable in general. Undeclared consistency constraints may be violated. In Figure 6.13(b) we give a breakdown of error rates in a stacked bar graph. Here the situation is simple because there are no deadlocks at any isolation level and the FCW is the overwhelming cause of transaction aborts. “Unsafe” errors contribute only a tiny fraction of the total (less than 1% of the FCW errors and too small to see on the graph). Figure 6.13 is the common case where avoiding SI anomalies has no measurable impact on performance.

We noticed the moderate to high rate of SI conflicts due to the first-committer-wins rule in Figure 6.13(b) and wanted to understand what was causing the conflicts. The benchmark is run without partitioning, so updates to the field `d_next_o_id` for a district in the New Order transaction conflict unnecessarily with updates to the `d_ytd` field in the Payment transaction.



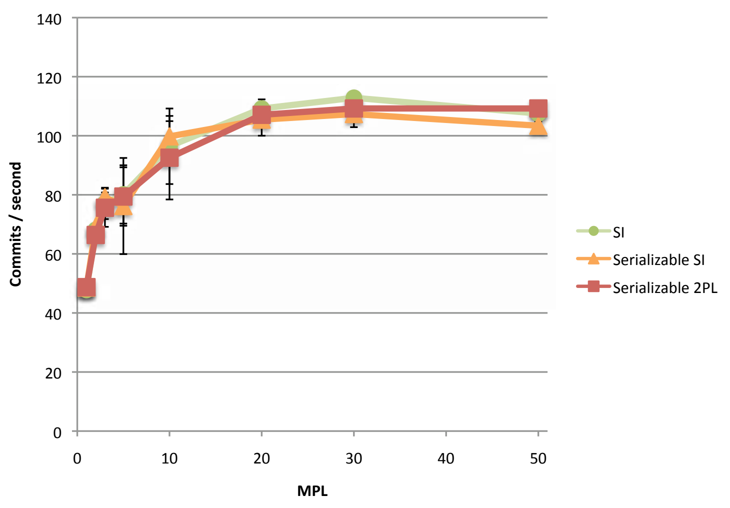
(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.



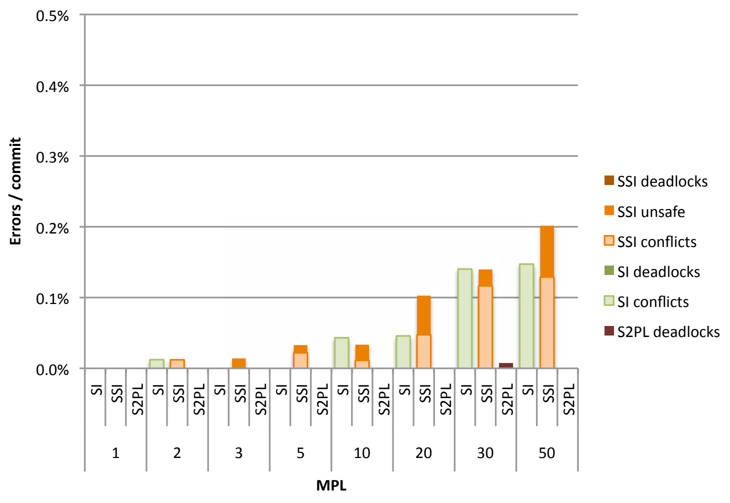
(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.13: Throughput and error rates for InnoDB running TPC-C++ with 10 warehouses, including year-to-date updates.

To investigate whether this spurious conflict is the source of the higher than expected error rates, we ran the benchmark again but modified the Payment transaction to avoid updating the `d_ytd` field in the District table and the `w_ytd` field in the Warehouse table, as described in Section 5.3.1. The results are shown in Figure 6.14. Comparing Figure 6.13(b) with Figure 6.14(b), we see can see that the rate of update conflicts at SI and Serializable SI has been reduced by approximately a factor of 100. This indicates that the year-to-date fields contribute the overwhelming majority of update conflicts in



(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.



(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.14: Throughput and error rates for InnoDB running TPC-C++ with 10 warehouses, skipping year-to-date updates.

TPC-C running at SI. Thus they would not occur if the District table were partitioned such that the `d_next_o_id` and `d_ytd` fields were stored in separate partitions.

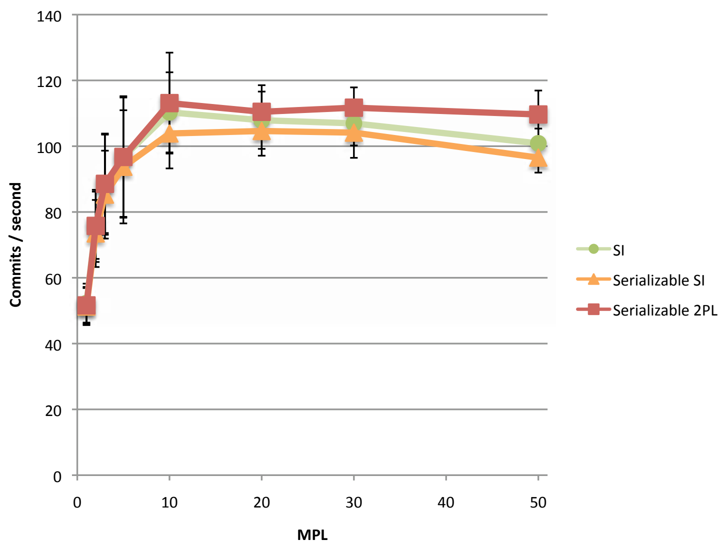
6.4.2 High Contention

In Figure 6.15, we show the results for the case where there are 10 warehouses and data is scaled according to the *tiny scale* parameters given in Section 5.3.6. This configuration creates some contention on the Customer table, since there are only 100 customers per district. However, the results in Figure 6.15 include updates to the year-to-date fields, and that contention dominates the contention for customer rows.

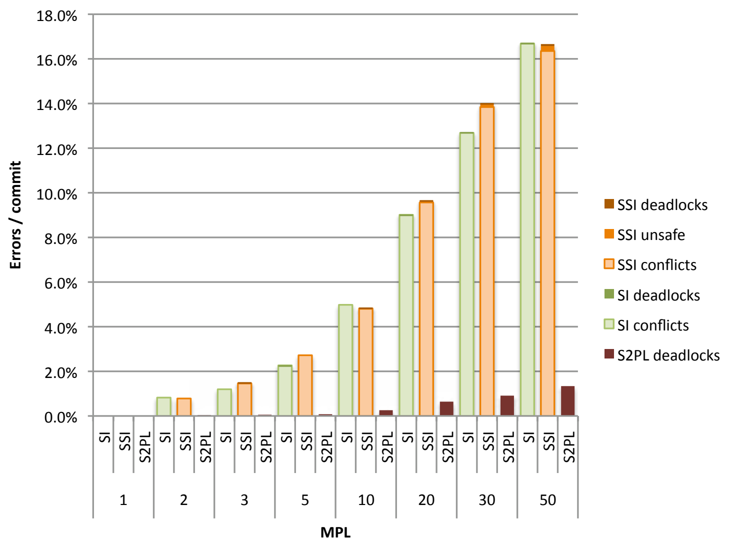
In Figure 6.15, we first note that the confidence intervals are larger than in our other results (i.e., there is more variance between runs). The inherent randomness of the workload combines with the reduced data set to increase the level of contention, and thus to increase the rates of deadlocks, but particularly update conflicts, as seen in Figure 6.15(b). Given the larger confidence intervals, it is difficult be sure of the relative performance of the three algorithms in this configuration, so next we try a more extreme case.

In Figure 6.16, we again use the tiny scale parameters for the data volume, but now with only a single warehouse. As mentioned in Section 5.3.6, this configuration represents under 2MB of data in total across all of the tables, so there are many more opportunities for concurrent transactions to conflict. To eliminate the most obvious of these, we show the results where the year-to-date fields are *not* updated. That is, the errors are due to conflicts on other fields, such as the next order ID field in each District, or from concurrent transactions choosing the same customer.

This is the only case where we found Serializable SI slower than S2PL (by approximately 10%). The error rates clearly show the incremental effect of unsafe errors above the conflicts of standard SI. The high rate of aborts at Serializable SI, including rates of "unsafe" errors up to 10% of the update conflict rate, has a measurable effect on throughput in this configuration. Note however, that this is an extreme case, with tiny data volumes chosen specifically in order to create high rates of conflicts.

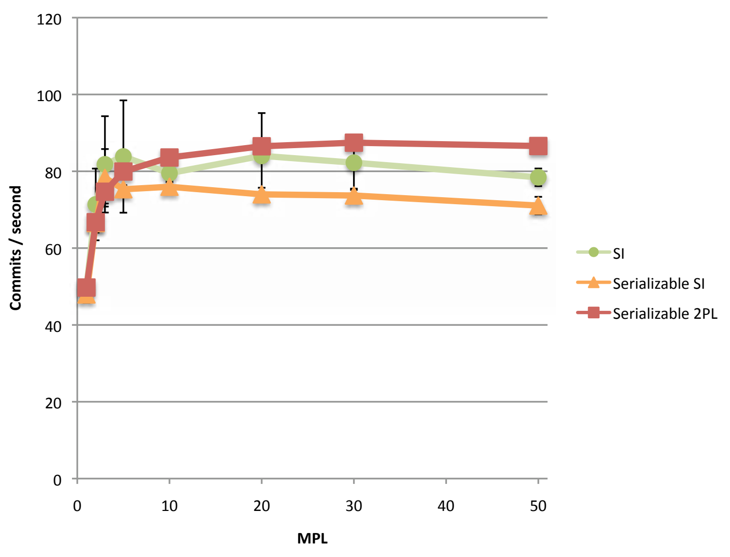


(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.

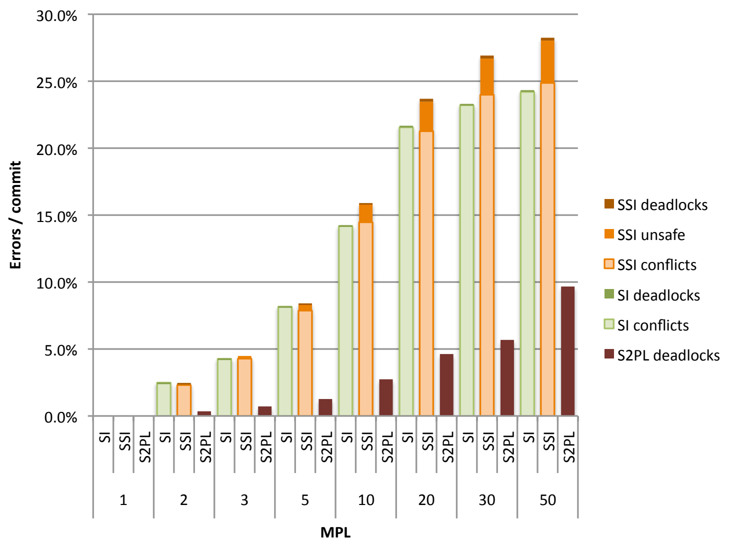


(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.15: Throughput and error rates for InnoDB running TPC-C++ with 10 warehouses and tiny data scaling, including year-to-date updates.



(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.



(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.16: Throughput and error rates for InnoDB running TPC-C++ with 1 warehouse and tiny data scaling, skipping year-to-date updates.

6.4.3 The Stock Level Mix

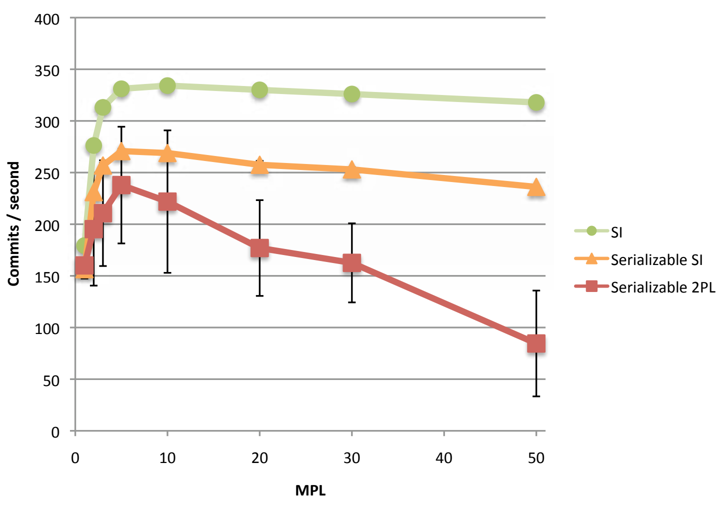
Finally, we tried to configure TPC-C++ to highlight the effects of concurrency control. Multiversion concurrency control provides most benefit in situations where read-write conflicts are the primary bottleneck, which correspond to dashed lines in the static dependency graph in Figure 5.3 of Section 5.3. Here we focus on the edge between the New Order transaction and the Stock Level transaction, which is due to the Stock Level transaction reading stock levels that are updated by the New Order transaction as orders are entered into the database.

The Stock Level transaction examines the 20 most recent orders, and there are an average of 10 order lines per order. The New Order transaction modifies approximately 20 rows on average. The result is that this mix is skewed, with approximately 100 rows read for each row updated. We call this the *Stock Level Mix*, as described in Section 5.3.5.

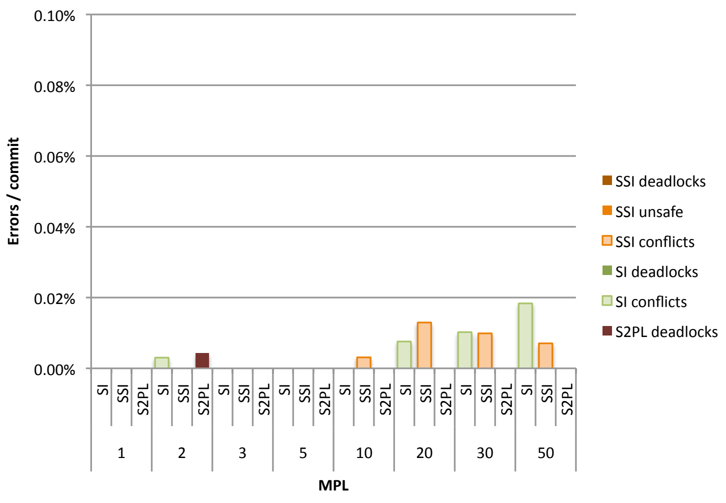
In Figure 6.17 are the results for running the Stock Level Mix with 10 warehouses. In Figure 6.17(a), we see that Serializable SI performs approximately 20% slower than SI. However, since the data for this configuration is larger than the cache, S2PL holds shared locks that block updates while waiting for I/O, and we see that S2PL performs far worse than Serializable SI, with negative scaling after MPL=5 and much higher variability.

Finally, we present Figure 6.18, where the data volume is scaled to much smaller than normal in order to make conflicts more common. These results were obtained with tiny scale data as described in Section 5.3.6: 1 warehouse, 10 districts, 100 customers per district, and 1000 items. The same mix of 10 Stock Level transactions per New Order transaction was measured, and Figure 6.18(a) shows that again, Snapshot Isolation performs best, with Serializable SI between 10 and 20% faster than S2PL.

In Figure 6.18(b), we see very high rates of deadlocks at S2PL, but a comparatively small gap between the throughput at SI and the throughput at S2PL (approximately 40% at MPL 20 in Figure 6.18(a)). The rate of deadlocks at S2PL in this configuration may be prohibitive for applications with more complex transactions than TPC-C++, where the cost of restarting a transaction is high. The throughput at Serializable SI is in between S2PL and SI, but we believe that this case is comparable to Figure 6.11, where the InnoDB lock manager implementation is the main factor that limits scaling.

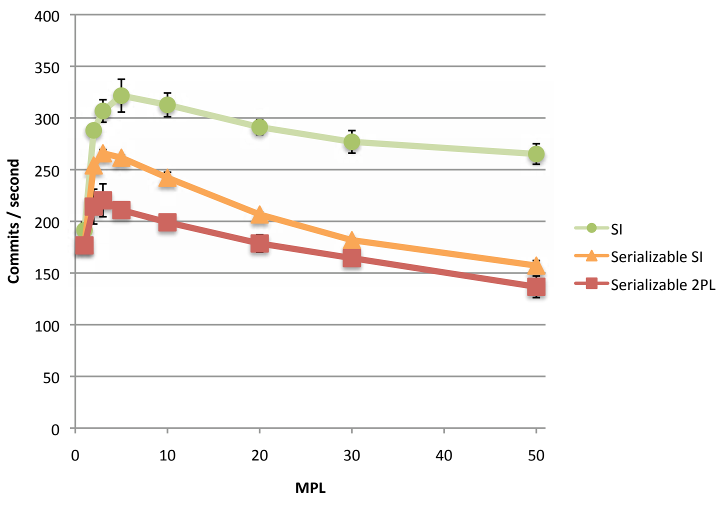


(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.

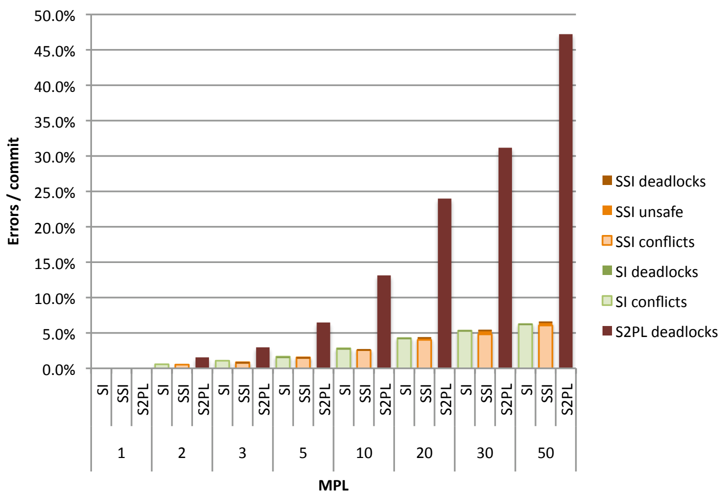


(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.17: Throughput and error rates for InnoDB running TPC-C++ Stock Level Mix with 10 warehouses.



(a) Throughput of TPC-C++ at SI, S2PL and Serializable SI.



(b) Error rates at SI, S2PL and Serializable SI.

FIGURE 6.18: Throughput and error rates for the TPC-C++ Stock Level Mix with 1 warehouse and tiny data scaling.

6.5 Summary

In this chapter, we presented performance measurements of our two prototype implementations of Serializable SI: one in Berkeley DB, evaluated with the `SmallBank` benchmark, and another in InnoDB, evaluated with the `sibench` microbenchmark and with TPC-C++. We compared each platform's SI and S2PL implementations with our prototype implementations of Serializable SI. Results for `SmallBank` and TPC-C++ at S2PL and Serializable SI reflect the throughput of executions that maintain consistency, so while the throughput of SI is higher in some cases, those executions may have left the database in an inconsistent state.

Our results show that for low-contention workloads, the cost of guaranteeing serializable executions is small. The additional precision and optimizations implemented in the InnoDB prototype give substantially better results than we measured in the Berkeley DB prototype. However, in some situations, where the rate of read operations was very high, the single-threaded design of the InnoDB lock manager became evident in our measurements, and Serializable SI did not always match the performance of SI.

Summarizing our observations in this chapter:

- When the workload is CPU-bound, as in Section 6.1.2, blocking severely limits throughput: we saw throughput using Serializable SI more than 10 times better than using S2PL, without risking inconsistencies.
- When the workload is I/O-bound, the choice of concurrency control algorithm affects throughput in cases where threads are holding locks while waiting for I/O. This is true for both the exclusive locks held while the transaction log is flushed, and also for shared locks held by S2PL while waiting to read pages that are not found in cache, as in Figure 6.17.
- Serializable SI gives performance close to SI unless the database is spending a significant amount of time servicing reads: in that case, scalability limitations of the InnoDB lock manager became apparent. We expect that a more sophisticated lock manager that does not rely on a single, global mutex would help in these cases.
- If we change the transaction mix to have many read-write conflicts, we see that SI outperforms S2PL, with Serializable SI somewhere in between, depending on the ratio of read operations to write operations.

When Serializable SI was implemented at page-level granularity (in Berkeley DB), there was a moderate to high rate of false positives, where page-level conflicts between transactions operating on different records caused unnecessary "unsafe" aborts, with up to 8% of all transactions aborted unnecessarily in Figure 6.4(b).

With row-level granularity (in InnoDB) together with the optimizations in Section 3.6, the rate of false positives detected by our prototype is much lower, because even in extreme cases such as Figure 6.16, the rate of "unsafe" aborts was never more than 2%. In the InnoDB prototype, there was no case where "unsafe" errors were a significant contributor to the measured throughput. This is because the true rate of non-serializable interleavings is small and the rate of "false positives" is also low. Even when skewing the data volume and transaction mix to induce more read-write conflicts, the interleaving required to cause a write skew was quite rare in practice.

If read-write conflicts are the primary bottleneck, using SI or Serializable SI can significantly outperform S2PL, but other factors (such as I/O) often dominate the effects of concurrency control. To measure any difference between the throughput with SI versus Serializable SI, we observed that the following combination of conditions is required:

(1) The workload is dominated by queries run at Serializable SI (so Serializable SI is using the lock manager but SI is not);
(2) The DBMS is using 100% CPU;
(3) There are many threads in the DBMS simultaneously (so that the lock table becomes large and the queue waiting for the lock manager grows long); and
(4) There is no think time in the benchmark.

Each of these conditions can be addressed in turn:

(1) Run pure queries at SI. As noted in (Fekete *et al.*, 2004), this can lead to anomalies where a read-only query sees a database state that could not have occurred in a serializable execution. However, data consistency will be maintained, because updates will be serializable, as described in Section 3.8.
(2) Buy a faster CPU: we used a single-CPU machine to generate results because it was more difficult to measure differences between the isolation levels on a faster quad-CPU machine.

(3) Limit the number of threads in the DBMS – we overrode the InnoDB default of a maximum of 8 simultaneous threads to amplify the trends.

(4) Use a benchmark that reflects a real application – we deliberately left think times out of our benchmarks because we wanted to stress the DBMS.

CHAPTER 7

Conclusions and Future Work

This thesis began by identifying a problem: many database applications risk inconsistent results in order to improve performance. In particular, we described the popular snapshot isolation algorithm that is provided by many database systems because of its superior performance when compared with strict two-phase locking. For applications that need to maintain data consistency using snapshot isolation, developers have to perform careful analysis to determine whether their transactions could experience anomalies due to interleaving of operations by the database system. If the potential for anomalies is discovered, the only recourse is to modify the application to avoid generating non-serializable interleavings. Choosing appropriate modifications is itself a delicate process of system- and application-specific measurement and design.

Our proposal, detailed in this thesis, is to shift the responsibility for guaranteeing serializable executions to the database management system rather than expecting application developers to detect dangerous structures in their programs and eliminate them manually. After all, these patterns are only "dangerous" as a consequence of the concurrency control algorithm implemented by the database platform: the same application, run on a platform providing true serializable isolation, would maintain data consistency.

7.1 Serializable Snapshot Isolation Algorithm

This thesis presents a new method for implementing serializable isolation based on a modification of snapshot isolation. Our Serializable Snapshot Isolation algorithm maintains the properties that make snapshot isolation attractive, including that read operations do not delay or wait for concurrent writes, but it prevents patterns of conflicts between transactions that could result in non-serializable executions. Our algorithm guarantees serializable executions by adding only a small amount of data per transaction, using existing mechanisms present in almost all database systems, and with low overhead under a wide range of workloads when compared with snapshot isolation.

The Serializable SI algorithm is conservative, and in some cases leads to slightly higher abort rates than SI. However, the impact on system throughput is generally small and in most cases when read-write conflicts are the primary bottleneck, performance is substantially better than the locking implementations of serializable isolation built into Berkeley DB and InnoDB.

We presented the Serializable SI algorithm by first describing a simple version that deals only with reads and writes to named data items, and then extending the basic algorithm to:

(1) detect phantoms introduced by predicate reads;
(2) improve its precision;
(3) reduce the cost of restarting transaction by aborting them earlier; and
(4) minimize its execution overhead by cleaning up suspended transactions and their locks earlier.

7.2 Implementation Experience

Prototype implementations of the algorithm in both Oracle Berkeley DB and the InnoDB transactional storage backend for MySQL are described and shown to perform significantly better that two-phase locking in a wide variety of cases, usually comparably with snapshot isolation.

One property of Berkeley DB that simplified that prototype implementation was working with page level locking and versioning. That meant that the “basic” version of the algorithm was sufficient in Berkeley DB to guarantee serializable executions. This thesis extends the basic algorithm to support locking and versioning with row-level granularity, and uses a “gap” locking protocol to detect and prevent phantoms. The enhanced algorithm was the basis of the InnoDB prototype, and our evaluation shows the advantages of finer granularity in much lower rates of “false positives” compared with the Berkeley DB implementation.

7.3 TPC-C++: a modification to TPC-C for snapshot isolation

In order to evaluate the performance of our algorithm, we needed a benchmark that is sensitive to SI anomalies. Otherwise, we could only test the cost of guaranteeing serializable executions in cases where the execution is already serializable!

Since the industry-standard TPC-C benchmark does not generate any anomalies when executed at SI, we modified it slightly with one new transaction type, called *Credit Check* that determines whether the total price of outstanding orders for a customer exceeds their credit limit, and if so, changes their credit rating. This transaction uses the existing TPC-C schema, and is a piece of plausible business logic in the scenario TPC-C models. However, the new transaction introduces the possibility of non-serializable executions, where a customer may be flagged as having a bad credit rating when placing an order, even after paying for their outstanding orders and placing another order that was not flagged as bad. We called the modified version of TPC-C with the new transaction type *TPC-C++*.

7.4 Experimental Evaluation

After running a variety of workloads with the concurrency control algorithms built into Berkeley DB and InnoDB as well as the prototype implementations of Serializable SI, we first noted that in many cases the overall system performance was the same regardless of the choice of concurrency control. In those situations, we recommend choosing an algorithm that guarantees serializable executions, such as S2PL or Serializable SI, because consistency is guaranteed for essentially no cost.

To measure any difference between the throughput with SI and Serializable SI, we observed that the workload must be dominated by queries run at Serializable SI (so Serializable SI is making many more requests of the lock manager than SI). In addition, there must be significant load on the DBMS (with the application spending very little time outside the DBMS). At the end of Chapter 6, we gave recommendations for how to avoid this situation, so that the cost of guaranteeing serializable execution is kept small.

7.5 Limitations

We cannot claim that our approach is superior to existing concurrency control algorithms in all situations:

- Some applications do not need strong guarantees that data is kept consistent. The data may not be particularly valuable, or it may be feasible to reconcile inconsistencies in order to recover from non-serializable executions.
- Our focus was on an algorithm for centralized database servers. While it would be possible to directly implement our algorithm in a replicated or "cloud" DBMS, the overhead of maintaining SIREAD locks as described in this thesis is likely to be prohibitive.

7.6 Future Work

In the future, we hope to further optimize the Serializable SI algorithm and improve its precision by further reducing the rate of false positives. Our current scheme for determining whether transactions have committed in an order that could cause a non-serializable execution (i.e., our tests for whether T_{out} commits first) are conservative but not precise, particularly if one transaction conflicts with multiple concurrent transactions before committing.

We remarked in Section 3.8 that if our algorithm is adopted by DBMS platforms, we anticipate that it will be attractive to many users to run updates at Serializable SI and run queries at ordinary SI. Although not serializable (under some conditions, queries may see a database state that could never have existed in a serializable execution), this combination has an attractive combination of properties:

- update transactions are serializable, so they cannot introduce inconsistencies into the database state;
- queries never need to obtain SIREAD locks, so they place no overhead on the lock manager and never abort due to "unsafe" errors; and
- queries never need to be suspended when they commit.

We have not yet carried out a thorough performance analysis of this configuration, and would like to pursue this in the future.

Our existing approach acquires separate SIREAD locks for each read operation, including for predicates. Efficient implementations of S2PL avoid acquiring many separate record locks and instead acquire intention locks on larger granules such as database pages or tables. We would like to use the same idea with Serializable SI by introducing intention-SIREAD locks (say at the table level) to reduce the number of lock operations required to detect rw-conflicts.

Our evaluation highlighted the importance of a scalable lock manager. In future we hope to implement this algorithm in a system with a highly concurrent lock manager to further explore its performance. Similarly, our approach is compatible with techniques that improve the performance of concurrent updates to a btree by holding short-term locks on interior pages. In addition, we are investigating how to adapt this technique to a distributed setting to provide serializable isolation for replicated database systems, where a centralized lock manager is not feasible, as it limits performance and is a single point of failure.

There has been a substantial body of research relating to database replication, also called “cloud databases”. In particular, several researchers have proposed algorithms for replication that offer snapshot isolation across the cluster of databases as their model of correctness (Elnikety *et al.*, 2005; Wu and Kemme, 2005). We would like to adapt Serializable SI to replicated databases such as these in order to guarantee serializable execution in a distributed context. The main obstacle to doing so is the cost of communicating the sets of data items read along with the write sets for each transaction in order for a commit to be certified, so that all machines in the cluster agree whether it will commit or abort. However, we do not need precise knowledge of all reads in order to detect potentially unsafe patterns of rw-conflicts. It is sufficient for correctness to have an overestimate of the read set (that may have a more compact representation), so we hope that the communication overhead will not be prohibitive.

References

[Adya1999] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions (PhD thesis)*. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, March.

[Agrawal *et al.*1987a] Rakesh Agrawal, Michael J. Carey, and Miron Livny. 1987a. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654.

[Agrawal *et al.*1987b] Rakesh Agrawal, Michael J. Carey, and Lawrence McVoy. 1987b. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, 13(12):1348–1363.

[Alomari *et al.*2008a] Mohammad Alomari, Michael J. Cahill, Alan Fekete, and Uwe Röhm. 2008a. The cost of serializability on platforms that use snapshot isolation. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, pages 576–585, Cancún, México, April 7-12. IEEE.

[Alomari *et al.*2008b] Mohammad Alomari, Michael J. Cahill, Alan Fekete, and Uwe Röhm. 2008b. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels? In Jayant R. Haritsa, Kotagiri Ramamohanarao, and Vikram Pudi, editors, *DASFAA*, volume 4947 of *Lecture Notes in Computer Science*, pages 267–281. Springer.

[Anonymous1985] Anonymous. 1985. A measure of transaction processing power. *Datamation*, 31(7):112–118.

[Ashdown and others2005] Lance Ashdown et al. 2005. *Oracle® Database Application Developer's Guide - Fundamentals, 10g Release 2 (10.2), part number B14251-01*. Oracle Corporation, November.

[Berenson *et al.*1995] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, , and P. O'Neil. 1995. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10. ACM Press, June.

[Bernstein and Goodman1981] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.

[Bernstein and Goodman1983] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483.

[Bernstein *et al.*2000] A. Bernstein, P. Lewis, and S. Lu. 2000. Semantic conditions for correctness at different isolation levels. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, pages 57–66. IEEE, February.

[Bober and Carey1992] Paul M. Bober and Michael J. Carey. 1992. On mixing queries and transactions via multiversion locking. In *ICDE '92: Proceedings of the eighth International Conference on Data Engineering*, pages 535–545, Washington, DC, USA. IEEE Computer Society.

[Cahill *et al.*2008] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 729–738, New York, NY, USA. ACM.

[Carey and Muhanna1986] Michael J. Carey and Waleed A. Muhanna. 1986. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems (TOCS)*, 4(4):338–378.

[Carey and Stonebraker1984] Michael J. Carey and Michael Stonebraker. 1984. The performance of concurrency control algorithms for database management systems. In *VLDB '84: Proceedings of the 10th International Conference on Very Large Data Bases*, pages 107–118, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Chan *et al.*1983] Arvola Chan, Umeshwar Dayal, Stephen Fox, Nathan Goodman, Daniel R. Ries, and Dale Skeen. 1983. Overview of an ada compatible distributed database manager. In *SIGMOD '83: Proceedings of the 1983 ACM SIGMOD international conference on Management of data*, pages 228–237, New York, NY, USA. ACM.

[Elnikety *et al.*2005] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. 2005. Database replication using generalized snapshot isolation. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Washington, DC, USA. IEEE Computer Society.

[Eswaran *et al.*1976] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633.

[Fekete *et al.*2004] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. 2004. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record*, 33(3):12–14.

[Fekete *et al.*2005] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528.

[Fekete1999] Alan Fekete. 1999. Serializability and snapshot isolation. In *Proceedings of Australian Database Conference*, pages 201–210. Australian Computer Society, January.

[Fekete2005] Alan Fekete. 2005. Allocating isolation levels to transactions. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 206–215, New York, NY, USA. ACM.

[Gray and Reuter1993] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[Gray *et al.*1975] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1975. Granularity of locks in a large shared data base. In Douglas S. Kerr, editor, *VLDB*, pages 428–451. ACM.

[Gray1993] Jim Gray. 1993. *The Benchmark Handbook (2nd ed)*. Morgan Kaufmann.

[Hadzilacos1988] Thanasis Hadzilacos. 1988. Serialization graph algorithms for multiversion concurrency control. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 135–141, New York, NY, USA. ACM.

[Hellerstein *et al.*2007] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a database system. *Found. Trends databases*, 1(2):141–259.

[Innobase Oy2008] Innobase Oy. 2008. InnoDB web site `http://www.innodb.com/`.

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

[Jorwekar *et al.*2007] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. 2007. Automating the detection of snapshot isolation anomalies. In *VLDB '07: Proceedings of the 33rd international conference on Very Large Data Bases*, pages 1263–1274. VLDB Endowment.

[Kumar1995] Vijay Kumar, editor. 1995. *Performance of concurrency control mechanisms in centralized database systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Kung and Robinson1981] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226.

[Lomet1993] David B. Lomet. 1993. Key range locking strategies for improved concurrency. In *VLDB '93: Proceedings of the 19th international conference on Very Large Data Bases*, pages 655–664, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Mohan and Levine1992] C. Mohan and Frank Levine. 1992. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 371–380, New York, NY, USA. ACM.

[Mohan *et al.*1992] C. Mohan, Hamid Pirahesh, and Raymond Lorie. 1992. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 124–133, New York, NY, USA. ACM.

[Mohan1990] C. Mohan. 1990. ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of the sixteenth international conference on Very large databases*, pages 392–405, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[MySQL AB2006] MySQL AB. 2006. *MySQL Administrator's Guide and Language Reference (2nd Edition)*. MySQL Press.

[Olson *et al.*1999] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191.

[PostgreSQL2009] Global Development Group for PostgreSQL. 2009. PostgreSQL web site `http://www.postgresql.org/`.

[Raz1993] Yoav Raz. 1993. Commitment ordering based distributed concurrency control for bridging single and multi version resources. In *Proceedings of Third International Workshop or Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, pages 189–198. IEEE, June.

[Reed1978] David P. Reed. 1978. Naming and synchronization in a decentralized computer system. Technical report, MIT, Cambridge, MA, USA.

[Sarin2009] Sunil Sarin. 2009. Dynamic Transaction Serializability in Netezza Performance Server. *New England Database Day*, page `http://db.csail.mit.edu/nedbday09/htdocs/papers.php`, January.

[Shi and Perrizo2002] Victor T.-S. Shi and William Perrizo. 2002. A new method for concurrency control in centralized database systems. In Rex E. Gantenbein and Sung Y. Shin, editors, *Computers and Their Applications*, pages 184–187. ISCA.

[Stearns *et al.*1976] R. E. Stearns, P. M. Lewis, and D. J. Rosenkrantz. 1976. Concurrency control for database systems. In *SFCS '76: Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 19–32, Washington, DC, USA. IEEE Computer Society.

[Sullivan2003] David Gerard Sullivan. 2003. *Using probabilistic reasoning to automate software tuning*. Ph.D. thesis, Harvard University, Cambridge, MA, USA. Adviser-Margo I. Seltzer.

[Tada *et al.*1997] Harumasa Tada, Masahiro Higuchi, and Mamoru Fujii. 1997. A concurrency control algorithm using serialization graph testing with write deferring. *Transactions of Information Processing Society of Japan*, 38(10):1995–2003, October 15.

[Transaction Processing Performance Council2005] Transaction Processing Performance Council. 2005. TPC-C Benchmark Specification. http://www.tpc.org/tpcc.

[Weikum and Vossen2002] G. Weikum and G. Vossen. 2002. *Transactional Information Systems: Theory, Algorithms and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, California.

[Wu and Kemme2005] Shuqing Wu and Bettina Kemme. 2005. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 422–433.

[Yang2007] Yang Yang. 2007. The adaptive serializable snapshot isolation protocol for managing database transactions. Master's thesis, University of Wollongong, NSW Australia.

Index