Correct Execution of Transactions at Different Isolation Levels

Shiyong Lu, *Member*, *IEEE*, Arthur Bernstein, *Fellow*, *IEEE*, and Philip Lewis, *Fellow*, *IEEE*

Abstract—Many transaction processing applications execute at isolation levels lower than SERIALIZABLE in order to increase throughput and reduce response time. However, the resulting schedules might not be serializable and, hence, not necessarily correct. The semantics of a particular application determines whether that application will run correctly at a lower level and, in practice, it appears that many applications do. The decision to choose an isolation level at which to run an application and the analysis of the correctness of the resulting execution is usually done informally. In this paper, we develop a formal technique to analyze and reason about the correctness of the execution of an application at isolation levels other than SERIALIZABLE. We use a new notion of correctness, semantic correctness, a criterion weaker than serializability, to investigate correctness. In particular, for each isolation level, we prove a condition under which the execution of transactions at that level will be semantically correct. In addition to the ANSI/ISO isolation levels of READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ, we also prove a condition for correct execution at the READ-COMMITTED with first-committer-wins and at SNAPSHOT isolation. We assume that different transactions in the same application can be executing at different levels, but that each transaction is executing at least at READ UNCOMMITTED.

Index Terms—Isolation levels, correctness, serializability, transactions.

1 Introduction

Serializability has been widely accepted as the correctness criterion for concurrently executing transactions. Its implementation is generally based on a strict two-phase locking mechanism [12], [8], in which a transaction holds the locks it has acquired until it terminates. Unfortunately, this implies that locks might be held for long periods of time, causing performance to suffer, particularly in applications having long-running transactions and/or data hot-spots. To support the trade off between correctness and performance, the notion of isolation levels was first introduced in [13] under the name of *Degrees of Consistency*, and then defined in the ANSI/ISO SQL-92 standard [1]. Currently, most commercial DBMSs support isolation levels other than SERIALIZABLE.

Unfortunately, executing transactions at isolation levels lower than SERIALIZABLE might result in schedules that are not serializable and, hence, not necessarily correct. The semantics of a particular application determines whether that application will run correctly at a lower level and, in practice, it appears that many applications do. The choice of an isolation level at which to run an application and the analysis of the correctness of the resulting schedules is usually done informally. The development of a formal technique for analyzing execution at different isolation levels will have both theoretical and practical impact.

Since the execution of an application at an isolation level other than SERIALIZABLE might be correct, the conventional correctness criterion, serializability, is too strong. In this paper, we use a correctness criterion proposed in [9], [5] and further developed in [6], called *semantic correctness*, to investigate correctness. The semantic correctness of a schedule resulting from the concurrent execution of a set of transactions requires that the schedule have the same *semantic* effect as a serial schedule of the same set of transactions. The semantic correctness condition for a transaction schedule is based on the conditions developed in [17] for the correct execution of an arbitrary concurrent program.

In this paper, we assume that the locking algorithm described in [3] is used to implement different isolation levels and we analyze the pattern of interleavings that results. For each isolation level, we prove a condition under which schedules will be semantically correct. In addition to the ANSI/ISO isolation levels [1] of READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ, we also prove a correctness condition for READ-COMMITTED with first-committer-wins and at SNAPSHOT isolation. We assume that different transactions of an application can be executing at different isolation levels, but that each transaction is executing at least at READ UNCOMMITTED. The main contributions of this paper are:

- We prove a correctness condition for both page databases (this will be defined later) and relational databases for each isolation level.
- We provide an algorithm to determine the lowest isolation level at which the execution of each transaction of an application will be semantically correct. This algorithm maximizes the performance of running an application without sacrificing correctness.
- While all possible interleavings need to be considered for analyzing correctness of a concurrent program [17], our results show that only a few

- *S. Lu is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: shiyong@cs.wayne.edu.*
- *A. Bernstein and P. Lewis are with the Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11790. E-mail: {art, pml}@cs.sunysb.edu.*

Manuscript received 24 July 2002; revised 20 Mar. 2003; accepted 25 Aug. 2003.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 117005.

Isolation Level	Item Read Lock	Predicate Read Lock	Item and Predicate Write Lock (same)
READ UNCOMMITTED	None	None	Long-term
READ COMMITTED	Short-term	Short-term	Long-term
REPEATABLE READ	Long-term	Short-term	Long-term
SERIALIZABLE	Long-term	Long-term	Long-term

Fig. 1. Locking implementation of isolation levels.

noninterference checks are necessary. This greatly reduces the amount of analysis work.

- The conditions we identify provide a formal basis that underlies the informal reasoning that justifies the use of isolation levels other than SERIALIZABLE.

This paper extends the result presented in [10] with complete proofs of the theorems for relational databases and with an axiomatization of SQL statements that forms the basis of these proofs. The theorems for page databases are stated without proofs, and readers interested in the proofs are referred to [10]. The rest of the paper is organized as follows: Section 2 presents an overview of the semantic-correctness theory. Section 3 identifies semantic correctness conditions for each isolation level for page databases. Section 4 identifies their counterparts for relational databases. Based on these conditions, Section 5 presents an algorithm to determine the lowest isolation level for each transaction of a particular application. Section 6 describes a customer order example to illustrate the analysis. Finally, Section 7 concludes the paper and points out some future work.

2 Semantic Correctness

The notion of isolation levels was introduced to improve the performance of transaction processing. In general, the lower the isolation level, the better the performance. However, the correctness of schedules produced at isolation levels lower than SERIALIZABLE is not ensured. For example, READ COMMITTED is subject to lost update, write skew, and phantom anomalies. A detailed description of the anomalies to which each isolation level is subject is presented in [3]. The goal of this research is to develop correctness conditions for each isolation level. We assume that the locking algorithm proposed in [3] is used to implement isolation levels. For convenience, we summarize this in Fig. 1, in which a long-term lock will be held until the owner transaction terminates, and a short-term lock will be released immediately when an operation completes.

We use *Hoare logic* [8] to reason about correctness. Well-formed formulas are called *triples* and have the form: $\{P\}\ S\ \{Q\}$, where P and Q are well-formed formulas of predicate logic [15], which we refer to as *preconditions* and *postconditions*, respectively, or *assertions*, generally, and S is a syntactically correct statement or sequence of statements in some imperative programming language. A triple has the following *partial correctness* semantics.

Definition 1 (Partial correctness). *$\{P\}\ S\ \{Q\}$ is valid if and only if, whenever S starts execution from a state satisfying P and execution terminates, the resulting state satisfies Q.*

Hoare logic is an extension of predicate logic in that any well-formed formula in predicate logic is also a well-formed formula in Hoare logic. In addition to the axioms and inference rules of predicate logic, Hoare logic includes axioms and inference rules related to the data types and the program constructs of a programming language. These axioms and inference rules form the proof system of Hoare logic. Readers are referred to [8], [15] for a detailed description. Although Hoare logic is sound, it is incomplete [2]. Cook circumvented the incompleteness problem by defining the notion of *relative completeness* [11]. A description of the expressiveness and completeness of Hoare logic can be found in [4].

Using Hoare logic, we describe the semantics of a transaction T_i by a triple as follows:

$$\{I_i \wedge B_i \wedge (\overline{x_i} = \overline{\mathcal{X}_i})\}\ T_i\ \{I_i \wedge Q_i\}. \tag{1}$$

I is the *consistency constraint* of the database, and I_i represents those conjuncts of I required to guarantee that I is preserved by the execution of T_i and that T_i produces its intended result, Q_i (described below). We assume that $I \equiv \bigwedge_{i=1}^{n} I_i$, where n is the number of transactions types in the system (i.e., each conjunct of I is required by at least one transaction type). For example, suppose a banking system has two transaction types: T_1 withdraws cash from a savings account and T_2 withdraws cash from a checking account. The consistency constraint I might assert that *both* savings and checking accounts have nonnegative balances. However, T_1 only requires that the balance of a savings account is nonnegative (specified by I_1), and T_2 only requires that the balance of a checking account is nonnegative (specified by I_2). I_i is a postcondition of T_i since we require that any conjunct of I that is made false during the execution of T_i is returned to the true state when T_i terminates. B_i describes all conditions that T_i assumes to be true of the arguments passed to it. For example, if T_i is a deposit transaction and dep is the parameter representing the money to be deposited, then B_i might assert $dep \geq 0$.

Q_i is called the *result* and asserts that T_i has performed its intended function. Continuing the above example, if T_i deposits dep dollars into an account whose balance is bal, we need to assert as a postcondition of T_i that the final balance is dep more than the initial balance. In order to refer to the initial balance in the postcondition, we introduce a logical variable, $\overline{\mathcal{X}_i}$, whose sole purpose is to record the initial value of a database variable, x_i. $\overline{\mathcal{X}_i}$ is not referenced by T_i. In the example, we characterize T_i with the following triple:

$$\{bal >= 0 \wedge dep \geq 0 \wedge bal = BAL\}\ T_i$$
$$\{bal >= 0 \wedge bal = BAL + dep\}.$$

Equation (1) goes beyond the requirement that a transaction maintains the consistency of the database by asserting that not only must T_i move the database from one consistent state to another, but that only a subset of the consistent states are acceptable when the transaction terminates. Equation (1) can be regarded as a formal restatement of the specification of T_i. We can demonstrate that T_i is correct by proving that (1) is a theorem using Hoare logic.

To overcome the limit of serializability and to increase performance, in [5], we propose a new correctness criterion, called *semantic correctness*. We assume that the execution of each statement of a transaction is atomic and isolated. A schedule, Sch, of transactions is *semantically correct* if

$$\{I\}\ Sch\ \{I \wedge Q_{Sch}\} \tag{2}$$

is true. A semantically correct schedule must maintain the consistency of the database, as indicated by the fact that I is a pre and postcondition of Sch. A semantically correct schedule must also transform the database to a state that reflects the cumulative results of all the transactions in Sch. We denote the assertion that describes that set of states by Q_{Sch}, the *cumulative result*. The relationship between Q_{Sch} and the results, Q_i, of the individual transactions is described in [6]. In essence, Sch is semantically correct if its postcondition is the same as the postcondition of a serial schedule of the same set of transactions, where the serial order is the order of transaction completion in Sch. For example, if Sch consists of several deposit transactions on some bank account, Q_{Sch} might assert that the final balance is greater than the initial balance by an amount equal to the sum of the deposits. We will state a sufficient condition that guarantees the semantic correctness of schedules in Theorem 1, which forms the basis of the results presented in this paper. Readers are not required to understand the notion of cumulative result, but are referred to [6] if they are interested.

As illustrated in [5], semantic correctness is weaker than serializability, and it allows schedules that result in states that could not have been reached in any serial schedule. For example, a stock trading application might have a buy transaction type that takes as parameters the identity of a stock and the number of shares, n, to be purchased and a result that states "when each share was purchased no cheaper unbought shares of the stock existed in the database." In a semantically correct schedule, two concurrent transactions, T_1 and T_2, could each buy some shares at \$30 and some at \$31 per share, even though initially there are n shares available at \$30. First, T_1 buys $n/2$ shares at \$30; then, T_2 buys $n/2$ shares at \$30; then, since there are no more shares available at \$30, T_1 buys $n/2$ shares at \$31; and, finally, T_2 buys $n/2$ shares at \$31. When each transaction terminates, its result is true since, when each share was bought, no cheaper unbought shares existed in the database. The final state could not have been produced by a serializable schedule since the purchase price of all shares bought by one or the other of the two transactions would have been \$30. Since the set of semantically correct schedules properly includes the set of serializable schedules, a concurrency control that enforces semantic correctness can perform significantly better than one that enforces serializability [7].

A proof of (1) can be abbreviated by an annotated program in which each statement $S_{i,j}$ of T_i is preceded by its precondition $P_{i,j}$ that describes the state of the system at the time $S_{i,j}$ is eligible for execution, and succeeded by its postcondition $P_{i,j+1}$ that describes the state of the system when the execution of $S_{i,j}$ terminates. Hence, assertions are associated with control points and we say that an assertion and its corresponding control point are *active* if the following statement is eligible for execution. In this paper, we assume transactions are executed in an interleaved fashion, and each statement of a transaction is executed as an isolated and atomic unit. A sufficient condition to guarantee the correctness of the concurrent execution of a set of transactions is that proofs of the transactions can be found such that for each pair of transactions in the set, T_i and T_k, the triple

$$\{P_{i,j} \wedge P_{k,l}\}\ S_{k,l}\ \{P_{i,j}\} \tag{3}$$

is a theorem for each assertion $P_{i,j}$ of T_i and each statement $S_{k,l}$ of T_k [17]. In this case, we will say that the proofs do not interfere and, more particularly, (3) asserts that *$S_{k,l}$ does not interfere with $P_{i,j}$*. The notion of *interference* is given below.

Definition 2 (Interference). *Let $P_{i,j}$ be an assertion in the proof of T_i and $S_{k,l}$ be a statement characterized by $\{P_{k,l}\}\ S_{k,l}\ \{P_{k,l+1}\}$ in the proof of T_k. If $\{P_{i,j} \wedge P_{k,l}\}\ S_{k,l}\ \{P_{i,j}\}$ is a theorem, then $S_{k,l}$ does not* interfere with *$P_{i,j}$; otherwise, $S_{k,l}$* interferes with *$P_{i,j}$.*

If $S_{k,l}$ interferes with $P_{i,j}$, the execution of $S_{k,l}$ from a state satisfying $P_{i,j}$ and $P_{k,l}$ might make $P_{i,j}$ false (the execution will not necessarily make $P_{i,j}$ false as explained later). If the execution makes $P_{i,j}$ false, then we say that the execution of $S_{k,l}$ *invalidates* $P_{i,j}$. Then, the subsequent behavior of T_i is unpredictable and semantic correctness is not guaranteed. We define the notion of invalidation formally as follows:

Definition 3 (Invalidation). *Let $P_{i,j}$ be an assertion in the proof of T_i, and $S_{k,l}$ be a statement characterized by $\{P_{k,l}\}\ S_{k,l}\ \{P_{k,l+1}\}$ in the proof of T_k. If $S_{k,l}$ is executed from a state satisfying $P_{i,j}$ and $P_{k,l}$, and the resulting state does not satisfy $P_{i,j}$, then the execution of $S_{k,l}$ has* invalidated *$P_{i,j}$.*

The following example illustrates the notions of interference and invalidation.

Example 1. Consider the triple $\{x = X\}\ x := x + 1\ \{x = X + 1\}$ characterizing the statement $S_{k,l}$ in the proof of transaction T_k. $S_{k,l}$ interferes with assertion $x = y$ in the proof of T_i, but not with assertion $x > y$; during execution, $x := x + 1$ will invalidate $x = y$, but not $x > y$.

However, interference does not necessarily lead to invalidation. For example, the locking scheme that implements a particular isolation level might prevent interleavings that lead to invalidation. The resulting schedules will be semantically correct despite the presence of interference. As another example, suppose the proof of T_i is replaced by another proof in which $P_{i,j}$ is replaced by a stronger assertion $P'_{i,j}$ such that $P'_{i,j} \Rightarrow P_{i,j}$. Let $S_{k,l}$ be a statement characterized by $\{P_{k,l}\}$ $S_{k,l}$ $\{P_{k,l+1}\}$ in the proof of transaction T_k. Suppose $S_{k,l}$ interfere with $P_{i,j}$, but not with $P'_{i,j}$. Then, executing $S_{k,l}$ from a state that satisfies $P'_{i,j}$ (and, thus, also $P_{i,j}$) and $P_{k,l}$ does not invalidate $P'_{i,j}$ and, therefore, does not invalidate $P_{i,j}$. Hence, condition (3) is too strong for semantic correctness. In [6], a weaker condition is identified to ensure the semantic correctness of schedules. We state it in Theorem 1 after we introduce the notion of semantic correctness of transaction execution.

Definition 4 (Semantic correctness of transaction execution). *The execution of a particular transaction, T_i, is* semantically correct *in schedule Sch if 1) $P_{i,j}$ is true when $S_{i,j}$ is executed and 2) Q_i is not invalidated by any statement of other transactions interleaved in Sch with T_i.*

Theorem 1. *A schedule Sch is semantically correct if the execution of each transaction T_i in Sch is semantically correct.*

Proof. See [6]. □

While (3) requires the proof of a significant number of triples of the form (3), the intuition that "the locking scheme that implements a particular isolation level might prevent some interleavings that lead to invalidation" motivates us to develop correctness conditions that only require the proof of a small number of *critical* triples. We prove that, if these triples are demonstrated to be theorems, then no invalidation will occur and, according to Theorem 1, the resulting schedule will be semantically correct. A major goal of this research is to determine, for each isolation level, which triples must be checked. This dramatically reduces the amount of analysis. For example, for SNAPSHOT, isolation only K^2 triples must be checked in a system of K transactions types, regardless of the number of operations per transaction.

3 Semantic Conditions for Page Databases

In this section, we present conditions which, for each isolation level, enumerate the noninterference theorems that must be demonstrated in order to ensure semantic correctness. We consider a mixed system in which different transactions can be executing at different isolation levels, but that each transaction is executing at least at the READ UNCOMMITTED level. SNAPSHOT isolation is not considered as part of the mixed system and is addressed separately since we assume it does not use a locking scheme. We first consider page databases and then relational databases. In page databases, no database items are deleted or inserted, and each item is referred to by name in read or write statements. In relational databases, predicates are used in SQL statements to specify the database items they access.

In page databases, a transaction program accesses local variables (in its workspace) and database variables using the following constructs:

- **Assignment statement.** There are three kinds of assignment statements: a read statement, which atomically assigns the value of a database item to a local variable; a write statement, which atomically assigns the value of a local variable to a database item; and a local assignment statement, which does not involve any database items.
- **Conditional statement.** We assume that the condition is constructed from local variables.
- **Loop statement.** We assume that the condition is constructed from local variables.

Local variables will be denoted X, Y and database variables will be denoted x, y. We use the notation $sp(P, S_{i,k})$ to denote the strongest postcondition of $S_{i,k}$ that can be asserted when it is executed starting in a state that satisfies P. We assume that the locking protocols given in [3] are used to implement all isolation levels.

A *write block* of a transaction T_j is a segment of code of T_j which contains at least one write statement. Thus, any single write statement in T_j is a write block of T_j, and T_j is a write block of T_j if it contains at least one write statement.

3.1 Semantic Condition for READ UNCOMMITTED

The locking implementation for READ UNCOMMITTED [3] requires that transactions obtain long-term write locks on items that they write,[1] but no read locks are acquired on items that they read. Long term locks are held until the transaction completes. The following theorem states a condition under which a transaction will execute correctly at READ UNCOMMITTED.

Theorem 2. *The execution of a transaction T_i at READ UNCOMMITTED will be semantically correct if each write statement (including those that rollback a transaction) in every transaction does not interfere with I_i, the postcondition of every read statement in T_i, and Q_i.*

Proof. See [10]. □

The first thing to note is that only a write statement that modifies a database item in one transaction can interfere with an assertion in another. Intuitively, we need to worry about the postcondition of a read statement since the postcondition might assert something about the database stronger than its precondition. We do not need to worry about the postcondition of a local assignment statement since the statement does not involve any database item and, thus, the postcondition cannot assert something about the database stronger than its precondition. Also, we do not need to worry about the postcondition of a write statement $x := X$ since the strongest thing we can say about x is $x = X$. As a long-term write lock will be held on x, this assertion cannot be invalided by the execution of any other concurrent transaction. Finally, since a transaction executing at READ UNCOMMITTED can read uncommitted data, it is necessary to consider the interference caused by write statements that rollback any transaction.

Example 2. Suppose there are n customers, each of whom has multiple valid email addresses. Here, we say an

1. Database systems often prohibit transactions at this level from updating the database. We ignore this restriction here.

email address is *valid* if it is indeed for the corresponding customer. The primary email address of each customer is registered in array $cust[n]$. Two transaction types access the array: The $Email\_List$ transaction type scans the array and prints out an email mailing list; the $Update\_Email$ transaction type changes an existing valid email address to another valid email address.

- I_1 is an integrity constraint of the database required for the correct execution of $Email\_List$ and it asserts "each email address in $cust[n]$ is valid."
- $Email\_List$ has n read statements in the form of $E_j := cust[j]$ $(j = 1, \cdots, n)$, where E_j is a local variable; each statement has a postcondition $P_{1,j}$ that asserts "E_j is a valid email address."
- Q_1 is the postcondition of $Email\_List$ that asserts "the n printed email addresses are valid."

It is obvious that the write statement of $Update\_Email$, as well as the abort of $Update\_Email$, does not interfere with I_1, $P_{1,j}$ $(j = 1, \cdots, n)$, and Q_1, respectively. Hence, $Email\_List$ can run at READ UNCOMMITTED with semantic correctness ensured.

3.2 Semantic Condition for READ COMMITTED

The locking implementation of READ COMMITTED [3] requires that transactions obtain long-term write locks on items that they write and short-term read locks on items that they read. A short-term lock is released when the operation completes. The following theorem states a condition under which a transaction will execute correctly at READ COMMITTED.

Theorem 3. *The execution of a transaction T_i at READ COMMITTED will be semantically correct if each transaction T_j (including those that rollback a transaction) does not interfere with the postcondition of every read statement in T_i, and with Q_i.*

Proof. See [10]. □

In contrast to Theorem 2 in which each write statement of a transaction is considered individually, here we consider an entire transaction as a single isolated unit. The intuition is that, since each transaction T_j is executed at READ UNCOMMITTED or a higher isolation level, T_j will hold a long term write lock on any item it writes. Since, at READ COMMITTED, T_i uses short term read locks, T_i cannot read any item written by T_j until T_j terminates. Thus, each read statement of T_i will not see any intermediate result of T_j and the whole transaction T_j can be considered as a single isolated unit to reason about its interference to T_i.

Example 3. Suppose the database contain an array emp, with one record for each employee. $emp[i].rate$ is the ith employee's hourly rate, $emp[i].num\_hrs$ is the number of hours that the employee has worked so far this week, and $emp[i].sal$ is that employee's accumulated salary for the week. A conjunct of the integrity constraint, I_{sal}, asserts that, for all records in emp, "$emp[i].rate * emp[i].num\_hrs = emp[i].sal$."

The granularity of locking is at the level of records. An instance of transaction type $Hours$ is executed at the end of each workday to record the number of hours worked by the ith employee that day. It executes one write statement to increment $emp[i].num\_hrs$ and another to update the accumulated salary. Hence, although the two write statements together preserve I_{sal}, individually they do not. A second transaction type, $Print\_Records$ causes the records to be printed. Its specification requires that each printed record be a consistent snapshot of that employee's information at the time the record is printed. Since each individual write statement of $Hours$ interferes with I_{sal}, the conditions of Theorem 2 are violated. Therefore, the correctness of the execution of $Print\_Record$ at READ UNCOMMITTED cannot be ensured.

However, since transaction $Hours$, as a whole isolated unit, does not interfere with the postcondition of read statement of $Print\_Record$, and with the postcondition of $Print\_Record$, the conditions of Theorem 3 are satisfied. Hence, the execution of $Print\_Record$ at READ COMMITTED will be semantically correct.

3.3 Semantic Condition for READ COMMITTED with First-Committer-Wins

The *READ COMMITTED with first-committer-wins* isolation level is an extension of READ COMMITTED with one feature from SNAPSHOT isolation. Transactions obtain long-term write locks on items that they write and short-term read locks on items that they read. In addition, if T_1 writes a data item and commits between the times that T_2 has read and attempts to write the item, T_2 will be aborted (first-committer-wins). READ COMMITTED with first-committer-wins is easily and often implemented in relational databases by running an application at READ COMMITTED and ensuring that whenever a database item x is updated, a sequence number $sn(x)$ that is associated with x is incremented. An update statement that updates x will read $sn(x)$ to determine whether the value of x has changed since it was read. This isolation level is also supported by a number of vendors. Some vendors call this level READ COMMITTED with optimistic reads.

Theorem 4. *The execution of a transaction, T_i, at READ COMMITTED with first-committer-wins will be semantically correct if each transaction does not interfere with the postconditions of those read statements in T_i that are not followed by a write statement on the same item, and with Q_i.*

Proof. See [10]. □

Note that READ COMMITTED with first-committer-wins effectively holds a read lock on an item that it reads, then subsequently writes for the time period between the execution of the read statement and the execution of the write statement and, henceforth, a long-term write lock.

Also note that, if transaction T_i writes all the data items it reads, then when T_i commits, it has effectively held long-term read locks on those items and, hence, in this case, READ COMMITTED with first-committer-wins is equivalent to REPEATABLE READ. We give an example of correct execution at the READ COMMITTED with first-committer-wins level in Section 6.

3.4 Semantic Condition for REPEATABLE READ

The locking implementation of the REPEATABLE READ isolation level [3] requires that a transaction acquire long-term read and write locks on the data items that it accesses. The only problem at the REPEATABLE READ level is the possibility of phantoms [12]. Since phantoms do not occur in page (nonrelational) databases, REPEATABLE READ ensures serializability. A serializable schedule can be transformed into an equivalent serial schedule, in which no assertions of an arbitrary transaction T_i will be invalidated by any step of a transaction interleaved with T_i (since no other transaction is interleaved with T_i in a serial schedule). Thus, the semantic correctness of T_i is guaranteed.

3.5 Semantic Condition for SNAPSHOT Isolation

SNAPSHOT isolation is not one of the ANSI/ISO standard isolation levels, but is implemented in at least one commercial DBMS. The implementation of SNAPSHOT isolation given in [3] does not use locks. Instead, it uses a multiversion concurrency control that satisfies each read request made by transaction T_i with values from the version of the database, called a *snapshot*, that reflects the effect of all committed transactions at the time T_i executed its first read statement. Hence, read requests never wait. Requests to write are deferred until the transaction commits. T_i can be committed as long as no other transaction that committed after T_i's first read has updated a data item that T_i has also updated. This mechanism is referred to as *first committer wins* because the first transaction that has updated a particular data item and requests to commit is allowed to do so, while concurrent transactions that have updated that item are ultimately aborted. Thus, *first committer wins* has at least the effect of a long-term write locks on the items written.

We model a transaction T_i at SNAPSHOT isolation as two isolated atomic steps: a *read step* followed by a *write step*. The read step reads a snapshot of the database that reflects the effect of all committed transactions at the time T_i executed its first read statement. The write step is the remainder of the transaction. The step boundary reflects the fact that other transactions can commit while T_i is active, creating new versions of the database that might invalidate assertions that T_i has made about the database based on its snapshot. If T_i commits, its write step must commute with the write steps of these other transactions because they must have written to disjoint data items. Note that the postcondition of the snapshot does not necessarily state that the value of a data item in a snapshot is equal to the most recent committed value of that data item. It only needs to be strong enough to support the proof of the transaction [9].

Theorem 5. *A schedule produced under SNAPSHOT isolation is semantically correct if, given any two transactions T_i and T_j from the schedule, either:*

- *T_i's write set intersects T_j's write set (in which case one of the transactions will be aborted) or*
- *T_j's write step does not interfere with the postcondition of the read step of T_i, and with Q_i.*

Proof. See [10]. □

```
Withdraw_sav(i, w)
BEGIN TRANSACTION
        $\{(acct\_sav[i].bal + acct\_ch[i].bal \geq 0)$
        $\wedge (acct\_sav[i].bal = Sav_0)\}$
  $Sav := acct\_sav[i].bal;$
  $Ch := acct\_ch[i].bal;$
        $\{(acct\_sav[i].bal + acct\_ch[i].bal \geq 0)$
        $\wedge (acct\_sav[i].bal + acct\_ch[i].bal \geq$
        $Sav + Ch) \wedge (Sav = Sav_0)\}$
  if $(Sav + Ch \geq w)$ then
        $\{(acct\_sav[i].bal + acct\_ch[i].bal \geq 0) \wedge$
        $(acct\_sav[i].bal + acct\_ch[i].bal \geq$
        $Sav + Ch) \wedge (Sav + Ch \geq w)$
        $\wedge (Sav = Sav_0)\}$
    $acct\_sav[i].bal := Sav - w;$
  fi
        $\{(acct\_sav[i].bal + acct\_ch[i].bal \geq 0)$
        $\wedge (acct\_sav[i].bal = Sav_0 - w)\}$
END TRANSACTION
```

Fig. 2. Withdraw from savings account.

Example 4. Suppose we have two types of withdraw transactions, $Withdraw\_sav(i, w)$ and $Withdraw\_ch(i, w)$, which withdraw w from the ith depositor's savings and checking accounts, respectively. Savings and checking account information is held in arrays $acct\_sav$ and $acct\_ch$, respectively, and a conjunct of the integrity constraint, I_i, requires that $acct\_sav[i].bal + acct\_ch[i].bal \geq 0$. An annotated version of the $Withdraw\_sav$ program is given in Fig. 2. The annotation for $Withdraw\_ch$ is similar.

Sav_0 is a logical variable that is introduced to record the initial value of $acct\_sav[i].bal$ and its value will not be changed. Sav and Ch are local variables. The first two statements constitute the read step of $Withdraw\_sav$, and the rest of the statements constitute the write step. The postcondition of the read step (the second assertion in Fig. 2) of $Withdraw\_sav(i, w_1)$ is interfered with by the write step of $Withdraw\_ch(i, w_2)$. Hence, the theorem states that a concurrent schedule of the two transactions might not be semantically correct. A schedule in which the write step is interleaved between the read and write step of the other exhibits write skew [3]. Note that, although this same precondition is also interfered with by another instance of $Withdraw\_sav$, a concurrent schedule in which two instances of $Withdraw\_sav$ are interleaved is semantically correct because the first-committer-wins rule implies that one of them will be aborted (this is reflected in the first condition of the theorem). Finally, a $Deposit\_sav$ ($Deposit\_ch$) transaction, which adds money to $acct\_sav$ ($acct\_ch$), does not interfere with the postcondition of the read step of $Withdraw\_sav$. In this case, their concurrent execution is semantically correct (this is reflected in the second condition of the theorem).

4 Semantic Conditions for Relational Databases

In adapting the conditions for semantic correctness to relational databases, we must deal with database operations that involve predicates. The read statement is now the SELECT and its postcondition might assert that it read all the tuples that satisfy a certain predicate. Similarly, the write statements are UPDATE, INSERT, and DELETE, and their postconditions might assert that they wrote, inserted, or deleted all the tuples that satisfy a certain predicate.

Interference now takes new forms. For example, the postcondition of a SELECT statement might assert that the statement read all the tuples that satisfy a predicate, P. That assertion can be interfered with by another transaction that inserts a phantom tuple that also satisfies P.

Phantoms can occur in connection with write statements as well as in connection with SELECT. Thus, the postcondition of an UPDATE that asserts that the value of all tuples satisfying P have been updated can be interfered with by an INSERT that inserts a phantom tuple that satisfies P. That interference might not cause invalidation of the predicate, however, if the locking policy prevented the INSERT from executing after the UPDATE had taken place.

The locking policy for implementing the ANSI isolation levels discussed in [3] states that all "write locks on data items and predicates (are) long duration." Thus, at an isolation level that uses predicate locking, when an UPDATE, INSERT, or DELETE statement is executed, the referred predicate is write-locked for the duration of the transaction, and phantoms cannot be inserted into that predicate. Most DBMSs do not implement predicate locking, but instead use a locking protocol (perhaps consisting of some combination of table locks and index locks) that is equivalent to, or stronger than, predicate locking. We assume in what follows that the DBMS uses such a locking protocol. Then, Theorems 2 and 3 remain valid for relational databases, although the proofs are different since we need to consider phantoms.

In the following, we model SELECT, INSERT, DELETE, and UPDATE as assignment statements and axiomatize them using their strongest postconditions. This axiomatization enables us to apply the reasoning technique employed in page databases to relational databases.

SELECT statement: Assume a SELECT statement reads all the tuples in relation T that satisfy predicate pr. We introduce the relation variable

$$r = \{t \mid pr(t) \wedge (t \in T)\}$$

to denote the set of tuples returned by the SELECT statement. We model the SELECT statement as an assignment statement that assigns all the tuples in r to a local relation variable R. Then, we can use the following triple to specify the semantics of a SELECT statement.

$$\{P\}\ R := r\ \{Q\}.$$

If Q is the strongest postcondition of the SELECT statement with precondition P, by [14], we can specify Q in terms of P:

$$Q : \exists v(P_v^R\ \wedge\ R = r).$$

INSERT statement: Assume an INSERT statement inserts all the tuples that satisfy predicate pr into relation T. Using the same relation variable r defined for the SELECT statement, we model the INSERT statement as an assignment statement that assigns to T the value $T \cup r$. Thus, we can use the following triple to specify the semantics of an INSERT statement

$$\{P\}\ T := T \cup r\ \{Q\}.$$

If Q is the strongest postcondition of the INSERT statement with precondition P, by [14], we can specify Q in terms of P

$$Q : \exists v(P_v^T\ \wedge\ T = v \cup r).$$

DELETE statement: Assume a DELETE statement deletes all the tuples in T that satisfy predicate pr. Using the same relation variable r as above, we model the DELETE statement as an assignment statement that assigns to T the value $T - r$. Thus, we can use the following triple to specify the semantics of a DELETE statement:

$$\{P\}\ T := T - r\ \{Q\}.$$

If Q is the strongest postcondition of the DELETE statement with precondition P, by [14], we can specify Q in terms of P:

$$Q : \exists v(P_v^T\ \wedge\ T = v - r).$$

UPDATE statement: Assume an UPDATE statement updates all the tuples in T that satisfy predicate pr. Let r be the set of tuples in T that satisfy pr and r' be the updated values of these tuples. We model the UPDATE statement as an assignment statement that is equivalent to a DELETE statement that deletes the tuples in r followed by an INSERT statement that inserts the tuples in r'. Thus, we can use the following triple to specify the semantics of an UPDATE (predicate) statement:

$$\{P\}\ T := (T - r) \cup r'\ \{Q\}.$$

If Q is the strongest postcondition of the UPDATE statement with precondition P, by [14], we can specify Q in terms of P:

$$Q : \exists v(P_v^T\ \wedge\ T = (v - r) \cup r').$$

The following lemma is the basis for proving the theorems for relational databases. We use Φ to stand for the empty set.

Lemma 1. *Let $S_{i,k}$ be an INSERT, DELETE, or UPDATE statement on table T in transaction T_i. Suppose $S_{i,k}$ is characterized by $\{P_{i,k}\}\ S_{i,k}\ \{P_{i,k+1}\}$ in the proof of T_i, and $pd_{i,k}$ is the predicate in the WHERE clause of $S_{i,k}$. Let $S_{j,h}$ be an INSERT, DELETE, or UPDATE statement on table T of another transaction T_j and $pd_{j,h}$ be the predicate in the WHERE clause of $S_{j,h}$, and $pd_{i,k} \cap pd_{j,h} = \Phi$. Let $sp(P_{i,k}, S_{i,k})$*

be the strongest postcondition of $P_{i,k}$ with respect to $S_{i,k}$. If the tuples that $S_{i,k}$ writes do not satisfy $pd_{j,h}$[2] *and the tuples that $S_{j,h}$ writes do not satisfy $pd_{i,k}$, then if $S_{j,h}$ does not interfere with $P_{i,k}$, it does not interfere with $sp(P_{i,k}, S_{i,k})$ either.*

Proof. To simplify notations, in this proof, we denote $P_{i,k}$ by P and $sp(P_{i,k}, S_{i,k})$ by Q. Suppose $S_{i,k}$ and $S_{j,h}$ are UPDATE statements. We characterize $S_{i,k}$ as

$$\{P\}\ T := (T - r) \cup r'\ \{Q\},$$

where Q is the strongest postcondition of $S_{i,k}$ with precondition P, given by [14]:

$$Q : \exists v(P_v^T \ \wedge\ T = (v - r) \cup r'). \tag{4}$$

Since $S_{j,h}$ does not interfere with P, the triple

$$\{P \wedge P'\}\ S_{j,h}\ \{P\}, \tag{5}$$

where P' is the precondition of $S_{j,h}$, is a theorem. Our goal is to show that

$$\{Q \wedge P'\}\ S_{j,h}\ \{Q\} \tag{6}$$

is also a theorem. Suppose q is an arbitrary state satisfying $Q \wedge P'$, and q' is the state that results after $S_{j,h}$ is executed starting in q. We would like to show that q' satisfies Q. Let v_0 be a value such that the assertion

$$P_{v_0}^T \ \wedge\ T = (v_0 - r) \cup r'$$

is true in q. The tuples in T in state q can be divided into two disjoint subtables: $st1$ and $st2$, where $st2 = r'$ and $st1 \cap r = \Phi$. Since no tuple in r' satisfies $pd_{j,h}$, $S_{j,h}$ can only change the value of $st1$. Suppose in the transition from q to q', $st1$ is changed to $st1'$. Then, in state q', $T = st1' \cup r'$. Since $S_{j,h}$ does not produce any tuple that satisfy $pd_{i,k}$, $st1' \cap r = \Phi$. Let $v_1 = st1' \cup r$. We have $T = (v_1 - r) \cup r'$ in state q'. Thus,

$$\{P_{v_0}^T \ \wedge\ (T = (v_0 - r) \cup r') \wedge P'\}\ S_{j,h}\ \{T = (v_1 - r) \cup r'\} \tag{7}$$

is a theorem. Since $pd_{j,h} \cap r' = \Phi$, the following triple must also be a theorem:

$$\{P_{v_0}^T \ \wedge\ (T = v_0 - r) \wedge P'\}\ S_{j,h}\ \{T = v_1 - r\}.$$

According to the conditions of the lemma, $pd_{i,k} \cap pd_{j,h} = \Phi$, thus, we get:

$$\{P_{v_0}^T \ \wedge\ T = v_0 \wedge P'\}\ S_{j,h}\ \{T = v_1\}. \tag{8}$$

From (5) and (8), we get

$$\{P_{v_0}^T \ \wedge\ T = v_0 \wedge P'\}\ S_{j,h}\ \{P_{v_1}^T \wedge T = v_1\}. \tag{9}$$

Since $pd_{i,k} \cap pd_{j,h} = \Phi$ and $pd_{j,h} \cap r' = \Phi$, from (9) we have

$$\{P_{v_0}^T \ \wedge\ T = (v_0 - r) \cup r' \wedge P'\}\ S_{j,h}\ \{P_{v_1}^T \wedge T = (v_1 - r) \cup r'\}. \tag{10}$$

From (10), we get (4) is true in state q'. Since q is an arbitrary state satisfying $Q \wedge P'$, (6) is a theorem. For other write statements, similar proofs can be carried out.

The semantic condition for the correct execution of a transaction at READ UNCOMMITTED in a relational database is the same as the one for the page database, although the proof is different since we need to consider phantoms. □

Theorem 6. *The execution of a transaction T_i at READ UNCOMMITTED will be semantically correct if each write statement (including those that rollback a transaction) in every transaction does not interfere with I_i, the postcondition of every read statement in T_i, and Q_i.*

Proof. Let $S_{i,k}$ be an arbitrary write statement (INSERT, DELETE, UPDATE) of T_i and we characterize it as triple $\{P\}\ S_{i,k}\ \{Q\}$, where Q is the strongest postcondition of $S_{i,k}$ with precondition P. Suppose $S_{j,h}$ is an arbitrary write statement (INSERT, DELETE, UPDATE) of T_j and it does not interfere with P. In the following, we prove that $S_{j,h}$ will not interfere with Q either, or if it does, it will not invalidate Q during execution.

Consider $S_{j,h}$ is interleaved after $S_{i,k}$. If $S_{i,k}$ is the last statement of T_i, $Q \equiv I_i \bigwedge Q_i$, according to the conditions of the theorem, Q is not interfered with by $S_{j,h}$. If not, since we assume the predicates in the WHERE clauses of $S_{i,k}$ and $S_{j,h}$, denoted by $pd_{i,k}$ and $pd_{j,h}$, respectively, are long-term, $pd_{i,k} \cap pd_{j,h} = \Phi$, otherwise, T_j will wait until T_i terminates. Furthermore, $S_{i,k}$ does not produce any tuples that satisfy $pd_{j,h}$ since, if it does, it will hold long-term write locks on the tuples that it produces, and T_j will be prevented from being interleaved right after it, as a matter of fact, T_j has to wait until T_i terminates; $S_{j,h}$ cannot produce any tuples that satisfy $pd_{i,k}$ either since, otherwise, the long-term predicate $pd_{i,k}$ will prevent $S_{j,h}$ from being interleaved right after $S_{i,k}$.

In any cases, either $S_{j,h}$ is delayed until T_i terminates, in which case, Q is not invalidated; or the conditions of Lemma 1 are satisfied, thus, Q will not be interfered with. The rest of the proof would be similar to the one for Theorem 2. □

The semantic condition for READ COMMITTED of relational databases is the same as the one for the page database, although the proof is different since we need to consider phantoms.

Theorem 7. *The execution of a transaction T_i at READ COMMITTED will be semantically correct if each transaction does not interfere with the postcondition of every read statement in T_i, and with Q_i.*

Proof. Let T_j be an arbitrary transaction in the system. Since the isolation level of T_j is at least READ UNCOMMITTED, T_j will hold a long-term write lock on any item it writes. Since at the READ COMMITTED level, T_i uses short-term read locks, T_i cannot read any item written by T_j until T_j terminates, thus, each read statement of T_i either sees the whole result of T_j or it does not see any result of T_j, and when we reason about the semantic correctness of a schedule that includes T_i, as far as T_i is concerned, T_j can be considered as a single isolated unit.

2. Despite $pd_{i,k} \cap pd_{j,h} = \Phi$, the tuples that $S_{i,k}$ writes still might satisfy $pd_{j,h}$.

Let $S_{i,k}$ be an arbitrary write statement (INSERT, DELETE, UPDATE) of T_i and we characterize it as triple $\{P\}\ S_{i,k}\ \{Q\}$, where Q is the strongest postcondition of $S_{i,k}$ with precondition P. Suppose T_j does not interfere with P. In the following, we prove that T_j will not interfere with Q either, or if it does, it will not invalidate Q during execution.

Consider T_j is interleaved after $S_{i,k}$. If $S_{i,k}$ is the last statement of T_i, $Q \equiv I_i \bigwedge Q_i$, according to the conditions of the theorem, Q is not interfered with by T_j. If not, since we assume the predicates in the WHERE clauses of $S_{i,k}$ and T_j are long-term, they must not intersect, otherwise, T_j will wait until T_i terminates. Furthermore, $S_{i,k}$ does not produce any tuples that satisfy the predicate of any write statement in T_j; otherwise, it will hold long-term write locks on the tuples that it produces, and T_j will be prevented from being interleaved right after it and T_j has to wait until T_i terminates. T_j cannot produce any tuples that satisfy $pd_{i,k}$ either, otherwise, the long-term predicate $pd_{i,k}$ will prevent $S_{j,h}$ from being interleaved right after $S_{i,k}$.

In any cases, either T_j is delayed until T_i terminates, in which case, Q is not invalidated; or the conditions of Lemma 1 are satisfied, thus, Q will not be interfered with. The rest of the proof would be similar to the one for Theorem 3. □

For READ COMMITTED with first-committer-wins, taking into account the effect of phantoms, the conditions have to be strengthened to be the one for READ COMMITTED (the same as Theorem 7).

The semantic condition for REPEATABLE READ in page databases can be restated for relational databases by considering the possibility of phantoms. At REPEATABLE READ, the long-term read locks obtained on tuples read by a SELECT statement block the execution of a statement in a concurrent transaction that attempts to delete or update such tuples. Hence, the postcondition of the SELECT statement cannot be invalidated by a transaction that attempts to delete or update such a tuple. As a result, we get the following theorem.

Theorem 8. *For a transaction, T_i, executed at REPEATABLE READ, let $S_{i,j}$ be an arbitrary SELECT in T_i. T_i will execute semantically correctly if each transaction does not interfere with Q_i and either 1) does not interfere with the postcondition of $S_{i,j}$, or 2) includes DELETE/UPDATE statements whose predicates intersect the predicate of $S_{i,j}$.*

Proof. If 1) applies, similar proof as the one for Theorem 7 exists; otherwise, 2) applies, and T_j will be delayed until T_i terminates, where Q_i is not interfered with. □

For SNAPSHOT isolation, phantoms will not occur in both page and relational databases. Theorem 5 remains valid for relational databases, so does the proof.

5 Choosing an Isolation Level

Given the set of transaction types of an application, the problem faced by the application designer is to determine, for each type, the lowest isolation level at which a transaction of that type can execute correctly. Since SNAPSHOT isolation is not generally offered in the context of the other isolation levels, we do not consider it in what follows.

```
$Mailing\_List()$

BEGIN TRANSACTION

        $\{true\}$

    SELECT cust_name, address FROM CUST

        $\{Returned\ data\ contains\ names\ and\ addresses\}$

    $Print\ labels\ using\ returned\ names\ and\ addresses$

        $\{Labels\ have\ been\ printed\}$

END TRANSACTION
```

Fig. 3. Prints out a mailing list.

Using the previous results, it follows that, while we determine the isolation level at which to execute transaction, T_1, we do not have to be concerned about the level of other transactions. Specifically, we are performing an interference analysis to determine the correctness of executing T_1 at READ UNCOMMITTED, we must consider the interference effects of each write of another transaction, T_2, individually, regardless of the level of T_2. Similarly, if we are considering executing T_1 at any higher level, we consider the interference effects of the whole transaction T_2 as an atomic isolated unit, regardless of the level of T_2. Then, a procedure for determining the lowest isolation level at which each transaction can execute semantically correctly is: For each transaction, T_i, in the set, consider the isolation levels READ UNCOMMITTED (Theorem 6), READ COMMITTED (Theorem 7), REPEATABLE READ (Theorem 8), and SERIALIZABLE in sequence and return the first at which execution is semantically correct.

6 An Example

To motivate the conditions for semantic correctness in a relational setting, consider a business application that accesses a schema with the following three tables (primary keys are underlined):

◇ ORDERS(<u>`order_info`</u>, `cust_name`, `deliv_date`, `done`)
◇ CUST(<u>`cust_name`</u>, `address`, `#orders`)
◇ MAXDATE(<u>`maximum_date`</u>)

A conjunct of I, I_o, asserts that each row of ORDER describes an order and `done` is true if that order has been delivered. MAXDATE is a table containing a single row that satisfies a second conjunct, I_{max}, that asserts that `maximum_date` is the maximum delivery date for any order in ORDERS. #orders records the number of orders for each customer.

This application has four transaction types shown in Figs. 3, 4, 5, and 6. Each figure shows an annotation of a transaction program indicating the pre and postconditions of the transaction and the postcondition of each SELECT statement. These are the critical assertions. The purpose of

```
$New\_Order(customer,\ address,\ order\_info)$
BEGIN TRANSACTION
        $\{no\_gap\ \wedge\ order\_consistency \wedge I_{max}\}$
    SELECT maximum_date FROM MAXDATE
    INTO  $:maxdate$
        $\{no\_gap\ \wedge\ order\_consistency$
        $\wedge\ I_{max} \wedge (maxdate \leq \texttt{maximum\_date})\}$
    UPDATE MAXDATE SET
    maximum_date $= :maxdate + 1$
    SELECT COUNT(*) INTO $:custcount$ FROM
    ORDERS WHERE cust_name $= :customer$
        $\{no\_gap\ \wedge\ order\_consistency\ \wedge\ I'_{max}$
        $\wedge(maxdate\ \leq \texttt{maximum\_date}) \wedge$
        $(custcount = 0) \Rightarrow\ (customer\ is\ new)\}$
    if (custcount == 0) then
        INSERT INTO CUST
        VALUES $(:customer,\ :address,\ 1)$
    else
        UPDATE CUST SET #orders = $:custcount+1$
        WHERE cust_name $= :customer$
    fi
    INSERT INTO ORDERS VALUES
    $(:order\_info,\ :customer,\ :maxdate + 1,\ false)$
        $\{no\_gap \wedge order\_consistency \wedge I_{max} \wedge$
        $(customer,\ address,\ custcount + 1) \in cust$
        $\wedge\ (order\_info,\ customer,\ maxdate + 1,\ false)$
        $\in\ orders\}$
END TRANSACTION
```

Fig. 4. Processes a new order.

```
$Delivery(today)$
BEGIN TRANSACTION
        $\{true\}$
    SELECT order_info INTO $:buff$ FROM ORDERS
    WHERE deliv_date $= :today$ AND done = FALSE
        $\{returned\ values\ are\ undelivered\ orders$
        $to\ be\ delivered\ today\}$
    while (($ord\_inf$ := next in $buff$)
        UPDATE ORDERS SET done = TRUE
        WHERE order_info $= :ord\_inf$
            $\{tuples\ in$ ORDERS $describing\ orders$
            $due\ today\ have$ done = TRUE$\}$
END TRANSACTION
```

Fig. 5. Delivers an order.

the figures is to display the critical assertions; the code is just sketched.

Mailing_List (Fig. 3): This transaction scans CUST and prints a label using cust_name and address. The specification of the transaction places no condition on the labels printed. Since no critical assertion is interfered with by any single write statement in any of the transaction types, this transaction will execute correctly at READ UNCOMMITTED.

New_Order (Fig. 4): This transaction inserts a new order into ORDERS and, if this is the first order for *customer*, inserts a new tuple into CUST. In order to keep the delivery truck busy, a business rule asserts that there can be no gaps in the sequence of delivery dates: There must be at least one order to be delivered on each date up to some maximum date which is the delivery date of the last outstanding order. A conjunct of the integrity constraint of the database, which we call "no_gaps," asserts this fact. However, there can be more than one order for any particular delivery date. Furthermore, the number of orders for a particular customer in ORDERS must be equal to the value of the #orders field of that customer's tuple in CUST. We refer to this integrity constraint as "$order\_consistency$." The intermediate assertion I'_{max} in Fig. 4 asserts that maximum_date is one greater than the latest delivery date in ORDERS. Thus, New_Order reads the value of maximum_date in MAXDATE into the workspace variable *maxdate*; and increments maximum_date in MAXDATE by 1. If the customer is new, it inserts the tuple $(customer, address, 1)$ into CUST; otherwise, it increments #orders in the customer's tuple in CUST.[3] Finally, it inserts $(order\_info, customer, maxdate + 1, false)$ into ORDERS.[4]

Since no critical assertion is interfered with by any transaction type, this transaction can run at READ COMMITTED. The transaction cannot run at READ UNCOMMITTED because, for example, the no_gap assertion that is a conjunct of assertions in a New_Order transaction, T_1, is interfered with by the rollback statement of another New_Order transaction, T_2, that deletes a tuple from ORDERS (it might leave a gap in delivery dates below the delivery date selected by T_1).

3. The postcondition of New_Order in Fig. 4 indicates that the inserted tuple has a particular value in the #orders field. Since the value will change as the customer adds new orders, in order to avoid interference, the postcondition should actually be weakened to assert that, *at commit time*, this tuple was an element of CUST.

4. Since the value of done will subsequently change, the comments in the previous footnote apply.

```
Audit(customer)
BEGIN TRANSACTION
        {$I_o$}
    SELECT COUNT(*) INTO : count1 FROM ORDERS
    WHERE cust_name = : customer
        {$I_o \wedge$ (count1 = the number of tuples
        in ORDERS for customer)}
    SELECT #orders INTO : count2 FROM CUST
    WHERE cust_name = : customer
        {$I_o \wedge$ (count1 = the number of tuples in
        ORDERS for customer) $\wedge$ (count2 = the value
        of #orders in CUST for customer)}
    retv := (count1 == count2);
        {$I_o \wedge$ (retv = order_consistency)}
END TRANSACTION
```

Fig. 6. Produces accounting information.

Suppose an additional business rule is imposed: there must be *exactly* one order with a particular delivery date for each date up to some maximum. The "*no_gap*" conjunct of the integrity constraint is replaced by the conjunct "*one_order_per_day*" which asserts the new requirement. The same *New_Order* transaction can be used to enforce this rule if it is run at READ COMMITTED with first-committer-wins. At READ COMMITTED, the INSERT into ORDERS in the *New_Order* transaction interferes with the conjunct *one_order_per_day* in the postcondition of the SELECT. However, since *New_Order* updates MAXDATE after reading it, *one_order_per_day* cannot be invalidated at the READ COMMITTED with first-committer-wins isolation level. Also note the postcondition of the whole transaction is not interfered with by any transaction type and, thus, this transaction can run at READ COMMITTED with first-committer-wins.

Delivery (Fig. 5): This transaction delivers an order. Thus, *Delivery* scans ORDERS to find all the orders that are due today and updates the done attributes in the orders to be delivered to *true*.

The postcondition of the SELECT statement of a *Delivery* transaction is interfered with by another *Delivery* transaction. Thus, this transaction type cannot execute at READ COMMITTED. However, if the transaction is executed at REPEATABLE READ, the selected tuples are read locked after the SELECT statement is executed. Hence, a *Delivery* transaction would not be allowed to update these tuples and the assertion could not be invalidated. Thus, this transaction meets the condition for correct execution at the REPEATABLE READ isolation level.

Audit (Fig. 6): This transaction checks that *order_consistency* is true. Thus, *Audit* scans ORDERS and counts the number of orders registered for a particular customer and reads the tuple for that customer in CUST and compares #orders with the count.

This transaction must run at the SERIALIZABLE level because the postconditions of both SELECT statements might be interfered with by a *New_Order* transaction that inserts a (phantom) new order. Note that this transaction does not satisfy the second half of the condition for correct execution at REPEATABLE READ because tuple locks do not prevent the insertion of a phantom new order.

7 Conclusions and Future Work

We have used semantic correctness as the criterion to investigate the correctness of schedules at different isolation levels. Specifically, for each isolation level, we prove a condition under which transactions that execute at that level will be semantically correct. This technique also clarifies the relationship between interference and invalidation. Interference does not necessarily lead to invalidation because the underling locking scheme might prevent the offending interleavings from happening. Furthermore, an assertion that is interfered with can often be replaced by a stronger assertion that is not interfered with. In that case, the weaker assertion is not invalidated. Our analysis is based on the assumption that all isolation levels except SNAPSHOT isolation are implemented using a locking scheme.

To promote the practical significance of the results in this paper, we are looking at Nipkow's recent work on formalizing Hoare logic in Isabelle/HOL [16], one of the most popular theorem provers. We plan to develop a tool based on Isabelle/HOL to automate the correctness analysis of the execution of transactions at various isolation levels.

References

[1] ANSI X3. 135-1992, Am. Nat'l Standard for Information Systems-Database Languages-sql, Nov. 1992.

[2] K. Apt, "Ten Years of Hoare's Logic: A Survey—Part I," *ACM Trans. Programming Language Systems,* vol. 3, no. 4, pp. 431-483, 1981.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," *Proc. 1995 ACM SIGMOD Int'l Conf. Management of Data,* pp. 1-10, May 1995.

[4] J. Bergstra and J. Tucker, "Expressiveness and Completeness of Hoare's Logic," *J. Computer and System Sciences,* vol. 25, no. 3, pp. 267-284, Dec. 1982.

[5] A. Bernstein, D. Gerstl, W.H. Leung, and P. Lewis, "Design and Performance of an Assertional Concurrency Control System," *Proc. 14th IEEE Int'l Conf. Data Eng.,* pp. 436-445, Feb. 1998.

[6] A. Bernstein, D. Gerstl, and P. Lewis, "A Concurrency Control for Step-Decomposed Transactions," *Information Systems,* vol. 24, no. 8, pp. 673-698, 1999.

[7] A. Bernstein, D. Gerstl, P. Lewis, and S. Lu, "Using Transaction Semantics to Increase Performance," *Proc. Eighth Int'l Workshop High Performance Trans. Systems,* Sept. 1999.

[8] *Concurrency in Programming and Database Systems.* A. Bernstein and P. Lewis, eds., Jones and Bartlett Publishers, 1993.

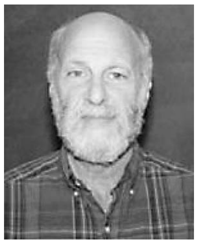
[9] A. Bernstein and P. Lewis, "High Performance Transaction Systems Using Transaction Semantics," *Distributed and Parallel Databases,* vol. 4, no. 1, Jan. 1996.

[10] A. Bernstein, P. Lewis, and S. Lu, "Semantic Conditions for Correctness at Different Isolation Levels," *Proc. 16th IEEE Int'l Conf. Data Eng.,* pp. 57-66, Mar. 2000.

[11] S. Cook, "Soundness and Completeness of an Axiom System for Program Verification," *SIAM J. Computing,* vol. 7, no. 1, pp. 70-90, Feb. 1978.

[12] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM,* vol. 19, no. 11, pp. 624-633, Nov. 1976.
[13] J. Gray, R. Lorie, G. Putzolu, and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database," *Modeling in Data Base Management Systems,* Amsterdam: Elsevier North-Holland, 1976.
[14] *The Science Of Programming,* D. Gries, ed., New York: Springer-Verlag, 1981.
[15] *Logic in Computer Science: Modelling and Reasoning about Systems,* M. Huth and M. Ryan, eds., Cambridge Univ. Press, 2000.
[16] T. Nipkow, "Hoare Logics in Isabelle/HOL," *Proof and System-Reliability,* H. Schwichtenberg and R. Steinbrüggen, eds., pp. 341-367, Kluwer, 2002.
[17] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica,* vol. 6, pp. 319-340, 1976.

Shiyong Lu received the PhD degree in the Department of Computer Science from the State University of New York at Stony Brook in 2002. He joined the faculty of Wayne State University in 2002, where he is currently an assistant professor in the Department of Computer Science. His research interests includes transaction processing and workflow management systems and, recently, the semantic Web and bioinformatics. He has published numerous papers in major international conferences and journals in the above areas. He is a member of the IEEE.



Arthur Bernstein received the PhD degree in electrical engineering from Columbia University in 1962. He is currently a professor in the Computer Science Department of the State University of New York at Stony Brook. His research interests include transaction processing and the analysis and synthesis of workflows. He is a fellow of the IEEE.

Philip Lewis received the BEE degree from Rensselaer Polytechnic Institute in 1952 and the MS and PhD degrees in electrical engineering from the Massachusetts Institute of Technology in 1954 and 1956, respectively. He is currently a professor in the Computer Science Department of the State University of New York at Stony Brook. His research interests include databases and transaction processing, concurrency, and software engineering. He is the author of about 50 journal articles and four textbooks. He is a fellow of both the IEEE and the ACM.