Ensuring Serializable Executions with Snapshot Isolation DBMS

THIS THESIS IS

PRESENTED TO THE

FACULTY OF ENGINEERING AND INFORMATION TECHNOLOGIES

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

OF

THE UNIVERSITY OF SYDNEY

By

Mohammad I Alomari

December 2008

Abstract

Snapshot Isolation (SI) is a multiversion concurrency control that has been implemented by open source and commercial database systems such as PostgreSQL and Oracle. 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 execution, in which database integrity constraints can be violated. Several techniques have been proposed to ensure serializable execution with engines running SI; these techniques are based on modifying the applications by introducing conflicting SQL statements. However, with each of these techniques the DBA has to make a difficult choice among possible transactions to modify.

This thesis helps the DBA's to choose between these different techniques and choices by understanding how the choices affect system performance. It also proposes a novel technique called 'External Lock Manager' (ELM) which introduces conflicts in a separate lock-manager object so that every execution will be serializable.

We build a prototype system for ELM and we run experiments to demonstrate the robustness of the new technique compare to the previous techniques. Experiments show that modifying the application code for some transactions has a high impact on performance for some choices, which makes it very hard for DBA's to choose wisely. However, ELM has peak performance which is similar to SI, no matter which transactions are chosen for modification. Thus we say that ELM is a robust technique for ensure serializable execution.

Prior Publications

Parts of this thesis are taken from the following jointly authored papers:

1. **Mohammad Alomari**, Alan Fekete and Uwe Rohm, A Robust Technique to Ensure Serializable Executions with Snapshot Isolation DBMS, in Proceedings of IEEE International Conference on Data Engineering (***ICDE09***), 2009, to appear.

2. **Mohammad Alomari**, Michael Cahill, Alan Fekete and Uwe Rohm, The cost of serializability on platforms that use snapshot isolation, in Proceedings of IEEE International Conference on Data Engineering (***ICDE08***), 2008, pp. 576-585.

3. **Mohammad Alomari**, Michael Cahill, Alan Fekete and Uwe Rohm, Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels? in Proceedings of ***DASFAA08***, 2008, pp. 267-281.

4. **Mohamad Alomari**, Michael Cahill, Alan Fekete and Uwe Rohm, When serializability comes without cost, Computer Systems and Applications, 2008. ***AICCSA08***. IEEE/ACS International Conference on, 2008,pp. 164-171.

The work described was done jointly with Alan Fekete, Uwe Röhm and Michael Cahill. My particular contributions to this have been

- Conceiving and implementing the ELM technique described in chapter 3

- Design a new benchmark that has the required characteristics to evaluate the different techniques
- Carrying out the performance experiments
- As well, I participated in interpreting the data and writing these papers

Acknowledgements

I would like to thank my research advisor, Alan Fekete, for his constant support and counsel during my stay as a graduate student. I have learned many principles for performing good research. He helped me focus directly on the essence of the problems, and persisted on doing excellent research. I feel deeply indebted to his encouragement and support.

I also would like to thank many of the technical and administrative staff of the school of Information Technology who provided the timely support.

Words cannot even come close to expressing my gratitude to my parents, and to my brother Dr. Ahmad Alomari who deserve the credit for whatever positive that I have achieved in my life. They have always supported me in all my endeavors and have patiently waited for the completion of my degree. Last but not least, my wife Esra'a has been supportive and encouraging while I have been trying to finish up my thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter introduces my thesis with general overview of Snapshot isolation (SI) and the serializability problem. Then it shows the motivations, contributions, and the structure of the thesis.

1.1 An Overview

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 constraints). 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.

The concept of isolation level was introduced under name 'Degrees of Consistency' [10]. The ANSI/ISO SQL standard defines several levels of transaction isolation with differing degrees of impact on transaction processing throughput, to allow the database industry to weaken the requirement of serializability, in case where absolute correctness is not critical in order to increase the performance of common multi-user application . A greater degree of concurrency and better performance can be achieved using a lower level of transaction isolation. However, lower isolation can allow anomalies which might affect the consistency of the data.

Berenson et al [10] 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 2005. 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, [10] 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 [25]. 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 [25], it was shown how to 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. Two modification techniques are 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.

1.2 Motivation and Contributions of this Work

In order that they ensure all executions are serializable (on a platform providing SI), the DBA has a complicated choice to make: 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 [34]. Also DBA's have several options of which technique to use with the chosen subset of edges; they can modify the application in different ways.

Therefore, we offer a new 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 Manger" (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.

The motivation of my thesis is to help DBA's (e.g., software engineers) to make sensible choices of edges and conflict-introduction techniques among those available that ensure correctness of their applications.

The key contributions of this work are

- Development of a novel algorithm: I have designed a new concurrency control algorithm called External Lock Manager (ELM). Several database systems only offer a concurrency control mechanism providing the transaction isolation level snapshot isolation which allows certain anomalies to occur. In this case, the transactions can be extended to access ELM in order to guarantee serializable executions avoiding any form of anomaly. I developed a prototype implementation of ELM.

- Performance Evaluation: I evaluated the performance of both the new algorithm ELM and the state of the art algorithms for making applications run serializable on snapshot isolation. We have found many situations where performance for ELM

approaches that of unmodified (not necessarily serializable) SI. Unlike with promote and materialize techniques, this seems to hold across a range of choices for edge set on which to introduce conflicts.

- New Benchmarks: Existing benchmarks do not allow evaluating the performance and the impacts of the snapshot isolation level appropriately because they do not have non-serializable executions at all. Thus, I have designed new benchmarks that allow to stress-test the behavior of different ways to guarantee serializability for transactions running under snapshot isolation.

1.3 Thesis Outline

This thesis is organized as follow:

Chapter 2 covers the background concepts and other material from previous literature that is related to the thesis. The discussion includes transaction processes, database concurrency control, serializability theory, and TPC performance benchmarks.

Chapter 3 presents our new algorithm called the External Lock Manager (ELM) with details. We discuss the alternative designs and the implementation of the ELM algorithm. Fault tolerance is also discussed briefly.

In ***Chapter 4***, we give a detailed discussion of how the experiments are designed and setup. It describes the environment and the workload that we use for the performance study.

Chapter 5 thoroughly presents the experiment results for various techniques that ensure serializable executions under SI. We clearly explain the results.

In ***Chapter 6***, we conclude the thesis with summary of our work and findings.

Chapter 2

Background Concepts

This chapter introduces all the background information underlying the work in this thesis and it reviews the previous research literature related to the topic of this thesis.

2.1 Transaction Processing

A *transaction* is a unit of work that consists of several operations. A unit of work means that a transaction must be completely processed or completely aborted, and it can not leave the system in an intermediate state between these extremes.

Transaction Interface A transaction generally consists of beginning, read/write operations, logic, and finally abort/commit.

Begin: By sending a begin command, a client explicitly starts a new transaction. Some databases implicitly start the transaction upon the arrival of the first request or operation.

Read/Write: After the client starts a transaction, it can submit read/write operations to retrieve or modify a data item. The implementation of these operations depends on the

database engine.

Commit: By committing a transaction, the state of the database is left in consistent state. Any changes to the data items will be reflected permanently in the database.

Abort: If a transaction decides to abort for any reason (e.g., violating constraint, locks conflict), then the database engine should undo all the changes it did so far in order to leave the database consistent. The client should re-submit the same transaction if he is still interested in achieving its outcome.

Transaction properties Among the database engine's responsibility is to ensure that a transaction preserves certain properties called "ACID" properties. ACID is referring to: ***A****tomicity,* ***C****onsistency,* ***I****solation and* ***D****urability* [28]. In the following we briefly explain each property:

Atomicity: Means that the system must ensure either a transaction runs successfully (completes), or, if it does not complete, it has no effect at all on data. Hence, if a client crashes while sending the operation of transaction or if it decides to abort the current transaction, the database must undo the effects of all operations of the client's transaction being executed so far.

Consistency: Requires that the effect of each transaction maintains all database integrity constraints i.e., it moves a database from one consistent state to another that correctly models the new state of an enterprise. For example if business rules say that the sum of a certain set of accounts must be greater than a given amount, then a transaction should not violate this constraint, otherwise, the whole transaction must be aborted.

Isolation: A collection of concurrent transactions has the same effect as that of some serial running of that set. Each transaction must observe the data in the database as if no other transaction would be currently running and the final effect should be as if we run the transactions one after each other.

Durability: The result of committed transactions are permanently reflected in the database.

This property must also be fulfilled in presence of hardware and software failures such as disk failure or crash of the operating system.

2.2 Serializability

One way to improve performance is to allow transactions to run concurrently [44]. This means that while one transaction is waiting for I/O operations, the CPU can process another transaction, as I/O operations can be done in parallel with CPU activity in a computer. However, this interleaving between transactions is dangerous, and could lead to data corruption unless interleaving is controlled. Database systems must control concurrent executions to keep data consistency, using a scheduler component[32]. This component is responsible for receiving operations from user transactions and it ensures that they will be executed in such a way that the execution will be correct.

Any sequence-preserving merge of the actions of a set of transactions into a single sequence is called a history for the set of transactions. A history indicates the order in which the operations of the transactions were executed relative to each other [28, 15].

Serializability is the precise concept for correctness. It can be guaranteed by ensuring that the final state of a database (after a set of concurrent transactions commit), is as if the transactions ran in some serial order. Serializability is the strictest correctness criterion, but there are other weaker forms of correctness or isolation levels. The levels differ according to the kinds of inconsistencies they allow. This is discussed in Section 2.4 below.

Serializability theory has been developed in order to provide more compact criteria for deciding whether a history is serializable. Database researchers have developed two main serializable theories. One is *conflict serializability*, and another is *view serializability*. A history is conflict serializable if it is equivalent to a serial schedule in the sense that

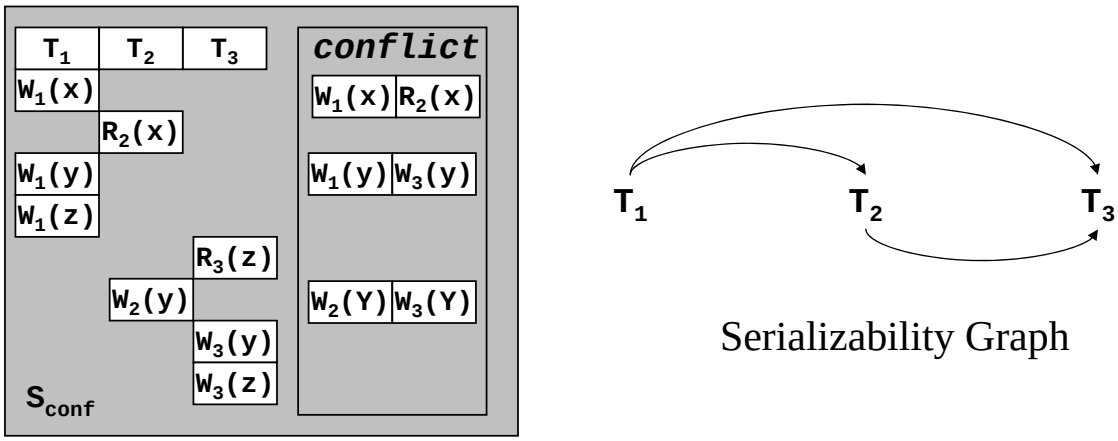

Figure 1: Conflict serializable schedule.

conflicting operations are ordered in the same way in both. We define a history *H* to be view serializable (VSR) if for any prefix *H'* of *H*, the committed projection, C(*H'*), is view equivalent to some serial history.

Theorem 1. *If* H *is conflict serializable then it is view serializable. The converse is not, generally, true [15].*

We can determine whether a certain history is serializable or not by analyzing a graph derived from the history called a serialization graph (SG). SG for particular schedule, S, of committed transactions is a directed graph in which the nodes are the transactions participating in the schedule (and there is a directed edge pointing from the node representing T_i to the node representing transaction T_j. When there exist conflicting operations I_i and I_j, and I_i occurs before I_j in the schedule.) We say instructions I_i and I_j conflict if they are operations by different transactions (I_i is an op of T_i, and I_j is an op of T_j) on the same data item, and at least one of these instructions is a write operation.

Theorem 2. *Serializability Theorem: A history* H *is conflict serializable iff SG(H) is acyclic.*

This theory implies that if a history *H* has no cycle in the SDG, then *H* is equivalent to a

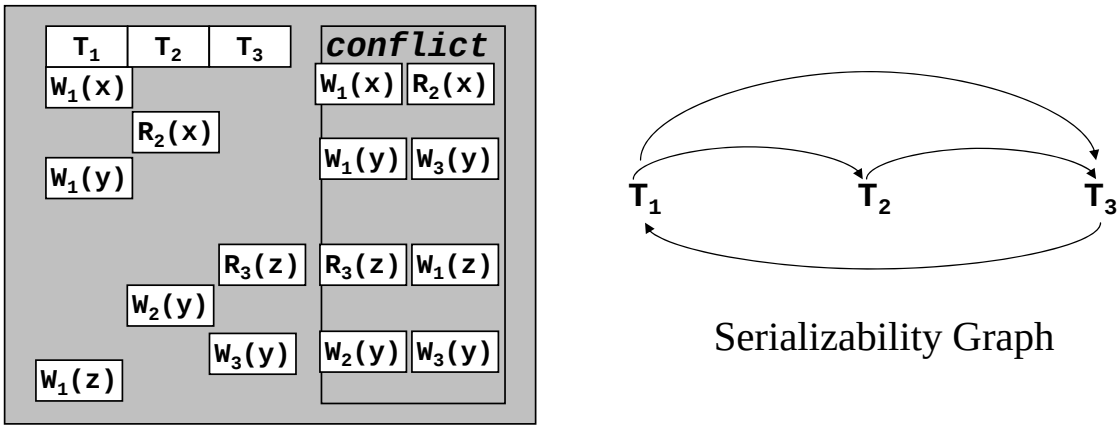

Figure 2: Non-serializable schedule.

serial history. [1]

Figure 1 shows an example where the SG(H) does not have any cycle, this implies that this schedule is serializable. Figure 2 shows another example where the schedule is non-serailizable, as SG(H) has a cycle between $T_1 \rightarrow T_3 \rightarrow T_1$.

In this thesis we use some variant definitions for multiversion systems. In the multiversion systems, each write on a data item *x* produces a new copy or version of *x*. The DBMS tells which version, chosen among the versions of *x*, to read. Serializability theory can be extended to such systems, ensure correctness. These concepts are presented in Section 2.5.

2.3 Concurrency control

Concurrency control (CC) ensures concurrent transactions will be executed at the same time with results as if they execute in sequence. Several concurrency control techniques have been proposed. Traditionally, concurrency control techniques have been classified into four categories- Locking, timestamp ordering, optimistic and hybrid.

[1] Proof of this theorem can be found in [15].

Locking: A transaction that needs to read or write a data item must acquire a lock on that item; two modes of locking are available, shared (a transaction which acquires this mode can read, but cannot write) and exclusive (a transaction which acquires this mode can read and write). All locks are implicitly released by commit of the holding transaction. Every transaction obtains its locks in a two-phase manner (growing phase, shrinking phase). During the growing phase, the transaction obtains locks without releasing any locks. During the shrinking phase, the transaction releases locks without obtaining any locks. An example to this is strict Two Phase Locking Protocol (2PL), which is used in most platforms. A basic strict 2PL scheduler follows the following rules:

1. Conflict test; when the 2PL scheduler receives a lock request, it tests whether the requested lock conflicts with the other locks that already set. If lock conflict is there, then it queues the lock request, otherwise it sets the lock.

2. During the transaction's life, the scheduler can not release any lock until the transaction commits or aborts.

Two variants of the basic 2PL are *Dynamic* and *Static* 2PL. In dynamic 2PL, a transaction obtains a lock only when it needs to access the data item, while in static 2PL, a transaction pre-declares and obtains all the locks it may need before it begins any operation or computation. Performance of locking protocols have been widely studied and investigated [8, 54, 57, 55, 31, 42].

Timestamp Ordering (TO): Each transaction is assigned a unique time stamp when it starts, and this is used to order transaction activity. A scheduler orders conflicting operations based on their timestamp. The scheduler rejects an operation if it has already executed a conflicting operation with a later timestamp. If the operation has been rejected, then its transaction must abort. A timestamp ordering technique that avoids restarts is conservative timestamp ordering (CTO); this delays operations until the system is sure

that there are no conflicting operations with lower timestamp [12]. Performance of timestamp ordering concurrency control has been studied in [8, 52, 42]

Optimistic: Multiple transactions are allowed to read and write without blocking, transactions keep histories of their data usage, and before committing a transaction checks for conflict. If any conflict is found then one of the conflicting transactions should abort. A transaction proceeds in three phases

1. Read Phase: Transaction reads and writes data from database, and saves it in private workspace.
2. Validation phase: Checks if there is any conflict between transactions.
3. Write phase: Commit the transaction in case there is no conflict, and copy new values from private workplace to the database.

The transaction may abort at a very late stage, when it has completed all its computation, thus resulting in a large amount of wasted processing. A variation on this concurrency control is adaptive optimistic concurrency control [6]. Optimistic concurrency control has been investigated in [8, 42, 36].

Hybrid: Several concurrency control methods that combine 2PL and timestamp schedulers has been discussed in [13]. In [17] Cary and Livny execute transactions using optimistic scheduler, but if a transaction aborts, they use 2PL to execute the transaction second time. In [41] a complex protocol which shares features of optimistic and multiversion concurrency control was presented. [50] claimed that these hybrid algorithms would perform better than algorithms based on a single concurrency control mechanisms. The performance of hybrid concurrency control algorithms has been investigated analytically as well as experimentally in different studies [56, 48, 53, 39]. These techniques have not been adopted in widespread platforms.

2.4 Isolation levels

The concept of isolation level was introduced under name ”Degrees of Consistency”. The most significant effort in this field was early work by Gray in [27], Gray tried to provide declarative definitions of consistency degree using locking techniques. Influenced by [27, 20], the ANSI/ISO SQL standard defines several levels of transaction isolation with differing degrees of impact on transaction processing throughput, because the database industry desires to weaken the requirement of serializability [27, 1, 32], in order to increase the performance of common multi-user application especially in cases where absolute correctness is not critical. The isolation levels are defined in terms of ”phenomena” that must be prevented between concurrency executing transactions in order to achieve the required isolation. A greater degree of concurrency and better performance can be achieved using lower levels of transaction isolation[33]. We have three types of phenomena:

1. **P1 (Dirty Read):** Transaction T_1 reads a data item *x* that was written by another concurrent transaction T_2. If T_1 reads *x* while T_2 is still active, then T_1 may read a value of *x* that never existed (as T_2 may aborted later).

2. **P2 (Non-Repeatable Read or Fuzzy Read):** Transaction T_1 reads a data item *x*, then another transaction T_2 writes into *x*. If T then reads *x* again, it will read a different value (or it will find out that *x* was deleted).

3. **P3 (Phantom):** Transaction T_1 reads a set of data elements which satisfy some search condition in the *where statement*. Another transaction T_2 then creates a new data item that satisfy the search condition. So, if T_1 repeats its read, it will get a different set of data items.

Transactions must be run at an isolation level of repeatable read or higher to prevent lost updates that can occur when two transactions each retrieve the same row, and then later

Isolation Level	Dirty Read	Non-repeatable Read	Phantom
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

Table 1: different isolation levels based on ANSI definitions [1]

update the row based on the originally retrieved values. If the two transactions update rows using a single UPDATE statement and do not base the update on the previously retrieved values, lost updates cannot occur at the default isolation level of read committed. According to ANSI definition, serializable isolation level must be truly serializable as defined in Section 2.2, as well as not allowing any of previous mentioned phenomenas.

[10] studied ANSI-SQL definitions of isolation level and provided a critique on these definitions, showing that some definitions can be interpreted ambiguously, while others are missing complectly. [10] shows that disallowing all phenomena does not imply disallowing all non-serializable schedules. They redefined phenomena P1-P3 by using operational patterns, and suggest an additional phenomenon P0 (Dirty writes) that all isolation levels should disallow.

P0 (Dirty Write): Transaction T_1 modifies a data item. Another transaction T_2 then further modifies that data item before T_1 performs a COMMIT or ROLLBACK. If T_1 or T_2 then performs a ROLLBACK, it is unclear what the correct data value should be.

An alternative definition of isolation levels was given in [7], which extended the classical data model in [15]. Those definitions are more portable because they can apply to database systems that implement concurrency control by other methods than locking, such as multi-version and optimistic systems. This approach mainly defines isolation levels based on the types of cycles that would be allowed in the serialization graph of a history.

2.4.1 Read Committed Isolation

The Read Committed (abbreviated as RC) isolation level is the default in most DBMS platforms. The usual locking implementation has the engine set a lock, which is kept in an in-memory component called the Lock Manager, before each access to a data item (typically a record in the data or in an index). Locks have the usual modes, including Shared Mode for read access and Exclusive mode for write access. In RC isolation level, exclusive locks are held as long as the transaction runs (that is, till commit or abort) but Shared locks are released early, once the transaction has finished the access to the item involved. This gives much better performance than 2PL (often throughput is 3 times greater) because updates are not blocked for as long; however it is vulnerable to lost update, inconsistent read, and other anomalies that can violate integrity constraints.
For example, consider the following history *H* under RC;
H: R_1(x, 50)..R_2(x, 50)..W_2(x, 10)..R_2(y, 50)..W_2(y, 90)..C_2..R_1(y, 90)..C_2
RC allows T_2 to write data items read by T_1 transaction. But explaining [10] if T_1 has written data item x, no other transaction should be allowed to write x until T_1 commits or aborts.

2.5 Multiversion Concurrency Control

Multi-Version Concurrency Control (MVCC) [14] is a highly developed technique for improving multi-user performance in a database. The aim of MVCC is to avoid the problem of writers which block readers and vice-versa, by making use of multiple versions of data. Multiversion concurrency control treats write operations as the creation of new versions of the database objects in contrast to the update-in-place (locking) and deferred-update (optimistic) semantics normally given to writing. Timestamps are used to return appropriate versions for read request. An advantage of maintaining multiple versions of

data item is that they may not add to the cost of concurrency control, because the versions may be needed anyway by the recovery algorithm [15]. The ANSI definitions are specified in terms of single-version history rather than multiversion histories. Therefore, Berenson et el.[10] suggested that the designer of the schema must first map the histories of such schema to single-version histories and then apply the consistency conditions. MVCC has been widely studied and investigated [16, 18, 14, 30, 29, 45, 52] under different conditions in several papers. It provides improvement in performance by allowing transactions to access previous versions of data items.

Implementing multiversion concurrency control faces several challenges and overheads; each UPDATE/DELETE operation generates a new version, the new versions kept in temporary place that leads to increase space usage and I/O's. Thus the system requires an efficient garbage collection mechanism to get rid of old versions when they are no longer needed.

There are two different approaches on how to implement multi-version concurrency control. The first approach is to store multiple versions of records in the database, and garbage collect records when they are no longer required. This is the approach used by PostgreSQL. The second approach is to keep only the latest version of data in the database, but reconstruct older versions of data dynamically as required by exploiting information within the Write Ahead Log. This is the approach used by Oracle.

2.5.1 Multiversion Serializability Theory

A multiversion timestamp ordering (MVTO) system maintain versions that could come with performance benefit. MVTO scheduler keeps the timestamps of different versions of data items, read(x) operation is executed by telling which version of x to use. The scheduler rejects a write if a read with later timestamps has already read an earlier version of the data item [15].

The operations from different transactions may be interleaved during an execution provided that the execution is 1-copy serializable (1-SR), which means it is equivalent to an execution in which transactions execute sequentially on a single copy of the objects.

Bernstein and Goodman [14] proposed a mechanism that characterize the correctness of general multiversion concurrency control algorithm. Serialization graph is modified to present the order of versions of data items that were accessed by concurrent transactions. This graph is called Multiversion Serialization Graph (MVSG).

For each data item *x*, we denote the versions of *x* by x_i, x_j, .., where the subscript is the index of the transaction that wrote the version. Thus, each write operation in an MV history is always of the form $W_i[x_i]$, where the version subscript equals the transaction subscript. A Read operation is denoted in the same way as $R_j[x_j]$.

Multiversion Serialization Graph (MVSG): For a given MV schedule *S* and a version order $<<$, the MVSG for schedule *S* and $<<$, MVSG(s,$<<$), is a graph with all edges that described by applying the definition of SG of single version schedule, with the following version order edges added: for each $R_k[x_j]$ and $W_i[x_i]$ where i, j, and k are distinct.

1. if $x_i << x_j$ then include $T_i \rightarrow T_j$;
2. otherwise include $T_k \rightarrow T_i$.

Then, the 1-copy serializability of multiversion schedule can be characterized by the acyclicity of corresponding MVSG.

Theorem 3. *MV Serializability Theorem: A multiversion schedule* S *is one-copy-serializable iff the multiversion serialization graph (MVSG) is acyclic.*

Figure 3 shows an example for a schedule of MVSG. The graph is acyclic, so this multiversion schedule is one copy serializable. Actually the edges in MVSG clearly indicated the execution order of transactions in an equivalent serial schedule.

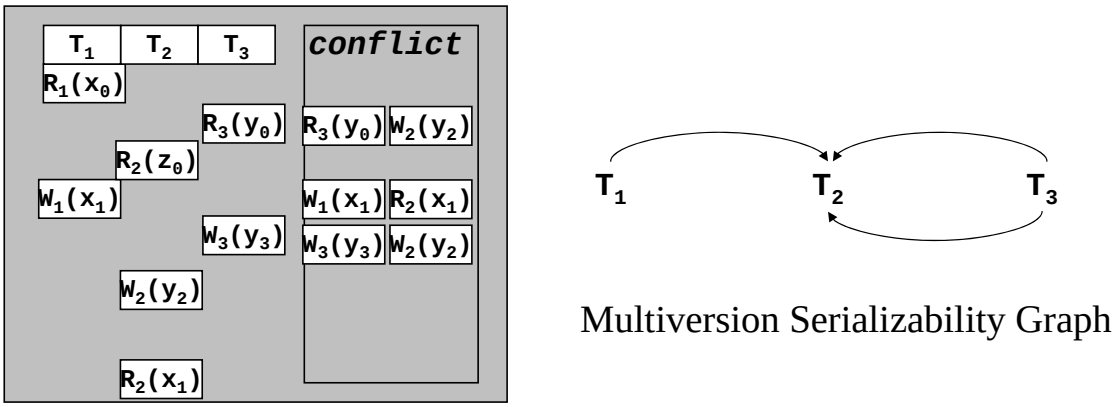

Figure 3: Non-serializable schedule.

2.6 Snapshot Isolation (SI)

Snapshot Isolation is a multi-version concurrency control mechanism used in databases. It was introduced in 1995 in [10]. 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 Two phase locking (2PL) mechanism, where many update operations could be blocked by reader even with no conflict between the update operations. 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.

First Committer Wins Implementation: The first committer wins rule can be implemented without write locks using a deferred update system. When a transaction completed, it is validated (optimistic). Validation is successful if a transaction snapshot number is greater than or equal to the version number of each item that it has updated. [2]

- Assume that we have transaction T_1 that requests to commit, and the version number that T_1 has updated is greater than T_1's snapshot number. This means that some other concurrent transaction committed a new version of the shared data item while T_1 was executing. T_1 should abort since T_2 is the first committer.
- Assume that we have T_1 that requests to commit, and the version number of data items that T_1 has updated is less than or equal to T_1's snapshot number. This means that T_1 is creating new versions, and T_1 does not conflict with other concurrent transactions. T_1 commits and increments that data item version number.

Another possible implementation is using exclusive write locks. Oracle uses this implementation to deal with conflicted data items. Assume that we have transaction T_1 request to write *x* data item.

1. if no concurrent transaction has a write lock on *x*

[2] A transaction snapshot number(TSN): it is a number used internally to determine the current system state. TSN acts as a timestamp. It is incremented when a transaction commits. When a system transaction starts, it makes a note of the current TSN. It is used to determine if the page contains the effects of transactions that should not be visible to the current transaction. Only those committed transactions should be visible whose TSN number is less than the TSN number noted by the current transaction. Also, transactions that have not yet committed should not be visible.

- If the version number of *x* is greater than T_1's snapshot number, T_1 is aborted since a concurrent transaction wrote *x*, committed and released the locks.
- If the version number of *x* is less than or equal T_1's snapshot number, T_1 is getting a write lock on *x*. The lock will be released after T_1 commits or aborts.

2. if another transaction T_1 is holding the lock on *x*, then T_1 must wait until T_1 commit or abort
   - If T_2 commits, then T_1 is aborted.
   - If T_2 aborts, then T_1 gets the lock on *x*.

In PostgreSQL, this is implemented by having each transaction set an exclusive write-lock on data it modifies, and also aborting 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" (in contrast to "First Committer Wins" described in [10]). Note that SI does not use read-locks at all, and a read is never blocked by any concurrent transaction.

SI was presented in [10] 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 [58, 40, 46]; 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 [22, 21] However, the improved performance comes not for free: while SI completely avoids the four extended ANSI SQL anomalies, it does not guarantee serializability as shown in [10]. They showed that SI could allows non-serializable executions in general as we discuss in 2.6.1 below.

2.6.1 Snapshot Isolation Anomalies

Snapshot Isolation may be associated with some anomalies such as Write Skew, and Phantom.

SI Write Skew Anomaly: Write skew is anomaly that could make application non-serializable and violate data integrity under SI. It happens when we have two or more concurrent transactions (T_1 and T_2) read a value each, then change the other's value, but neither sees the result of the other's update, under the assumption that the other remained stable (that's the reason we call it "Write Skew"). An example is shown below

T_1: R(x_0) R(y_0) W(x_1) commit

T_2: R(x_0) R(y_0) W(y_1) commit.

A well known example can be given on a bank scenario. Suppose we have two values *x* and *y* representing checking account balances of a couple at a bank, with a invariant that $x+y>0$. Thus the bank permits either account to be overdrawn, as long as the sum of the account balances remains positive. Assume 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 SI will end up in violation of the invariant $x+y>0$, since T_2 has been successfully executed even though the sum of the accounts is negative [26]. Unfortunately, this problem will not be detected by First Committer Wins because two different data items were updated.

Read-Only Anomaly: Read-Only Transaction Anomaly [26] is another problem which appears in SI. For example suppose we have *x* and *y* as two data items representing checking account balance and saving account balance. However, suppose a withdrawal is allowed to make *x*+*y* negative, but an extra one is withdrawn in that case, as penalty fee. Consider this sequence of operations where T_1 tries to deposit 20 to saving balance *y*, and T_2 subtracts 10 from the checking account *x*.

T_1: R(y=0) W(y=20) C1.

T_2: R(x=0)R(y=0) W(x=-11) C2.

T_3: R(x=0)R(y=20) C3.

Then the anomaly arises here is that read-only transaction T_3 has *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. Because if 20 were added to *y* before 10 were subtracted from *x* in any serial execution, no charge of 1 could occur. So the final result should be -10 not -11 in a serial execution where T_3 sees *x*=0, *y*=20. [25]

Phantom Anomaly: There is no agreed-upon definition in the literature of what phantom is. Some sources say that an isolation level permits phantom if, when a transaction executes the same SELECT statement twice, the second execution can return a result set containing different number of rows. Considering this definition, Phantoms are not possible under snapshot isolation since it will return the same set using the same versions. Using another definition of phantom, where inserting a new value during the life of transaction is a phantom, then snapshot isolation can have a phantom [35].

2.6.2 Multiversion Concurrency in PostgreSQL

PostgreSQL is an open source database system. It was developed from an earlier research system Postgres coded at University of California under Michael Stonebraker. PostgreSQL uses a multi-version concurrency control idea: when a row is updated, a new version of the row is created and replaces the old version in the table, but the old version is provided with pointer to the new version and marked as expired. Garbage collection collects the expired version later in the process. In order to support multi-versioning, each row has additional data recorded with it

- Xmin- The ID of the transaction that created, inserted, and updated this row
- Xmax- The ID of the transaction that created a new later version or which deleted this version.

Initially Xmin and Xmax are equal to NULL value. PG_LOG is a table where the system can track the status of a transaction. The table can contain two bits of status information for each transaction; the possible status are committed, in progress, and aborted. In case of failure, PostgreSQL does not remove the transaction's versions, instead, it marks the transaction as aborted in PG-LOG. Therefore, PostgreSQL tables can have data from aborted transactions. A vacuum operation in PostgreSQL is responsible for garbage collecting expired/aborted versions from tables and associated indexes. Indexes in PostgreSQL do not contain any versioning information.
SnapshotData is a data structure that contains a list of all active transactions at the time the snapshot is taken. Using the above information, two conditions should be satisfied for a tuple to be visible at the beginning of a transaction:

1. The Xmin (creation transaction) ID of the tuple:
    - Is a committed transaction according to PG_LOG and

- Is less than the transaction counter stored in *SnapshotData* and
- Was not in process at query start according to *SnapshotData*.

2. The Xmax (expire transaction) ID:

- Is blank or Aborted according to PG_LOG or
- Is greater than the transaction counter stored in *SnapshotData* or
- Was in process at query time accroding to *SnapshotData*.

To avoid consulting the PG_LOG table repeatedly, PostgreSQL also keeps some status bits in the table to indicate whether the tuple is already committed or aborted. These bits are updated by the first transaction that queries the PG_LOG table. Finally, PostgreSQL is very similar to multi-version timestamp ordering, since it does not use locks for DML commands [49, 2, 51].

2.6.3 Multiversion Concurrency in Oracle

Oracle is a commercial database system. Oracle's multiversion concurrency control system differs from other concurrency controls used by database vendors. It supports both statement and transaction level read consistency based on the isolation levels (Read Committed or Snapshot). Oracle does not maintain multiple versions of data on tables (as in PostgreSQL). Instead, it rebuilds older versions of data on the fly as and when required in the rollback segment. A rollback segment is a special table where undo records are stored while a transaction is in progress. Rollback segments manage their space so that new transactions can reuse storage from older transactions that have committed or aborted; this automatic facility enables Oracle to manage large numbers of transactions using a finite set of rollback segments. Modifications to rollback segments are logged so that their contents can be recovered in the event of a system crash.

Oracle uses System Change Number (SCN) to determine the current system state. SCN acts as a timestamp. SCN consists of a set of numbers that points to the transaction entry (slot) in a Rollback segment header. The System Change Number (SCN) is incremented when a transaction commits. When an Oracle transaction starts, it makes a note of the current SCN. When reading a table or an index page, Oracle uses the SCN number to determine if the page contains the effects of transactions that should not be visible to the current transaction. Only those committed transactions should be visible whose SCN number is less than the SCN number noted by the current transaction. Also, Transactions that have not yet committed should not be visible. Oracle checks the commit status of a transaction by looking up the associated Rollback segment header, but, to save time, the first time a transaction is looked up, its status is recorded in the page itself to avoid future lookups.

If the page is found to contain the effects of invisible transactions, then Oracle recreates an older version of the page by undoing the effects of each such transaction. It scans the undo records associated with each transaction and applies them to the page until the effects of those transactions are removed. The new page created this way is then used to access the tuples within it. Since Oracle applies this logic to both table and index blocks, it never sees tuples that are invalid.

Oracle records the Transaction ID that inserted or modified a row within the data page. Rather than storing a transaction ID with each row in the page, Oracle saves space by maintaining an array of unique transactions IDs separately within the page, and stores only the offset of this array with the row. Along with each transaction ID, Oracle stores a pointer to the last undo record created by the transaction for the page. The undo records are chained, so that Oracle can follow the chain of undo records for a transaction/page, and by applying these to the page, the effects of the transaction can be completely undone. Not only are table rows stored in this way, Oracle employs the same techniques when

storing index rows.

Since older versions are not stored in the DBMS, there is no need to garbage collect data. Since indexes are also versioned, when scanning a relation using an index, Oracle does not need to access the row to determine whether it is valid or not. In Oracle's approach, reads may be converted to writes because of updates to the status of a transaction within the page.

Reconstructing an older version of the page is an expensive operation. However, since Rollback segments are similar to ordinary tables, Oracle is able to use the Buffer Pool to effectively ensure that most of the undo data is always kept in memory. In particular, Rollback segment headers are always in memory and can be accessed directly. As a result, if the Buffer Pool is large enough, Oracle is able create older versions of blocks without incurring much disk I/O. Reconstructed versions of a page are also stored in the Buffer Pool. An issue with Oracle's approach is that if the rollback segments are not large enough, Oracle may end up reusing the space used by completed/aborted transactions too quickly. This can mean that the information required to reconstruct an older version of a block may not be available. Transactions that fail to reconstruct older version of data will abort. [49, 5]

2.6.4 Multiversion Concurrency in Microsoft SQL Server

Microsoft SQL Server has implemented concurrency control in two ways

1. Locking: Where traditional locking concurrency control has been used.

2. Row versions: Where Multi-version concurrency control is used, and no read locks.

If row versioning is enabled, then whenever a transaction modifies a row, the image of the row before modification is copied into a page in the version store. The version store is a collection of data pages in tempdb. If multiple transactions modify a row, multiple

versions of the row are linked in a version chain. Read operations using row versioning retrieve the last version of each row that had been committed when the transaction or statement started. These versions are garbage-collected when there are no active transactions that could require them. The tempdb database must have enough space for the version store. When tempdb is full, update operations will stop generating versions and continue to succeed, but read operations might fail because a particular row version that is needed no longer exists. When application developers decide to use multi-version concurrency control (timestamp), they actually decide not to use locks and instead use the old versions in case of conflicts. The SQL Server added a few bytes to each row to keep the following information:

- XTS (transaction sequence number). It takes 6 bytes. This is used for marking the transaction that did the DML operation on the row.
- RID (row identifier) that points to the versioned row. It takes 8 bytes.

These extra bytes are used to decide the visible and invisible data blocks at start time for a transaction. When a transaction reads a row that has a version chain, the Database Engine follows the chain and retrieves the row where the transaction sequence number is:

- Closest to but lower than the sequence number of the snapshot transaction reading the row.
- Not in the list of the transactions active when the snapshot transaction started.

Read operations performed by a transaction retrieve the last version of each row that had been committed at the time the snapshot transaction started. This provides a transactionally-consistent snapshot of the data as it existed at the start of the transaction. The transaction Uses row versions to select rows to update. It tries to acquire an exclusive lock on the

actual data row to be modified, and if the data has been modified by another transaction, an update conflict occurs and the snapshot transaction is terminated.

2.6.5 Analysis Using SDG

The experts in the Transaction Processing Council could not find any non-serializable executions when the TPC-C benchmark [4] 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 [23], and a much more extensive theory is in [25]. The latter paper proves that the TPC-C benchmark has every execution serializable on an SI-based platform. Jorwekar et al [34] have shown that one can automate the detection of some cases where the theory of [25] holds. Fekete [24] 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 [11] showed how to prove that certain programs maintain a given integrity constraint when run under a variety of weak isolation levels, including SI.

The key result of [25] 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 (Figure 4 shows the edge).

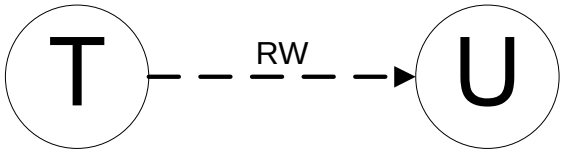

Figure 4: Vulnerable edge.

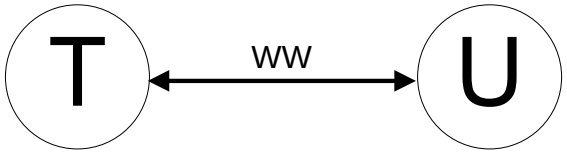

Figure 5: Write Dependencies edge

2. Write Dependencies edge (WW): We say that the edge from P to Q is *Write Dependencies* if P can give rise to transaction T, and Q can give rise to U, and T and U can execute concurrently with a write-write conflict ; One of the transaction should abort as result of FCW rule(Figure 5 shows the edge).

3. Read Dependencies edge (WR): We say that the edge from P to Q is *Read Dependencies* 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 (Figure 6 shows the edge).

Within the SDG, we say 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 7 shows a dangerous structure. The main theorem of [25] is that if a SDG has no dangerous structure, then every execution of the programs is serializable (on a DBMS using SI for concurrency control).

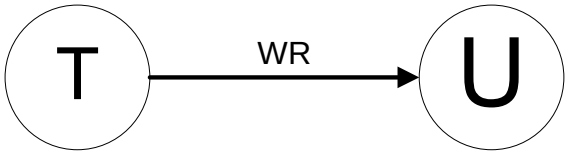

Figure 6: Read Dependencies edge.

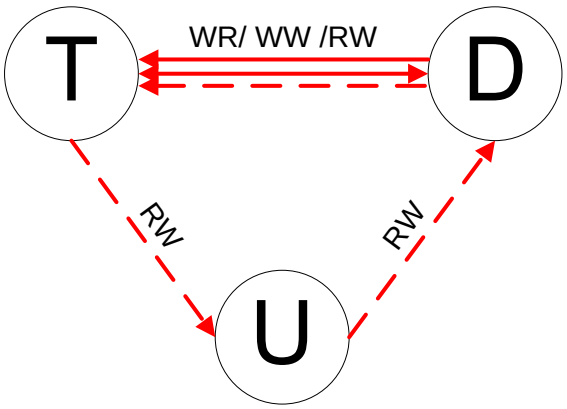

Figure 7: Dangerous structure example.

2.6.6 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 [25], 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. [34] showed that choosing a minimal set of appropriate edges is NP-hard.

Different techniques were proposed by [25] to ensure serializability using snapshot isolation. As mentioned above, the main idea behind these techniques is that we choose one edge of the two successive vulnerable edges that define a dangerous structure, and modify the programs joined by the edge, so that the edge no longer vulnerable.

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
3-   WHERE id = :x
```

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 for some parameter values, Q modifies some item in T where a condition C holds, and P contains

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

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 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

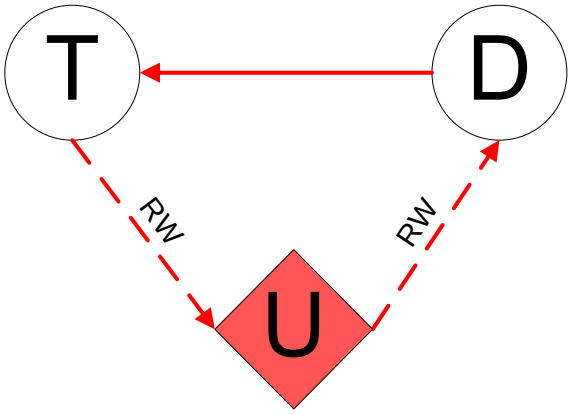

Figure 8: Pivot Example.

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 [24]. [24] defines a node as a pivot, if it has incoming and outgoing vulnerable edges (Figure 8 shows the pivot as a diamond), and the path from to original node is a chord-free-cycle. 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. In contrast, most installations insist on extensive testing before approving any changes to the application code (even ones as simple as Update x=x). In many cases, each application is a stored procedure in the database, so modification requires substantial permissions on the server; but changing isolation level happens in the client without any authorization. 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; however the explicit locks

are all of table granularity and thus will have very poor performance. We studied the performance of these methods in a platform that does offer both SI and 2PL in [9], and we found that running the pivot with strict two-phase locking has significantly worse throughput than promotion and materialize. We do not consider this technique further in this thesis.

2.7 Benchmark

A database benchmark is a way of doing a quantitative comparison of different database management systems (DBMS) in terms of performance or price/performance metrics. These metrics are obtained by means of the execution of a performance test on applications [4, 28]. Different benchmarks have been released but the most important one were developed by the Transaction Processing Council (TPC). These benchmarks has been designed to be run on computers, networks, and database of different size, from the small to the largest. Using these benchmarks, we can compute the throughput, which is shown in transaction per second or transaction per minute, and we can also compute price/performance.

Table Name	Transaction Name
Warehouse	New-order
District	Payment
Customer	Delivery
History	Order-status
New-order	Stock-level
Order	
Order-line	
Item	
stock	

Table 2: TPC-C Benchmark.

TPC-A: Defined in 1989, it is a simple banking transaction that measures the performance and the price of a computer network in addition to the database system. It simulates a typical banking application by a single type of transaction that models cash withdraw and deposit at bank teller. The database operations are a mix of main memory, random, and sequential accesses. The system definition and price includes 10 terminals per tps.

TPC-B: Was a new version of TPC-A with the terminals, network, and two-third of the long term storage removed. It's only designed to give high throughput rating and low price/performance rating to database systems. Its price/performance rating is often 10 times better than TPC-A.

TPC-C: Involves a mix of five concurrent transactions of different types and complexity either executed on-line or queued for late execution. The domain is an order and inventory system. TPC-C is more complex than TPC-A and TPC-B, it simulates realistic features for a production system, such as queued transactions, response time, and aborting. TPC-C has been approved as a standard and it is still widely used today [49]. The database contains nine types of records with a wide range of record and population sizes. Table 2 shows the transactions and the tables used in this benchmark.

TPC-D: Was designed to measure the performance of database systems on decision support queries. TPC-A, TPC-B, and TPC-C do not measure the performance of decision support queries, but measure the performance of transaction workload. TPC-D simulates a sales/distribution application.

TPC-E: This benchmark is based on a number of different transaction types that are executed on a complex database. The TPC-E benchmark measures the performance of online transaction processing systems (OLTP). TPC-E is a hardware and software independent and can thus be run on every test platform, i.e., proprietary or open. In addition to the results of the measurement, all the details of the systems measured and the measuring method must also be explained in a measurement report (Full Disclosure Report or FDR). Consequently, this ensures that the measurement meets all benchmark requirements and is reproducible. TPC-E does not just measure an individual server, but a rather extensive system configuration. Keys to performance in this respect are the database server, disk I/O and network communication.

TPC-H: TPC-H is a benchmark that support a business intelligence database environment. The performance of a system is measured when the system is tasked with providing answers for business analysis on a data set. This analysis can include pricing, promotion, demand management, shipping Management, and more.

The server system runs a read-intensive Decision Support System (DSS) style database to provide the results for the business analysis. The DSS database is designed to mimic a repository of commercial order-processing Online Transaction Processing Databases.
Different TPC benchmarks (e.g., TPC-W) has been defined as standard for database systems, more details can be found in [3]. Other benchmarks have been published; some of them were not successful because lack of general statement, while others did not obtain much help from vendors and provided only few results. In this thesis we used our own benchmarks described in Chapter 4; [25] proved that TPC-C already serializable under snapshot isolation (SI), therefore, we need benchmarks that have certain characteristics (e.g., such as having a dangerous structures, or write skew) for the thesis study.

Chapter 3

The External Lock Manager (ELM) Technique

In this Chapter 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.

Roadmap: This chapter is structured as follows: In Section 3.1 we present the ELM approach. We describe the proof of ELM serializability in Section 3.2. In Section 3.3 we illustrate the architecture and design of ELM. ELM prototype implementation is covered in Section 3.5. Section 3.7 summarizes the chapter.

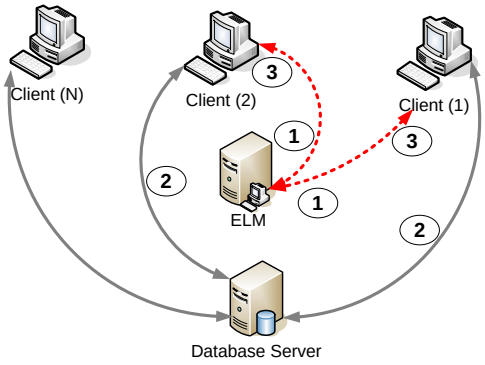

Figure 9: System Architecture with ELM.

3.1 The ELM Approach

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 9. 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 Chapter 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, [1] 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 (for example, setting a lock on a primary key for the account), we can achieve fine grained 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 locking.

[1] We use our own ELM locks, rather than the locking available directly in the database engine, to get fine-grained exclusion. While most platforms use record-level locks for automatic locking, they typically offer user-controlled locks only at table-granularity (eg SET TABLE LOCK ON tablename).

3.1.1 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 simple 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 column (*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:integer) reads a value from $Table_1$ using the parameter *x* to satisfy the where statement. The essential SQL in P_1 is:

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

Similarly, T_2(*y:integer*) updates $Table_1$ using the parameter y to satisfy the where statement. Its SQL is:

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

If 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 positive.

- **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 *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 (*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 positives 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 our experiments in Chapter 5. 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...x_n) and T_2 passes another set of parameters (e.g., y_1,y_2...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 y_2.

Finding the minimum set is quite easy with a simple set of parameters. However, if the set of parameters are complex and big, it is more difficult. Further research is still needed In this topic.

- **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. In our implementation, we do not use such super-fine locking. Instead we set the ELM locks on the parameters *x* or *y* respectively. This slightly imprecise choice of lock actually can have some benefits for performance. Setting a lock just on the primary key of the item involved in the read-write dependency also introduces a conflict between two instances of T_2 if they share the parameter value; this conflict is not necessary for correctness, since the transactions would have a write-write conflict anyway, but the ELM conflict leads to waiting, whereas otherwise one of the two instances would be aborted by FCW, and then restarted. Thus the blocking of ELM can reduce aborts compared even to the unmodified programs.

3.2 Proof of ELM Serializability

The proof that the ELM algorithm ensures serializable execution is immediate from the main theorem of [25].

Proof. Suppose we have a history *H* execution under SI, and suppose that history is not

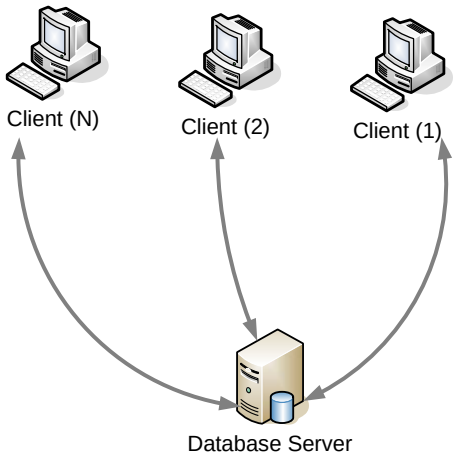

Figure 10: System Architecture for Experiments without ELM.

serializable. Then *H* has a dangerous structure such as $T_1 \dashrightarrow T_2$ is *rw-vulnerable edge* and $T_2 \dashrightarrow T_3$ 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 not any more valid. Using the theorem [25] which insists that the absence of dangerous structure ensures serializbility with SI, we guarantee that the history *H* is serializable under SI using ELM. □

3.3 Architecture And Design of ELM

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. This is illustrated in Figure 10. 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.

3.3.1 Design Features

We believe that introducing conflict on an edge by using ELM locks has considerable potential advantages compared to the previous approaches in Chapter 2 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). 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 [49]. 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 [47].

  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.

- **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;[2] that is, we code each application that needs multiple

[2]Another approach to avoid deadlock is if each transaction requests its locks all at once in a single

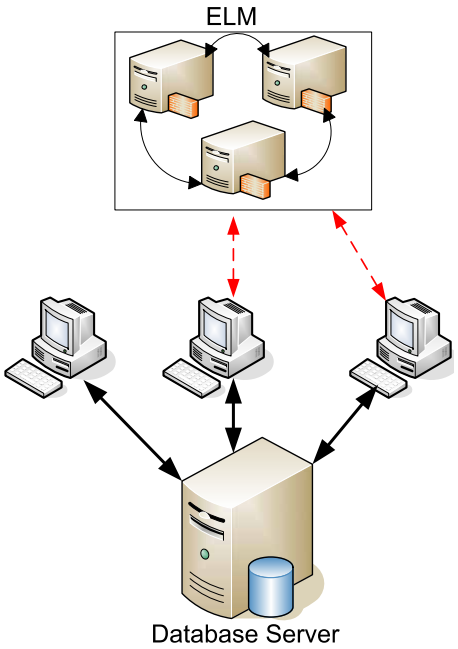

Figure 11: Architecture of ELM-Failure.

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

- **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 3.3.2), 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

interaction with the ELM lock manager. We only implemented the first approach.

is when we have the ELM as separate component as shown in Figure 9. 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 shown that in many modern systems, the network communication times are less than the disk access times [37, 43].

- **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. We discuss the issue of fault tolerance in Section 3.4.

- **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 [28]. 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.

There are another additional drawbacks to which we did not pay much attention, and need further research: the additional task switching by the operating system that may

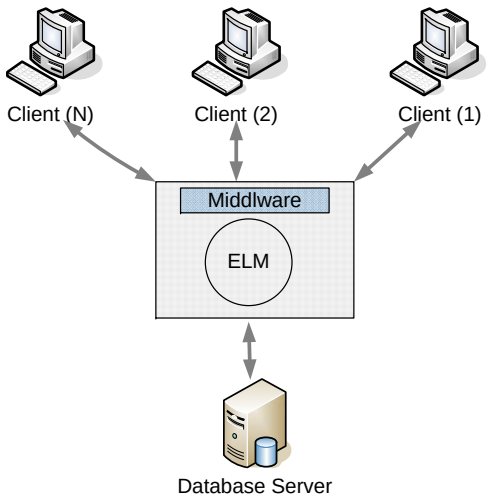

Figure 12: ELM Middleware Design.

result, and increased susceptibility to partial failure of the network.

3.3.2 Location of ELM

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

- **Separate Machine:** We can implement the ELM on a separate machine as in Figure 9. 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 (Chapter 5).

- **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. Figure 12 shows this design. 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. Figure 13 shows this design.

3.4 ELM Fault Tolerance

Fault tolerance is a major concern in transaction processing. Applications such as flight-reservation systems and real-time market data feeds must be fault-tolerant. This means that important services remain available in spite of the failure of part of the computers on which the servers are running (high availability), and that no information is lost or

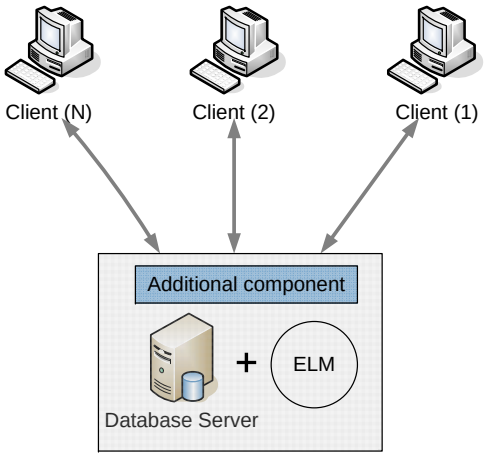

Figure 13: Additional Component Inside the database.

corrupted when a failure occurs (consistency). If we consider a distributed system as a collection of servers that communicate with each other and their clients, they not adequately providing services mean that servers, communication channels, or possibly both, are not doing what they are supposed to do.

Including ELM in the system design introduces extra failure modes. We discuss some of these, and also mention some possible ways to mitigate the failures; however the experiments reported later in Chapter 5 were done on a system without any fault-tolerance (and they measure executions without failures). If a lock request or release message is lost, or if an acknowledgement is lost, the system can be blocked. To mitigate this, one would introduce reliable messaging with retransmission. If the ELM component fails, we either leave the system blocked, or else we need to ensure that all locks are re-acquired before lock request processing resumes in a replacement ELM. Information about locks which were held could be obtained by contacting the clients (using information from the DBMS engine to identify them), or by keeping a persistent log in the ELM, or by a combination of these methods. If a client fails, the ELM will need to release any locks held for that client. Failure of the DBMS does not affect the ELM at all besides releasing the locks of the failed transactions. Another way to make the ELM design fault tolerant would be to replicate the ELM over several machines kept in consistent states. This ”state-machine

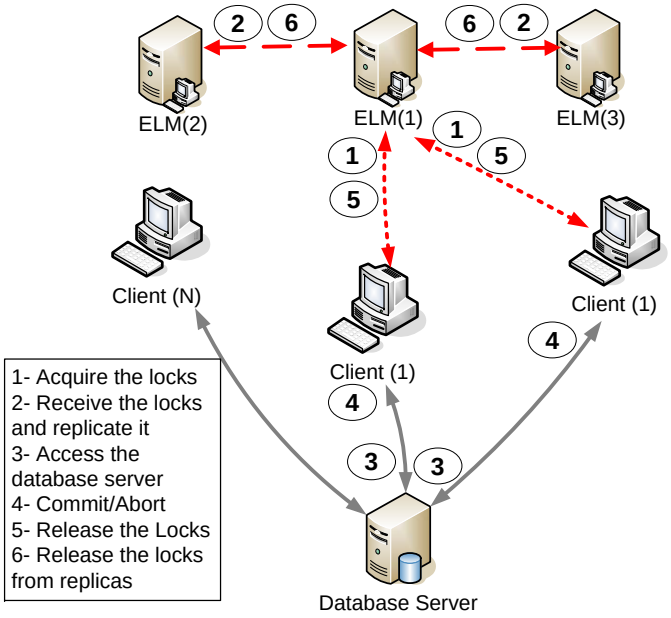

Figure 14: Client-Server Architecture with Fault Tolerance.

replication" is a well-known technique in distributed systems [49, 28, 38, 15, 16]. Here we discuss the fault tolerance based on the ELM location considering three types of failures.

1. The ELM failure.
2. Network failure.
3. Client failure.

- **Separate Machine:** Using this architecture, the ELM can be replicated, which means it is provided redundantly on multiple computers (Figure 14). The replication algorithm uses a leader elected from the set of nodes. The other replicas keep up-to-date with the leader, ready to take over when needed [38].

  On failure. If the ELM fails, another replica is elected to perform as new ELM. This can eliminate the first type of failure. Network failure can be handled by distributing the replicas on different locations, so if one connection fails, the system can automatically use one of the other connections. However, using replicas will

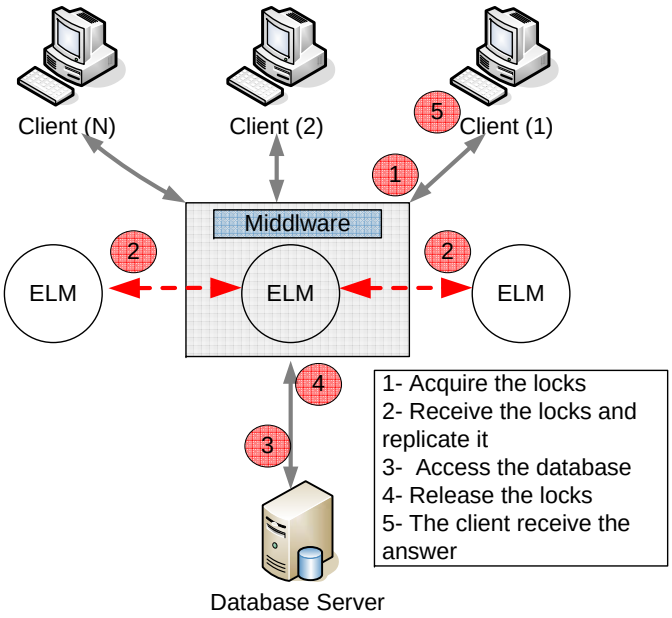

Figure 15: Middleware Architecture with Fault Tolerance.

not prevent the system from client's failure; assume a client program fails after transaction completes but before releasing the locks, the other clients that try to obtain a conflict lock will hang without any chance for success.

On recovery. When the ELM recovers from a crash, it requests a full transfer of lock state from the current ELM leader.

- **Middleware:** Several studies have been performed on middleware fault tolerance. In our middleware architecture, we can use the same previous idea to deal with fault tolerance. The ELM Middleware combination can be replicated across a small set of nodes. The replication algorithm uses a leader elected from the set of nodes [58, 40, 21, 22].

  On failure. When the ELM fails, another replica can be used to ensure the availability of the ELM services. Network failure can be handled by distributing the replicas on different locations, so if one replica fails, the system can automatically use one of the other connections.

  The advantage of using the middleware over the separate machine architecture that

failing of a client after submitting the transaction will not effect other transactions since the middleware is responsible of releasing the lock after it receive the result from database server (Figure 15).

On recovery. When the ELM middleware node recovers from a crash, it requests an update from the leader.

- **Additional Component**: The database server uses standard recovery schemas, redoing/undoing transactions in the database log as necessary.

  On failure. If the database server fails, the clients hang until the database server restarts. The previous status of the ELM does not need to be maintained since the server itself has failed and all active transactions will be terminated.

  On recovery. After the database server recovers, nothing need to be done for the ELM component. The ELM starts a new set of lock/release operations.

3.5 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:

```
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.

```
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.

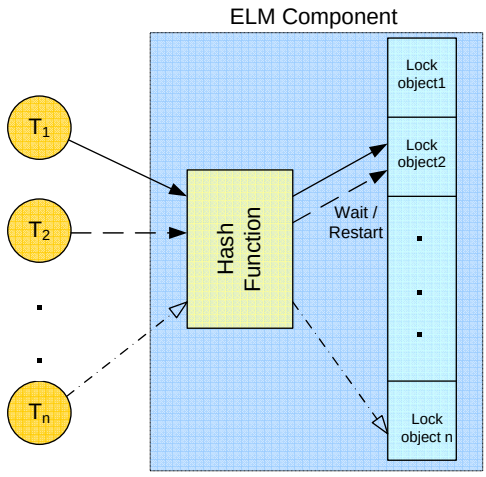

Figure 16: Implementation Example.

3.6 Implementation of Lock Manager

Within ELM, locks are managed by the usual techniques from [28], 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 16 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, in our experiments 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.

```
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;
6-}
7-while (holder != null)
8-try {
9-if (noWait)
10-return LOCK_FAILED;
11-this.wait();
12-} catch (Exception e) {
13-// ignore it
14-}
15-holder = locker;
16-this.key = key;
17-refcount = 1;
18-return LOCK_NEW;
19-  }
```

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.

```
1-   public synchronized void release() {
2-//System.out.println("Unlocking " + key);
3-if (refcount == 1) {
4-holder.held.remove(this);
5-holder = null;
6-key = null;
7-this.notify();
8-}
9---refcount;
10-  }
```

Now when we acquire multiple looks, 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.

```
1-  public Lock[] getLocks(String[] keys, boolean noWait)
2-  throws RemoteException {
3-  /* Put the keys into hash bucket order to avoid deadlock. */
4-Arrays.sort(keys, new Comparator() {
5-public int compare(Object o1, Object o2) {
6-int k1 = LockManagerImpl.getLockNum((String)o1);
7-int k2 = LockManagerImpl.getLockNum((String)o2);
8-return k2 - k1;
9-}
```

```
10-public boolean equals(Object o) {
11-return false;
12-}
13-});
14-/* Return the locks in reverse order so they are fully nested. *
15-Lock[] locks = new Lock[keys.length];
16-for (int i = 0; i < keys.length; i++)
17-locks[(keys.length - 1) - i] = getLock(keys[i], noWait);
18-return locks;
19-  }
```

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.

3.7 Summary

This chapter details our new component ”External Lock Manager’ (ELM). ELM is a new component that allows the application to ensure serializability with SI using a lock and block technique outside the database, without changes to the database engine. We presented the ELM-algorithm and different ways to implement it. The architectural designs

of ELM haves been described with their pros and cons. For experimental evaluation, we mentioned the implementation prototype for ELM.

In the following chapter we explain the experimental framework for our experiments in detail.

Chapter 4

Experimental Framework

This chapter 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.

Roadmap: This chapter is structured as follows: In Section 4.1 we present the software and hardware used in the experiments. We describe the performance metrics in Section 4.2. In Section 4.3 we describe the workload parameters that we vary in our experiments. Two benchmarks used in this thesis are explained in Section 4.4. Section 4.5 summarizes the chapter.

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. Because we are investigating attempts to avoid data corruption, 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 = 1ms, thus taking advantage of group commit.

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) and communicates using Java Remote Method Invocation (Java RMI). 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 actual 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. It emulates a varying number of concurrent clients (the multiprogramming level, MPL) using multiple threads.

Note that a single ELM is shared among all the clients, which may have different JVMs for the application programs. 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.

We use two DBMS platforms: one is PostgreSQL 8.2 which an open source database engine, so we can benefit from implementation details, and the second is Oracle 10g, which 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.

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. 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 thesis 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 (in PostgreSQL this is indicated by a Serialization Failure exception).

4.3 Workload Parameters

We vary number of parameters that can affect the overall throughput in our experiments. Some of these parameters are:

- *Data contention*: We designed our experiments 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) to produce realistic contention patterns. We consider 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 measures 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.

- *DBMS Platforms*: We mainly used the PostgreSQL 8.2 platform, but to ensure that our results are not platform specific, we ran some experiments on Oracle 10g as a commercial platform to support our conclusions.

- *Multiprogramming Level (MPL)*: 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.

- *Disk Write Cache(On/Off)*: Enabling the write cache allows the drive to do write-back caching. This can improve the overall performance by reducing the mean response time. However, it will increase the chance of data corruption and data

loss if the system crashes. Therefore, we disable this feature to ensure that our data is recoverable is case of failure. In all of the displayed results we use disk write cache off.

- *AutoCommit(True/False)*: If AutoCommit is set to True, all the data operations that modify data in the database are automatically committed after the statement is executed. On the other hand, If AutoCommit is set to False, you need to use the transaction methods (BeginTrans, CommitTrans, and Rollback) to control transactions. To execute the business logic as one using autocommit, we found that autocommit is platform specific; for example in PostgreSQL we set the autocommit to true, where in Oracle and SQL Server we set it to false.

4.4 Benchmarks

Usually, performance measurements use a standard benchmark such as TPC-C [4] 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 [25]. Thus in this thesis we have used new benchmark mixes which are contrived to offer a diverse choice among modifications that will ensure serializable execution on SI.

4.4.1 Smallbank benchmark

SmallBank benchmark is based on the example of an SI anomaly from [26], 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 benchmark 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.[1]

Programs 1 to 3 show the SQL statements for these tables in PostgreSQL.

It is generally considered worthwhile to create a table with WITHOUT OIDS since it

Program 1 Account(Name, CustomerID) using PostgreSQL.

```
-- Table: sitest.account

-- DROP TABLE sitest.account;

CREATE TABLE sitest.account
(
  name character varying NOT NULL,
  custid integer UNIQUE NOT NULL,
  CONSTRAINT account_pkey PRIMARY KEY (name)
)
WITHOUT OIDS;
ALTER TABLE sitest.account OWNER TO sitester;
```

will reduce OID consumption and thereby postpone the wraparound of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which makes them considerably less useful. In addition, excluding OIDs from a table

[1]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 cusomers. Then, by making CustID the primary keyof 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 2 Saving(CustomerID, Balance) using PostgreSQL.

```
-- Table: sitest.saving

-- DROP TABLE sitest.saving;

CREATE TABLE sitest.saving
(
  custid integer references account(custid),
  bal real DEFAULT 0.0,
  CONSTRAINT saving_pkey PRIMARY KEY (custid)
)
WITHOUT OIDS;
ALTER TABLE sitest.saving OWNER TO sitester;
```

Program 3 Checking(CustomerID, Balance) using PostgreSQL.

```
-- Table: sitest.checking

-- DROP TABLE sitest.checking;

CREATE TABLE sitest.checking
(
  custid integer references account(custid),
  bal real DEFAULT 0.0,
  CONSTRAINT checking_pkey PRIMARY KEY (custid)
)
WITHOUT OIDS;
ALTER TABLE sitest.checking OWNER TO sitester;
```

reduces the space required on disk to storage the table by 4 bytes per row, leading to increased performance.

Transaction Mix

The SmallBank benchmark runs instances of five transaction programs. These transactions are:

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 4 shows Balance transaction using PostgreSQL, and Program 5 shows it using Oracle.

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. Program 6 shows the essential SQL of DepositChecking transaction.

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. Program 7 shows the core of the SQL.

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. Program 8 shows the core of the SQL.

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

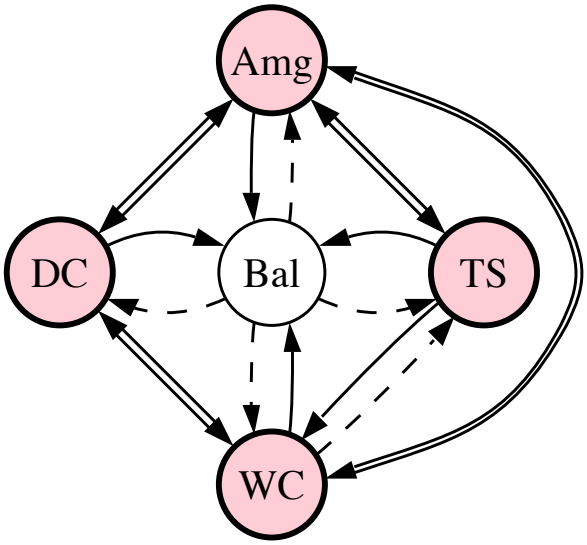

Figure 17: The SDG for the SmallBank benchmark.

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. Program 9 shows the core of the SQL code.

The SDG for SmallBank

Figure 17 shows the SDG for the SmallBank benchmark. We use dashed edges to indicate vulnerability, and we shade the 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

Program 4 Balance(N) transaction using PostgreSQL.

```
-- 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;
```

Program 5 Balance(N) transaction using Oracle.

```
CREATE OR REPLACE PROCEDURE Balance(n IN STRING , total OUT REAL)
IS     cid INTEGER;
       a REAL;
       b REAL;
BEGIN

       SELECT custid INTO cid
        FROM account
       WHERE name=n;

       SELECT bal INTO a
        FROM saving
       WHERE custid=cid;

       SELECT bal INTO b
        FROM checking
       WHERE custid=cid;

       total:=a+b;

   /* no commit needed after here when used with AutoCommit() */


EXCEPTION
   WHEN NO_DATA_FOUND THEN  -- catches all 'no data found' errors
     ROLLBACK; -- free lock
        raise_application_error(+100,'Balance: customer ' || n ||
' not found.');
END;
/
```

Program 6 DepositCecking(N,V) transaction.

```
SELECT CustomerId INTO :x
  FROM Account
 WHERE Name=:N;

SELECT Balance INTO :b
  FROM Checking
 WHERE CustomerId=:x;

 UPDATE Checking
  SET Balance = Balance+:V
   WHERE CustomerId=:x;

COMMIT;
```

Program 7 TransactSaving(N,V) transaction.

```
SELECT CustomerId INTO :x
  FROM Account
 WHERE Name=:N;

SELECT Balance INTO :a
  FROM Saving
 WHERE CustomerId=:x;

UPDATE Saving
  SET Balance = Balance+:V
   WHERE CustomerId=:x;

COMMIT;
```

Program 8 Amalgamate(N1,N2) transaction.

```
SELECT CustomerId INTO :x
  FROM Account
 WHERE Name=:N1;

 SELECT CustomerId INTO :y
  FROM Account
 WHERE Name=:N2;

SELECT Balance INTO :a
  FROM Saving
 WHERE CustomerId=:x;

SELECT Balance INTO :b
  FROM Checking
 WHERE CustomerId=:x;

Total := :a+:b;

UPDATE Saving
 SET Balance = 0.0
  WHERE CustomerId=:x;

UPDATE Checking
 SET Balance = 0.0
  WHERE CustomerId=:x;

UPDATE Checking
 SET Balance = Balance + :Total
  WHERE CustomerId=:y;

COMMIT;
```

Program 9 WriteCheck(N,V) transaction.

```
SELECT CustomerId INTO :x
  FROM Account
 WHERE Name=:N;

SELECT Balance INTO :a
  FROM Saving
 WHERE CustomerId=:x;

SELECT Balance INTO :b
  FROM Checking
 WHERE CustomerId=:x;

IF (:a+:b) < :V THEN
    UPDATE Checking
       SET Balance = Balance-(:V+1)
     WHERE CustomerId=:x;
ELSE
    UPDATE Checking
       SET Balance = Balance-:V
     WHERE CustomerId=:x;

END IF;

COMMIT;
```

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)* $\dashrightarrow$ *WriteCheck (WC)* $\dashrightarrow$ *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 [26], in which Bal sees a total balance value which implies that a overdraw penalty would not be charged, but the final state shows such a penalty because WC and TS executed concurrently on the same snapshot.

Ways to Ensure Serializable Executions for SmallBank

We have two options 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). We further have three alternatives on how to make each option non 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 *MaterializeWT*. 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 *PromoteWT-upd*. To be precise, PromoteWT-upd 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 (Oracle) we consider, there is also a strategy *PromoteWT-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
4-    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();). The lock-choice are based on the parameter-Value technique discussed in 3.1.1. 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.
4-  try {
5-    cstmt.setString(1, names[counter]);
6-    cstmt.execute();  //Execute the transaction.
7-    con.commit();
8-  } finally {
9-    l.release();  //Release the locks.
10-  }
```

And here is the client for modified TransactSaving.

```
1-  cstmt  = con.prepareCall
2-            ("{call TransactSaving(N,V)}");
3-  Lock l = locker.getLock(names[counter]; //To acquire locks.
4-  try {
5-    cstmt.setString(1, names[counter]);
6-    cstmt.execute();  //Execute the transaction.
7-    con.commit();
8-  } finally {
9-    l.release();  //Release the locks.
```

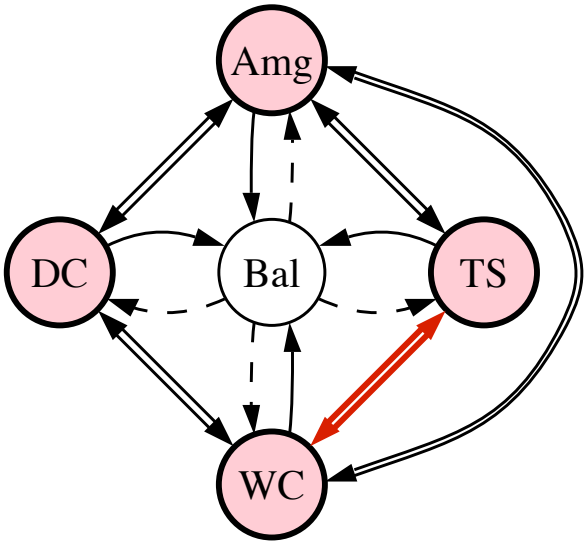

Figure 18: SDG for Option promoteWT and MaterializeWT.

```
10-   }
```

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

In Figure 18, we show the SDG for promote and materialize the WT options, Figure 19 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
```

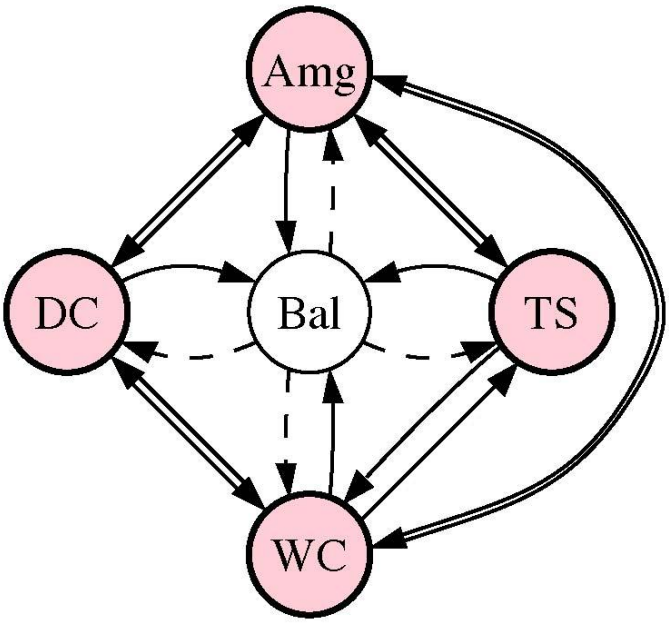

Figure 19: SDG for Option ELM-WT.

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

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

Or (in the commercial platform only) we can promote with select-for-update on table Checking in Bal, and we call this *PromoteBW-sfu*.

```
1- SELECT Balance INTO :b
2-   FROM Checking
3-  WHERE CustomerId=:x
4-    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.execute();
11-con.commit();
12-}finally {
13-l.release();
14-         }
15-}
```

In Figure 20 to 22 are the SDGs for PromotionBW, MaterializeBW and ELM-BW technique. Note that the Balance transaction is no longer read-only in Figure 20 and 21, other outgoing edges from Balance have changed.

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 vulnerable edges. This can be done by considering each pair of transactions, and deciding 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*.

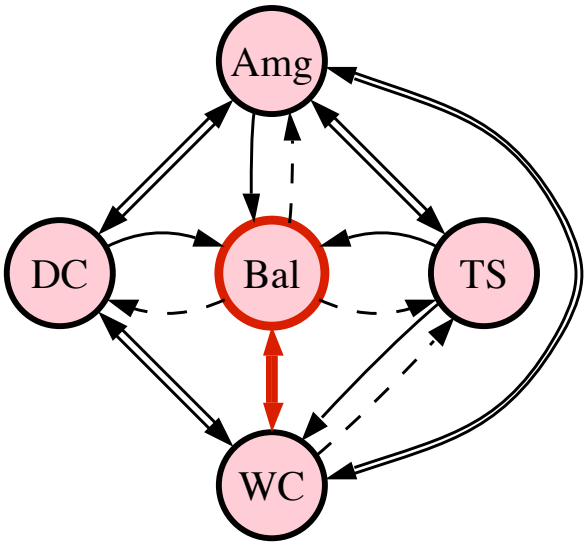

Figure 20: SDG for MaterializeBW.

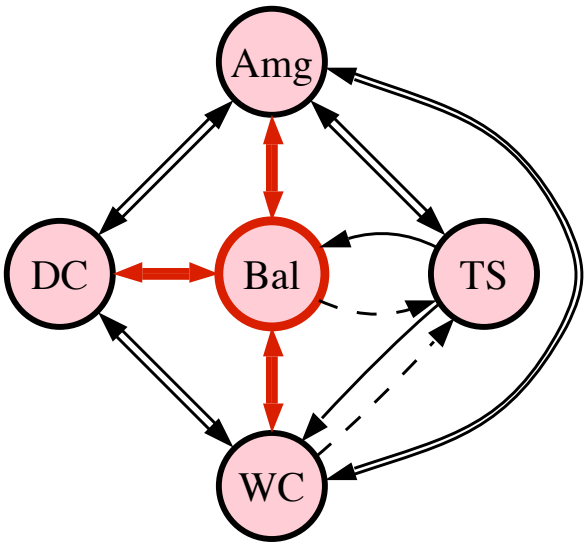

Figure 21: SDG for PromoteBW-upd.

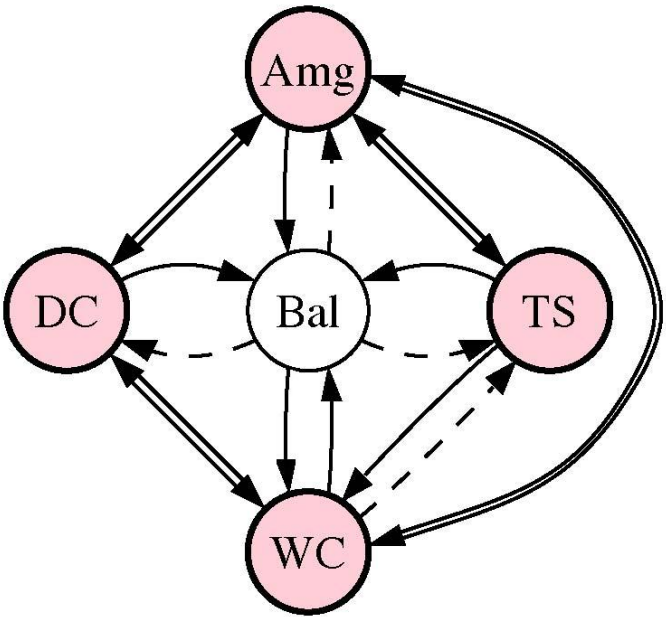

Figure 22: SDG for Option ELM-BW.

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. [2]

Table 3 summarize the different options which we compare. It lists for each option to ensure serializable executions, and for each type of transaction, which modifications are

[2]ALL transactions use customer name ”Name[counter]” parameter to control the concurrent update. Amg transaction needs to lock two parameters ”names[account1], names[account2]”.

Table 3: Overview of Modification Introduced with each Option.

Option / TX	**Bal**	**WC**	**TS**	**Amg**	**DC**
ELM-BW	Lock	Lock			
ELM-WT		Lock	Lock		
ELM-ALL	Lock	Lock	Lock	Lock	Lock
PromoteBW	Chk				
PromoteWT		Sav			
PromoteALL	Chk,Sav	Sav			
MaterializeBW	Cnf	Cnf			
MaterializeWT		Cnf	Cnf		
MaterializeALL	Cnf	Cnf	Cnf	Cnf	Cnf

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).

4.4.2 MoreChoices Benchmark

The SmallBank benchmark 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 benchmark 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), Table2(ID, Value2). Programs 10 to 12 show the SQL statements for these tables using PostgreSQL.

Program 10 Table0(CharID, Id) using PostgreSQL.

```
-- Table: sitest.Table0

-- DROP TABLE sitest.Table0;

CREATE TABLE sitest.Table0
(
  name character varying NOT NULL,
  custid integer,
  CONSTRAINT table0_pkey PRIMARY KEY (name)
)
WITHOUT OIDS;
ALTER TABLE sitest.Table0 OWNER TO sitester;
```

Program 11 Table1(ID, Value1) using PostgreSQL.

```
-- Table: sitest.Table1

-- DROP TABLE sitest.Table1;

CREATE TABLE sitest.Table1
(
  custid integer NOT NULL,
  val1 real DEFAULT 0.0,
  CONSTRAINT table1_pkey PRIMARY KEY (custid)
)
WITHOUT OIDS;
ALTER TABLE sitest.table1 OWNER TO sitester;
```

Program 12 Table2(ID, Value2) using PostgreSQL.

```
-- Table: sitest.Table2

-- DROP TABLE sitest.Table2;

CREATE TABLE sitest.Table2
(
  custid integer NOT NULL,
  val2 real DEFAULT 0.0,
  CONSTRAINT table2_pkey PRIMARY KEY (custid)
)
WITHOUT OIDS;
ALTER TABLE sitest.Table2 OWNER TO sitester;
```

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 13 shows the core of the SQL code.
- Transaction2(T_2): T_2 reads Table1,Table2 and it updates Table1. Program 14 shows the core of the SQL code.
- Transaction3(T_3): T_3 reads Table2 and updates Table2. Program 15 shows the core of the SQL code.
- Transaction4(T_4): T_4 reads Table1, and Table2 and it updates Table2 in order to create write skew with T_2. Program 16 shows the core of the SQL code.

Figure 23 shows the SDG for MoreChoices benchmark. We use dashed edges to indicate vulnerability, and we shade the nodes representing update transactions. Computing

Program 13 Transaction1(N) transaction.

```
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;
```

Program 14 Transaction2(N,V) transaction.

```
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;

UPDATE table1
  SET val1 = val1 - v
 WHERE custid=cid;

COMMIT;
```

Program 15 Transaction3(N,V) transaction.

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

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

UPDATE table2
  SET val2 = val2 + v
 WHERE custid=cid;

COMMIT;
```

Program 16 Transaction4(N,V) transaction.

```
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;

UPDATE table2
  SET val2 = val2 + v
 WHERE custid=cid;

COMMIT;
```

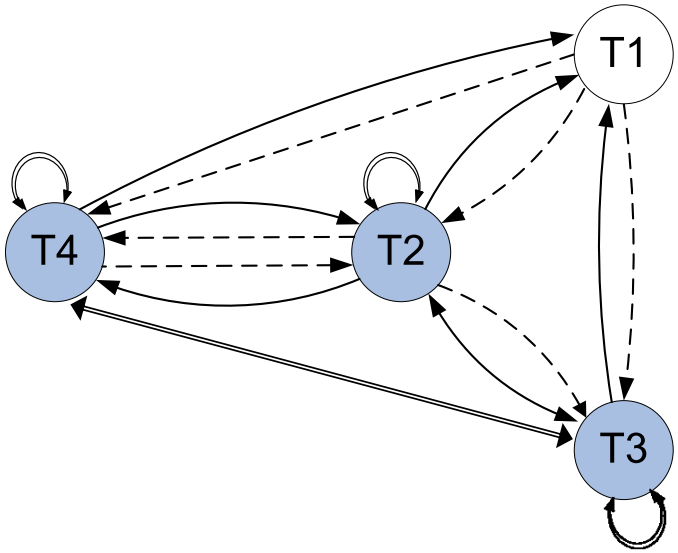

Figure 23: SDG for MoreChoices benchmark.

the SDG is very similar to analyzing SmallBank benchmark. As explained in Chapter 2 we consider an edge to be vulnerable when we have read-write without write-write conflict between the same pair. For example we have analyzed the MoreChoices SDG by hand following the approaches described in Chapter 2. We have found five dangerous structures.

1. $T_1 \dashrightarrow T_2, T_2 \dashrightarrow T_4, T_4 \rightarrow T_1$.
2. $T_4 \dashrightarrow T_2, T_2 \dashrightarrow T_3, T_3 \rightarrow T_4$.
3. $T_2 \dashrightarrow T_4, T_4 \dashrightarrow T_2$.
4. $T_1 \dashrightarrow T_2, T_2 \dashrightarrow T_3, T_3 \rightarrow T_1$.
5. $T_1 \dashrightarrow T_4, T_4 \dashrightarrow T_2, T_2 \rightarrow T_1$.

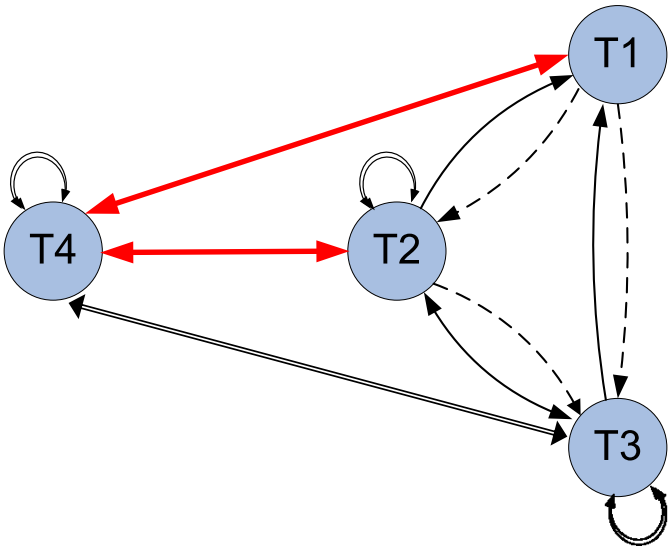

Figure 24: SDG for MoreChoice, Choice1-Materialize.

Ways to ensure serializablity with MoreChoices benchmark:

We consider three options of edge set to deal with. We further have three alternatives on how to make each option non vulnerable (Promotion, Materialize, and ELM [3]). 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 ALL vulnerable edges.

- Choice1: Removing the vulnerable edges $\{ T_1 \dashrightarrow T_2$, and $T_4 \dashrightarrow T_2\}$. Figure 24, 26, 25 show the SDG for MoreChoice benchmark after we materialize, promote, and using ELM with choice1 edges. [4]

- Choice2: Removing the vulnerable edges $\{ T_2 \dashrightarrow T_4$, $T_4 \dashrightarrow T_2$, and $T_2 \dashrightarrow T_3\}$. Figure 27 and 28 show the SDG for MoreChoice benchmark after we materialize,

[3] 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.

[4] Note that T_1 is not read-only transaction any more after we promote or materialize the edge $T_1 \dashrightarrow T_2$.

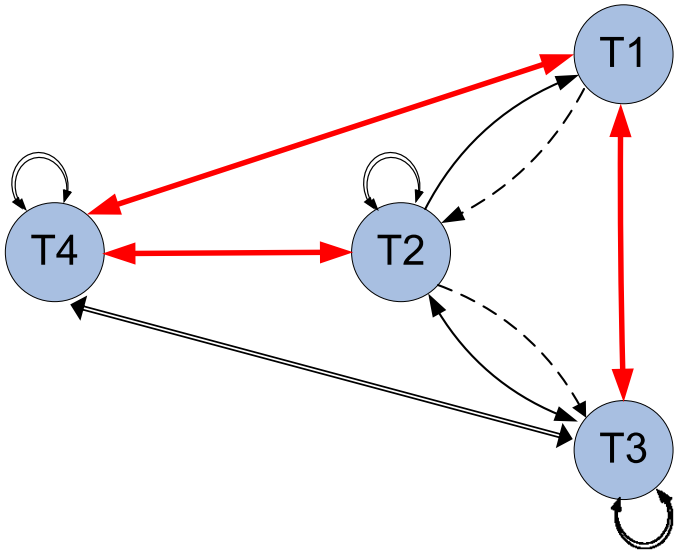

Figure 25: SDG for MoreChoice, Choice1-Promotion.

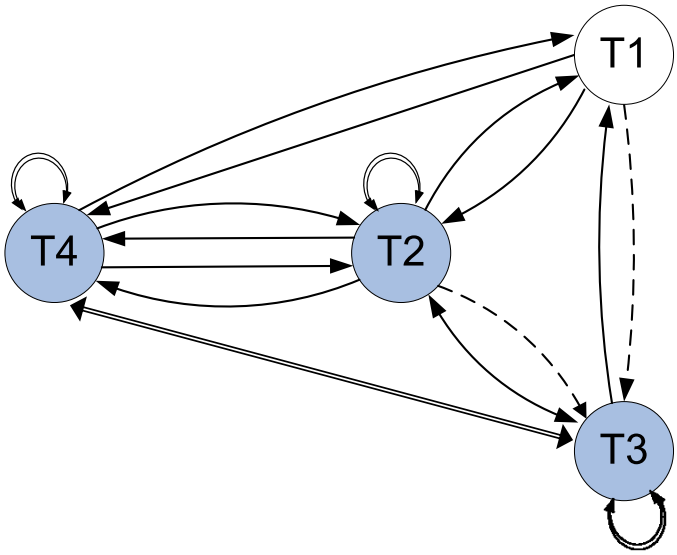

Figure 26: SDG for MoreChoice, ELM-Choice1.

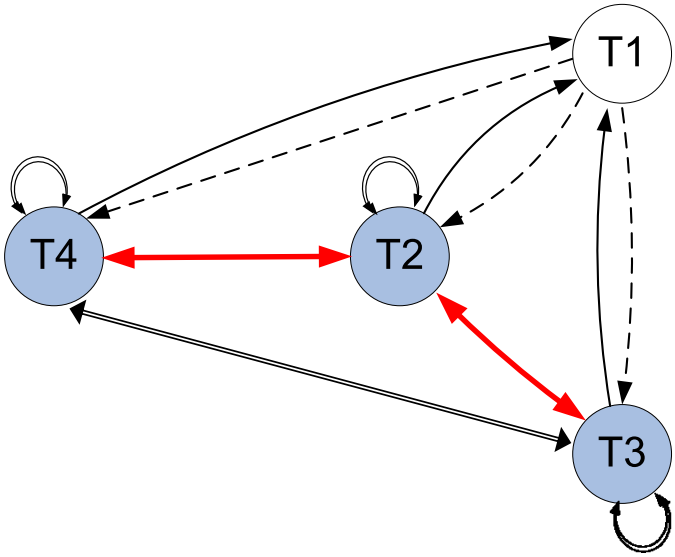

Figure 27: SDG for MoreChoice, Choice2-Promotion and Choice2-Materialize.

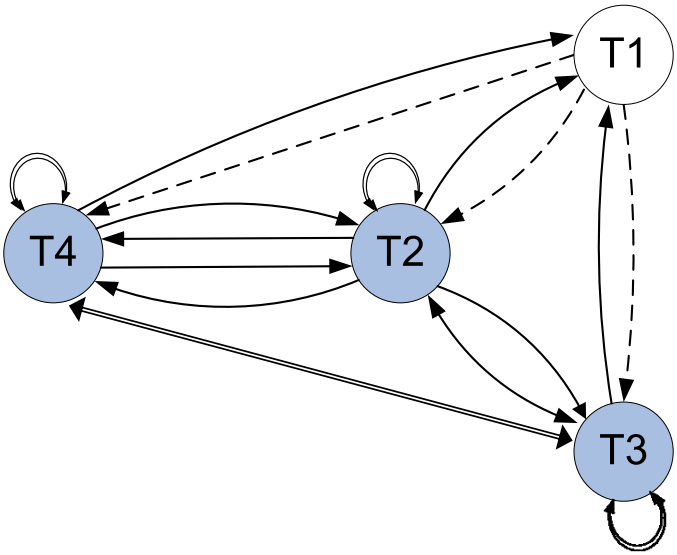

Figure 28: SDG for MoreChoice, ELM-Choice2.

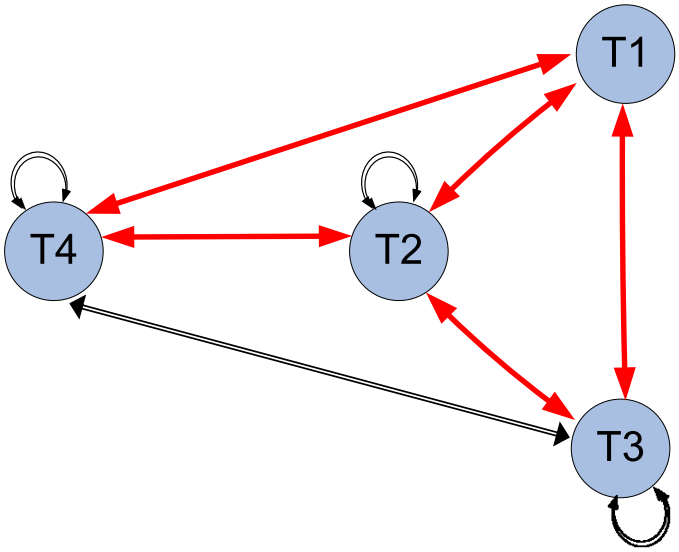

Figure 29: SDG for MoreChoice, ALL-Promotion and ALL-Materialize.

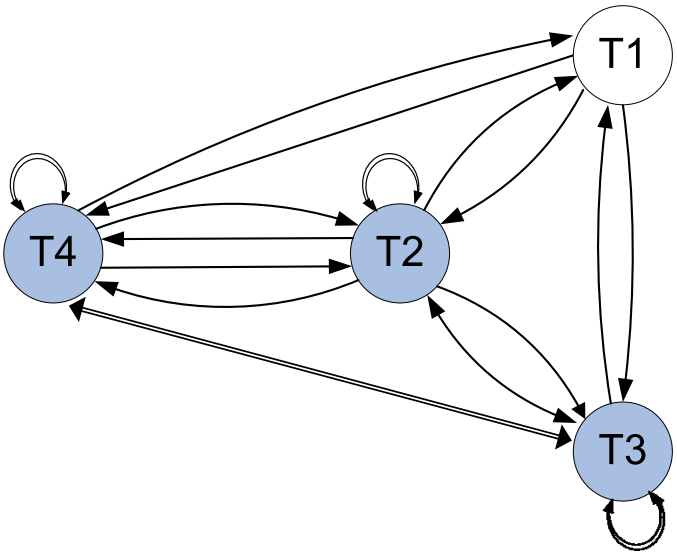

Figure 30: SDG for MoreChoice, ELM-ALL.

promote, and use ELM with choice2 edges.

- ALL: Removing ALL vulnerable edges $\{ T_1 \dashrightarrow T_2, T_1 \dashrightarrow T_3, T_1 \dashrightarrow T_4, T_2 \dashrightarrow T_4, T_4 \dashrightarrow T_2$, and $T_2 \dashrightarrow T_3\}$. Figure 29 and 30 show the SDG for MoreChoice benchmark after we materialize, promote, and use ELM with choice3 edges.

4.5 Summary

This chapter describes the experimental framework: the software and the hardware, different factors that may affect our conclusions, and finally we describe the two benchmarks that we use in the next chapter.

In the next chapter, we use the experimental framework to explore in detail the performance impact of the various techniques that guarantee serializable execution.

Chapter 5

Evaluation

Th is chapter evaluates the various techniques described previously, including our new ELM proposal as well as Materialize and Promotion described in Chapter2. As also discussed in Section 2.6.6 there is another approach called pivot_2PL which was shown in [24, 9] to be worse than the existing techniques, therefore we do not discuss it anymore.

We compare these techniques using one open source platform-PostgreSQL (so we have access to the source code) and one commercial paltform-Oracle, 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.

Roadmap: This chapter is structured as follows: In Section 5.1 we briefly review the SmallBank benchmark and its options for making SI serializable. Section 5.2 we present performance comparison between ELM and other techniques that ensure serializable execution under SI on PostgreSQL. In Section 5.3 we use a different platform: Oracle. Finally, we evaluate with the MoreChoices benchmark in 5.4. We conclude the previous sections in 5.5. Summary of the chapter in 5.6

5.1 Options to ensure serializable execution with SmallBank

In this chapter we use Table 3 from Chapter 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).

5.2 Serializability of SI on PostgreSQL, for SmallBank

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

5.2.1 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 31 shows the throughput in transaction per second (TPS) as a function of MPL for

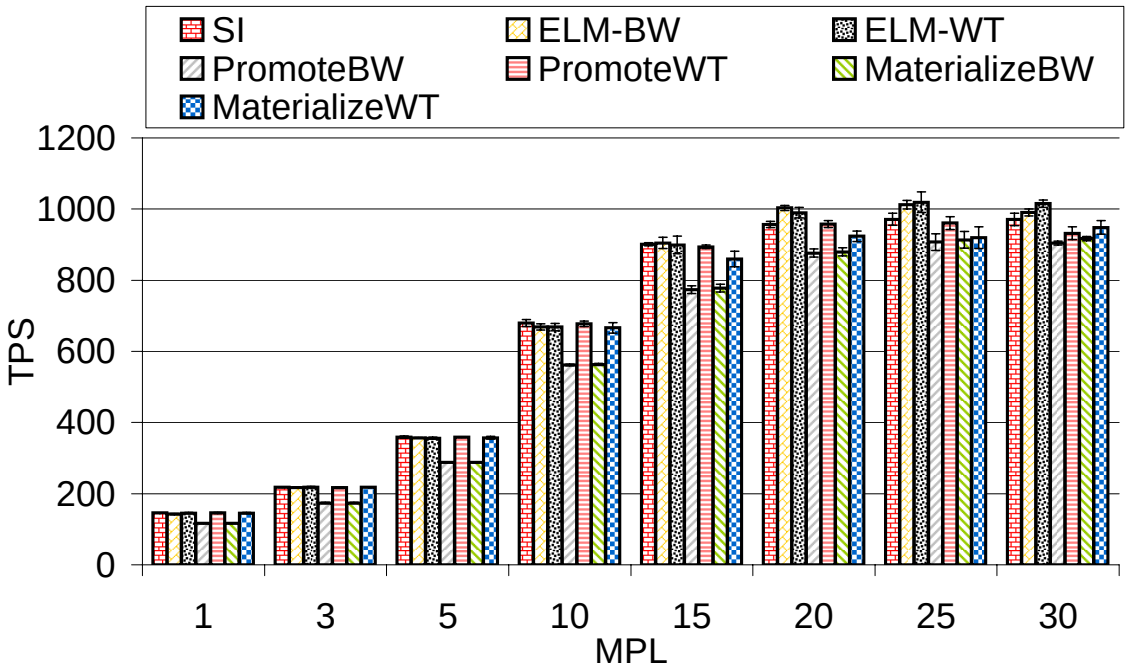

Figure 31: Throughput over MPL, Low Contention, High Update, SmallBank, PostgreSQL.

the more sophisticated among the different options available in SmallBank for guaranteeing serializable execution. [1] We also include the figures (labelled SI) for the unmodified application under SI. From the same data, we derive Figure 32 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!).

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_upd (Identity update), and also for MaterializeBW,

[1] These options all required the DBA to identify dangerous structures, and chose a minimal set of edges to make non-vulnerable.

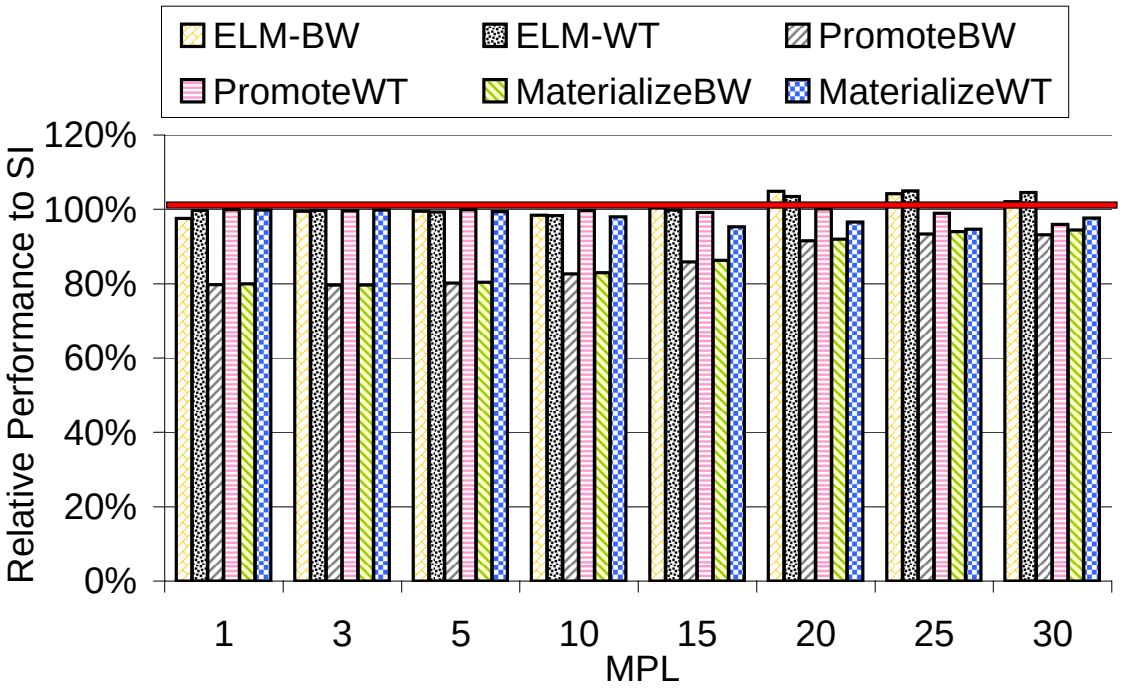

Figure 32: Throughput relative to SI, Low Contention, High Update, SmallBank, PostgreSQL.

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

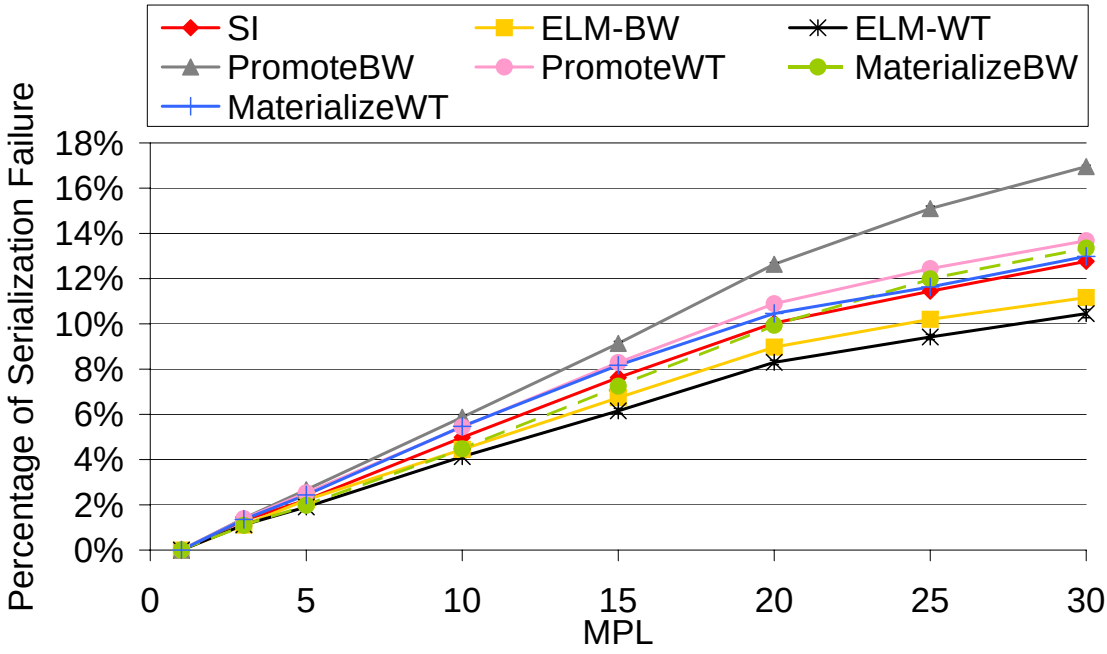

Figure 33: Serialization Failure, Low Contention, high Update, SmallBank, PostgreSQL.

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 32). 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 MaterilizeWT, 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 33 shows the percentage of serialization failure of 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, [2] whereas in this same scenario, one instance of the unmodified program will abort due to FCW.

Figure 34 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 this should not be changed neither using materialisation or promotion. However, introducing locks into this transaction using the ELM technique is quite acceptable.

[2] 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.

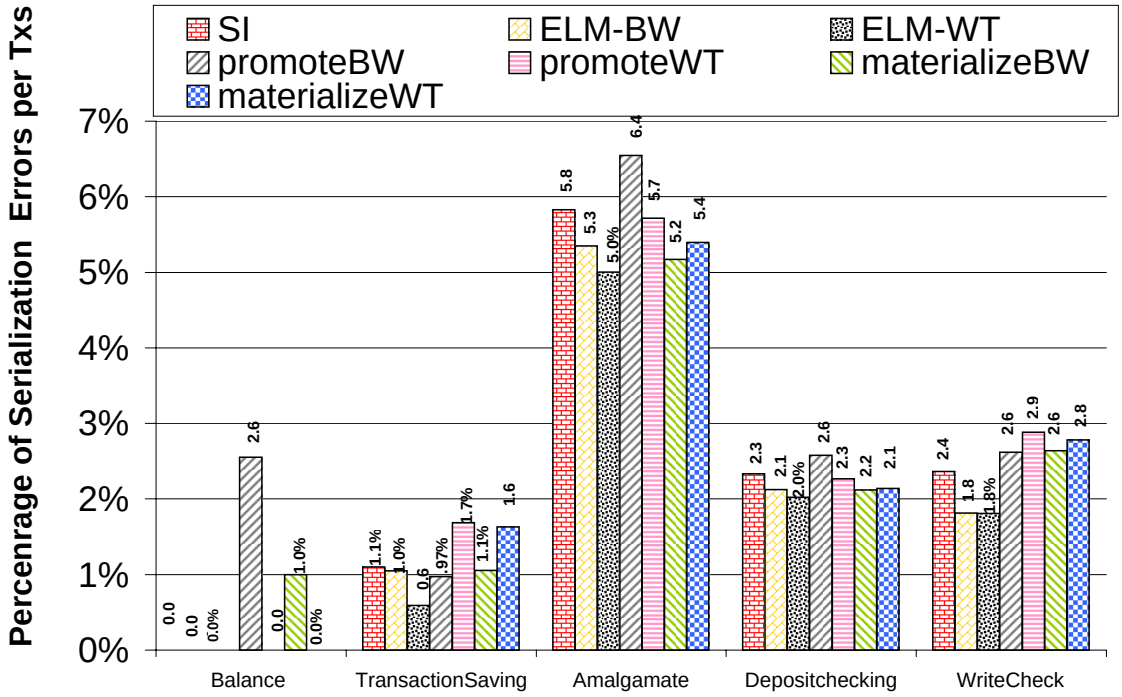

Figure 34: Serialization Failure Ratio per Transaction Type, Low Contention, SmallBank, PostgreSQL.

Figure 35 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.

Figure 36 shows the message sequence diagram. The mean response time consists of three components which are:

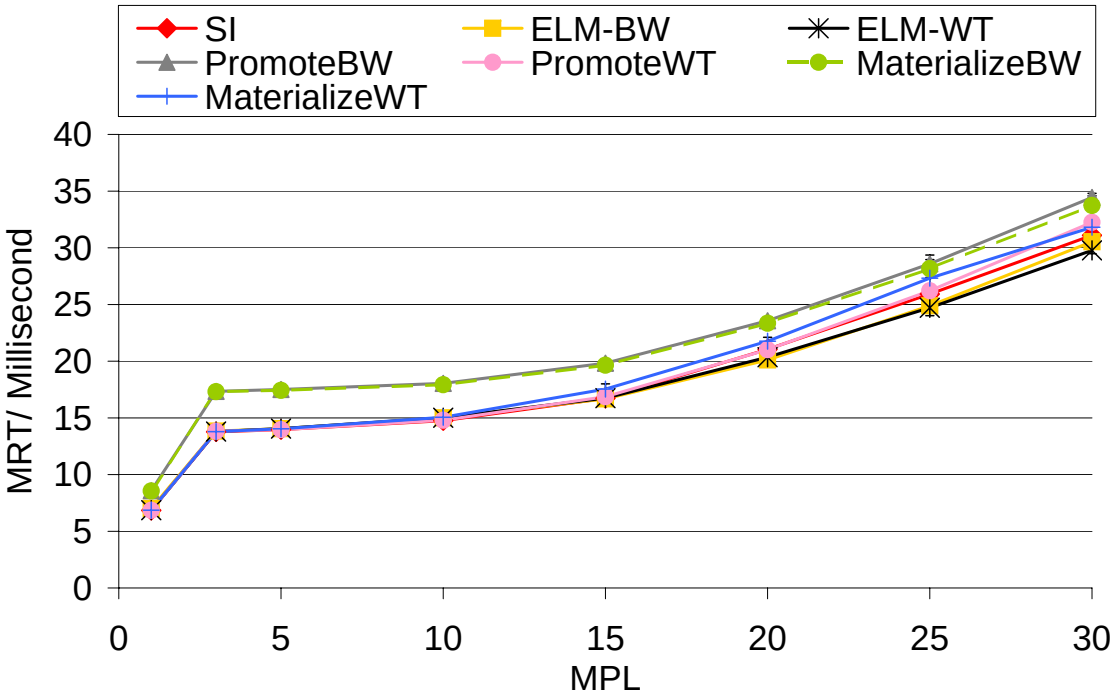

Figure 35: Mean Response Time, Low Contention, SmallBank, PostgreSQL.

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 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 37 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

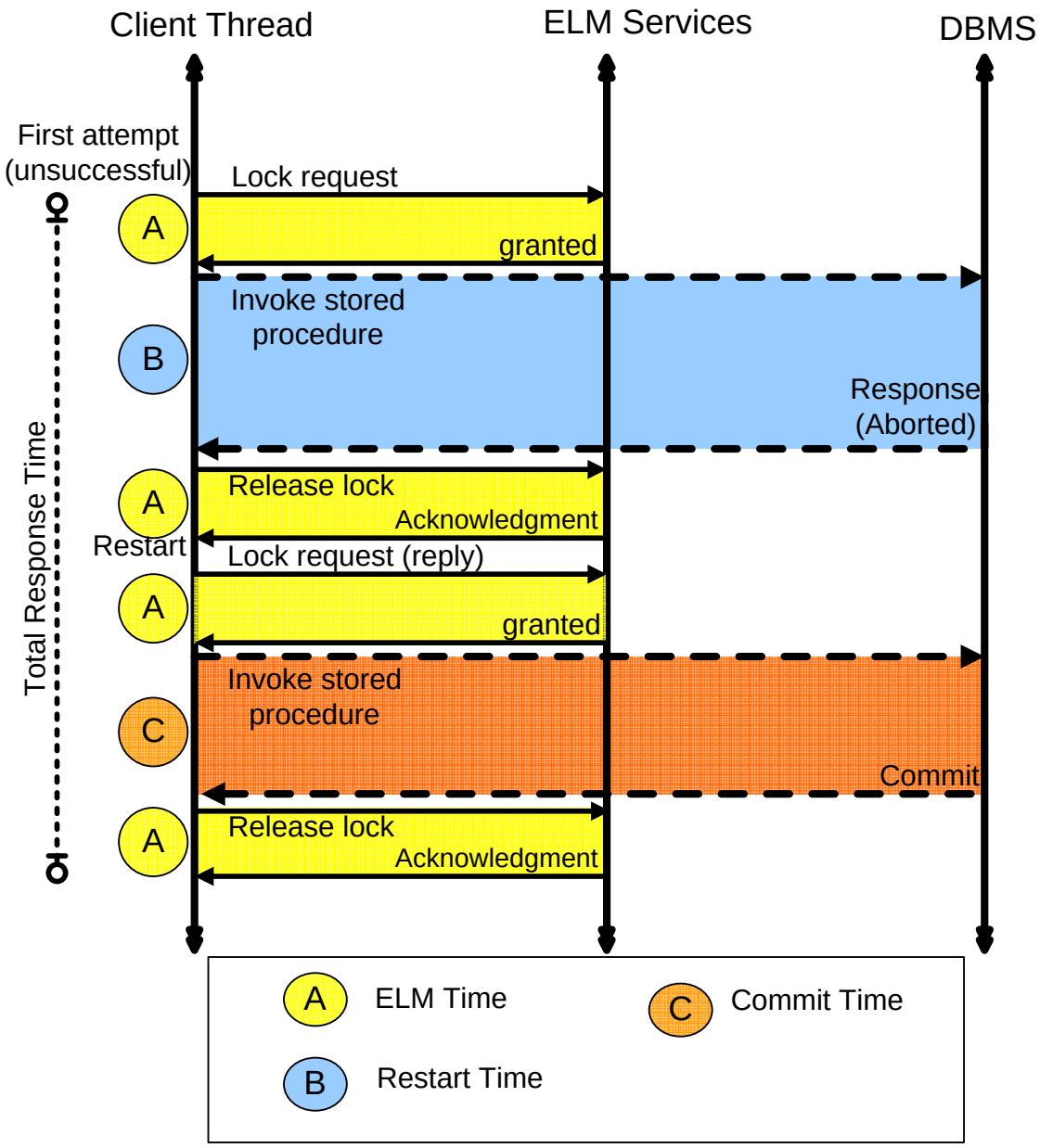

Figure 36: Message sequence diagram.

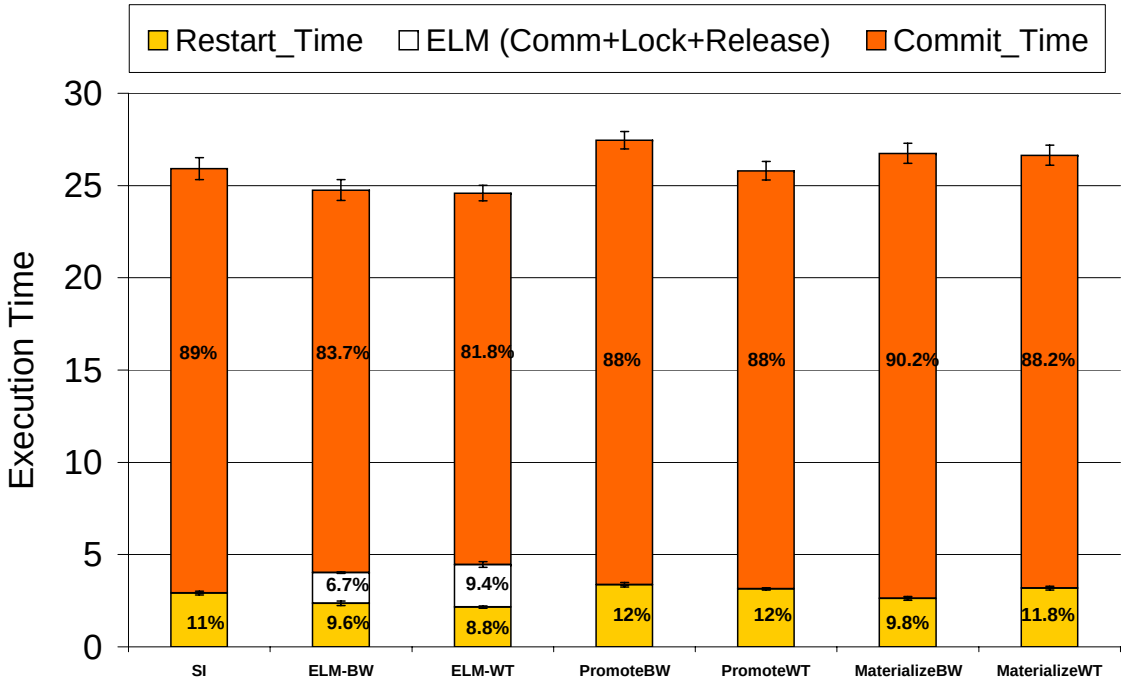

Figure 37: Average Execution time, Low Contention, SmallBank, PostgreSQL.

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 commits 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, which reduces the probability of conflict 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.

Finally, we consider the straight-forward strategies that remove the vulnerability from

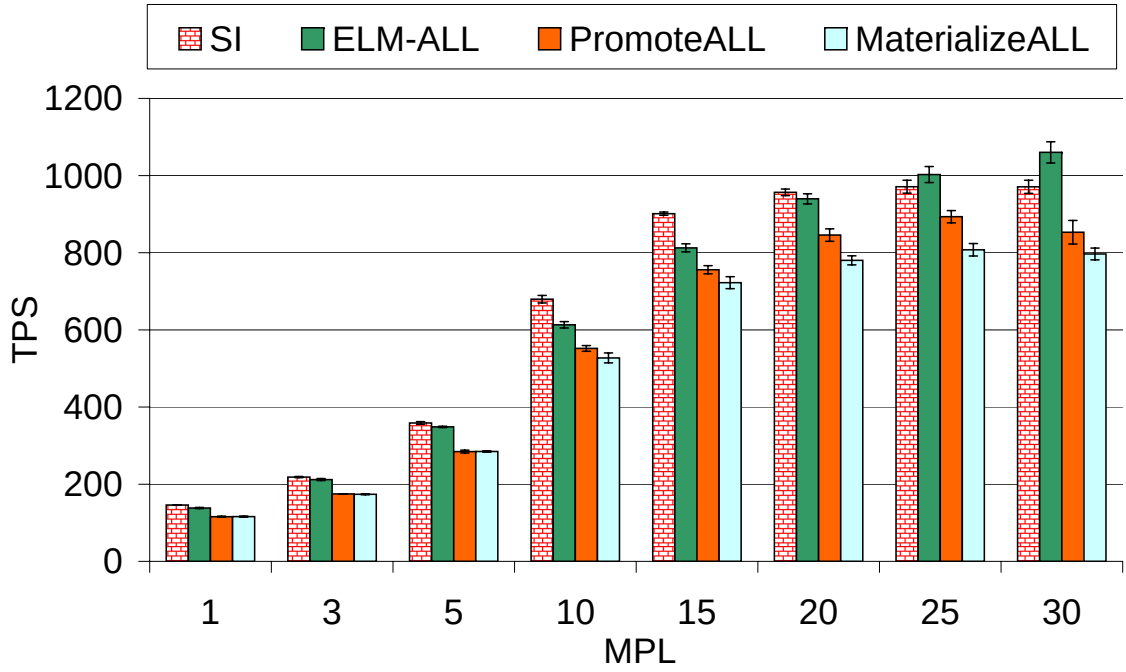

Figure 38: Costs for SI-serializability when eliminating ALL vulnerable edges, Low Contention, SmallBank, PostgreSQL..

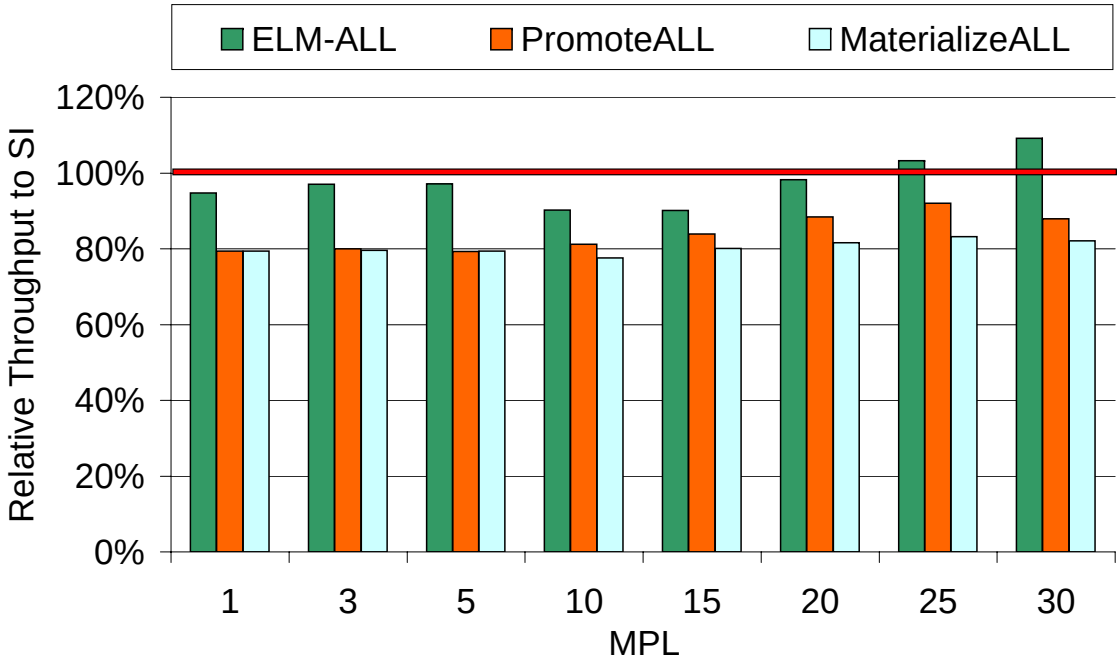

Figure 39: Relative Throughput to SI (ALL vulnerable edges), Low Contention, SmallBank, PostgreSQL..

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 38 shows the resulting throughput in Transactions Per Second (TPS) as a function of MPL. Figure 39 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 40 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, [3] 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 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.

[3] 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.

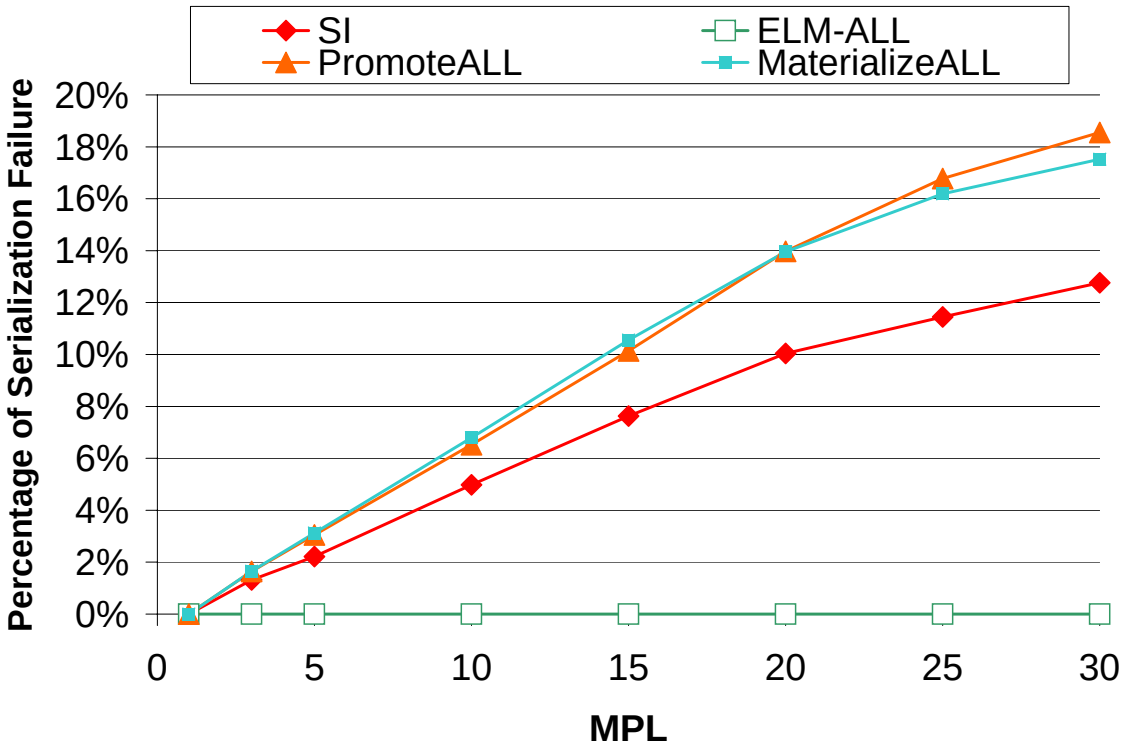

Figure 40: Percentage of Serialization Failure (ALL edges), Low Contention, SmallBank, PostgreSQL..

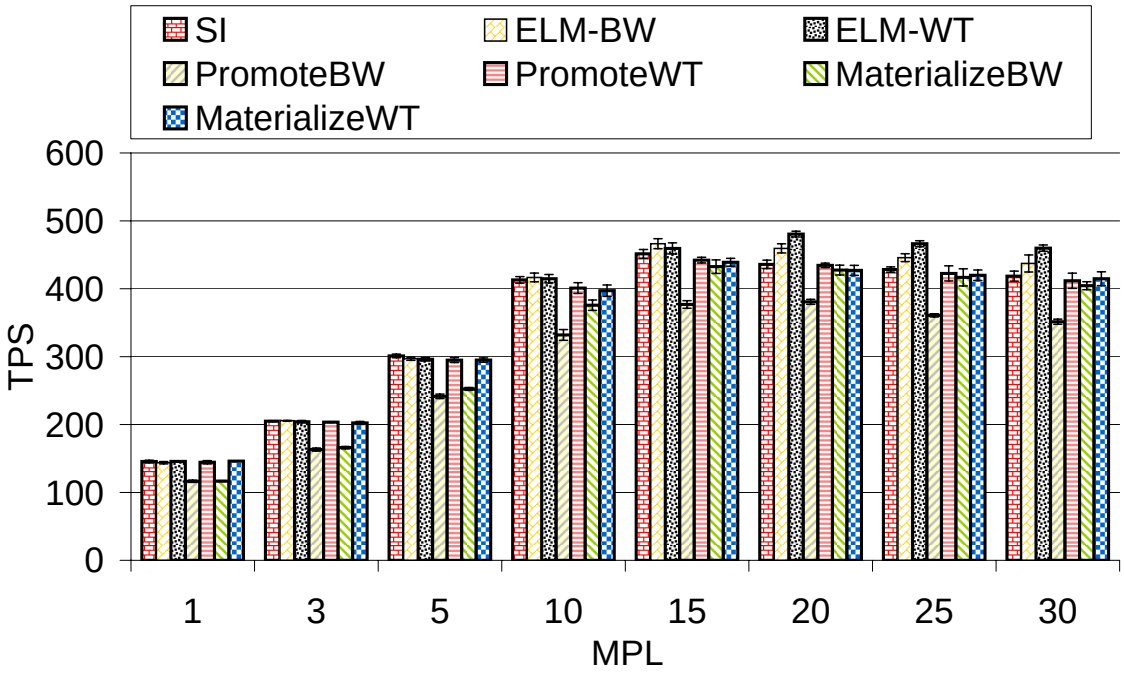

Figure 41: Throughput over MPL, High Contention, SmallBank, PostgreSQL.

5.2.2 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).

Figure 41 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 42 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.

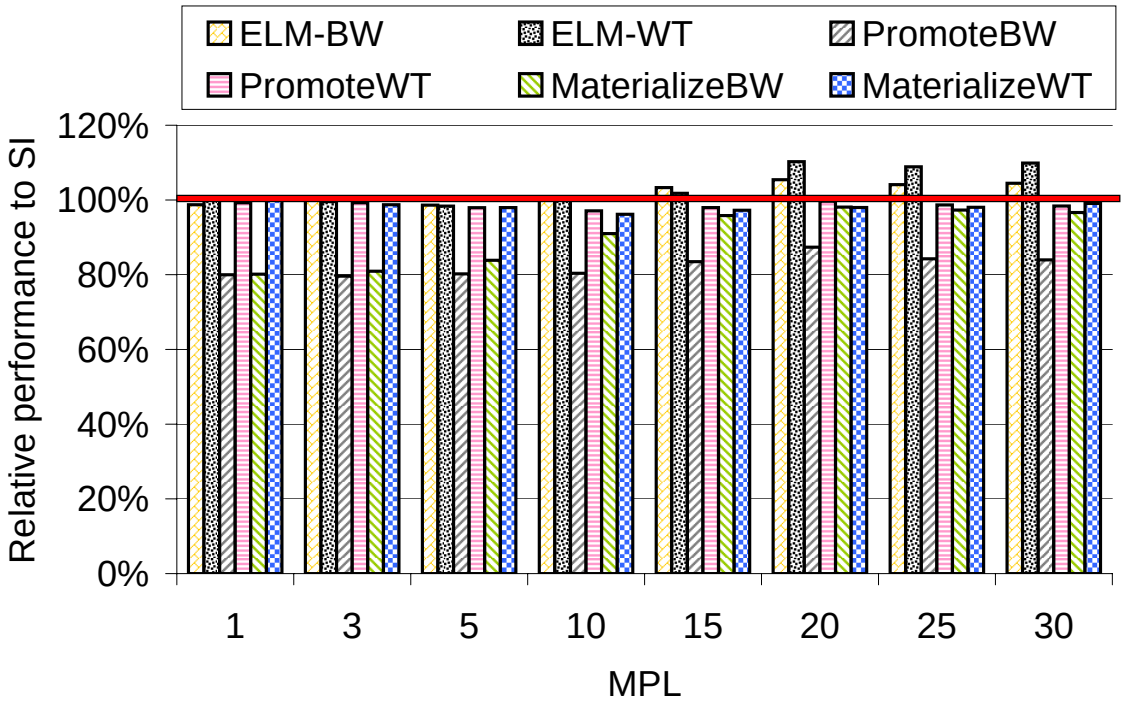

Figure 42: Throughput relative to SI, High Contention, SmallBank, PostgreSQL.

- 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 that PromoteBW due to the extra abort rate and restarts with PromoteBW.

Figure 43 shows the percentage of serialization failure of different options. Under this extreme condition, the percentage of serialization failure has been increased due to the

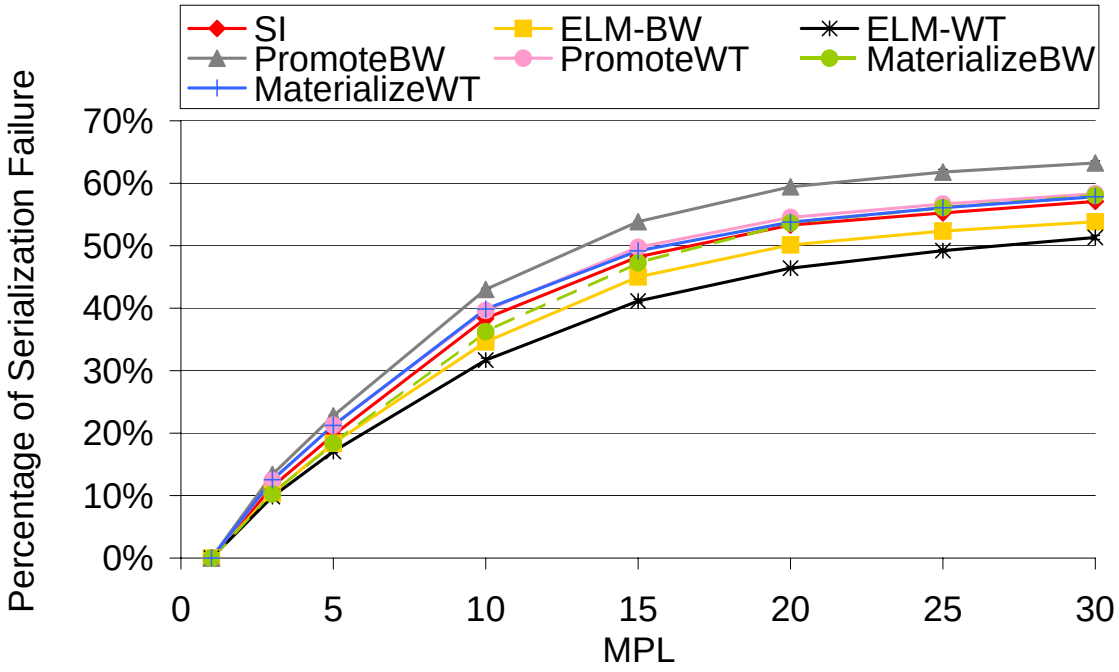

Figure 43: Serialization Failure, High Contention, SmallBank, PostgreSQL.

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 44 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 45 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

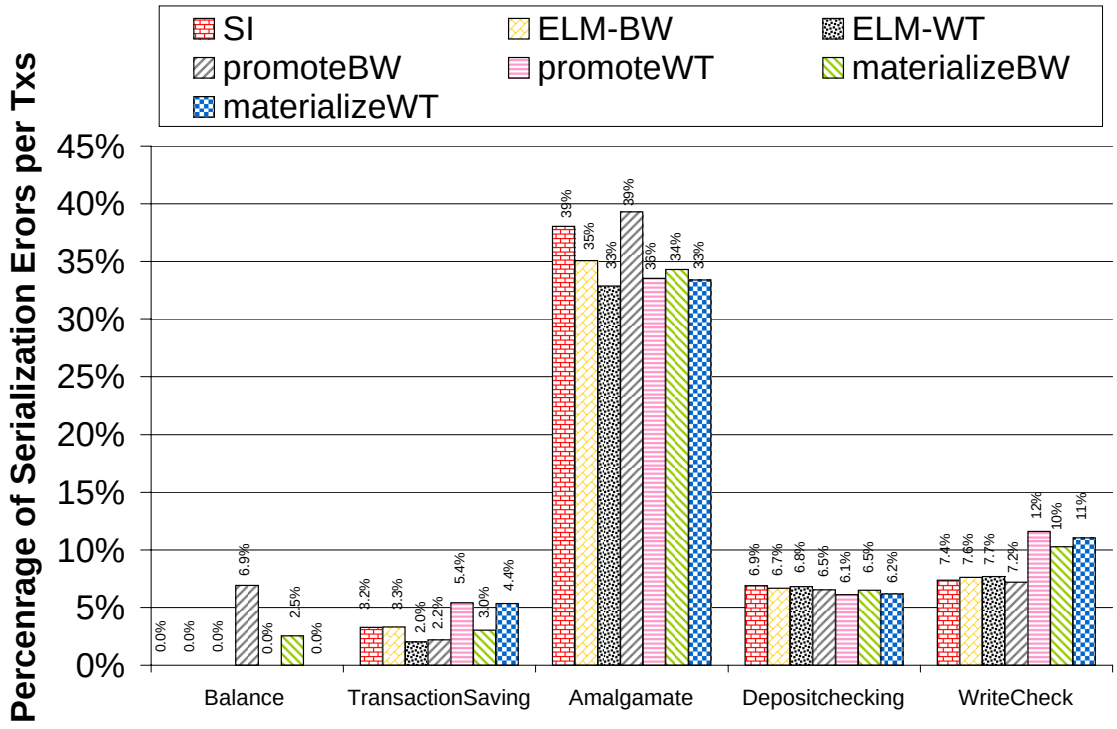

Figure 44: Serialization Failure Ratio per Transaction Type, High Contention, SmallBank, PostgreSQL.

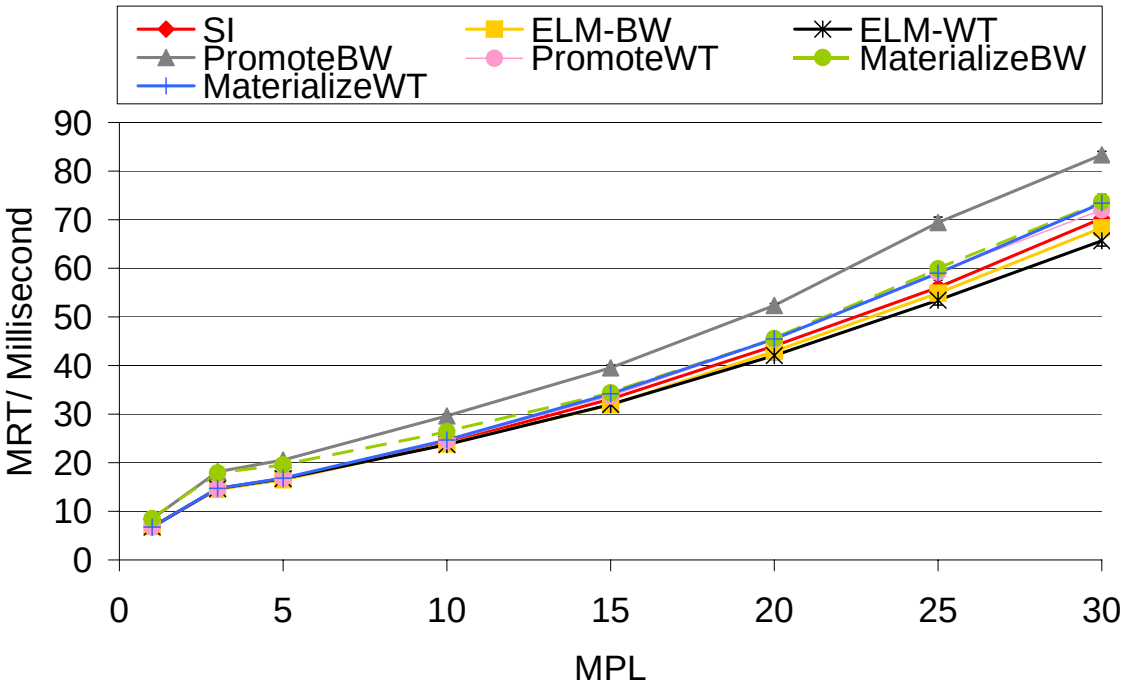

Figure 45: Mean Response Time, High Contention, SmallBank, PostgreSQL.

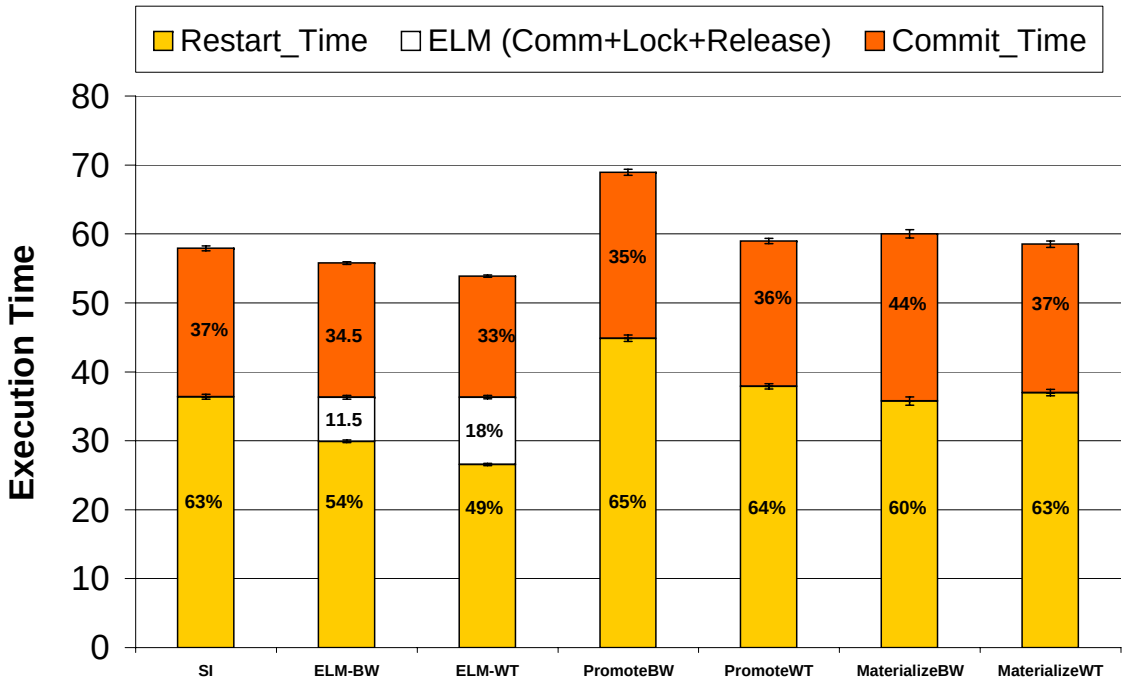

Figure 46: Average Execution time, High Contention, SmallBank, PostgreSQL.

(Bal transaction), and due to the high percentage of restart.

Figure 46 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 (these effects are ranked opposite to the low data contention case).

Figure 47 shows the relative performance as compared to the throughput with SI (shown as thick horizontal line) for each option that ensures serializable executions by removing ALL vulnerable edges. 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.

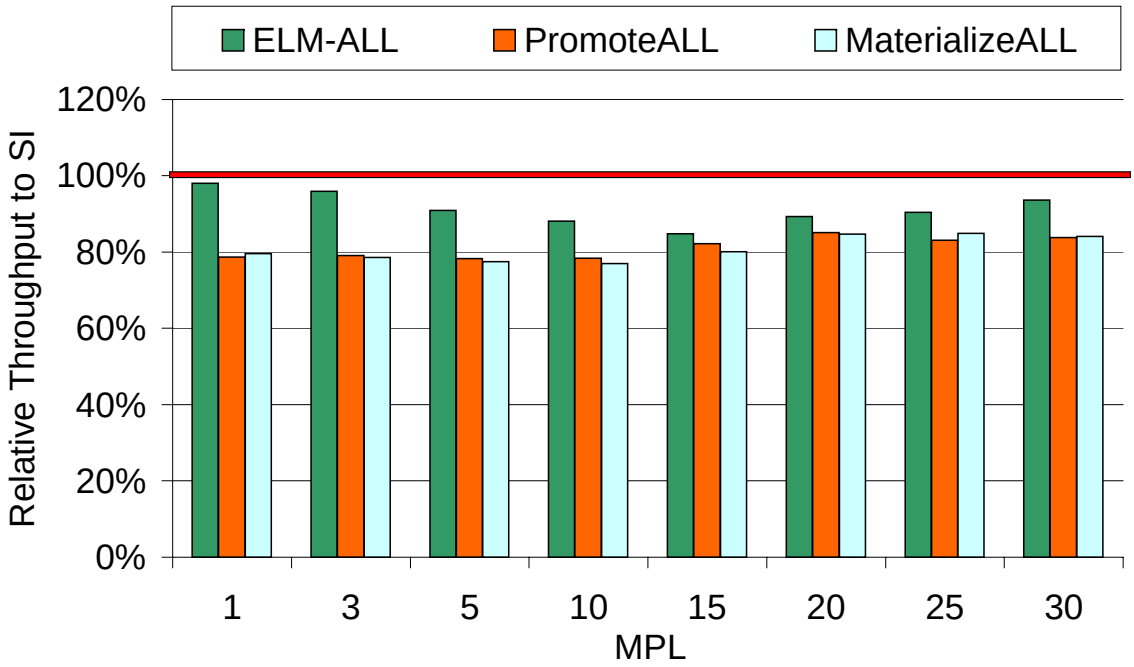

Figure 47: Relative Throughput to SI (ALL vulnerable edges), High Contention, SmallBank, PostgreSQL.

The qualitative conclusions are the same as the low contention case: any techniques that affect 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 SI).

5.2.3 Low Update Rate

Many real world applications have more frequent read-only transactions than update [19]. 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 serailizable execution with SI behaviors.

Figure 48 shows the throughput in transaction per second (TPS) as a function of MPL

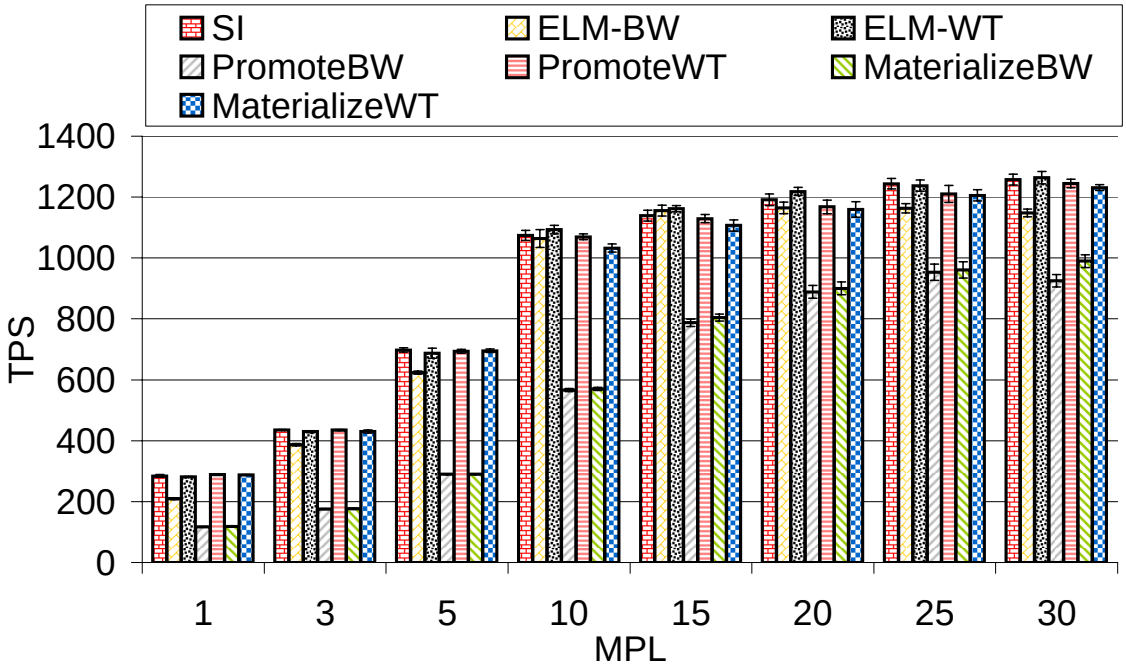

Figure 48: Throughput with 60% read-only, Low Contention, SmallBank, PostgreSQL.

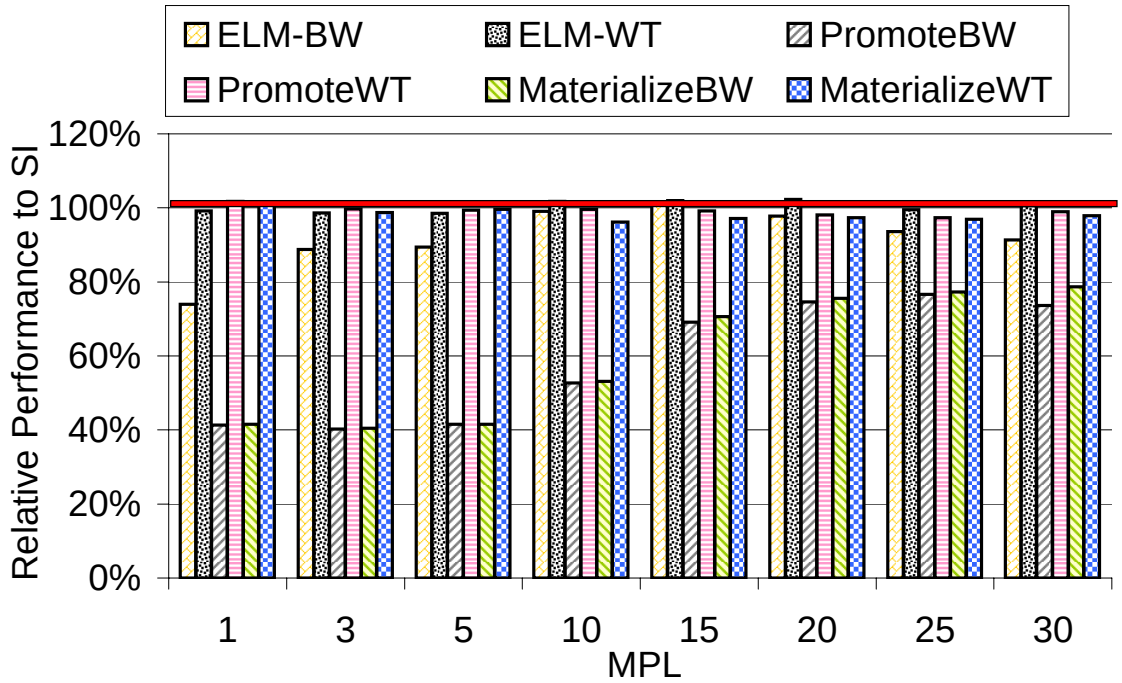

Figure 49: Relative Throughput with 60% read-only, Low Contention, SmallBank, PostgreSQL.

for the different options available in SmallBank for guaranteeing serializable execution. Figure 49 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 50) 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 du to the small percentage of WC and TS update transactions (10% for each).

The high level conclusions from Figure 51 is similar to Figure 33. Promoting BW edge comes with a high cost especially with 60% of Balance transaction, Balance transaction increases the probability of extra abort rate as we discussed before. 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 51 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 33, Materialize BW edge is really close to unmodified SI, PromoteWT and MaterializeWT, where percentages of transactions are fixed.

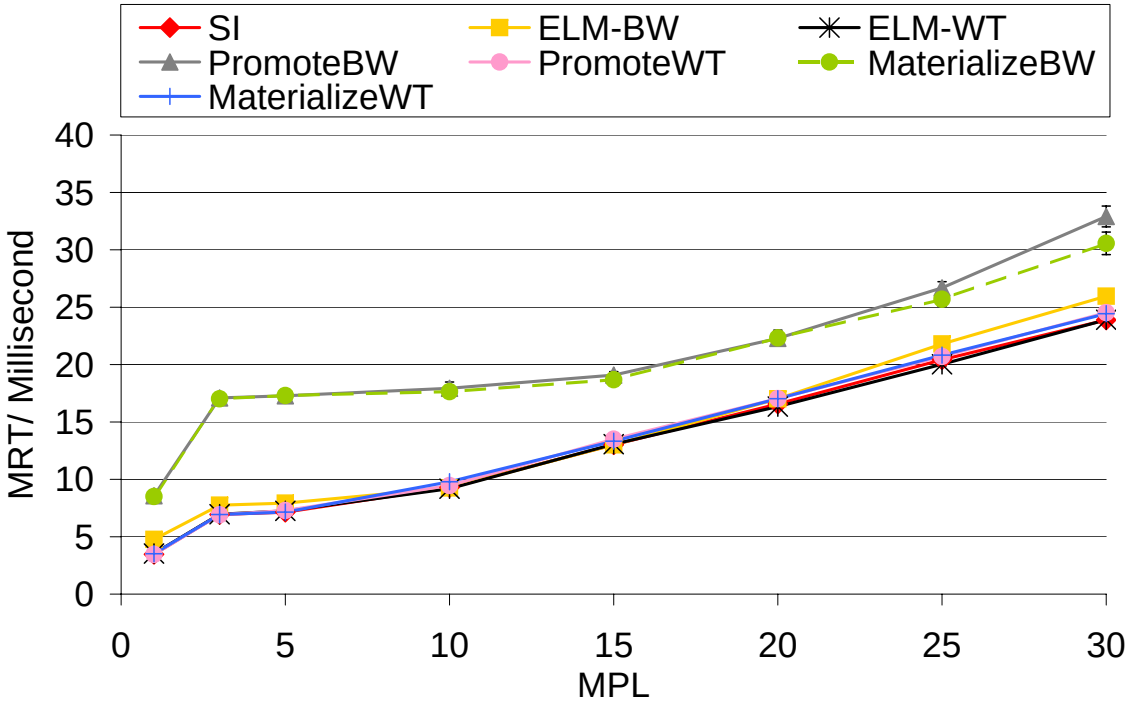

Figure 50: Mean Response Time with 60% read-only, Low Contention, SmallBank, PostgreSQL.

We also tested this mixture under a high contention scenario where the hotspot was 10. Figure 52 shows the throughput in transaction per second (TPS) as a function of MPL, and Figure 53 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.

5.2.4 Comparison with Low Isolation Level

So far, we compared the throughput of the different options with the standard unmodified SI, but it also interesting to investigate how much we may lose compared to a common,

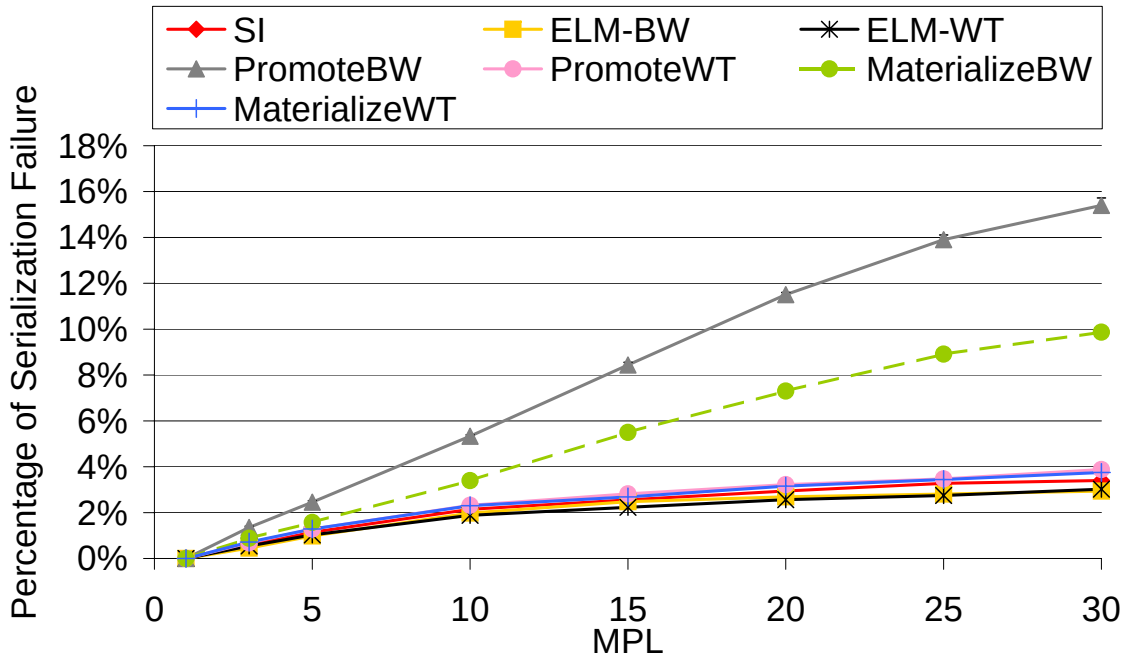

Figure 51: Serialization Failure with 60% read-only, Low Contention, SmallBank, PostgreSQL.

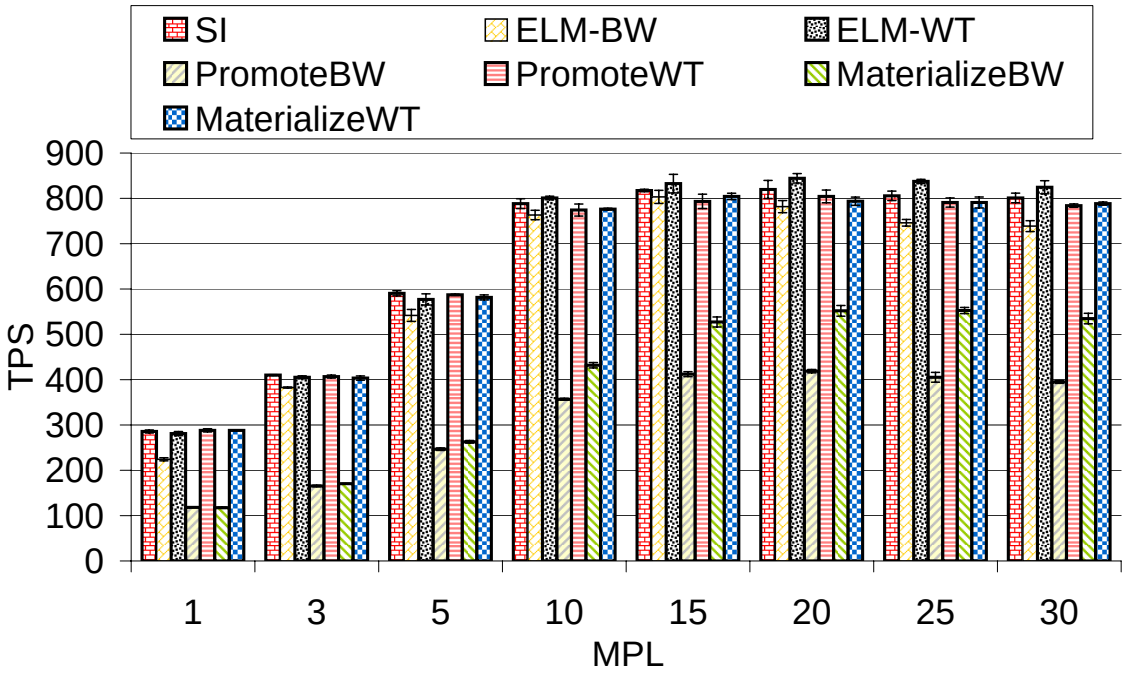

Figure 52: Throughput with 60% read-only-, High Contention, SmallBank, PostgreSQL.

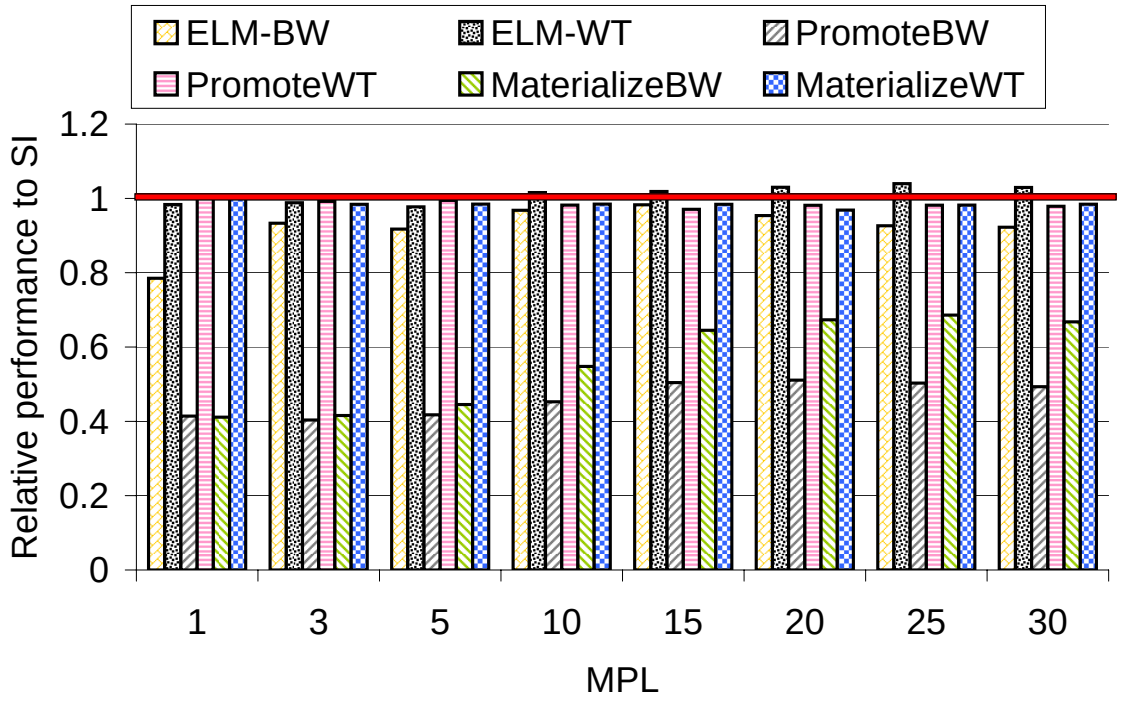

Figure 53: Relative Throughput with 60% read-only, High Contention, SmallBank, PostgreSQL.

less restricted isolation level such as read committed.

Indeed in most SQL-based platforms, the Read Committed (abbreviated as RC) isolation level is the default, used when the application developer has not explicitly set an isolation level. In a platform that offers SI (such as PostgrSQL), when a transaction runs with RC and updates a data item, it sets locks just as in 2PL, but read operation does not require any locks (concurrent transactions may create new versions, but the reader sees the data from the version at the time the read query started). In RC, Lost Update can not occur, because any transaction that modifies the data keeps an exclusive lock on the item; however the Inconsistent Read anomaly is possible. This is why RC is seen as offering better performance, in return for accepting the risk of data corruption from anomalies such as an inconsistent read.

In this section we show that sometimes, guaranteed correctness can be obtained along with better throughput than RC, by use of the multiversion Snapshot Isolation mechanism

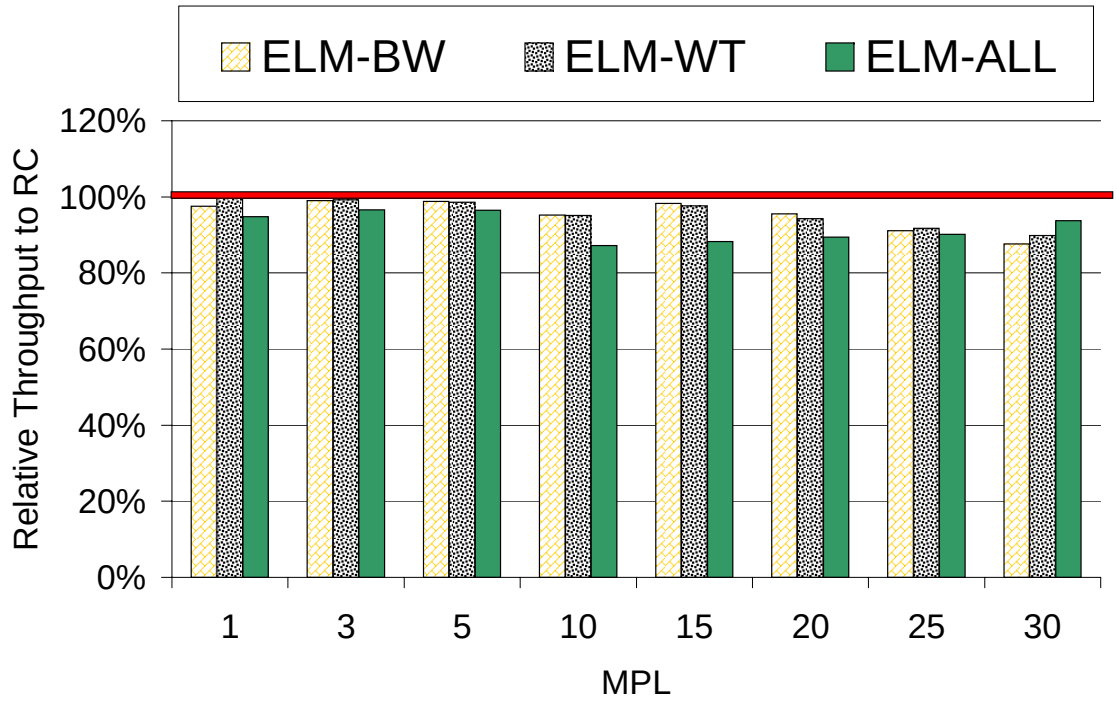

Figure 54: Relative Throughput to Read Committed, Low Contention, SmallBank, PostgreSQL.

along with ELM technique. Figure 54 shows the relative throughput to RC. [4] We perceive that throughput of ELM-BW and ELM-WT starts indistinguishable from that RC up to MPL=5, and then decreases up to 10% of that RC with MPL=30.

We clearly see that by running on SI and modifying the SDG using ELM, we can make all execution serializable for very low cost (between 0%-10% less than RC even with ALL option) compared to using RC which can suffer from many phenomena of data corruption. Figure 55 shows the mean response time for different options. We clearly see that RC comes with lowest mean response time followed by the ELM technique. SI mean response time is higher than RC mean response time due the extra time it takes to restart the failed transactions.

We also re-evaluate the experiments where we reduced the size of hotspot region from

[4] We ran new experiments to measure performance of RC, but otherwise these graphs use the same data from other measurement.

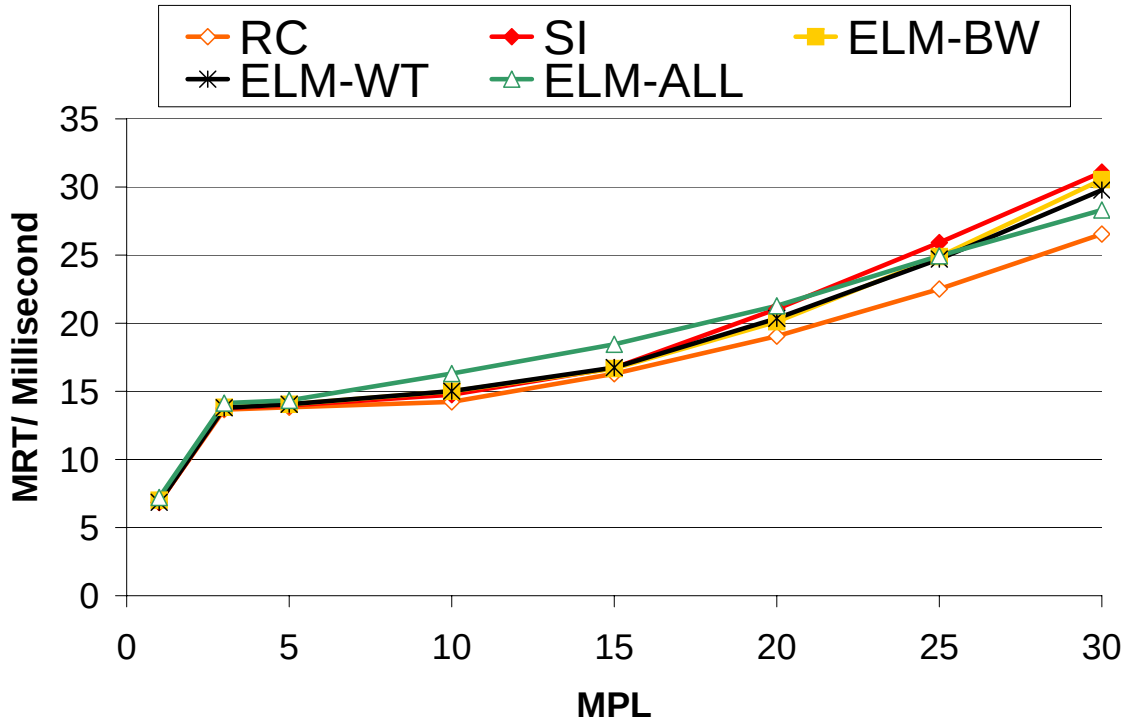

Figure 55: Mean Response Time, Low Contention, SmallBank, PostgreSQL.

100 to 10 customers. Such high data contention reduces the overall peak performance for each option. Figure 56 shows the relative throughput to RC. We perceive that throughput for ELM-WT and ELM-WT edge starts indistinguishable from that RC and keep rising till it reaches about 137% of that for RC with MPL=30.

If we have two transactions T_1 and T_2 trying to modify the same data item, and one of them commits and releases its locks, a RC transaction proceeds with its intended update. Therefore, transactions using RC need to wait in a queue until the predecessor transactions commit (on the same data item). On the other hand, an SI-using transaction does not wait, but it aborts, because the other transaction has committed a change that was made since the serializable transaction began. Re-starting the transactions reduces the contention for the high conflict scenario; this explains the reason behind higher performance with options that makes SI serializable.

This illustrates that using a SDG analysis of the transaction mix and then run them under

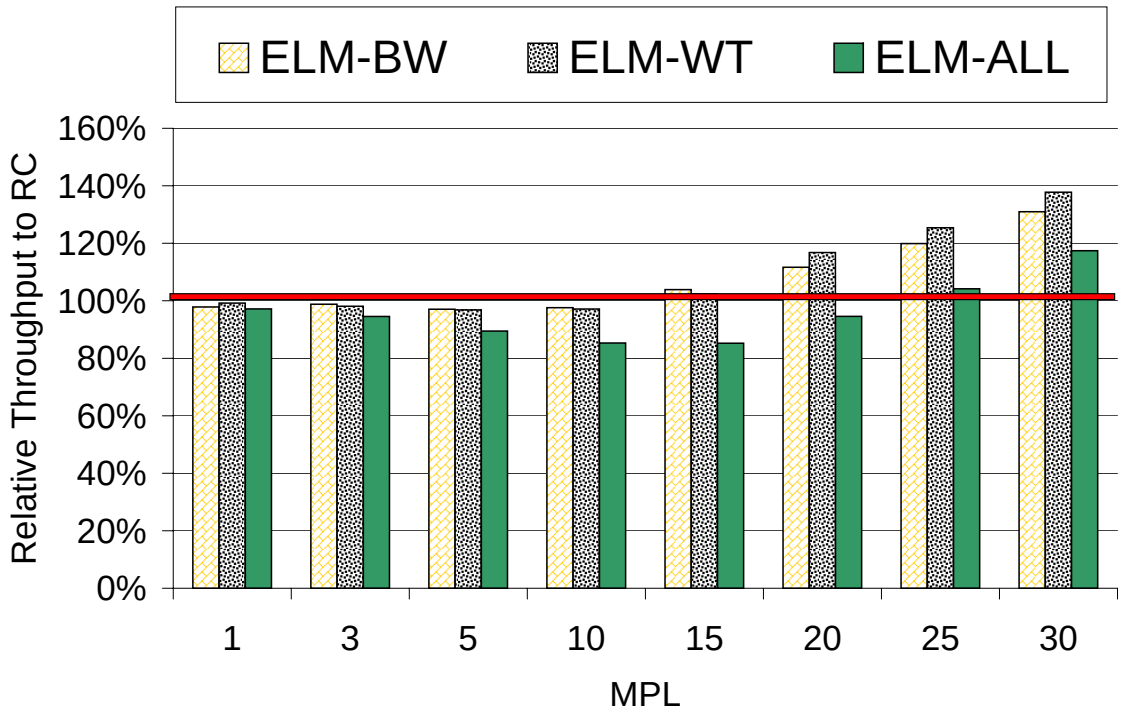

Figure 56: Relative Throughput to Read Committed, High Contention, SmallBank, PostgreSQL.

SI using our ELM technique can give both serializable executions and good performance; and is thus a much preferable approach rather than running with RC.

5.3 Serializability of SI on Oracle

So far, we have focused on PostgreSQL because we can seek understanding of our observations from knowledge of the detailed implementation. For comparison, we also ran our experiments on one of the commercial platforms that offers Snapshot Isolation concurrency control, we use Oracle version 10g for this purpose. We investigate the behaviors of the different options that ensure serializable executions with SI on this platform. We run the same experiments under low and high data contention. Note that there is no sensible comparison between the absolute numbers here, and those in previous sections.

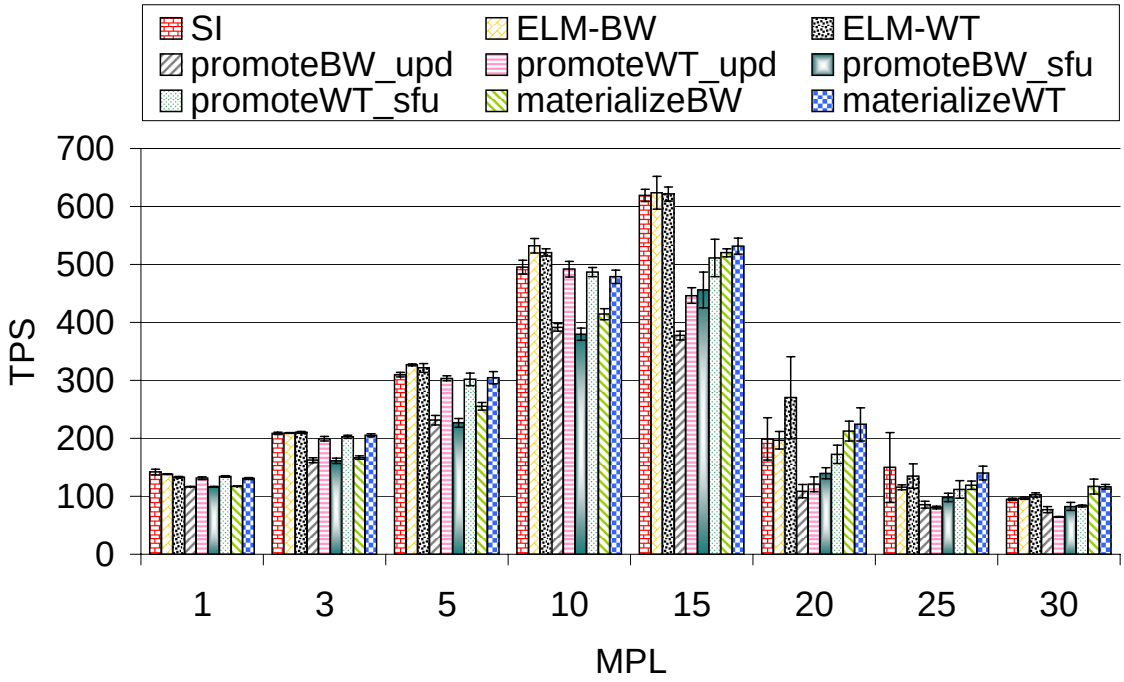

Figure 57: Throughput over MPL, Low Contention, SmallBank, Oracle.

5.3.1 Low Contention

We will explore in some detail the case where hotspot has 100 rows. We consider a new option called *Promote_sfu*, an option works only with Oracle. [5] Figure 57 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 58 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

[5] Some SQL dialects also allow the statement Select ... For Update (SFU). On Oracle SFU is treated for concurrency control like an Update, and so promotion can be done by changing the read into SFU; however 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.

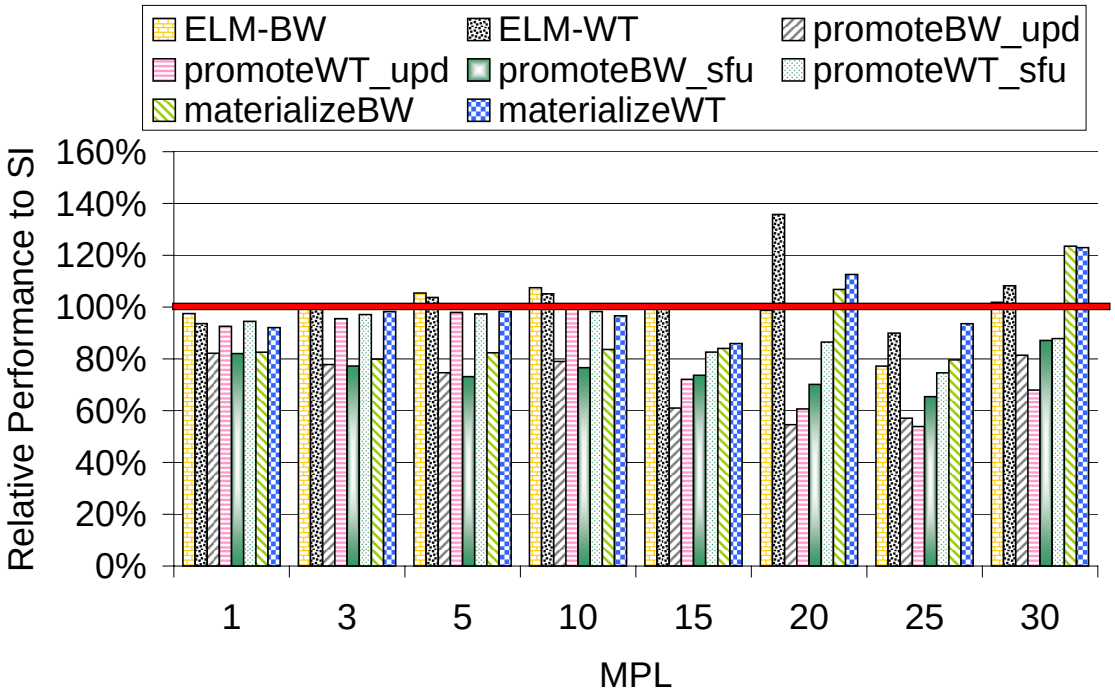

Figure 58: Throughput relative to SI, Low Contention, SmallBank, Oracle.

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.

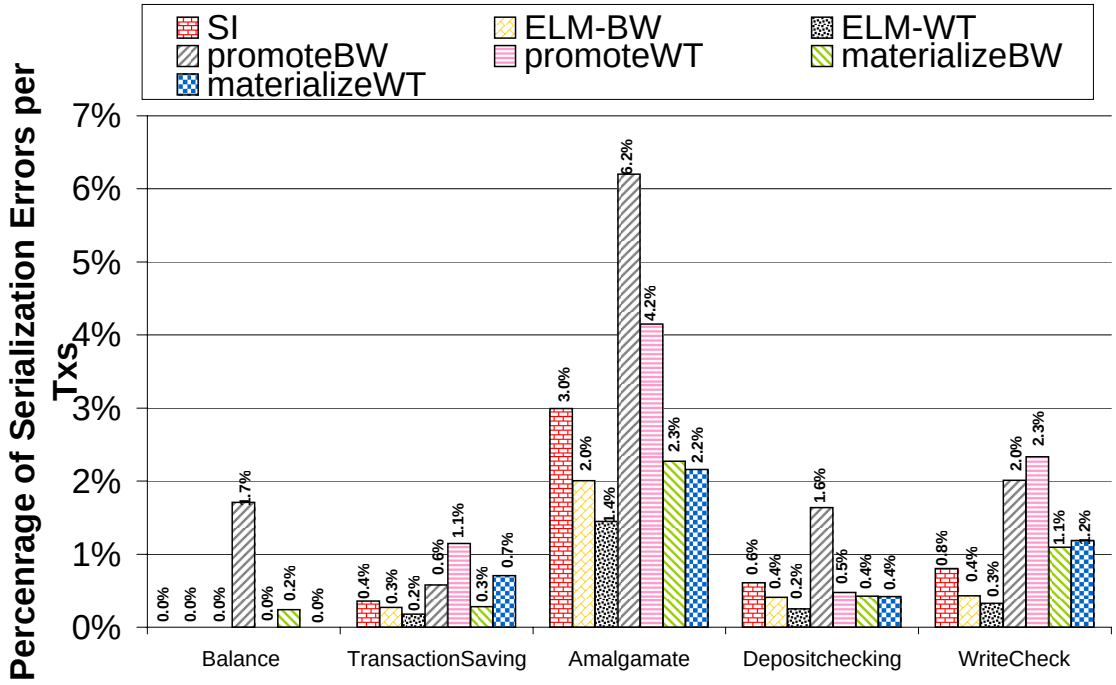

Figure 59: FCW Error per Transaction Type, Low Contention, SmallBank, Oracle.

We see a very different overall shape of that for PosgreSQL: the throughput for different options rise to a peak but then quickly drops away as MPL increases.

Notice that Promotion leads to very poor performance under any edge choice, [6] 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 59 shows the percentage of "Can not Serialize" errors per transaction type with MPL=15 (Peak throughput). [7] We see that every 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

[6] 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 cast 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.

[7] Oracle called the FCW errors "Can not serialize", where PostgrSQL call it "Serialization Failure".

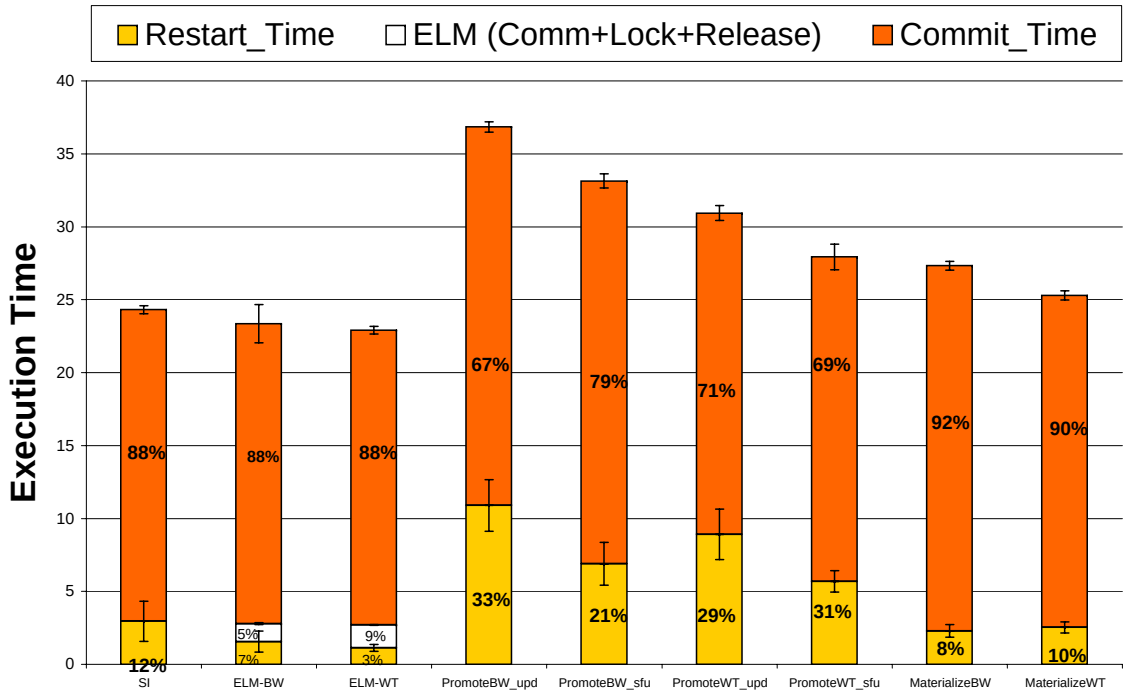

Figure 60: Average Execution time, Low Contention, SmallBank, Oracle.

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 60 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 only by reduction in restart time.

Finally, we consider the straight-forward strategies that remove the vulnerability from every vulnerable edge. Figure 61 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

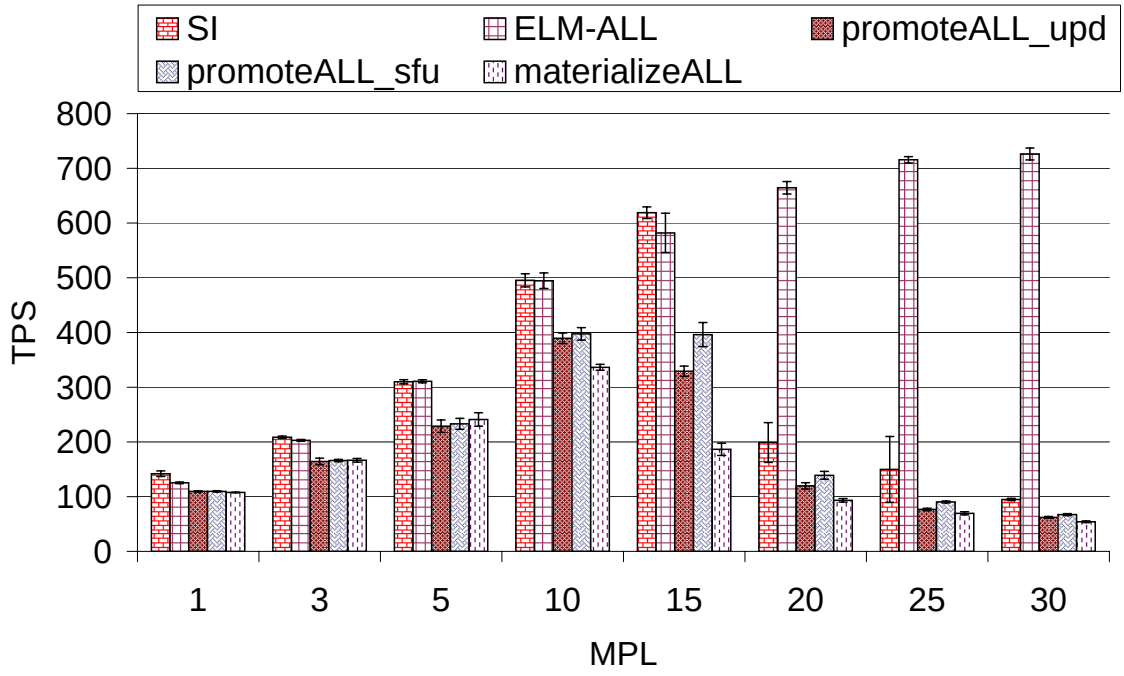

Figure 61: FCW Error per Transaction Type, Low Contention, SmallBank, Oracle.

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.

5.3.2 High Contention

Finally, we reduce the hotspot to 10 to create more contention in Oracle. The high contention situation in Figure 62 does not change the overall story, but it confirms the conclusions from PostgreSQL and from the Oracle at low contention. Figure 62 shows that materialize performs generally better than promotion but it still depends dominantly on the MPL. Again, ELM technique performs very close to unmodified SI and even better

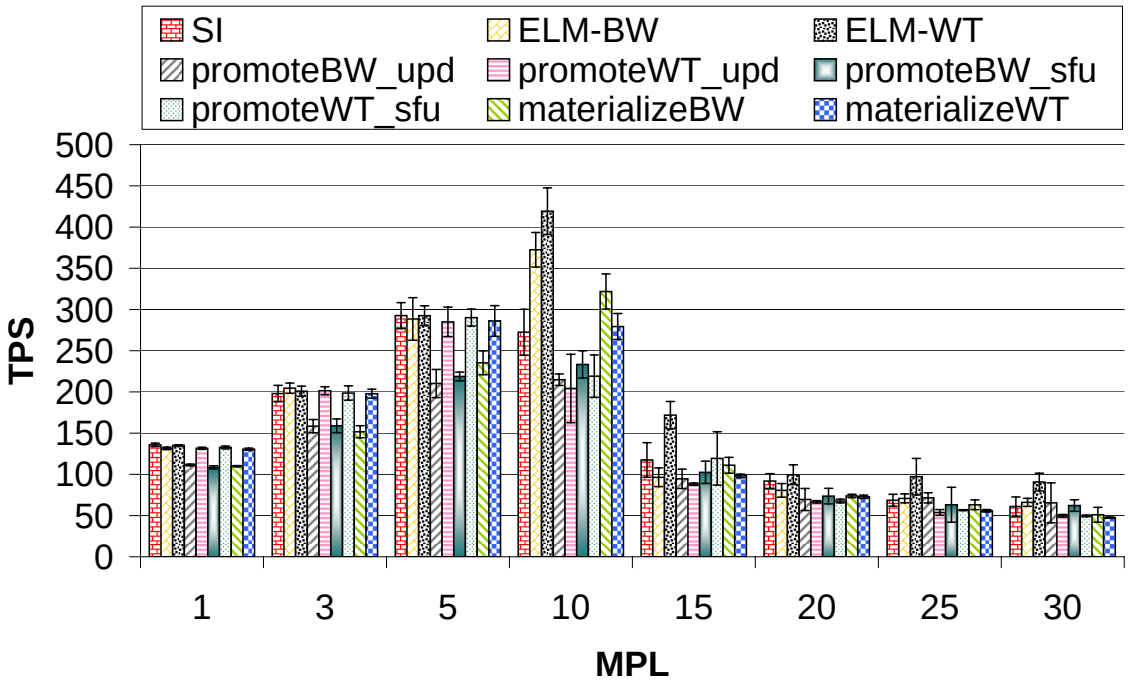

Figure 62: Throughput over MPL, High Contention, SmallBank, Oracle.

in some situations.

Figure 63 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 at 61 TPS with MPL=30.

The overall story for both high and low data contention in Oracle 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.

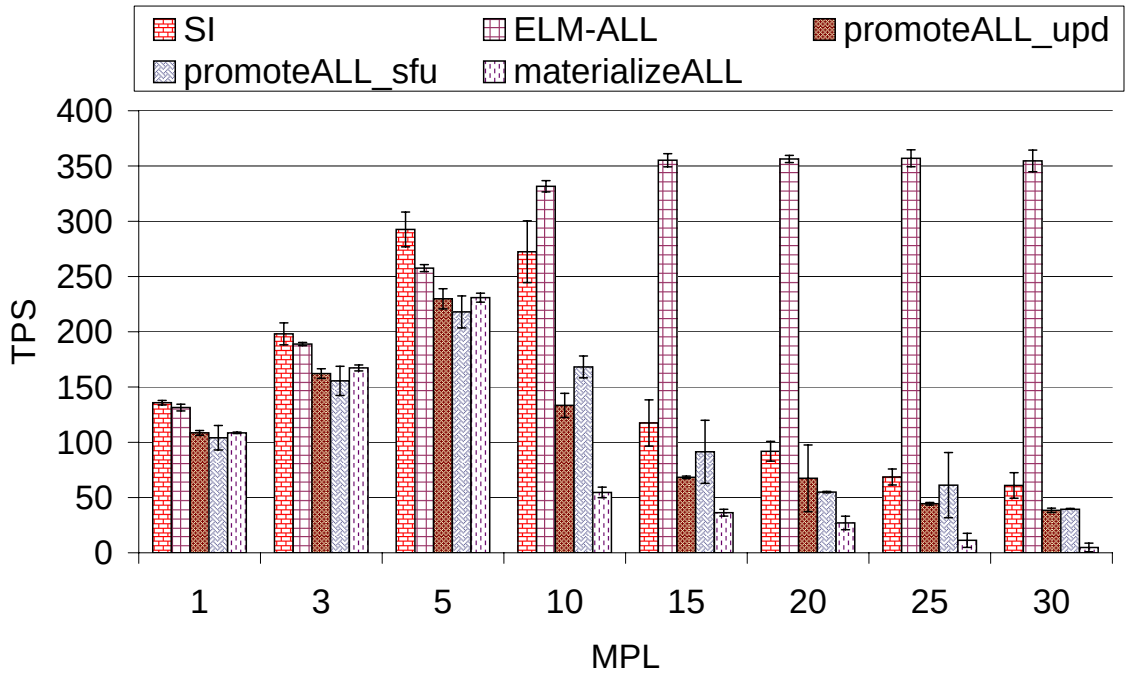

Figure 63: FCW Error per Transaction Type, High Contention, SmallBank, Oracle.

5.4 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 Chapter 4.

The choices for this benchmark are:

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

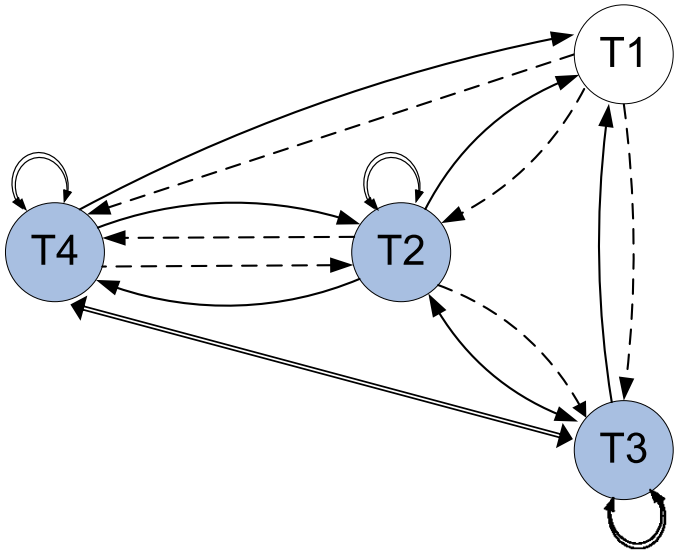

Figure 64: SDG for MoreChoice benchmark.

T1 ⇢ T4, T2 ⇢ T4, T4 ⇢ T2, T2 ⇢ T3.}

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

5.4.1 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 65. The overall message is: ELM performs as well, and even slightly higher than unmodified SI. [8] 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 66 shows the throughput in transaction per second (TPS) as a function of MPL for the different options with choice1 that guaranteeing serializable execution

[8] choice1 overlap with SI, choice2 is higher that SI after MPL=25.

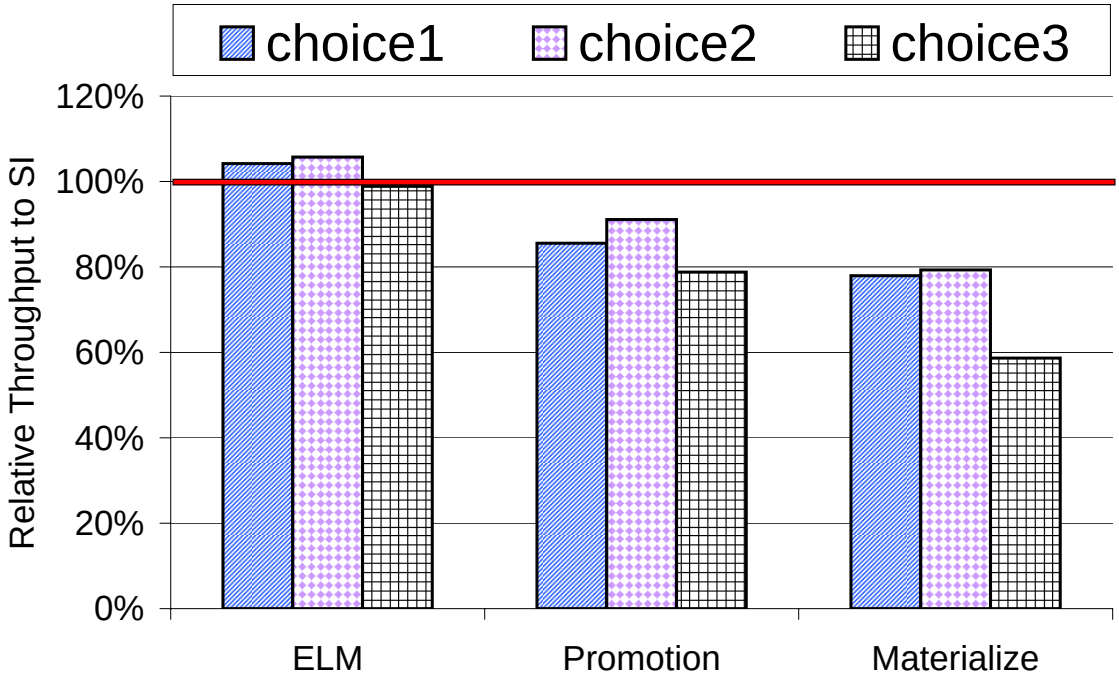

Figure 65: Throughput Summery, Low Contention, MoreChoices, PostgreSQL.

with the new benchmark. Where Figure 67 shows the percentage of serialization failure for each option with choice1. 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

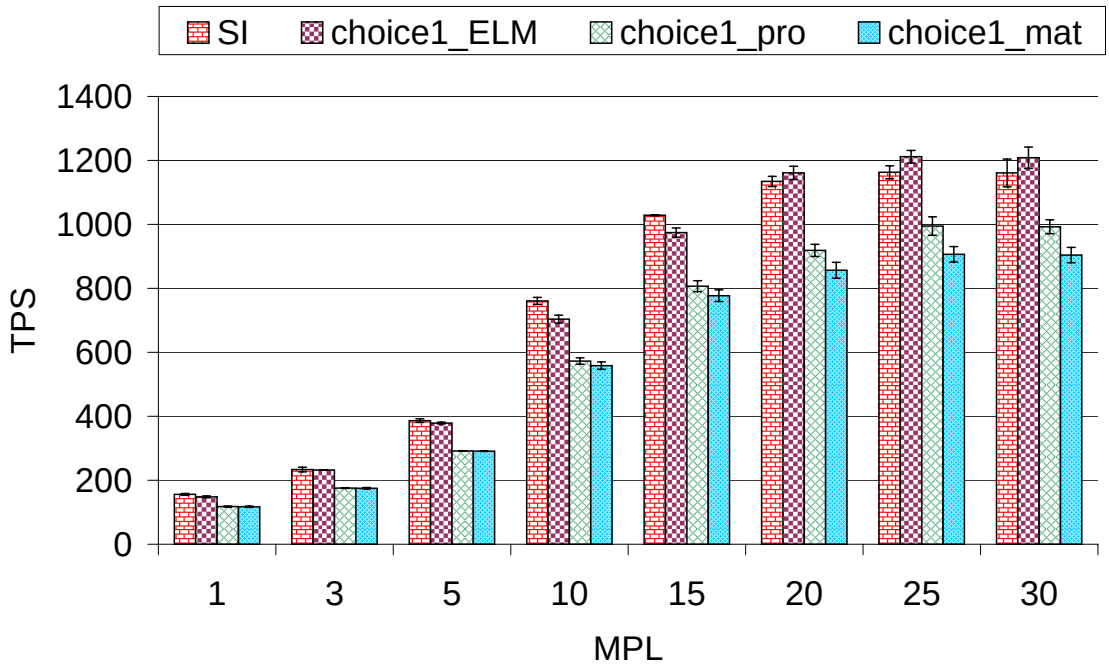

Figure 66: Choice1 Throughput, Low Contention, MoreChoices, PostgreSQL.

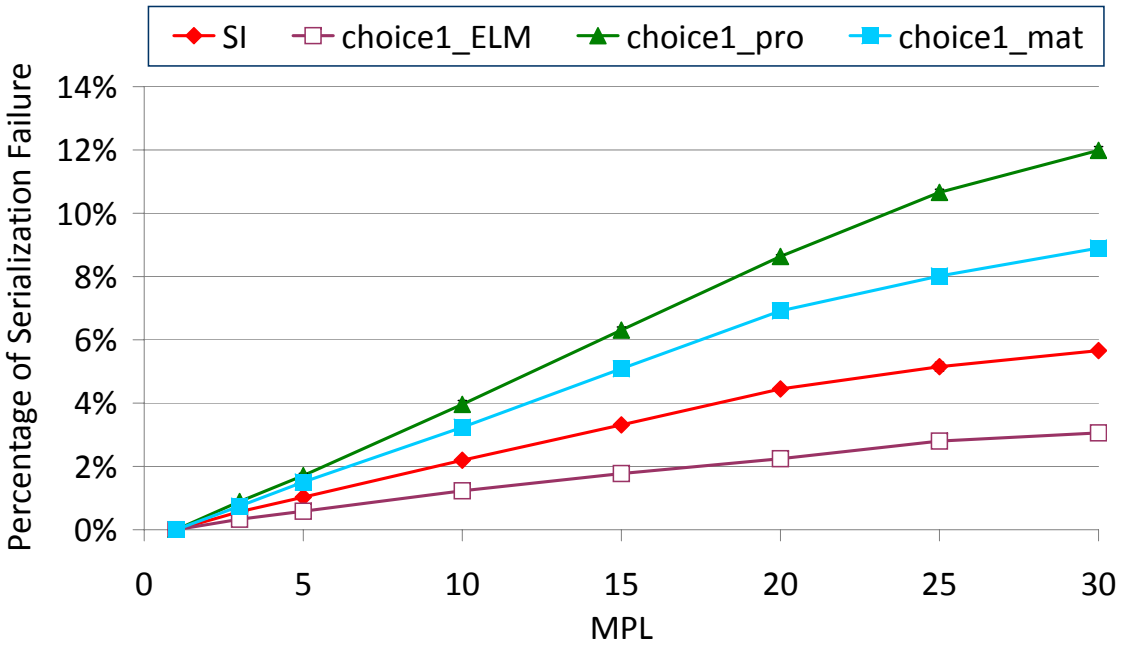

Figure 67: Serialization Failure for choice1, Low Contention, MoreChoices, PostgreSQL.

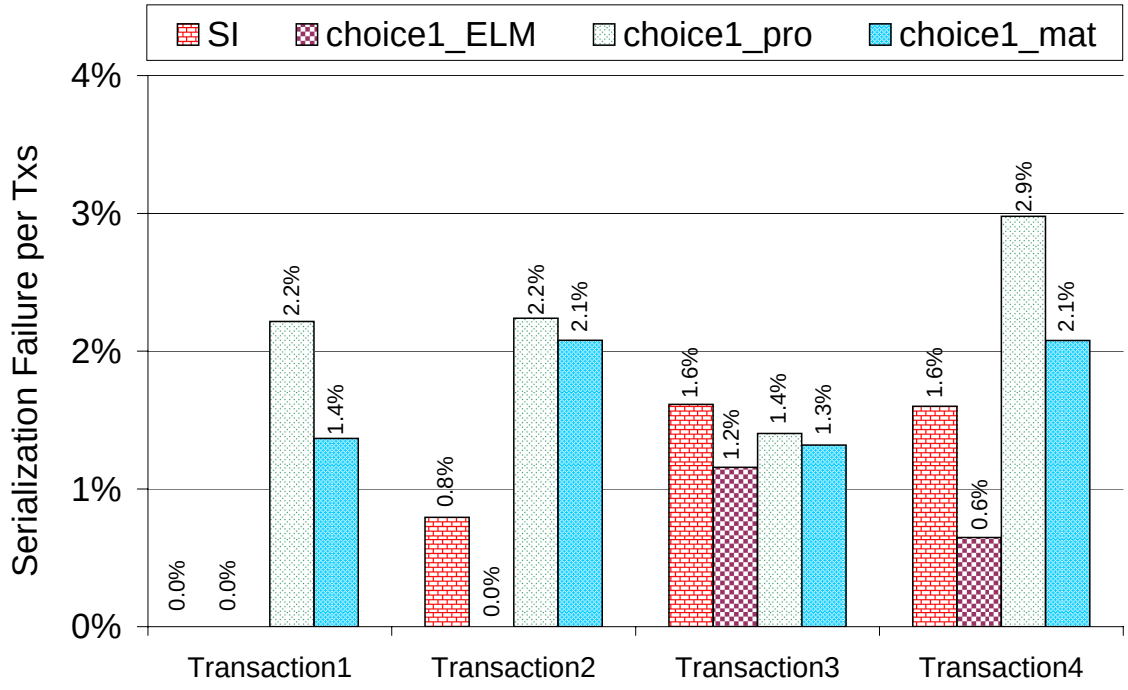

Figure 68: Serialization Failure per Transaction for choice1, Low Contention, MoreChoices, PostgreSQL.

Figure 68) .

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. [9] This explains Figure 67 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 69 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.

[9]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.

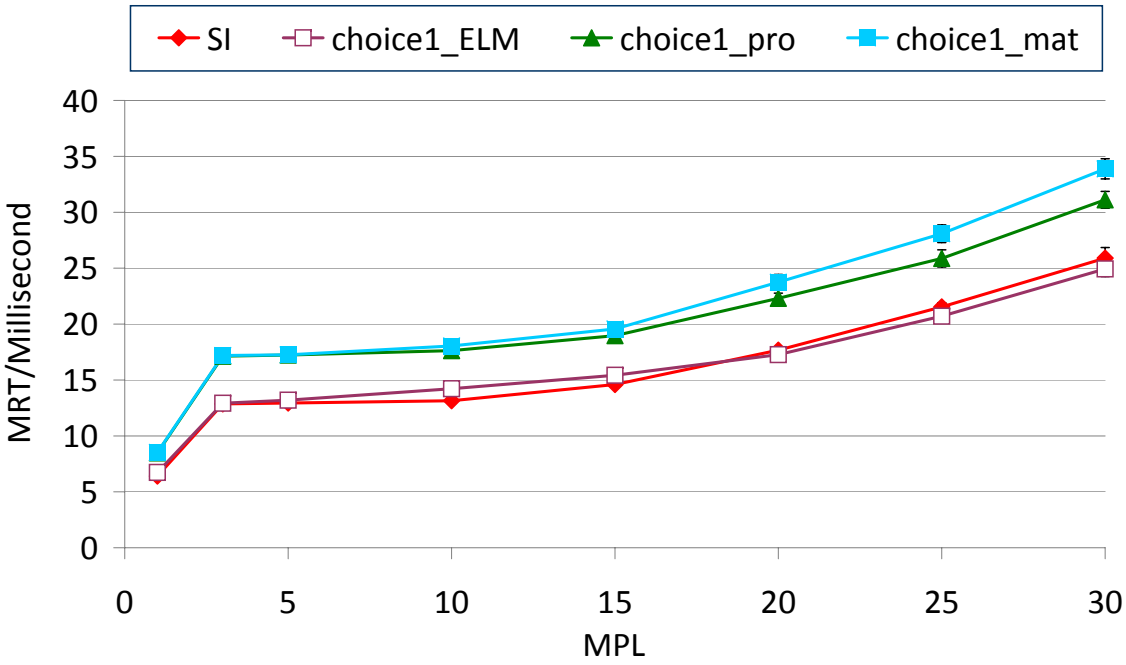

Figure 69: Choice1 Throughput, Low Contention, MoreChoices, PostgreSQL.

Choice2: Figure 70 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. Where figure71 shows the percentage of serialization failure for each option with choice2.

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%). [10]

Choice3: Choice3 considers the option when we remove every vulnerable edges from the SDG. Figure 72 and Figure 73 shows that the ELM technique is still superior over other available options, with zero 'Can not Serialize' errors.

[10]Choice2 control the concurrent update transactions $\{T_1, T_2, \text{and } T_4\}$ which explain zero serialiaztion failure.

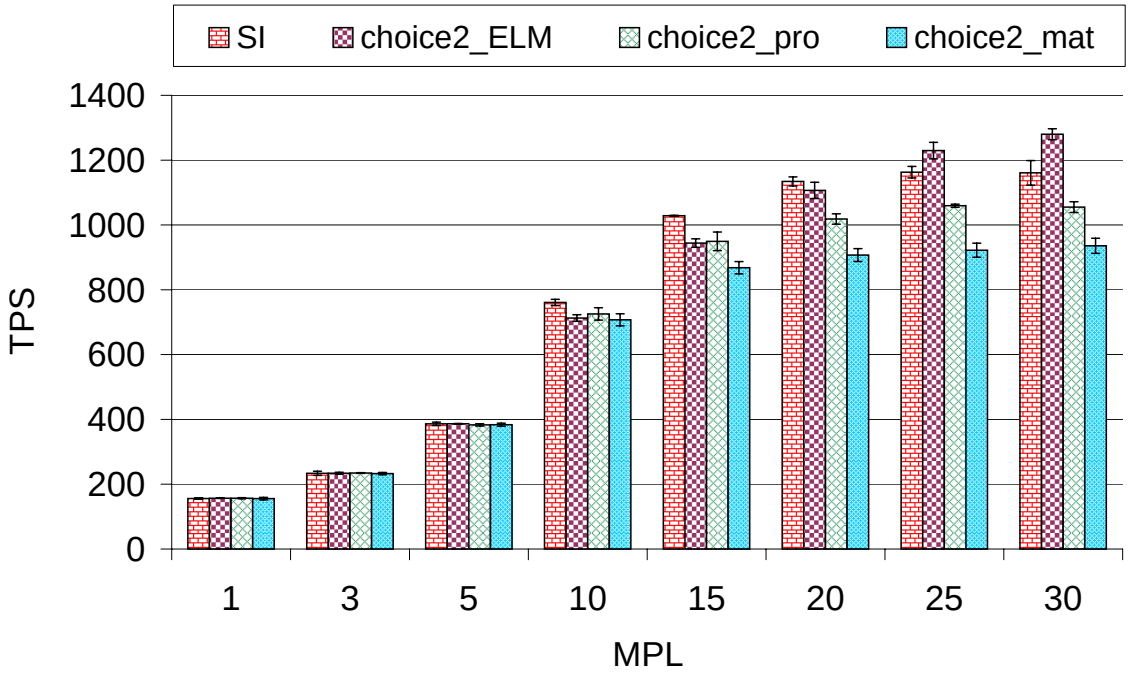

Figure 70: Choice2 Throughput, Low Contention, MoreChoices, PostgreSQL.

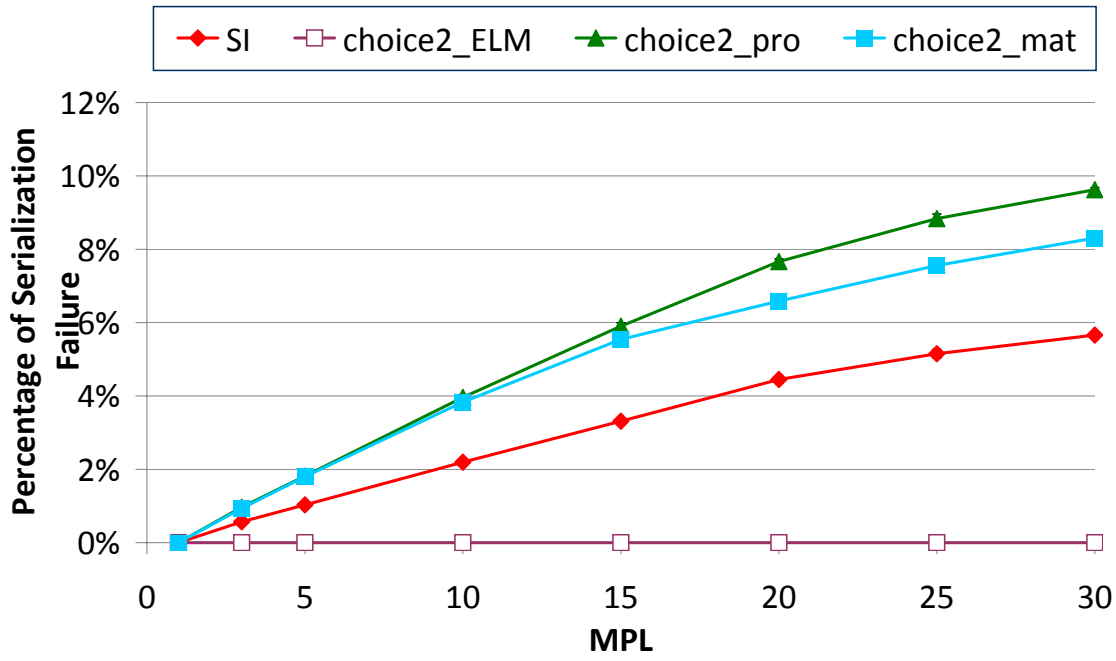

Figure 71: Serialization Failure for choice2, Low Contention, MoreChoices, PostgreSQL.

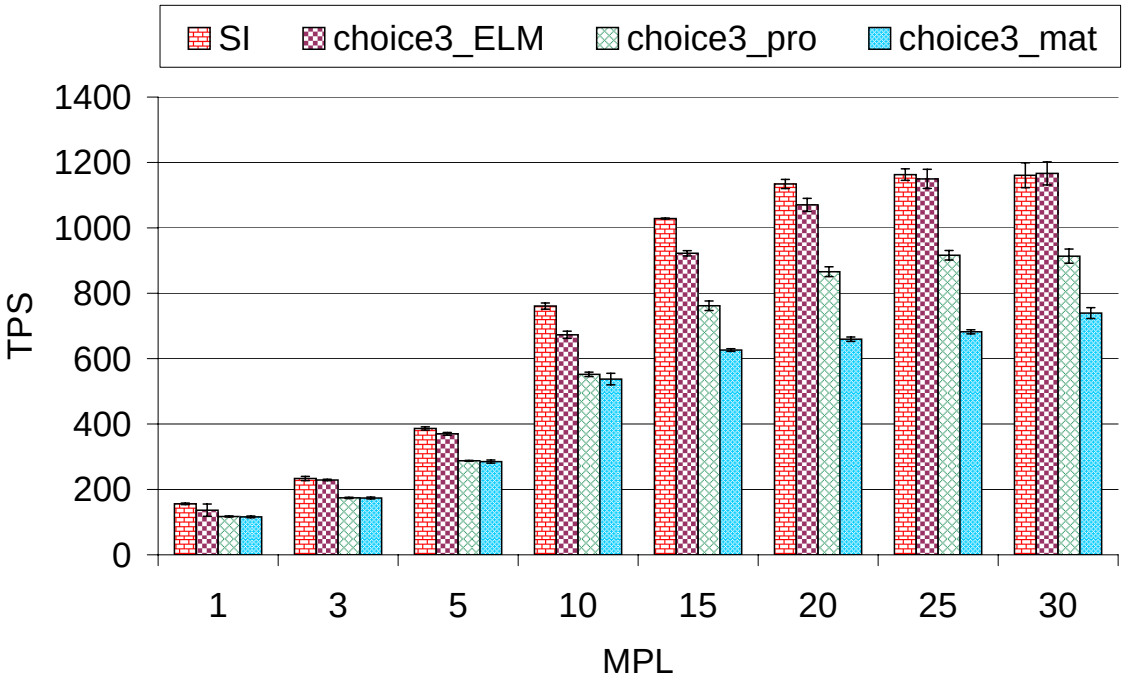

Figure 72: Choice3 Throughput, Low Contention, MoreChoices, PostgreSQL.

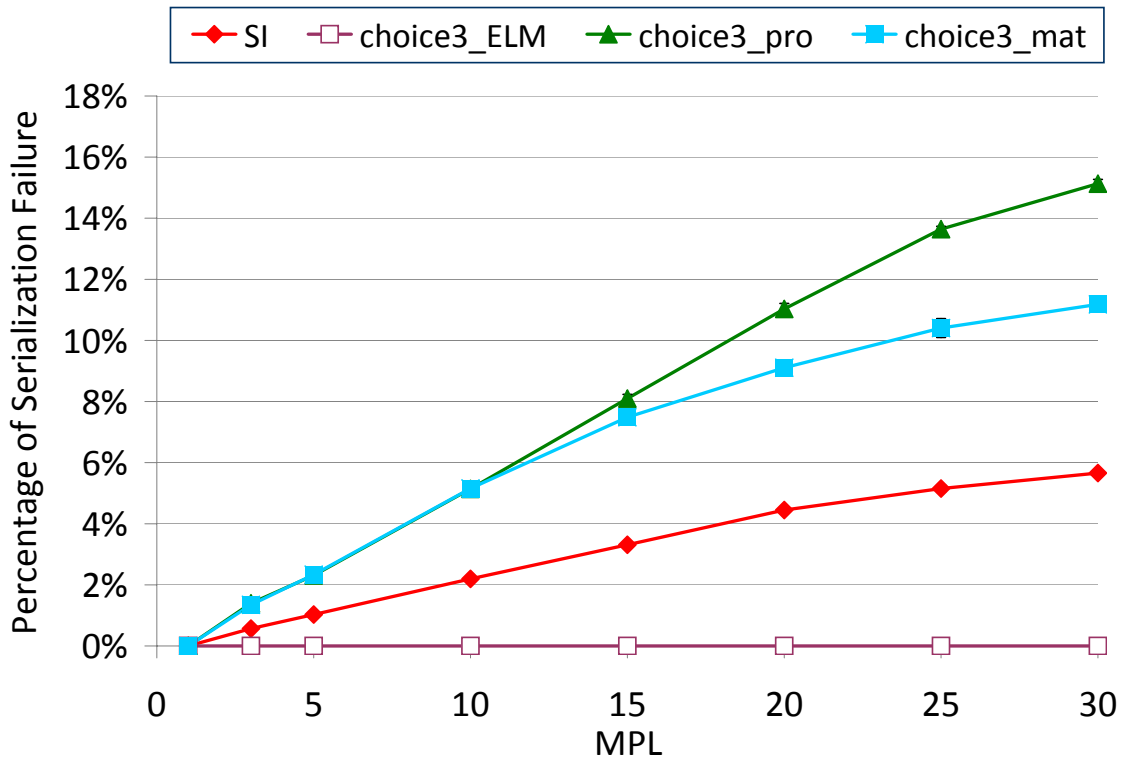

Figure 73: Serialization Failure for choice3, Low Contention, MoreChoices, PostgreSQL.

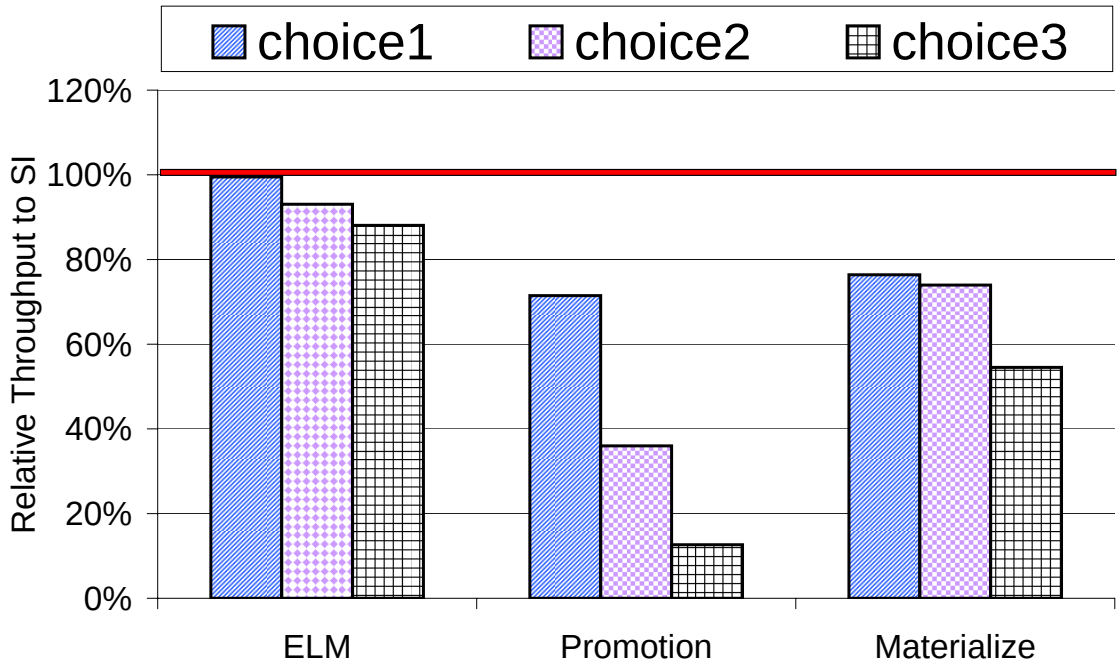

Figure 74: Throughput summary, High Contention, MoreChoices, PostgreSQL.

5.4.2 High Contention

We also ran the new benchmark under extreme conditions where the hotspot is only 10 rows. To make the whole picture understandable, we show the summary of the choices using different options (Materialize, Promotion, and ELM) with MPL=25 in Figure 74. The overall message is: ELM performs very close to unmodified SI: choice1_ELM is indistinguishable from SI, choice2_ELM is 93% of SI, and choice3_ELM is 88%. In contrast to the low contention conclusion, materialization is higher than promotion. Next we explore each choice in some detail.

Choice1: Figure 75 shows the throughput in transaction per second (TPS) as a function of MPL for the different options with choice1 that guaranteeing serializable execution with the new benchmark. Figure 76 shows the percentage of serialization failure for each option with choice1. We perceive that

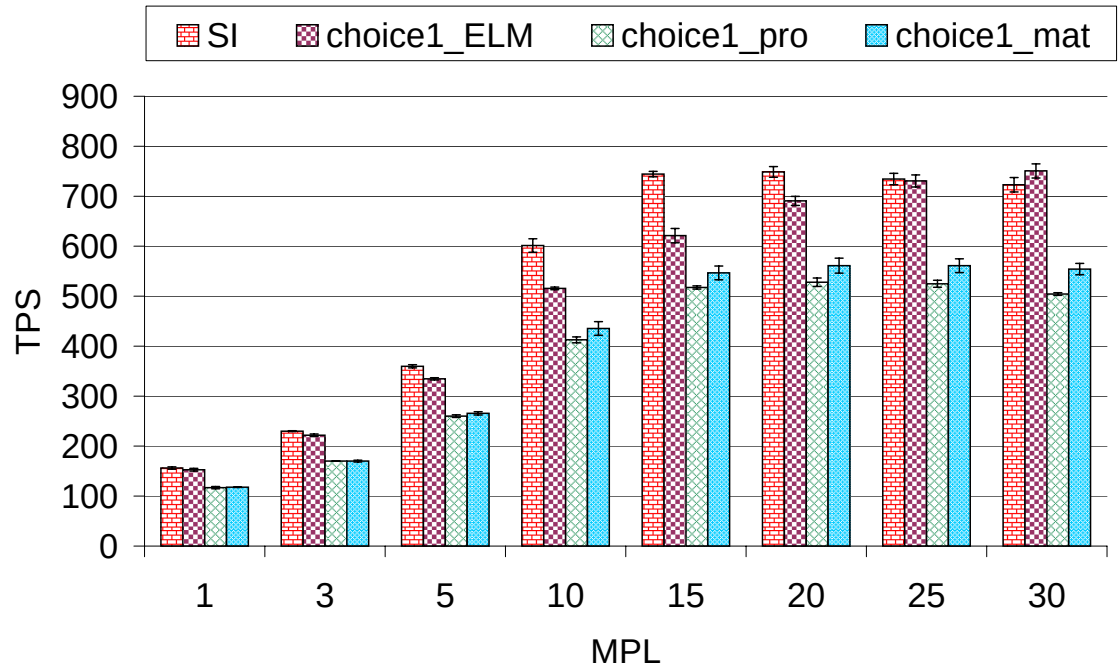

Figure 75: Choice1 Throughput, High Contention, MoreChoices, PostgreSQL.

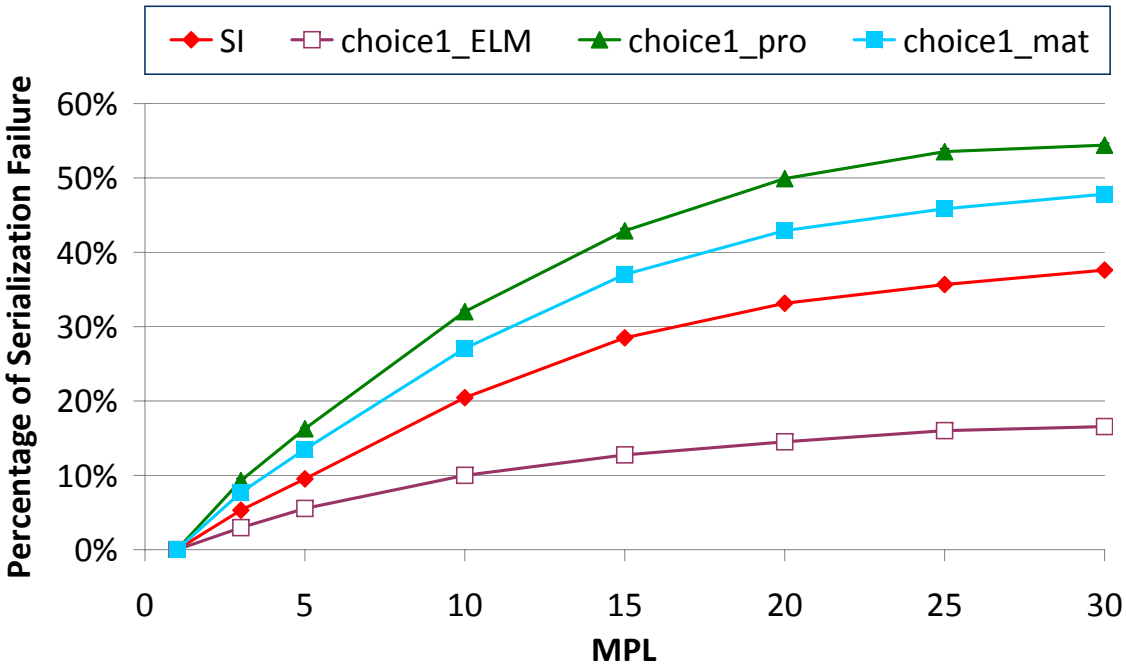

Figure 76: Serialization Failure for choice1, High Contention, MoreChoices, PostgreSQL.

- Throughput for choice1_Promotion (abbreviated as choice1_pro) using identity update is 25% lower than SI, and rises 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 is 5% lower than SI and rises till it reaches about 104% of that for SI with MPL=30.

We see that the conclusion from choice1 under a high data contention does not change much from the low data contention.

Choice2: Figure 77 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. Figure 78 shows the percentage of serialization failure for each option with choice2.

The data from choice2 shows some different conclusions than that with low contention. We see that materialize throughput is much better than promotion. The throughput degradation with promotion option comes as a result of increasing number of deadlock after we promote T_2 and T_4 transactions. [11] Figure 79 shows the percentage of deadlocks with choice2. We clearly see that choice2_pro has a high percentage of deadlocks where other options have zero deadlocks. Choice2_ELM does not increase the overall probability of deadlock, because of resource-ordering idea. Deadlock has an extreme affect on the throughput, since both transactions involved with the deadlock problem wait for a certain time till it resolves, which causes reduction in the number of committed transactions.

[11] after we promote T_2 and T_4, both transactions are waiting for resources that the other transaction hold. They keep holding the resources until the deadlock detection algorithm abort one of them.

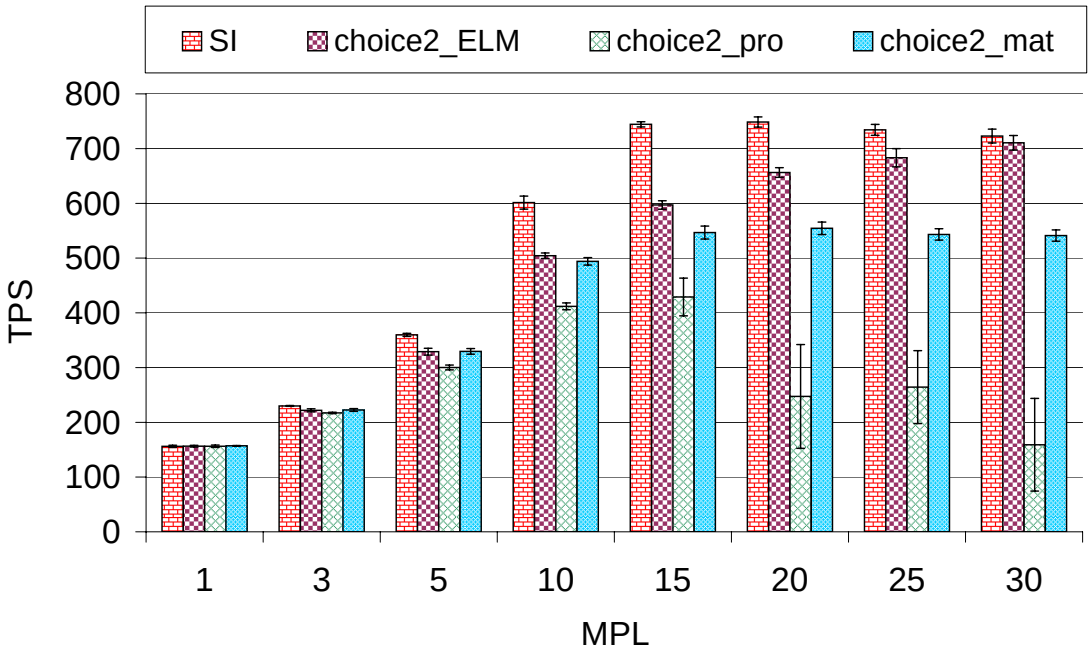

Figure 77: Choice2 Throughput, High Contention, MoreChoices, PostgreSQL.

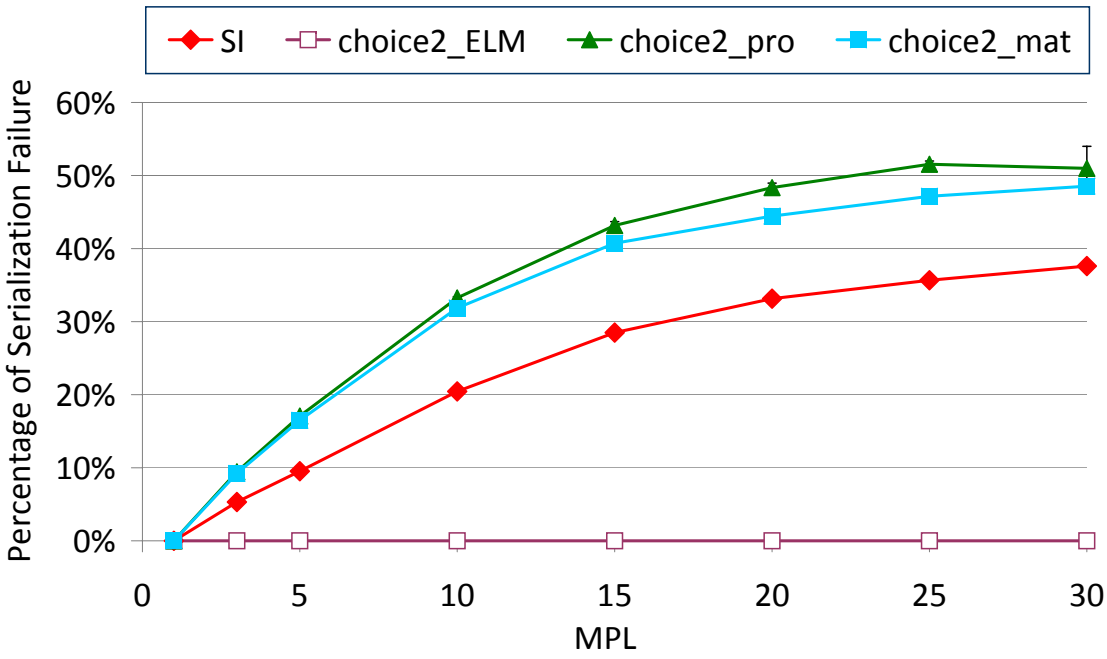

Figure 78: Serialization Failure for choice2, High Contention, MoreChoices, PostgreSQL.

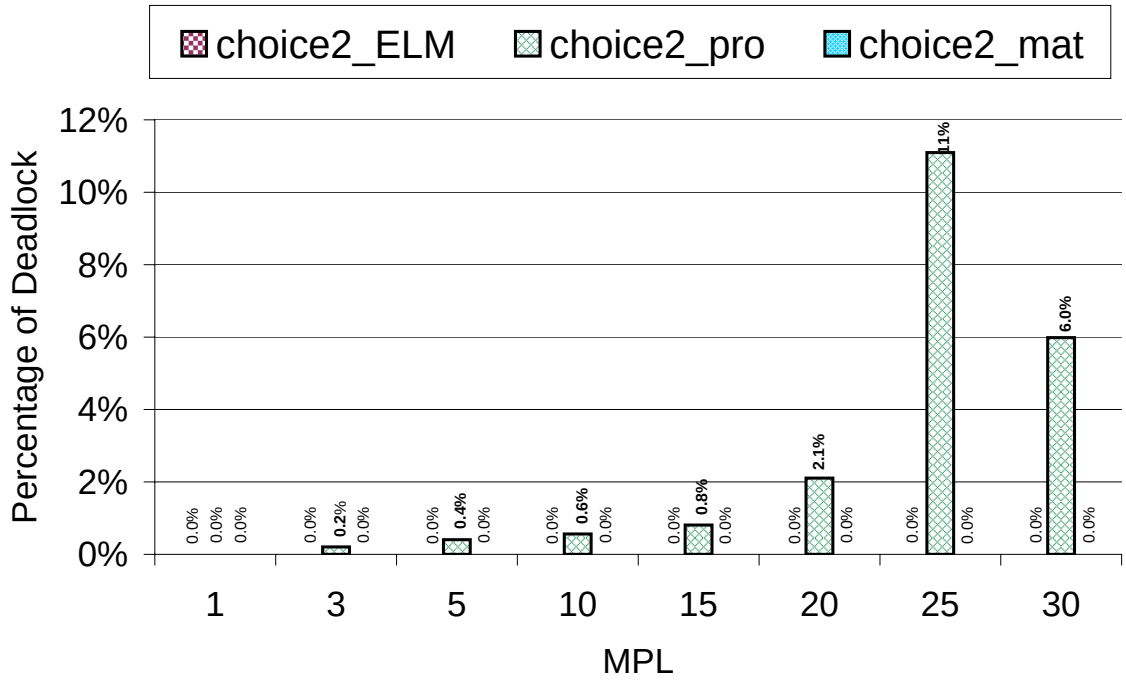

Figure 79: Percentage of Deadlock with choice2, High Contention, MoreChoices, PostgreSQL.

Choice2_ELM performs reasonably compared to unmodified SI and absolutely better than choice2_pro and choice2_mat.

Choice3: Figure 80 shows the throughput in transaction per second (TPS) as a function of MPL for the different options with choice3 that guaranteeing serializable execution with the new benchmark. Where Figure 81 shows the percentage of serialization failure for each option with choice3. choice3 has a very close throughput shape to choice2. choice3_pro also suffer from deadlock problem, which cause less throughput numbers than materialize option.

choice3_ELM throughput is 3% lower than SI and decreases relative to SI till it reaches about 91% of that for SI with MPL=30. However, choice3_ELM throughput is still reasonable compared to other techniques promotion and materialize.

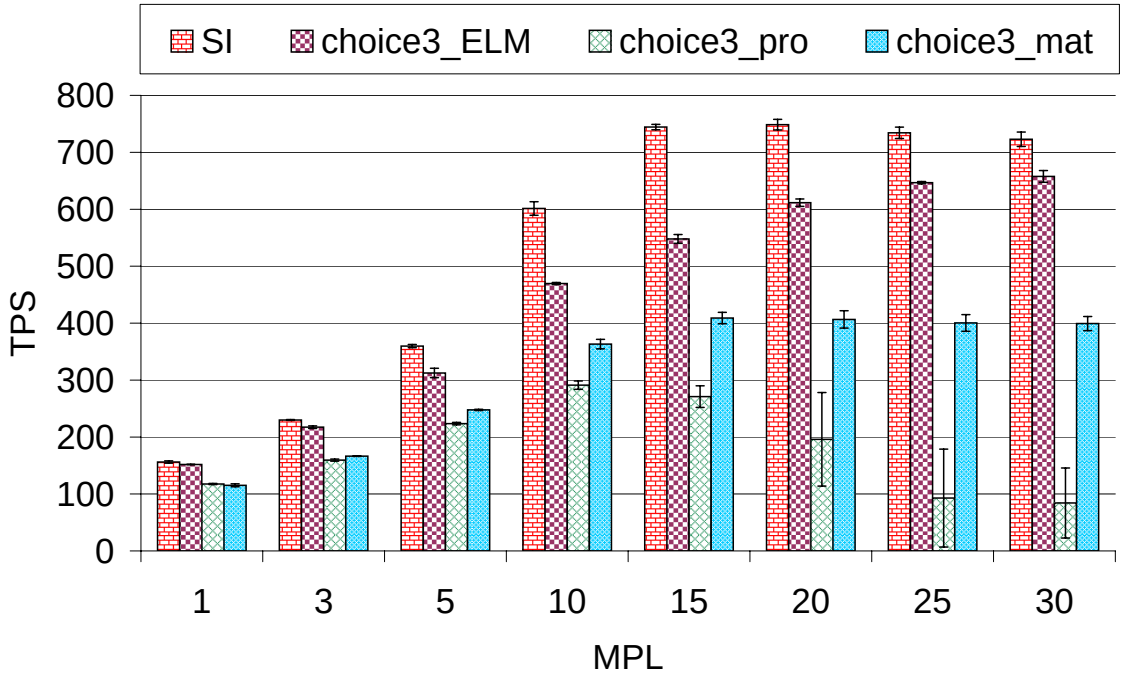

Figure 80: Choice3 Throughput, High Contention, MoreChoices, PostgreSQL.

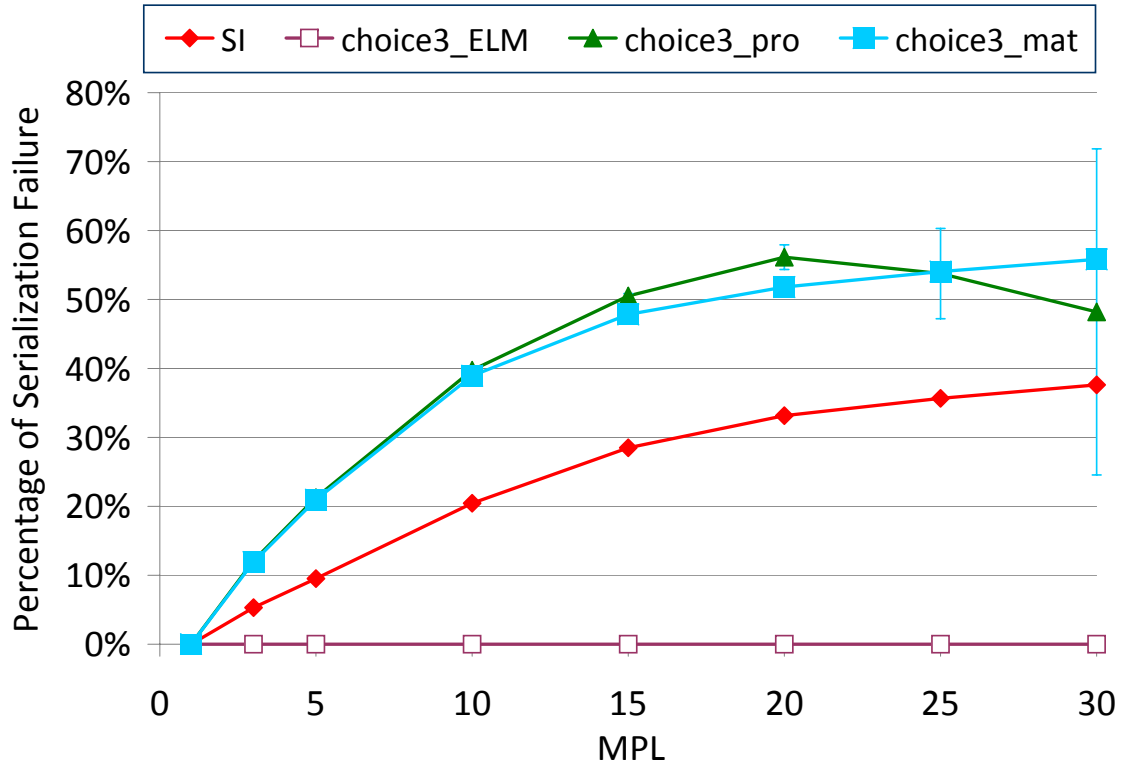

Figure 81: Serialization Failure for choice3, High Contention, MoreChoices, PostgreSQL.

5.5 Conclusions

We have run our experiments using two different platforms: one is an open sources database (PostgreSQL), and one is a commercial database (Oracle 10g). We used two of our own benchmarks called SmallBank and MoreChoices described in detail in Chapter 4. The experiments were run under two main data contentions: low contention (hotspot=100 customers) and high contention (hotspot=10 customers), while varying the number of concurrent clients (MPL) between 1 to 30. In some scenarios we changed the mix of the transaction from uniform distribution (20% each) to 60% for read-only transaction(BAL transaction in SmallBank) and 40% to the other update transaction (10% each) to create more realistic scenarios. We conclude the following

- Choosing which edge/edges to remove from the dangerous structure to ensure serializable execution is not straight forward when using Promotion and materialize techniques, since the choice has so much impact on performance.
- Materialize performs better than promotion in some cases and vice versa (this depends on which platform, contention, MPL, ..etc). In general, Promotion and Materialize are fragile, in that some edge choices (even some minimal edge choices) reduce throughput greatly.
- The percentage of each transaction in the mix can strongly affect the overall throughput, and it makes the choices between edges and techniques (Promotion and Materialize) is more complicated.
- On the other hand, ELM throughput is similar to the unmodified (anomaly-prone) programs, across the range of choices of which edges to modify. Thus ELM is robust against the DBA's choice of edge set for modification; even a simplistic choice, of introducing conflict in every vulnerable edge, does not lead to poor performance.

- With ELM, developers do not need to trade consistency for performance since it performs very close to RC in a low contention scenario (between 0%-10% less than RC) and much better than RC in a high data contention case (up to 137% of that for RC).

5.6 Summary

We have evaluated the External Lock Manager (ELM) technique using two platforms which support SI (PostgreSQL and Oracle), and using two benchmarks. These benchmarks have been designed to satisfy our need to find mixes of applications which can produce anomalies if run unmodified under SI.

ELM guarantees that all executions are serializable, and our experiments show that the modified applications have throughput which is similar to the unmodified (anomaly-prone) programs, across the range of choices of which edges to modify. Thus ELM is robust against the DBA's choice of edge set for modification; even a simplistic choice, of introducing conflict in every vulnerable edge, does not lead to poor performance. In contrast, the previous techniques of Promotion and Materialize are fragile, in that some edge choices (even some minimal edge choices) reduce throughput greatly.

Chapter 6

Conclusions

This thesis has focused on the problem of serializability of Snapshot isolation (SI) on platforms such as Oracle, SQL Server 2005, and PostgeSQL. It demonstrates the limitation of the existing techniques [25] called Promotion and Materialize that make SI serializable and it also presents a new technique called External Lock Manager (ELM) that shows more robustness.

The DBA has a complicated choice in order to ensure that all executions are serializable. The DBA needs to choose a subset of edges on which to introduce conflicts. There are different subsets of the edges in the SDG that include one from every pair of vulnerable edges that are consecutive in a cycle, and modification of the edges in one such subset are sufficient to guarantee serializable execution. [34] found that finding such a set with the fewest number of edges is NP-hard. Moreover, the DBA's have several options of which technique to use with the chosen subset of edges; they can modify the application in different ways.

The existing techniques, Promotion and Materialize, introduce SQL statements and these statements involve write operations, so that the standard SI mechanism in DBMS engine will prevent the transactions executing concurrently; one of the conflicting transactions

will abort with a 'Cannot Serialize" error which may require re-submitting the aborted transactions. Another effect of extra write operations is additional logging. We found that these aspects can greatly reduce throughput if a write is introduced into a program which was originally read-only.

We have designed and implemented a new technique called 'External Lock Manager' (ELM) that ensures serializable execution with platforms that support SI. 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. Locking is fine-grained, and, unlike traditional two-phase locking, it is parsimonious. Conflict is introduced when necessary to prevent SI anomalies, but not between most transaction executions. Most transactions set no ELM locks (if the program isn't adjacent to a chosen edge) or only one ELM lock (if the program is adjacent to one chosen edge, and the potential read-write dependency of that edge is on a single item). Because ELM locks are obtained before the database transaction begins, by doing resource ordering, we prevent any risk of deadlock involving ELM.

To evaluate these different techniques we need a benchmark, but the existing benchmarks do not allow evaluating the performance and the impacts of the above techniques because they do not have non-serializable executions at all. Thus, we have designed new benchmarks that allow us to stress-test the behavior of different ways to guarantee serializability for transactions running under snapshot isolation. These benchmarks demonstrated different cases (e.g. write skew, multiple cycles).

Our results shows that with ELM, the modified applications have throughput which is similar to the unmodified (anomaly-prone) programs, across the range of choices of which edges to modify. Thus ELM is robust against the DBA's choice of edge set for modification; even a simplistic choice, of introducing conflict in every vulnerable edge, does not lead to poor performance. In contrast, the previous techniques of Promotion and Materialize are fragile, in that some edge choices (even some minimal edge choices) reduce throughput greatly.

6.1 Future Work

Our ELM design has been implemented with ELM located on a separate machine, which leads to some limitations that can be improved. Some of these limitations which we saw earlier include:

- The two extra communication round-trips, that are placed in the execution path of those programs that are involved in the chosen SDG vulnerable edges.
- The ELM server could be seen as an additional single-point-of failure for those transaction programs that require an ELM lock.

Chapter 3 discusses alternative designs where we can implement ELM as middleware or as an additional component in the DBMS. One drawback to a middleware design that each client needs to access the ELM middleware which could cause overloading. Another drawback that is in case of middleware/ELM crash, clients need to wait until we fix the ELM. Note that implementing (coding and maintaining) the middleware is more complex than coding ELM itself as a separate node in the system. A comprehensive performance study needs to be conducted on the middleware design to compare it to other designs.

Further research is still needed on how we can integrate the ELM functionality into 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. This could also be beneficial for multi-core architectures, because no central synchronization between concurrent transactions is needed other than a few fine-granular ELM-locks for such transactions marked vulnerable by the DBA. Most importantly, such an integrated approach would be fully declarative to the DBA, not requiring any changes to client code. However, unlike our current design, we can not consider this design with commercial databases, since their codes are not visible. To implement this design we need to use open-source platforms such as PostgreSQL.

Fault tolerance is a non-functional (QoS) requirement that requires a system to continue to operate, even in the presence of faults. It should be achieved with minimal involvement of users or system administrators (who can be an inherent source of failures themselves). Distributed systems can be more fault tolerant than centralized (where a failure is often total), but with more processor hosts generally the occurrence of individual faults is likely to be more frequent. Fault tolerance in distributed systems can be achieved by: Hardware redundancy, i.e. replicated facilities to provide a high degree of availability and fault tolerance, and Software recovery, e.g. by rollback to recover systems back to a recent consistent state upon detection of a fault. However, the experiments reported in this thesis were done on a system without any fault-tolerance (and they measure executions without failures). We have suggested different alternatives in Chapter 3 for each design.

Our plan is to build a complete system that includes fault-tolerance, and then to evaluate it.

Also, in Section 3.1.1 we mentioned several options of what the ELM can lock. We discussed Edge-Name technique, Item-Name, Parameter-Value, or Very Fine-Granularity technique. We did not study the performance implications of these different choices. Future work is needed to explore these locking options, and come up with guidelines for making ELM locking choices. In addition, the choice of lock is so far done on a manual basis. We aim to develop a tool that can assist DBA automatically, to determine what locks to set in the ELM.

Bibliography

[1] Information technology-database languages. Found on the web at URL http://www.sqlteam.com/article, 2006.

[2] PostgreSQL 8.2.9 documentation. Found on the web at URL http://www.postgresql.org/docs/8.2/interactive, 2006.

[3] Tpc benchmarks. Found on the web at URL http://www.tpc.org, 2006.

[4] Tpc-c benchmark. Found on the web at URL http://www.tpc.org/tpcc/, 2006.

[5] Oracle documentation. Found on the web at URL http://www.oracle.com/technology/documentation/index.html, 2006-2008.

[6] Atul Adya, Robert Gruber, Barbara Liskov and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *SIGMOD Rec.*, Volume 24, Number 2, pages 23–34, 1995.

[7] Atul Adya, Barbara Liskov and Patrick E. O'Neil. Generalized isolation level definitions. In *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE)*, pages 67–78, 2000.

[8] Rakesh Agrawal, Michael J. Carey and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, Volume 12, Number 4, pages 609–654, 1987.

[9] Mohammad Alomari, Michael Cahill, Alan Fekete and Uwe Röhm. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels? In *Proceedings of DASFAA'08*, pages 267–281., 2008.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, 1995.

[11] Arthur J. Bernstein, Philip M. Lewis and Shiyong Lu. Semantic conditions for correctness at different isolation levels. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 57–66, 2000.

[12] Philip A. Bernstein and Nathan Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, pages 285–300. VLDB Endowment, 1980.

[13] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, Volume 13, Number 2, pages 185–221, 1981.

[14] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, Volume 8, Number 4, pages 465–483, 1983.

[15] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] Albert Burger and Vijay Kumar. Performance of multiversion concurrency control mechanism in partitioned and partially replicated databases. In *Proceedings of the 1992 ACM annual conference on Communications*, pages 109 – 119, 1992.

[17] Michael J. Carey and Miron Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, Volume 16, Number 4, pages 703–742, 1991.

[18] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst.*, Volume 4, Number 4, pages 338–378, 1986.

[19] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. Volume 2672, pages 997–1017, Rio de Janeiro, Brazil, 2003. ACM.

[20] C.J Date. *An Introduction to Database Systems*. Addison Wesley, sixth edition, 1995.

[21] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases (VLDB'06)*, pages 715–726. VLDB Endowment, 2006.

[22] Sameh Elnikety, Fernando Pedone and Willy Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, 2005.

[23] Alan Fekete. Serializability and snapshot isolation. In *Proceedings of the Australasian Database Conference (ADC'99)*, pages 201–210, 1999.

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

[25] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, Volume 30, Number 2, pages 492–528, 2005.

[26] Alan Fekete, Elizabeth O'Neil and Patrick O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, Volume 33, Number 3, pages 12–14, 2004.

[27] Jim Gray, Raymond A Lorie, G. R. Putzolu and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. pages 181–208, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[28] Jim Gray and Andreas Reuter. *Transaction Processing : Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1993.

[29] Thanasis Hadzilacos. Multiversion concurrency control scheme for a distributed database system. In *RIMS Symposia on Software Science and Engineering II*, pages 158–180. Springer Berlin / Heidelberg, 1986.

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

[31] Carl S Hartzman. The delay due to dynamic two-phase locking. *IEEE Transactions on Software Engineering*, Volume 15, Number 1, pages 72–82, 1989.

[32] Joseph M. Hellerstein, Michael Stonebraker and James Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, Volume 1, Number 2, pages 141–259, 2007.

[33] Ken Jacobs. Concurrency control: Transaction isolation and serializability in sql92 and oracle7. Technical Report A33745 (White Paper), Oracle Corporation, 1995.

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

[35] Michael Kifer, Arthur Bernstein and Philip M. Lewis. *Database Systems: An Application Oriented Approach, Compete Version*. Addison-Wesley, 2nd edition, 2006.

[36] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, Volume 6, Number 2, pages 213–226, 1981.

[37] A. Leff, J.L. Wolf and P.S. Yu. Efficient lru-based buffering in a lan remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, Volume 7, Number 2, pages 191–206, Feb 1996.

[38] Philip M. Lewis, Arthur J. Bernstein and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley, 2001.

[39] Wen-Te K. Lin and Jerry Nolte. Basic timestamp, multiple version timestamp, and two-phase locking. In *Proceedings of the 9th International Conference on VLDB*, pages 109 – 119, 1983.

[40] Yi Lin, Bettina Kemme, no-Martínez Marta Pati and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2005. ACM.

[41] Yi Lin and Sang Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. *In Proceedings of the Real-Time Systems Symposium, 11th*, pages 104–112, Dec 1990.

[42] Song Chun Moon. *Performance of concurrency control methods in distributed database management systems (timestamp ordering, two-phase locking, optimistic scheme, restart, transaction blocking)*. Ph.D. thesis, Champaign, IL, USA, 1985.

[43] David F. Nagle, Gregory R. Ganger, Jeff Butler, Garth Goodson and Chris Sabol. Network support for network-attached storage. In *Proc. of Hot Interconnects*, pages 86–92, Stanford, California, U.S.A, 1999.

[44] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, Volume 26, Number 4, pages 631–653, 1979.

[45] Chanjung Park and Seog Park. Alternative correctness criteria for multiversion concurrency control and its applications in advanced database systems. In *In Proc. of the Nineth Int'l Workshop on Database and Expert Sys. and Applications*, pages 864–869, 1998.

[46] Christian Plattner and Gustavo Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware'04)*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York.

[47] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, Volume 10, Number 1, pages 26–52, 1992.

[48] In Kyung Ryu and Alexander Thomasian. Analysis of database performance with dynamic locking. Volume 37, pages 491–523, New York, NY, USA, 1990. ACM.

[49] Abraham Silberschatz, Henry Korth and S. Sudarshan. *Database Systems Concepts*. McGraw Hill, 5th edition, 2005.

[50] Sang H. Son, Juhnyoung Lee and Yi Lin. Hybrid protocols using dynamic adjustment of serialization order for real-time concurrency control. *Real-Time Syst.*, Volume 4, Number 3, pages 269–276, 1992.

[51] Michael Stonebraker. The design of the postgres storage system. In Peter M. Stocker, William Kent and Peter Hammersley (editors), *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 289–300. Morgan Kaufmann, 1987.

[52] R. Sun and G. Thomas. Performance results on multiversion timestamp concurrency control with predeclared writesets. In *PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 177–184, New York, NY, USA, 1987. ACM.

[53] Yong Chiang Tay. *Locking performance in centralized databases*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[54] Alexander Thomasian. Performance limits of two-phase locking. In *Proceedings of the Seventh International Conference on Data Engineering (ICDE)*, pages 426–435, Washington, DC, USA, 1991. IEEE Computer Society.

[55] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Trans. Database Syst.*, Volume 18, Number 4, pages 579–625, 1993.

[56] Alexander Thomasian. Concurrency control: methods, performance, and analysis. *ACM Comput. Surv.*, Volume 30, Number 1, pages 70–119, 1998.

[57] Alexander Thomasian and In Kyung Ryu. Performance analysis of two-phase locking. *IEEE Trans. Softw. Eng.*, Volume 17, Number 5, pages 386–402, 1991.

[58] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE)*, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.