Making Snapshot Isolation Serializable

ALAN FEKETE
University of Sydney
DIMITRIOS LIAROKAPIS, ELIZABETH O'NEIL, and PATRICK O'NEIL
University of Massachusetts
and
DENNIS SHASHA
Courant Institute

Snapshot Isolation (SI) is a multiversion concurrency control algorithm, first described in Berenson et al. [1995]. SI is attractive because it provides an isolation level that avoids many of the common concurrency anomalies, and has been implemented by Oracle and Microsoft SQL Server (with certain minor variations). SI does not guarantee serializability in all cases, but the TPC-C benchmark application [TPC-C], for example, executes under SI without serialization anomalies. All major database system products are delivered with default nonserializable isolation levels, often ones that encounter serialization anomalies more commonly than SI, and we suspect that numerous isolation errors occur each day at many large sites because of this, leading to corrupt data sometimes noted in data warehouse applications. The classical justification for lower isolation levels is that applications can be run under such levels to improve efficiency when they can be shown not to result in serious errors, but little or no guidance has been offered to application programmers and DBAs by vendors as to how to avoid such errors. This article develops a theory that characterizes when nonserializable executions of applications can occur under SI. Near the end of the article, we apply this theory to demonstrate that the TPC-C benchmark application has no serialization anomalies under SI, and then discuss how this demonstration can be generalized to other applications. We also present a discussion on how to modify the program logic of applications that are nonserializable under SI so that serializability will be guaranteed.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Transaction processing*

General Terms: Theory, Reliability

The research of all the authors was supported by National Science Foundation (NSF) Grant IRI 97-11374 (see http://www.cs.umb.edu/~isotest/summary.html).
The work of Professor Shasha was also supported by NSF Grants IRI 97-11374, IIS-9988636, and N2010-0115586.
Authors' addresses: A. Fekete, School of Information Technologies, Madsen Building F09, University of Sydney, N.S.W. 2006, Australia; email: fekete@it.usyd.edu.au; D. Liarokapis, E. O'Niel, and P. O'Niel, Umass/Boston, Boston, MA 02125-3393; email: {dimitris,eoniel,poneil}@cs.umb.edu; D. Shasha, Courant Institute, 251 Mercer Street, New York, NY 10012; email: shasha@cs.nyu.edu.

Additional Key Words and Phrases: Concurrency control, serializability, anomaly, consistency, weak isolation, snapshot isolation, multiversion concurrency

1. MOTIVATION AND PRELIMINARIES

Many database researchers think of concurrency control as a solved problem, since there exists a proven set of sufficient conditions for serializability. The problem is that those sufficient conditions can lead to concurrency control bottlenecks, so most commercial systems offer *lower isolation level* concurrency settings as defaults that do not guarantee serializability [Berenson et al. 1995]. This poses a new task for the theorist: to discover how to guarantee correctness at the least cost for such lower isolation levels. In this article, we address the isolation level known as Snapshot Isolation (SI). SI, defined below in Section 1.1, is a multiversion concurrency control mechanism that prevents many classical serialization anomalies, but allows nonserializable executions in certain cases. It is widely used due to its good performance, especially the fact that it does not block reads; some major vendors support only SI rather than traditional serializability.

The contribution of this article is to lay a theoretical foundation for giving a database administrator the tools to decide whether a particular database application, consisting of several interacting programs acting under Snapshot Isolation, will produce only serializable executions. When this is not the case, we give guidance as to how to modify the application to guarantee serializable execution. As a demonstration of the applicability of our theory, we show how to analyze the TPC-C application, running on the Oracle DBMS under what is designated as the SERIALIZABLE isolation level, a form of Snapshot Isolation. We note that when two official auditors for the TPC-C benchmark were asked to certify that the Oracle SERIALIZABLE isolation level acted in a serializable fashion on the TPC-C application, they did so by "thinking hard about it" (Francoi's Raab and Tom Sawyer, personal communication, 1995). It is noteworthy that there was no theoretical means to certify such a fact, a drawback that we rectify in this article.

In what follows, we define a *transactional application* to comprise a set of *transaction programs* that perform some integrated set of services for an enterprise, for example, to execute all services that take place at teller windows of a bank: deposits, withdrawals, transfers, etc. We assume that we are provided with all the program logic for the transaction programs, each of which contains numerous statements that access and update a database, bracketed between operations that begin and end (commit or abort) each transaction execution. These transaction programs should not be thought of as executing in a straight line on specific, named data items[1] however, even though that situation is typically assumed in textbooks where multi-transactional history notation is

[1]We leave the term *data item* ambiguous in the usual way; all definitions and proofs go through for any granularity of data substituted for this term. We note, however, that in the analysis of TPC-C, we refer explicitly to *field* granularity, a *field* being a specific column value on a specific row of a table.

defined, for example, by the history given in (1.1).

$$R_1(X)\ R_2(X)\ W_2(X)\ C_2\ W_1(X)\ A_1 \tag{1.1}$$

While this notation is useful to represent specific executions of transactional programs, we cannot normally analyze programs at compile time to determine which specific data items will be accessed. Such programs are implemented in a programming language that executes SQL statements, called with appropriate arguments (e.g., Customer_ID in a banking application) to determine which customer account is to be accessed. The programs retrieve and update data items determined by these arguments, but no compile-time analysis can know which specific data items will be accessed, since different data items will be determined by different parameter values. Furthermore, conditional tests on the arguments passed and other data the programs access in the database can cause branching that might lead to many different executions, accessing quite different parts of the database, so straight-line execution is likewise unrealistic. Our model is quite general, but for practical application of our theory one would usually avoid having functionally different transactional tasks arise from different branches of logic in a program. Instead, we will usually expect that each program carries out the logic to perform a single transaction type, comparable to the ones described in the TPC-C benchmark specification [TPC-C].

1.1 Versioned Histories and Snapshot Isolation

In order to reason about the execution of a collection of application programs on a database under SI, we need a formal representation of histories rich enough to describe multiple versions of data items and set-oriented data access. We use the model of Adya et al. [2000]. (Much of our notation and some of our definitions will vary from that of Adya et al. [2000], however.)

We assign version numbers to data items in Snapshot Isolation histories (and various other multiversion histories) as subscripts equal to the ID of the transaction that last wrote them. All data items start with zero versions, X_0, Y_0, $Z_0, \ldots$ (we imagine that $X_0, Y_0, Z_0, \ldots$ have all been installed by a "progenitor transaction" T_0), and when transaction T_i writes a data item X (for the last time among possible multiple writes in that transaction), it creates a version that inherits the data item name and the Transaction ID: X_i. This operation is denoted as $W_i(X_i)$. On commit, a transaction's final versions are said to be installed, or written, in the database state. Note particularly that the numbering of versions reflects their writers, and not their temporal version order. However, the temporal version order of all updates of any single data item must always be reflected in the history, even for histories of transactions executing on parallel shared-nothing processors that provide only a partial order for many operations. The notation $R_i(X_k)$ represents a read operation by transaction T_i that returns the version of the data item X_k written earlier in the history by transaction T_k. To simplify the treatment of deletes and inserts, making them special cases of writes, deletion of data item X by transaction T_i is modeled as writing a special dead version, X_w, $W_i(X_w)$, and insertion of data item X by transaction T_i is treated as writing a new (initial) data item version X_i, where the prior version had a special subscript representing unborn data, X_α.[2] (Note that the creating Transaction ID is used only for version numbers as a convenience for recognizing later in the history what transactional output is being read; since unborn and dead versions are never read at a later time, there is no contradiction in using special version subscripts in these cases.) Snapshot Isolation, or SI, is a multiversion concurrency control algorithm introduced in Berenson et al. [1995]. In what follows, we define time to be measured by a counter that advances whenever any transaction starts, commits, or aborts, and we designate the time when a transaction T_i starts as start(T_i) and the time when T_i commits or aborts as complete(T_i); when T_i is successful, we also write complete(T_i) as commit(T_i).

Definition 1.1 (*Snapshot Isolation*). A transaction T_i executing under Snapshot Isolation reads data from the committed state of the database as of time start(T_i) (the *snapshot*), and holds the results of its own writes in local memory store, so if it reads data it has previously written, it will read its own output. Predicates evaluated by T_i are also based on data item versions from the committed state of the database as of time start(T_i), adjusted to take T_i's own writes into account. The interval in time from the start to the completion of a transaction, represented [start(T_i), complete(T_i)], is called its *transactional lifetime*. We say that two transactions are *concurrent* if their transactional lifetimes overlap, that is, [start(T_1), complete(T_1)] $\cap$ [start(T_2), complete(T_2)] $\neq \oslash$. Writes performed by transactions that were active after T_i started, that is, writes by *concurrent transactions*, are not visible to T_i. When T_i is ready to commit, it obeys the *First Committer Wins* rule, stated as follows: T_i will successfully commit only if no other concurrent transaction T_k has already committed writes (updates) to data items where T_i has written versions that it intends to commit. (We can't say *if and only if* in this definition because we want the algorithm to be defined so that it applies to a wide range of different implementations, and sometimes a transaction will abort in a particular DBMS' implementation because of *nearby* concurrent updates by other transactions, for example, updates of data items on the same page.) Note that the check need only be done against transactions that are concurrent with T_i and have committed when T_i commits (no check is needed if T_i aborts); those that are still active when T_i finishes need not be not checked, but instead, when they try to commit they will check for modifications by T_i. See also the discussion of the variant *First Updater Wins* rule below.

The First Committer Wins rule is reminiscent of certification in optimistic concurrency control, but only items written by a committing T_i are checked for concurrent modification, not items read.

In the Oracle implementation of Snapshot Isolation (referred to as the *SERIALIZABLE* Isolation Level by Oracle in Jacobs et al. [1995]), an attempt

[2]When we speak of inserting or deleting a data item of field granularity, we assume that the insert or delete deals simultaneously with all the fields (column values) contained in a given row that is inserted or deleted.

by T_i to read a row[3] that has changed since start(T_i) will cause the system to read an older version, current as of start(T_i), in the rollback segment. Indexes are also accessed in the appropriate snapshot version, so that predicate evaluation retrieves row versions current as of the snapshot. The First Committer Wins rule is enforced, not by a commit-time validation, but instead by checks done at the time of updating. If T_i and T_k are concurrent (their transactional lifetimes overlap), and T_i updates the data item X, then it will take a Write lock on X; if T_k subsequently attempts to update X while T_i is still active, T_k will be prevented by the lock on X from making further progress. If T_i then commits, T_k will abort; T_k will be able to continue only if T_i drops its lock on X by aborting. If, on the other hand, T_i and T_k are concurrent, and T_i updates X but then commits before T_k attempts to update X, there will be no delay due to locking, but T_k will abort immediately when it attempts to update X (the abort does not wait until T_k attempts to commit). For Oracle, we rename the *First Committer Wins* rule to *First Updater Wins*; the ultimate effect is the same—one of the two concurrent transactions updating a data item will abort. Aborts by T_k because of being beaten to an update of data item X are known as serialization errors, ORA-08177 (Oracle Release 9.2).

Snapshot Isolation is an attractive isolation level. Reading from a snapshot means that a transaction never sees the partial results of other transactions: T sees all the changes made by transactions that commit before start(T), and it sees no changes made by transactions that commit after start(T). Also, the First Committer Wins rule allows Snapshot Isolation to avoid the most common type of lost update error, as shown in Example 1.1.

Example 1.1 (*No Lost Update*). If transaction T_1 tries to modify a data item X while a concurrent transaction T_2 also tries to modify X, then Snapshot Isolation's First Committer Wins rule will cause one of the transactions to abort, so the first update will not be lost. For example (we include values read and written for data items in this history):

$$H_1\colon R_1(X_0, 50)\ R_2(X_0, 50)\ W_2(X_2, 70)\ C_2 W_1(X_1, 60)\ A_1.$$

This history leaves X with the value 70 (version X_2), since only T_2, attempting to add an increment of 20 to X, was able to complete. T_1 can now retry and hopefully add its increment of 10 to X without interference. Note that many database system products with locking-based concurrency default to the READ COMMITTED isolation level, which takes long-term write-locks but no long-term read locks (it only tests reads to make sure they do not read write-locked data); in that case, the history above without versioned data items would succeed in both its writes, causing a Lost Update.

Despite its attractions, Snapshot Isolation does not ensure that all executed histories are serializable, as defined in classical transactional theory (e.g., in Bernstein et al. [1987], Papadimitriou [1986], and Gray and Reuter [1993]). Indeed, it is quite possible for a set of transactions, each of which in isolation

[3]We refer specifically to rows having versions when discussing the Oracle implementation; however we note that the concepts still apply to data items at field granularity: specific column values on specific rows.

respects an integrity constraint, to execute under Snapshot Isolation in such a way as to leave the database in a corrupted state. One such problem was called "Write Skew" in Berenson et al. [1995].

Example 1.2 (*SI Write Skew*). Suppose X and Y are two data items representing checking account balances of a married couple at a bank, with the constraint that $X + Y > 0$ (the bank permits either account to be overdrawn, as long as the sum of the account balances remains positive). Assume that initially $X_0 = 70$ and $Y_0 = 80$. Under Snapshot Isolation, transaction T_1 reads X_0 and Y_0, then subtracts 100 from X, assuming it is safe because the two data items added up to 150. Transaction T_2 concurrently reads X_0 and Y_0, then subtracts 100 from Y, assuming it is safe for the same reason. Each update is acting consistently, but Snapshot Isolation will result in the following history:

$$H_2\colon R_1(X_0, 70)\, R_2(X_0, 70)\, R_1(Y_0, 80)\, R_2(Y_0, 80)\, W_1(X_1, -30)\, C_1\, W_2(Y_2, -20)\, C_2$$

Here the final committed state (X_1 and Y_2) violates the constraint that $X + Y > 0$. This problem was not detected by First Committer Wins because two different data items were updated, each under the assumption that the other remained stable. Hence the name "Write Skew".

While Example 1.2 displays one of a number of anomalous situations that can arise in Snapshot Isolation execution, the prevalence of such situations may be rare in real-world applications. As mentioned earlier, the TPC-C benchmark application, consisting of five transactional programs, displays no such anomalies, and it is reasonably representative of a large class of applications. We should also point out that it is quite easy to modify application or database design to avoid the anomaly of Example 1.2. One way to do this is to require in the transactional program that each Read of X and Y to update Y give the impression of a Write of X (this is possible in Oracle using the Select For Update statement). Then it will seem that both X and Y are updated and First Updater Wins collision will occur in the history H_2. Another approach requires that each constraint on the sum of two accounts X and Y be materialized in another data item Z, and insists that all updates of X and Y must keep Z up to date. Then the anomaly of history H will not occur, since collision on updates of Z will occur whenever X and Y are updated by two different transactions.

Another type of anomaly can occur resulting from read-only transaction participation, as previously shown in Fekete et al. [2004]. Such problems have long been known [Papadimitriou 1986] in executions with weak isolation levels like Read Committed; however, starting with Berenson et al. [1995], it was assumed that under SI read-only transactions always execute serializably, without ever needing to wait or abort because of concurrent update transactions; this is because all reads in a transaction appear to take place at an instant of time, when all committed transactions have completed their writes and no writes of noncommitted transactions are visible. The implied guarantee is that read-only transactions will not read anomalous results so long as the update transactions with which they execute do not write such results. However, Example 1.3 shows this is not true.

Example 1.3 (*SI Read-Only Transaction Anomaly*). Suppose X and Y are two data items representing a checking account balance and a savings account balance. Assume that initially $X_0 = 0$ and $Y_0 = 0$. In the following history, transaction T_1 deposits 20 to the savings account Y, T_2 subtracts 10 from the checking account X, considering the withdrawal covered as long as $X + Y > 0$, but accepting an overdraft with a charge of 1 if $X + Y$ goes negative, and T_3 is a read-only transaction that retrieves the values of X and Y and prints them out for the customer. For one sequence of operations, these transactional operations will result in the following history under SI:

$$H_3\text{: } R_2(X_0, 0)\ R_2(Y_0, 0)\ R_1(Y_0, 0)\ W_1(Y_1, 20)$$
$$C_1\ R_3(X_0, 0)\ R_3(Y_1, 20)\ C_3 W_2(X_2, -11)\ C_2.$$

The anomaly that arises in this transaction is that read-only transaction T_3 prints out $X = 0$ and $Y = 20$. This can't happen in any serializable execution which ends in the same final state as H_3, with $X = -11$ and $Y = 20$, because if 20 were added to Y before 10 were subtracted from X in any serial execution, no charge of 1 would ever occur, and thus the final balance should be -10, not -11. A customer, knowing a deposit of 20 was due and worried that his check for 10 might have caused an overdraft charge, would conclude that he was safe, based on this balance result. Indeed, such a print-out by transaction T_3 would be very embarrassing for a bank should bank regulators ask how the charge occurred. We also note that any execution of T_1 and T_2 (with arbitrary parameter values) without T_3 present will always act serializably. As we will show later in the article, the anomalies of Example 1.2 and 1.3 are allied, in the sense that properties of a graph we will define can be used to predict both problems (and both problems are preventable). We note that if T_1 and T_2 used two phase locking instead of SI, this anomaly would not occur, and every execution would be serializable.

We have observed the histories of Examples 1.2 and 1.3, and other violations of nonserializable behavior, to take place on an Oracle DBMS after "set transaction isolation level serializable". Similarly, while Snapshot Isolation prevents most classical examples of phantoms (because the snapshots read by the transactions include index data, and therefore predicate evaluations will be repeatable in spite of modifications by other transactions), certain phantom problems analogous to Write Skew can also occur under SI. We will discuss this in the next section.

1.2 Predicate Reads and Phantom Anomalies

It's common for simple textbook presentations of transaction theory to represent the execution as a sequence of operations each being a read or write on a named data item. This model is not a good reflection of the reality of application programs written with set-oriented query languages like SQL. Furthermore a theory which treats all operations as being on named items can seem to justify incorrect concurrency control mechanisms (such as two-phase locking applied to the records which are selected, updated, inserted or deleted). A particular issue is the possibility of phantoms, where a program performs two subsequent queries, each retrieving the records satisfying some condition; with data item locking, it is possible for one query to miss a row satisfying this condition which is about to be inserted by a concurrent transaction, while the later query sees this row after the inserting transaction commits. This is a *nonrepeatable read*, and such a history is obviously not serializable. The classic paper [Eswaran et al. 1976] identified this issue, although the solution proposed there has been found to be unrealistic in practice, and instead most locking-based systems use algorithms which lock index records (or associated information such as the index key) as well as data records (see Section 7.8 of Gray and Reuter [1993]). To allow us to reason properly about programs with set-oriented operations, in this article, we use a formalism in which there are three types of operation a transaction can perform: item read, item write, and "*predicate read*." This section explains our model and how it can represent SQL statements.

We define a predicate read as an operation which identifies a set of items in the database, based on the state of the database. Formally, a predicate read is represented in a history by the operation $PR_i(P)$, or, when we need to also show the return value, by PR_i (P, list of data items), where P is some function which takes a state (a mapping from item names to values) and returns a set of item names. It is expected that the transaction will follow this by later performing item read or item write operations on (some of) the items which were returned in the predicate read.

Let us see how this applies to a SQL SELECT statement

```
SELECT T.col1 FROM T WHERE T.col2 = :x;
```

in a database where each field is regarded as an item. We represent the SELECT statement with, first, a predicate read, which reflects the evaluation of the WHERE clause. This predicate read determines which fields occur in those rows of T that have the given value for their col2 field. The return value of the predicate read will be a list of field names (i.e., a sequence of (rowid, columnid) pairs). The actual retrieval of the target list (T.col1) takes place in successive item reads, each of which reads the col1 value in one of the rows found by the predicate read. Note that with Snapshot Isolation, the state of the database against which T_i's predicate read is evaluated consists of the version of each item which was most recently committed before the start of T_i, or the most recent version produced by T_i if T_i has itself produced a version of that item.

It is important to avoid a common misconception with set-oriented operations: in our model *there is no such thing as a Predicate Write operation* to conflict with a Predicate Read. A SQL operation

```
UPDATE T SET T.col1 = :y WHERE T.col2 = :x;
```

will be modeled as a Predicate Read which returns the relevant fields, followed by a succession of item write operations which modify each returned col1 field to the new value y. The concept of a Predicate Write has not been used since the prototype System R demonstrated that estimating intersections of set-oriented update data item collections with set-oriented read data item collections led to unnecessary conflicts (as explained in Chamberlin et al. [1981], near the end of Section 3). Instead, in our model, a predicate read can conflict only with an item write *which changes the state of the database in such a way as to change the list of items returned by the predicate read*. For example, the WHERE clause conflicts with any write that produces a new version of some col2 field, if the new version has value x and the old version did not, or if the new version has a different value while the previous version was equal to x.

In Snapshot Isolation, the traditional phantom examples of Eswaran et al. [1976] mentioned above (the nonrepeatable read anomaly) cannot arise, because repeated evaluation of a WHERE clause within an SI transaction always returns the same set of items (as the evaluation is always based on the state at the start of the transaction). However, a somewhat more complex anomaly can occur involving predicate reads, as shown in the following example.

Example 1.4 (*Predicate-Based Write Skew*). Consider a database with the following tables: an **employees** table with primary key eid, a **projects** table with primary key projid, and an **assignments** table with columns (eid, projid, workdate, hours). We assume employees are assigned a certain number of hours of work on a given project during a given workdate by placing a data item in the **assignments** table, and the program to do this maintains a constraint that no employee can be assigned more than eight hours of work during any given day. Consider the following SI history where two different transactions concurrently assign an employee a new assignment for the same day. (The predicate P below specifies: eid = 'e1234' and workdate = '09/22/03'; all operations of the history are on the **assignments** table.)

$$\begin{aligned} H_4: PR_1(P, \text{empty})\ & W_1(Y_1, \text{eid} = \text{`e1234'}, \text{projid} = 2, \\ & \qquad \text{workdate} = \text{`09/22/02'}, \text{hours} = 5) \\ PR_2(P, \text{empty})\ & W_2(\text{Z}_2, \text{eid} = \text{`e1234'}, \text{projid} = 3, \\ & \qquad \text{workdate} = \text{`09/22/02'}, \text{hours} = 5)\ C_1\ C_2. \end{aligned}$$

Transaction T_1 evaluates the predicate P and finds no data item in the **assignments** table to retrieve for the given eid and workdate, and then inserts a 5-hour **assignments** data item Y_1. T_2 does exactly the same thing as T_1, seeing the same snapshot (since T_1 has not committed when T_2 started), and inserting another 5-hour **assignments** data item Z_2. The result is that 10 hours of work have been assigned to the employee with two **assignments** data items, breaking the constraint of no more than eight hours assigned although both transactions acted appropriately in isolation. This is clearly a form of Write Skew, but it is different in kind from that of Example 1.2, since it is the two inserts that cause conflicts with the other transaction's Predicate Read that are responsible for the anomaly. It is clearly different since it is not possible to avoid this anomaly by performing Select For Update on any set of data items involved. However, we ***can avoid*** the anomaly by creating a table **total_work_hours**, with a data item for each employee-day pair, and keeping a total_work_hours column up to date as each new **assignments** data item is inserted. With this design, T_1 and T_2 would collide on a common total_work_hours data item under First Committer Wins.

We remark that even though we call the set-oriented operation a "predicate" read, we do not limit ourselves by assuming that it evaluates a traditional logic predicate (that is, we do not insist that it returns exactly those items for which a function, from item values to Booleans, is true). We allow the predicate read to evaluate an arbitrary function from database states to item sets. For example, a predicate read can model the determination of all the fields in those records where the salary is greatest, corresponding to the SQL WHERE clause

WHERE T.salary = (select max(T1.salary) from T as $T1$);

1.3 Outline of the Article and Related Work

Outline of the Article. In Section 2, we define a number of transactional dependencies (conflict types) that apply in versioned histories, and define the Dependency Serialization Graph of a history H, DSG(H), to investigate how serialization anomalies can arise in SI histories. In Section 3, we define relationships between application transaction programs A, and define a Static Dependency Graph, SDG(A), to show how cycles in a DSG(H) can arise out of an application. Our main theorem gives a condition on SDG(A) that implies all executions of the application are serializable under Snapshot Isolation. In Section 4, we apply our results to analyze the TPC-C application and show that it is serializable under SI. In Section 5, we consider how to modify general application programs to avoid possible violations of serializability. Finally, Section 6 summarizes our results.

Related Work. A large number of concurrency control algorithms have been published, but little attention has been paid to the issue of designing applications so as to preserve data integrity even when run under isolation levels that sometimes permit non-serializable executions. All major database systems are delivered with lower levels of isolation as a default, implicitly requiring DBA's and programmers to "be careful" that their application programs don't lead to isolation errors, but little or no guidance is provided in most database system documentation as to how such applications should be designed. This is exactly the situation we are attempting to remedy in the case of Snapshot Isolation.

One area that has received attention in this regard concerns data replication, where lazy propagation of updates to replicas can violate correctness. Several papers [Gray et al. 1996; Anderson et al. 1998; Breitbart et al. 1999] have shown that serializability can be guaranteed, provided appropriate conditions hold on the pattern of access to items from different sites.

Another aspect of the problem of correct use of not necessarily correct isolation concerns "chopping" transaction programs into smaller transactions (i.e., placing intermediate commit points in the logic). In general, chopping a program into smaller transactional pieces can allow additional concurrent interleaving. This can compromise serializability, but Shasha et al. [1995] derives a condition on a special type of conflict graph, called the *SC-graph*, which ensures that the chopped application executes as serializably as the original without sacrificing improved concurrency. Some elaborations and corrections are described in Shasha and Bonnet [2002].

As for work on Snapshot Isolation, Fekete [1999] presented some preliminary results to the current article, deriving a rather restrictive condition on applications, which ensured correct execution when run under Snapshot Isolation. The condition in Fekete [1999] is generalized by the current article. Also new here is guidance on how to alter an application to prevent nonserializable executions, which could otherwise occur.

Bernstein et al. [2000] provided a theory that can be used to show that certain applications preserve data integrity constraints when executing under weak isolation levels, including Snapshot Isolation. Bernstein et al. [2000] uses explicit knowledge of the data integrity conditions required by each transaction, and it concludes that those specific integrity conditions are preserved. In contrast our work does not require knowledge of the integrity constraints and we show serializability which preserves all integrity conditions. Thus, the results of Bernstein et al. [2000] are complementary with ours; our result applies in fewer circumstances than theirs, but offers a stronger conclusion when it does apply.

Schenkel and Weikum [2000] show how to ensure the serializability of transactions executed against a federation of independent databases some of which provide Snapshot Isolation rather than two-phase locking.

Elnikety et al. [2004] generalize the concept of Snapshot Isolation to include stale caches updated through lazy replication. They show the serializability of applications against this style of system, under conditions on the applications conflicts similar to those in Fekete [1999].

2. TRANSACTIONAL HISTORY THEORY

Throughout this section, we will develop the theory in terms of a model of execution where reads and writes are performed on data items, and where predicate reads return a set of data item identities. In the abstract theory, we do not need to specify any particular granularity of data items. However, in Section 4, when we come to apply the theory to TPC-C, we will take a data item to be a single field within a row, and those readers who like more concrete concepts should feel free to use field granularity for other sections as well.

2.1 Transactional Dependencies in SI

Transactional dependencies can be defined as ordered conflicts that can arise in multiversion histories, cataloged by type; for example, one type relates two different transactions which produce successive versions of the same data item, and a write which produces an item version seen by a later item read is another type. The dependency definitions, which apply to arbitrary multiversion histories, were introduced in Adya et al. [2000], and can also be applied to the various locking isolation levels. Indeed, it was proposed in Adya et al. [2000] that dependencies defined there be used in an ANSI definition of Isolation Levels that would be neutral with respect to the specific concurrency control method.

The dependency definitions from Adya et al. [2000] depend on an ordering on the versions of each data item, i.e., knowing for each version which previous version it replaces. Recall from Section 1.1 that a new version X_m of a data item

X is said to be *installed* in the database only when transaction T_m, which writes the data item version X_m, commits. In the case of SI, which is the only versioned isolation level we consider in this article, the version order is defined by the temporal sequence of the commit times of the transactions that produced the versions. Thus, when we use the phrase "data item version X_n is an *immediate successor* of data item version X_m" in the definitions that follow, we mean that T_m writes X_m and T_n writes X_n and C_m precedes C_n in the history, and also that no transaction that commits between C_m and C_n installs a version of X.

Definition 2.1 (*Transactional Dependencies*). Following the concepts of Adya et al. [2000], but with new terminology, we characterize several types of transactional dependency in an interleaved SI history.

—We say that there is a $T_m \xrightarrow{i-wr} T_n$ dependency (an *item-write-read dependency*) if T_m installs a data item version X_m (of item X) and T_n later reads this version of X.
—We say that there is a $T_m \xrightarrow{i-ww} T_n$ dependency (an *item-write-write dependency*) if T_m installs a data item version X_m and the immediate successor version X_n is installed by T_n.
—We say that there is a $T_m \xrightarrow{i-rw} T_n$ dependency (an *item-read-write dependency* or an *item-anti-dependency*) if T_m reads a version X_k of some data item X, and T_n installs the immediate successor version X_n of X.
—We say that there is a $T_m \xrightarrow{pr-wr} T_n$ dependency (a *predicate-write-read dependency*) if T_m installs a data item version X_m so as to alter the set of items retrieved for a predicate read by T_n, and also the commit of T_m precedes the start of T_n, so that the result of the predicate read by T_n takes into account a version of data item X equal to or later than the one installed by T_m.
—We say that there is a $T_m \xrightarrow{pr-rw} T_n$ dependency (a *predicate-read-write dependency* or *predicate-anti-dependency*) if T_n changes a data item X to version X_n so as to alter the set of items retrieved for a predicate read by T_m, and also the commit of T_n follows the start of T_m, so that the result of the predicate read by T_m takes into account a version of data item X prior to the one installed by T_n.

Please note that in all definitions above with form $T_m \xrightarrow{x-yz} T_n$, the operations '*yz*' include a y operation in T_m (y is r or w) and a conflicting z operation in T_n (z is w or r). If the letter '*x*' in '*x-yz*' is an '*i*', this means that both operations, y and z are on data items, but if x is '*pr*', this means that there is an '*r*' in '*yz*', representing a predicate read and the other operation in '*yz*' is '*w*' for a data item write that conflicts with it; no other operations are possible for $x =$ '*pr*'.

We define a few more generic dependencies based on these basic ones.

—We say there is a $T_m \xrightarrow{wr} T_n$ dependency (a *wr dependency*) if $T_m \xrightarrow{i-wr} T_n$ or $T_m \xrightarrow{pr-wr} T_n$.
—We say there is a $T_m \xrightarrow{ww} T_n$ dependency (a *ww dependency*) if $T_m \xrightarrow{i-ww} T_n$; that is, a write-write dependency must be an item-write-write-dependency (there are no predicate writes), so the two terms are synonymous.

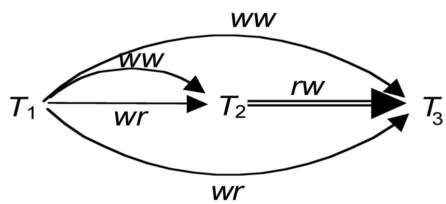


Fig. 1. DSG(H_1).

—We say there is a $T_m \xrightarrow{rw} T_n$ dependency (a *rw dependency* or an *anti-dependency*) if $T_m \xrightarrow{i-rw} T_n$ or $T_m \xrightarrow{pr-rw} T_n$.

—We say there is a $T_m \rightarrow T_n$ dependency (an *arbitrary dependency*) if any of the dependencies above hold: $T_m \xrightarrow{wr} T_n$, $T_m \xrightarrow{ww} T_n$, or $T_m \xrightarrow{rw} T_n$.

We now define *DSG*(*H*), the *Dependency Serialization Graph* for a history H, a counterpart to the Serialization Graph definition of Bernstein et al. [1987], but where the edges of the DSG are labeled to indicate which dependencies occur.

Definition 2.2 (*DSG*(*H*)). A directed graph DSG(H) is defined on a multi-version history H, with *vertices* (often called *nodes*) representing transactions that commit, and each distinctly labeled *edge* from T_m to T_n corresponding to a $T_m \xrightarrow{wr} T_n$, $T_m \xrightarrow{ww} T_n$, or $T_m \xrightarrow{rw} T_n$ dependency.

Example 2.1. Consider the SI history H_1:$W_1(X_1)\ W_1(Y_1)\ W_1(Z_1)\ C_1\ W_3(X_3)\ R_2(X_1)\ W_2(Y_2)\ C_2\ R_3(Y_1)\ C_3$

In the SI history H_1, the versions X_1, Y_1, and Z_1 are installed at time commit(T_1) and version Y_2 is the immediate successor of Y_1 and is installed at time commit(T_2) (thus we have $T_1 \xrightarrow{ww} T_2$). Similarly, X_3 is the immediate successor of X_1 and is installed at time commit(T_3), so $T_1 \xrightarrow{ww} T_3$. The operation $R_2(X_1)$ means $T_1 \xrightarrow{wr} T_2$ (note that although $R_2(X)$ occurs after $W_3(X_3)$ in H_1, there is no $T_3 \xrightarrow{wr} T_2$ because X_3 has not been *installed* when $R_2(X)$ takes place, so we have instead $R_2(X_1)$ and $T_1 \xrightarrow{wr} T_2$; a more intuitive way of putting this is that T_2 and T_3 are concurrent, so T_2 cannot read writes by T_3. The operation $R_3(Y_1)$ means $T_1 \xrightarrow{wr} T_3$. Finally, there is a $T_2 \xrightarrow{rw} T_3$ dependency because the version X_3 succeeds X_1, and there is an operation $R_2(X_1)$; note that this is so even though $W_3(X_3)$ occurs prior to $R_2(X_1)$ in the temporal order. Figure 1 shows DSG(H_1). The H_1 transactions are serializable in order: $T_1\ T_2\ T_3$.

Note that the rw edge from T_2 to T_3 in Figure 1 is drawn as a "double line" edge. This is a convention we will use to indicate that a *rw* dependency (an anti-dependency) connects two concurrent transactions (clear because $W_3(X_3)$ and $R_3(Y_1)$ are separated by operations of T_2). An anti-dependency connecting concurrent transactions will turn out to have special significance in the analysis that follows. When the transactions connected by a dependency are not concurrent, the edge (including a rw edge) in the DSG would be drawn as a "single line" edge.

2.2 Theory of Snapshot Isolation Anomalies

In what follows, we wish to characterize SI histories that contain transactional anomalies. We will see below that serializability of any SI history H is implied by the acyclicity of its DSG(H). Of course, a cycle in a directed graph must be directed in the order of the directed edges. We will demonstrate that for an SI history H, if H is nonserializable, we can conclude that DSG(H) contains a cycle of a very particular form. We start with a number of remarks and lemmas.

Remark 2.1. Since the result of any read by T_n is taken from the snapshot as of start(T_n), and that snapshot contains only the versions produced by transactions that commit before this time, it is clear that if there is a $T_m \xrightarrow{wr} T_n$ dependency then T_m must completely precede T_n, that is, T_m must commit before T_n starts so the transactions cannot be concurrent. Similarly, if there is a $T_m \xrightarrow{ww} T_n$, then because of the First Committer Wins rule, T_m and T_n cannot be concurrent, and again T_m must commit before T_n starts. If there is an anti-dependency $T_m \xrightarrow{rw} T_n$ then it is possible for T_m and T_n to be concurrent, or for T_m to completely precede T_n, but we now argue that it is not possible for T_n to completely precede T_m. The existence of the anti-dependency $T_m \xrightarrow{rw} T_n$ means that T_m reads a version X_k of an item X which precedes the version X_n installed by T_n; but the algorithm for SI requires T_m the most recent version whose commit precedes the start of T_m (or in rereads to reread its own version); thus if the commit of T_n preceded the start of T_m, then the version read by T_m would necessarily be equal to or later than X_n in the version order, so the anti-dependency $T_m \xrightarrow{rw} T_n$ could not exist.

LEMMA 2.2. *In a history executed under SI, if there is a $T_m \rightarrow T_n$ dependency, then T_m starts before T_n commits.*

PROOF. By Remark 2.1, for wr and ww dependencies, T_m commits before T_n starts, which is certainly implies that T_m starts before T_n commits, while for an *rw*-dependency, either T_m precedes T_n or else T_m is concurrent with T_n, but in either situation T_m starts before T_n commits. □

LEMMA 2.3. *In a history executed under SI, if we know that there is some $T_m \rightarrow T_n$ dependency, and that T_m and T_n are concurrent, we can conclude that $T_m \xrightarrow{rw} T_n$. Recall that this is called an anti-dependency.*

PROOF. By Remark 2.1, none of the other dependency types can occur between concurrent transactions. □

In Adya et al. [2000], a multiversion history H was shown to exhibit Isolation Level PL-3 (serializability) if the following two conditions held: (1) no transaction T_n reads transactional updates of T_m that were later aborted or modified again by T_m, and (2) DSG(H) did not contain any cycles of dependencies of any of the types described above: *wr*, *ww*, and *rw*, with at least one anti-dependency edge, *rw*. We note condition (1) can never occur in Snapshot Isolation, and we will show below in Theorem 2.1 that any cycle in a DSG must contain a structure having at least two anti-dependency edges each of which is between concurrent transactions.

THEOREM 2.1. *Suppose H is a multiversion history produced under Snapshot Isolation that is not serializable. Then there is at least one cycle in the serialization graph DSG(H), and we claim that in every cycle there are three consecutive transactions $T_{i.1}$, $T_{i..2}$, $T_{i.3}$ (where it is possible that $T_{i.1}$ and $T_{i.3}$ are the same transaction) such that $T_{i.1}$ and $T_{i.2}$ are concurrent, with an edge $T_{i.1} \rightarrow T_{i.2}$, and $T_{i.2}$ and $T_{i.3}$ are concurrent with an edge $T_{i.2} \rightarrow T_{i.3}$.*

Before we proceed with the proof, we make a few remarks. By Lemma 2.3, both concurrent edges whose existence is asserted must be anti-dependencies: $T_{i.1} \xrightarrow{rw} T_{i.2}$ and $T_{i.2} \xrightarrow{rw} T_{i.3}$. Example 2.2 in the next subsection shows that $T_{i.1}$ and $T_{i.3}$ can be the same transaction; Example 2.3 shows that $T_{i.1}$ might be a read-only transaction as long as $T_{i.2}$ and $T_{i.3}$ are update transactions.

PROOF. By Adya et al. [2000], a cycle exists in DSG(H). Take an arbitrary cycle in DSG(H), and let $T_{i.3}$ be the transaction in the cycle with the earliest commit time; let $T_{i.2}$ be the predecessor of $T_{i.3}$ in the cycle, and let $T_{i.1}$ be the predecessor of $T_{i.2}$ in the cycle. Suppose for the sake of contradiction that $T_{i.2}$ and $T_{i.3}$ are not concurrent: then either $T_{i.2}$ finishes before $T_{i.3}$ starts (but then this is before $T_{i.3}$ commits, contradicting the choice of $T_{i.3}$ as the earliest committed transaction in the cycle), or $T_{i.3}$ finishes before $T_{i.2}$ starts (but this contradicts the presence of an edge $T_{i.2} \rightarrow T_{i.3}$ in DSG(H) by Lemma 2.2.). Thus we have shown that $T_{i.2}$ must be concurrent with $T_{i.3}$, and therefore the edge from $T_{i.2}$ to $T_{i.3}$ is an anti-dependency, $T_{i.2} \xrightarrow{rw} T_{i.3}$. Because $T_{i.2}$ and $T_{i.3}$ are concurrent, the start of $T_{i.2}$ precedes the commit of $T_{i.3}$ and the start of $T_{i.3}$ precedes the commit of $T_{i.2}$.

Now suppose for the sake of contradiction that $T_{i.1}$ was not concurrent with $T_{i.2}$. Then either the commit of $T_{i.1}$ precedes the start of $T_{i.2}$ (which by the last sentence of the prior paragraph precedes the commit of $T_{i.3}$, so the commit of $T_{i.1}$ precedes the commit of $T_{i.3}$ contradicting the choice of $T_{i.3}$ as earliest committed transaction in the cycle), or the commit of $T_{i.2}$ precedes the start of $T_{i.1}$ (contradicting the presence of an edge $T_{i.1} \rightarrow T_{i.2}$ in DSG(H)). Thus we have shown that $T_{i.1}$ is concurrent with $T_{i.2}$, and so the edge from $T_{i.1}$ to $T_{i.2}$ must be an anti-dependency, $T_{i.1} \xrightarrow{rw} T_{i.2}$. □

2.3 SI-RW Diagrams

Theorem 2.1 is a fundamental result we use to characterize how SI serialization anomalies can arise. However, many readers have not had much experience with multiversion histories, so in this section, we offer an alternative way to think about Snapshot Isolation within the framework of traditional single-version histories. We note that the multiversion histories we have been looking at up to now give the temporal order in which operations such as R_i, W_j, C_k, etc. are requested by users and passed through by the scheduler to return values read and accept new versions to be written, inserted, etc. Thus, in this section, we will refer to these as *user-oriented histories*.

In contrast, a *scheduler-oriented history* is formally defined as one where each transaction's operations occur in two blocks: all the reads take place in one uninterrupted block, and all the writes take place in a later uninterrupted block; furthermore, each read sees the version produced in the nearest preceding write to the same item (and so the history is essentially a single-version history). Given any user-oriented history allowed by the SI algorithm, we can produce a scheduler-oriented history by treating all data item reads of a transaction as if they take place at the single instant of time when the transaction starts, and placing all writes at the single instant when (and if) the transaction commits. This model is appropriate only so long as no transaction rereads data that it had previously written, but such reads are redundant in any event (we can assume the transaction simply remembers what it wrote last) so we ignore them.

For example, recall the SI history presented in Example 2.1 (a user-oriented history):

$$H_1\colon W_1(X_1)\ W_1(Y_1)\ W_1(Z_1)\ C_1 W_3(X_3)\ R_2(X_1)\ W_2(Y_2)\ C_2 R_3(Y_1)\ C_3.$$

We note $W_3(X_3)$ precedes $R_2(X_1)$, implying this is a multiversion history. But here is the scheduler-oriented equivalent:

$$H_1'\colon W_1(X_1)\ W_1(Y_1)\ W_1(Z_1)\ C_1\ R_3(Y_1)\ R_2(X_1)\ W_2(Y_2)\ C_2\ W_3(X_3)\ C_3.$$

In the history H_1', $W_3(X_3)$ takes place after $R_2(X_1)$, and indeed the entire SI history requires only a single value of any data item at any time. Note that the reads of T_3 take place prior to the reads of T_2 in H_1' and the writes of T_3 take place after the writes of T_2; thus, the two transactions are concurrent, but there is no conflict so this is fine.

Now we wish to illustrate how cycles can occur in a scheduler-oriented representation of a SI history; this will provide intuition for the meaning of Theorem 2.1.

In a general non-SI single-version scheduler without any concurrency control that delays user R and W requests, reads and writes of various transactions can take place at arbitrary times between transactional starts and commits; in this case, the user-oriented history matches the scheduler-oriented history (since there is no concurrency control), and reads of one transaction T_2 can access data item values written by another transaction T_1 before T_1 has committed. This makes it extremely easy for cycles to occur in a Serialization graph for a history arising in such a scheduler. Consider the user-oriented (but not SI) history H_{SV1}, an inconsistent analysis caused by a read by T_1 prior to a money transfer by T_2 and another read by T_1 after the money transfer has been accomplished.

$$H_{\mathrm{SV1}}\colon R_1(A_0, 50)\ R_2(A_0, 50)\ W_2(A_2, 70)\ R_2(B_0, 50)\ W_2(B_2, 30)\ R_1(B_2, 30)\ C_1 C_2.$$

Such a cycle as this clearly cannot happen in a scheduler-oriented SI history, since all reads of T_1 would occur at one instant of time. In this case, the history would become:

$$H_{SI}\colon R_1(A_0, 50)\ R_1(B_0, 50)\ R_2(A_0, 50)\ R_2(B_0, 50)\ W_2(A_2, 70)\ W_2(B_2, 30)\ C_1 C_2.$$

There's no reason that C_1 couldn't occur right after the second read, since T_1 has no writes, but this fact makes it clear that T_1 can't conflict with T_2. Clearly,

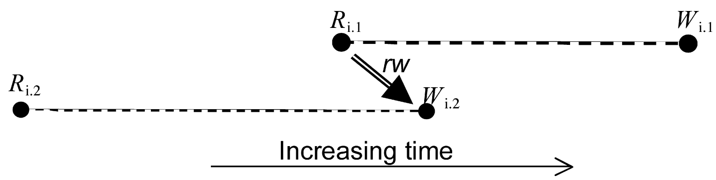


Fig. 2. An SI-RW diagram with a *rw* dependency going back in time.

the reads of T_1 cannot occur after one write by T_2 and before another, because both writes take place at an instant of time.

For a picture of how to devise cycles in a scheduler-oriented SI history, a much more difficult undertaking, we start by defining what we call an *SI-RW diagram* for a scheduler-based history; in an SI-RW diagram, we represent all reads of transaction T_i to occur at an instant of time at a vertex R_i (a pseudonym for start(T_i)), and all writes at an instant of time at a vertex W_i (a pseudonym for C_i); vertex R_i appears to the left of vertex W_i (time increases from left to right), with the two connected by a horizontal dotted line segment. See Figure 2 for an example. The SI-RW diagram is based on a structure known as an SC-Graph defined in Transaction Chopping [Shasha et al. 1995; Shasha and Bonnet 2002], with two types of edges: *Sibling edges* and *Conflict edges*; in our SI-RW diagrams, the only sibling edges are the dotted line segment connecting the R_i and W_i vertices for any transaction T_i that has both Reads and Writes. The *time* at which an SI transaction occurs can be arbitrarily defined at any point between R_i and W_i, but for purposes of this analysis we will say the transaction T_i *occurs at time* W_i (i.e., at commit time C_i).

Now when we consider the three types of dependency, $T_m \xrightarrow{wr} T_n$, $T_m \xrightarrow{ww} T_n$ and $T_m \xrightarrow{rw} T_n$, we note that the *wr* and *ww* dependencies always point in the direction such that T_n occurs after T_m: if $T_m \xrightarrow{ww} T_n$, W_m must be executed before W_n, while if $T_m \xrightarrow{wr} T_n$, W_m must be executed before R_n, and therefore before W_n. But any cycle among transactions must somehow "go back in time" at some point, which cannot occur in a history with only *wr* and ww dependencies. A *rw* dependency $T_m \xrightarrow{rw} T_n$ allows W_n to occur *before* W_m when the two transactions are concurrent. See Figure 2, where $T_{i.1}$ is T_m and $T_{1.2}$ is T_n; clearly $T_{i.1} \xrightarrow{rw} T_{i.2}$, but $T_{i.2}$ occurs before $T_{i.1}$. Of course Figure 2 does not illustrate a cycle, since $T_{i.1}$ and $T_{i.2}$ are concurrent. However, a sequence of two successive $T_m \xrightarrow{rw} T_n$ dependencies can complete such a cycle, as we see in Figure 3.

The cycle we see in Figure 3 is along the double-line rw dependency from $R_{i.1}$ to $W_{i.2}$, back in time along the sibling edge from $W_{i.2}$ to $R_{i.2}$, then along the double-line rw dependency from $R_{i.2}$ to $W_{i.3}$, and finally along the *wr* dependency from $W_{i.3}$ to $R_{i.1}$. This exemplifies the cycle that was described in the proof of Theorem 2.1. All dependency edges are directed, and any cycle must follow the direction of the directed edges. The Sibling edges on the other hand are undirected, and a cycle representing an anomaly can traverse a sibling edge in either direction. We note that the sibling edge separation of the R_i and W_i vertices provides an intuitive feel for how a cycle "goes back in time", but doesn't add any power to the DSG(H) diagrams of Figure 1. After providing illustrative

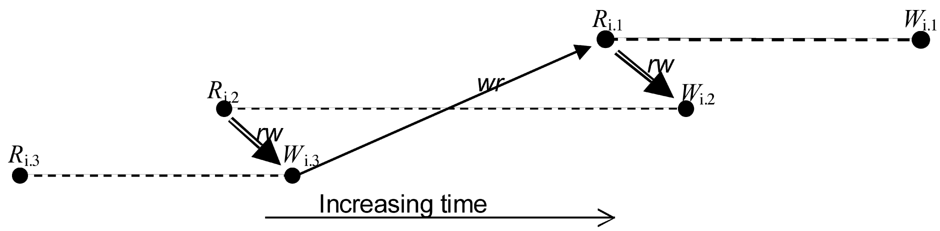


Fig. 3. An SI-RW diagram with two successive rw dependencies and a cycle.

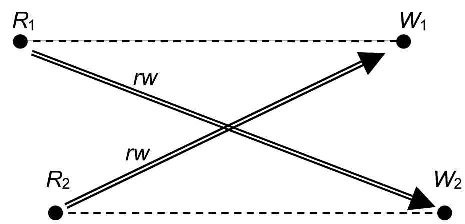


Fig. 4. An SI-RW diagram for H_2 in Example 2.2.

SI-RW diagrams for a number of anomaly examples, we will return to using DSG(H) diagrams.

Example 2.2. Recall Example 1.2 that gives the SI Write Skew anomaly, with the history:

$$H_2: R_1(X_0, 70)\, R_2(X_0, 70)\, R_1(Y_0, 80)\, R_2(Y_0, 80)\, W_1(X_1, -30)\, C_1 W_2(Y_2, -20)\, C_2.$$

H_2 is clearly not serializable since the constraint that $X + Y > 0$ is broken by two transactions that act correctly in isolation; thus we should be able to exhibit successive transactions $T_{i.1}$, $T_{i.2}$, $T_{i.3}$ in a cycle of DSG(H_2) with the properties specified in Theorem 2.1. We note there are only two transactions in H_2, but Theorem 2.1 allows that $T_{i.1}$ and $T_{i.3}$ can be the same transaction, and we demonstrate this here: thus we need a cycle $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_1$ to satisfy Theorem 2.1. Now Transactions T_1 and T_2 are clearly concurrent. Furthermore, DSG(H_2) has an anti-dependency edge $T_1 \xrightarrow{rw} T_2$ because of the two operations $R_1(Y_0, 80)$ and $W_2(Y_2, -20)$, and also an anti-dependency edge $T_2 \xrightarrow{rw} T_1$ because of the two operations $R_2(X_0,70)$ and $W_1(X_1, -30)$. Thus, this cycle, shown in Figure 4, satisfies the Theorem 2.1 conditions.

Example 2.3. Recall Example 1.3 that gives an SI Read-Only serialization anomaly, with the history:

$$H_3: R_2(X_0, 0)\; R_2(Y_0, 0)\; R_1(Y_0, 0)\; W_1(Y_1, 20)$$
$$C_1\; R_3(X_0, 0)\; R_3(Y_1, 20)\; C_3 W_2(X_2, -11)\; C_2.$$

T_3 is a read-only transaction that reads and prints out the values $X = 0$ and $Y = 20$, which could not happen in any serializable execution where X ends with a value -11, meaning that a withdrawal of 10 was made from X prior to the deposit of 20 being registered in Y so that a charge was levied. Therefore

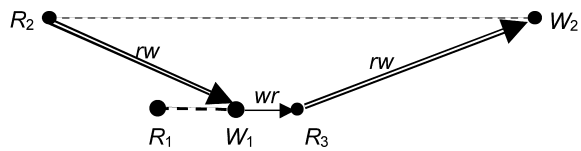


Fig. 5. An SI-RW diagram for H_3 in Example 2.3.

H_3 is not serializable, and there must be three successive transactions $T_{i.1}$, $T_{i.2}$, $T_{i.3}$ in a cycle of DSG(H_3) with the properties specified in Theorem 2.1. Examining H_3, we derive the following DSG(H_3) edges: (1) $T_2 \xrightarrow{rw} T_1$, because of operations $R_2(Y_0, 0)$ and $W_1(Y_1, 20)$; (2) $T_3 \xrightarrow{rw} T_2$ because of operations $R_3(X_0, 0)$ and $W_2(X_2, -11)$; (3) $T_1 \xrightarrow{wr} T_3$ because of operations $W_1(Y_1, 20)$ and $R_3(Y_1, 20)$. Thus, the cycle in H is: $T_1 \xrightarrow{wr} T_3 \xrightarrow{rw} T_2 \xrightarrow{rw} T_1$. By the proof of Theorem 2.1, we recognize T_1 as the first committed transaction in the cycle, so T_1 is $T_{i.3}$; the predecessor of T_1 in the cycle is T_2, so T_2 is $T_{i.2}$; the predecessor of T_2 is T_3, so T_3 is $T_{i.1}$. (Recall we noted after the statement of Theorem 2.1 that T might be a read-only transaction, and this Example demonstrates this.) Examining the cycle we see, as required by Theorem 2.1: $T_{i.1} \xrightarrow{rw} T_{i.3}$ ($i.1$ is 3, $i.2$ is 2), and $T_{i.2} \xrightarrow{rw} T_{i.3}$ ($i.2$ is 2, $i.3$ is 1). The SI-RW diagram for H_3 is given in Figure 5. Note that R_3 has no sibling node W_3 in Figure 5 since T_3 is read-only.

Examples 2.2 and 2.3 demonstrate that SI Write Skew and the SI Read-Only serialization anomaly are both characterized in the same way, by Theorem 2.1. We leave it to the reader to verify that predicate-based write-skew in Example 1.5 is also characterized by Theorem 2.1. It is also easy to find examples where multiple transactions are found in the cycle, and only three successive transactions $T_{i.1}$, $T_{i.2}$, $T_{i.3}$ in the cycle have the properties of Theorem 2.1.

The following result shows how the transactions must be placed in time, when they lie in a cycle in the dependency graph.

COROLLARY 2.4. *Consider a cycle starting with* $T_{i.1} \xrightarrow{rw} T_{i.2} \xrightarrow{rw} T_{i.3}$ *as defined in Theorem* 2.1*, and a sequence of* n *transactions* $T_{m.1}$ *to* $T_{m.n}$ *completing the cycle with dependencies wr or ww, represented below as wx:*

$$T_{i.1} \xrightarrow{rw} T_{i.2} \xrightarrow{rw} T_{i.3} \xrightarrow{wx} T_{m.1} \xrightarrow{wx} T_{m.2} \xrightarrow{wx} \cdots T_{m.n} \xrightarrow{wx} T_{i.1}.$$

Then each $T_{m.i}$ runs fully within the lifetime of $T_{i.2}$.

PROOF. $T_{m.1}$ must start after $T_{i.3}$ commits and $T_{m.n}$ must commit before $T_{i.1}$ starts because of the sequence of wx dependencies (by Remark 2.1.) Similarly, the $T_{m.i}$'s are all disjoint from each other. Thus, $T_{i.3}$'s write precedes $T_{m.1}$'s read and $T_{i.1}$'s read follows $T_{m.n}$'s write. $T_{i.2}$'s read must precede $T_{i.3}$'s write since $T_{i.2} \xrightarrow{rw} T_{i.3}$, so $T_{i.2}$'s read precedes $T_{m.1}$'s read. Similarly, $T_{i.1}$'s read precedes $T_{i.2}$'s write, so $T_{m.n}$'s write precedes $T_{i.2}$'s write. □

Since each $T_{m.i}$ runs within $T_{i.2}$'s lifetime, they cannot write any data in common with $T_{i.2}$. Figure 6 provides an SI-RW diagram that illustrates Lemma 2.4 with only two intervening transactions $T_{m.1}$ and $T_{m.2}$, that is with $n = 2$.

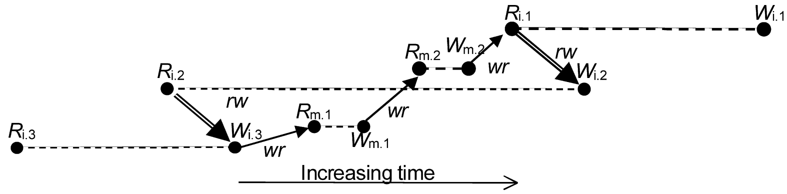


Fig. 6. An SI-RW diagram representing Theorem 2.1 with n = 2.

Of course, there is no implicit limitation in an SI history that there be only one sequence of three successive transactions with *rw* dependencies that goes back in time. There might be multiple such *rw* dependencies in sequence or several separated sequences of two or more *rw* dependencies.

3. PROGRAM CONFLICTS AND STATIC DEPENDENCIES

In this section, we explore how to use a *static analysis* of a set of transactional programs $P_1, P_2, \ldots, P_k$ in an application A, to allow a DBA to run the application under a DBMS using Snapshot Isolation with confidence that no serialization anomaly will arise. In other words, we wish to know whether or not it is the case that every possible interleaved execution of the transaction programs under SI is in fact serializable. By Theorem 2.1, we can determine this if we can decide whether or not some execution of these programs can result in a history H that has a cycle in DSG(H). Of course for a given application, there might be some executions with cycles and other executions without cycles; however, it is the *potential* for such a cycle in some history arising out of the application A that will require us to perform modification of program logic to avoid serialization anomalies during execution.

Definition 3.1 (*A Serializable Application*). We say that *an application A* is *serializable* if every history H arising out of executions of the programs of A is a serializable history.

In this section of the article, we will simply take it as given that the DBA can determine the possibility of dependencies between every pair of transaction programs, by looking at the program logic and using his/her understanding of their operations. In general, the program logic could be arbitrarily complicated, and it is not known how to identify dependencies automatically; however, in Section 4, we will show by example how such a static analysis can be arrived at for the TPC-C programs; we will also suggest ways to perform similar analysis with other applications.

Definition 3.2 (*Static Dependencies*). For each of the transactional dependencies $T_m \stackrel{x-yz}{\longrightarrow} T_n$ in Definition 2.1, there is a corresponding definition for a static dependency. We say that there is a static dependency $P_m \stackrel{x-yz}{\longrightarrow} P_n$ between a transactional program P_n and a transactional program P_m when Snapshot Isolation allows for the existence of an interleaved multiversion history h containing a transaction T_m, which arises from running P_m, and a transaction T_n which arises from running program P_n, such that $T_m \stackrel{x-yz}{\longrightarrow} T_n$. We say that the static dependency is *vulnerable* if there exists a history which has the properties above and in which T_m and T_n are concurrent. We represent a vulnerable static dependency from P_m to P_n by $P_m \Rightarrow P_n$.

We make some observations about these concepts. Firstly, diagrams of Section 2 show vulnerable rw dependencies with double-line edges. Lemma 2.3 implies that the only vulnerable static dependencies are *rw*-dependencies; however there can exist non-vulnerable static *rw*-dependencies (e.g., in Section 4.2 we will see some nonvulnerable *rw*-dependencies when transactions write a common item in every execution where a *rw*-dependency occurs). Also, it is important to be aware that there can be a static dependency from a transactional program to itself, even though no transaction can directly depend on itself. This happens because a history can contain several transactions, which all arise from the same application program, and two of these transactions can have a *rw*-dependency.

It is usually found that wherever there is a static dependency $P_m \xrightarrow{i-rw} P_n$ there will also be another static dependency $P_n \xrightarrow{i-wr} P_m$. However, this is not always the case, since the program P_m can contain logic that leads it to abort whenever it sees the data produced by P_n.

Definition 3.3 (*SDG(A)*). The graph SDG(A), a Static Dependency Graph defined on an application A, has vertices representing programs $P_1, \ldots, P_k$ of A, and distinctly labeled edges $P_m \xrightarrow{wr} P_n$, $P_m \xrightarrow{ww} P_n$, and $P_m \xrightarrow{rw} P_n$ (non-vulnerable) or $P_m \Rightarrow P_n$ (vulnerable) representing static dependencies. Note that usually a *ww* edge from P_m to P_n is accompanied by a reverse *ww* edge from P_n to P_m, and when that happens we draw an undirected edge to represent the pair of directed ones.

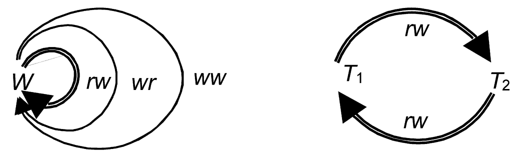
In the rest of this section, we investigate how conditions on the static dependency graph SDG(A) can be used to guarantee that the application *A* is serializable.

3.1 Lifting Paths from SDG(A) to DSG(H)

Definition 3.4 (*Graph Theory Definitions*). Consider the following graph theory definitions found in [Berge 1976]. A *path* on a graph G (with edges either directed or undirected) is an alternating sequence of vertices v_i and edges e_j:

$$v_0, e_1, v_1, e_2, \ldots, e_n, v_n,$$

beginning and ending with a vertex, such that each edge is immediately preceded and followed by the two vertices that lie on it; directed edges must be *incident from* the vertex that precedes it and *incident to* the vertex that follows it. We refer to the number of edges in the sequence, n, as the *length* of the path. A *simple path* is a path whose edges are all distinct. A *circuit* is a path where $v_0 = v_n$, and where the edges are all distinct. An edge with identical initial and terminal vertices is called a *loop*. Note that a loop forms a circuit of length 1. An *elementary circuit* is a circuit whose vertices are distinct except for the beginning vertex and the ending vertex. (Other mathematics references give essentially the same definitions with slightly different names.)



(a) (b)

Fig. 7. (a) SDG(A) and (b) DSG(H) for Example 3.1.

When we speak of a *cycle* in a digraph, we follow traditional database use which differs from the convention in mathematics and in Berge [1976]; for us, a cycle has neither vertices nor edges that are repeated (except for $v_0 = v_n$), so the term *cycle* as we use it is what is defined above as an *elementary circuit*. We say that a digraph is *acyclic* if it has no cycles.

Consider a history H that occurs as a result of some execution of an application A. Now looking at the succession of transactional vertices and directed edge dependencies that make up a path in DSG(H), it is clear that each transactional vertex can be mapped uniquely to the corresponding program that executes the transaction, and each transactional dependency can be mapped to the corresponding static dependency in SDG(A). Picturing the application A and SDG(A) to exist at a higher level schematically than DSG(H), we refer to this mapping as a *lifting* from DSG(H) to SDG(A). For example, we say we can *lift* a path in DSG(H) to a path in SDG(A). Now a cycle in DSG(H) does not necessarily lift to a cycle in SDG(A), but it can always be shown to lift to a *circuit* (not necessarily elementary—so there might be duplicate vertices other than the beginning and ending vertex); the reason is that multiple transactions in DSG(H) might arise as executions of the same program. We give an illustration of how this can happen in Example 3.1.

Example 3.1 (*SI Write Skew*). Recall the Write Skew anomaly given in Example 1.2 with history H_2:

$$H_2\colon R_1(X_0, 70)\, R_2(X_0, 70)\, R_1(Y_0, 80)\, R_2(Y_0, 80)\, W_1(X_1, -30)\, C_1 W_2(Y_2, -20)\, C_2$$

We can picture these transactions arising because of a Withdrawal program that is passed two arguments, an Account identifier A, and a Withdrawal quantity Delta, and then reads Account row A to see if a co-account with identifier B exists with the required constraint that A.bal + B.bal > 0. If so, and the sum of the two balances (which become X and Y in execution) is greater than zero after subtracting Delta, then Transaction T_1, withdrawing Delta = 100 from $A = X$ will act as in H_2, and a concurrent transaction T_2 will also act as in H_2, withdrawing Delta = 100 from $B = Y$. Under these assumptions, these two transactions are concurrent executions of the same program.

If the Withdrawal program W is the only one in the simple application A, then SDG(A) will be as pictured in Figure 7(a), and DSG(H_2) will be as pictured in Figure 7(b).

Note that SDG(A) has three different loop edge static dependencies from the vertex W to itself, the vulnerable $W \Rightarrow W$ edge (which is traversed twice in the path on SDG(A) corresponding to the $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_1$ cycle in Figure 7(b)),

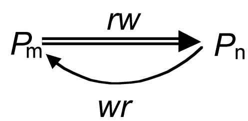


Fig. 8. One cycle in SDG(A) that doesn't translate to a cycle in DSG(H).

the $W \xrightarrow{wr} W$ edge (which might arise, for example, if one execution of W committed before a second execution started), and the $W \xrightarrow{ww} W$ edge (which might arise if two successive executions of W performed a withdrawal from the same account).

The inverse of *lifting* (which might be called a *sinking*) is not always well defined. For example, there is no way we can *sink* the cycle in SDG(A) of the *rw* and *wr* edges Figure 8 to a cycle in DSG(H). This is clear because $T_n \xrightarrow{wr} T_m$ implies that start(T_m) comes after commit(T_n) and the edge $T_m \xrightarrow{rw} T_n$ implies that start(T_m) comes before commit(T_n) (concurrently or not). More generally, Theorem 2.1 states that no cycle in DSG(H) under SI can fail to have two successive concurrent rw anti-dependencies, but the transactional cycle we attempted to sink from Figure 8 does not have two vulnerable anti-dependencies. It turns out that even cycles in an SDG(A) with two successive vulnerable static anti-dependencies will sometimes not successfully sink to a cycle in DSG(H), and the problem becomes complex, so from now, we shall concentrate on what we can learn from studying lifting from DSG(H) to SDG(A).

3.2 Dangerous Structures in SDG(A)

We define a *dangerous structure* that can appear in SDG(A), then demonstrate that if this structure does not appear, then no cycle can occur in DSG(H), for any history H that is an execution of programs in A.

Definition 3.5 (*Dangerous Structures*). We say that the static dependency graph SDG(A) has a *dangerous structure* if it contains nodes P, Q and R (some of which may not be distinct) such that:

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

We can easily see how to detect dangerous structure algorithmically, once the graph is known. For each node P in turn, one can simply scan the incoming edges; whenever one is found to be vulnerable, in a nested loop one scans the outgoing edges; when an outgoing edge is also found to be vulnerable, a simple test of reachability in a digraph can be performed, from the target of the vulnerable outgoing edge to the source of the vulnerable incoming one. If that test shows reachability, then the graph has a dangerous structure.

THEOREM 3.1. *If an application A has a static dependency graph SDG(A) with no dangerous structure, then A is serializable under SI; that is, every history* *resulting from execution of the programs of A running on a database using SI is serializable.*

We stress that the absence of dangerous structures in SDG(A) is only a *sufficient* condition for A to be serializable; the presence of a dangerous structure in SDG(A) does *not* necessarily imply there is a cycle in any history that is an execution of programs in A, so it is not *necessary* that dangerous structure be absent for A to be serializable!)

PROOF. Our proof is given for the contrapositive. We take an arbitrary collection of transaction programs in A which is nonserializable, that is, there is a nonserializable history H resulting from an execution of A under Snapshot Isolation. We show that SDG(A) has a dangerous structure.

Since H is nonserializable, Theorem 2.1 implies there is some cycle in DSG(H) having three consecutive transactions $T_{i.1}$, $T_{i..2}$, $T_{i.3}$ ($T_{i.1}$ and $T_{i.3}$ might be identical) such that $T_{i.1}$ and $T_{i.2}$ are concurrent, with an edge $T_{i.1} \xrightarrow{rw} T_{i.2}$, and $T_{i.2}$ and $T_{i.3}$ are concurrent with an edge $T_{i.2} \xrightarrow{rw} T_{i.3}$. Our plan is to follow the cycle of transactions starting at $T_{i.1}$, and "lift" it to a circuit in SDG(A) which will contain a dangerous structure. When we lift each edge in turn, we get a circuit having three consecutive nodes $P_{i.1}$, $P_{i.2}$, $P_{i.3}$, where $P_{i.1}$ is the program whose execution gives rise to $\mathrm{T}_{i.1}$ etc. We remark that several of the transactions in the cycle in DSG(H) may lift to the same program in SDG(A), and thus the lifting of the cycle may have repeated edges or vertices. That is, the lifting may not be a cycle; however it is certainly a circuit. Now because the execution H contains $T_{i.1}$ and $T_{i.2}$ which are concurrent, with an edge $T_{i.1} \xrightarrow{rw} T_{i.2}$, the definition of SDG shows that the edge $P_{i.1} \Rightarrow P_{i.2}$ is vulnerable. Similarly, because H contains $T_{i.2}$ and $T_{i.3}$ which are concurrent with an edge $T_{i.2} \xrightarrow{rw} T_{i.3}$, the edge $P_{i.2} \Rightarrow P_{i.3}$ is vulnerable. Finally, we note that either the cycle in DSG(H) has length 2, in which case either $T_{i.1}$ equals $T_{i.3}$, so $P_{i.1} = P_{i.3}$, or else the remainder of the cycle gives a path from $T_{i.3}$ to $T_{i.1}$, in which case the lifting of this is a path from $P_{i.3}$ to $P_{i.1}$. Thus, the nodes $P_{i.2}$, $P_{i.3}$, and $P_{i.1}$ (in that order) can take the roles of P, Q and R in the definition of dangerous structure. That is, SDG(A) contains a dangerous structure. □

The following obvious sufficient condition is particularly easy to check in practice, as no cycle-detection is needed.

COROLLARY 3.2. *Suppose an application A has a static dependency graph SDG(A) where, for every vertex v, either there is no vulnerable anti-dependency edge leaving v, or else there is no vulnerable anti-dependency edge entering v. Then A is serializable under SI; that is, every history resulting from execution of the programs of A running on a database using SI is serializable.*

4. ANALYZING AN APPLICATION

In this section, we show how to determine the static dependencies that define the graph SDG(A) for the TPC-C application. At the end of this section, we consider the issues that might arise with other applications. We need the following information to construct an appropriate static dependency graph about an application *A*.

(1) *Relational Table Descriptions*. Descriptions of the tables accessed by the application: we assume we have access to all table schema information typically stored in System Catalogs (i.e., System Views, or Data Dictionaries), specifically table names, column names, and primary keys, as well as descriptions of how the columns are used.
(2) *Program Descriptions*. We assume that the various programs of the application A are described in some form of pseudo-code as in TPC-C; programs with if-then-else structures that have branches with different static dependencies to other program vertices must be split into two or more programs with preconditions, as part of the analysis.

In this section, we analyze the programs of the application manually, looking at pseudo-code descriptions of the programs and their effects on the columns of the accessed data to construct the SDG. One might automate such an analysis, using custom software to read application programs and parse the SQL. However it might be impossible without human understanding of the data to determine whether Where conditions of conflicting SQL statements actually retrieve common row/column data items, so there must be some way for experts to input information to this software process. For example, an expert might be needed to point out that a banking program to insert a new account cannot have a conflict with a scan to find all overdrawn accounts, because a newly inserted account cannot be overdrawn.

The idea of *splitting* programs is that two different branches from an if-then-else construct in a program can be represented as distinct programs, with preconditions that determine which branch is valid. We assume that any accesses that lead up to the branch decision is repeated in each split program, as well as any common actions that take place after the two branches. The preconditions for the split programs can be thought of as *oracles* that determine which distinct static dependencies in the SDG are incident with the distinct program nodes. The motivation for splitting is to avoid dependencies that would appear to impinge on a single program and imply a dangerous structure in the SDG, when in fact some of the dependencies come from two different branches of the logic and cannot appear together in any transactional execution. Once appropriate splits have been made, we redefine the application A to contain the new split programs.

4.1 The TPC-C Benchmark

The TPC-C benchmark was introduced in 1992 to measure the performance of transaction processing systems, superseding the earlier TPC-A and TPC-B benchmarks, which had a very simple debit-credit scenario. The TPC-C benchmark has a rather complex structure, with nine different tables and five different transactional profiles, dealing with most of the aspects of ordering, paying for, and delivering goods from warehouses. The standard specification of the TPC-C benchmark is found at the URL cited at [TPC-C]. Figure 9 lists

Tables

Tbl Abbrev	Table Name
W.	Warehouse
D.	District
C.	Customer
H.	History
NO.	New-Order
O.	Order
OL.	Order-Line
I.	Item
S.	Stock

Transactions

Tx Abbrev	Transaction Name
NEWO	New-Order
PAY	Payment
DLVY	Delivery
OSTAT	Order-Status (RO)
SLEV	Stock-Level (RO)

Fig. 9. Abbreviations and names of tables and transactions in the TPC-C benchmark.

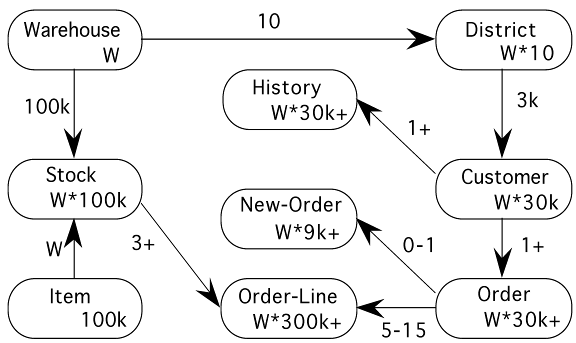


Fig. 10. TPC-C table layout with relationships.

the tables and transactions, both the actual names and abbreviations (listed as Tbl Abbrev and Tx Abbrev) that we use in our analysis.

Note that we list table abbreviations with concluding dots (.), something that is not done in the TPC-C benchmark specification [TPC-C]. In what follows, we also list tablename/columname pairs in the form T.CN; for example, to specify what Warehouse a Customer row is affiliated with (see definition below), we would use C.WID, whereas in [TPC-C] it is represented as C_W_ID. We find that the complexity of names in the TPC-C specification is distracting; the abbreviations we use should be easily understood by readers wishing to tie our discussion to the original.

Figure 10 provides a schema for how the tables of Figure 9 relate to one another. If we assume that a database on which the TPC-C benchmark is run has W Warehouse rows, then some of the other tables have cardinality that is a multiple of W. There are 10 rows in the District table associated with each Warehouse row, and the primary key for District is D.DID D.WID, specifying each district row's parent Warehouse; there are 3000 Customer rows for each District, and the primary key for Customer is C.CID C.DID C.WID, specifying each Customer row's parent District and grandparent Warehouse.

We now describe tables and programs, while discussing the program logic. The reader may wish to refer back to Figure 9 to follow this short (and incomplete) discussion. Full Transaction Profiles for programs are given in [TPC-C].

NEWO Program. An order, represented by a row in the Order table (O), is placed for a customer (a row in the Customers table, C), by a call to the New-Order program (NEWO); in addition to creating (inserting) a row in the Order table O, the NEWO program inserts a row in the New-Order table (NO, which basically *points* to the Order row) and a series of rows in the Order-Line table (OL) that make up the individual stock orders that constitute a full order. (The NO row inserted by NEWO is later deleted when a Delivery program (DLVY) delivers this particular order; the O row is retained.) To accomplish its task, the NEWO program reads and/or updates a number of fields from the Warehouse table (W), District table (D), Customer table (C), Item table (I), and Stock table (S), but it turns out that only two of these accesses have potential conflicts, the one in S to S.QTY, the quantity of a stock item on hand, which NEWO decrements, and the one in D to the column D.NEXT, the next order number for the district, which NEWO uses and increments. We only know about potential conflicts after annotating table accesses by all programs, but we use this future knowledge here to simplify our discussion

PAY Program. The Payment program (PAY) accepts a payment for a specific customer, increments the customer balance (C.BAL) and also reflects this incremental information in the warehouse (W.YTD—payments Year-To-Date) and District (D.YTD), then inserts a record of this payment to the History table H. PAY is the only program to access H. The payment is not associated with a particular order, so the O table is not accessed. We note that PAY looks up the appropriate C. row by primary key (C.CID, C.DID, C.WID) some fraction of the time, but sometimes looks up the row by lastname and firstname (C.WHERE: PY lookup by C.LAST, sort by C.FIRST, select the *median* row, in position ROOF(n/2), where there are n rows with the given C.LAST). Once the target row of C is located by either means, PAY increments C.BAL.

DLVY Delivery. The DLVY program is executed in deferred mode to deliver one outstanding order per transaction (or more—up to 10 orders for each of the ten distinct districts within a warehouse, but for now we consider only one order per transaction). An outstanding order is one that remains in the NO table, and the logic of DLVY starts by searching for a row in NO with a specific NO.WID, a specific NO.DID, and the *minimum* (oldest undelivered) NO.OID. If no such NO row is found, then delivery for this warehouse/district is skipped. Else, if a NO row is found, it is deleted, and DLVY retrieves the O row with identical WID, DID, and OID, reads the O.CID (Customer ID) and updates O.CAR (to reflect the "Carrier" performing delivery); then DLVY retrieves all the OL. row with matching WID, DID, and OID and for each one sets OL.DD (Delivery Date) to the current system time, and aggregates a sum of all OL.AMT (dollar charge). Finally, the C. row with matching WID, DID, and CID (found in O) is retrieved and C.BAL is incremented by the aggregated OL.AMT charge for delivering the order line items.

OSTAT Order-Status. The OSTAT program executes a RO program to query the status of a customer's last order, returning information from all OL rows for this order: OL.IID (Item ID), OL.DD (delivery date), and others. To access the appropriate OL rows, OS starts by retrieving a row of C using a WHERE clause with primary key (C.CID C.DID C.WID) 40% of the time and C.LAST, sorted by C.FIRST 60% of the time, selecting the *median* row–in position ROOF(n/2). Then the row in the O table with matching O.CID, O.DID, O.WID and largest existing O.OID (latest order) is retrieved, a few columns read, and the OL rows with matching O.OID retrieved, as explained above.

SLEV Stock-Level. The SLEV program executes as a Read-Only (RO) transaction to determine which of the items ordered by the last twenty orders in a given warehouse and district have fallen below a specified threshold value. The SLEV program reads D.NEXT to find the next order number to be assigned for this district, then accesses all OL rows with OL.OID in the range D.NEXT-20 to D.NEXT to learn OL.IID (all item IDs of the last twenty orders), then checks the quantity on hand in the STOCK table, S.QTY, for this S.WID and S.IID to test it against the given threshold.

4.2 Finding Dependencies in the TPC-C Application

Here is a list of steps we take to find dependencies in the TPC-C application.

Step 1. *Splitting.* Recall the idea of *splitting* programs from the introduction to this section. Programs with if-then-else branches in TPC-C are the following. First, both PAY and OSTAT access a customer row by primary key 40% of the time and by C.LAST and C.FIRST 60% of the time; neither C.LAST nor C.FIRST nor the primary key for C. is ever updated by any programs however, so there is no difference in dependencies between the two branches. Thus we do not split these programs in our analysis. Second, the program DLVY has an if-then-else branch following the search for the *minimum* (oldest undelivered) NO row. If the row is *not* found, then there is a DLVY $\Rightarrow$ NEWO dependency, since a later insert of such a NO row by NO modifies the set of rows found by the search WHERE clause, but if such a row *is* found there is no such dependency. Thus, we need to split DLVY into two programs, DLVY_1 (where no row is found) and DLVY_2 (where a row is found). The TPC-C Application is redefined to have these two different programs (see Figure 12 below). It will turn out that this splitting is needed to show serializability.

Step 2. *Tabulating Program Read/Write Accesses by Table/Column.* We start from table schemas, that is, a list of column names for each table, thus providing a list of table.column names that partially specify individual data items. For each table.column instance, we set up two entries for annotations, called the Reads Entry (for predicate reads (PR) and data-item reads (R)) and Writes Entry (W), and fill them in as follows. For each program P, we consider all Select and Update statements affecting data items, and annotate the associated table.column instance with the program name P in its appropriate entry, the Reads Entry for a Select, or Reads and Writes Entries (or just Writes) for an Update. If a program P is shown by annotation to Write a data item of some table.column and also shown to Read (R) the data item, the implication will be that it *always* Writes the *same* data item it originally Reads. This is true of all Read-Writes of data items in TPC-C, but may not hold in general.

Step 3. *Annotating Program "Where Restrictions" by Table*. The columns accessed in WHERE clauses are involved in predicate read operations, a kind of read, and are tabulated in the PR (predicate read) part of the Reads entry. Clearly, in some cases, this listing is too crude, but it will cover all potential conflicts. After conflicts are located, they can be further investigated, that is, the actual predicate can be studied to see if the apparently conflicting write provides a real conflict.

If a WHERE clause of program P retrieves a unique row through a primary key, the only predicate conflict with an Update statement of a different program R that can occur will be with an Update that changes primary key values, a very rare occurrence in online transactional applications, and not present anywhere in TPC-C. In fact, there are no updates to any columns involved in predicates, primary key or not. Conflicts involving predicate-read operations vs. inserts and deletes do show up in TPC-C.

Step 4. *Annotating Deletes and Inserts by Table*. We also need to take note of Delete and Insert statements on a table T that can be performed by any program P. We treat Inserts and Deletes as a Write of all columns by P, with additional annotation of (I) or (D).

Figure 11 gives the result of these steps for all tables that have any potential conflicts; table columns not accessed are left out. Note that each of the tables T have their primary key (PK) defined immediately following their table name, listed at the top. In the tables, PK is used as an abbreviation for that table's primary key columns. We do not include table columns that have no accesses. The STOCK table is not listed here, but is similarly handled. Three of the tables have no potential conflicts at all, and are also not listed: Warehouse (W), History (H), Item (I).

We will not discuss any further table annotations of Figure 11. Most of them arise directly from the Program Profiles we detailed in Section 4.2. Any accesses in Figure 11 that weren't mentioned in the Program Profiles are relatively unimportant and can be found in the more complete profiles of [TPC-C].

Step 5. *Determining Potential Conflicts and Real Conflicts.* Potential conflicts arise from data item accesses in the annotated table when two program names P_1 and P_2 both occur for the same table.column in Figure 11, and one or both of them is in the Writes (W) column. P_1 and P_2 can be the same program. In some cases, it is necessary to go back to the transaction programs for more details on a particular conflict, but all conflicts show up in this tabulation.

Since writes figure in all conflicts, the most straightforward way to process the annotated schema is by studying each filled-in Writes entry in turn. If P shows up alone in the Writes entry of an instance, and Q in its Reads entry, then there are conflicts in both directions $Q \xrightarrow{rw} P$ and $P \xrightarrow{wr} Q$, and the rw conflict is vulnerable. However, if P and Q are in the Writes entry, and Q in the Reads, both conflicts exist but neither is vulnerable, because the writes to the data item read by Q prevent concurrency in any rw conflict. Note that this conclusion requires that the read and write are to the same data item (same column, same row), as occurs in TPC-C as noted in Step 2 of Section 4.2.

DISTRICT(D), PK: WID, DID

Field Name	PR (Reads)	R (Reads)	W (Writes)
D.PK	NEWO, SLEV, PAY		
D.NAME		PAY	
D.STR1, STR2		PAY	
D.CITY, STA		PAY	
D.ZIP		PAY	
D.TAX		NEWO	
D.YTD		PAY	PAY
D.NEXT		NEWO, SLEV	NEWO

CUSTOMER(C), PK: WID, DID, CID

Field Name	PR	R	W
C.PK	PAY(40%), OSTAT(40%)		
C.FIRST		PAY, OSTAT	
C.MID		PAY, OSTAT	
C.LAST	NEWO, PAY(60%), OSTAT(60%)	PAY, OSTAT	
C.STR1,STR2		PAY	
C.CITY,STA		PAY	
C.ZIP		PAY	
C.PHONE		PAY	
C.SINCE		PAY	
C.CREDIT		PAY, NEWO	
C.CRELIM		PAY	
C.DISC		PAY, NEWO	
C.BAL		PAY, $DLVY_2$, OSTAT	PAY, $DLVY_2$
C.YPAY		PAY	PAY
C.PCNT		PAY	PAY
C.DCNT		$DLVY_2$	$DLVY_2$
C.DATA		PAY(maybe)	PAY(maybe)

NEW-ORDER(NO), PK: WID, DID, OID

Field Name	PR	R	W
NO.PK	$DLVY_1$,$DLVY_2$	$DLVY_2$ (NO.OID)	NEWO(I), $DLVY_2$ (D)

ORDER(O), PK: WID, DID, OID

Field Name	PR	R	W
O.PK	OSTAT, $DLVY_2$		NEWO(I)
O.CID		$DLVY_2$	NEWO(I)
O.ENT		OSTAT	NEWO(I)
O.CAR		OSTAT	NEWO(I), DLVY2

ORDER-LINE(OL), PK: WID,DID,OID,NUM

Field Name	PR	R	W
OL.PK	$DLVY_2$, SLEV, OSTAT		NEWO(I)
OL.IID		OSTAT, SLEV	NEWO(I)
OL.SWID		OSTAT	NEWO(I)
OL.QTY		OSTAT	NEWO(I)
OL.DD		OSTAT	NEWO(I), $DLVY_2$
OL.AMT		$DLVY_2$	NEWO(I)

Fig. 11. Annotated TPC-C tables.

Starting with the District table in Figure 11, we see that the only conflicts involve D.YTD and D.NEXT, since the other columns of D are only read, not written. We note a potential data item conflict between SLEV and NEWO on D.NEXT. This leads to an SLEV $\Rightarrow$ NEWO static dependency and an NEWO$\xrightarrow{wr}$SLEV dependency in our SDG(TPC-C) graph, see Figure 13. In addition, we have ww conflicts of NEWO with itself and PAY with itself, indicated by back arcs on Figure 13.

In the New-Order table of Figure 11, we see four programs listed as accessing NO.PK, that is, NO.WID, NO.DID, and NO.OID. We need to consider them by pairs, where each pair has a writing program member.

(1) $DLVY_1$ and NEWO. $DLVY_1$ has a predicate read for the smallest OID for given WID and DID, and finds none. There is a $DLVY_1 \Rightarrow$ NEWO conflict arising from a predicate conflict, because NEWO could insert a new row,

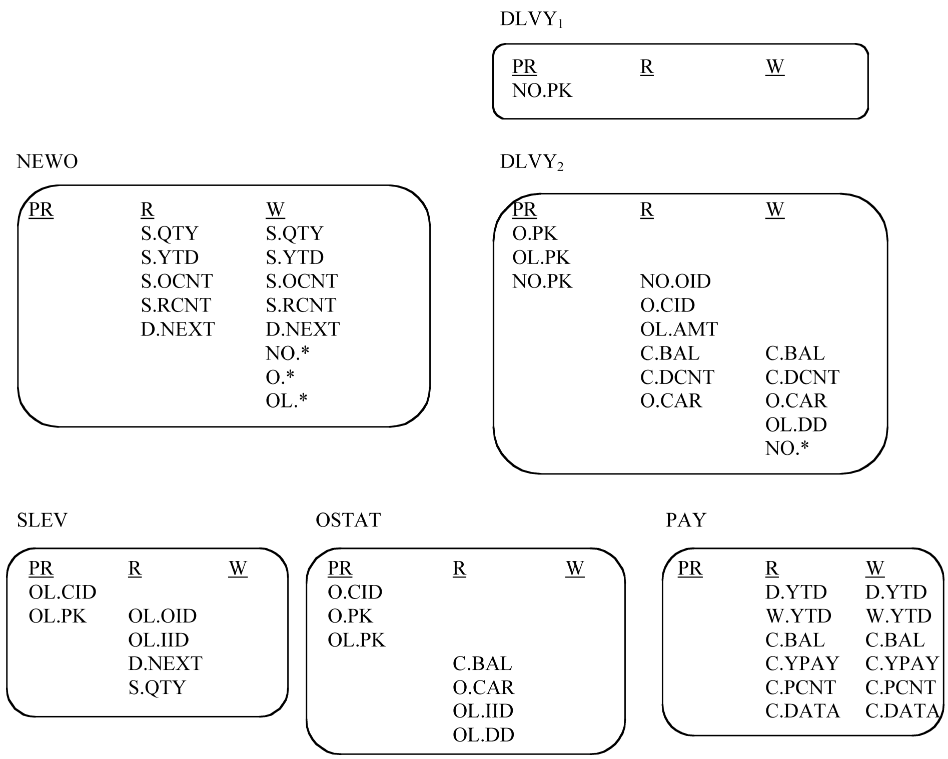


Fig. 12. Potentially conflicting TPC-C accesses grouped by program.

concurrently, that would change the result of DLVY_1s predicate read. There is no $\text{NEWO}\xrightarrow{wr}\text{DLVY}_1$ conflict, however, because if DLVY successfully reads a row output by NEWO, it is counted as DLVY_2.

(2) DLVY_2 and NEWO. DLVY_2 has the same predicate read, but finds such a row. It can Read what NEWO inserts, so we have $\text{NEWO}\xrightarrow{wr}\text{DLVY}_2$. However, NEWO cannot make a difference to the predicate read of DLVY_2, because additional rows beyond the one retrieved will have higher OIDs, so we have no rw conflict here.

(3) NEWO and NEWO. There is only a *ww* conflict.

(4) DLVY_2 and DLVY_2. There are *ww*, *rw* and *wr* conflicts here, but none vulnerable. Concurrent transactions of this program would access the same smallest OID, and have a ww conflict.

Note that although table OL shows reads by SLEV and a write by DLVY_2, it generates no field-level conflicts between SLEV and DLVY_2, since these accesses are to different columns.

We leave the rest of the details of finding conflicts in the annotated table of Figure 11 as an exercise for the reader. The result, after regrouping by program name is given in Figure 12. Cases where all columns in a table are involved in conflicts are indicated by wild-card notation. For example, the NEWO transaction is shown as writing O.*, that is, it writes all columns in the O. table.

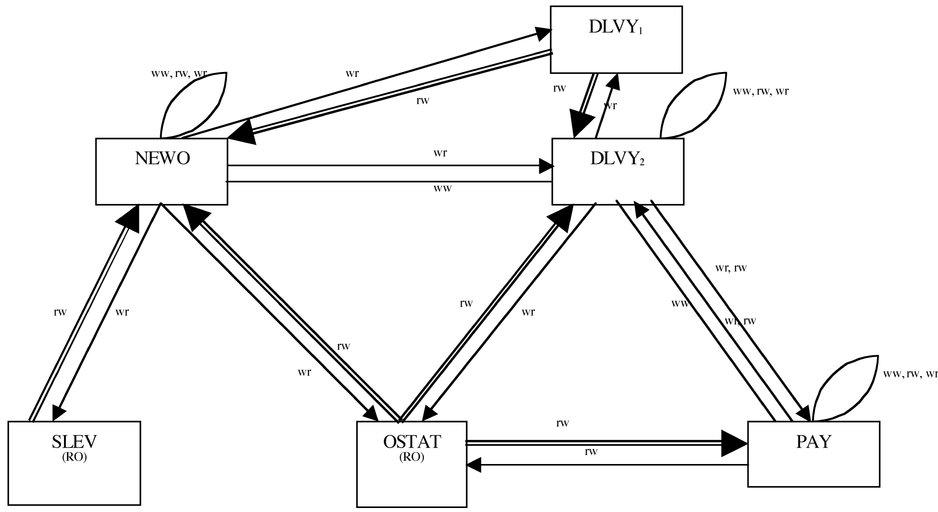


Fig. 13. Static dependency graph for TPC-C.

Figure 12 includes some data-item conflicts involving the STOCK table that were not shown in Figure 11. Note that although all programs use predicate reads to access tables, some are not shown because they cannot provide conflicts: they are accessing tables that never get inserted or deleted in this transaction mix, and the columns involved in the predicate are never updated.

4.3 Creating the Static Dependency Graph for TPC-C

The static dependency graph for TPC-C, SDG(TPC-C), is displayed in Figure 13. It is derived from examination of Figures 11 and 12, and the argument about the conflicts involving DLVY_1 and DLVY_2 above.

It is easy to see by inspection that SDG(TPC-C) has no program vertex with a vulnerable *rw* dependency edge both entering and leaving. Therefore, there are no dangerous structures, and TPC-C must be serializable when executed under SI. Note, however, that if we hadn't split D into DLVY_1 and DLVY_2, then there would have been a static dependency OSTAT $\Rightarrow$ DLVY (the DLVY here is the current DLVY_2) and another one, DLVY $\Rightarrow$ NEWO (the DLVY in this case is currently DLVY_1). Therefore, the act of splitting is crucial to our demonstration.

Many of the conflicts of Figure 13 require special conditions to occur, but since they don't show a dangerous structure, we don't need to consider the conditions. One case merits further discussion, that is, the conflicts between DLVY_2 and PAY, since the *ww* conflict is needed to make the *rw* conflicts nonvulnerable. In this case, all the conflicts arise from the same field, C.BAL, so if the rw conflict exists (same customer), so does the ww conflict that keeps them non-concurrent.

4.4 TPC-C vs. Other Applications

Although TPC-C is a nontrivial example of an OLTP system, it is simpler than many real applications, so we were able to perform a simplified analysis.

Possible complications that might arise in more complex applications include:

(1) Programs that update more than one row of a certain table.column. Our use of table.column to track a single field depended on the property of TPC-C that it updates at most one row from each table in a program execution.
(2) Programs that read a field from a row and then write the same column in another row. This is a skew read-write situation, one that can cause the simplest form of dangerous structure involving a single program.
(3) Updates to columns involved in predicates. This can generate a skew read-write as well, if the predicate read-set is a superset of the update-set.
(4) Fields that are conditionally updated. We often used an unconditional write to claim non-concurrency.
(5) Loops in program logic.

To handle cases (1), (2), and (3), we need more bookkeeping in the annotated tables like Figure 11. For case (1), in the W column, we mark (N) for multiple rows after the program name. For example, since DLVY_2 writes multiple rows of OL.DD in ORDER-LINE, it should be notated DLVY_2 (N) in the W column of the entry for OL.DD, but in this case all N rows are read or not by OSTAT, so there is no new behavior. Again, the default is that if a field shows up in the R or PR and W column, all rows read or predicate-read (in this field) are also written (in this field.) If this is not true (case (2) or (3) above), so there is a row with a certain field read or predicate-read but not written, and another row with this field written, the R or PR should be marked (SKEW).

We still look for potential conflicts by using the same rule in Step 5. Special attention needs to be paid to fields with (N) and/or (SKEW) markings. If P shows up alone in the Writes entry of an instance (with or without (N)), and Q in its Reads entry (with or without SKEW), then there are conflicts in both directions $Q \xrightarrow{rw} P$ and $P \xrightarrow{wr} Q$, and the rw conflict is vulnerable. If P (with or without (N)) and Q (unmarked) are in the Writes entry, and Q (unmarked) in the Reads, both conflicts exist but neither is vulnerable, because the writes to data item read by Q prevent concurrency in any rw conflict. All other cases with P and Q in W, Q in R result in vulnerable conflicts. Additionally, P (SKEW) for a field causes vulnerable conflicts for P with itself, unless they can be eliminated by further arguments. The resulting static dependency graph can be analyzed in light of Theorem 3.1.

Cases (4) and (5) involve program flow. Conditional writes could be handled by splitting. In some cases, a conditional write of a field brings no new behavior. There is an example of this in TPC-C, where the C.DATA field is conditionally updated by PAY. The R and W are marked (maybe). Since PAY always updates C.BAL, this conditional write cannot cause a vulnerable dependency.

For the more complicated program flow patterns like loops, we need to lump together all the accesses, and use (N) for the markings as appropriate.

5. AVOIDING ISOLATION ERRORS IN THE APPLICATION

Theorem 3.1 will often allow the DBA to determine that the mix of programs in an application is *serializable*, that is, that every execution, interleaving the programs from the application and running under Snapshot Isolation, will be serializable. This is the case when the static dependency graph has no dangerous structures. However, when the DBA produces the graph, he/she may find in it some dangerous structures. In some situations such as happened with TPC-C, a more precise analysis using application splitting could resolve matters and allow us to be sure that all executions are serializable. However, some applications may allow nonserializable executions when SI is the concurrency control mechanism. This means that some integrity constraints, which are not declared explicitly in the schema and so are not enforced by the DBMS, could be violated during execution. We think that the DBA would be very unwise to carry the risk of this happening. In such cases, the solution we suggest is to modify the application programs so as to avoid dangerous structures without changing the functionality of the programs. This will normally require a rather small set of changes. The DBA should identify every place in the static dependency graph where a dangerous structure exists. Every such structure is defined by one or two distinct vulnerable edges. Then, the DBA can then choose one of these vulnerable edges in each dangerous structure, and modify one or both application programs corresponding to the vertices of the edge so that the edge ceases to be vulnerable.[4]

5.1 Materialize the Conflict

Note that all forms of dangerous structures have two vulnerable anti-dependencies. One of the simplest ways to remove vulnerability is to invent a new data item that materializes the conflict, and add to each program involved a statement that updates this item. For example, we can create a special table, Conflict(Id, val), with Id the primary key. Then for each anti-dependency we wish to materialize, we can add a row (X, 0), X a unique value corresponding to the specific edge. Finally, we can add the following statement to both of the programs that have the vulnerable anti-dependency.

```
UPDATE Conflict SET val = val+1 WHERE Id = X;
```

The First Committer Wins rule will now operate to prevent these programs from generating concurrent committed transactions. Note that this method of materializing conflicts works particularly well for predicate conflicts, where the promotion method described below may not be available as the two programs in conflict might otherwise have no data items in common.

We need not use a single data item for each vulnerable edge where we want to remove the vulnerability. Often, the semantics of the programs involved will show that conflict does not occur in all transactions of the programs, but rather, conflict depends on parameter values. If so, we can similarly use the parameter values in new items to materialize the conflicts. Consider the Predicate-Based Write Skew Example 1.4. In that case, two concurrent executions of a program to insert new tasks to the WorkAssignments table for the same employee on

[4]The Oracle White Paper [Jacobs et al. 1995] provided a number of program modification suggestions that follow. What is new in our development is an effective procedure to guarantee that all serializability problems are addressed.

the same day can result in a non-serializable execution that breaks a constraint requiring the total number of hours worked by any employee on any day to be no more than eight. If we were to use a single value X in a Conflict table to materialize any conflict of this sort, it would become impossible to assign two tasks to two different employees concurrently, a problematic situation for a large company. But a minor variation of the Conflict table approach would factor the problem along natural parameters of the program. Simply create a table TotalHours(eid, day, total), with eid and day a primary key. The program to add new tasks for employees should then keep the row for each eid-day up to date with the total number of hours assigned: whenever the WorkAssignments table has a row added, the program making the assignment will also update the TotalHours row, guaranteeing that no vulnerable conflict can arise. It is even acceptable for the program that inserts new tasks to also insert the TotalHours row when none is found; the system-enforced primary key constraint will then guarantee that there are no cases of write skew on the TotalHours table itself!

5.2 Promotion

To remove vulnerability from a data-item anti-dependency, we can use a simpler approach. The data item on which the conflict arises already exists in both programs, so all we need to do is guarantee that when we read a data item we are going to depend on, another transaction cannot simultaneously modify the same data item. We have a way of ensuring this that we call *Promotion*, which has the advantage that only one of the programs needs to be modified.

A data-item anti-dependency implies that one can find one data item identified by a program that is both read by the transaction generated by P_1 and written by the transaction generated by P_2 in every case where there is a conflict. Therefore, we can alter P_1 so that in addition to reading this item, it performs an identity write. That is, if the data item is column c of table X, P_1 updates X to set X.c = X.c under the same conditions (WHERE clause) as used in the read. We describe this as a *Promotion* of the read of X to a write. Again, the First Committer Wins rule will prevent both transactions committing in any concurrent execution of P_1 and P_2. In fact, with the Oracle implementation of Snapshot Isolation, it is enough to modify P_1 so that the read of X is done in a SELECT FOR UPDATE statement; Oracle treats SELECT FOR UPDATE just like a write in all aspects of concurrency control, but has smaller overhead because no logging has to take place to recover an update.

5.3 Performance Impact of Modifications

When Theorem 3.1 applies, a DBA can be sure that serializability is indeed guaranteed when running on a DBMS such as Oracle, which offers a variant of SI as its SERIALIZABLE isolation level. Thus, in such cases, serializability is obtained without additional cost. However, when the conditions of Theorem 3.1 are not met, the DBA needs to modify the application by introducing extra conflicts, as explained above. This will certainly reduce the performance of the application by causing aborts whenever the crucial conflicts arise between concurrently executing programs, but this impact is essential to avoid non-serializable executions. We now report on some observations about the extent of these costs. We carried out some simple experiments on an Oracle DBMS, involving a single program with a vulnerable loop in its static dependency graph. The measurements were performed using Histex, a tool for executing transactional histories described in Liarokapis [2001]. Comparing the original application to one modified by promotion through SELECT FOR UPDATE, we found that the CPU usage for the modified (serializable) application exceeded the (not necessarily serializable) original by no more than 20% so long as the rate of conflict was low, for example, when each transaction had a rw-dependency on only 1 in 80 others, in a system with up to 20 concurrent transactions. However, if the contention was great, such as with each transaction having a *rw*-dependency on 1 in 20 others among 10 concurrent threads, then obtaining serializability caused the CPU usage to increase by about 35%. We found that the increase in the number of aborts was a good signal of the performance impact of the modification, and this can be easily estimated with a back-of-the-envelope calculation. We also found that promotion using an identity write of the promoted item had a cost similar to that when promotion was done with SELECT FOR UPDATE.

6. CONCLUSIONS

Snapshot Isolation is now an important concurrency mechanism offering good performance in many commonly needed circumstances, but it can produce errors in common situations. This may corrupt the database, leading to financial loss or even injury.

This work provides a mechanism to avoid serializability violations through analysis of the transaction programs, without requiring any modification of the DBMS engine. In some cases, our results show that an application will run serializably under Snapshot Isolation as it stands. When this is not the case, we have shown how to identify specific pairs of programs where small modifications of the text will lead to a semantically equivalent application that executes serializably. Given the ability to detect static dependencies between the application programs, and an understanding of the static dependency graph construction, programmers will have a practical way to ensure that their applications execute correctly even when the underlying database management doesn't. A utility providing the user an intuitive interface to perform this kind of analysis is the next step in our research.

REFERENCES

ADYA, A., LISKOV, B., AND O'NEIL, P. 2000. Generalized isolation level definitions. In *Proceedings of IEEE International Conference on Data Engineering,* (Feb.) IEEE Computer Society Press, Los Alamitos, Calif., 67–78.

ANDERSON, T., BRETBART, Y., KORTH, H., AND WOOL, A. 1998. Replication, consistency and practicality: are these mutually exclusive? In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June). ACM, New York, 484–495.

BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June) ACM, New York, 1–10.

BERGE, C. 1976. *Graphs and Hypergraphs* (2nd edition). North-Holland Mathematical Library, Volume 6.

BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. (This text is now out of print but can be downloaded from http://research.microsoft.com/pubs/ccontrol/default.htm)

BERNSTEIN, A., LEWIS, P., AND LU, S. 2000. Semantic conditions for correctness at different isolation levels. In *Proceedings of IEEE International Conference on Data Engineering* (Feb.). IEEE Computer Society Press, Los Alamitos, Calif., 57–66.

BREITBART, Y., KOMONDOOR, R., RASTOGI, R., SESHADRI, S., AND SILBERSCHATZ, A. 1999. Update propagation protocols for replicated databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June). ACM, New York, 97–108.

CHAMBERLIN, D., ASTRAHAN, M., BLASGEN, M., GRAY, J., KING, W., LINDSAY, B., LORIE, R., MEHL, J., PRICE, T., PUTZOLU, F., SELINGER, P., SCHKOLNICK, M., SLUTZ, D., TRAIGER, I., WADE, B., AND YOST, R. 1981. A history and evaluation of System R. *Commun. ACM 24*, 10 (Oct.). 632–646. (Also in: M. Stonebraker and J. Hellerstein, *Readings in Database Systems*, Third Edition, Morgan Kaufmann 1998.)

ELNIKETY, S., PEDONE, F., AND ZWAENEPOEL, W. 2004. Generalized snapshot isolation and a prefix-consistent implementation. Tech. Rep. IC/2004/21, EPFL, Mar.

ESWARAN, K., GRAY, J., LORIE, R., AND TRAIGER, I. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov.), 624–633.

FEKETE, A. 1999. Serializability and snapshot isolation. In *Proceedings of the Australian Database Conference* (Auckland, New Zealand, Jan.). 201–210.

FEKETE, A., O'NEIL, E., AND O'NEIL, P. 2004. A read-only transaction anomaly under snapshot isolation. ACM *SIGMOD Record 33*, 3 (Sept.), 12–14.

GRAY, J. (ED.). 1993. *The Benchmark Handbook* (2nd edition). Morgan-Kaufmann, San Francisco, Calif.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, San Francisco, Calif.

GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. 1996. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June). ACM, New York, 173–182.

JACOBS, K., BAMFORD, R., DOHERTY, G., HAAS, K., HOLT, M., PUTZOLU, F., AND QUIGLEY, B. 1995. *Concurrency Control: Transaction Isolation and Serializability in SQL92 and Oracle7.* Oracle White Paper, Part No. A33745 (July).

LIAROKAPIS, D. 2001. *Testing Isolation Levels of Relational Database Management Systems*, Ph.D. dissertation, University of Massachusetts, Boston, Mass. (Dec.). (This thesis can be downloaded from http://www.cs.umb.edu/~dimitris/thesis/thesisDec20.pdf.)

PAPADIMITRIOU, C. 1986. *The Theory of Database Concurrency Control*. Computer Science Press.

SHASHA, D. AND BONNET, P. 2002. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan-Kaufmann, San Francisco, Calif.

SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. 1995. Transaction chopping: Algorithms and performance studies. *ACM Trans. Datab. Syst. 20*, 3 (Sept.), 325–363.

SCHENKEL, R. AND WEIKUM, G. 2000. Integrating snapshot isolation into transactional federations. In *Proceedings of 5th IFCIS International Conference on Cooperative Information Systems (CoopIS 2000)* (Sept.), 90–101.

TPC-C BENCHMARK SPECIFICATION, available at http://www.tpc.org/tpcc/.

Received August 2004; accepted November 2004