On Correctness of Non-serializable Executions

Rajeev Rastogi
Sharad Mehrotra
Yuri Breitbart
Henry F. Korth
Avi Silberschatz

1 Introduction
In the standard transaction model [BH87], a database state is said to be consistent if all database integrity constraints are satisfied. Each transaction, when executed in isolation, maps a consistent database state to another consistent database state. In the case of concurrent transaction executions, database consistency is ensured by requiring that the resulting execution be serializable; that is, equivalent to some serial execution of the transactions. Since each transaction, when executed alone, preserves database consistency, a serializable execution preserves database consistency.

While this theory is attractive from the point of view of simplicity, in recent years, several papers have shown that employing the standard transaction model for certain "new" database applications or in a distributed database environment has unacceptable practical consequences. For example, in applications such as computer-aided design, where transactions are of a long duration, the standard transaction model may result in long-duration waits imposed by the locking protocols. In distributed database applications, a potential loss of the local autonomy of the participating databases may result. These problems are not only issues concerning performance but also issues concerning usability, since many of the "new" applications are interactive and/or distributed.

The approaches taken to alleviate the above mentioned problems can be broadly classified into the following three categories:

- The first approach is to exploit the semantics of the database operations, while still ensuring serializability of executions. A transaction can be considered as a sequence of high-level operations (e.g., increment.) Commutativity among operations, and not the set of read and writes resulting from their execution, is used to determine conflicts between transactions, thus enhancing concurrency. Early work in this area includes [Kor83]. More recent work includes [Wei88, Her90, BR87, CRR91].

- The second approach is to exploit the semantics of transactions [FO89, GM83, GMS87]. In this approach, a transaction is broken into a set of subtransactions, with each of which a type is associated. The application administrator, then determines the set of interleavings of the subtransactions that are acceptable and that would not result in the violation of database consistency. As an example, consider the saga transaction model in which a transaction $T$ is broken into a sequence of subtransactions $T_1, T_2, \ldots, T_n$. Each $T_i$ is an independent activity by itself. After the termination of $T_i$ the locks on data items held by $T_i$ can be released and the effects of $T_i$ externalized. Thus, in the saga transaction model all possible interleavings of the subtransactions are permitted.

- The third approach is to exploit the knowledge of the integrity constraints of the system. A fundamental technique used in several application domains is to partition the database into subdatabases, each with its own consistency constraints. Based on the application domain, assumptions can be made regarding the degree to which these subdatabases are independent (as in the concept of global and local data used in some work in multidatabase systems [MRKS91]) or regarding the nature of transactions executing against these subdatabases (such as an assumption of preservation of subdatabase consistency).
such application domain is the computer-aided design and manufacturing environment for which \textit{predicatewise serializability} (PWSR) was introduced as a correctness criterion in [KKKB88]. In a nutshell, the PWSR correctness criterion states that if the database consistency constraint is expressed as a conjunction of predicates, then for each possible schedule \( S \), the restriction of \( S \) to operations that access data items in every conjunct is serializable.

While substantial research has been done towards developing approaches to relax the serializability requirement, very little work has been done on identifying what consistency guarantees result from the above approaches to extending the standard transaction model. For example, little work exists on characterising the application domains in which the usage of the saga transaction model ensures database consistency.

In this paper, we study the formal implications of choosing PWSR as the correctness criterion. The importance of the PWSR correctness criterion has been demonstrated in [KKKB88, BGMS92]. In [KKKB88], it was shown that the PWSR correctness criterion can be used to alleviate the long-duration waits associated with long-duration transactions in computer-aided design and manufacturing environments. More recently, in [BGMS92] it was shown that the PWSR correctness criterion (referred to as the local serializability (LSR) criterion) can be used to help alleviate the autonomy-induced problems in \textit{multidatabase} system applications. In this paper, we explore the formal implications of the assumptions that may be required in order to ensure database consistency when PWSR is employed as the correctness criterion. The assumptions examined include:

- Partitioning of the database into subdatabases.
- Restricting the form of consistency constraints.
- Restricting the structure of programs that generate transactions.
- Restricting the concurrent execution of transactions.
- Restricting the order in which transactions access data.

We show what combination of assumptions are and are not sufficient to guarantee consistency.

The remainder of the paper is organized as follows. In Section 2, we develop our transaction and schedule models that are suited for proving that non-serializable executions preserve database consistency. In Section 3, we identify the restrictions that must be placed on PWSR schedules so that database consistency is preserved. Concluding remarks are offered in Section 4.

2 Preliminaries

In this section, we develop the basic notation that will be used in the remainder of the paper. We first formalize the notion of database consistency in terms of the preservation of integrity constraints. We then develop our transaction model to help us in reasoning about non-serializable executions. Finally, we discuss our notion of correctness of a schedule.

2.1 Database Consistency

In the standard transaction model [Pap86], a consistent database state is implicitly defined by assuming that each transaction, when executed in isolation, maps a consistent database state to another consistent database state. Correctness in case of concurrent execution is defined in terms of serializability. In order to develop a theory of non-serializable executions, we must explicitly define what a consistent database state is. We do this in terms of integrity constraints, which are discussed below.

A database consists of a finite set, \( D \), of data items. For each data item \( d' \in D \), \( Dom(d') \) denotes the domain of \( d' \). A \textit{database state} maps every data item \( d' \) to a value \( v' \), where \( v' \in Dom(d') \). Thus, a database state, denoted by \( DS \), can be expressed as a set of ordered pairs of data items in \( D \) and their values,

\[
DS = \{(d', v') : d' \in D \text{ and } v' \in Dom(d')\}.
\]

\( DS \) has the property that if \((d', v_1') \in DS\) and \((d', v_2') \in DS\), then \( v_1' = v_2' \). The restriction of \( DS \) to data items in \( d \subseteq D \), is denoted by \( DS^{d} \). Thus, \( DS^{d} = \{(d', v') : d' \in d \text{ and } (d', v') \in DS\} \). Let \( d_1 \subseteq D \), \( d_2 \subseteq D \), and \( DS_{1}, DS_{2} \) be database states. The union of \( DS_{1} \) and \( DS_{2} \) is denoted by \( DS_{1} \cup DS_{2} \). The \( \cup \) operation is similar to the one traditionally defined for sets, except that \( DS_{1} \cup DS_{2} \) is undefined if \((d', v_1') \in DS_{1}, (d', v_2') \in DS_{2} \), and \( v_1' \neq v_2' \).

Integrity constraints in a database, denoted by \( IC \), distinguish inconsistent database states from consistent ones. Traditionally, integrity constraints are defined as a subset of all the possible database states, and a database state is consistent if it belongs to that subset [Pap86]. In our model, integrity constraints are quantifier-free first order formulae over the language consisting of:

- Numerical and string constants (e.g., 5, 100, 'Jim'),
- Functions over numeric and string constants (e.g., +, max),
- Comparison operators (e.g., >, =), and
- Set of variables (data items in \( D \)).

The terms and well-formed formulae are defined as in [Apt90]. Since a database state maps data items (variables) to values it can be viewed as a \textit{variable assignment} [Apt90].

98
A database state, $DS$, is consistent if it does not violate the integrity constraints. Let $I$ be the standard interpretation for numerical and string constants, function symbols, and comparison operators. Thus, a database state $DS$ is consistent iff $I \models DS IC$. We shall use $DS \models IC$ to denote $I \models_{DS} IC$. For example, consider a database consisting of data items: $a$, $b$ and an integrity constraint: $IC = (a = b)$. The database state $DS_1 = \{(a, 5), (b, 5)\}$ is consistent. However, the database state $DS_2 = \{(a, 5), (b, 6)\}$ is not consistent. The restriction of a database state, $DS$, to data items in $d \subseteq D$, $DS^d$ is consistent iff there exists a consistent database state $DS_1$ such that $DS_1^d = DS^d$. Note that even though $DS$ is not consistent, its restrictions may be consistent. For example, $DS_2$ above is not consistent. However, its restrictions, $DS_2^{(a)} = \{(a, 5)\}$ and $DS_2^{(b)} = \{(b, 6)\}$ are consistent. In the following lemma, we establish the relationship between the consistency of database states and consistency of its subsets.

Lemma 1: Let $IC = C_1 \land C_2 \land \cdots \land C_l$, where $IC, C_e$ are defined over data items in $D$, $d_e$ respectively such that $d_e \cap d_f = \emptyset$ for all $e \neq f$. Let $d'_e \subseteq d_e$ and $DS$ be a database state. $\bigcup_{e=1}^l DS^d_e$ is consistent iff for all $e = 1, 2, \ldots, l, DS^d_e$ is consistent.

Proof:

$\Leftarrow$: If $\bigcup_{e=1}^l DS^d_e$ is consistent, then for all $e = 1, 2, \ldots, l, DS^d_e$ is consistent. This follows directly from the definition of database consistency.

$\Rightarrow$: We now prove that if for all $e = 1, 2, \ldots, l, DS^d_e$ is consistent, then $\bigcup_{e=1}^l DS^d_e$ is consistent. Since $DS^d_e$ are consistent, there exist consistent database states, $DS_e$, such that $DS^d_e = DS^d$, $e = 1, 2, \ldots, l$. Let $DS_0$ be a database state such that $DS_0^d = DS^d$, $e = 1, 2, \ldots, l$ (such a $DS_0$ exists since $d_e \cap d_f = \emptyset, e \neq f$). Since $DS_e \models C_e$, $DS_0^d = DS^d_c$, and $C_e$ is defined only over data items in $d_e$, $DS_0 \models IC_e$, $e = 1, 2, \ldots, l$. Thus, $DS_0 \models IC = C_1 \land C_2 \land \cdots \land C_l$. Also, $\bigcup_{e=1}^l DS^d_e \subseteq DS_0$. Thus, there exists a consistent database state $DS_0$ such that $\bigcup_{e=1}^l DS^d_e \subseteq DS_0$. Hence, by definition of database consistency, $\bigcup_{e=1}^l DS^d_e$ is consistent. $\Box$

Note that it is essential for the data items over which conjuncts are defined to be disjoint, if Lemma 1 is to hold. If data items over which conjuncts are defined are not disjoint, then database state $DS_0$ in the proof of Lemma 1 may not exist. For example, let $IC = ((a = 5 \rightarrow b = 5) \land (c = 5 \rightarrow b = 6))$. Consider $d'_1 = \{a\}$ and $d'_2 = \{c\}$. Let $DS^d_1 = \{(a, 5)\}$ and $DS^d_2 = \{(c, 5)\}$. Thus, even though $DS^d_1$ and $DS^d_2$ are consistent, $DS^d_1 \cup DS^d_2$ is inconsistent.

2.2 Transactions and Schedules

In this section, we develop the transaction model, which deviates from the standard development to facilitate reasoning about non-serializable executions. A transaction is a sequence of operations resulting from the execution of a transaction program. A transaction program is usually written in a high-level programming language with assignments, loops, conditional statements and other complex control structures. Execution of a transaction program starting at different database states may result in different transactions. This observation has serious implications when dealing with non-serializable executions as will become evident later in the paper.

Formally, a transaction $T_i = (O_T, \prec_T)$, where $O_T = \{o_1, o_2, \ldots, o_n\}$ is a set of operations and $\prec_T$ is a total order on $O_T$. An operation $o_i$ is a 3-tuple $(action(o_i), entity(o_i), value(o_i))$. $action(o_i)$ denotes an operation type, which is either a read ($r$) or write ($w$) operation. entity($o_i$) is the data item on which the operation is performed. If the operation is a read operation, value($o_i$) is the value returned by the read operation for the data item read. For a write operation, value($o_i$) is the value assigned to the data item by the write operation. For simplicity of the exposition, we assume that each transaction reads and writes a data item at most once and that it does not read a data item after writing the data item.

Our transaction definition differs from the way they are traditionally defined in the literature (see, for example, [BH97], [Pap86]). We include, along with every operation, a value attribute, in addition to action and entity attributes. Since we relax the requirement of serializability as the correctness criterion, we need to deal with certain non-serializable executions. The value attribute helps us in proving that such non-serializable executions preserve database consistency.

A schedule is a sequence of operations resulting from the concurrent execution of a set of transaction programs. Formally, a schedule $S = (T_1, \prec_S)$ is a finite set $\tau_S$ of transactions, together with a total order, $\prec_S$, on all operations of the transactions such that for any two operations $o_1, o_2$ in $S$ and some transaction $T_i \in \tau_S$, if $o_1 \prec_S T_i$, then $o_1 \prec_S o_2$. We use the notation $\{DS_1\} TP \{DS_2\}$ to denote the fact that when a transaction program $TP_1$ executes from a database state $DS_1$, it results in a database state $DS_2$. Similar notation is used to denote execution of operations, transactions and schedules (the intended meaning will be clear from the context). We next introduce some notation for sequences of operations. Let $seq = o_1 o_2 \cdots o_n$ be a subsequence of a schedule $S$ (seq may be a transaction), $p$ be an operation in $S$, and let $d \subseteq D$. We define the following:

- $\text{legal}(DS, seq)$: denotes if it is possible to execute seq from the database state $DS$ (since operations have values associated with them, execution of operations is possible only from certain database states). In order to formalize $\text{legal}(DS, seq)$, we first identify when the execution of a single operation $o_i$ is possible from a database state. A database state
DS is legal with respect to operation \( \alpha_i \), denoted by \( \text{legal}(DS, \alpha_i) \), if either one of the following conditions hold:

- \( \text{action}(\alpha_i) = w \),
- if \( \text{action}(\alpha_i) = r \), then \( (\text{entity}(\alpha_i), \text{value}(\alpha_i)) \in DS \).

Thus, if \( \text{legal}(DS, \alpha_i) \), then it is possible to execute \( \alpha_i \) from \( DS \). A database state \( DS \) is legal for a sequence of operations \( \text{seq} \); that is, \( \text{legal}(DS, \text{seq}) \) if the following two conditions hold:

- \( \text{legal}(DS, \alpha_1) \)
- if \( n > 1 \), then \( \text{legal}(DS', \alpha_2 \cdots \alpha_n) \), where \( \{DS\} \cup \{DS'\} \).

Thus, if a database state \( DS \) is legal with respect to a sequence of operations \( \alpha_1 \alpha_2 \cdots \alpha_n \) then it is possible to execute \( \alpha_1 \alpha_2 \cdots \alpha_n \) from \( DS \). Execution of a sequence of operations \( \alpha_1 \alpha_2 \cdots \alpha_n \) from a database state which is not legal with respect to \( \alpha_1 \alpha_2 \cdots \alpha_n \) is undefined.

- \( \text{RS}(\text{seq}) \): denotes the set of data items read by operations in \( \text{seq} \).
  \[
  \text{RS}(\text{seq}) = \{ y : \alpha_i \in \text{seq} \land y = \text{entity}(\alpha_i) \land \text{action}(\alpha_i) = r \}
  \]

- \( \text{read}(\text{seq}) \): denotes the database state “seen” as a result of the read operations in \( \text{seq} \).
  \[
  \text{read}(\text{seq}) = \{ (y, z) : \alpha_i \in \text{seq} \land y = \text{entity}(\alpha_i) \land z = \text{value}(\alpha_i) \land \text{action}(\alpha_i) = r \}
  \]

- \( \text{WS}(\text{seq}) \): denotes the set of data items written by operations in \( \text{seq} \).
  \[
  \text{WS}(\text{seq}) = \{ y : \alpha_i \in \text{seq} \land y = \text{entity}(\alpha_i) \land \text{action}(\alpha_i) = w \}
  \]

- \( \text{write}(\text{seq}) \): denotes the effects that the write operations in \( \text{seq} \) have on the database.
  \[
  \text{write}(\text{seq}) = \{ (y, z) : \alpha_i \in \text{seq} \land y = \text{entity}(\alpha_i) \land z = \text{value}(\alpha_i) \land \text{action}(\alpha_i) = w \}
  \]

- \( \text{seq}^d \): denotes the subsequence of \( \text{seq} \) consisting of all operations \( \alpha_i \) such that \( \text{entity}(\alpha_i) \in d \).

- \( \text{before}(\text{seq}, p, S) \): denotes the subsequence of \( \text{seq} \) consisting of all the operations that precede \( p \) in \( S \). If \( p \) belongs to \( \text{seq} \), then \( \text{before}(\text{seq}, p, S) \) includes \( p \).

- \( \text{after}(\text{seq}, p, S) \): denotes the subsequence of \( \text{seq} \) consisting of all operations not in \( \text{before}(\text{seq}, p, S) \).

- \( \text{depth}(p, S) \): denotes the number of operations preceding operation \( p \) (but not including \( p \)) in schedule \( S \).

- \( \text{struct}(\text{seq}) \): denotes the structure associated with \( \text{seq} \), which is derived from \( \text{seq} \) by ignoring the values associated with the operations in \( \text{seq} \). Thus every operation \( \alpha_i \) in \( \text{struct}(\text{seq}) \) is a 2-tuple \( (\text{action}(\alpha_i), \text{entity}(\alpha_i)) \).

- \( \tau_w(d, S) \): denotes the set of transactions in \( S \) that have at least one write operation on some data item in \( d \). Formally,
  \[
  \tau_w(d, S) = \{ T_i \in \tau_s : (WS(T_i) \cap d) \neq \emptyset \}
  \]

We illustrate the above notation in Example 1 below, where \( T_i \) is used to denote the transaction resulting from the execution of the transaction program \( TP_i \). Operations belonging to transaction \( T_i \) are subscripted by \( i \). Thus, a read operation on data item \( a \) belonging to transaction \( T_i \) is denoted by \( r_i(a, v) \), where \( v \) is the value returned by the read operation (this notation is used throughout the paper).

**Example 1:** Consider the following transaction programs \( TP_1 \) and \( TP_2 \).

\[
TP_1 : \quad \text{if}(a \geq 0) \text{ then } b := c \quad TP_2 : \quad d := a \quad \text{else } c := d
\]

Consider the schedule \( S \) below resulting from the execution of \( TP_1 \) and \( TP_2 \) from database state \( DS_1 = \{(a, 0), (b, 10), (c, 5), (d, 10)\} \).

\[
S : \quad r_2(a, 0) \quad r_1(a, 0) \quad w_2(d, 0) \quad r_1(c, 5) \quad w_1(b, 5)
\]

In \( S \), the transactions corresponding to \( TP_1 \) and \( TP_2 \) are as follows:

\[
T_1 : \quad r_1(a, 0) \quad r_1(c, 5) \quad w_1(b, 5) \\
T_2 : \quad r_2(a, 0) \quad w_2(d, 0)
\]

Note that \( \text{legal}(DS_1, S) \) and further \( \{DS_1\}S\{DS_2\} \), where \( DS_2 = \{(a, 0), (b, 5), (c, 5), (d, 0)\} \). Let \( p = w_2(d, 0) \). The following assertions can be made about transactions \( T_1 \) and \( T_2 \).

\[
\text{RS}(T_1) = \{a, c\} \quad \text{read}(T_1) = \{(a, 0), (c, 5)\} \\
\text{WS}(T_1) = \{b\} \quad \text{write}(T_1) = \{(b, 5)\} \\
T_1^{(b)} = w_1(b, 5) \quad \text{depth}(p, S) = 2 \\
\tau_w(\{a, b\}, S) = \{T_1\}
\]

\[
S^{(a, c)} = r_2(a, 0) \quad r_1(a, 0) \quad r_1(c, 5) \\
\text{before}(T_2, p, S) = r_2(a, 0) \quad w_2(d, 0) \\
\text{after}(T_1, p, S) = r_1(c, 5) \quad w_1(b, 5) \\
\text{struct}(T_1) = r_1(a) \quad r_1(c) \quad w_1(b) \quad \Box
\]
2.3 Strong Correctness

In the traditional transaction model, transaction programs, when executed in isolation, are assumed to be correct; that is, transactions preserve the integrity constraints of the database. The task of the concurrency control scheme is to ensure that schedules resulting from the concurrent execution of the transaction programs preserve database consistency. However, a concurrency control scheme which ensures that schedules preserve the database integrity constraints does not necessarily prevent transactions from "seeing" inconsistent database states. To overcome this, we define the notion of strong correctness, which requires that transactions in a schedule read consistent data values, in addition to the requirement that schedules preserve database integrity constraints.

Definition 1: A schedule $S$ is strongly correct iff

- for all consistent database states $DS_1$, if it is the case that $\{DS_1\}S\{DS_2\}$, then $DS_2$ is consistent, and
- for all transactions $T_i \in \tau_S$, $read(T_i)$ is consistent.

Every serializable schedule is strongly correct, but there are strongly correct schedules that are not serializable. In the next section, we show that PWSR schedules are strongly correct if transaction programs and integrity constraints are of a restricted nature. In the remainder of the paper, we assume that all transaction programs and transactions are correct.

3 Predicatewise Serializability

The notion of predicatewise serializability (PWSR) was introduced as an alternative consistency criterion to serializability for applications with long-duration transactions, CAD/CAM applications, etc. [KKB88]. Formally, PWSR is defined as follows.

Definition 2: Let $IC = C_1 \land C_2 \land \ldots \land C_l$, where $IC$, $C_e$ are defined over data items in $D$, $d_e$ respectively. A schedule $S$ is said to be PWSR if for all $e$, $e = 1, 2, \ldots , l$, $S^{d_e}$ is serializable.

Since PWSR schedules may not be serializable, they, in general, do not preserve database consistency. This is illustrated by the following example.

Example 2: Consider a database containing data items $D = \{a, b, c\}$ and the following transaction programs $TP_1$ and $TP_2$.

\[
TP_1 : \quad a := 1; \\
\text{if}(c > 0) \quad \text{then} \quad b := |b| + 1
\]

\[
TP_2 : \quad \text{if}(a > 0) \quad \text{then} \quad c := b
\]

Let $IC = (C_1 \land C_2)$, where $C_1 = (a > 0 \rightarrow b > 0)$ and $C_2 = (c > 0)$. The conjuncts are defined over disjoint sets of data items. Consider the following schedule resulting from the execution of $TP_1$ and $TP_2$ from database state $\{(a, -1), (b, -1), (c, 1)\}$.

\[
S : \quad w_1(a, 1) \quad r_2(a, 1) \quad r_3(b, -1) \quad w_2(c, -1) \quad r_4(c, -1)
\]

The database state resulting from the execution of the above schedule is $\{(a, 1), (b, -1), (c, -1)\}$, which is inconsistent. Thus, PWSR schedules may not be strongly correct.

In this section, we identify various restrictions that need to be placed on the structure of transaction programs, the structure of schedules, and the order in which transactions access data items, in order to ensure that PWSR schedules are strongly correct.

3.1 Restricting Transaction Programs

Let us examine the transaction programs whose concurrent execution resulted in a loss of database consistency in Example 2. In Example 2, transaction program $TP_1$ read the value of data item $c$ as being less than 0. Thus, it did not execute its write operation on data item $b$. One way of preventing this is to restrict transaction programs such that the structure of the transaction that results from their execution is independent of the state from which the transaction program executes. We refer to such transaction programs as fixed-structured transaction programs.

Definition 3: Transaction program $TP_1$ has fixed-structure if for all pairs $(DS_1, DS_2)$ of database states, $struct(T_1) = struct(T_2)$, where $T_1$ and $T_2$ are transactions resulting from the execution of $TP_1$ from $DS_1$ and $DS_2$ respectively.

Note that in Example 2, the transaction program $TP_1$ does not have fixed-structure. However, it is easy to see that $TP_1$ can be converted into the following fixed-structured transaction program $TP_1'$.

\[
TP_1' : \quad a := 1; \\
\text{if}(c > 0) \quad \text{then} \quad b := |b| + 1 \\
\text{else} \quad b := b
\]

If we replace $TP_1$ with $TP_1'$ in Example 2, then the schedule in the example would not be PWSR since the restriction of the schedule to the set of data items in conjunct $C_1$ would not be serializable.

It can be shown that PWSR schedules that result from the execution of fixed-structured transaction programs are strongly correct. To prove this, we need to first show that every transaction in a PWSR schedule $S$ "sees" a consistent database state. Since a PWSR schedule may not be serializable, correctness cannot be
simply proved using induction over the order in which transactions are serialized in the schedule. Rather, to prove that transactions see a consistent database state, we need to use induction over the order in which operations occur in the schedule. Thus, we need to show that for each operation \( p \) and transaction \( T_i \) in a schedule \( S \), the set of data items that \( T_i \) reads before the occurrence of \( p \), that is, \( \text{read}(\text{before}(T_i, p, S)) \), is consistent. It is trivial to see that the claim holds if \( p \) is the first operation in \( S \). Assume that the claim holds for every operation in \( S \) before the occurrence of \( p \). We need to show that for each transaction \( T_i \), \( \text{read}(\text{before}(T_i, p, S)) \) is consistent. Let \( p \) belong to some transaction \( T_j \) and further let entity\( (p) \in d_k \), where \( d_k \) is the set of data items in conjunct \( C_k \). By induction hypothesis, \( \text{read}(\text{before}(T_j, p, S)) \), for each \( T_j \neq T_i \) is consistent. We, therefore, only need to prove that \( \text{read}(\text{before}(T_i, p, S)) \) is consistent. To do so, we first identify the subset of data items in \( d_k \) that \( T_i \) can possibly read before the occurrence of \( p \). We refer to the above set as the view set of transaction \( T_i \), before operation \( p \), with respect to data items in \( d_k \).

**Lemma 2:** Let \( S \) be a schedule and \( d \subseteq D \) such that \( S^d \) is serializable. Let \( T_1, T_2, \ldots, T_n \) be a serialization order of transactions in \( S^d \) and \( p \) be an operation in \( S \). For all \( i = 1, 2, \ldots, n \), \( \text{RS}(\text{before}(T_i^d, p, S)) \subseteq V S(T_i, p, d, S) \), where

\[
V S(T_i, p, d, S) =
\begin{cases}
  d, & \text{if } i = 1 \\
  V S(T_{i-1}, p, d, S) \cup W S(\text{before}(T_{i-1}^d, p, S)) - W S(\text{after}(T_{i-1}^d, p, S)), & \text{if } i > 1
\end{cases}
\]

**Proof:** \( \text{RS}(\text{before}(T_i^d, p, S)) \subseteq d = V S(T_i, p, d, S) \). Thus, the result holds for \( T_1 \). To show that the result holds for any \( T_i \), \( i = 2, 3, \ldots, n \), we will show that if \( d' \in d \) and \( d' \notin V S(T_i, p, d, S) \), then \( d' \notin \text{RS}(\text{before}(T_i, p, S)) \). From the definition of the view set of a transaction, we have the following property about data items which do not belong to a transaction’s view set. If \( d' \notin d \), and \( d' \notin V S(T_i, p, d, S) \), then for some \( j < i \), \( d' \notin W S(\text{after}(T_j^d, p, S)) \) and for all \( k \), \( k = j + 1, \ldots, i - 1 \), \( d' \notin W S(T_k^d) \). Since \( d^d \) is serializable and \( T_j \) is serially before \( T_i \), if \( T_i \) reads \( d' \), then \( T_i \) must read the value of \( d' \) written by \( T_j \). Thus, before \( p \), \( T_i \) cannot read \( d' \); that is, \( d' \notin \text{RS}(\text{before}(T_i^d, p, S)) \).

We next associate the notion of a “state” with a transaction. The state associated with the transaction is a possible state of the data items in a conjunct that the transaction may have seen. The state seen by the transaction is an abstract notion and may never have been physically realized in a schedule.

**Definition 4:** Let \( S \) be a schedule and \( d \subseteq D \) such that \( S^d \) is serializable. Let \( T_1, T_2, \ldots, T_n \) be a serialization order of transactions in \( S^d \) and \( D_{S_1} \) be a database state such that \( \text{legal}(D_{S_1}, S) \). The state of the database before the execution of each transaction with respect to data items in \( d \) is defined as follows.

\[
\begin{align*}
\text{state}(T_i, d, S, D_{S_1}) &= \left\{ \begin{array}{ll}
D_{S_1}, & \text{if } i = 1 \\
\text{state}(T_{i-1}, d, S, D_{S_1})^{d-W S(\text{after}(T_{i-1}^d, p, S))} \cup \text{write}(T_{i-1}^d), & \text{if } i > 1
\end{array} \right.
\end{align*}
\]

In the above definition, \( \text{state}(T_{i-1}, d, S, D_{S_1})^{d-W S(\text{after}(T_{i-1}^d, p, S))} \) denotes the restriction of \( \text{state}(T_{i-1}, d, S, D_{S_1}) \) to data items in \( d - W S(\text{after}(T_{i-1}^d, p, S)) \). Thus, \( \text{state}(T_i, d, S, D_{S_1}) \) is the state of the database with respect to data items in \( d \) as seen by \( T_i \). Note that from the definition of state and serializability, it follows that

- \( \text{read}(T_i^d) \subseteq \text{state}(T_i, d, S, D_{S_1})^d \), and

- if it the case that \( \text{state}(T_i, d, S, D_{S_1})^d = \{D_{S_2}\} \), then it must be the case that \( \{\text{state}(T_i, d, S, D_{S_1})\} \subseteq \{D_{S_2}\} \).

The state of a transaction depends on the initial state and the serialization order chosen and thus, may not be unique. In Example 1, \( S \) is serializable with serialization orders \( T_1, T_2 \) or \( T_2, T_1 \). With serialization order \( T_1, T_2 \),

\[
\text{state}(T_2, \{a, b, c\}, S, D_{S_1}) = \{(a, 0), (b, 5), (c, 5)\}.
\]

However, with serialization order \( T_2, T_1 \),

\[
\text{state}(T_2, \{a, b, c\}, S, D_{S_1}) = \{(a, 0), (b, 10), (c, 5)\}.
\]

Recall that our task is to show that under the induction hypothesis, \( \text{read}(\text{before}(T_i, p, S)) \) is consistent, where \( p \) is an operation belonging to transaction \( T_i \) and further, entity\( (p) \in d_i \). In order to do so, we need to establish conditions under which database consistency is preserved during the execution of transactions. For an arbitrary transaction, it is difficult to make any assertion about the consistency of the database state during its execution, since all we know about a transaction is that, as an atomic unit, it is correct. However, if we restrict transactions to those resulting from the execution of fixed-structured transaction programs, we can make the following assertion about the states which exist during its execution.

**Lemma 3:** Let \( S \) be a schedule consisting of a transaction \( T_i \) which results from the execution of a fixed-structure transaction program \( TP_i \) (note that \( S = T_i \)). Let \( D_{S_1}, D_{S_2} \) be database states such that \( \{D_{S_1}\} \subseteq \{D_{S_2}\} \), and \( p \) be an operation in \( S \). If \( D_{S_1}^d \cup \text{read}(\text{before}(T_i, p, S)) \) is consistent, then \( D_{S_2}^{d-W S(\text{before}(T_i, p, S))} \) is consistent.

**Proof:** Let \( D_{S_0} \) be a consistent state such that \( D_{S_0}^{d-W S(\text{before}(T_i, p, S))} = D_{S_1}^d \cup \text{read}(\text{before}(T_i, p, S)) \). Let \( \{D_{S_3}\} TP_i \{D_{S_4}\} \). Let \( T_i \) be the transaction and

\footnote{This may not be true if \( S^d \) is final-state serializable (FSR) \cite{Pap86} but not VSR; however it is true if \( S^d \) is VSR, as we assume here.}
\( S' \) be the schedule resulting from the execution of \( TP_i \) from \( DS_3 \) (note that \( S' = T_i' \)). Since \( TP_i \) has fixed-structure, \( \text{structure}(T_i') = \text{structure}(T_i) \). Thus, there exists an operation \( p' \) in \( S' \) such that \( RS(\text{before}(T_i, p), S) = RS(\text{before}(T_i', p'), S') \) and \( WS(\text{after}(T_i, p), S) = WS(\text{after}(T_i', p'), S') \). Since \( DS_3^{\text{read}(before(T_i, p), S)} = \text{read}(before(T_i, p), S) \) and \( \text{structure}(T_i') = \text{structure}(T_i) \), \( \text{read}(before(T_i', p'), S') \). Since writes are a function of the reads before them, \( T_i \) and \( T_i' \) result from the execution of the same transaction program \( TP_i \). \( DS_2^{\text{read}(before(T_i, p), S)} = \text{read}(before(T_i, p), S) \) and \( \text{structure}(T_i') = \text{structure}(T_i) \), \( DS_2^{\text{write}(after(T_i, p), S)} = \text{write}(after(T_i, p), S) \). Since \( TP_i \) is a correct transaction program, \( DS_2 \) is consistent, and the lemma has been proven. \( \square \)

In Lemma 3, if transaction program \( TP_i \) does not have fixed-structure, then \( \text{structure}(T_i') \) may not be equal to \( \text{structure}(T_i) \). As a result, \( WS(\text{after}(T_i, p), S') \) may not be equal to \( WS(\text{after}(T_i, p), S) \) and thus, \( DS_3^{\text{write}(after(T_i, p), S)} \) may not be consistent. This is illustrated by the following example.

Example 3: Consider a database containing data items \( D = \{ a, b, c \} \). Let \( IC = (a > 0 \rightarrow b > 0) \land c > 0 \). Consider the following transaction program, \( TP_1 \).

\[
TP_1 : \quad a := 1; \quad \text{if}(c > 0) \quad b := 1
\]

Let \( d = \{ a, b \} \), and \( DS_1 = \{(a, 1), (b, -1), (c, -1), \} \). Execution of \( TP_1 \) from initial state \( DS_1 \) results in the following transaction:

\[
T_1 : \quad w_1(a, 1) \quad r_1(c, -1)
\]

Let \( p = w_1(a, 1) \). The state resulting from the execution of \( T_1 \) from \( DS_1 \) is \( DS_2 = \{(a, 1), (b, -1), (c, -1), \} \). Thus, though \( DS_2^{\text{write}(before(T_1, p), S)} \) is consistent, \( DS_2^{\text{write}(after(T_1, p), S)} \) is inconsistent \( \{(a, 1), (b, -1), (c, -1), \} \). This is due to the fact that \( TP_1 \) does not have fixed-structure. \( \square \)

Using the property of fixed-structured transaction programs identified in Lemma 3, we can prove that, under the induction hypothesis, the state of the view set of transaction \( T_i \) before operation \( p \), with respect to the data items in \( d_k \), is consistent.

Lemma 4: Let \( IC = C_1 \land C_2 \land \cdots \land C_l \), where \( IC, C_k \) are defined over data items in \( D, d_k \) respectively such that \( d_k \cap d_k = \emptyset, e \neq f \). Let \( S \) be a schedule resulting from the execution of transaction programs with fixed-structure, \( p \) be an operation in \( S \), and \( DS \) be a database state such that \( \text{legal}(DS, S) \). For any \( k, k = 1, 2, \ldots, l \), if \( S^{d_k} \) is serializable with serialization order \( T_1, T_2, \ldots, T_n \), \( S^{d_k} \) is consistent, and for all \( j = 1, 2, \ldots, i - 1 \), \( \text{read}(before(T_j, p), S) \) is consistent, then \( state(T_i, d_k, S, DS)^{\text{read}(before(T_i, p), d_k, S)} \) is consistent, \( i = 1, 2, \ldots, n \).

Proof: The proof is by induction on the number of transactions.

Basis \((i = 1)\): Trivial, as \( state(T_1, d_k, S, DS)^{d_k} = DS^{d_k} \), which is given to be consistent.

Induction: Assume true for \( i = m \) \((1 \leq m < n)\), that is, if for all \( j = 1, 2, \ldots, m - 1 \), \( \text{read}(before(T_j, p), S) \) is consistent, then \( state(T_m, d_k, S, DS)^{\text{read}(before(T_m, p), d_k, S)} \) is consistent. We need to show the above to be true for \( i = m + 1 \). By IH, we know \( state(T_m, d_k, S, DS)^{\text{read}(before(T_m, p), d_k, S)} \) is consistent. By Lemma 2, \( RS(\text{before}(T_{m+1}, p), S) \subseteq VS(T_m, p, d_k, S) \). Since \( d_k \cap d_k = \emptyset, e \neq f \), and \( \text{read}(before(T_{m+1}, p), S) \) is consistent, by Lemma 1, \( state(T_{m+1}, d_k, S, DS)^{\text{read}(before(T_{m+1}, p), d_k, S)} \) is consistent. As transaction program \( TP_{m+1} \) has fixed-structure, by Lemma 3, \( state(T_{m+1}, d_k, S, DS)^{d_k} \) is consistent \((d = VS(T_m, p, d_k, S) \cup WS(\text{before}(T_{m+1}, p), S) - WS(\text{after}(T_{m+1}, p), S)) \). Since \( VS(T_{m+1}, p, d_k, S) = d_k, state(T_{m+1}, d_k, S, DS)^{\text{read}(before(T_{m+1}, p), d_k, S)} \) is consistent. \( \square \)

Since \( RS(\text{before}(T_{m+1}, p), S) \) is a subset of the view set, we have proved that \( \text{read}(before(T_{m+1}, p), S) \) is consistent. Further, since the sets of data items over which conjuncts are defined are disjoint, it follows from Lemma 1 that \( \text{read}(before(T_1, p), S) \) is consistent, as is stated in the following lemma.

Lemma 5: Let \( IC = C_1 \land C_2 \land \cdots \land C_l \), where \( IC, C_k \) are defined over data items in \( D, d_k \) respectively such that \( d_k \cap d_k = \emptyset, e \neq f \). Let \( S \) be a schedule consisting of transactions resulting from the execution of transaction programs with fixed-structure, \( DS \) be a consistent database state such that \( \text{legal}(DS, S) \) and \( p \) be an arbitrary operation in \( S \). If \( S \) is a PWSR schedule, then for all transactions \( T_i \in S \), \( \text{read}(before(T_i, p), S) \) is consistent.

Proof: The proof is by induction on \( \text{depth}(p, S) \).

Basis \( \text{depth}(p, S) = 0 \): There are two cases:

Case 1 \((p \notin T_1)\): \( \text{read}(before(T_1, p), S) = \emptyset \), which is consistent.

Case 2 \((p \in T_1)\): Since \( \text{depth}(p, S) = 0 \), it follows that \( \text{read}(before(T_1, p), S) \subseteq DS \), which is consistent.

Induction: Assume true for \( \text{depth}(p, S) = m \) \((m \geq 0)\), that is, for all transactions \( T_i \in S, \text{read}(before(T_i, p), S) \) is consistent. We need to show that for \( \text{depth}(p, S) = m + 1 \), for all transactions \( T_i \in S, \text{read}(before(T_i, p), S) \) is consistent. Consider two cases.

Case 1 \((p \notin T_1)\): Trivially by IH, \( \text{read}(before(T_1, p), S) \) is consistent.

Case 2 \((p \in T_1)\): Let \( \text{entity}(p) \in d_k \), for some \( k = 1, 2, \ldots, l \). Since \( S \) is PWSR, \( S^{d_k} \) is serializable. Let a serialisation order of transactions in \( S^{d_k} \) be \( T_1, T_2, \ldots, T_{l-1}, T_1, T_{l+1}, \ldots, T_n \). As \( p \in T_1, \)
p \not\in T_j, for all \(j = 1, 2, \ldots, i-1\). Thus, by Case 1 of the induction step above, \(\mathsf{read(before(T_j, p, S))}\) is consistent, for all \(j = 1, 2, \ldots, i-1\). Hence, since transaction programs have fixed-structure, from Lemma 4, \(\mathsf{state(T_i, d_i, S, DS)}\) is consistent. By Lemma 2, \(\mathsf{RS(before(T_i^{d_i}, p, S))} \subseteq \mathsf{VS(T_i, p, d_i, S)}\). Thus, \(\mathsf{read(before(T_i^{d_i}, p, S))}\) is consistent. By IH, \(\mathsf{read(before(T_i^{d_i}, p, S))}\) is consistent. Since \(d_i \cap d_j = \emptyset, e \neq f\), by Lemma 1, \(\mathsf{read(before(T_i, p, S))}\) is consistent. □

We have, therefore, established the following theorem.

**Theorem 1:** Let \(IC = C_1 \wedge C_2 \wedge \ldots \wedge C_l\), where \(IC, C_s\) are defined over data items in \(D, d_s\) respectively such that \(d_s \cap d_j = \emptyset, e \neq f\). Let \(S\) be a schedule consisting of transactions resulting from the execution of transaction programs with fixed-structure. If \(S\) is a PWSR schedule, then \(S\) is strongly correct.

**Proof:** Let \(DS_1\) be a consistent database state such that \(\mathsf{legal(DS_1, S)}\). Let \(\{DS_1\} \subseteq \mathsf{DS_2}\). By Lemma 5, for all \(T_i \in \tau_S\), \(\mathsf{read(T_i)}\) is consistent (Choose \(p\) to be the last operation in the schedule). We now show that \(\mathsf{DS_2}\), for any \(k = 1, 2, \ldots, i\) is consistent. Let \(T_1, T_2, \ldots, T_n\) be a serializability order of transactions in \(S^{d_i}\). Since \(\mathsf{DS_1}\) is consistent, and \(d_s \cap d_i = \emptyset, e \neq f\), by Lemma 4, \(\mathsf{state(T_n, d_s, DS_1, S)}\) is consistent (Choose \(p\) to be the last operation in the schedule). \(\mathsf{DS_2}\) can be shown to be consistent by a simple application of Lemma 3. Thus, by Lemma 1, \(\mathsf{DS_2}\) is consistent, and hence, \(S\) is strongly correct. □

### 3.2 Restricting Schedules

In this section, we examine how, instead of restricting transaction programs, restricting the nature of schedules may enable us to ensure stronger correctness. To develop the intuition, let us examine Example 2 once again. Note that in Example 2, in schedule \(S\), transaction program \(TP_2\) reads data item written by transaction program \(TP_1\) before \(TP_1\) finishes execution (since \(TP_1\) executes the operation \(r_1(c, -1)\) after \(TP_2\) performs its read operation on \(a\)). Had the read operation of \(TP_2\) been delayed until after \(TP_2\) finished execution, the schedule in Example 2 that resulted in a loss of database consistency would not have been permitted. We refer to a schedule \(S\) in which a transaction \(T_i\) cannot read a data item written by transaction \(T_j\) until after \(T_j\) has completed all its operations as a delayed read (DR) schedule. In this section, we will show that if \(S\) is a DR schedule, then the hypothesis of Theorem 1 (that requires transaction programs to have fixed-structure if PWSR schedules are to be strongly correct) can be relaxed. Our interest in DR schedules results from practical considerations. In practice, schedules produced by most DBMSs avoid cascading aborts; that is, most DBMSs produce ACA schedules [BHGS72]. It is trivial to see that every ACA schedule is also DR - hence our interest.

In order to formally define the class of DR schedules, we first define the reads from relation over operations in a schedule. Let \(o_1, o_2 \in \omega\) such that \(\mathsf{action(o_1)} = w, \mathsf{action(o_2)} = r, \mathsf{entity(o_1)} = \mathsf{entity(o_2)}\), and \(o_1 <_S o_2\). Operation \(o_2\) reads from \(o_1\) if for all \(o_k \in \omega\) such that \(o_1 <_S o_k <_S o_1, \mathsf{either} \mathsf{entity(o_k)} \neq \mathsf{entity(o_1)}, \mathsf{or} \mathsf{action(o_k)} \neq w\). We can now formally define the class of DR schedules.

**Definition 5:** A schedule \(S\) is referred as a delayed read (DR) schedule if for all operations \(o_1, o_2 \in \omega, o_1 \in T_1, o_2 \in T_2\), if \(\mathsf{o_2 reads from o_1, then after(T_1, o_1, S) = \epsilon, the empty sequence.}\)

In a DR schedule, a transaction \(T_i\) does not read a data item written by transaction \(T_j\) until \(T_j\) completes execution. Thus, the view set of a transaction in a DR schedule is restricted as is stated in the following lemma.

**Lemma 6:** Let \(S\) be a DR schedule and \(d \subseteq D\) such that \(S^d\) is serializable. Let \(T_1, T_2, \ldots, T_n\) be a serialization order of transactions in \(S^d\) and \(p\) be an operation in \(S\). For all \(i = 1, 2, \ldots, n\), \(\mathsf{RS(before(T_i^d, p, S))} \subseteq \mathsf{VS(T_i, p, d, S)}\), where

\[
\begin{align*}
\mathsf{VS(T_i, p, d, S)} = \\
\quad d, \text{if } i = 1 \\
\quad \mathsf{VS(T_{i-1}, p, d, S)} - \mathsf{WS(T_{i-1}^d)}, \\
\quad \text{if } \mathsf{after(T_{i-1}, p, S) \neq \epsilon} \\
\quad \mathsf{VS(T_{i-1}, p, d, S) \cup WS(T_{i-1}^d)}, \\
\quad \text{if } \mathsf{after(T_{i-1}, p, S) = \epsilon}
\end{align*}
\]

**Proof:** \(\mathsf{RS(before(T_i^d, p, S))} \subseteq d = \mathsf{VS(T_i, p, d, S)}\), Thus, the result holds for \(T_1\). To show that the result holds for any \(T_i, i = 2, 3, \ldots, n\), we will show that if \(d' \in d\), and \(d' \not\in \mathsf{VS(T_i, p, d, S)}\), then \(d' \not\in \mathsf{RS(before(T_i^d, p, S))}\). From the definition of the view set of a transaction, we have the following property about data items which do not belong to a transaction's view set. If \(d' \in d\), and \(d' \not\in \mathsf{VS(T_i, p, d, S)}\), then for some \(j < i\), \(d' \not\in \mathsf{WS(T_j^d)}\), such that \(\mathsf{after(T_j, p, S) \neq \epsilon}\), and for all \(k, k = j + 1, \ldots, i - 1\), \(d' \not\in \mathsf{WS(T_k^d)}\).

Let \(o_1\) be \(T_1\)'s write operation on \(d'\). Suppose that \(d' \in \mathsf{RS(before(T_i^d, p, S))}\) and \(o_2\) is \(T_i\)'s read operation of \(d'\) such that \(o_2 <_S o_2 \not\in p\). Since \(T_i\) is serialized before \(T_1\), it must be the case that \(o_2 <_S o_2\) and further \(o_2\) reads from \(o_1\). However, if \(o_1 <_S o_2\), and \(o_2\) reads from \(o_1\), then since \(S\) is DR, \(\mathsf{after(T_1, o_1, S) = \epsilon}\), which leads to a contradiction. Thus, before \(p, T_i\) cannot read \(d'\); that is, \(d' \not\in \mathsf{RS(before(T_i^d, p, S))}\). □

In the previous section, in our inductive proof of the correctness of PWSR schedules, the only place where we required transaction programs to have fixed-structure was in Lemma 4 in which we showed that for a transaction \(T_i\), a conjunct \(C_k\), and an operation \(p\), if every transaction \(T_j\) serialized before \(T_i\) in \(S^d\) reads consistent data before \(p\), then the view set of \(T_i\), before \(p\), with
respect to \( d_k \), is consistent. We could prove the above in case transaction programs have fixed-structure since it is possible to make claims about the consistency of the data items during the execution of fixed-structured transaction programs (Lemma 3). If, however, transaction programs are not fixed-structured, then the only claim we can make is that a transaction, if it executes from a consistent state, leaves the database in a consistent state.

**Lemma 7:** Let \( S \) be a schedule consisting of a transaction \( T_i \) which results from the execution of a transaction program \( TP_i \) (note that \( S = T_i \)). Let \( DS_1 \) be a database state such that \( \{DS_1\} T_i \{DS_2\} \). If \( DS_1^4 \cup \text{read}(T_i) \) is consistent, then \( DS_2^{\text{d}_{\text{w}}S(T_i)} \) is consistent.

**Proof:** Let \( DS_3 \) be a consistent database state such that \( DS_3^{\text{d}_{\text{w}}S(T_i)} = DS_1^4 \cup \text{read}(T_i) \). Furthermore, let \( \{DS_3\} TP_i \{DS_4\} \). Let \( T_i' \) be the transaction and \( S' \) be the schedule resulting from the execution of \( TP_i \) from \( DS_3 \) (note that \( S' = T_i') \). Since \( DS_3^{\text{d}_{\text{w}}S(T_i')} = \text{read}(T_i') \), \( \text{read}(T_i') = \text{read}(T_i) \). Since writes are a function of the reads before them, \( T_i \) and \( T_i' \) result from the execution of the same transaction program \( TP_i \) we have that \( T_i = T_i' \). Since, \( DS_2^{\text{d}_{\text{w}}S(T_i')} = DS_2^{\text{d}_{\text{w}}S(T_i)} \). Since \( TP_i \) is a correct transaction program, \( DS_2 \) is consistent, and the lemma has been proven. \( \square \)

Note that Lemma 7 requires \( DS_1^4 \cup \text{read}(T_i) \) to be consistent if \( DS_2^{\text{d}_{\text{w}}S(T_i)} \) is to be consistent. Consistency of \( DS_1^4 \) and \( \text{read}(T_i) \) does not ensure consistency of \( DS_2^{\text{d}_{\text{w}}S(T_i)} \) since \( DS_1^4 \cup \text{read}(T_i) \) may not be consistent. As a result, even if \( DS_1^4 \) and \( \text{read}(T_i) \) are consistent, a consistent state \( DS_3 \) such that \( DS_3^{\text{d}_{\text{w}}S(T_i)} = DS_1^4 \cup \text{read}(T_i) \) may not exist.

**Example 4:** Consider a database with data items \( D = \{a, b, c\} \) and the following transaction program \( TP_1 \).

\[
TP_1: \quad a := c
\]

Let IC = \((a = b \land b = c)\). Let \( d = \{a, b\} \) and \( DS_1 = \{(a, -1), (b, -1), (c, 1)\} \). Consider the execution of transaction program \( TP_1 \) from \( DS_1 \) that results in the following transaction:

\[
T_1: \quad r_1(c, 1) \quad w_1(a, 1)
\]

The database state \( DS_2 \) resulting from the execution of \( TP_1 \) from \( DS_1 \) is \( \{(a, 1), (b, -1), (c, 1)\} \). Thus, even though \( DS_1^4 \) and \( \text{read}(T_1) \) are consistent, their union \( \{(a, -1), (b, -1), (c, 1)\} \) is inconsistent and as a result, \( DS_2^{\text{d}_{\text{w}}S(T_i)} = \{(a, 1), (b, -1)\} \) is inconsistent. \( \square \)

Since in the case that schedules are DR, the view set of transactions is restricted, the requirement in Lemma 4 for transaction programs to be fixed-structured can be relaxed. We thus can prove the following lemma.

**Lemma 8:** Let IC = \( C_1 \land C_2 \land \cdots \land C_l \), where IC, \( C_e \) are defined over data items in \( D \), \( d_f \) respectively such that \( d_e \cap d_f = \emptyset, e \neq f \). Let \( S \) be a DR schedule, \( p \) be an operation in \( S \), and \( DS \) be a database state such that \( \text{legal}(DS, S) \). For any \( k, k = 1, 2, \ldots, l \), if \( S^k \) is serialisable with serialisation order \( T_1, T_2, \ldots, T_n, DS^{k} \) is consistent, and for all \( j = 1, 2, \ldots, i - 1, \text{read(before}(T_j, p, S) \) is consistent, then \( \text{state}(T_i, d_k, S, DS)^S(T_{i-1}, p, d_k, S) \) is consistent, \( i = 1, 2, \ldots, n \).

**Proof:** The proof is by induction on the number of transactions.

**Basis (\( i = 1 \)):** Trivial, as \( \text{state}(T_1, d_k, S, DS)^S(T_{1-1}, p, d_k, S) = DS^{k} \), which is given to be consistent.

**Induction:** Assume true for \( i = m (1 \leq m < n) \), that is, if for all \( j = 1, 2, \ldots, m - 1, \text{read(before}(T_j, p, S) \) is consistent, then \( \text{state}(T_m, d_k, S, DS)^S(T_{m-1}, p, d_k, S) \) is consistent. We need to show the above to be true for \( i = m + 1 \). There are two cases to consider:

**Case 1** after \( (T_m, p, S) \) \( \neq \epsilon \) By III, we know that \( \text{state}(T_m, d_k, S, DS)^S(T_{m-1}, p, d_k, S) \) is consistent. We need to show that

\[
\text{state}(T_{m+1}, d_k, S, DS)^S(T_{m}, p, d_k, S) - WS(T_{m}^*)
\]

is consistent, since \( VS(T_{m+1}, p, d_k, S) = VS(T_m, p, d_k, S) - WS(T_{m}^*) \).

From the definition of state, we have

\[
\text{state}(T_{m+1}, d_k, S, DS)^S(T_{m}, p, d_k, S) - WS(T_{m}^*) \geq \text{state}(T_m, d_k, S, DS)^S(T_{m}, p, d_k, S) - WS(T_{m}^*)
\]

Since \( \text{state}(T_m, d_k, S, DS)^S(T_{m}, p, d_k, S) \) is consistent,

\[
\text{state}(T_{m+1}, d_k, S, DS)^S(T_{m}, p, d_k, S) - WS(T_{m}^*) \text{ is consistent. Thus, } \text{state}(T_{m+1}, d_k, S, DS)^S(T_{m}, p, d_k, S)
\]

is consistent.

**Case 2** after \( (T_m, p, S) = \epsilon \) Since after \( (T_m, p, S) = \epsilon \), \( RS(\text{before}(T_m, p, S)) = RS(T_m) \). By III, we know that \( \text{state}(T_m, d_k, S, DS)^S(T_{m-1}, p, d_k, S) \) is consistent. By Lemma 6, \( \text{state}(T_m, d_k, S, DS)^S(T_{m-1}, p, d_k, S) \) is consistent, by \( \text{state}(T_m, d_k, S, DS)^S(T_{m-1}, p, d_k, S) \cup \text{read}(T_m) \) is consistent. Thus, by Lemma 7, \( \text{state}(T_{m+1}, d_k, S, DS)^S(T_{m}, p, d_k, S) - WS(T_{m}^*) \) is consistent. As \( VS(T_{m+1}, p, d_k, S) = VS(T_m, p, d_k, S) \cup WS(T_{m}^*) \), \( \text{state}(T_{m+1}, d_k, S, DS)^S(T_{m}, p, d_k, S) \) is consistent. Hence proved. \( \square \)

Using the above lemma, it is easy to see that the proof developed in the previous section can be used to show that for every operation \( p \) and transaction \( T_i \) in a schedule \( S \), \( \text{read(before}(T_i, p, S) \) is consistent.
Lemma 9: Let $IC = C_1 \wedge C_2 \wedge \cdots \wedge C_l$, where $IC$, $C_e$ are defined over data items in $D$, $d_e$ respectively such that $d_e \cap d_j = \emptyset$, $e \neq f$. Let $S$ be a schedule, $DS$ be a consistent database state such that legal($DS$, $S$) and $p$ be an arbitrary operation in $S$. If $S$ is a PWSR schedule as well as DR, then for all transactions $T_i \in S$, read$(before(T_i, p), S)$ is consistent.

Proof: The proof is by induction on depth$(p, S)$.

Basis (depth$(p, S) = 0$): There are two cases:

Case 1 ($p \notin T_i$): read$(before(T_i, p), S) = \emptyset$, which is consistent.

Case 2 ($p \in T_i$): Since depth$(p, S) = 0$, it follows that read$(before(T_i, p), S) \subseteq DS$, which is consistent.

Induction: Assume true for depth$(p, S) = m$ ($m \geq 0$), that is, for all transactions $T_i \in S$, read$(before(T_i, p), S)$ is consistent. We need to show for depth$(p, S) = m + 1$, for all transactions $T_i \in S$, read$(before(T_i, p), S)$ is consistent. Consider two cases.

Case 1 ($p \notin T_i$): Trivially by IH, read$(before(T_i, p), S)$ is consistent.

Case 2 ($p \in T_i$): Let entity$(p) \in d_k$, for some $k = 1, 2, \ldots, l$. Since $S$ is PWSR, $S^{d_k}$ is serializable. Let a serialization order of transactions in $S^{d_k}$ be $T_1, T_2, \ldots, T_{i-1}, T_i, T_{i+1}, \ldots, T_n$. As $p \in T_i$, $p \notin T_j$, for all $j = 1, 2, \ldots, i-1$. Thus, by Case 1 of the induction step above, read$(before(T_j, p), S)$ is consistent, for all $j = 1, 2, \ldots, i-1$. Hence, since $S$ is DR, by Lemma 8, state$(T_i, d_k, S, DS)^{VS(1,p,d_k,S)}$ is consistent. By Lemma 6, $RS(before(T_i, p), S) \subseteq VS(T_i, p, d_k, S)$. Thus, read$(before(T_i, p), S)$ is consistent. By IH, read$(before(T_i, p), S)$ is consistent. Since $d_i \cap d_j = \emptyset$, $e \neq f$, by Lemma 1, read$(before(T_i, p), S)$ is consistent. □

Thus, we can establish the following theorem.

Theorem 2: Let $IC = C_1 \wedge C_2 \wedge \cdots \wedge C_l$, where $IC$, $C_e$ are defined over data items in $D$, $d_e$ respectively such that $d_e \cap d_j = \emptyset$, $e \neq f$. Let $S$ be a schedule that results from the execution of transactions programs. If $S$ is both PWSR and DR, then $S$ is strongly correct.

Proof: Let $DS_1$ be a consistent database state such that legal($DS_1$, $S$). Let $\{DS_1\}S$($DS_2$). By Lemma 9, for all $T_i \in S$, read$(T_i)$ is consistent (choose $p$ to be the last operation in the schedule). We now show that $DS_2^{d_k}$, for any $k = 1, 2, \ldots, l$, is consistent. Let $T_1, T_2, \ldots, T_n$ be a serial number of operations in $S^{d_k}$. Since $DS_2^{d_k}$ is consistent, and $d_k \cap d_j = \emptyset$, $e \neq f$, by Lemma 8, state$(T_i, d_k, DS_1, S)$ is consistent (choose $p$ to be the last operation in the schedule). $DS_2^{d_k}$ can be shown to be consistent by a simple application of Lemma 7. Thus, by Lemma 1, $DS_2$ is consistent, and hence, $S$ is strongly correct. □

3.3 Restricting Data Accesses

In the previous sections, we showed that PWSR schedules are strongly correct if either transaction programs are fixed structured, or schedules are DR. If, however, we can order the conjuncts in a manner that ensures that no transaction reads a data item belonging to a higher numbered conjunct and writes a data item belonging to a lower numbered conjunct, then these requirements are unnecessary. In Example 2, transaction $T_1$ reads data item $c$ from conjunct $C_2$ and writes data item $a$ in conjunct $C_1$, while $T_2$ reads data item $a$ from conjunct $C_1$ and writes data item $c$ in conjunct $C_2$. Thus, transactions $T_1$ and $T_2$ access data items in the conjuncts in a cyclic fashion that causes database consistency to be violated.

We can formalize the requirement for the ordering between conjuncts by defining a directed graph which we refer to as the data access graph. Let $IC = C_1 \wedge C_2 \wedge \cdots \wedge C_l$, where $IC$, $C_e$ are defined over data items in $D$, $d_e$ respectively, and $S$ be a schedule. The data access graph for $S$ and $IC$, denoted by DAG$(S, IC)$, consists of a set of nodes, one for every conjunct $C_i$ in $IC$. Furthermore, for conjuncts $C_i, C_f$ in $IC$, if $C_i \neq C_f$, there is a directed edge $(C_i, C_f)$ in DAG$(S, IC)$ if there exists a transaction in $S$ that reads a data item in $d_i$ and writes a data item in $d_j$. In the following, we show that a PWSR schedule $S$ is strongly correct if DAG$(S, IC)$ is acyclic.

Earlier, in Lemma 7, we specified conditions under which the execution of a transaction preserves the consistency of a set of data items. We now use Lemma 7 to develop conditions under which schedules preserve the consistency of a set of data items.

Lemma 10: Let $S$ be a schedule, $d \subseteq D$, and $DS_1$ be a database state such that $\{DS_1\}S$($DS_2$). If

- $S^{d}$ is serializable (let $T_1, T_2, \ldots, T_n$ be a serialization order of transactions in $S^{d}$),
- for all $T_i \in \tau_w(d, S)$, if state$(T_i, d, S, DS_1)$ is consistent, then state$(T_i, d, S, DS_1) \cup$ read$(T_i)$ is consistent, and
- $DS_2^{d_i}$ is consistent,

then $DS_2^{d}$ is consistent and state$(T_i, d, S, DS_1)$ is consistent for all $i, i = 1, 2, \ldots, n$.

Proof: The proof is by induction on $i$.

Basis ($i = 1$): state$(T_1, d, S, DS_1) = DS_1^{d_i}$, which is given to be consistent.

Induction: Assume true for $i = m$ ($1 \leq m < n$), that is, state$(T_m, d, S, DS_1)$ is consistent. We need to show the above for $i = m + 1$; that is, state$(T_{m+1}, d, S, DS_1)$ is consistent. Consider the following two cases:

Case 1 ($T_m \notin \tau_w(d, S)$): Since $T_m \notin \tau_w(d, S)$, it follows that state$(T_{m+1}, d, S, DS_1) = state(T_m, d, S, DS_1)$. 106
By IH, since \( \text{state}(T_m, d, S, DS_1) \) is consistent, 
\( \text{state}(T_{m+1}, d, S, DS_1) \) is consistent.

Case 2 (\( T_m \in \tau_w(d, S) \)): Since \( \text{state}(T_m, d, S, DS_1) \) is consistent, 
\( \text{state}(T_m, d, S, DS_1) \cup \text{read}(T_m) \) is consistent. By Lemma 7, \( \text{state}(T_{m+1}, d, S, DS_1) \) is consistent. Thus, by Lemma 7, (using a similar argument as above) \( DS_2^d \) is consistent. □

Theorem 3: Let IC = \( C_1 \wedge C_2 \wedge \cdots \wedge C_r \), where IC, \( C_e \) be defined over data items in D, \( d_e \) respectively such that \( d_e \cap d_f = \emptyset, e \neq f \) and S is a PWSR schedule. If \( \text{DAG}(S, IC) \) is acyclic, then S is strongly correct.

Proof: Without loss of generality, let \( C_1, C_2, \ldots, C_l \) be a topological ordering of nodes in \( \text{DAG}(S, IC) \). Thus, every transaction that updates a data item in \( d_k \) only reads data items belonging to conjuncts \( d_1, d_2, \ldots, d_k \). Let \( DS_1 \) be a consistent database state such that \( \text{legal}(DS_1, S) \). Let \( \{DS_1, S\} \cup \{DS_2, S\} \). We prove by induction on k that for all \( k = 1, 2, \ldots, l \), \( DS_2^d_k \) is consistent, and for every transaction \( T_i \in \tau_w, \text{read}(T_i^d) \) is consistent.

Basis (\( k = 1 \)): Let \( T_1, T_2, \ldots, T_n \) be a serialisation order of transactions in \( S^d \). Since transactions that update data items in \( d_1 \) can only read data items in \( d_1 \), \( RS(T_1) \subseteq d_1 \), for all \( T_i \in \tau_w(d_1, S) \). Thus, for every transaction \( T_i \in \tau_w(d_1, S) \), \( \text{read}(T_i) \subseteq \text{state}(T_i, d_1, S, DS_1) \) if \( \text{state}(T_i, d_1, S, DS_1) \) is consistent, then \( \text{state}(T_i, d_1, S, DS_1) \cup \text{read}(T_i) \) is consistent. Thus, by Lemma 10, \( DS_1^d \) is consistent and for all \( i, i = 1, 2, \ldots, n \), \( \text{state}(T_i, d_1, S, DS_1) \) is consistent. Thus, for all \( i = 1, 2, \ldots, n \), \( \text{read}(T_i^d) \subseteq \text{state}(T_i, d_1, S, DS_1) \), \( \text{read}(T_i^d) \) is consistent.

Induction: Assume true for \( k \leq m \) (\( 1 \leq m < k \)). We show that the above is true for \( k \leq m + 1 \). Let \( T_1, T_2, \ldots, T_n \) be a serialisation order of transactions in \( S^{m+1} \). Consider any transaction \( T_i \in \tau_w(d_{m+1}, S) \). Since \( \text{DAG}(S, IC) \) is acyclic, \( T_i \) can only read data items in \( d_1, \ldots, d_m, d_{m+1} \). By IH, \( \text{read}(T_i^d) \) is consistent for all \( e = 1, 2, \ldots, m \). Thus, since \( d_e \cap d_f = \emptyset \), \( e \neq f \), and \( \text{read}(T_i^{m+1}) \subseteq \text{state}(T_i, d_{m+1}, S, DS_1) \), by Lemma 1, if \( \text{state}(T_i, d_{m+1}, S, DS_1) \) is consistent, \( \text{state}(T_i, d_{m+1}, S, DS_1) \cup \text{read}(T_i) \) is consistent. Thus, by Lemma 10, \( \text{state}(T_i, d_{m+1}, S, DS_1) \) is consistent for all \( i, i = 1, 2, \ldots, n \) (and thus, \( \text{read}(T_i^{m+1}) \) is consistent) and \( DS_2^{m+1} \) is consistent.

Thus, we have proved that for all \( k = 1, 2, \ldots, l \), \( DS_2^d_k \) and \( \text{read}(T_i^d), T_i \in \tau_w \), is consistent. Since \( d_e \cap d_f = \emptyset \), \( e \neq f \), by Lemma 1, \( DS_2^d \) is consistent and \( \text{read}(T_i^d) \) is consistent for all \( T_i \in \tau_w \). □

Example 5: Consider a database containing data items \( D = \{a, b, c, d\} \) and the following transaction programs \( TP_1, TP_2 \) and \( TP_3 \).

\[
TP_1: \quad b := c - 5 \\
TP_2: \quad \text{temp} := c; \quad a := \text{temp} + 20; \\
TP_3: \quad d := a - b \quad c := \text{temp} + 20
\]

Let IC = \((a > b) \land (a = c) \land (d > 0)\). The conjuncts are not disjoint and share data item a. Transaction programs \( TP_1, TP_2 \) and \( TP_3 \) have fixed-structure. Consider the following schedule resulting from the execution of \( TP_1, TP_2 \) and \( TP_3 \) from database state \( \{(a, 10), (b, 0), (c, 10), (d, 5)\} \).

\[
S: \quad r_3(a, 10) \quad r_2(c, 10) \quad w_3(a, 30) \quad w_2(c, 30) \quad r_1(c, 30) \quad w_1(b, 25) \quad r_3(b, 25) \quad w_3(d, -15)
\]

Note that \( S \) is DR and \( \text{DAG}(S, IC) \) is acyclic. The database state resulting from the execution of the above schedule is \( \{(a, 30), (b, 25), (c, 30)(d, -15)\} \), which is inconsistent. Thus, if conjuncts are defined over sets of data items which are not disjoint, PWSR schedules may not preserve integrity constraints. □

4 Conclusion

The traditionally accepted correctness criterion, serializability, has unacceptable practical consequences for certain "new" database applications. Among the adverse results are long-duration waits imposed by locking protocols and potential loss of autonomy by sites in distributed systems. A way of enhancing concurrency and improving performance in such environments is to relax the serializability requirement.

While substantial research has been done towards developing approaches for relaxing the serializability requirement, very little work exists on identifying what consistency guarantees result from the above approaches. Most approaches have been ad-hoc in nature and have been defined in only an operational manner, an example of which is cursor stability (degree 2 consistency). In this paper, we studied the formal implications of choosing predicatewise serializability (PWSR) as the correctness criterion, and identified various restrictions on transaction programs and integrity constraints under which PWSR schedules preserve database consistency. We developed novel proof techniques in order to show that PWSR schedules preserve database consistency. The proof techniques

- exploit the structural properties (not semantics) of integrity constraints,
- exploit the the semantics of read and write operations,
• perform induction on read and write operations in the schedule (as opposed to transactions).

All the restrictions identified by us require the sets of data items over which integrity constraints are defined to be disjoint. The various restrictions highlight the trade-off between the generality of the transactions in the system and the degree of concurrency permitted.

The importance of the PWSR correctness criteria has been demonstrated in [KKB88, BGMS92]. In [KKB88], it was shown that the PWSR correctness criterion can be used to alleviate the long-duration waits associated with long-duration transactions in computer-aided design and manufacturing environments. More recently, in [BGMS92] it was shown that the PWSR correctness criterion (referred to as the local serializability (LSR) criterion) can be used to help alleviate the autonomy-induced problems in multidatabase system (MDBS) applications.

An MDBS is a distributed database system in which each of the individual local databases (DBMSs) may belong to autonomous organizations, may be pre-existing, and may employ different transaction management schemes. In such a system, guaranteeing global serializability may result not only in poor performance, but depending upon the interface exported by the pre-existing system, may not even be possible [BGMS92]. Since these pre-existing systems belong to autonomous organizations, in [BGMS92], it is argued that for certain MDBSs, the consistency constraints present are local ones that involve only data located at a single DBMS. We can thus view the integrity constraints of the system as a conjunction of predicates, where each conjunct is defined over the set of data items in a local DBMS. Since each local DBMS ensures serializability of its local schedule, the resulting global schedule is PWSR, where the data items in each conjunct is disjoint. Thus, the results of this paper are directly applicable to such MDBS environments.

References


108