Automating the Detection of Snapshot Isolation Anomalies.

Conference Paper · January 2007

Source: DBLP

Krithivasan Ramamritham
Indian Institute of Technology Bombay
588 PUBLICATIONS **17,324** CITATIONS

SEE PROFILE

S. Sudarshan
Indian Institute of Technology Bombay
161 PUBLICATIONS **6,920** CITATIONS

SEE PROFILE

Automating the Detection of Snapshot Isolation Anomalies

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Sudhir Jorwekar
Roll No: 04305902

under the guidance of

Prof. S. Sudarshan
Prof. Krithi Ramamritham

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
2007

Abstract

Snapshot isolation (SI) provides significantly improved concurrency over 2PL, allowing reads to be non-blocking. Unfortunately, it can also lead to non-serializable executions in general. Despite this, it is widely used, supported in many commercial databases, and is in fact the highest available level of consistency in Oracle and PostgreSQL. Sufficient conditions for detecting whether SI anomalies could occur in a given set of transactions were presented recently, and extended to necessary conditions for transactions without predicate reads.

In this thesis we address several issues in extending the earlier theory to practical detection/correction of anomalies. We first show how to mechanically find a set of programs which is large enough so that we ensure that all executions will be free of SI anomalies, by modifying these programs appropriately. We then address the problem of false positives, i.e., transaction programs wrongly identified as possibly leading to anomalies, and present techniques that can significantly reduce such false positives. Unlike earlier work, our techniques are designed to be automated, rather than manually carried out. We describe a tool which we are developing to carry out this task. The tool operates on descriptions of the programs either taken from the application code itself, or taken from SQL query traces. It can be used with any database system. We have used our tool on two real world applications in production use at IIT Bombay, and detected several anomalies, some of which have caused real world problems. We believe such a tool will be invaluable for ensuring safe execution of the large number of applications which are already running under SI.

Acknowledgments

I would like to thank my advisors **Prof. S. Sudarshan** and **Prof. Krithi Ramamritham** for the guidance and consistent directions they have fed into my work and also for the constant encouragement and motivation.

A special thanks to **Prof. Alan Fekete** for introducing me to this area and for the continuous guidance all along.

I take this apportunity to thank my colleagues in Application Software Cell (ASC) for the friendly company and for the resources they provided. I would like to thank our In-Charge **Prof. S. Sudarshan** and Project Manager **Mrs. Asmita Shetye** for balancing my workload in ASC.

Finally, I would like to thank my family members and friends for their invaluable support.

- Sudhir Jorwekar

Contents

List of Figures

List of Tables

List of Publications

Automating the Detection of Snapshot Isolation Anomalies by S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan in *Proceedings of International Conference on Very Large Data Bases (VLDB'07)*, Vienna, Austria, September 2007, to appear.

Chapter 1

Introduction

Databases provide different isolation levels to meet different concurrency and consistency requirements. The highest isolation level (serializable) ensures the highest level of consistency i.e., serializability. However, lower isolation levels provide significantly better concurrency, and are widely used, even though they can lead to a reduced level of consistency.

Snapshot isolation (SI) is an attractive optimistic concurrency control protocol, which is widely implemented and widely used. Among its attractive features are that under SI, reads are always non-blocking, and it avoids some of the common types of anomalies. However, as pointed out in [2], SI is vulnerable to an anomaly called write-skew, as well as a related situation involving phantoms. Thus, transactions running under snapshot isolation can lead to non-serializable schedules (anomalies) and can cause database inconsistency.

Despite this, not only is SI widely supported, it is also the highest level of consistency supported by widely used systems such as Oracle and PostgreSQL, which in fact use SI even if the user requests serializable level of isolation. Many organizations use these databases for running their applications, and so they are potentially at risk of corrupted data.

It was observed in [6] that transactions in many applications, such as those in the TPC-C benchmark [15], have certain properties that ensure serializable executions even if run under SI. In fact in many applications, SI anomalies either do not occur because data items that are read are also updated, or the SI anomaly results in violation of an integrity constraint, such as primary key, and so rollback eliminates the anomaly. As a result, consistency problems are not widely seen.

However, it is not a wise idea to assume that an application is safe just because consistency problems have not been observed, since isolation problems are often hard to detect and to reproduce. In one of the applications in use at IIT Bombay, for example, financial auditors twice found problems with accounts which could be traced back to problems with SI in one of the cases, and to bad transaction boundaries in the other. Even after such problems are found, it is non-trivial to find their cause, since race conditions are extremely hard to reproduce.

Fekete et al. [6] and Fekete [5], provide a theory for determining which transactions among a given set of transactions can be run under SI, and which must be run under 2PL, to ensure serializable executions. The theory is based on analyzing read-write and write-write conflicts between transactions, and identifying a set of "pivot" transactions; if the pivot transactions are run under 2PL (or by using other techniques described in Section 2), all executions are guaranteed to be serializable.

The theory of [5] assumes that transactions work on prespecified data items, whereas real world transactions are programmed as SQL statements containing predicates. Moreover, analysis must be done at an abstract level where the SQL statements are parametrized, and actual parameter values are not available to the analysis tool. Fekete et al. [6] do consider parametrized SQL statements with predicates, but their analysis is applied manually to prove that transactions in the TPC-C benchmark could all be run under SI, while guaranteeing serializability. They do not address the automation of such analysis.

Our Contributions: In this thesis we describe the architecture of a tool we have developed to analyze application transactions and automatically detect SI anomalies. In the process we also address several issues in extending the earlier theory (described in Section 2) to practical detection/correction of SI anomalies.

1. We describe (in Section 3) a syntactic analysis technique, which combines the pivot detection technique of [5] with the column-based analysis of [6] to flag a set of transactions as those that could cause anomalies if run under SI.

2. The syntactic analysis technique is conservative, and may over-approximate the set of transactions that could cause anomalies if run under SI; in other words, the technique may result in false positives. We therefore present (in Section 4) three sufficient conditions to determine whether transactions are safe (i.e. cannot cause anomalies). We make use of information about updates/deletes present in the transaction, or database integrity constraints, either of which can sometimes ensure that the transactions cannot run concurrently with certain other transactions. Using these techniques, we are able to significantly reduce the incidence of false positives.

   We also note that the above analysis takes into account the presence of artificially introduced conflicts or *select for update* statements, which are used to avoid some anomalies in SI. Thus, if an application is analyzed, found to contain anomalies, and the anomaly is (manually) fixed by introducing conflicts or by using *select for update* statements, our analysis can be run on the modified program to detect if it is indeed safe.

3. We develop a tool to automate the analysis of applications. The tool (described in Section 5), can automatically analyze a given set of transactions and report all transactions that could result in SI anomalies, and can be used with any database system.

   A challenge in building such a tool lies in identifying the set of transactions in a given application. In our analysis, as in [6], we assume that each transaction is a sequence of (parametrized) SQL statements, including queries, updates, inserts and deletes. Transactions are usually coded in the form of programs with control structures like loops and conditional branches, but as described in [6], such transactions can be split into multiple straight-line transactions, one representing every possible execution. (Since our analysis deals with parametrized SQL statements, the number of iterations of a loop is not relevant, so the above set is finite.)

   It is difficult to automatically analyze transaction code containing control flow statements and generate all possible transactions, as described above. This step can be done manually, and the resultant transactions provided to our tool. However, our tool supports an alternative mode, where execution traces (i.e., sequences of SQL statements generated by the application) are collected, and the tool processes these to extract parametrized transactions. While it is hard to ensure that all possible transactions are captured, this approach can be used as a way to test, if not verify, the safety of an application.

We have used our tool on two real world applications in use at IIT Bombay, and detected several transaction programs that could result in executions with anomalies; these could cause (and some have caused) real world problems. We believe such a tool will be invaluable for ensuring safe execution of the large number of applications which are already running under SI. We present a summary of results obtained using our tool; our tool was able to detect several instances where anomalies are possible in these applications. The techniques from Section 4 were successful in eliminating several false positives. We also ran our tool on the TPC-C benchmark, which was shown to be free of anomalies in [6] using manual analysis. Our tool was able to come to the same conclusion, automatically.

4. We discuss practical issues in ensuring serializability (in Section 6). We show that there may be multiple ways of choosing which transactions to modify, to ensure serializability. We discuss the problem of finding a minimum-sized set of such transactions to minimize the efforts required. We also discuss a generic approach for *materializing conflicts* technique. See Section 6 for details. We provide an alternative to the table locks in order to ensure freedom from phantoms based on key value locking and materializing conflict approach.

The rest of the thesis is organized as follows. Chapter 2 provides a background of snapshot isolation testing including types of anomalies, and a theory for detecting anomalies and techniques for avoiding anomalies. Chapter 3 describes the syntactic analysis technique. Chapter 4 addresses the problem of reducing false positives. Chapter 5 presents an overview of various steps involved in snapshot isolation testing tool. It explains the basic approach for getting a set of transaction programs, and performing a conservative analysis. Chapter 6 discusses issues in using various methods to remove anomalies. Chapter 7 contains discussion related to miscellaneous issues in developing an SI testing tool.

Chapter
2

Background

In this section we briefly recap the existing knowledge about snapshot isolation anomalies. To illustrate the various concepts, we use a simplified banking application as the running example in this thesis. The schema for this application is shown in Figure 2.1 while Figure 2.2 shows the different kinds of transactions supported by this application.

Relation *account* contains information about each account number, available balance and account type (checking / savings). Relation *owner* contains mapping between account customer and account number (accno). Relation *txn* stores the information about each *withdrawal/deposit* transaction.

2.1 Snapshot Isolation (SI)

Snapshot Isolation is an extension of multiversion concurrency control. It was first defined in [2] as follows:

Definition 2.1 Snapshot Isolation. *A transaction T1 executing with Snapshot Isolation always reads data from a snapshot of committed data valid as of the (logical) time T1 started, called the start-timestamp. (The snapshot could be any point in logical time before the transactions first read.) Updates of other transactions active after T1 started are not visible to T1. When T1 is ready to commit, it is assigned a commit-timestamp and allowed to commit if no other concurrent transaction T2 (i.e., one whose active period [start-timestamp, commit-timestamp]*

account(accno, balance, acctype)
customer(id, name, address);
owner(id, accno)
txn(txnid, txntype, accno, id, amount, timestamp)
batchaudit(bid, starttimestamp, endtimestamp, inamount, outamount)

Figure 2.1: Schema for mini banking system

Creation of new account: A new account is created for a customer using this transaction. If customer is opening his first account in bank, his personal information is also saved in *customer* relation. A new customer id (resp. a new account number) is generated by finding the maximum value of the column id in the *customer* relation (resp. the column accno in the *account* relation), and adding one.

Update of contact information of customer: Personal information of a customer is updated using this transaction.

Deposit: The specified amount is deposited in the specified account number by updating the account balance. The transaction is recorded in the relation *txn*.

Withdrawal: A withdrawal is permitted from an account provided the sum of balances of all accounts belonging to the user remains non-negative. Overdraft beyond this is allowed, but with a penalty of 1 unit. The transaction updates the account balance and records the transaction in the relation *txn*.

End-of-the-day audit: At the end of each day an audit batch is formed. These batches are stored in relation *batchaudit*. These batches define non overlapping intervals $[\mathit{starttimestamp}, \mathit{endtimestamp})$. Each batch identifies all transactions listed in *txn* that have $\mathit{starttimestamp} \leq \mathit{timestamp} < \mathit{endtimestamp}$. The sums of the amounts deposited and withdrawn by transactions in the batch are calculated and stored, for each batch.

Figure 2.2: Transaction Types in the Simplified Banking Application

overlaps with that of T1) has already written data that T1 intends to write; this is called the First-committer-wins rule to prevent lost updates. □

In fact, most implementations use exclusive locks on modified rows, so that instead of First-Committer-wins, a First-Updater-wins policy is applied. The distinction is not important, the key impact of either policy is that one can't have two transactions that are concurrent (overlapping in time) and modify the same data item. This means that the "lost-update" anomaly can't occur. Similarly, the way all reads by a transaction T see the same set of complete transactions (those that committed before T started), and they see no effects of incomplete transactions, means that SI prevents the "inconsistent read" anomaly.

2.2 Snapshot Isolation Anomalies

Even though SI avoids all the classically known anomalies such as lost update or inconsistent read, there are some non-serializable executions that can occur. Two types of anomalies have been identified in a set of transactions running using SI [2]. Write skew is a very common anomaly, illustrated by the example below, which is not detected by the *first-committer-wins* policy of SI.

Example 2.1 ***Write skew.*** *A person P owns two bank accounts. Let X and Y be the balance in these accounts. This bank provides a facility where M units can be withdrawn from any of the two accounts as long as $X + Y \geq M$. Let $X = 100$ and $Y = 0$. Consider a scenario where P initiates two withdrawal transactions (T_1, T_2) on the different accounts simultaneously, both trying to withdraw 100 units.*

$$\begin{array}{ll} T_1: & r_1(X,100) \;\; r_1(Y,0) \;\; w_1(X,0) \;\; c_1 \\ T_2: & r_2(X,100) \;\; r_2(Y,0) \;\; w_2(Y,-100) \;\; c_2 \end{array}$$

Snapshot Isolation allows both these withdrawals to commit. Thus P is able to withdraw 200 *units and the final sum of balance is* -100*. This is not possible in any serial execution of the two transactions.* □

Write skew can also happen between reads and inserts, as illustrated later in Example 4.4.

Fekete et al. [7] also describe another kind of anomaly that they call a read-only-transaction anomaly. The anomaly is illustrated by following example.

Example 2.2 ***Read Only Transaction Anomaly in Banking System****.* *Two persons (P and Q) share a checking and a savings bank account. Let* X = *checking account balance and* Y= *savings account balance. Withdrawal is covered (without penalty) as long as* $X + Y > 0$*. Penalty charge of 1 unit is applied, if* $X + Y < 0$*. Initially* $X_0 = 0$ *and* $Y_0 = 0$*. P is is trying to withdraw 10 units from the checking account X in* T_2*. After realizing that balance is low, P informs Q to deposit money in savings account and waits. In transaction* T_1*, Q deposits 20 to the savings account Y and using transaction* T_3*, gets the receipt stating current sum of account balance* $(X + Y)$*. Q now asks P to proceed with the transaction. But P is still penalized!*

$$\begin{array}{ll} T_1: & r_1(Y)\ \ w_1(Y)\ \ c_1 \\ T_2: & r_2(X)\ \ r_2(Y)\ \ w_2(X)\ \ c_2 \\ T_3: & r_3(X)\ \ r_3(Y)\ \ c_3 \end{array}$$

Following is the transaction history of above scenario.

$$\begin{array}{lll} H3 & : & r_2(X_0,0)\ \ r_2(Y_0,0)\ \ r_1(Y_0,0)\ \ w_1(Y_1,20)\ \ c_1 \\ & & r_3(X_0,0)\ \ r_3(Y_1,20)\ \ c_3\ \ w_2(X_2,-11)\ \ c_2 \end{array}$$

i.e. Readonly transaction T_3 *prints out* $X = 0$ *and* $Y = 20$*, while final values are* $Y = 20$ *and* $X = -11$*. This can not happen in any serializable execution.* □

2.3 Theory of Anomaly Detection

The starting point for understanding how transactions can produce anomalies under SI is the theory of multiversion serializability. There are several variants of this theory ([1, 3, 10]), but all define a serialization graph for a given execution, with nodes for the transactions, and edges reflecting conflicts or dependencies. For example, there can be an edge from T_i to T_j if T_i reads a version of an item and T_j produces a later version of the same item; this is called a rw-edge. The key theorem is that when the serialization graph is acyclic, then the execution is serializable, and no anomalies can occur.

For the DBA or application developer, the real concern is not whether a given execution is serializable, but rather whether every possible execution is serializable (that is, whether or not anomalies are possible). There are two different sources for the variation between different executions of a single system. The system is made of programs, each of which can execute in different ways, depending on inputs (such as different parameter values) or by making control flow decisions based on the values read in earlier database accesses. Also, there can be many executions as the transactions interleave in different orders, based on non-deterministic outcomes of process scheduling and lock contention. Earlier work [6] has established some important conditions on a set of applications, that guarantee that every execution of these applications is serializable when SI is the concurrency control mechanism.

Static Dependancy Graph: The main intellectual tool in [6] is a graph called the *static dependency graph* (SDG). This can be drawn for a given collection of application programs A. The nodes of SDG(A) are the programs in the collection. An edge is drawn from P_1 to P_2 if there is some execution of the system, in which T_1 is a transaction that arises from running program P_1, and T_2 arises from running P_2, and there is a dependency from T_1 to T_2. Of special importance are edges called *vulnerable edges.* An edge from P_1 to P_2 is vulnerable if there is some execution of the system, in which T_1 is a transaction that arises from running program P_1, and T_2 arises from running P_2, and there is a read-write dependency from T_1 to T_2, and T_1 and T_2 are concurrent (that is, they overlap in execution time). In diagrams, vulnerable edges are shown specially (as dashed arrows).
Within the SDG, certain patterns of edges are crucial in determining whether or not anomalies might occur. [6] defines a *dangerous structure* in the graph SDG(A) to be where there are programs P, Q and R, which may not all be distinct, such that there is a vulnerable edge from R to P, there is a vulnerable edge from P to Q, and there is a path from Q to R (or else $Q = R$). We can call a program P with the properties above a *pivot* program in the collection A. Note that the definition includes the possibility of a pivot with a vulnerable self-loop (then $P = Q = R$), and also of a pair of pivots with vulnerable edges in each direction between them. The main theorem of [6] shows that if the collection of programs A has SDG(A) without pivots, then every execution of the programs in A, all running on a DBMS with SI as concurrency control mechanism, is serializable.

2.4 Conflicts Between Sql Statements

Transactions are specified using SQL statements. Fekete et al.[6] used a formalism suitable for set oriented operations in SQL. According to this formalism, a transaction can only perform *predicate reads, item reads and item writes.*

2.4.1 Predicate Read

A *predicate read* is an operation, which identifies a set of data items in the database which satisfy the given predicate, based on the state of the database. SQL use predicates to read or modify rows. It is expected that the predicate read will be followed by item read or item write operations on the identified set of data items. The concept of *predicate write* is not used for identifying conflicts because it is unrealistic in practice [4].

Example 2.3 ***Predicate Read.***

S: `SELECT * FROM tab WHERE` $col_1 = 1$;
U: `UPDATE tab SET` $col_1 = 1$ `WHERE` $col_2 = 1$;

tab		
rid	col_1	col_2
1	*1*	*0*
2	*0*	*1*

Statement S on given database state will be represented as given below.

S_{pr} : *PredicateRead*(`FROM tab WHERE` $col_1 = 1$)
S_{ir} : *ItemRead*($rid = 1, col_1 = 1, col_2 = 0$)

Statement U on given database state will be represented as given below.

$$\begin{aligned} U_{pr} &: PredicateRead(\texttt{FROM tab WHERE } col_2 = 1) \\ U_{iw} &: ItemWrite(rid = 2, col_1 = 1, col_2 = 1) \end{aligned}$$

□

2.4.2 Conflict between Predicate Read and Item Write

A predicate read conflicts with an item write, if item write can change the data-item set returned by predicate read.

Example 2.4 . *In example 2.3, operation U_{iw} : $ItemWrite(rid = 2, col_1 = 1, col_2 = 1)$ conflicts with predicate read operation S_{Pr} : $PredicateRead(\texttt{FROM tab WHERE } col_1 = 1)$, as*

1. *$rid = 2$ was not satisfying S_{pr} before executing U_{iw}.*
2. *due to values written by U_{iw}, $rid = 2$ satisfies predicate read of S_{pr}.*

□

2.4.3 Lack of Predicate Locking

In most commercial databases which provide snapshot isolation, predicate locks are not used while locking. In order to identify rows to be locked for a statement with predicate P in a transaction T, they first find out the rows that satisfy given predicate using sequential/index scan in current snapshot without any read lock and then obtain exclusive locks on the qualified rows. There are two possible cases of phantom situation [4],

1. values of attributes involved in predicate might be changed by concurrent transactions
2. new rows inserted would be missed by this locking protocol as they are not part of the database snapshot used by T

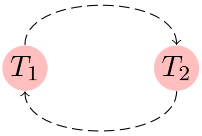
Such phantom phenomena can result into non serializable schedules i.e. *phantom anomalies.* Let T' be a concurrent transaction that operates on some of the tables used in predicate P but on different rows than T has locked. If T' can alter attributes involved in predicate P then we have *vulnerable* edge $T \xrightarrow{vul} T'$.

Example 2.5 . *Consider transactions T_1 and T_2.*

$$\begin{aligned} T_1 &: \quad \texttt{UPDATE tab SET } col_1\texttt{=}col_1\texttt{+1 WHERE } col_2\texttt{=1;} \\ T_2 &: \quad \texttt{UPDATE tab SET } col_2\texttt{=}col_2\texttt{+1 WHERE } col_1\texttt{=1;} \end{aligned}$$

It can be observed that these transactions have a read-write predicate conflict, since the updates done by one affects the set of rows satisfying the predicate of the other transaction. However, under SI these two can run concurrently resulting in a phantom.
Both these transactions operate concurrently on following database state.

tab		
rid	*col*$_1$	*col*$_2$
1	*1*	*0*
2	*0*	*1*



Following diagram shows results of different execution order of T_1 and T_2 on a particular initial state I of table tab. $T_1 \gg T_2$ indicates serial execution $T_2(T_1(I))$. We can see that state produced by concurrent execution under SI does not correspond to either of the serial executions.

tab		
rid	col_1	col_2
1	*1*	*0*
2	*0*	*1*

Initial State

tab		
rid	col_1	col_2
1	*1*	*1*
2	*1*	*1*

T_1 *& T_2 concurrent*

tab		
rid	col_1	col_2
1	*2*	*1*
2	*1*	*1*

$T_1 \gg T_2$

tab		
rid	col_1	col_2
1	*1*	*1*
2	*1*	*2*

$T_2 \gg T_1$

□

2.5 Removing Anomalies

Of course, the application developer hopes that analysis using SDG will show that the set of programs in the system will generate executions which are all serializable. If there are no pivots, this is true. But what if there are some pivots? The general approach is to modify some of the application programs, in ways that do not alter their business logic, but lead to a new set of programs that has no pivots. In general, one can make changes to eliminate pivots, by changing at least one vulnerable edge to be non-vulnerable, in every dangerous structure. Here we summarize the main known ways (from [5, 6]) to do this modification; note that most of these involve modifying the code of the pivot programs. Not all of these may be applicable to a given program or for given DBMS. Also, they have different impacts on concurrency.

2.5.1 Strict Two Phase Locking (S2PL) for Pivots

The cleanest modification is to run the pivot programs with true serializability (using strict-two-phase locking), rather than using SI. This does not require any change in the code of the program except for configuration information for the session. As shown in [5], the above modification ensures serializability as long as the following properties hold for the concurrency control mechanism: (a) The version of an item x produced by an SI transaction T must be protected by an exclusive lock from the time it leaves any private universe of T, until (and including the instant when) the version is installed because T commits. The exclusive lock must be obtained following the normal locking rules. (b) The check against overwriting a version which was installed while an SI transaction T was active covers versions produced by locking transactions as well as versions produced by SI transactions. These conditions are in fact met by all implementations of SI whose details we are aware of, including Oracle, PostgreSQL and Microsoft SQL Server.

Unfortunately, among the widely used database systems as far as we are aware, only Microsoft SQL Server 2005 and MySQL with the InnoDB storage manager support both SI and 2PL as concurrency control choices. On other platforms such as Oracle and PostgreSQL, asking for a transaction to run with “Isolation Level Serializable” actually leads to it running using SI. Thus this approach to preventing anomalies is often hard to utilize. We can work around this problem by simulating 2PL in the application and explicitly obtaining table locks, as discussed further in Section 6. However, this approach has a significant impact on performance.

2.5.2 Materializing Conflicts

Programmers can explicitly introduce extra conflicts in transactions, in order to prevent the transactions from running concurrently. Typically, one introduces a new table, and both the transactions are made to write the same row of this table (for a given vulnerable edge). This will mean that First-committer-wins is invoked, and the transactions won't run concurrently; thus the edge becomes non-vulnerable. In many cases, the new data item can be seen as a materialization of some integrity constraint which is violated in non-serializable executions.

2.5.3 Promotion

There is sometimes another approach to modify programs, in order to remove the vulnerable nature from an edge in a dangerous structure. In this approach we change the program at the tail end of the edge (the one with the read in the read-write conflict) so the transaction writes the data item involved, or is treated as if it writes the item. We say that the read is *promoted* to a write.

Suppose that the read of interest is a statement "*select* T.c from T where ....". Promotion can be done by introducing a statement that updates the item from its current value to the same value ("*update* T set T.c=T.c where ..."). In some platforms such as Oracle, a similar effect is obtained by replacing the *select* by *select for update* (we abbreviate this as SFU) [1]; the implementation does not do a write, but anyway it treats all the items read just like writes when checking for *first-committer-wins* strategy. In either of these modifications, the transaction will not be able to run concurrently with the other transaction on the (formerly) vulnerable edge, which is writing the same item.

Note that promotion might apply to the pivot P (on the item which is read in P and produces a vulnerable edge leaving the P), or it could apply to P's predecessor in the vulnerable structure, by promoting the read in the predecessor which conflicts with a write in the pivot program P.

This technique is not usable if the conflict involves a read which is part of evaluating a predicate (deciding which rows satisfy a WHERE clause) rather than simply obtaining a value (in the SELECT clause); such a transaction would be vulnerable to the phantom problem [4, 6] even if promotion is used.

[1] SFU in PostgreSQL holds exclusive locks till commit, but does not prevent a conflicting concurrent transaction from committing subsequently. Thus, PostgreSQL's SFU does not promote the reads to writes. i.e., a transaction T_1 does not rollback if the rows updated by it were concurrently selected by another transaction T_2 using SFU but not modified by T_2. Hence, actual updates must be done to promote read[16].

Chapter 3

Syntactic Analysis

The starting point for the tool we have built is a syntactic analysis based on the names of the columns accessed in the SQL statements that occur within the transaction. This is similar to (but less subtle than) the style of argument used in [6], when the TPC-C programs were manually analyzed for dependencies.

In our column-name based syntactic analysis, our starting point is a set of *transaction programs*, each consisting of a set of SQL statements. Every execution of the transaction program is assumed to execute each of these SQL statements. These can be found in two ways: by extraction transaction programs from the source code of the application programs, or by logging the SQL statements submitted to the database. Each of these has some complex issues that need to be resolved. For example, if an application has control flow, different executions of an application program may execute different sets of SQL statements. As in [6], such application level transactions can be "split" into multiple straight-line transactions. We discuss these issues further in Section 5.1.

However it is done, we assume that we have the complete set of transaction programs for our application. In the syntactic analysis based on column names, we define, for each transaction program, a readset and a writeset. Each of these is a set of tablename.column entries, determined by seeing which names appear in the SQL statements. Note that because SQL allows the tablename to be omitted if it is deducible from context, our tool must first fill in any missing table names. For example, in Create new transaction, the SELECT clause "select max(accno+1) as m from account" is rewritten to "select max(account.accno+1) as m from account", and the readset will contain the entry account.accno.

Example 3.1 ***Determining Transaction Programs.*** *Consider transaction programs for the mini banking application mentioned in Figure 2.2. The Update customer information transaction does not contain any control structure. Hence, we get only one transaction program UCI from it. Similarly, Deposit transaction and End of the day audit transaction are covered with one transaction program each, namely DEP and EOD. The transaction for creation of new accounts has two possible execution paths depending on whether the customer is already recorded in the customer table or not. Let CAc_1 be the program where the customer is already recorded, and CAc_2 be the one where the customer is not already recorded, and a new customer record has* *to be created. Similarly, the withdrawal transaction is covered with two transaction programs, one for the case where the resultant balance is non-negative, and the other for the case where the balance is negative, requiring an overdraft penalty to be deducted from the account balance. Let* ShW_1 *and* ShW_2 *denote the respective transactions programs. Thus, we have seven transaction programs*

$$\{UCI, DEP, EOD, CAc_1, CAc_2, ShW_1, ShW_2\}$$

in our simplified banking system. □

Assumption: For the remainder of this thesis, we assume for simplicity that in any SQL SELECT statement, no table is named in the FROM clause unless some column of that table is mentioned in the SELECT clause or the WHERE clause or both. The results in this thesis can however, be easily extended to remove this restriction.

Definition 3.1 Syntactic read and write sets. *The readset for syntactic column-name analysis consists of every tablename.column that appears in the SELECT clause, or in the WHERE clause, or on the righthandside of the equals in the UPDATE clause. The writeset consists of every tablename.column that appears on the lefthandside of the equals in the UPDATE clause, and also every column in any table that is mentioned in an insert or delete statement. We denote the syntactic readset of transaction program P as rset(P), and the syntactic writeset as wset(P).* □

Example 3.2 ***Read write sets based on table.column as data items.*** *The rset and wset for example statements S and U are calculated as shown below.*

S	:	*select balance from account where accno=100*
U	:	*update customer set name='xyz' where id=103*
I	:	*insert into customer values(102,' xyz',' PQR')*
D	:	*delete from account where accno=104*

	rset	*wset*
S	*account.accno, account.balance*	$\emptyset$
U	*customer.id*	*customer.name*
I	ϕ	*customer.**
D	*account.accno*	*account.**

□

There is a relationship between these definitions and the actual read and write sets of the generated transaction instances, where individual column values are treated as data items (that is, for a generated transaction instance, we regard the value of one attribute in a single row as an item). Our calculated syntactic sets are both upper bounds for the true sets. That is, the individual database fields that are read all lie in the columns named in the syntactic readset, and similarly all the fields written lie in the columns named in the syntactic writeset. Furthermore, the predicate in any predicate read operation in a generated transaction instance is computed only on columns that are part of the syntactic read set.

From these, we define a graph called the column-based syntactic dependency graph (CSDG), as follows.

Definition 3.2 Column-based Syntactic Dependency Graph. *The nodes of the CSDG consist of the transaction programs that make up the applications. Given two programs* P_j *and*

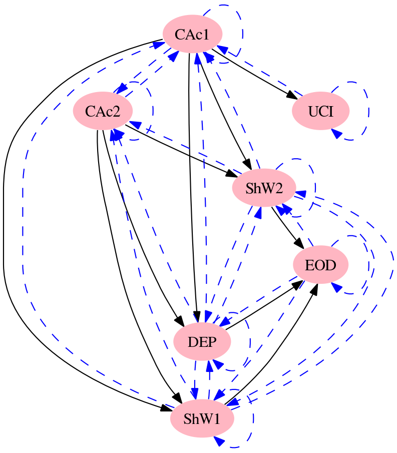


Figure 3.1: CSDG for Mini Banking Application. Dashed edges denote *vulnerable* edges, and solid edges denote *non-vulnerable* edges. Shaded nodes are Syntactic Pseudopivots.

P_k, there is an edge $P_j \rightarrow P_k$ whenever

$$\begin{aligned}(rset(P_j) \cap wset(P_k) &\neq \emptyset) \vee \\ (wset(P_j) \cap rset(P_k) &\neq \emptyset) \vee \\ (wset(P_j) \cap wset(P_k) &\neq \emptyset)\end{aligned}$$

This edge is marked as pseudovulnerable (written as $P_j \xrightarrow{vul} P_k$, and shown as a dashed arrow in diagrams) when

$$rset(P_j) \cap wset(P_k) \neq \emptyset$$

□

Based on CSDG, we identify certain transaction programs as syntactic pseudopivots.

Definition 3.3 Syntactic Pseudopivot. *A transaction program P_B is a syntactic pseudopivot if there exist transaction programs P_A and P_C (which may be the same), such that $P_A \xrightarrow{vul} P_B \xrightarrow{vul} P_C$ is a subpath in **some cycle** of edges in CSDG.* □

Example 3.3 ***CSDG and Syntactic Pseudopivots for Mini banking application.*** *If we find the syntactic read, write sets and create the CSDG for transactions in mini banking system, we get the graph shown in Figure 3.1.* □

This analysis is safe, that is, there are no false negatives (where a potential anomaly is not identified). CSDG has an edge whenever the true static dependency graph has an edge, and the edge in CSDG is pseudovulnerable whenever the corresponding edge in SDG is vulnerable. This means that any pivot is a syntactic pseudopivot and we have a theorem (which follows immediately from Theorem 3.1 in [6]).

Theorem 1 Syntactic column-based analysis is safe. *If a set of transaction programs contains no syntactic pseudopivots, then every execution under SI will in fact be serializable.* □

One might imagine relying on the first committer wins property of SI, and propose a stricter definition of pseudovulnerable edges that has the same form as the definition of exposed edge in [5]. That is, one could consider the alternative definition where the edge P_j to P_k is not labeled as vulnerable unless $(rset(P_j) \cap wset(P_k) \neq \emptyset) \wedge (wset(P_j) \cap wset(P_k) = \emptyset)$. This alternative definition would not be safe, because the syntactic writeset can be an overapproximation of the true write set. That is, there are cases where some generated instances are allowed to execute concurrently because they are not writing to any common data item (even though their syntactic write sets do overlap).

Chapter 4

Eliminating False Positives

A false positive is erroneous identification of a threat or dangerous condition that turns out to be harmless. In this thesis, by false positive we refer to a transaction which is falsely detected as potentially contributing to an anomaly, for example, a transaction which is a syntactic pseudopivot but not in fact a pivot.

The analysis done with the syntactic column-name analysis, and expressed in CSDG, is safe. It never misses noticing the possibility of anomalies (non-serializable executions). However, it is so conservative that it identifies many false positives: in our experience with some real-world application mixes, almost every transaction program is a syntactic pseudopivot. In this section we identify some situations where the syntactic analysis is unnecessarily conservative, so one can prove that certain transaction programs which are syntactic pseudopivots are not in fact pivots, based on properties of the columns and of the programs.

4.1 Modification Protected Readset

The Oracle and PostgreSQL implementations of Snapshot Isolation treat a tuple as the lowest level data item; that is, write sets identify rows, rather than specific columns of rows, and the first-committer-wins rule forces that two transaction are not concurrent if they both commit updates on any columns (not necessarily the same columns) of some row. This will give us a valuable technique to argue that certain pseudovulnerable edges are not vulnerable, and this will sometimes show that some pseudopivot is not a pivot. The essential property we need to look for, is where a transaction modifies the rows it selects (or at least, the rows involved in read-write dependencies). We have seen many cases of this in real application code, especially a common coding pattern is to select a row by primary key before updating or deleting that row. We will now build up to a fairly broad definition, that covers a significant number of false positives among the syntactic pseudopivots in the applications we have examined.

Definition 4.1 Stable Predicates. *A predicate C used in transaction program P_1 is stable w.r.t. transaction program P_2, iff for every possible schedule H containing execution instances of transaction program P_1 and P_2 as T_1 and T_2 respectively, the set of rows identified by C in*

T_1 does not depend on the serialization order of T_1 and T_2. □

Definition 4.2 ***Select*** **with modification protected** ***rset*****(*****MPR-Select*****).** *A select statement S (which could be in a sub-query) in transaction program P_1 is said to be MPR w.r.t. transaction program P_2, if* ***either***

$$rset(S) \cap wset(P_2) = \emptyset$$

or all *of following conditions are true*

- *The WHERE clause predicate C used in S is stable w.r.t. P_2.*
- *P_1 contains a statement M, such that*
  - *M is an update or delete statement* [1]
  - *The WHERE clause predicate D used by M to identify rows to be modified is such that $C \Rightarrow D$, and D must be stable w.r.t. P_2.*
  - *Whenever the program executes S, it either also executes M, or aborts.*

□

The above condition ensures that whatever rows are selected by S in P_1 either do not conflict with P_2 at all (i.e. P_2 does not update any columns read in S), or the rows are modified subsequently in P_1.

Definition 4.3 Transaction with modification protected ***rset*** **(MPR Transaction).** *A transaction program P_1 is said to be MPR w.r.t. transaction program P_2 if*

1. *every select query as well as every subquery of an insert, delete or update in P_1 is an MPR-Select w.r.t. P_2.*
2. *WHERE clause predicates of every update/delete statement in P_1 are stable w.r.t. P_2.*

□

Theorem 2 *If transaction program P_1 is MPR w.r.t. transaction program P_2, and if the DBMS uses row-level granularity for the first-committer-wins checks, then in SDG, the edge from P_1 to P_2 can not be vulnerable.* □

We omit the proof details, but here is a sketch. Suppose P_1 is MPR *w.r.t.* P_2, T_1 arises from executing P_1, T_2 arises from P_2, and there is some read-write dependency from T_1 to T_2. The definition shows that T_2 cannot affect a predicate based on which T_1 select rows, so there is no predicate-read-to-write dependency. Thus the dependency must be data-item-read-to-write, but when T_1 reads a row (possibly selected using a predicate), and T_2 updates the row then T_1 and T_2 both modify that row, and so the two cannot run concurrently to commitment.

As we have mentioned, for DBMS's which apply *first-committer-wins* at row granularity, the MPR property implies that an edge in SDG is not vulnerable, even though the corresponding edge in CSDG might be pseudovulnerable. If enough edges are not actually vulnerable, a syntactic pseudopivot might not be a pivot at all, and therefore there is no danger of anomalies. Thus a tool that adopts the conservative approximation, and reports all syntactic pseudopivots, would be delivering a false positive.

[1] M may also be a SFU, on platforms where SFU is treated like a modification when it or other transactions do the first-committer-wins checks.

Definition 4.4 MPR Analysis. *We say that a transaction is found to be a false positive using MPR analysis if*

- *it is detected as a syntactic pseudopivot, and*
- *after eliminating vulnerable edges using Theorem 2, the transaction is found to not be a pivot.*

□

In order to build a tool that does not report many false positives, we want to automatically identify some cases where transactions are MPR *w.r.t.* others. This requires using syntactic sufficient conditions for the concepts defined above.

We wish to show that the set of rows returned by a WHERE clause are not affected by another program. The rows returned are filtered from the rows in (a cross product of) some tables, based on the value of a predicate. Thus we need to consider ways to show that the set of rows in the cross product doesn't change, and also ways to show that the value of the attributes used in the predicate doesn't change. This suggests the following definitions.

Definition 4.5 Insert-Delete Stable Table. *Table t is said to be insert-delete stable w.r.t. transaction program P, if P does not contain any insert or delete statement which operates on table t.* □

Definition 4.6 Syntactically Stable Column. *Column c of table t, denoted by $t.c$, is said to be syntactically stable w.r.t. transaction program P, if $t.c \notin wset(P)$.* □

Note that, if a *tablename.column t.c* is syntactically stable *w.r.t. P* then *t.c* is not affected by *insert*, *delete* or *update* statement in P. With the help of Definition 4.6 and Definition 4.5 we can conservatively identify if a predicate is stable *w.r.t.* some transaction program.

Definition 4.7 Syntactically Stable Predicate. *Consider a predicate C and a transaction program P. If every tablename.column used in C is stable w.r.t. P and every table on which C operates is insert-delete stable w.r.t. P, then C is syntactically stable w.r.t. transaction program P.* □

Definition 4.8 *Select* with syntactic modification protected *rset* (Syntactically MPR-Select). *A select statement S (which could be in a sub-query) in transaction program P_1 is said to be syntactically MPR w.r.t. transaction program P_2, if* ***either***

$$rset(S) \cap wset(P_2) = \emptyset$$

or all *of following conditions are true*

- *The WHERE clause predicate C used in S is syntactically stable w.r.t. P_2.*
- *P_1 contains a statement M, such that*
  - *M is an update or delete statement* [2]
  - *The WHERE clause predicate D used by M to identify rows to be modified is such that $C = (D$ and $D')$ for some D', and D must be syntactically stable w.r.t. P_2.*
  - *Whenever the program executes S, it either also executes M, or aborts.*

[2] M may also be a SFU, on platforms where SFU is treated like a modification when it or other transactions do the first-committer-wins checks.

```
begin;
select * from customer where id=:id;
update customer set name=?, address=? where id=:id;
commit;
```

```
rset={customer.id, customer.name, customer.address}
wset={customer.name, customer.address}
```

Figure 4.1: *Update Customer Information* Program

□

Notice that in the preceding definition, we use an easy syntactic test which ensures that $C \Rightarrow D$. One frequent case is where $C = D$ (so D' is true).

Definition 4.9 Transaction with syntactically modification protected *rset* (Syntactically MPR Transaction). *A transaction program P_1 is said to be syntactically MPR w.r.t. transaction program P_2 if*

1. *every select query as well as every subquery of an insert, delete or update is a syntactically MPR-Select w.r.t. P_2.*

2. *WHERE clause predicates of every update/delete statement in P_1 are syntactically stable w.r.t. P_2.*

□

The following theorem expresses that these syntactic judgments are safe.

Theorem 3 *If S is a select statement in transaction program P_1 such that S is syntactically MPR w.r.t. transaction program P_2, then S is MPR w.r.t. P_2.* □

We will now try to use the MPR analysis to detect some of the false positives in our simplified banking application.

Example 4.1 ***Update Customer transaction***. *Consider the update customer information transaction program UCI (Figure 4.1). Using the column-based syntactic rules mentioned in Definition 3.2, we get $UCI \xrightarrow{vul} UCI$. i.e., in CSDG (Figure 3.1), there is a pseudovulnerable self-loop from UCI to itself. Thus, UCI satisfies the definition of syntactic pseudopivot (Definition 3.3). It is easy to verify that transaction UCI is a syntactically MPR transaction program w.r.t. itself. Hence, by Theorem 2 and 3, the edge from UCI to itself in SDG is not vulnerable. Unless there are some other exposed edges involving UCI, we do not have UCI as a true pivot. i.e., this is an example of a false positive produced by over-approximating in the syntactic analysis.* □

Example 4.2 ***Deposit transaction***. *Consider the deposit transaction program DEP (Figure 4.2). DEP has a pseudovulnerable self-loop in CSDG (Figure 3.1). It reads the value of current_timestamp, but does not modify it. Also, it has an extra write operation which inserts a new row in relation txn. The update statement uses a predicate which is stable w.r.t. itself. As in Example 4.1, DEP is MPR w.r.t. itself, hence it is a false positive as long as it doesn't participate in other vulnerable edges.* □

```
begin;
select current_timestamp as c;
update account set balance=balance+m  where accno=:a and acctype in ('current','saving');
insert into txn values (:a||:c, :a, 'Deposit', :id, :m, :c);
commit;

rset={account.accno, account.acctype, account.balance}
wset={account.accno, account.balance, account.acctype, txn.*}
```

Figure 4.2: *Deposit* Program

```
begin;
select max(accno)+1 as m from account;
insert into account values (:m, 0, :type);
insert into owner values (:id, :m);
commit;

rset={account.accno}
wset={account.*, owner.*}
```

Figure 4.3: Create New Account Program

Example 4.3 ***Promotion and MPR****. Consider the shared withdrawal transaction program* $ShW1$ *which we used in Example 2.1.* $ShW1$ *is a syntactic pseudopivot due to vulnerable edge in a self-loop. If we used promotion on the select statements in* $ShW1$*, then according to Definition 4.3,* $ShW1$ *will become MPR w.r.t. itself. Thus, if we rerun the analysis after introducing promotion,* $ShW1$ *will be detected as false positive.* □

The above examples illustrate how our techniques not only help to find transactions that could not cause any anomalies but also to check that the programs are safe after they have been modified.

4.2 Integrity Constraints (ICs)

The database system ensures the preservation of some integrity constraints which are explicitly declared to the system in the schema definition, such as uniqueness of primary key and referential integrity. Some of the SI anomalies are avoided due to the dbms enforcement of these constraints.

Example 4.4 ***Primary key constraint avoids write skew****. Consider two instances* (T_1, T_2) *of the create account program (Figure 4.3) where new accounts are created for existing users. If* T_1 *and* T_2 *are executed concurrently, both transactions would try to create a new account with same account number. However, the account number is a primary key, and hence duplicates are not allowed. As a result, only one of the two transaction will be committed by the database. (In the absence of the primary key constraint, both transactions would be able to execute concurrently and commit, resulting in a non-serializable schedule.)*

Note that above transaction will be detected as syntactic pseudopivot and is a case of false positive. □

The pattern of *select max()+1 as m ... insert new tuple with value m*, illustrated in the above example, is commonly used for assigning a numeric primary key for new tuples.

```
begin;
select accno as found from account where accno=:m;
if(found==null)
  insert into account values (:m, 0, :type);
else
  print 'Error: Requested account number is already in use';
endif
commit;
```

```
rset={account.accno}
wset={account.*}
```

Figure 4.4: Create New Account With Desired Account Number

We therefore explicitly check for the situation where an edge in CSDG is labeled vulnerable *only* because of a conflict between one program which has select max that is used to create the value of primary key in a subsequent insert, and another program which has an insert to the same table. We must be careful not to identify an edge which has such a conflict but also has other read-write conflicts; this edge may be truly vulnerable. Note that our checks also apply to self-loop edges, that is, the two programs involved may be the same.

Definition 4.10 New Identifier Generation Analysis. *We say that a transaction is found to be a false positive using New Identifier Generation analysis if*

- *it is detected as a syntactic pseudopivot, and it is not found to be a false positive by MPR analysis, and*
- *after eliminating vulnerable edges created only because of a conflict between select-max used to calculate a primary key for insertion, and insert, the transaction is found not to be a pivot.*

□

It is common practice to test whether an identifier is in use, before inserting a tuple with that identifier. Such a select statement can't be in conflict with any *insert* to the table in a concurrent transaction, because if they are dealing with different key values there is no conflict, and if they are dealing with the same key value, then both will try to insert and one must fail to maintain the primary key uniqueness. If this situation is the only reason for an edge from P, that edge is not in fact vulnerable. If however there are other read-write conflicts as well, the edge should be kept as vulnerable

Example 4.5 ***Check for existence before inserting***. *Consider a new account creation transaction with provision to prespecify desired account number (Figure 4.4). The programmer tries to make sure that specified account number is not already assigned. Note that this program will be detected as syntactic pseudopivot and is a false positive.* □

Definition 4.11 Existence Check before Insert Analysis. *We say that a transaction is found to be a false positive using Existence Check before Insert analysis if*

- *it contains a select using equality on primary key and also does insert with that same primary key value in the same table whenever the select returns zero rows.*
- *it is detected as a syntactic pseudopivot, and it is not found to be a false positive by MPR analysis, and*

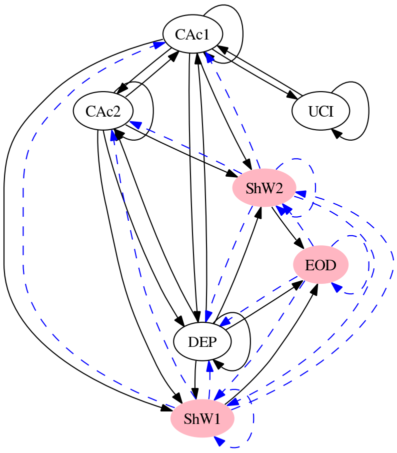


Figure 4.5: CSDG for simplified banking application after removal of false positives. Shaded nodes are the remaining syntactic pseudopivots.

- *after eliminating vulnerable edges created only because of a conflict between select that uses an equality predicate on primary key and insert, the transaction is found not to be a pivot.*

□

Extending the above idea to more general classes of programs, and to other integrity constraints, such as foreign key constraints, is an area of future work.

Example 4.6 ***Reducing False Positives for the Mini Banking Application.*** *The transactions* DEP, CAc_1, CAc_2 *and* UCI *were found as false positives in the simplified banking application, from Example 3.1 using the MPR analysis and the New Identifier Generation analysis. The Figure 4.5 shows the resulting CSDG.* □

Chapter

5

Architecture of SI Testing Tool

We have built a tool for analyzing application programs, with the goal of identifying possible anomalies, using the theory presented in the earlier sections. In this section we outline the architecture of the tool, detailing the steps taken to analyze an application.

5.1 Extracting Transactions

The first step in analysis is to extract the set of transaction programs that can be generated by the application. As mentioned in Section 3, we can get transaction programs either by analyzing application code, or by getting traces of queries submitted to the database.

If we have access to the source code of the application programs, we can try to extract every SQL statement found in the program. The extracted SQL statements are parametrized by the inputs to the application. Not all SQL statements in an application program may be executed on every invocation of the application. For example, in a program of the form "*W; if C then X else Y; Z*" either X or Y is executed but not both. Following [6], we can "split" such a program into two straight-line transactions: "*if not(C) abort; W; X; Z*" and "*if C abort; W; Y; Z*". For the case of loops, we could consider all possible unrollings of the loop, but this would be inefficient. Since SQL statements are parametrized anyway, and duplicates can be ignored for analysis, we can get finite transaction programs even in the presence of loops. However, in addition to being hard to automate, this may be difficult (or even impossible) if the programs construct the SQL statement dynamically, for example by concatenating string fragments. We therefore assume that if extraction from source code is required, it is done manually. For example, we did this to analyze the programs in the TPC-C benchmark.

The other way to obtain transaction programs is to capture the SQL statements submitted to the database during execution. This might be done while the system is executing normally, with transaction identifiers or session information used to link together the statements that form each separate transaction. Alternatively, we may execute the application programs serially, each with a wide variety of parameter values.

The drawback of using traces of SQL statements submitted to the database to obtain transaction programs is that one cannot be sure that every significant path of control flow has been exercised.

If some path does not get executed during testing, the corresponding transaction instances will not be considered by the tool, and as a result some anomalies may escape detection. If a good test suite is available, which exercises all parts of the application code, we can use it to generate a set of transactions with good coverage. In this case, the tool is still of great value as a testing tool, even though it cannot be a verification tool. It is possible to augment these transactions with transactions generated by manual analysis of the application logic, to ensure complete coverage.

Our tool supports the extraction of transaction programs from logs of SQL statements collected from the database. The tool parametrizes the SQL statements and eliminates duplicates, which allows a large set of transactions to be compacted to a much smaller set of transaction programs. In our experiments we obtained logs containing SQL statements by using the *auditing* feature provided by Oracle, or the *statement logging* feature of PostgreSQL.

Our tool parses the SQL statements using the JavaCC parser generator with the SQL grammar available at [14], and extracts the syntactic read and write sets. The tool also extracts predicates for analysis, using the expression parser provided by [13]. The CSDG is displayed in the graphical form using graph layout product Dot[17] where each transaction is a node in the graph. If a query includes the *select for update* clause, and the platform treats these rows as modified when doing first-committer-wins checks, then the contents of the read set are moved to the write set, leaving the read set empty. This reflects the effect of *select for update* on snapshot isolation.

5.2 Steps in Analysis

The analysis begins with the syntactic column-name analysis from Section 3. Our tool then eliminates false positives due to MPR transactions, using the theory from Section 4, as follows: For each syntactic pseudopivot P detected through the analysis

1. Consider any cycle in CSDG containing a subpath $R \xrightarrow{vul} P \xrightarrow{vul} Q$. If for every such cycle, P is MPR with respect Q, then declare P as a false positive.
2. If P is not found as a false positive by the previous test, apply the New Identifier Generation protection test and the Existence Check before Insert Test (Section 4.2). If either test succeeds, declare P as a false positive.

The output of the tool consists of a CSDG, with highlighting on all pseudopivots that are not found as false positives. Transactions in CSDG are identified by transaction identifiers, and we also provide a list of all transactions with their identifiers and their contents, that is, the (parametrized form of the) statements executed by the transactions. These can be used to locate the corresponding transactions in the application code, and we can use the techniques described in Section 2.5 to avoid anomalies.

Here is a summary of the flow of activities in the tool, when applied on an application:

1. **Step 1:** Find the set of transaction programs for the application.
2. **Step 2:** Use conservative analysis for creating the column-based syntactic dependency graph (CSDG). Use CSDG to detect syntactic pseudopivots in the application.
3. **Step 3:** Reduce false positives present in the set of syntactic pseudopivots obtained in step 2.
4. **Step 4:** Select appropriate techniques to avoid anomalies for the set of potential pivots remaining after step 3. This step is not currently implemented in the tool and must be carried out manually, using techniques described in [6] (outlined in Section 2.5).

	Acad.	Finance	TPC-C	Bank
Distinct txns	26	34	7	7
Syntactic Pseudopivots detected	25	34	4	7
MPR detected	11	3	4	2
New Identifier Generation Protection detected	3	3	0	2
Existence Check before Insert Protection detected	2	0	0	0
Remaining Potential Pivots	9	28	0	3
Verified True Pivots	2	2	0	3

Table 5.1: Results for Academic System (Acad.), Financial Application (Finance), TPC-C benchmark (TPC-C), and simplified banking application (Bank)

5.3 Experimental Results

We used our tool to analyze two applications, a financial application which runs on Oracle 10g, and an academic system which runs on PostgreSQL 8.1.4, which are in use at IIT Bombay.

The academic system is used to automate various academic activities, including course registration, online course feedback, grade allocation, modification of courses, faculty information and student information, and generation of numerous reports such as grade cards and transcripts. For the case of the academic system we instrumented the live database, and collected logs of all transactions that were executed in one day and supplemented with seasonal transactions, such as registrations, that were not active when we collected the logs.

Among the transactions that caused conflicts was an end-of-semester summarization transaction, which reads all grades allocated to each student in the semester, calculates grade point averages, updates a summary table, and inserts records into a transcript table. There were several other transactions, each of which updated a single row of one table with values provided by the user, which appeared to be pivots, but were found to be MPR since the only row that they read was the row that they updated.

The financial system is used to track all payments and receipts, starting from creation of bills, approval of bills and payment (posting) of bills, budget management, payroll management, and generation of a large number of reports. For the case of the financial application, we (manually) executed a test suite of transactions, and used the corresponding transaction logs. One transaction worth mentioning is the end-of-day transaction, which aggregates information about all bills that were paid or money received in that day, and moves data about all such transactions from a current-day table to a history table. This transaction conflicts with all transactions related to payment or receipt of money. There were several transactions that created new bills or purchase orders which were found as false positives.

Table 5.1 shows the results of running our tool on 4 different applications: the academic system and the financial application (which are live systems in use at IIT Bombay), as well as TPC-C and the simplified banking application used in our examples.

As can be seen from Table 5.1, our tool detected a fair number of pseudopivots, some of which were subsequently found as false positives and eliminated using MPR analysis, New Identifier Generation analysis and Existence Check before Insert Test. For the case of the academic system, our automated analysis was quite successful in finding and removing false positives. For the case of the financial application, the tool did eliminate some false positives, but a number of potential anomalies remained for manual examination, since the queries were too complex

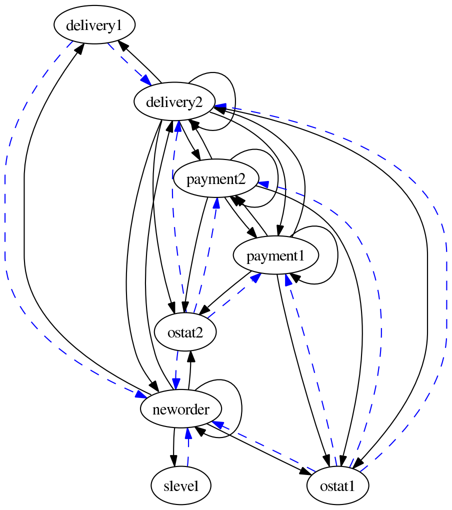


Figure 5.1: CSDG for TPC-C

for our current implementation (they contained outer joins and subqueries, which our current implementation does not handle). We don't have the full application code for financial system, and hence do not have enough semantic information to know which of the potential pivots can be eliminated.

However, it is important to note that the tool did in fact find several cases which turned out to be real pivots. Some of these are very unlikely to occur in practice and could be ignored. Others had to be fixed, in particular the end-of-day and end-of-semester transactions mentioned earlier were potentially dangerous. As mentioned in Section 1, financial auditors at IIT found a problem with an account, which we eventually traced to an SI anomaly. Our tool was able to detect this problem, as well as some other problems, helping us to fix them, and it allowed us to ignore several other cases since it found them to be false positives.

The set of transactions in TPC-C were obtained in the form of parameterized SQL queries by analyzing the procedures and splitting at the control structures (if, while, goto etc.) manually. (We included the splitting of *payment* and *ostat* transactions, which were skipped in [6] based on manual analysis showing they were not relevant.) All the *non-readonly* transactions were found to be MPR with respect to all the other transactions, matching the manual analysis in [6]. The results obtained are shown in Table 5.1, while the CSDG obtained by our tool is shown in Figure 5.1.

Table 5.1 also lists results for our banking example, where initially all the transactions were detected as syntactic pseudopivots. Using the techniques to find false positives, we narrowed down the set of pivots (Figure 4.5). The remaining pivots are real and can cause anomalies. For e.g., ShW_1 and $ShW2$ can cause write skew anomaly whereas EOD can cause a phantom anomaly.

In all cases our tool executed in less than 1 minute on the parametrized transaction programs described above. The task of generating parametrized transaction programs from large SQL traces can be somewhat slower, and took less than 5 minutes for 16000 SQL statements. These overheads are clearly still acceptable for the benefits provided.

Chapter 6

Implementation Issues in Avoiding Anomalies

We will now discuss various issues in using some of the techniques for modifying transaction programs to avoid anomalies, as mentioned in Section 2.5.

6.1 S2PL for Pivots

As we have seen in Section 2.5.1, running pivots with strict two-phase locking (S2PL) will avoid all anomalies. This can be done on platforms like Microsoft SQL Server. However Oracle and PostgreSQL do not provide an isolation level that uses S2PL. We can use *select for update* to partially simulate S2PL, but this does not protect against phantoms. We can simulate the effect of table-granularity S2PL on Oracle, or on PostgreSQL, by explicitly setting locks. Note that table-granularity locking means that there are no phantoms or anomalies due to predicate-read-to-write conflicts. But, the table-granularity locks also reduce concurrency greatly. To simulate S2PL on these platforms, the programmer can use the following approach.

1. Declare the pivot transaction T to have Isolation Level “Read Committed” (so each read or write sees the latest committed data at the time it is executed);
2. Then, explicitly LOCK Table (in appropriate mode) for every table read or written, before the *select* or *update/insert/delete* statement is executed.

Note that this does not properly simulate S2PL if the pivot runs at isolation level ”serializable” (i.e. Snapshot) because then selects use older data from the transaction snapshot, rather than current data as required of S2PL. This is a surprising situation, where raising the declared isolation level actually introduces anomalies.

6.2 Promotion

The promotion technique uses row level locks and has a lower impact on concurrency than using table locks. When applied to a transaction T, promotion is supposed to convert the outgoing vulnerable edges from T into non-vulnerable edges.

6.2.1 Phantom Problem

It may not suffice use the promotion technique in some transactions where the conflict is between a predicate read and a write, because it does not prevent phantoms.

```
begin;
select max(endtimestamp) as s, current_timestamp as c  from batchaudit;
select sum(amount) as d from txn where type='Deposit';
select sum(amount) as w from txn  where type='Withdraw';
insert into batchaudit(starttimestamp, endtimestamp,  inamount, outamount) values (:s,:e,:d,:w);
commit;
```

Figure 6.1: *End of day audit* transaction

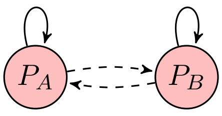
Example 6.1 ***Example where promotion is not sufficient.*** *Consider the End-of-the-day audit transaction (EOD) shown in Figure 6.1. The batches created by EOD are supposed to be non-overlapping. In Figure 4.5, EOD is detected as syntactic pseudopivot due to vulnerable edge to itself. One might think of using promotion to convert this self vulnerable edge to non-vulnerable edge. Now consider two execution instances (T_1 and T_2) of EOD modified to use promotion. If predicate locks are not supported by DBMS, promotion used in T_1 would only check rows from its own snapshot for first-committer-wins policy and miss the row concurrently inserted by T_2 and vice versa. i.e. if T_1 and T_2 execute concurrently, they both will read same value of max(endtimestamp) and would create overlapping batches and both will be allowed to commit. This indicates that the self vulnerable edge of EOD is not converted to non-vulnerable by promotion.* □

We can use the MPR test to decide whether use of *promotion* can convert a vulnerable edge into non-vulnerable edge. Consider a vulnerable edge $P_A \xrightarrow{vul} P_B$. Let P'_A be the modified transaction program after applying promotion to P_A. If P'_A is MPR *w.r.t.* P_B then according to Theorem 2, the edge from P'_A to P_B can not be vulnerable.

The overheads of promotion can be expected to be significant in the presence of contention, since promotion prevents some concurrency. As a result, we seek to minimize the use of promotion to that necessary to ensure serializable executions.

6.2.2 Optimizations

Consider the case of two transaction programs P_A and P_B, with $P_A \xrightarrow{vul} P_B$ and $P_B \xrightarrow{vul} P_A$, as shown in the following figure:



Both the transactions are syntactic pseudopivots, and need to be modified to ensure serializability. In case we use the S2PL approach, we would need to run both the pivots under S2PL. However, in case we use promotion, it is sufficient to modify either one of P_A and P_B to use promotion. Use of promotion in a pivot replaces all outgoing vulnerable edges from the pivot, by non-vulnerable edges. Thus, use of promotion might require modification of a fewer number of transactions than using S2PL.

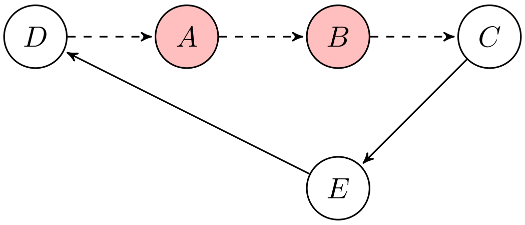


Figure 6.2: Example of *Dangerous Structure*

Definition 6.1 Dangerous Structure. *[6] A cycle in CSDG with consecutive vulnerable edges is a dangerous structure.* □

Definition 6.2 Dangerous Edge Pair (DEP). *The consecutive vulnerable edges in a dangerous structure form a Dangerous Edge Pair.* □

Definition 6.3 CanFix relation. *The set of dangerous edge pairs which can be removed using promotion in a pivot P is given by the relation $CanFix(P)$.* □

Every dangerous structure in CSDG identifies some syntactic pseudopivot transactions. E.g. the dangerous structures in Figure 6.2, identifies pivot transaction A, B, with (DA, AB) and (AB, BC) as the dangerous edge pair. Also, $CanFix(A) = \{(DA, AB), (AB, BC)\}$ and $CanFix(B) = \{(AB, BC)\}$.

Depending upon our goals, we can seek to make changes in a minimum number of transactions or in a minimum number of statements.

In order to avoid anomalies, we need to ensure that the set of all dangerous edge pairs are covered by $\bigcup_{P_i \in P} CanFix(P_i)$, where P is the subset of set of all transaction programs that are modified by promotion. Suppose we seek to minimize the number of programs in the set P.

Definition 6.4 DEPs Cover Problem (DEPC). *Let $G = (S_P, E)$ be the Column-based syntactic dependency graph for a set of transaction programs S_P with a set of static dependency edges E. Let S_{DEP} be the set of DEPs. Given such CSDG, the DEPs cover problem (DEPC) is to find a set of transaction programs in S_P such that replacing all vulnerable edges out of them by nonvulnerable edges results in removal of all dangerous edge pairs in S_{DEP}. In the DEPs cover optimization problem the task is to find a DEPs cover which uses the fewest transactions programs.* □

In the example shown in Figure 6.2, as $CanFix(A)$ covers the set of all dangerous pairs, it is sufficient to modify transaction A only. This minimization problem can be shown to be NP-Hard. We can extend the above model to define another optimization problem to minimize the number of statements to be modified in given a set of pivots. We will need to define a different $StmtCanFix$ relation, which gives the set of dangerous edge pairs removed by using promotion in a given statement.

6.3 Materializing Conflicts

The technique of *materializing conflicts* can be applied to remove any pivot conditions (unlike *promotion*) and under any *database*. But, the conflicts to be materialized should be chosen carefully. Otherwise, we might introduce unnecessary conflicts *w.r.t.* other transactions, which would cause greater impact on concurrency. A generic method for *materializing conflicts* should be designed in order to ease the selection of the conflicts to be materialized.

	IX	S	X
IX	Y		
S		Y	
X			

Table 6.1: Relevent lock compatibility matrix for KVL

6.3.1 KVL

KVL using B^+ tree handles concurrency problem of concurrent data access or modifications and concurrent index structure access and modifications [8, 9].

It is assumed that the typical key in the B^+ tree index is of the form (keyvalue, RID). Locks are obtained for the keyvalue part of key. Locking a keyvalue does not lock any physical tuple or index entry, but represents a lock on every key having this keyvalue. Every row read from tables is locked in S mode and every row updated is locked in X mode. The locks for keys in indices along and for rows and/or table are used for data access control, whereas latches are used for controlling access to index pages.

We will focus on the problem of concurrent data access, not of index access. There are four basic B^+ tree index operations that KVL supports: Fetch, Fetch Next, Insert and Delete.

6.3.1.1 Fetch and FetchNext

Given a range query (a predicate), the Fetch and FetchNext operations help retrieving RID of keys having their keyvalue in the specified range.

1. Given a potential keyvalue represented by kv, various Fetch calls can be executed. $Fetch(> kv)$ will retrieve the RID of the entry with the lowest keyvalue greater than kv, if any exists.

2. The keyval of the RID retrieved is S locked. $Fetch(> kv)$ with kv greater than the largest keyvalue in the index will retrieve nothing, and a conceptual EOF keyvalue, higher than any other value in the index, will be locked.

3. There is a natural generalization to $Fetch(\geq kv)$ and $Fetch(= kv)$, where for $Fetch(= kv)$, the kv value must specifically exist in the sequence or nothing will be retrieved (in which case the lowest keyvalue greater than kv will be S locked).

4. After opening a scan on a keyvalue range such as $[kv_1, kv_2]$ (which can include both endpoints) with $Fetch(>= kv_1)$, the transaction continues with a sequence of calls to fetch the next entry. These calls are represented as $FetchNext(\leq kv_2)$, and operate like successive Fetch calls through a Cursor (unnamed) that remembers the prior index entry retrieved from one call to the next.

5. Each time an entry from a new keyvalue is retrieved (every retrieval in a unique index), the new keyvalue is KVL locked in S mode, and this will continue until kv2 has been reached or passed (the only way we can know to stop if kv2 does not exist). When a keyvalue beyond kv2 is observed, it will be S locked as well; an end to the index below kv2 will lock EOF.

6. In addition to this, we have to lock tuples from table itself too.

- Every tuple in table corresponding to the RID, retrieved by Fetch and FetchNext should be S locked.
- The use of indices advised by query plan may retrieve a superset of tuples which satisfy all the tests specified. E.g. if condition is $c_1 = 4$ *and* $c_2 = 5$, and optimizer advices use of index on c_1 only (may be because index is present only on c_1), then we will get superset of tuples which satisfy given condition. Whether the tuple accessed satisfies or not, it will remain S locked until the transaction commits.

6.3.1.2 Insert and Delete

1. Insert or Delete of an index entry with a known keyvalue locks the nextkeyvalue in IX mode in the case of Insert, and X mode in the case of Delete. The nextkeyvalue is the 'next' keyvalue up in the index order.
2. An X lock should be obtained on the current keyvalue which is the the keyvalue of the entry to be inserted or deleted.
3. In case of delete, X lock should be obtained for tuple being modified.

If the nextkeyvalue (and its covered gap) has been stabilized by a lock of a different transaction that is incompatible with an X or IX lock attempt, the Insert or Delete will be forced to wait. Many commercial implementations of SI do not provide protection against phantom. We propose following method based on simulation of key-value locking approach [8] under SI.

6.3.2 Simulating KVL under SI

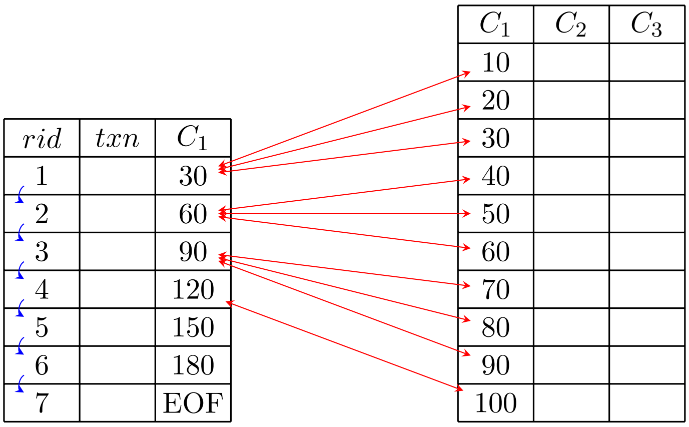
For KVL, we need to have control over index access. We can simulate such indices with the help of relations which are similar to sparse index on a single column of a table.

Definition 6.5 Pseudo-Sparse-Index Relation. *For a relation R we will simulate the index on its column c by a new relation* $psir.R.c(rid, c, txn)$ *such that*

- *The column named c has the same data type as c. This field is populated with tuples (randomly) sampled from the domain of data type of c along with the largest and lowest value in domain of data type of c.*
- *The rid column stores the position of tuple in the sorted list of c values in psir.R.c.*
- *The txn column stores the id of last transaction that locked the corresponding tuple.*

Once the data is populated, the transactions are not allowed to modify rid and c attributes. Also, insert and deletes are not performed on this relation. However, txn attribute of existing tuples can be modified as detailed below. We will call such relations used for simulating indices as Pseudo-Sparse-Index Relation (abbreviated as PSIR). Each tuple in the PSIR is called column-value (similar to keyvalue in KVL). □

These PSIRs will be used to materialize conflict between conflicting transactions. i.e., transactions that could conflict will be modified to access a tuple in a PSIR in conflicting mode. We will refer to this protocol for materializing conflicts as *PSIR protocol.*



rid	txn	C_1
1		30
2		60
3		90
4		120
5		150
6		180
7		EOF

C_1	C_2	C_3
10		
20		
30		
40		
50		
60		
70		
80		
90		
100		

Figure 6.3: Mapping of column-values in PSIR and rows of Table.

		T_2 Requests	
		RS (SFU)	RX (Snapshot Write)
T_1 holds	RS (SFU)	Not Standard	Not Standard
	RX (Snapshot Write)	Not Standard	✗

Table 6.2: Relevant lock compatibility matrix for row level locks and Snapshot Read Writes

6.3.2.1 Lock Modes

In KVL, the S-locks are used by Fetch/FetchNext operation and IX locks are used by insert operation to improve the concurrency. But in most commercial databases which provide SI, S-locks and IX locks are not supported. In such cases, the behavior of row level shared and exclusive mode locks can be conservatively approximated by using Row Exclusive locks for both modes. The comparison of locks obtained by KVL and simulated KVL is given in Table 6.3.

The row-exclusive locks are obtained using insert, delete or update statements. We can use update statement to obtain RX lock on a row, but we need to make sure that there are no side effect on the data of the row. Hence, we will use *identity update* on the row, where one of the attribute of the row is updated to the same value it has.

		Fetch-FetchNext, Select	Insert	Delete
KVL	Current KV	S mode	X mode	X mode
	Next KV	S mode	IX mode	X mode
PSIR KVL	Current KV	RX mode	RX mode	RX mode
	Next KV	RX mode	RX mode	RX mode

Table 6.3: Comparison of Index keyvalues locked in KVL and PSIR columnvalues locked in Simulated KVL

```
Procedure IdentityUpdatePSIRRow(PSIR psir, int rid)
  Data: psir=PSIR, rid=serial row number
  /* Identity Update for given tuple (identified by rid) in given PSIR    */
  Execute: UPDATE psir SET txn=:current-transactionid WHERE rid=:rid;
```

6.3.2.2 Choice of PSIRs

In KVL, the query plan decides what indices to use. The phantoms are avoided by

- conflict on table locks if no index is used
- conflict on keyvalue locks or row level locks if atleast one index used per table

In available commercial implementations of SI, table locks do not prevent against phantoms, because the new rows inserted by the transaction holding a table lock, would not be treated as part of the snapshot of a transaction waiting for that lock to be released.

However, materializing conflicts on column-values of PSIR (along with row level locks on relation) can avoid phantoms. Also, we can explicitly specify what PSIRs to be locked. Therefore for any predicate read that could be affected by phantoms, we need to make sure that atleast one PSIR is locked for each table involved in a predicate P.

Definition 6.6 SARG and Sargable Predicate. *A sargable predicate is one of the form (or which can be put into the form) 'columnname comparison-operator const'. SARGS are expressed as a boolean expression of such predicates in disjunctive normal form.* □

The procedure GetPSIRsSARGs gives us atleast one PSIR along with SARG for a given table and for given conjunctive predicate. This procedure tries to reduce the number of locks to be obtained using SARGs. But if it can not find any SARG, then it conservatively chooses all the rows in any one PSIR for given table to be RX locked. Note that, this is equivalent to table lock in KVL as inserts and deletes are not allowed on the PSIRs.

6.3.2.3 PSIR Protocol for Predicate Reads

In SQL, the *select* (including subqueries), delete and update statements use predicates. In KVL, predicate is processed by query plan generator and suitable index to be used is decided. The Fetch and FetchNext operations then use the indices to retrieve the rowids. Note that the rows identified by Fetch/FetchNext may not satisfy the entire query and are still locked in appropriate mode as mentioned in Section 6.3.1.1.

Similarly, we have the procedure PSIRProtocolForPred to identify and update the tuples for materializing conflict for a given predicate. Consider the transactions T_1 and T_2. If there is a predicate-read to write dependency from a predicate P in T_1 to some write in transaction T_2 then T_1 should be augmented with PSIRProtocolForPred method for the predicate P. This procedure takes as input the predicate D which is in disjunctive normal form and the set of tables ST on which D operates. Following steps are executed for each conjunction C_D in D.

1. It uses procedure GetPSIRsSARGs to get a PSIR and corresponding SARG for each table T in ST for the predicate C_D. This SARG also identifies the superset of tuples present in T on which result of predicate read depends.
2. Identity updates are executed on the rows in PSIR and on the rows in T which satisfy SARG (to materialize the conflict).

Procedure GetPSIRsSARGs(*Table T, Predicate C*)

```
Data: T=Table, C= Conjunctive Predicate
Result: Array of PSIRs and corresponding SARGs for table T in conjunct C
initializations;
foreach column c of table T do
    if SARG for column c exists in conjunct C then
        if Pseudo-Sparse-Index Relation is defined for column then
            sarg = SARG in conjunct for column;
            psir = Pseudo-Sparse-Index Relation for column;
            PSIRsSARGs[i++]=(psir, sarg);
        end
    end
end
if PSIRsSARGs is empty then
    /* No PSIR is suitable : We need to reserve entire table. i.e. all
       tuples in any one psir should be updated.                     */
    psir = Any PSIR on table;
    sarg = true;
    PSIRsSARGs[i++]=(psir, sarg);
end
return PSIRsSARGs;
```

Procedure PSIRProtocolForPred(*Predicate D, Set⟨Table⟩ST*)

```
Data: D = Predicate in DNF
Data: ST = Set of tables on which D operates
foreach conjunct C in D do
    foreach table T in ST do
        PSIRsSARGs = GetPSIRsSARGs(T, C);
        Conjunctive predicate conjsarg=true;
        foreach (psir, sarg) in PSIRsSARGs do
            Let [lowvalue, highvalue] be the range defined by sarg;
            minRid = maximum rowid in psir with $psir.C \geq lowvalue$;
            maxRid = minimum rowid in psir with $psir.C > highvalue$;
            forall $minRid \leq rid \leq maxRid$ do
                IdentityUpdatePSIRRow(psir, rid);
            end
            $conjsarg = conjsarg \wedge sarg$
        end
        IdentityUpdateTableRows(table, conjsarg);
    end
end
```

Procedure `IdentityUpdateTableRows(`*Table T, Predicate C*`)`

```
Data: T=table, C=Predicate
/* Execute identity update for any column c of Table T                  */
c= any column of T;
Execute : UPDATE T set c=c WHERE C;
```

6.3.2.4 Avoiding Phantoms due to Insert, Delete or Update

To avoid phantoms, the Insert, Delete or Update should take into account the locks obtained by predicate read operations. In KVL, this is achieved by updating all indices affected by the tuple-insert or tuple-delete. For all such indices, in case of insert operation the keyvalue to be inserted is X-locked and the the next keyvalue is IX-locked. And for delete operation keyvalue to be deleted and the next keyvalue are both X-locked.

We simulate this by invoking procedure PSIRProtocolForTupInsDel for each tuple inserted or deleted. This procedure does identity updates on certain rows (identified by given tuple) in all PSIRs defined for the affected columns.

Procedure `PSIRProtocolForTupInsDel(`*Table T, Tuple t*`)`

```
Data: T=table, t=Tuple
foreach column c of T do
    v = value of t.c;
    LckPSIRForColVal(T,c,v);
end
```

Procedure `LckPSIRForColVal(`*Table T, Column c, ColumnValue v*`)`

```
Data: T=table, c=column, v=ColumnValue, Mode M
if Pseudo-Sparse-Index Relations defined for column T.c then
    psir = Pseudo-Sparse-Index Relations defined for column T.c;
    min = maximum rowid in psir with $psir.c \geq v$;
    /* current columnvalue                                              */
    max = minimum rowid in psir with $psir.c > v$;
    /* next columnvalue                                                 */
    IdentityUpdatePSIRRow(psir, min);
    IdentityUpdatePSIRRow(psir, max);
end
```

The tuple-update operation, however, might update only a subset of the indices on the table. Hence, we identify the affected PSIRs using the attributes assigned value in the SET clause of update (see procedure PSIRProtocolForTupUpd). Both the old value and the new value of the tuple are used to detemine the rows in PSIR to be updated.

Procedure `PSIRProtocolForTupUpd(`*Table T, Tuple ot, Tuple nt, Set⟨Column⟩ C*`)`

```
Data: T=table, ot=OldTuple, nt=NewTuple, C=set of columns updated
foreach column c of C do
    ovl = value of ot.c;
    nvl = value of nt.c;
    LckPSIRForColVal(T,c,ovl);
    LckPSIRForColVal(T,c,nvl);
end
```

In procedures PSIRProtocolForTupUpd and PSIRProtocolForTupInsDel, we invoke procedure LckPSIRForColVal. This procedure identifies and updates two rows for given column-value v, in PSIR *psir* defined for column c on table R:

1. Current column-value: the row in psir with $psir.c \geq v$
2. Next column-value: the row 'next' to Current column-value. It is identified by the row in psir with $psir.c > v$

6.3.2.5 Correctness of PSIR Protocol

We prove the correctness of our approach on the lines of comments given in [9].

Theorem 4 *Under Snapshot Isolation, if a transaction program P_1 reading from table t uses procedure PSIRProtocolForPred for each of the predicate on t and another transaction program P_2 which modifies the contents of table t is forced to use PSIRProtocolForTupInsDel and PSIRProtocolForTupUpd for all insert, delete or update statements on table t, then the edge from P_1 to P_2 in SDG can not be vulnerable due to conflict on rows from table t.*
Proof : Consider any execution schedule where T_1 and T_2 are execution instances of P_1 and P_2 respectively, if T_1 and T_2 are concurrent, then for any predicate D used by T_1 on table t, following are the only cases when we can say the edge from P_1 to P_2 is vulnerable

1. T_2 updates or deletes some row in t that satisfies D.
2. T_2 inserts or updates any row in t that such that the new/modified row satisfies D.

We now show that both these cases can not occur if P_1 uses procedure PSIRProtocolForPred for D and P_2 uses PSIRProtocolForTupInsDel and PSIRProtocolForTupUpd for all insert, delete or update statements on table t.

1. T_2 can not update or delete any row in t if it satisfies D:
   Let r be any row that satisfies D. Now, r is updated by PSIRProtocolForPred T_1. But under SI, if multiple transactions modify the same row concurrently, only one of them is allowed to commit. Hence, either T_2 can not be concurrently with T_2 or it can not modify any rows that satisfy D.
2. T_2 can not insert or update any row in t that such that the new/modified row satisfies D:
   Let the new/modified row be r. IU_{PSIR} be the set of PSIRs updated by T_2 for such insert or delete statement and S_PSIR be the PSIR defined on column $t.c$ which is used by T_1 for D. There are two possible cases:
   (a) $S_{PSIR} \in IU_{PSIR}$:
   T_2 would have updated rows with column-value *ccv* in S_{PSIR} for $r.c$ and the next-column-value *ncv* for $r.c$ (Note that, in some cases *ccv* and *ncv* might be the same). If r satisfies D then either *ccv* or *ncv* is already updated by PSIRProtocolForPred in T_1. Hence, either r does not satisfy D or T_1 and T_2 do not execute concurrently.
   (b) $S_{PSIR} \notin IU_{PSIR}$:
   This can not happen if r is a new tuple inserted. Hence r must be due to update of a tuple r'. We claim that if r satisfies D then r' was also updated by T_2. The case where r' is not updated by T_1 appears only when there is some SARG on column $t.c$ which r' does not satisfy. But S_{PSIR} was not modified by T_2. i.e., the value of $r.c$ must be same as $r'.c$. Hence,

- either r still does not satisfy the SARG and hence the predicate D,
- or r and r' do satisfy the SARG and hence r' was also updated by T_1. i.e., T_1 and T_2 can not be concurrent.

Thus, in either of the cases there can not be a read write conflict from T_1 to T_2 involving table t.

□

Chapter 7

Discussion

In this chapter we discuss some other issues related to the snapshot isolation testing tool.

Triggers And Integrity Constraints: Often, triggers are used to preserve integrity constraints (ICs). Under SI, a trigger operates on the same snapshot as the transaction invoking it and hence can be vulnerable to SI anomalies. Therefore, some ICs can not be preserved using triggers which run under SI, unless the trigger itself uses explicit locking, promotion, or materialization to protect against anomalies. (In fact we found one such instance where a trigger failed to preserve an integrity constraint, due to SI, in the financial application used at IIT Bombay.) To detect which triggers need such protection, we can first find which transactions could invoke each trigger, and augment the transaction code with the trigger code. We then run our analysis on the augmented transactions, and wherever an augmented transaction is found to be a pivot, we have to protect the transaction using one of the techniques discussed in Section 2.5. Whether done through explicit table locking, or through additional writes or promotion, the overhead will be paid by all transactions that could cause the trigger to be fired.

Transaction Chopping: Large update transactions are often chopped into smaller transactions, to reduce the impact on locking and on the size of the active part of the log. Suppose a set of transactions S has a pivot. It is possible that if one of the transactions in S is chopped into two or more pieces, none of the transactions in the modified S may be pivots (e.g, Figure 7.1). Given a set S of transactions, Shasha et al. [12] provide sufficient conditions for chopping of a transaction T to be safe, in that the execution will be serializable. These conditions however also ensure that there will be no pivots, so using SI does not cause any further problems.

Problems Caused by Program Extensions: Let S be the set of transactions that are analyzed and found free of any anomalies. If S is then extended to include some other transactions, we need to run the analysis again. Note that the extensions might be free of anomalies themselves.

Example 7.1 . *Let initial set of transactions be as shown in Figure.7.2.*

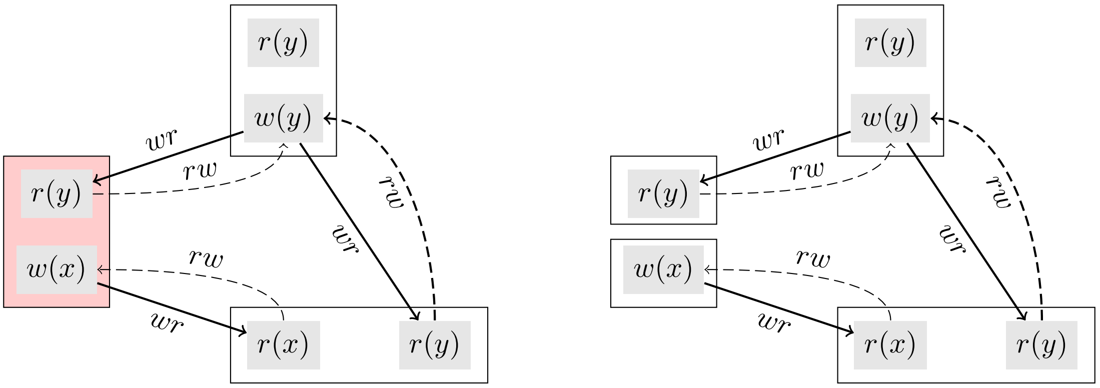


Figure 7.1: Chopping can miss a potential *syntactic pseudopivot*

T_1	$R(X)$	$R(Y)$	$W(Y)$
T_2	$R(X)$	$W(X)$	

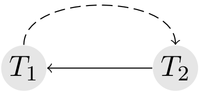


Figure 7.2: Pivot free set of transaction

T_1 becomes pivot, after addition of transaction T_3 as shown in Figure.7.3.

T_3	$R(X)$	$R(Y)$	
T_1	$R(X)$	$R(Y)$	$W(Y)$
T_2	$R(X)$	$W(X)$	

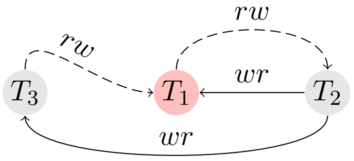


Figure 7.3: Pivot introduced due to addition of T_3

□

Workflow Constraints Some transactions can be guaranteed not to execute concurrently due to the workflow constraints. We can use such information to detect more transaction programs as false positives. Details of how to do so are part of future work.

Chapter

8

Conclusions & Future Work

Snapshot isolation, although widely used, can potentially cause non-serializable transaction histories. Applications running under SI are at risk of data inconsistency due to transaction anomalies. A theory that gives sufficient conditions for these anomalies was presented by Fekete et al. [6], and by Fekete [5]. We used this theory to define a syntactic condition that can be used to over-approximate the set of transactions that may cause anomalies. We studied some general patterns where a transaction can apparently cause anomalies, but it actually cannot, due to certain actions that the transaction performs such as modifying the data that it read. We proposed sufficient conditions for inferring that certain syntactic pseudopivot transactions are false positives, and the transactions are thus safe with respect to serializability. Our conditions take care of phantoms. Further, when pivots are detected and fixed using promotion or S2PL, reapplying the conditions can infer safety (as long as the conditions are satisfied after the fixes). We have also developed a tool that can automate the testing of database applications for safety against SI anomalies. Our tool has been used in practice with good effect, identifying some genuine problems in production code, and also verifying safety for many transactions.

Our approach of simulating KVL for materializing conflicts should be modified for PSIRs on multiple columns. Its performance also needs to be tested to see if it actually provides better concurrency than that of table level locks.

We feel there is need to standardize the behaviour of *select for update* in terms of *first-committer-wins* strategy. An efficient approximation algorithm for the DEPC optimization problem (Section 6) is another area of future work.

Yet another area of future work is in developing a theory for including workflow constraints (e.g. grading will never run concurrently with course registration), and integrity constraints other than primary keys (such as foreign keys) to reduce false positives.

Bibliography

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[2] 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, New York, NY, USA, 1995. ACM Press.

[3] P. Bernstein and N. Goodman. Mulitversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[5] 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.

[6] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

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

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

[9] Patrick O'Neil, Alan Fekete, Elizabeth O'Neil, and Dimitrios Liarokapis. Some considerations of locking to prevent phantoms. `"http://www.cs.umb.edu/~poneil/ppp.sigmod.ps"`.

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

[11] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM Press.

[12] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, 1995.

[13] Cayenne (exprerssion parser). http://cayenne.apache.org/doc/expressions.html.

[14] Javacc. https://javacc.dev.java.net/.

[15] Tpc-c benchmark. http://www.tpc.org/tpcc/, 2006.

[16] Postgresql 8.3devel documentation. http://developer.postgresql.org/pgdocs/postgres/transaction-iso.html#XACT-SERIALIZABLE, 2007.

[17] Dot and dotty. http://hoagland.org/Dot.html, 2006.