PERFORMANCE OF PROGRAM MODIFICATION TECHNIQUES THAT ENSURE SERIALIZABLE EXECUTIONS WITH SNAPSHOT ISOLATION DBMS

TECHNICAL REPORT 677

MOHAMMAD ALOMARI, ALAN FEKETE, AND UWE RÖHM

AUGUST, 2011
Performance Of Program Modification
Techniques That Ensure Serializable Executions
with Snapshot Isolation DBMS

Mohammad Alomari, Alan Fekete, and Uwe Röhm

School of Information Technologies, University of Sydney 2006 Australia
miomari@it.usyd.edu.au, {alan.fekete, uwe.roehm}@sydney.edu.au

Abstract. Snapshot Isolation (SI) is a multiversion concurrency control that has been implemented by several open source and commercial database systems (Oracle, PostgreSQL, Microsoft SQL Server). The main feature of SI is that a read operation does not block a write operation and vice versa, which allows higher degree of concurrency than traditional two-phase locking. SI prevents many anomalies that appear in other isolation levels, but it still can result in non-serializable executions, in which database integrity constraints can be violated. Several techniques are known to modify the application code in order to ensure that every execution is serializable on engines running SI. Each modification technique introduces some conflicts, and so prevents certain interleavings, without altering the functionality of each application program. In following one of the possible techniques of application modification, there are choices that must be be made, between different conflicts where interleavings are affected.

This paper investigates the performance impact of the different techniques, and different choices of modifications, that can ensure that all executions are serializable. We here propose one technique called 'External Lock Manager' (ELM) which introduces conflicts in a separate lock-manager object rather than through SQL statements in the DBMS. We find that ELM performs well under a range of situations, with peak performance close to SI no matter which conflicts are chosen to be introduced.

1 Introduction

One of the main reasons that application developers use databases is to maintain the integrity and consistency of their data. Transactions are typically viewed as sequences of read and write operations that run as one unit, and the interleaved operations of read and write requests for a concurrent execution of transactions is called the schedule. Serializability of any interleaving is a notion of correctness, based on whether that schedule is equivalent to some serial one. The essential property is that all integrity constraints are valid at the end of a serializable execution (so long as each transaction separately is written to maintain the
Many database vendors provide two-phase locking (2PL) to ensure serializability. The data integrity guaranteed by 2PL comes at a considerable cost in performance, as a read operation is delayed until the commit of a concurrent transaction which wrote the same item, and as a write operation is delayed until there is no active transaction that has read the item.

Other mechanisms for concurrency control can be more permissive than 2PL, offering the application developer the possibility of having more concurrency between transactions, and higher overall performance, with the risk that some anomalies might occur that might affect the consistency of the data. Gray et al [1] introduced several locking mechanisms which are similar to 2PL, but keep some locks for shorter periods (or not at all). These mechanisms are said to offer lower isolation level than Serializability, and the ANSI/ISO SQL standard defines commands to indicate that a particular transaction should use a given isolation level.

Berenson et al [2] defined a new concurrency control algorithm called Snapshot Isolation (SI), variants of which are implemented in platforms such as Oracle, PostgreSQL and Microsoft SQL Server. SI saves the old versions of any updated data item, in order to use these later to satisfy read requests, with each transaction seeing each data item in the version that committed most recently before the start of the reading transaction. SI does not allow inconsistent read anomalies, and it also prevents lost updates since the First Committer Wins (FCW) rule prevents two concurrent update transactions from modifying the same data item.

However, non-serializable executions are possible with SI, and data can be corrupted so that (undeclared) integrity constraints are violated. In particular, [2] shows an anomaly called Write Skew that is possible with SI. Fortunately some applications have specific patterns of data access, so that for these particular sets of programs, all executions are serializable even if SI is the concurrency control mechanism. The TPC-C benchmark has this property. There is a theory which allows one to prove this situation, when it occurs [3]. To apply the theory, the DBA looks at the transaction programs, finds conflicts between the programs, and represents these conflicts in a graph called Static Dependency Graph (SDG). An SDG without any cycle containing two consecutive edges of a particular sort (called vulnerable edges) indicates that every execution of the programs under SI will be serializable.

In [3], two techniques were described, that can take a given collection of programs and modify them, so that serializable execution under SI is ensured. The modifications place extra SQL statements in some programs; this will introduce extra conflicts between them, but they do not change the semantic effect of any program. These modification techniques are called “Promotion” and “Materialize”. To guarantee serializable executions, Promotion or Materialize must be done for an appropriate set of edges in the original SDG; the essential requirement is the set of edges (for which conflict-introduction is done) must include
at least one from every pair of vulnerable edges that are consecutive within a cycle.

We here offer another technique to ensure serializability with SI, through introducing lock conflicts outside the DBMS as a way to control concurrent execution. We suggest coding an application-level component called “External Lock Manager” (ELM). ELM provides an interface for a transaction to set an exclusive lock; a subsequent request by another transaction to set the same lock will be blocked until the lock holder releases the lock. To introduce a conflict along an edge which is vulnerable in the SDG, we place at the start of each program, a call to ELM to set a lock. The lock being requested should be such that the transactions will try to get the same lock, in those cases where their parameters give rise to conflict between data accesses that makes for a vulnerable edge.

Note that ELM is different in several ways from using traditional two-phase locking. Those transactions that are not involved in chosen edges do not set locks at all. There are only exclusive locks, no shared locks. Even if a transaction touches many objects, it may need to lock only one or a few string values. All locking is done at the start of the transaction, before any database activity has occurred; together with resource-ordering in obtaining locks, we can prevent any deadlock involving ELM.

Thus, there are many ways in which a DBA or application developer can ensure that all executions are serializable for their applications, when running on a platform providing SI. They must decide which modification technique to use (Promotion, Materialize, or ELM), and also they must choose the set of edges of the SDG on which extra conflicts will be introduced. In general, there are many different subsets of the edges in the SDG that include one from every pair of vulnerable edges that are consecutive in a cycle, and so modification of these are sufficient to guarantee serializable execution. There might be many such sets of edges which are minimal (no subset of the set is sufficient) and finding such a set with the fewest number of edges is NP-hard [4].

This paper studies the performance impact of these alternative ways available to a DBA or developer, who wishes to guarantee that their system will have serializable executions. We look at two different microbenchmarks (sets of application programs) under a range of situations, varying the platform, the ratio of read-only transactions, the amount of contention over data items, and so on.

This paper is organized as follow: Section 2 covers the background concepts and other material from previous literature that is related to the paper. Section 3 presents our new algorithm called the External Lock Manager (ELM) with details. We discuss design choices and the implementation of the ELM algorithm. Fault tolerance is also discussed briefly. In section 4, we give a detailed discussion of how the experiments are designed and run. Section 5 presents the experiment results comparing various techniques that ensure serializable executions under SI. In section 6, we conclude the paper with summary of our work and findings.
Acknowledgement: Preliminary versions of parts of this paper have appeared as [5, 6]. This work is supported in part by Australian Research Council grant DP0987900. Michael Cahill helped greatly with setting up the experiments.

2 BACKGROUND

2.1 Snapshot Isolation (SI)

Snapshot Isolation is a multi-version concurrency control mechanism used in databases. It was introduced in 1995 in [2]. Among popular database engines (commercial and open source) that use SI are Oracle, Microsoft SQL Server and PostgreSQL. One of the most important properties of SI is the fact that a read does not block any write, and write does not block any read, by allowing transactions to access previous versions of each record, rather than always accessing the most recent state of the record. This property is a big win in comparison to systems that implement the traditional Two Phase Locking (2PL) mechanism, where update operations could block readers for a lengthy duration. Each transaction in a SI-based system has a logical timestamp, indicating when the transaction started. Any read done by the transaction T, sees the state of the data which reflects exactly the other transactions that committed before $start-time(T)$.

Similarly, any where clause evaluated in a statement of T uses values from this snapshot. (There is an exception, that T sees changes it has made itself.) This already prevents the inconsistent-read anomaly: T can never see part but not all of the effects of another transaction. It also prevents traditional phantoms, where a predicate is evaluated twice with different results.

First Committer Wins (FCW): The other essential in the SI mechanism is that whenever there are two concurrent transactions (ie when the interval $[start-time(T), commit-time(T)]$ overlaps with the interval $[start-time(U), commit-time(U)]$) and both have written to the same item, at least one of them will abort. This prevents the lost update anomaly.

SI was presented in [2] as a particular concurrency control mechanism, but Adya et al [7] have offered an abstract definition of the isolation level it provides (by analogy to the way true serializability is provided by two-phase locking, but can be characterized more abstractly as the absence of cycles in the serialization graph). SI has been widely studied as a help in managing replicated data [8, 9, 10]; it is much cheaper to obtain global SI than to ensure true global serializability. Other work on replication has used slight variants of SI, following the same principles but relaxing the choice of start-timestamp [11, 12].

However, the improved performance of SI compared to 2PL brings with it a risk of corrupting data integrity. While SI completely avoids the four extended ANSI SQL anomalies, it does not guarantee serializability. As shown in [2], SI could allow non-serializable executions as we discuss in Section 2.3 below.
2.2 Implementing SI

SI is usually based on having a system that keeps versions\(^1\) of each item. When a transaction modifies data item \(x\), rather than actually altering the value stored for \(x\), instead, the previous value remains, and as well, another version is created with the new value. Each version is given a version number, that identifies the transaction that produced the version. Usually, a system-wide monotonically increasing counter is used. This is incremented at every transaction commit, and the transaction that commits gets the new value as its “commit timestamp”. The version number assigned to a version of some data item is just the commit timestamp of the transaction that produced the version. When a transaction starts, the value of the counter becomes the “snapshot timestamp” of that transaction, and whenever the transaction reads a data item \(x\), the version that is returned is chosen from among all committed versions of \(x\), to be that version whose version number is as large as possible but no larger than the snapshot timestamp of the reading transaction.

The first committer wins rule can be implemented without write locks using an optimistic strategy, and deferring the installation of new versions to the end of the transaction. When a transaction completes, it is validated. Validation of transaction \(T\) is successful only if \(T\)’s snapshot timestamp is greater than or equal to the current version number of each item that \(T\) has updated. In that case, \(T\) commits, and new versions are installed for all the items modified by \(T\), each with the newly incremented counter as version number. Otherwise (when an updated item has already a version with version number greater than the snapshot timestamp of \(T\), produced by a concurrent transaction that already committed) \(T\) must abort, and no new versions are installed.

Another possible implementation of FCW can be done with versions installed during transaction execution rather than at transaction completion. In this implementation technique, exclusive write locks\(^2\) are used (but in a different way from 2PL). Each transaction sets an exclusive write-lock on data it modifies, and the system aborts a transaction if it ever tries to write to an item whose most recent version is not the one in its snapshot. Thus we can describe this as “First Updater Wins”. This is the approach taken by Oracle and PostgreSQL. In more detail, when we have transaction \(T_1\) requesting to write data item \(x\), then \(T_1\) must try to obtain an exclusive lock on \(x\); if another transaction \(T_2\) is currently holding the lock on \(x\), then \(T_1\) must wait until \(T_2\) completes:

- If \(T_2\) commits, then \(T_1\) is aborted.
- If \(T_2\) aborts, then \(T_1\) gets the lock on \(x\).

---

\(^1\) The versions may be stored on disk, all in the same format; alternatively, in some systems only the latest version is stored in the normal way on disk, and other versions are kept in the undo log, and reconstructed as needed.

\(^2\) Note that SI does not use read-locks at all, and a read is never blocked by any concurrent transaction.
Once $T_1$ has obtained the exclusive lock on $x$, its fate depends on the highest version number belonging to a non-aborted version of $x$.

- If the highest version number of $x$ is less than or equal $T_1$’s snapshot number, the write by $T_1$ can occur, creating a new version of $x$. When eventually $T_1$ commits or aborts, this version is either assigned a version number that reflects the commit timestamp of $T_1$, or else the version is marked as aborted.
- If the highest version number of $x$ is greater than $T_1$’s snapshot number, $T_1$ is aborted since a concurrent transaction already wrote $x$, committed and released the locks.

2.3 Snapshot Isolation Anomalies

Snapshot Isolation may allow non-serializable executions with some anomalies.

**SI Write Skew Anomaly.** Write skew is an anomaly identified by [2] that could occur under SI, and in which one can violate data integrity. It happens when we have two or more concurrent transactions ($T_1$ and $T_2$) where each changes a value that the other read. When neither sees the result of the other’s update, we have ”Write Skew”. FCW doesn’t prevent this, since different items are changed in each transaction. An example schedule is shown below.

$T_1$: R($x_0$) R($y_0$) W($x_1$) C1.

$T_2$: R($x_0$) R($y_0$) W($y_1$) C2.

To see how an (undeclared) data integrity constraint can be violated, consider the well-known bank scenario. Suppose we have two values $x$ and $y$ representing checking account balances of a couple at a bank, with an invariant that $x+y > 0$. The bank’s business logic may permit either account to be overdrawn, as long as the sum of the account balances remains positive. Suppose that initially $x = 50$ and $y = 50$. Under SI, transaction $T_1$ reads $x$ and $y$, then subtracts 90 from $x$, assuming it is safe because the two data items added up to 100. Transaction $T_2$ concurrently reads $x$ and $y$, then subtracts 80 from $y$, assuming it is safe for the same reason. Each update is safe by itself, but when both occur (as is allowed by SI) we will end up in violation of the invariant $x+y > 0$. Unfortunately, this problem will not be detected by First Committer Wins because two different data items were updated.

**Read-Only Anomaly.** Another kind of non-serializable execution that can occur with SI was identified by Fekete et al [13]. Here data is not corrupted (because the collection of update transactions is serializable), but a read-only transaction reports information that could not occur in any serial execution that gives rise to the actual final state.

An illustration of this from [3] is where we have $x$ and $y$ as two data items representing checking account balance and saving account balance. In this system, the business logic is that a withdrawal is allowed to make $x+y$ negative, but in that case an extra dollar is withdrawn, as penalty fee. Consider this sequence of operations, from an initial state with both items value is zero. $T_1$ tries to deposit
Performance of Program Modification for Serializability Over SI

20 to saving balance $y$, and $T_2$ subtracts 10 from the checking account $x$.

$T_1$: $\text{R}(y=0) \text{W}(y=20)$ C1.

$T_2$: $\text{R}(x=0) \text{R}(y=0) \text{W}(x=-11)$ C2.

$T_3$: $\text{R}(x=0) \text{R}(y=20)$ C3.

The anomaly arises here as read-only transaction $T_3$ has seen $x=0$ and $y=20$, and this situation does not happen in any serializable execution that produces the observed final state $x=-11, y=20$. If 20 were added to $y$ before 10 were subtracted from $x$ in a serial execution, no penalty charge could occur (so the final result of $x$ should be -10). On the other hand, if 10 were subtracted first in a serial execution, the system would not be in a state where $T_3$ sees $x=0, y=20$.

**Phantom Anomaly.** There is no agreed-upon formal definition in the literature of the concept of “phantom”. The term is often illustrated by examples where a transaction executes the same SELECT statement twice, and the second execution can return a result set containing a different set of rows. In this narrow definition, Phantoms are not possible under snapshot isolation since all reading in a transaction uses the same snapshot, so a repeated query will return the same set. A broader sense of the term [14] also covers cases where a single SELECT statement’s predicate misses an item that is being inserted or deleted by a concurrent transaction. This situation can occur under SI, and again non-serializable execution and data corruption can be observed. For example, Jorwekar et al [4] observed a real system with application code that generated an identifier by selecting the maximum already present, and adding to that. In a serial system, the identifiers would be unique, but under SI, executions were found with duplicate identifiers.

### 2.4 Applications that are always serializable under SI

The experts in the Transaction Processing Council could not find any non-serializable executions when the TPC-C benchmark [15] executes on a platform using SI, and so Oracle7 was allowed to be used in benchmarks. This leads one to explore what features of a set of programs will ensure all executions are serializable (when the DBMS uses SI). The first example of a theorem of this sort was in [16], and a much more extensive theory is in [3]. The latter paper proves that the TPC-C benchmark has every execution serializable on an SI-based platform. Jorwekar et al [4] have shown that one can automate the detection of some cases where the theory of [3] holds. Fekete [17] deals with platforms (like SQL Server 2005) which support both SI and conventional two-phase locking, by showing how one can decide which programs need to use 2PL, and which can use SI. Earlier, Bernstein et al [18] showed how to prove that certain programs maintain a given integrity constraint when run under a variety of weak isolation levels, including SI. All of these theories involve static analysis, that is, one considers the code of the application programs that can run in the system, and this happens at design or compile time. This is in contrast to traditional serializability theory which dynamically looks at the sequence of events that arise...
in transaction instances (which are generated from the code during a particular execution).

The key result of [3] is based on a particular graph, called the Static Dependency Graph (SDG). This has nodes which represent the transaction programs that run in the system. There is an edge from program P to program Q exactly when P can give rise to a transaction T, and Q can give rise to a transaction U, with T and U having a conflict (for example, T reads item x and U writes item x). Different types of edges are defined:

1. Vulnerable edge (RW): We say that the edge from P to Q is vulnerable if P can give rise to transaction T, and Q can give rise to U, and T and U can execute concurrently with a read-write conflict (also called an anti-dependency); that is, where T reads a version of item x which is earlier than the version of x which is produced by U. We represent a vulnerable edge with a dashed arrow as $P \rightarrow Q$.

2. Write Dependency edge (WW): We say that the edge from P to Q is Write Dependency if P can give rise to transaction T, and Q can give rise to U, and T and U can execute with a write-write conflict. If the transactions are concurrent, one of them should abort as result of FCW rule. Since a write-dependency among transaction programs from P to Q implies also one from Q to P, we represent a write dependency edge with a double-headed solid line as $P \leftrightarrow Q$.

3. Read Dependency edge (WR): We say that the edge from P to Q is Read Dependency if P can give rise to transaction T, and Q can give rise to U, and T writes a value $x$ and commits, then later, U reads $x$. We represent this with a solid single arrow as $P \rightarrow Q$; note however that the same notation is also used where there is a non-vulnerable RW edge.

Within the SDG, [3] defines that a dangerous structure occurs 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), Figure 1 shows a dangerous structure. The main theorem of [3] is that if a SDG has no dangerous structure, then every execution of the programs is serializable when run on a DBMS using SI for concurrency control.

### 2.5 Options to ensure Serializability

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 we want that the modifications do not alter the essential functionality of the programs. In [3], several such modifications were proposed. 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 get an additional statement which modifies 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 becomes 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. [4] showed that choosing a minimal set of appropriate edges is NP-hard.

Here we give more detail on the different techniques that were proposed by [3] to ensure serializability when using snapshot isolation. These techniques each allow further choices of which edges to make not vulnerable; our experiments in Section 5 will compare the performance impacts of all these choices that the DBA can make.

**Materialization:** To make an edge not vulnerable by materialization, we introduce an update statement into each program involved in the edge. The update statement modifies a row of the special table Conflict, 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 (because the First Updater Wins property, or the First Committer Wins property, insists on this). However, we usually try to introduce contention only if it is needed. Thus if we have programs P and Q which have a read-write conflict when they share the same value for some parameter x, then we can place into each a statement

1- UPDATE Conflict
2- SET val = val+1
This gives a write-write conflict only when the programs share the same parameter x, which is exactly the case where we need to prevent committing both of the concurrent transactions.

**Promotion:** To use promotion to make an edge from P to Q not vulnerable, we add to P an update statement called an *identity write* which does not in fact change the item on which the read-write conflict occurs; we do not alter Q at all. Thus suppose that the read-write dependency is that Q modifies some item in T for which a condition C holds, and P contains

```
1- SELECT ...
2- FROM T
3- WHERE C
```

To use Promotion, we include in P an extra statement

```
1- UPDATE T
2- SET col = col
3- WHERE C
```

Once again, the First Updater Rule will ensure that (the modified) P and Q do not run concurrently except in the situations where parameter values mean that there is not a read-write conflict either. 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. Fortunately this is rare in typical code, where most predicates use a primary key to determine which record to read.

Another related approach to promotion is by replacing the SELECT statement (that is in a vulnerable read-write conflict) by Select..For Update (SFU). This does not modify the data, but it is treated for concurrency control in Oracle like an Update, and the statement cannot appear in a transaction that is concurrent with another that modifies the item. In other platforms, such as PostgreSQL and SQL Server, this statement prevents some but not all of the interleavings that give a vulnerable edge. In particular, in PostgreSQL the interleaving `begin(T) begin(U) read-sfu(T, x) commit(T) write(U, x) commit(U)` is allowed, even though it gives a vulnerable rw-edge from T to U. Oracle supports another version of SFU Select..For Update NOWAIT. If NOWAIT is not specified and a row to be locked is locked by another transaction, Select..For Update will wait indefinitely until the lock is released. If NOWAIT is specified and a row to be selected is locked by another transaction, the Select..For Update will return immediately with a "ORA-00054: resource busy and acquire with NOWAIT specified" error.

**Using 2PL:** Another possible way to modify application programs is provided by [17] which defines a node as a pivot, if it has incoming and outgoing vulnerable
edges, and the path from to original node is a chord-free-cycle. Transaction U in Figure 1 is an example of a pivot. Fekete [17] shows that if every pivot transaction is run with 2PL, rather than SI, then all executions will be serializable. Allocating each transaction with the appropriate isolation level does not require any changes to the application code, or recompilation. It can be done at run-time, entirely in the client. Unfortunately, many platforms, including PostgreSQL and Oracle, do not offer declarative use of conventional 2PL. In these platforms it is possible to explicitly set locks, and so one can simulate 2PL for pivot transactions; however the explicit locks are all of table granularity and thus can be expected to have very poor performance. Even when we can make use of declarative 2PL with row level locks, it has been found [19] that there is significantly lower throughput than can be obtained with promotion and materialize. Therefore, we do not consider the use of 2PL for pivots in our experiments later in this paper.

2.6 Changing the concurrency control mechanism

Another approach to ensure serializable executions is given by Cahill et al [20, 21]. This proposes a new concurrency control algorithm (called SerializableSI) which is similar to SI but dynamically checks for pairs of dependency edges among concurrent transactions, so that it does not allow anomalies. Another new algorithm called PSSI is based on testing for dependency cycles and was given by Revilak et al [22]. These new algorithms are incomparable in functionality to the techniques considered here in this paper. SerializableSI or PSSI require modifying the engine internals, whereas promotion, Materialize or our new ELM technique can work with a SI-supporting dbms out of the box. On the other hand, all of the techniques that we consider require analyzing and then modifying the application code, and therefore we need knowledge of the complete set of application programs that could be run on the platform; SerializableSI and PSSI work with unmodified applications, and can even support ad-hoc transactions.

3 THE EXTERNAL LOCK MANAGER (ELM) TECHNIQUE

In this section we introduce a new technique called “External Lock Manager (ELM)” that ensures serializability with snapshot isolation. We extend the overall system of application clients and DBMS with an object which manages locks (unlike traditional lock-managers, the ELM lock manager can sit outside the DBMS). In order to introduce a conflict between application programs P and Q, the DBA modifies the chosen programs (but not other programs), so that each obtains an ELM lock before beginning a database transaction, and it releases the ELM lock after the database transaction completes or aborts.
Our proposed ELM approach introduces an additional software component to manage locks. In any application program for which a conflict is introduced, the client begins by sending a request to the ELM component in order to request an appropriate lock or locks. The client blocks until it receives a reply from the ELM component, granting the lock. Once the request is granted, the client can then invoke the rest of the business logic for the application program, for example, by calling a stored procedure on the database server. Finally, after the transaction has completed in the database, the client again sends a message to the ELM component to release the lock(s) it holds. This interaction is shown in Figure 2. The labels 1, 2 and 3 on the message exchanges indicate the order of events within one program (1-Sending lock request to ELM, 2-Communicating with database server, 3-Releasing locks).

Let’s suppose that the DBA has decided to introduce conflict on a vulnerable edge in the SDG that goes from program P to program Q. As described in section 2, the definition of vulnerable edge says that there can be transactions T and U, where T arises from invoking P and U arises from invoking Q, such that there is a read-write dependency from T to U, and also such that T and U can execute concurrently. The DBA will introduce into P a call to set a lock in ELM, and a later call to release the lock; these calls should completed surround the database transaction T that P invokes. For example, if P invokes a transaction through a JDBC call to a stored procedure, the lock request will precede the call and the lock release will follow it; if P contains several separate SQL statements that make up T, we place the lock request before the first SQL statement, and the lock release after the last SQL statement in the program. Similarly, program Q is modified so that a lock request and release surround the whole invocation of transaction U.

In order to introduce the necessary conflicts to remove the vulnerability of an SDG edge, we surround transactions with ELM lock-set and lock-release calls. However, we only need to make sure that there are lock conflicts, in those cases where the transactions have a read-write dependency. In many programs, the particular items accessed depend on parameters of the application program. For example, a program representing depositing money in a bank account may take the account number as parameter. We want the ELM locking to be fine-grained,
that is, we prefer that the ELM locks do not block one another unless the two programs are actually dealing with the same data object (e.g., the same account); two programs that deal with different data items should set different locks (and thus they can run concurrently). By appropriate choice of the lock to request (as discussed in the next subsection), we can achieve carefully targeted exclusion. If the transaction program logic is too complex, and the DBA can not identify an appropriate lock that will conflict when necessary, then we suggest reversion to coarse-grained ELM locks, which are easy to determine from static analysis and which do not require any form of predicate analysis.

It is our desire for fine-grained locking that leads us to provide our own ELM component, rather than the locking available directly in the database engine. While most platforms use record-level locks for automatic locking, they typically offer user-controlled locks only at table-granularity (e.g., SET TABLE LOCK ON tablename).

3.2 Lock Alternatives

The performance of ELM depends dominantly on the specific details of the locks we use and the frequency of conflict this leads to. In this section we discuss alternative techniques to choose what exactly will be locked. We use the following example to describe each technique.

Example: Let us assume that we have two transaction programs $P_1$ and $P_2$. Suppose $P_1$ has a parameter $x$, and $P_2$ has a parameter $y$, and both of them access table $Table_1$. $Table_1$ has two columns ($tabID$:integer, $value$:real). Let $T_1(x)$ denote the transaction that arises when $P_1$ is run on the parameter $x$. In this example, $T_1(x)$ reads a value from $Table_1$ using the parameter $x$ as primary key in the where clause. The essential SQL in $P_1$ is:

```
1- ...
2- SELECT val
3- FROM Table1
4- WHERE tabID=x
5- COMMIT;
```

Similarly, $T_2(y)$ updates $Table_1$ using the parameter $y$ as primary key. Its SQL is:

```
...
1- UPDATE Table1
2- SET val=val+1
3- WHERE tabID=y
4- COMMIT;
```

When we use ELM to remove the vulnerability from the edge $P_1$ to $P_2$, then we need to make sure that whatever we lock in each transaction will stop them
running concurrently when they in fact conflict that is, when \( x = y \). Here are some techniques to achieve this.

**Edge-Name Technique:** One technique is to lock the edge’s name. Edge name could be the concatenation of names of the programs that joined the chosen edge. The edge name can be \( P_1 + P_2 \). So when we have two transaction \( T_1(x) \) and \( T_2(y) \) running concurrently, let us say \( T_1(x) \) starts first, then \( T_1(x) \) will acquire the ELM to lock the edge’s name \( (P_1 + P_2) \) and hold this until the time of commit. Since \( T_2(y) \) will try to acquire the same lock, \( T_2 \) will wait in the queue until \( T_1 \) commit and release the lock. This technique will stop \( T_2(y) \) conflicting transactions and \( T_1(x) \) running concurrently. It also stops two instances of \( P_1 \) running together (and similarly it prevents concurrent \( T_2 \) transactions). However, this technique is not fine-grained and it reduces the number of concurrent transactions, since even when \( x \neq y \), so \( T_1(x) \) has no conflict with \( T_2(y) \), \( T_1(x) \) will block \( T_2(y) \). This can be considered a false conflict.

**Item-Name Technique:** An alternative technique is to lock the common column name that the transactions have a conflict on. Transactions with conflicts share the same data item in the schema. Using the previous example, \( T_1(x) \) and \( T_2(y) \) access \( \text{tabID} \) which is the same item name (field name) in the schema. Now if \( T_1(x) \) acquires the ELM to lock the item name \( (\text{tabID}) \), and \( T_2(y) \) concurrently tries to acquire the same lock, therefore \( T_2(y) \) will be blocked until \( T_1(x) \) commits and releases the lock. Now this delay will ensure that \( T_1(x) \) and \( T_2(y) \) can not run concurrently. Unfortunately, this technique can have false conflicts as in the previous technique, since it prevents concurrency even when \( x \neq y \).

**Parameter-Value Technique:** A third technique is what we actually use in most of our experiments in this paper. Here a transaction locks on the transaction’s parameter value. When two transactions have the same parameter values then we can use these values to stop concurrent transactions that can cause non-serializable executions. Assume \( T_1(x) \) accesses the ELM and locks the value of parameter \( x \), and then if \( T_2 \) concurrently tries to lock the value of the parameter \( y \), then:

- If \( x = y \), then the transaction who started later will wait until the earlier transaction commits and releases the lock.
- If \( x \neq y \), then both transactions can acquire the locks and invoke their business logic to the database without any delay.

But what about in the case where the transaction has more than one parameter? Since our aim is to increase the number of serializable concurrent transactions inside the database, therefore we should try to find the minimum set of parameters that need to be locked to stop non-serializable executions in any history. For example suppose \( T_1 \) passes different types of parameters (e.g., \( x_1, x_2, \ldots, x_n \)) and \( T_2 \) passes another set of parameters (e.g., \( y_1, y_2, \ldots, y_m \)), and suppose \( T_1 \) and \( T_2 \) have only a conflict when \( x_1 = y_1 \) and \( x_2 = y_2 \), then we lock a minimum set of the parameters that can stop \( T_1 \) and \( T_2 \) from running concurrently. In this case \( T_1 \) could lock \( x_1 \) and \( T_2 \) lock \( y_1 \); alternatively \( T_1 \) could lock \( x_2 \) and \( T_2 \) could lock...
Finding a good set of parameters is quite easy with a simple set of parameters, as in the programs we use for benchmarking. However, if the parameters or programs are complex, finding the suitable locknames is more difficult. Further research is still needed for this to be effectively automated.

**Very-Fine-Granularity Technique:** The parameter-value algorithm described above is fine-grained but it does still allow some unnecessary conflicts. For example, suppose $T_1$ has a parameter $x$, and $T_2$ has a parameter $y$, and $T_3$ has a parameter $z$, and we want to make sure that $T_1$ and $T_2$ are not concurrent when $x=y$ (but they can run concurrently provided $x$ and $y$ differ), and $T_2$ and $T_3$ are not concurrent when $y=z$, but we do not need to introduce conflict between $T_1$ and $T_3$, perhaps because this edge is non-vulnerable in the original SDG. Our description above said that $T_1$ would request an ELM lock on $x$, $T_2$ would request an ELM lock on $y$, and $T_3$ would request an ELM lock on $z$. These locks conflict as required, but as well there will be a lock-conflict between $T_1$ and $T_3$ when their parameters agree. This can be avoided by using more complicated String values as the names to be locked, where the name encodes both the edge and the parameter. For example, we could have $T_1$ set a lock on the String which is a concatenation "$T_1$" + "$T_2$" + $x$, and $T_2$ sets two locks, one on the concatenation "$T_1$" + "$T_2$" + $y$, and the other on "$T_2$" + "$T_3$" + $y$; finally $T_3$ sets a lock on "$T_2$" + "$T_3$" + $z$. This would remove the vulnerability on the edge $T_1$ to $T_2$ when $x=y$, and on $T_2$ to $T_3$ when $y=z$, but they would not lead to conflict between $T_1$ and $T_3$.

To help DBA to understand the impact of these different lock alternatives, we studied the performance in section 5.1.

### 3.3 Proof of ELM Serializability

The proof that the ELM algorithm ensures serializable execution is immediate from the main theorem of [3].

**Proof.** Suppose we have a history $H$ execution under SI, and suppose that history is not serializable. Then $H$ has a dangerous structure such as $T_1 \rightarrow T_2$ is *rw-vulnerable edge* and $T_2 \rightarrow T_3$ is another *rw-vulnerable edge*, and there is a path between $T_3$ and $T_1$ (or $T_1$ and $T_3$ are identical). Then if we stop $T_1$ and $T_2$ from running concurrently (or $T_2$ and $T_3$) by blocking one of them in case of conflict (until the lock is released), then the chosen edge is no longer vulnerable, as a result, the definition of dangerous structure is no longer applicable to the graph. Using the theorem [3] which insists that the absence of dangerous structure ensures serializability with SI, we guarantee that the history $H$ is serializable under SI using ELM.

### 3.4 Design Features

In our design, we assume a client-server or multi-tier architecture, with a separate machine acting as the database server, invoked across a network by clients. One
way to execute the business logic is to create stored procedures on the database server; thus each transaction involves a single request/response exchange between the client and the server. Another way of executing the business logic is with multiple round-trips; here the client sends multiple requests, and receives multiple responses, to execute one transaction. In our experiments, we consider the business logic as stored procedure on a database server.

We believe that introducing conflict on an edge by using ELM locks has considerable potential advantages compared to the previous approaches such as Materialize or Promotion, where conflict is introduced by additional SQL statements that lead to updates in the database (so that the conflict is provided by the FCW mechanism of SI). These benefits are:

**Logging Cost:** Data modifications are recorded in a data structure called the log to ensure Atomicity. The log is a sequence of log records, recording all the update activities in the database (permanently). These records are used later in case of any type of failure [23]. The previous techniques (Materialize and Promotion) introduce update statements, and thus need to write a log record to disk during the life of transaction. Logging increases the number of I/O operations needed, and that reduces the overall performance. Over the last decade CPU speeds have increased dramatically while disk access times have only improved slowly and this trend is likely to continue in the future and it will cause more and more applications to become disk bound [24]. ELM involves no change at all in the database server. Also ELM does not need to preserve the previous status of transactions locks to perform correctly. Therefore, ELM does not cause any additional logging even on the ELM system.

**Resource Management:** A second benefit of the use of ELM locks is that one of a pair of conflicting transactions may delay, being blocked while waiting for the ELM lock. In contrast, in Promotion or Materialize, the conflict leads to one transaction aborting, and restarting after the other has finished. Thus ELM avoids a lot of wasted work in transactions that eventually abort. Also, the blocking that occurs in ELM happens before the database transaction starts, and so there are no database resources being occupied while a program waits. It is important that we understand that ELM differs from traditional database locking, and thus it should not have the poor performance often experienced by 2PL. The most important difference is that in ELM, we do not lock every item that is accessed, and indeed many transactions operate without any locks at all. Locks are only set by the transactions involved in the set of vulnerable edges that the DBA has chosen for conflict-introduction, and even then, the requested lock is chosen so that it will collide with the other transaction involved in that edge in those situations when the parameter values require conflict. Since only a few locks are set, and there are only exclusive locks, we do not need to be concerned with lock mode upgrade, hierarchical locking, etc.

**Easy Deadlock Avoidance:** Traditional deadlock scenario may develops between two update transactions $T_1$ and $T_2$. Assume $T_1$ holds an exclusive lock on $x$ and $T_2$ holds an exclusive lock on $y$. Then, $T_2$ tries to update $x$ and $T_1$
tries to update $y$. Neither $T_1$ nor $T_2$ can proceed as each is waiting for the other. Such scenario can occur when we try to promote an edge that is part of a write skew anomaly. Actually any suggestion of blocking in a system raising fears of deadlock in the minds of experienced developers. In ELM, however, we can make sure that our proposal never introduces deadlock. We first observe that because each application obtains any ELM locks before starting the database transaction, no thread can possibly be holding any database resources while waiting on a queue in ELM (that is, no waiting cycle can go between the DBMS and the ELM subsystems). Thus the only risk of additional deadlock is within ELM itself, and this can be avoided through resource-ordering; that is, we code each application that needs multiple ELM locks, so that there is a canonical order in which they locks are requested (note that we know exactly which locks will be needed, based on the parameter values of the transaction, before requesting any ELM locks). If the application is coded this way, no deadlock can involve ELM. Thus we have not needed to introduce any deadlock detection mechanism nor any additional restart mechanism, outside what already exists in the DBMS engine.

Any design comes with some limitations and drawbacks. Some of these limitations of ELM are now described.

**Extra Communication:** Communication between the chosen programs and the ELM depends on the ELM location. If the ELM resides in the database server as extra component or in a middleware (see Section 3.5), then no extra communications are needed, since the programs already communicate with the database server and the middleware. The only case where communication need to be considered is when we have the ELM as separate component as shown in Figure 2. In this case, the ELM needs two extra communications: when the program acquires the lock and when it releases it. However, this extra communications are needed only by some of the programs not all of them. Different studies showed that in many systems, the network communication times are less than the disk access times [25, 26].

**Lock Overhead:** ELM uses exclusive locks during the transaction life, so the time to get these locks and to release them could be considered as extra computational operations. But we found in our experiments that the lock operation inside the ELM can be worthwhile, as they reduce the wasted work inside the database server.

**Component Failure:** System failures refer to main memory loss or corruption due to a power failure or an operation system failure. Media failure refers to damaged disks or other stable storage. The ELM server could be seen as an additional single-point-of failure for those transaction programs that require an ELM lock. There are also complexities that may arise with network losses.

---

3 Another approach to avoid deadlock is if each transaction requests its locks all at once in a single interaction with the ELM lock manager. We only implemented the first approach.
Extra coding and maintenance: Any extra component need to be coded and maintained to be integrated in a system correctly. The ELM basic idea is very simple derived from [27]. ELM uses techniques such as having a collection of waiting queues, indexed by a hash of the key being locked. ELM was developed once, so we do not need to re-write it every time we create a new database. Also, this component is shared among all applications (clients) for different platforms.

3.5 Location of ELM

There is nothing in our design that would limit where to place the ELM. Here we consider different locations as follows:

- **Separate Machine:** We can implement the ELM on a separate machine as in Figure 2. Each client can communicate with the ELM based on locking for the chosen edges. This design is easy to implement, and no modification to database source code is required. Failure of ELM node does not need recovery and undoing/redoing transactions in the database. We used this design in our experiments.

- **Middleware:** Another design inspiration can be by placing the ELM as a middleware. Each client is connected directly to the ELM. When a client sends a transaction (request) to database server, each transaction will be filtered in the ELM middleware based on the conflicts between these transactions. If a transaction has a conflict with other transactions, it will be delayed until the other transactions release the locks. One drawback to this design that each client needs to access the ELM middleware which could cause overloading to the middleware. Another drawback that is in case of middleware/ELM crash, clients need to wait until ELM is restarted. The previous design is potentially less harmful because some of clients need to wait not all. Note that implementing (coding and maintaining) the middleware is more complex than coding ELM itself as a separate node in the system.

- **Additional Component in the DBMS:** The ELM server could be seen as an additional single-point-of failure for those transaction programs that require an ELM lock. Thus we consider that database vendors could actually integrate the ELM functionality into their own DBMS code. Given that all transactions are implemented as stored procedures (which is a common practice nowadays) the ELM functionality could be leveraged to a fully declarative approach inside a DBMS: A corresponding DBMS could offer a declarative interface for the DBA to specify potential SI conflicts between stored procedures; these conflicts could then be enforced by the DBMS by automatically acquiring an ELM lock for the procedure’s argument values just before executing a marked transaction, and by automatically releasing this lock just after the commit. Most importantly, such an integrated approach would be fully declarative to the DBA, not requiring any changes to client code.
3.6 Prototype ELM Implementation

In our prototype implementation, we deal with client applications which are written in Java and invoke stored procedures in the database through JDBC. We have implemented the ELM through a software component written in Java, and we use Java Remote Method Invocation (RMI) for the message communication between the clients and the ELM component. The ELM object is a singleton instance of the LockManager class. At system startup, the client must execute the following:

```java
1- LockManager lmgr =
2- (LockManager)Naming.lookup
3- ("///LockManagerServer");
4- Locker locker = lmgr.newLocker();
```

We wrap the transaction call by a lock/locks request at the beginning and release lock/locks after the transaction commit. Here is what the code of the client looks like, after modifying it to use ELM in a case where more than one lock are required.

```java
1- cstmt = con.prepareCall("{call SomeTransaction(? , ?)}");
2- String[] keys = {key1, key2...keyn};
3- Lock[] locks = locker.getLocks(keys, false);
4- numlocked++ = 1;
5- try {
6-    cstmt.setString(1, key1);
7-    cstmt.setString(2, key2);
8-    .
9-    .
10-   cstmt.setString(2, keyn);
11-   cstmt.execute();
12-   con.commit();
13- } finally {
14-    for (int i = 0; i < locks.length; i++)
15-       locks[i].release();
16- }
```

ELM grants a lock through a factory method `Lock getLock(String name)`; the lock is released by calling the Lock instance’s method `void release()`. These locks are exclusive locks that stay during the life of the transactions.

3.7 Implementation of Lock Manager

Within ELM, locks are managed by the usual techniques from [27], such as having a collection of waiting queues, indexed by a hash of the key being locked. The main lock data structure in our design is "lock hash array". Each array entry
defined as a lock object. Then we use a hash function to assign a parameter to an entry in that array. So if we have two transactions with different parameters, the hash function will point them with high probability to a different array entries. But if the two transactions use the same parameters, then they will be hashed to the same entry causing one of them to wait using `wait()` function. Figure 3 shows that $T_1$ and $T_2$ hashed to the same entry, and $T_2$ starts after $T_1$, then $T_2$ has to wait or restart depending on the way we solve the conflict. The coding of the lock manager is somewhat simple, as we do not upgrade locks and we do not have multiple lock modes (only exclusive). Our implementation is deadlock free, and does not need to re-implement in case of using different databases or different platforms. We have implemented getting a lock in two different ways:

1. When a transaction requests a lock, if the lock is taken by other transaction, we can restart the request and submit it again (by using the flag boolean `noWait`). This approach has a drawback in some cases, since restarting the request needs extra communication (lock/release) between the client and the ELM component.

2. Instead, if the lock is taken by other transaction, the request can wait in the queue until the lock released. This approach can save and reduce the communication cost. Our experiments use this implementation.

ELM returns the locks in reverse order so they are fully nested. Here is what the code of the lock Implementation looks like.

```java
1- synchronized int lock(LockerImpl locker, String key,
2-     boolean noWait) {
3-     if (holder == locker) { /* we already have this lock */
4-         ++refcount;
5-     return LOCK_HELD;
```
while (holder != null)
try {
if (noWait)
return LOCK_FAILED;
this.wait();
catch (Exception e) {
// ignore it
}
holder = locker;
this.key = key;
refcount = 1;
return LOCK_NEW;
}

Releasing the locks can be much easier than getting the locks. Once the transaction commits inside the database, then the client who initiated that transaction communicates with the ELM to release the locks. The code of releasing the locks after the transaction commit in the database server looks like

```java
public synchronized void release() {
    //System.out.println("Unlocking " + key);
    if (refcount == 1) {
        holder.held.remove(this);
        holder = null;
        key = null;
        this.notify();
    }
    --refcount;
}
```

Now when we acquire multiple locks, we sort them in order to avoid deadlock. Sorting the parameters enforces the transactions to acquire the locks in order, so the conflict will arise earlier rather than later. Here is what the code to perform sorting looks like

```java
public Lock[] getLocks(String[] keys, boolean noWait)
throws RemoteException {
    /* Put the keys into hash bucket order to avoid deadlock. */
    Arrays.sort(keys, new Comparator() {
        public int compare(Object o1, Object o2) {
            int k1 = LockManagerImpl.getLockNum((String)o1);
            int k2 = LockManagerImpl.getLockNum((String)o2);
            return k2 - k1;
        }
        public boolean equals(Object o) {
```
In this prototype, we make a separate round-trip communication from client to the ELM machine for each request and each release. This is not a significant drawback in our design, since each transaction is usually protected by zero or one locks (or in a single case in our benchmarks it must obtain two locks). To improve performance with more complicated application logic, where several locks are needed to bracket a single database transaction, a production implementation would also allow batching, for example, there might be a single method which obtains locks on a whole collection of names (and returns only when all the requested locks have been obtained), and also the LockManager class itself could provide a method which releases all the locks held by the calling thread.

4 EXPERIMENTAL FRAMEWORK

This section describes the experimental setup we used to evaluate the different techniques that ensure serializability with snapshot isolation. We implemented a client-server system, where business logic is saved as stored procedures in the database server. We used multiple threads in a single test driver to simulate concurrent clients.

4.1 Software and Hardware

We use a local network with three dedicated machines for our experiments. All our experiments were performed on a dedicated database server running Windows 2003 Server SP2 that has 2 gigabytes of RAM, a 3.0 GHz Pentium IV CPU, and 2 IDE disks as separate log and data disks.

We use two DBMS platforms: one is PostgreSQL 8.2 which an open source database engine supporting SI (when a transaction is declared to be “SERIALIZABLE”), and the second DBMS is a commercial database engine. We do not compare the two platforms with one another; rather we use each platform separately to compare the behavior of the various techniques that ensure serializable execution with SI.

With PostgreSQL, we have made sure that the log disk on the database server has caching disabled; thus WAL disk writes are performed on the persistent storage itself, before the call returns to the DBMS engine. We configured commit-delay
Performance of Program Modification for Serializability Over SI

= 1ms, thus taking advantage of group commit. For the commercial platform, we did not alter the default settings, except to set the isolation level to "Serializable" which in fact causes the platform to use SI for concurrency control.

The additional component (the ELM instance) is running on a separate machine, equipped with 1 gigabyte of RAM, a 2.5 GHz Pentium CPU, and running Windows 2003 Server SP2. The lock-manager class is written in Java (SDK 1.5.0). Thus in experiments that measure performance of Promote or Materialize techniques, there is no overhead from the existence of the lock manager, on any machine where the application is doing work.

The test driver is running on a separate client machine that connects to the database server and ELM through Fast Ethernet. The client machine is running Windows 2003 Server SP2 and is equipped with 1 gigabyte of RAM and a 2.5 GHz Pentium CPU. The test driver is written in Java 1.5.0 and connects via JDBC to the database server, and via Java Remote Method Invocation (Java RMI) to the ELM component in experiments using ELM. The test driver emulates a varying number of concurrent clients (the multiprogramming level, MPL) using multiple threads. Our experimental system is a closed system: each client calls the database server to run the selected transaction and waits for the reply. If a transaction aborts, it is retried repeatedly; eventually it commits and then the client thread immediately (with no think time) initiates another transaction. Note that a single ELM is shared among all the clients. In our experiments these client threads are in a single JVM, but that is not intrinsic to the system design. The ELM design is not application specific, nor DBMS-engine or JDBC specific, and indeed one ELM component can be used by multiple application sets which are on different SI platforms.

Each experiment is conducted with a ramp-up period of 30 seconds followed by a one minute measurement interval. Each thread tracks how many transactions commit, how many abort (and for what reasons), and also the average response time. We repeated each experiment five times; the figures show the average values plus a 95% confidence interval as error bar.

4.2 Performance Metrics

The primary performance metric used throughout the paper is the transaction throughput, which is how many transactions commit per second. As MPL increases, we expect throughput to increase until some resource saturates; thrashing can lead to throughput which drops again as MPL increases even further. Note that for a given size of hotspot, there is an increasing probability of a transaction having a conflict with a concurrent transaction, as MPL increases.

The average response time, expressed in milliseconds, is also measured to reflect the difference between when a client first begins to process a new program, and when the transaction returns to the client following its commit; this includes any time spent waiting blocked in ELM, and also it includes the time spent while being restarted. Another useful measurement is the percentage of transaction
invocations that are aborted because of the FCW mechanism in the DBMS engine and indicated by return of an appropriate exception (in PostgreSQL this is a Serialization Failure Exception).

4.3 Workload Parameters

In our experiments, we vary a number of parameters that can affect the overall throughput. The main independent variable in experiments is MPL (Multi-Programming Level). MPL is the number of concurrent client threads which submit transactions; we vary it from 1 client to 30 clients. We generally found 30 threads sufficient to reach maximum throughput. I As well, we consider

– Data contention: To produce access patterns that are appropriately contended, we designed our clients to have 90% of the transactions access a portion of database called hotspot, and the other 10% access the rest of the database (database size - hotspot). We consider experiments when the hotspot which has size 100 rows (out of 20,000 in the whole table) as a low contention scenario, whereas a hotspot with 10 rows is a high contention scenario. The low contention scenario is more realistic than the high contention; the high contention hotspot is used to explore the robustness of the techniques under extreme conditions.

– Transaction Mix: Each experiment runs several different transaction programs, according to the particular benchmark application. Some of the programs are read-only. In some experiments each call chooses a transaction type with uniform probability, but other experiments give greater frequency for read-only transactions.

4.4 MicroBenchmarks

Usually, performance measurements use a standard benchmark such as TPC-C [15] which contains several transaction types, and which is carefully designed to exercise a range of features of a system. We cannot use TPC-C itself to compare different ways of making applications serializable, since TPC-C generates only serializable executions on SI-based platforms, as has been known since Oracle obtained benchmarks. This was proved formally in [3]. Thus in this paper we have used new mixes of transaction programs, which are contrived to offer a diverse choice among modifications that will ensure serializable execution on SI.

**SmallBank microbenchmark** SmallBank microbenchmark is based on the example of an SI anomaly from [13], and provides some functionality reflecting a small banking system, where each customer has a pair of accounts, one for savings and one for checking.
**SmallBank Schema:** Our proposed microbenchmark is a small banking database consist of three main tables: `Account(Name, CustomerID)`, `Saving(CustomerID, Balance)`, `Checking(CustomerID, Balance)`). The `Account` table represents the customers; its primary key is Name and we declared a DBMS-enforced non-null uniqueness constraint for its CustomerID attribute. Similarly, CustomerID is a primary key for both `Saving` and `Checking` tables. `Checking.Balance` and `Savings.Balance` are numeric valued, each representing the balance in the corresponding account for one customer.\(^4\)

**Transaction Mix:** The SmallBank microbenchmark runs instances of five transaction programs. These transactions are as follows.

*Balance*, or `Bal(N)`, is a parameterized transaction that represents calculating the total balance for a customer. It looks up `Account` to get the CustomerID value for N, and then returns the sum of savings and checking balances for that CustomerID. Program 1 shows an example of `Balance` transaction using PostgreSQL.

*DepositChecking*, or `DC(N,V)`, is a parameterized transaction that represents making a deposit on the checking account of a customer. Its operation is to look up the `Account` table to get CustomerID corresponding to the name N and increase the checking balance by V for that CustomerID. 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 the savings balance by V for that 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 to look up `Account` to get the CustomerID value for N, evaluate the sum of savings and checking balances for that CustomerID. 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.

**The SDG for SmallBank** Figure 4 shows the SDG for the SmallBank microbenchmark. We use dashed edges to indicate vulnerability, and we shade the

\(^4\) It is worth while to mention that the SmallBank schema is not a realistic example; in the account table name, rather than `CustID`, is the primary key. This means we can not have two people with same name as bank customers. Then, by making `CustID` the primary key of the checking and account table, it becomes impossible for a customer to have more than one checking account. Likewise there is a limit of only one saving account. However, this has no effect on the true purpose of the example for testing the ELM performance.
Program 1 Balance(N) transaction using PostgreSQL.

```sql
-- Function: sitest.balance(n character varying)
-- DROP FUNCTION sitest.balance(n character varying);
CREATE OR REPLACE FUNCTION sitest.balance(n character varying)
  RETURNS real AS
$BODY$
DECLARE
  cid INTEGER;
  a REAL;
  b REAL;
  total REAL := 0;
BEGIN

  SELECT custid INTO cid
  FROM account
  WHERE name=n;

  IF NOT FOUND THEN
    RAISE EXCEPTION 'Balance: customer % not found', n;
  END IF;

  SELECT bal INTO a
  FROM saving
  WHERE custid=cid;

  SELECT bal INTO b
  FROM checking
  WHERE custid=cid;

  total := a + b;

  RETURN total;
END;
$BODY$
LANGUAGE 'plpgsql' STABLE;
ALTER FUNCTION sitest.balance(n character varying) OWNER TO postgres;
```
nodes representing update transactions. 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 edges 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.

We see that the only dangerous structure is Balance (Bal) -- WriteCheck (WC) -- TransactSaving (TS). 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 [13], in which Bal sees a total balance value which implies that an overdraw penalty would not be charged, but the final state shows such a penalty because WC and TS executed concurrently on the same snapshot.

Ways to Ensure Serializable Executions for SmallBank: We have two options for edge sets on which to introduce conflicts, to eliminate the dangerous structure in the SmallBank SDG: either we make the edge from WriteCheck to TransactSaving non vulnerable (Option WT), or we make the edge from Balance to WriteCheck not vulnerable (Option BW). For each choice of edge set, we further have three alternatives on how to make that edge not vulnerable (Promotion, Materialize, and ELM).

- Option WT: In Option WT we eliminate the dangerous structure by making the edge from WriteCheck to TransactSaving not vulnerable. This can be done by materializing the conflict (that is, placing “update table conflict”
statements into both WriteCheck and TransactSaving). Thus we define a table `Conflict`, not mentioned elsewhere in the application, whose schema is `Conflict(Id, 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), we update only the row in table `Conflict` where the primary key=x, where x is the CustomerId of the customer involved in the transaction. We call this strategy `Materialize WT`. Here is the statement we include in both programs, WC and TS.

1- UPDATE Conflict
2- SET Value = Value+1
3- WHERE id=:x

For this to work properly, we must initialize `Conflict` with one row for every CustomerId, before starting the benchmark; otherwise we need more complicated code in WC and TS, that inserts a new row if now exists yet for the given id.

An alternative approach which also eliminates the vulnerability is by promotion, adding an identity update in WriteCheck. We represent this strategy by `Promote WT`. To be precise, PromoteWT includes the following extra statement in the code of WC above.

1- UPDATE Saving
2- SET Balance = Balance
3- WHERE CustomerId=:x

In the commercial platform we consider, we use the term `Promote WT-upd` for the Promotion modification described above, because there is also a strategy `Promote WT-sfu`, where the second SELECT statement in the code above for WC is replaced by

1- SELECT Balance INTO :b
2- FROM Saving
3- WHERE CustomerId=:x FOR UPDATE

Finally, using the ELM technique, we only need to wrap WriteCheck and TransactSaving transactions with a few statements to ensure that they are not running concurrently, so at the beginning we acquire the locks (using `getLock();`), execute the stored procedure, and then release the locks (using `release();`). In most experiments, the lock-choice is based on the Parameter-Value technique discussed in 3.2. We represent this technique by `ELM-WT`. Here is how the client calling WriteCheck looks after we modify it. The N parameters of the WriteCheck(N,V) transaction is taken as one element name[] from an array of possible account holder names.

1- `cstmt = con.prepareCall
2- "\{call WriteCheck(N,V)\}"
3- `Lock l = locker.getLock(names[counter]); //To acquire locks.```
try {
    cstmt.setString(1, names[counter]);
    cstmt.executeUpdate(); // Execute the transaction.
    con.commit();
} finally {
    l.release(); // Release the locks.
}

And here is the client for modified TransactSaving.

```java
stmt = con.prepareCall
("{call TransactSaving(N,V)}");
Lock l = locker.getLock(names[counter]); // To acquire locks.
try {
    cstmt.setString(1, names[counter]);
    cstmt.executeUpdate(); // Execute the transaction.
    con.commit();
} finally {
    l.release(); // Release the locks.
}
```

Note that we only modify WriteCheck and TransactSaving to acquire locks, and leave the other transactions unmodified.

(a) SDG for Option PromoteWT and MaterializeWT.
(b) SDG for Option ELM-WT.

Fig. 5. WT edge options.

In Figure 5(a), we show the SDG for promote and materialize with the WT option, and Figure 5(b) shows the SDG for using ELM with WT option. Only the edge between WriteCheck and TransactSaving has changed, the remaining edges are unchanged.
Option BW: We can also ensure that all executions are serializable, by changing the programs so that the edge from Balance to WriteCheck is not vulnerable. This can again be done by materializing (which includes an update on Conflict in both programs Bal and WC), and we call it MaterializeBW. Here is the statement we include in both programs Bal and WC.

1- UPDATE Conflict
2- SET Value = Value+1
3- WHERE id=:x

The second choice is by promoting with identity update on the table Checking in Bal, and we call it PromoteBW.

1- UPDATE Checking
2- SET Balance = Balance
3- WHERE CustomerId=:x

In the commercial platform, we use PromoteBW-upd for this, and we also consider promoting with select-for-update on table Checking in Bal, which we call PromoteBW-sfu.

1- SELECT Balance INTO :b
2- FROM Checking
3- WHERE CustomerId=:x FOR UPDATE

Finally, using the ELM technique, we only need to wrap Balance and WriteCheck transactions with few statements to ensure that they are not running concurrently.

1- cstmt = con.prepareCall("{call Balance(?,?)}");
2- if ("LockBW".equals(serialMethod)||"LockALL".equals(serialMethod))
3- {
4- numlocked += 1;
5- Lock l = locker.getLock(names[counter], false);
6- try
7- {
8- cstmt.setString(1, names[counter]);
9- cstmt.registerOutParameter(2, Types.REAL);
10- cstmt.executeUpdate();
11- con.commit();
12- }finally {
13- l.release();
14- }
15- }

In Figure 6 and 7 are the SDGs for PromotionBW, MaterializeBW and ELM-BW technique. Note that the Balance transaction is no longer read-only in Figure 6(a) and 6(b), other outgoing edges from Balance have changed.
(a) SDG for MaterializeBW. (b) SDG for PromoteBW.

Fig. 6. BW edge options.

Fig. 7. SDG for Option ELM-BW.
Option ALL: All the strategies discussed so far work from a detailed examination of the SDG, and identifying the dangerous structures in that. An approach which has less work for the DBA is to simply eliminate all the vulnerable edges (without thought of cycles, or consecutive edges). The DBA needs simply to consider each pair of transactions, and decide whether or not there is an RW conflict without a WW one; if so we remove the vulnerability on that edge (by materialization, promotion, or by using ELM). We refer to these strategies as MaterializeALL, PromoteALL, and ELM-ALL.

Because every transaction (except Bal itself) has a vulnerable edge from Bal, the approach MaterializeALL includes an update on table Conflict in every transaction (and indeed, transaction Amg must update two rows in Conflict, one for each parameter, since either customer could be involved in a vulnerable conflict from Bal). PromoteALL adds an identity update on Savings to transaction WC, and it adds identity updates to both Savings and Checking tables in transaction Bal, since Bal has a vulnerable conflict on Checking with WC, Amg and DC and a vulnerable conflict on Savings table with TS and Amg. Using ELM ELM-ALL technique, every pair of transactions joined by a vulnerable edge, must acquire ELM lock on the common parameters that construct the vulnerable edges, so they do not run concurrently. We use the parameter-value technique to control concurrent transactions.5

Figure 8 summarizes the different options whose performance we compare in our experiments. It lists for each option to ensure serializable executions, and for each type of transaction, which modifications are introduced. For Promote and Materialize, the modifications are additional updates on either the Savings table (Sav), the Checking table (Chk), or to the dedicated Conflict table (Cnf); for each option within the ELM approach, and for each transaction, the modification can be to set a lock in the ELM (Lock).

<table>
<thead>
<tr>
<th>Option / TX</th>
<th>Bal</th>
<th>WC</th>
<th>TS</th>
<th>Amg</th>
<th>DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELM-BW</td>
<td></td>
<td>Lock</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELM-WT</td>
<td></td>
<td>Lock</td>
<td>Lock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELM-ALL</td>
<td>Lock</td>
<td>Lock</td>
<td>Lock</td>
<td>Lock</td>
<td>Lock</td>
</tr>
<tr>
<td>PromoteBW</td>
<td></td>
<td>Chk</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PromoteWT</td>
<td></td>
<td>Sav</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PromoteALL</td>
<td></td>
<td>Chk, Sav</td>
<td>Sav</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaterializeBW</td>
<td>Cnf</td>
<td>Cnf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaterializeWT</td>
<td>Cnf</td>
<td>Cnf</td>
<td>Cnf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaterializeALL</td>
<td>Cnf</td>
<td>Cnf</td>
<td>Cnf</td>
<td>Cnf</td>
<td>Cnf</td>
</tr>
</tbody>
</table>

Fig. 8. Overview of Modification Introduced with each Option.

5 The transactions use customer name "Name[counter]" as parameter to control the concurrent update. Thus under ELM-ALL, Amg transaction needs to lock two parameters "account1", "account2".
**MoreChoices MicroBenchmark** The SmallBank microbenchmark has been useful for exploring the performance of different approaches that each guarantee serializable execution. However, SmallBank has a number of characteristics that are atypical (for example, its SDG has only one dangerous structure and no examples of Write Skew). In order to check that our conclusions are not specific to these aspects of SmallBank, we have designed another set of application programs, designed to have different characteristics (e.g., more cycles and write skew). We call this microbenchmark MoreChoices. In this benchmark, unlike SmallBank or TPC-C, we do not try to make the schema or programs meaningful for any domain.

**MoreChoices Benchmark Schema:** Our proposed benchmark consists of three main tables: Table0(CharID, Id), Table1(ID, Value1), and Table2(ID, Value2).

**Transaction Mix:** Our MoreChoices benchmark runs four different types of transactions. $T_1$ is a read-only transaction, and $T_2$, $T_3$, and $T_4$ are update transactions. The SQL logic here does not have any meaning, it only exercises the DBMS.

- **Transaction1($T_1$):** $T_1$ reads Table1 and Table2. Program 9(a) shows an example of the core of the SQL code for $T_1$.
- **Transaction2($T_2$):** $T_2$ reads Table1, Table2 and it updates Table1.
- **Transaction3($T_3$):** $T_3$ reads Table2 and updates Table2.
- **Transaction4($T_4$):** $T_4$ reads Table1, and Table2 and it updates Table2 in order to create write skew with $T_2$.

**Program 2** Transaction1(N) transaction.

```sql
SELECT Id INTO :x
FROM Table0
WHERE CharID=:N;

SELECT val1 INTO :a
FROM Table1
WHERE Id=:x;

SELECT val2 INTO :b
FROM Table2
WHERE Id=:x;

COMMIT;
```

(a) Program1 SQL code. (b) SDG for MoreChoices benchmark.

**Fig. 9.** MoreChoices Benchmark.

Figure 9(b) shows the SDG for MoreChoices microbenchmark. We use dashed edges to indicate vulnerability, and we shade the nodes representing update
transactions. Computing the SDG is very similar to analyzing SmallBank. We have analyzed the MoreChoices SDG by hand and there are five dangerous structures.

1. \( T_1 \rightarrow T_2, T_2 \rightarrow T_4, T_4 \rightarrow T_1 \).
2. \( T_4 \rightarrow T_2, T_2 \rightarrow T_3, T_3 \rightarrow T_4 \).
3. \( T_2 \rightarrow T_4, T_4 \rightarrow T_2 \).
4. \( T_1 \rightarrow T_2, T_2 \rightarrow T_3, T_3 \rightarrow T_1 \).
5. \( T_1 \rightarrow T_4, T_4 \rightarrow T_2, T_2 \rightarrow T_1 \).

Ways to ensure serializability with MoreChoices benchmark: We consider three options of edge set to deal with. Each of these different choices guarantees that we do not have a dangerous structure in our SDG graph. There are 2 minimal sets of edges that break each dangerous cycle. We also consider the option where we remove vulnerability on certain edges.

- Choice1: Removing the vulnerable edges \{ \( T_1 \rightarrow T_2 \), and \( T_4 \rightarrow T_2 \) \}. Note that \( T_1 \) is not a read-only transaction any more after we promote or materialize the edge \( T_1 \rightarrow T_2 \).
- Choice2: Removing the vulnerable edges \{ \( T_2 \rightarrow T_4, T_4 \rightarrow T_2 \), and \( T_2 \rightarrow T_3 \) \}.
- ALL: Removing a commercial platform vulnerable edges \{ \( T_1 \rightarrow T_2, T_1 \rightarrow T_3, T_1 \rightarrow T_4, T_4 \rightarrow T_2, T_2 \rightarrow T_3 \) \}.

We further have three alternatives on how to make each option non vulnerable (Promotion, Materialize\(^6\), and ELM).

5 EVALUATION

This section evaluates the various techniques described previously, including our new ELM proposal as well as Materialize and Promotion described in the background. We compare these techniques using one open source platform PostgreSQL and one commercial platform, to be able to generalize our findings and conclusions. We evaluate each technique under different conditions, such as low data contention, high data contention, varying the number of concurrent clients (MPL), and changing the percentage of read-only transactions in the mix. The goal of our experiments is to understand the performance implications of each technique so the DBA can make a sensible choice of modification technique, given a particular platform and workload pattern. Thus, it would not be appropriate to compare absolute numbers between different platforms or benchmarks; rather one should consider the trends for each scenario under the different modifications.

\(^6\) In using Materialize to ensure that we do not increase the amount of contention by introducing the new table "Conflict", we make sure that each edge has its own conflict table.
5.1 ELM Lock Naming

In this section we study the performance of the several alternative lock naming choices suggested in section 3.2. This will justify our decision to use only the Parameter-Value lock naming, in the later comparisons with other techniques.

We implemented the different lock naming techniques, and tested them under different conditions such as high and low data contention, using both PostgreSQL and the commercial engines.

Figure 10 shows the throughput in transaction per second (TPS) as a function of MPL for both low and high data contention, for the different lock namings for ELM-BW. We perceive that:

– The edge-name technique and item-name technique throughputs rise with MPL till it reaches a plateau of about 200 TPS, from MPL=5. These two techniques behave indistinguishably from one another, regardless the data contention and MPL.

– The Parameter-value and very-fine-granularity techniques rise with MPL till it reaches a plateau of about 800 TPS at MPL=20 with low contention, and a peak of about 400 TPS at MPL=15 with high data contention. Under high contention, performance then drops slightly as MPL increases further. The two techniques' behavior is very similar, and their performance is much better than the edge-name technique and item-name technique (≃70% or more improvement with MPL=15-30 for the low contention, and ≃45% improvement with MPL=15-30 for the extreme environment with high contention)).

The edge-name and item-name techniques cause a hefty performance loss compared to more precise locking; both techniques reduce the number of concurrent transactions by false conflicts which block transactions that are not required to be blocked as discussed in section 2.1. While we do not observe any difference between them in these experiments, in some other scenarios, the item-name technique could have even more false conflicts than the edge-name technique.

Suppose we have more than two transactions that have a conflict on the same data item; for example if we have \(T_1 \xrightarrow{balance/e1} T_2\) and \(T_2 \xrightarrow{balance/e2} T_3\) with conflict over a data item called \(balance\), then with the item-name technique, \(T_1\) and \(T_3\) may conflict on \(balance\) even they are not suppose to have a conflict, while with edge-name technique \(T_1\) and \(T_3\) will not falsely conflict since they use different edge names. This does not occur in SmallBank in the BW option, as only Balance and Writechecking transactions need locks.

In general, the Parameter-value technique allows more concurrent transactions than the edge-name and item-name techniques, but less than very-fine-granularity technique. However, the experiment does not show any significant performance difference between the Parameter-value and very-fine-granularity techniques, because there is only one edge that is involved in the locking.

Figure 11 shows the throughput in transaction per second (TPS) as a function of MPL with a low and high data contention for the different lock techniques for WT-option. The figure clearly confirms the same patterns we observed.
from the BW-option figures; that the edge-name and item-name techniques throughput is indistinguishable, and both are less than the Parameter-value and very-fine-granularity techniques. Parameter-value and very-fine-granularity techniques throughput are in the same range; we do not expect to gain any further performance with very-fine-granularity technique since we have only two transactions (Writechecking and Transactionsaving) that do locking with WT-option.

We also tested these lock name alternatives with the choice where we lock ALL the vulnerable edges with SDG graph. Figure 12 shows the throughput in transaction per second (TPS) as a function of MPL with a low and high data contention for the different lock techniques for ALL-option. The behaviours of the lock alternatives techniques show the same trends (with different throughput numbers) under this extreme option, which confirms the patterns we derived from BW-option and WT-option. Even with ALL transactions using the ELM, the performance of the very-fine-granularity technique and the Parameter-value technique is very close (and indeed, while the difference between them is not enough to be clearly significant in our data, the advantage with low contention actually lies with the Parameter-Value technique).

The results with PostgreSQL engine illustrate that these techniques gathered into two groups in term of their behaviour and performance; the edge-name and
item-name techniques each do markedly worse than the Parameter-value and very-fine-granularity techniques.

We also run the same experiments under the same conditions, using a commercial database engine. Figure 13 shows the throughput in transaction per second (TPS) as a function of MPL with a low and high data contention for the different lock techniques for BW-option. We perceive that:

- The edge-name technique and item-name technique throughputs rise with MPL till it reaches a plateau at MPL=10. These techniques performance is indistinguishable under both low and high data contention.
- The Parameter-value and very-fine-granularity techniques rise with MPL till it reaches a plateau at MPL=15 with low contention, and with MPL=10 with high contention, then the throughputs drop dramatically for both techniques. Both techniques throughputs are almost similar.
- The peak throughput for Parameter-Value and very-fine-granularity naming are similar (around 580 TPS with low contention and around 400 TPS with high contention) and much higher than the peak of about 220 TPS for the item-name and edge-name techniques.
- If we consider the throughput at a given MPL, rather than the peak, we see that the Parameter-value and very-fine-granularity techniques have throughputs that are higher than both the edge-name technique and item-name technique till MPL=15 with low data contention, and till MPL=10 with high data contention. After that, the opposite is happening; both the edge-name technique and item-name technique start beating the Parameter-value and very-fine-granularity techniques (e.g. more than $\approx 60\%$ with MPL=30 for both low and high data contention).

The results are surprisingly different than what we saw with PostgeSQL, especially that the edge-name technique and the item-name technique could perform better than the Parameter-value or very-fine-granularity techniques under the situations where performance has dropped far below peak, due to excess conflicts. This can be explained that the edge-name technique and the item-name technique reduce the number of concurrent transactions inside the commercial database engine whose resources can become badly overloaded (especially the space for storing versions may be an issue), which as consequence reduces the
amount of contention inside the engine and provide more availability of the resources to the running transactions.

Figure 13 shows the throughput in transaction per second (TPS) as a function of MPL with a low and high data contention for the different lock techniques for ALL-option, where ALL vulnerable edges have been affected by inserted ELM locks. Here the edge-name technique and item-name techniques perform almost the same (and poorly) under a range of conditions. The parameter-value and very-fine-granularity techniques rise with MPL till it reaches a plateau at MPL=30 with low contention, and with MPL=20 with high data contention. These give much better peak throughput than the edge-name technique and item-name techniques. The parameter-value technique has slightly higher throughput than the very-fine-granularity technique in the low data contention graph while under high data contention, the very-fine-granularity technique performs slightly better.

These experiments justify our decision to fix on the Parameter-Name lock approach when we consider ELM in subsequent experiments comparing it with

![Throughput with low contention.](image1)

![Throughput with high contention.](image2)
other techniques, as (for our microbenchmarks) Parameter-Value peak performance is among the best of the naming alternatives under the diverse scenarios we considered.

5.2 Comparing Options to ensure serializable execution with SmallBank

In this section we use Figure 8 from section 4 that summarizes the different options which we compare. It lists for each option to ensure serializable executions, and for each type of transaction, which modifications are introduced. For Promote and Materialize, the modifications are additional updates on either the Saving table (Sav), the Checking table (Chk), or to the dedicated Conflict table (Cnf); for each option within the ELM approach, and for each transaction, the modification can be to set a lock in the ELM (Lock).

Serializability of SI on PostgreSQL, for SmallBank Through several subsections we will explore the performance of the application modification techniques for guaranteeing serializable execution on SI platforms. Each subsection deals with a particular platform, and a particular benchmark of programs that are executed. In this subsection, we use PostgreSQL as the DBMS engine, and we use the SmallBank benchmark set of application programs.

Low Contention, High Update Rate In this experiment we select each transaction uniformly. That is Bal is 20% of transactions, WC is 20%, Dc is 20%, TS is 20%, and Amg is 20%. This means that 80% of the transactions update the database and only 20% are read-only. Here we explore in detail the case where hotspot has 100 rows; this means that even at MPL=30, a given transaction sees no contention about 2/3 of the time.

Figure 16(a) shows the throughput in transaction per second (TPS) as a function of MPL for the more sophisticated among the different options available

\[\text{These options all required the DBA to identify dangerous structures, and chose a minimal set of edges to make non-vulnerable.}\]
in SmallBank for guaranteeing serializable execution. We also include the figures (labelled SI) for the unmodified application under SI. From the same data, we derive Figure 16(b) which shows the relative performance as compared to the throughput with SI at the same MPL for each option that ensures serializable executions. In this graph, we use the thick horizon line at the 100% level, which is the score of SI, that is, running unmodified applications (these may have anomalies!).

![Graph](image)

(a) Throughput over MPL.  
(b) Throughput relative to SI.  

**Fig. 16.** Low Contention, High Update, SmallBank, PostgreSQL.

We perceive that

- For each option, throughput rises with MPL till it reaches a plateau. The plateau (maximal) value for throughput of the unmodified application (SI) is about 971, reached with MPL between 20 and 25.
- Throughput for PromotionBW, and also for MaterializeBW, starts 21% lower than SI and rises till it reaches about 94% of that for SI with MPL=30.
- PromoteWT and MaterializeWT are very close to SI until MPL=20 (for PromoteWT) or till MPL=10 (for MaterializeWT). Beyond this, they drop a bit but still are around 95%.
- ELM-BW and ELM-WT are often indistinguishable from results for SI, and sometimes slightly higher.

We now attempt to explain why these effects arise.

PromoteBW and MaterializeBW, have a somewhat lower peak and reach it more slowly (at MPL=30). MaterializeBW and PromoteBW introduce a write into Balance, and thus make every transaction need a disk write. This is clearly seen in the performance with MPL=1, where (with a single thread submitting transactions) there is no contention at all, and the slowdown comes only from the overhead. We see a slowdown of 20% for those modifications that increase the fraction of transactions that must do disk-writes by 5/4, and no slowdown at MPL=1 for the other modifications (cf. Figure 16(b)). This clearly shows that the need to write to disk is overwhelmingly dominant in the work done; once a
transaction needs one write, as happen for example in WC under MaterializeWT, extra writes have negligible extra cost.

PromoteWT and MaterializeWT come close to the peak of SI. Materialization or promotion on WT introduce updates only into programs (WC and TS) that already have them, and so one-fifth of the transactions remain read-only (the Balance transactions). ELM-BW and ELM-WT have very similar or even slightly higher throughput than unmodified SI. With ELM-BW, the extra cost of communication between the driver (when a Bal or WC transaction is to be run) and the ELM is negligible and it does not affect the overall throughput compared to SI itself.

Figure 17(a) shows the percentage of transaction abort arising from serialization failure, under different options. As we expect, the Promote and Materialize techniques increase the ratio of “Serialization Failure” aborts compared to the unmodified application under SI, because they introduce conflicts through the FCW mechanism. Promotion of BW does lead to contention between Bal and DC, and also between Bal and Amg. This is because both DC and Amg include updates on Checking, and the promoted version of Bal has an identity update, on the appropriate row of the Checking table. We see that for MPL of 25 or more, PromoteBW reaches a worrying level where over 17% of transactions must abort. MaterializeBW has a lower abort rate than PromoteBW, since MaterializeBW only stops the conflicts between Bal and WC without creating any extra conflict as happens with PromoteBW. In contrast, the options that use the ELM technique have a lower rate of these errors even than the unmodified application. This is because when two threads concurrently try to run one of the modified programs, with the same account number, the ELM lock will delay one till the
other finishes, whereas in this same scenario, one instance of the unmodified program will abort due to FCW.

Figure 17(b) shows how different transaction types have different patterns for the ratio of Serialization Failure errors with MPL=25. We see that every transaction type individually shows lower abort rates in the ELM techniques, even than for the unmodified application under SI. On the other hand, PromoteBW and MaterializeBW cause aborts in the (originally abort-free, because read-only) Balance transaction, similarly PromoteWT and MaterializeWT raise the abort rates in WC and TS transaction types.

We conclude that performance of Promotion and Materialize techniques are dominantly affected by the transactions that join the chosen edges, so if the developers are interested in high performance for a specific transaction type, then the conflicts that affect this type should not be increased when using materialisation or promotion. However, introducing locks into this transaction using the ELM technique is quite acceptable.

Figure 18(a) shows the mean response time averaged over all transactions, in milliseconds. We see that PromoteBW and MaterializeBW have the highest mean response time. This seems to be due to two reasons:

- Changing the read-only (Bal transaction) to update transaction, which adds a lot of extra time by forcing Bal transaction to access the disk when writing the log.
- Since our system restarts the aborted transactions, and PromoteBW and MaterializeBW have the highest abort rate, then extra time is needed to re-try those transactions.

Each of the ELM techniques generally has mean response time which is less than the best available among other approaches.

8 The lock is introduced to prevent concurrency between the two programs at opposite ends of an SDG edge, but it also causes conflicts between each program and itself, where the SDG has a non-vulnerable loop edge.
Performance of Program Modification for Serializability Over SL

Figure 19 shows the message sequence diagram. The mean response time consists of three components which are:

1. Commit Time: It is the average time for successful transaction to commit. It starts from the last time we submit the business logic (which is the attempt that succeeds) to the time we receive the answer (commit).
2. Restart Time: It is the average wasted time for transactions that could not commit their jobs. It starts from the time we submit business logic to the time we receive an error message (Abort).
3. ELM overhead Time: It is the time we need to communicate with the ELM, acquire locks, and release them. It includes the period that starts from the time we submit a request to the ELM until we get the answer that the lock was obtained, plus the time we need to release these locks after commit.

Figure 18(b) shows the detailed breakdown of the mean response time for MPL=25. We have also labeled each portion of time as percentage of the total response time for that option. For example, ELM-BW mean response time is split between 83.7% (Commit time) + 6.7% (ELM overhead) + 9.6% (Restart overhead). The commit time for ELM-BW and ELM-WT is less than for any other option available. This simply because the ELM reduces the amount of contention inside the database, therefore, the average waiting time of a transaction is less even than for unmodified uncommitted SI. Notice that the ELM-WT commit time is even lower than ELM-BW, because ELM-WT prevents more conflicts inside the database by controlling both WC and TS which would otherwise invoke FCW and cause more transactions to abort. ELM-WT stops WC and TS from running concurrently with parameters that lead to conflict. This reduces the probability of conflict arising between (WC and TS) and (Amg and DC). In contrast, in ELM-BW we only reduce the conflict between WC and (Amg and DC), while TS still has higher chance to conflict with (Amg and DC). We obviously see
that the waiting time for ELM locks is highly compensated by the lower commit time, and slightly by the reduction in time wasted in restarts.

![Graph](image)

(a) Throughput over MPL (ALL vulnerable edges).

(b) Relative Throughput to SI (ALL vulnerable edges).

**Fig. 20.** Low Contention, High Update, SmallBank, PostgreSQL.

Finally, we consider the straightforward strategies that remove the vulnerability from every vulnerable edge. These modify many transactions, but they do not require the DBA to look for cycles and dangerous structures in the SDG; instead the DBA can think about each pair of transactions separately. Figure 20(a) shows the resulting throughput in Transactions Per Second (TPS) as a function of MPL. Figure 20(b) shows the relative performance as compared to the throughput with SI (shown as thick horizontal line) for each option that ensures serializable executions. As we see, the simple approaches induce hefty performance costs except with ELM-ALL. Promoting every vulnerable edge has performance that starts 20% lower than SI and rises till it reaches about 91% of that for SI. Materializing on every vulnerable edge gives performance that peaks at about 807 TPS (about 18% less than that for SI). The relative performance between these is understandable: when we promote every vulnerable edge, we simply add two writes to Balance, and one to WriteCheck, without changing the other programs, and so we do continue to allow DC and TS to run concurrently (they do not conflict at all). In contrast, materializing all, by including a write to the conflict table in every transaction, means that a conflict is likely between any pair of transactions which deal with the same customer.

Figure 21 shows that ELM has zero serialization failure, where other options have 18-19%.

While we notice that ELM has better throughput than the other options, and sometimes it is even better than SI, the improvement is often small, and so this is not what we consider the central benefit of ELM. Rather, we notice that ELM is quite robust among the different choices of edge set. Even with the simplistic

---

9 When we use ELM with every vulnerable edge, we actually prevent every single program from getting into a conflict with other programs and with itself.
Performance of Program Modification for Serializability Over SI

Fig. 21. Percentage of Serialization Failure (ALL edges), Low Contention, SmallBank, PostgreSQL.

ALL choice, ELM never loses much. That is, ELM is a robust approach, which protects the DBA against making a poor choice of edge set.

High Contention, High Update Rate We repeated our experiments with a reduced hotspot size of 10 rows (out of 20,000 in the tables), to create a situation in which conflicts are very frequent. Testing the different techniques’ performance under such extreme conditions assists us to verify the robustness of these techniques. The transaction types are uniformly selected (20% for each).

Fig. 22. High Contention, High Update, SmallBank, PostgreSQL.

Figure 22(a) shows the throughput in transaction per second (TPS) as a function of MPL for the different options available in SmallBank for guaranteeing serializable execution. We also include the figures (labeled SI) for the unmodified application under SI. Where Figure 22(b) shows the relative performance as compared to the throughput with SI (shown as thick horizontal line) for each option that ensures serializable executions. We perceive that

- For each option, the overall shapes look similar to low data contention, but with less throughput in each case.
- Throughput for PromotionBW_upd (Identity update) edge starts 21% lower than SI and rises till it reaches about 83% of that for SI with MPL=30.
- Throughput for MaterializeBW edge starts 20% lower than SI and rises till it reaches about 97% of that for SI with MPL=30.
- PromoteWT and MaterializeWT are very close to SI.
- ELM-BW and ELM-WT are indistinguishable of that for SI, and sometime slightly higher.

Our observations from the low data contention are still valid for high data contention except that MaterializeBW throughput is higher than PromoteBW due to the extra abort rate and restarts with PromoteBW.

![Graph showing Serialization Failure](image1)

**Fig. 23.** High Contention, High Update, SmallBank, PostgreSQL.

Figure 23(a) shows the percentage of serialization failure of different options. Under this extreme condition, the percentage of serialization failure has been increased due to the high conflict, PromoteBW still has the highest number between the options (around 63%). Again ELM-WT and ELM-BW have lower failure rates than unmodified SI (around 50-52%).

Figure 23(b) shows the percentage of serialization failure per transaction type. PromoteBW and MaterializeBW cause aborts in the Balance transaction (2.5%-6.9%), similarly PromoteWT and MaterializeWT raise the abort rates in WC and TS transaction types. On the other hand, we see that every transaction type individually shows lower serialization failure rates in the ELM techniques, even than for the unmodified application under SI (especially with TS, and AMG). Our conclusion for low contention is still valid here: the ELM technique has lower abort rate for each specific transaction type than Promotion and Materialize techniques.

Figure 24(a) shows the mean response time for the different options that make SI serializable. We still see that PromoteBW and MaterializeBW have the highest response time due to the same reason of changing a read-only transaction to be an update transaction (Bal transaction), and due to the high percentage of restart.
Figure 24. High Contention, High Update, SmallBank, PostgreSQL.
Figure 24(b) shows the detailed mean response time for MPL≈25. The percentage of restart with ELM-BW is higher than ELM-WT. ELM-WT stops WC and TS from running concurrently, which reduces the probability of conflict between (WC and TS) and (Amg and DC). ELM-BW only reduces the conflict between WC and (Amg and DC), while still TS has higher chance to conflict with (Amg and DC). We clearly see that ELM overhead is highly compensated by a reduction in restart time and slightly by the commit time (the relative importance of these effects is opposite to what we found the low data contention case).

Figure 25. High Contention, SmallBank, PostgreSQL.
Figure 25(a) shows the throughput in transaction per second (TPS) as a function of MPL for each option that ensures serializable executions by removing ALL vulnerable edges. Figure 25(b) shows the relative performance as compared to the throughput with SI (shown as thick horizontal line). As we see, all simple approaches induce hefty performance costs. Promoting and Materializing every vulnerable edge has performance that start 20% lower than SI and rises till it reaches around 85% of that for SI. ELM-ALL perform better than both techniques.

The qualitative conclusions are the same as the low contention case: any techniques that affect only the WT edge do quite well, but Promotion and Materialize are fragile, losing performance if ALL edges (or even just BW edge) are chosen.
for conflict introduction. In contrast, ELM never does very badly, even with ALL edges chosen (between 2%-12% lower than the unmodified programs under SI).

**Low/High contention, Low Update Rate** Some real world applications have more frequent read-only transactions than update [28]. Therefore we also run experiments where we increased the percentage of Balance transaction (which is the only read-only transaction in SmallBank) to 60% instead of 20%. The update transactions are submitted each 10% of the time, with total update rate 40%. We vary the data contention between low (100 rows) and high (10 rows) to understand the options that ensure serializable execution on SI platforms.

![Graph showing throughput and relative performance](image)

(a) Throughput with 60% read-only.  
(b) Relative Throughput with 60% read-only.

**Fig. 26.** Low Contention, SmallBank, PostgreSQL.

Figure 26(a) shows the throughput in transaction per second (TPS) as a function of MPL for the different options available in SmallBank for guaranteeing serializable execution. Figure 26(b) shows the relative performance as compared to the throughput with SI (shown as thick horizontal line) for each option that ensures serializable executions. We see that increasing the number of Balance transactions has a high impact on the performance of PromoteBW since more transactions become update transactions under this option and there is more chance for extra conflict between Bal and DC, and also between Bal and Amg. PromoteBW and MaterializeBW have performance that starts 60% lower than SI and rises till it reaches about 77-79% of that for SI. However, ELM-BW suffers much less when we increase the percentage of Balance transaction. Its performance starts 32% lower than SI and rises till it reaches about 92-94% of that for SI (indistinguishable between MPL=15-20).

The mean response time (Figure 27(a)) for PromoteBW and MaterializeBW is much higher than any other option due to the percentage of Bal transaction in the mix. ELM-BW has mean response time which is slightly higher than for SI (after MPL=20) due to the extra communication with ELM. Other options (PromoteWT, MaterializeWT, and ELM-WT) have mean response times which are close to unmodified SI due to the small percentage of WC and TS update transactions (10% for each).
The high level conclusions from Figure 27(b) is similar to Figure 17(a). Promoting BW edge comes with a high cost especially with 60% of Balance transaction, as Balance transaction increases the probability of extra abort rate. ELM technique has lower failure rate even than the unmodified SI. Between these these extremes, it really depends on the percentage of the transactions in the mix. For example, in Figure 27(b) Materialize BW edge has higher failure rate than unmodified SI, PromoteWT and MaterializeWT, because Balance transaction is 60% and (TransactionSaving and Writechecking) is only 20%. On the other hand, in Figure 17(a), Materialize BW edge is really close to unmodified SI, PromoteWT and MaterializeWT, where percentages of transactions are fixed.

![Graph](a) Mean Response Time with 60% read-only. 
(b) Serialization Failure with 60% read-only.

Fig. 27. Low Contention, SmallBank, PostgreSQL.

We also tested this mixture under a high contention scenario where the hotspot was 10. Figure 28(a) shows the throughput in transaction per second (TPS) as a function of MPL, and Figure 28(b) shows the relative throughput to SI. While such high data contention reduces the overall peak performance for each technique (SI peaks at 1257 TPS with hotspot 100, and at 820 TPS with hotspot 10), it does not change the overall picture from the low data contention graph, except that MaterializeBW performs better than PromoteBW, since there more chance for extra conflict with PromoteBW between Bal and DC, and also between Bal and Amg.

**Serializability of SI on a commercial database engine** So far, we have focused on PostgreSQL. For comparison, we also ran our experiments on one of the commercial platforms that offers Snapshot Isolation concurrency control. We investigate the behaviors of the different options that ensure serializable executions with SI on this platform. We also consider a new option called Promote_{sfu}, as on this platform the statement Select..For Update (SFU) is treated for concurrency control like an Update, and so promotion can be done by changing the read into SFU. In contrast, on PostgreSQL, we have found that using SFU does not always prevent an update in a concurrent transaction, and so SFU can not be used to make an edge non-vulnerable. We run the same experiments as
Low Contention, High update rate We will explore in some detail the case where hotspot has 100 rows, and each transaction type is equally frequent. Figure 29(a) shows the throughput in transaction per second (TPS) as a function of MPL for the different options available in SmallBank for guaranteeing serializable execution. We also include the figures (labeled SI) for the unmodified application under SI. Figure 29(b) shows the relative performance as compared to the throughput with SI (shown as thick horizontal line) for each option that ensures serializable executions.

We perceive that

- Throughput for PromotionBW_upd (Identity update) starts 20% lower than SI. It peaks at MPL=10 and then drops. It decreases relative to SI, till it reaches about 57% of that for SI with MPL=25.
Throughput for PromotionBW_sfu (Select..For Update) edge starts 20% lower than SI and decreases till it reaches about 65% of that for SI with MPL=25.

Throughput for MaterializeBW edge starts 20% lower than SI and rises till it reaches about 80% of that for SI with MPL=25.

Throughput for PromotionWT_upd edge starts 8% lower than SI and decreases till it reaches about 53% of that for SI with MPL=25.

Throughput for PromotionWT_sfu edge starts 8% lower than SI and decreases till it reaches about 74% of that for SI with MPL=25.

Throughput for MaterializeWT edge starts 8% lower than SI and rises till it reaches about 94% of that for SI with MPL=25.

Throughput for ELM-BW and ELM-WT edge are indistinguishable of that for SI.

We see a very different overall shape compared to that for PostgreSQL: the throughput for different options rise to a peak but then quickly drops away as MPL increases. Notice that Promotion\(^{10}\) leads to very poor performance under any edge choice, and Materialize does reasonably with some choices. ELM on the other hand, is robust; it does well (indeed better than unmodified SI) no matter which edge set is chosen.

Figure 30(a) shows the percentage of exceptions arising from FCW per transaction type with MPL=15 (where Peak throughput occurs). We see that every

\(^{10}\)PromoteBW_upd uses identity update to use FCW rule to force one of the transactions that join the chosen edge to abort. This technique requires the transactions to access the disk, which cause extra cost over PromoteBW_sfu. When we issue a Select..For Update statement, the RDBMS automatically obtains exclusive row-level locks on all the rows identified by the SELECT statement, holding the records "for your changes only". No one else will be able to change any of these records until you perform a ROLLBACK or a COMMIT. Furthermore, you do not have to actually UPDATE or DELETE any records just because you issued a Select..For Update, that act simply states your intention to be able to do so. This explains the slight throughput difference between PromoteBW_upd and PromoteBW_sfu.
transaction type individually shows lower abort rates in the ELM techniques; even than for the unmodified application under SI (this is the same as we saw in PostgreSQL). On the other hand, PromoteBW and MaterializeBW cause aborts in the (originally abort-free, because read-only) Balance transaction, similarly PromoteWT and MaterializeWT raise the abort rates in WC and TS transaction types. This confirms our conclusion from PostgreSQL that Promotion and Materialize options are affecting the percentage of aborts in transactions that are joined by the chosen edge.

Figure 30(b) shows the detailed mean response time for MPL=15 (Peak throughput). As we see, PromoteBW_upd, and MaterializeBW have the highest commit time due to change read-only transaction to update transaction, and PromoteBW_upd and PromoteWT_upd have the highest restart time. We clearly see that ELM overhead is compensated by reduction in restart time.

Finally, we consider the straightforward strategies that remove the vulnerability from every vulnerable edge. Figure 31 shows the resulting throughput in Transactions Per Second (TPS) as a function of MPL. Promote-ALL performs better than materialize-ALL. Materialize-ALL is including a write to the conflict table in every transaction, this means that a conflict is likely between any pair of transactions which deal with the same customer. However, with Promote-ALL, we add two writes to Bal, and one to WC, without changing the other programs, and so we do continue to allow DC and TS to run concurrently (they do not conflict at all). ELM-ALL throughput starts close to SI and continue to increase dramatically up to 726 TPS with MPL=30, while SI is only 95 TPS at the same MPL.

**High Contention, High update rate** Finally, we reduce the hotspot to 10 to create extreme contention in the commercial database engine. The high contention situation in Figure 32(a) and Figure 32(b) do not change the overall story, but they confirm the conclusions from PostgreSQL and from the commercial database engine at low contention. Figure 32(a) shows that materialize performs generally better than promotion but it still depends dominantly on
Figure 32(b) shows the resulting throughput in Transactions Per Second (TPS) as a function of MPL for removing the vulnerability from every vulnerable edge. ELM-ALL throughput starts close to SI (3% less than unmodified SI) and continue to increase up to 354 TPS compared to SI which has dropped to 61 TPS with MPL=30. Figure 33 shows the percentage of FCW errors per transaction. The overall story for both high and low data contention in the commercial database engine is the same: Promotion and Materialize are fragile, losing performance depending on the choice of edge, MPL and the contention. In contrast, ELM never does very badly, and even performs much better than SI with ALL edges chosen.

5.3 MoreChoices Benchmark Programs

The SmallBank benchmark has been useful for exploring the performance of different approaches that each guarantees serializable execution. However, SmallBank has a number of characteristics that are atypical (for example, its SDG has only one dangerous structure and no examples of Write Skew). In order
to check that our conclusions are not specific to these aspects of SmallBank, we have repeated experiments with another set of application programs called MoreChoices, designed to have different characteristics, and in particular to have a more complicated SDG mentioned in detail in section 4.

Recall that the choices for this benchmark are:

- Choice1: Introduce conflicts on the vulnerable edges \{ T1 \rightarrow T2, T4 \rightarrow T2. \}
- Choice2: Introduce conflicts on the vulnerable edges \{ T2 \rightarrow T4, T4 \rightarrow T2, T2 \rightarrow T3. \}
- Choice3: Introduce a conflict on ALL vulnerable edges \{ T1 \rightarrow T2, T1 \rightarrow T3, T1 \rightarrow T4, T2 \rightarrow T4, T4 \rightarrow T2, T2 \rightarrow T3. \}

We run our experiments using PostgreSQL platform, with low and high data contention, varying the number of concurrent transactions to study this benchmark.

**Low Contention** We will explore in detail the case where hotspot has 100 rows out of 20,000. To make the whole picture understandable, we show the summary of the choices using different options (Materialize, Promotion, and ELM) with MPL=25 (maximum throughput-plateau) in Figure 34. The overall message is: ELM performs as well, and even slightly higher than unmodified SI. 11 Promotion is a little higher than materialize, but both are lower than SI or ELM. Next we explore each choice in some detail.

**Choice1**: Figure 35(a) shows the throughput in transaction per second (TPS) as a function of MPL for the different modification options (with choice1 as edge set) that guarantee serializable execution with the new benchmark. Where Figure 35(b) shows the percentage of serialization failure for each option with choice1.

11 We see that the confidence interval for choice1 overlaps with SI, and choice2 is higher than SI, after MPL=25.
We perceive that

- Throughput for choice1_Promotion (abbreviated as choice1_pro) using identity update starts 25% lower than SI and arises till it reaches about 86% of that for SI with MPL=30.
- Throughput for choice1_Materialize (abbreviated as choice1_mat) starts 25% lower than SI and rises till it reaches about 78% of that for SI with MPL=30.
- Throughput for choice1_ELM 5% lower than SI and rises till it reaches about 104% of that for SI with MPL=30.

We see here that Promotion performs slightly better that Materialize after MPL=20. Both techniques promotion and materialize change the read-only transaction ($T_1$) to update transaction. choice1_Promotion add two update statements one to $T_1$ and another to $T_4$, and that causes extra abort rate between $T_1$ and $T_4$ since both of them update table1 (see Figure 36(a)).
Choice1: Materialize adds four update statements to \{T_1, T_2, \text{and } T_4\}, but does not cause any extra abort rate between \(T_1\) and \(T_4\).\(^{12}\) This explains Figure 35(b) where choice1.pro has the highest serialization failure (about 12%).

Choice1_ELM throughput is indistinguishable from that unmodified SI, and this is because we keep \(T_1\) as read-only transaction and we does not cost any additional log forces or re-starts. Choice1_ELM has the lowest serialization failure, even lower than unmodified SI, at around 3%.

Figure 36(b) shows the mean response time for choice1 options, as we see choice1_mat and choice1.pro have the highest MRT between the options. And choice1_ELM has the lowest, due to the high number of re-start.

**Choice2:** Figure 37(a) shows the throughput in transaction per second (TPS) as a function of MPL for the different options with choice2 that guaranteeing serializable execution with the new benchmark. Also Figure37(b) shows the percentage of serialization failure for each option with choice2.

![Choice2 Throughput](image1)

![Serialization Failure for choice2](image2)

Fig. 37. Low Contention, MoreChoices, PostgreSQL.

We clearly see that promotion still slightly perform better than materialize, and the ELM has the best throughput numbers between the options. choice2.pro has the highest serialization error rate (9.6% with MPL=30) and choice2_ELM has the lowest (zero%). Choice2 controls the concurrent update transactions \{T_1, T_2, \text{and } T_4\} which explains zero serialization failures.

**Choice3:** Choice3 considers the option when we remove every vulnerable edges from the SDG. Figure 38(a) and Figure 38(b) shows that the ELM technique is still superior over other available options, with zero ‘Can not Serialize’ errors.

\(^{12}\) Each edge has it’s own conflict table with materialize, so \(T_1\) add two update statements, one for \(T_4\) and one for \(T_2\). Therefore we have no conflict between \(T_1\) and \(T_4\).
6 Conclusion

In this paper we have carefully studied the performance implications of a range of techniques that allow the DBA to modify a suite of application programs, introducing conflicts so that the modified programs will have all their executions serializable, even when running on a platform with Snapshot isolation concurrency control. We find that the new ELM technique performs well under a wide range of situations, giving good throughput (close to that of the unmodified programs under SI, or even better in some cases). This is true no matter which of the choices the DBA makes for the edge set on which to introduce the modifications. In contrast, the previous techniques of Materialize and Promotion do well with some choices, but poorly with others (especially those where a read-only transaction is modified to include an update). Thus we can say that ELM is a robust technique to ensure serializability; it allows the DBA to feel comfortable that they won’t lose much performance through an unwise choice of edge set for modification.

In future work, we hope to develop a fairly simple performance model, that can allow rapid estimation of the performance that can be obtained for a given set of modified programs, based on easy-to-estimate values such as the amount of reading, the amount of writing, the extent of conflict, etc.

References


