The Cost of Serializability on Platforms That Use Snapshot Isolation

Mohammad Alomari, Michael Cahill, Alan Fekete, Uwe Röhm

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

***Abstract*— Several common DBMS engines use the multi-version concurrency control mechanism called Snapshot Isolation, even though application programs can experience non-serializable executions when run concurrently on such a platform. Several proposals exist for modifying the application programs, without changing their semantics, so that they are certain to execute serializably even on an engine that uses SI. We evaluate the performance impact of these proposals, and find that some have limited impact (only a few percent drop in throughput at a given multi-programming level) while others lead to much greater reduction in throughput of up-to 60% in high contention scenarios. We present experimental results for both an open-source and a commercial engine. We relate these to the theory, giving guidelines on which conflicts to introduce so as to ensure correctness with little impact on performance.**

I. Introduction

The concepts of flat transactions and serializable execution are among the legacy Jim Gray has left in the database field [1]. While serializability can be seen as a transparency definition ("a correct system looks to the users just like a batch one, only faster"), in fact the value of this definition lies in how it supports the DBA who wants to reason about data integrity. In order to know that the final state of the data meets an integrity condition, the DBA can check each application program one-by-one. As long as each program separately preserves the constraint, the same will be valid for a serializable execution in which programs interleave. This is so, no matter what the integrity constraint is. Executions which are not serializable do not allow this; indeed a collection of correct programs can interleave non-serializably and cause data integrity to be lost, through lost-update, phantom reads, etc.

While it is well-known how to ensure serializable executions with concurrency control (through strict two-phase locking, along with appropriate techniques to lock indices), the performance impact of these techniques remains a deterrent to their wide usage. In recent years, many vendors have built platforms which make use of a multiversion concurrency control technique called Snapshot Isolation (abbreviated as SI) [2]. This is available in Oracle, PostgreSQL and Microsoft SQLServer 2005. Because SI does not delay readers, even if concurrent transactions have written the data involved, it generally offers high throughput (anecdotally, up to 3 times that for 2PL). SI is designed to completely prevent all the well-known anomalies that are taught in DB textbooks, and in particular, that are mentioned in the SQL standard as situations that should not happen at "isolation level serializable". SI ensures that executions avoid many bad situations, — it does not allow lost updates, or inconsistent reads, or phantom reads, etc.— and indeed several platforms (Oracle and PostgreSQL) actually use SI when the application is configured for "isolation level serializable". However, the executions produced by SI do not fit the textbook definition of serializable, because they can exhibit a phenomenon called *Write Skew*. Executions under SI can corrupt data when programs interleave, even though each program individually preserves the integrity constraints [2].

Some applications always give serializable applications, even when the platform uses SI. A famous example is the set of transaction programs that make up TPC-C. Theory has been developed to prove this, by considering a graph called the Static Dependency Graph (SDG) which shows inter-program conflicts [3]. The essential characteristic of a mix of programs without anomalous executions, is that the SDG does not contain a *dangerous structure* with two edges of a particular sort (called *vulnerable* edges) in a row.

If an application's mix of programs has no dangerous structures in the SDG, then we know that all executions will be serializable, even on a platform using SI for concurrency control. But if the graph does have a dangerous structure, we are at risk of data corruption! Thus we want to know how to modify the application's programs, so they will not have non-serializable executions. The SDG provides guidance in how to do this. Several methods have been proposed in [3], based on introducing extra update commands that do not alter the important data, but make sure that any anomalous execution would include a Lost Update, and therefore SI's existing mechanisms cause one or other transaction to abort in this situation.

The question arises: does the undoubted benefit of ensuring data integrity come at too high a price in lost performance? That is the issue we address in this paper.

Our main contribution is an extensive exploration of the performance impacts of different approaches each of which can guarantee serializable executions, despite the use of a platform with SI as its concurrency control mechanism. This is in Section IV of the paper. We also relate our results to the underlying theory and develop guidelines on which conflicts

978-1-4244-1837-4/08/$25.00 © 2008 IEEE

to introduce so as to ensure correctness with as little impact on performance as possible. In Section II, we review the related work, and give details of the definitions and results we use from the literature. In Section III, we present the particular mix of transaction programs on which we do our experiments. Section V is the conclusion.

II. Background

The Snapshot Isolation concurrency control mechanism (SI) was first published in 1995 in [2], where it was used to illustrate that non-serializable behavior could happen without any of the previously known anomalies which were listed explicitly in the SQL standard description of isolation levels. Also in 1995, Oracle7 introduced the mechanism as its implementation of "isolation level serializable" [4].

SI is a multiversion concurrency control mechanism. It depends on allowing transactions to access previous versions of each record, rather than always accessing the most recent state of the record. Thus reasoning about SI depends on a theory of multiversion concurrency control, such as [5] or [6].

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.

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. 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 it's snapshot. Thus we can describe this as "First Updater Wins" (in contrast to "First Committer Wins" described in [2]). Note that SI does not use read-locks at all, and a read is never blocked by any concurrent transaction.

SI was presented in [2] as a particular concurrency control mechanism, but Adya et al [7] have offered an abstract definition of the isolation level it provides (by analogy to the way true serializability is provided by two-phase locking, but can be characterized more abstractly as the absence of cycles in the serialization graph).

SI has been widely studied as a help in managing replicated data [8], [9], [10]; it is much cheaper to obtain global SI than to ensure true global serializability. Other work on replication has used slight variants of SI, following the same principles but relaxing the choice of start-timestamp [11], [12]

While [2] showed that SI allows non-serializable executions in general, it soon became apparent that some applications never experience these anomalies. In particular, the experts in the Transaction Processing Council could not find any non-serializable executions when the TPC-C benchmark [13] 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 [14], and a much more extensive theory is in [3]. The latter paper proves that the TPC-C benchmark has every execution serializable on an SI-based platform. Jorwekar et al [15] have shown that one can automate the detection of some cases where the theory of [3] holds. Fekete [16] 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 [17] showed how to prove that certain programs maintain a given integrity constraint when run under a variety of weak isolation levels, including SI.

A. Analysis Using SDG

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

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). The main theorem of [3] 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).

The papers described above give theorems which state that, under certain conditions on the programs making up an application mix, all executions of these programs will be serializable. What is the DBA to do, however, when s/he is faced with a set of programs that do not meet these conditions, and indeed may have non-serializable executions? A natural idea is to modify the programs so that the modified forms do satisfy the conditions; of course we want that the modifications do not alter the essential functionality of the programs. In [3], several such modifications were proposed. The simplest idea to describe, and the most widely applicable, is called *"materializing the conflict"*. In this approach, a new table is introduced into the database, and certain programs get an additional statement which modifies a row in this table.

Another approach is "*promotion*"; this can be used in many, but not all, situations. We give more detail of these approaches below.

The idea behind both techniques is that we choose one edge of the two successive vulnerable edges that define a dangerous structure, and modify the programs joined by that edge, so that the edge becomes no longer vulnerable. We can ensure that an edge is not vulnerable, by making sure that some data item is written in both transactions (to be more precise, we make sure that some item is written in both, in all cases where a read-write conflict exists). Clearly we need to do this for one edge out of each pair that makes up a dangerous structure. If there are many dangerous structures, there are many choices of which edges to make non-vulnerable. [15] showed that choosing a minimal set of appropriate edges is NP-hard.

B. 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

```
UPDATE Conflict
   SET val = val+1
 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 both committing concurrent transactions.

C. 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

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

We include in P an extra statement

```
UPDATE T
   SET col = col
 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.

In some platforms, there is another approach to promotion by replacing the SELECT statement (that is in a vulnerable read-write conflict) by `Select ... For Update`. In the commercial platform which we use in some experiments, this does not modify the data, but it is treated for concurrency control 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, 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.

D. Using 2PL

Another possible way to modify application programs is provided by [16]. If every pivot transaction is run with 2PL, rather than SI, then all executions will be serializable. Unfortunately, many platforms, including PostgreSQL, 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 have studied performance of these methods in a platform that does offer both SI and 2PL in [18]. In this paper, in contrast, we focus on comparing the methods suggested in [3], namely promotion and materialization, and especially in considering which edges to deal with.

III. The SmallBank Benchmark

Usually, performance measurements use a standard benchmark such as TPC-C [13] which is carefully designed to exercise a range of features of a system. We cannot use TPC-C to compare different ways of making applications serializable, since TPC-C itself generates only serializable executions on SI-based platforms, as has been known since Oracle obtained benchmarks. This was proved formally in [3]. Thus in this paper we have used a new benchmark mix which is contrived to offer a diverse choice among modifications that will ensure serializable execution on SI. We call it the *SmallBank* benchmark. It is based on the example of an SI anomaly from [19], and provides some functionality reflecting a small banking system, where each customer has a pair of accounts, one for savings and one for checking.

A. 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.

B. Transaction Mix

The SmallBank benchmark runs instances of five transaction programs. *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.

DepositChecking, or DC(N,V), is a parameterized transaction that represents making a deposit on the checking account of a customer. Its operation is to look up the Account table to get CustomerID corresponding to the name N and increase the checking balance by V for that CustomerID. If the value V is negative or if the name N is not found in the table, the transaction will rollback.

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

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

WriteCheck, or WC(N,V), represents writing a check against an account. Its operation is to look up Account to get the CustomerID value for N, evaluate the sum of savings and checking balances for that CustomerID. If the sum is less than V, it decreases the checking balance by V+1 (reflecting a penalty of 1 for overdrawing), otherwise it decreases the checking balance by V. Program 1 shows the core of the SQL code.

Program 1 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;
```

C. The SDG for SmallBank

Figure 1 shows the SDG for our 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 edge from WC (which reads the appropriate row in both Checking and Saving, and only updates the row in Checking). Since TS writes Saving but not Checking, the edge from WC to TS is vulnerable. In contrast, whenever Amg writes a row in Saving it also writes the corresponding row in Checking; thus if there is a read-write conflict from WC to Amg on Saving, there is also a write-write conflict on Checking (and so this cannot happen between concurrently executing transactions). That is, the edge from WC to Amg is not vulnerable.

We see that the only dangerous structure is from Balance (Bal) to WriteCheck (WC) to 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 [19], 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.

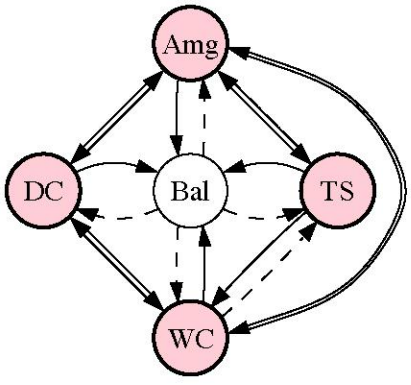


Fig. 1. The SDG for the SmallBank benchmark.

D. Ways to Ensure Serializable Executions

We have two options to eliminate the dangerous structure: 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). And as discussed in Section II-A, we further have three alternatives on how to make each option non vulnerable.

*a) **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.

```
UPDATE Conflict
   SET Value = Value+1
 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.

```
UPDATE Saving
   SET Balance = Balance
 WHERE CustomerId=:x
```

In the commercial platform we consider, there is also a strategy *PromoteWT-sfu*, where the second SELECT statement in the code above for WC is replaced by

```
SELECT Balance INTO :b
  FROM Saving
 WHERE CustomerId=:x
   FOR UPDATE
```

In Figure 2 above, we show the SDG for these. Only the edge between WriteCheck and TransactSaving has changed, the remaining edges are unchanged.

*b) **Option BW**:* We can also ensure that all executions are serializable, by changing the programs so that the edge from Bal to WC is not vulnerable. This can again be done by materializing (which includes an update on Conflict in both programs Bal and WC), or by promoting with identity update on the table Checking in Bal, or (in the commercial platform only) by promoting with select-for-update on table Checking in Bal. We represent these strategies as *MaterializeBW*, *PromoteBW-upd* and *PromoteBW-sfu*.

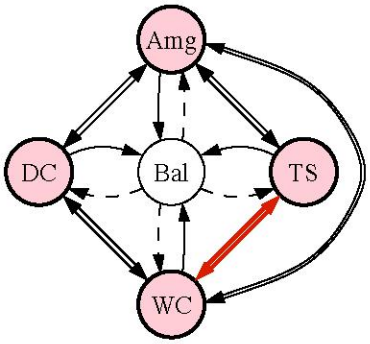
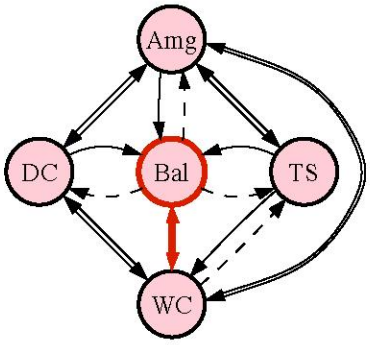
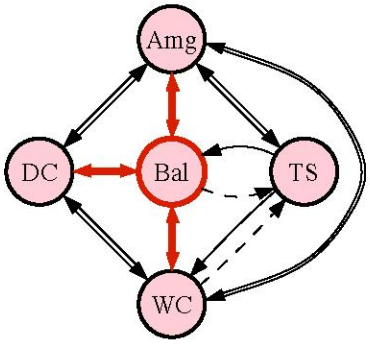


Fig. 2. SDG for Option WT.



(a) SDG for MaterializeBW.



(b) SDG for PromoteBW-upd.

Fig. 3. SDGs for Option BW.

In Figure 3 are the SDGs for these; again we only show the vulnerable edges. Note that the Balance transaction is no longer read-only and that in Figure 3(b) (PromoteBW-upd), other outgoing edges from Balance have changed.

*c) **MaterializeALL and PromoteALL**:* 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 or by promotion). We refer to these strategies as *MaterializeALL* and *PromoteALL*. 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.

E. Summary of Transaction Modifications

We presented six strategies to ensure serializable executions of the SmallBank benchmark, each of which modifies some of the transactions with an additional update, or at least introduces a select for update (with the commercial platform). Table I summarises these different modifications. It lists for each type of transaction which additional updates on either the `Saving` table (Sav), the `Checking` table (Check) or to the dedicated `Conflict` table (Conf) have been introduced. Note that, except for Option WT, all options introduce updates into the originally read-only Balance transaction.

TABLE I
OVERVIEW OF TABLES UPDATED WITH EACH OPTION.

Option / TX	Bal	WC	TS	Amg	DC
MaterializeWT		Conf	Conf		
PromoteWT		Sav			
MaterializeBW	Conf	Conf			
PromoteBW	Check				
MaterializeALL	Conf	Conf	Conf	Conf	Conf
PromoteALL	Check, Sav	Sav			

IV. EVALUATION

We have conducted a series of experiments with our SmallBank benchmark to explore the impact of different ways of modifying application programs in order to ensure the serializable execution on a SI-based platform.

The experiments were carried out 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. The database engine was PostgreSQL 8.2 and one commercial database system supporting SI. We populated our SmallBank database with 18000 randomly generated customers and their checking and savings accounts. 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 = 1s, thus taking advantage of group commit.

The actual test driver is running on a separate client machine that connects to the database server through Fast Ethernet. The client machine is running Windows XP 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 multithreading.

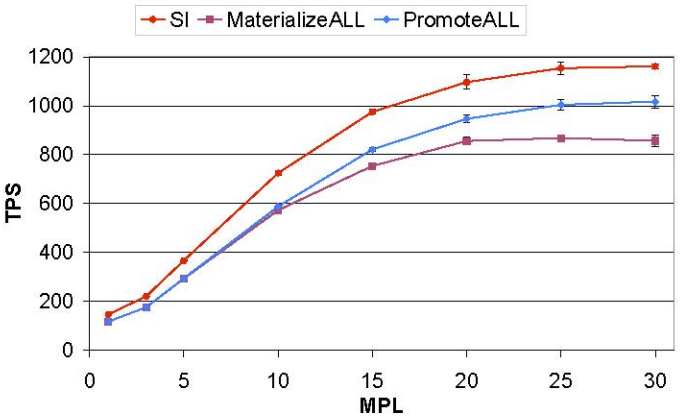


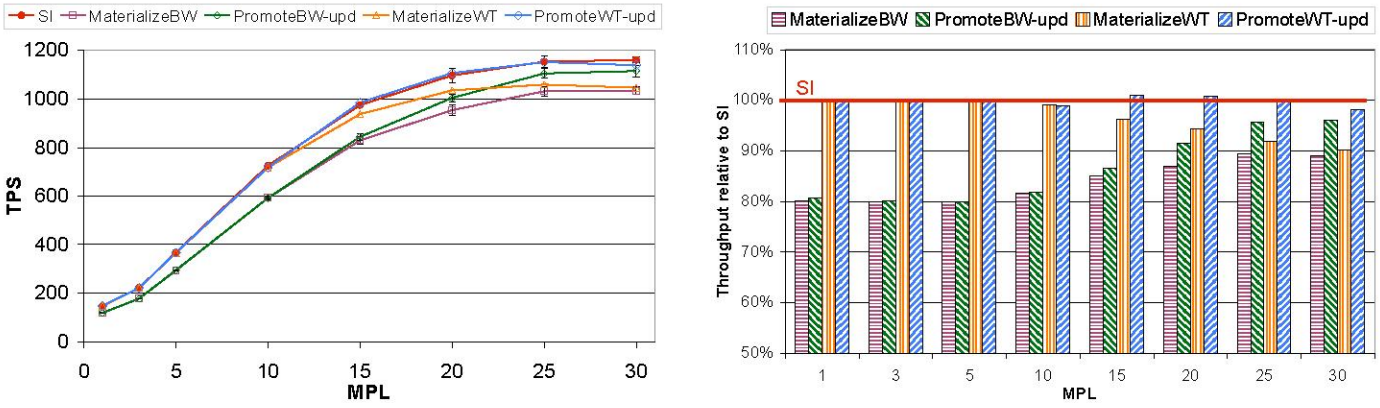
Fig. 4. Costs for SI-serializability when eliminating ALL vulnerable edges (PostgreSQL).

The test driver runs the five possible transactions. Most experiments do so with uniform random distribution, but we also consider a mix with 60% Balance transactions. The actual transaction code is executed as stored procedures on the database server, which the client threads invoke with randomly chosen parameters. A fixed portion of the table is a hotspot, and 90% of all transactions deal with a customer which is chosen uniformly in the hotspot. In most of our experiments the hotspot has 1000 (of the 18000) customers, but when we consider high contention, we make the hotspot have 10 customers only. The other 10% of transactions access uniformly from the customers outside the hotspot. Our system is a closed system; each thread runs the selected transaction and waits for the reply, after which it immediately (with no think time) initiates another transaction.

We ran experiments on PostgreSQL comparing the throughput of SI (which allows non-serializable executions) to our different approaches to guarantee serializability as discussed in Section 3. 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.

A. Eliminating All Vulnerable Edges

We first consider the straight-forward strategies that remove the vulnerability from every vulnerable edge. These modify many transactions, but they do not require the DBA to look for cycles and dangerous structures in the SDG; instead the DBA can think about each pair of transactions separately. Figure 4 shows the resulting throughput in Transactions Per Second



(a) Throughput over MPL.

(b) Throughput relative to SI.

Fig. 5. Eliminating the BW and WT vulnerabilities (PostgreSQL).

(TPS) as a function of MPL. As we see, all simple approaches induce hefty performance costs. Promoting every vulnerable edge has performance that start 20% lower than SI and rises till it reaches about 95% of that for SI. Materializing on every vulnerable edge gives performance that peaks at about 850 TPS (about 25% 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.

B. Eliminating the Vulnerable Edge WT

Next, we are comparing SI on PostgreSQL with MaterializeWT and PromoteWT-upd (each of which guarantees that all executions are serializable by eliminating the vulnerability between WriteCheck and TransactSaving). In Figure 5, we show two graphs: on the left is again the throughput in TPS as a function of MPL, and on the right we show the relative performance as compared to the throughput with SI (shown as red line) for each strategy that ensures serializable executions.

We observe that

- Throughput for SI increases at first with MPL, but plateaus for MPL from 25, at about 1150 TPS.
- Throughput for PromoteWT is indistinguishable from that for SI.
- Throughput for MaterializeWT matches that for SI at low MPL, but then grows less rapidly. It reaches a plateau of about 1050 TPS, so about 90% of the peak performance of SI.

C. Eliminating the Vulnerable Edge BW

We ran similar experiments on PostgreSQL to explore the strategies that eliminate the vulnerability of the edge from Balance to WriteCheck. The results are also shown in Figure 5. This time, we observe that

- Throughput for SI increases at first with MPL, but plateaus for MPL from 25, at about 1150 TPS.
- Throughput for PromoteBW rises less rapidly than for SI, but reaches the same peak performance as SI at MPL 30.
- Throughput for MaterializeBW grows less rapidly, and reaches a plateau of about 1050 TPS, so about 90% of the peak performance of SI.
- For all approaches, the performance costs are higher with low MPLs than with higher MPLs, which is the reverse of the findings for Option WT.

D. Discussion, Comparing Strategies

Our results clearly show that better throughput at a given MPL usually comes from eliminating the vulnerability on the WT edge than on the BW edge. This fits the analysis from Figure 2 and Figure 3, since 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). In contrast MaterializeBW and PromoteBW-upd introduce a write into Balance, and thus make every transaction need a disk write. This is clearly seen in the performance with MPL=1, where (with a single thread submitting transactions) there is no contention at all, and the slowdown comes only from the overhead. We see a slowdown of 20% for those modifications that increase the fraction of transactions that must do disk-writes, by 5/4, and no slowdown at MPL=1 for the other modifications (cf. Figure 5(b)). This clearly shows that the need to write to disk is overwhelmingly dominant in the work done; once a transaction needs one write, extra writes have negligible extra cost.

At higher MPL, there is also a performance impact from the extra contention that a modification can cause (reflected either in blocking or aborts). Here we can use the SDGs Figures 1, 2 and 3. We see that promotion or materialization of WT does not introduce extra contention beyond that intended (between Bal and DC), but that 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

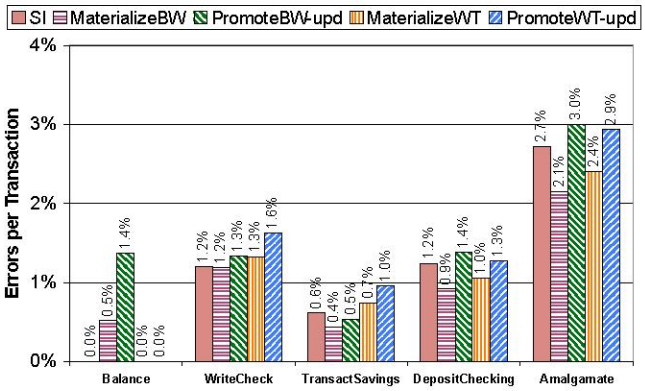


Fig. 6. Comparison of abort rates (PostgreSQL).

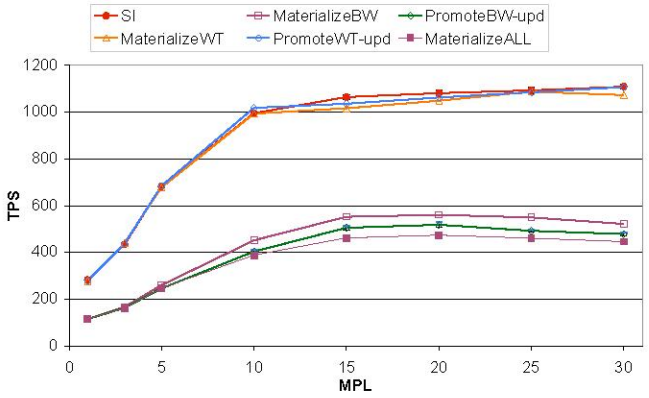


Fig. 7. Costs with high contention (PostgreSQL).

update, or select-for-update, on the appropriate row of the Checking table. This is confirmed in Figure 6 which shows the number of aborts due to a "serialization failure" error, of each separate transaction type. This is measured at MPL=20 in each different strategy. We see a significant increase in abort rates for Balance, DepositChecking and Amalgamate, under PromoteBW, compared to those under SI, MaterializeBW or MaterializeWT.

We do not have an explanation of the poor throughput of Materialize options compared to promoting the corresponding edge (especially as Materialize often has lower abort rates). In Section IV-F we see that a different implementation of SI, in a commercial platform, does not show this effect.

E. High Contention

We have also run experiments to show more extreme effects. We changed the size of the hotspot region, from 1000 customers (out of 18000 in the table) to only 10 customers, and we also changed the transaction mix, so that 60% of transactions are Balance. The increased contention from the small hotspot works against overall throughput, but the increasing amount of Balance transactions (which are read-only in the original coding) helps throughput. Peak throughput for SI in this workload is about 1100 transactions per second. The impact of various strategies for ensuring serializable execution varies greatly.

We still find that eliminating the vulnerable WT edge gives minimal (if any) drop in performance compared to SI, while other strategies have much lower peak performance, reaching about 560 TPS for MaterializeBW, and about 460 TPS for MaterializeALL or PromoteALL. Figure 7 illustrates this, showing again that a SDG analysis of the transaction mix is essential to achieve reasonable performance and serializability under SI at the same time.

F. A Commercial Platform

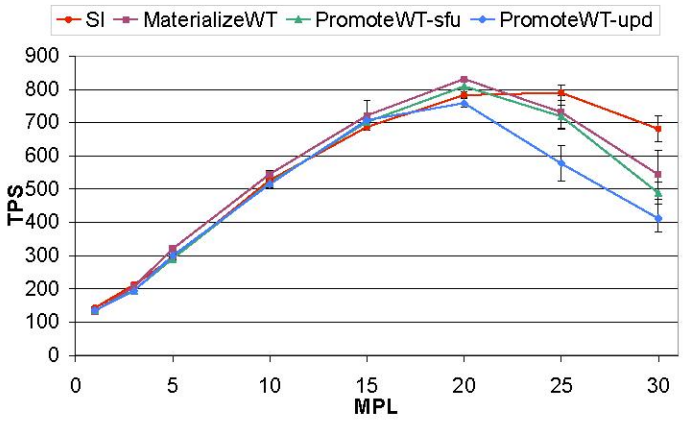
So far, we have focused on PostgreSQL because we can seek understanding of our observations from the detailed implementation. For comparison, we also ran our experiments on one of the commercial platforms that offer Snapshot Isolation concurrency control. Figures 8 and 9 show the results. It is notable that the overall shape of the graphs is quite different, with much more sensitivity to the careful choice of MPL: TPS reaches a peak of around 800 TPS between 20 and 25 MPL and then declines rapidly, instead of staying at a plateau. We find that PromoteWT-sfu reaches essentially the same peak as SI itself, though it then declines faster at MPLs over 20. PromotionWT-upd has similar performance up to the peak and then declines even faster. All methods of eliminating the vulnerability on BW do substantially worse, with peak throughput lower by at least 10% than that for SI. In particular PromoteBW-upd peaks at only 630 TPS, which is only 80% of SI's throughput.

Thus we see that the general conclusions remain: some ways of ensuring serializable executions do not reduce performance significantly, while others do; and eliminating the vulnerability on WT is generally preferable to eliminating it on BW. However, differences in implementation lead to different optimal decisions. In particular, on the commercial platform, promotion by select-for-update often does better than promotion by update, and materialization generally does better than promotion (the reverse of our findings for Postgres).

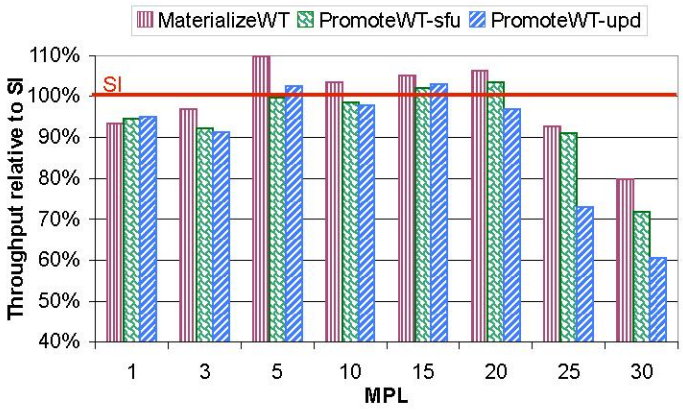
G. General Guidelines.

As our experiments have shown, there are substantial performance differences among the proposed approaches. We conclude the following guidelines for guaranteeing serializability on SI-based platforms from our experiments:

1) Analyse the program dependencies using the SDG. Simple approaches to remove non-serialisability without taking a SDG into account should be avoided because they can occur very high performance penalties.
2) One should avoid modifying such vulnerable edges which will make a read-only transaction an updater.
3) In general, if one is interested in high performance for a specific transaction type, then this should not be changed neither using materialisation or promotion.
4) Promotion is faster than materialisation in PostgreSQL, and vice-versa on the commercial system.

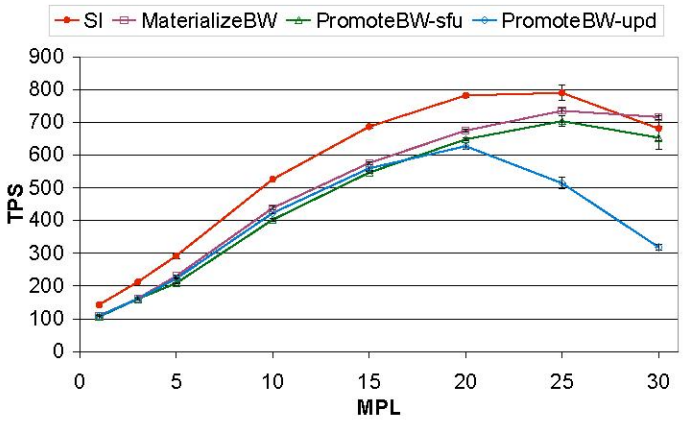


(a) Throughput over MPL (Commercial Platform).

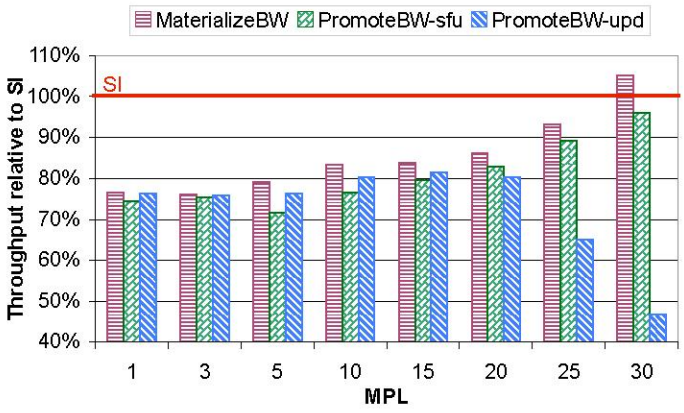


(b) Throughput relative to SI (Commercial Platform).

Fig. 8. Eliminating vulnerability between WriteCheck and TransactSaving (Commercial Platform).



(a) Throughput over MPL (Commercial Platform).



(b) Throughput relative to SI (Commercial Platform).

Fig. 9. Eliminating vulnerability between Balance and WriteCheck (Commercial Platform).

V. Conclusions

We have studied the performance impact of modifying application code to ensure that all executions are serializable, when running on a platform with SI as the concurrency control mechanism (which therefore can suffer data corruption with some programs). We showed that, for a benchmark mix of programs, there are ways that ensure serializable executions with negligible reduction in throughput (as compared to SI). We also showed that there is a substantial performance penalty from some other ways which were previously suggested to ensure serializability. Especially, it really matters which vulnerable edges are removed, from any dangerous structure in the SDG. The highest performance costs are induced by the 'simple' approaches without SDG, e.g. by plainly materializing all anti-dependency conflicts, with up-to 60% lower throughput (high contention case). These results hold on both PostgreSQL and an alternative implementation of SI, found in a commercial platform, though the specific mechanisms with best performance differ between the platforms.

In future work, we intend to develop a performance model, that can predict the impact of different mechanisms; we especially hope for a tool that can suggest which vulnerable edges to deal with, for least impact on performance.

References

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, May 22-25, San Jose, California*. ACM Press, 1995, pp. 1–10.

[3] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 492–528, 2005.

[4] K. Jacobs, "Concurrency control: Transaction isolation and serializability in SQL92 and Oracle7," Oracle Corporation, Tech. Rep. A33745 (White Paper), 1995.

[5] P. Bernstein and N. Goodman, "Mulitversion concurrency control - theory and algorithms," *ACM Transactions on Database Systems*, vol. 8, no. 4, pp. 465–483, December 1983.

[6] Y. Raz, "Commitment ordering based distributed concurrency control for bridging single and multiple version resources," in *Proceedings of the International Workshop on Research Issues in Data Engineering (RIDE'93)*, 1993, pp. 189–199.

[7] A. Adya, B. Liskov, and P. E. O'Neil, "Generalized isolation level definitions," in *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000), 28 February - 3 March, San Diego, California, USA*. IEEE Computer Society, 2000, pp. 67–78.

[8] S. Wu and B. Kemme, "Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation," in *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005), 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 2005, pp. 422–433.

[9] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 14-16, Baltimore, USA*. ACM Press, 2005, pp. 419–430.

[10] C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," in *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference (Middleware'04), October 18-20, Toronto, Canada*. Springer-Verlag, 2004, pp. 155–174.

[11] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database replication using generalized snapshot isolation," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005), 26-28 October, Orlando, FL, USA*, 2005, pp. 73–84.

[12] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB2006), Seoul, Korea, September 12-15*, 2006, pp. 715–726.

[13] "TPC-C benchmark," URL: http://www.tpc.org/tpcc/, Transaction Processing Performance Council, 2006.

[14] A. Fekete, "Serializability and snapshot isolation," in *Proceedings of the 10th Australasian Database Conference (ADC '99), Auckland, New Zealand*, 1999, pp. 201–210.

[15] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the detection of snapshot isolation anomalies," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB2007), Vienna, Austria, September 23-27*, 2007, pp. 1263–1274.

[16] A. Fekete, "Allocating isolation levels to transactions," in *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), May 22-25, San Jose, California*. ACM Press, 2005, pp. 206–215.

[17] A. J. Bernstein, P. M. Lewis, and S. Lu, "Semantic conditions for correctness at different isolation levels." in *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000), 28 February - 3 March, San Diego, California, USA*. IEEE Computer Society, 2000, pp. 57–66.

[18] M. Alomari, M. Cahill, A. Fekete, and U. Röhm, "Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels?" in *Proceedings of the 13th International Conference on Database Systems for Advanced Applications (DASFAA'08), New Delhi, India*, 2008, p. to appear.

[19] A. Fekete, E. O'Neil, and P. O'Neil, "A read-only transaction anomaly under snapshot isolation," *SIGMOD Record*, vol. 33, no. 3, pp. 12–14, 2004.