Maintaining HDDBS consistence: The Quasi Serializability approach

Ahmed K. Elmagarmid

Purdue University, ake@cs.purdue.edu

Weimin Du

Report Number:
90-1017
MAINTAINING HDDBS CONSISTENCY:
THE QUASI SERIALIZABILITY APPROACH
Ahmed K. Elmagarmid
Weimin Du

CSD-TR-1017
September 1990
Maintaining HDDBS Consistency:  
The Quasi Serializability Approach*  

Ahmed K. Elmagarmid and Weimin Du  
Indiana Center for Database Systems  
Department of Computer Sciences  
Purdue University  
West Lafayette, IN 47907  
{ake,du}@cs.purdue.edu  

Abstract  

In this paper, we introduce quasi serializability, a correctness criterion for concurrency control in heterogeneous distributed database systems (HDDBSs). Quasi serializability is a weaker criterion than serializability in that it only controls execution of global transactions. Quasi serializability is suited to HDDBS applications where local concurrency controllers (LCCs) maintain serializability of local executions. Quasi serializable executions maintain HDDBS consistency if local transactions at different sites do not affect each other. We propose a concurrency control mechanism that generates quasi serializable executions only. The mechanism imposes no restriction on and requires no information about LCCs. It also provides a higher degree of concurrency than that based on serializability.  

Keywords: Distributed database systems, heterogeneous databases, concurrency control, transaction management  

*Sponsored by the Indiana Corporation for Science and Technology (CST), a PYI Award from the NSF, and grants from the AT&T Foundation, Tektronix, SERC, Mobil Oil, and a David Ross Fellowship from the Purdue Research Foundation.
1 Introduction

A heterogeneous distributed database system (HDDBS) is a federation of pre-existing database systems (called local database systems, or LDBSs) supporting global applications accessing more than one LDBS. An important feature of HDDBSs is the autonomy of LDBSs. Generally, local autonomy reflects the fact that LDBSs were independently developed and administered by different organizations. They are later integrated in a bottom-up fashion. Existing applications (called local transactions) are expected to continue to execute after integration. New applications accessing more than one LDBS (called global transactions) are decomposed into subtransactions which are then executed at local sites along with local transactions.

An active research area in HDDBSs is concurrency control. A concurrency control mechanism coordinates concurrent execution of global and local transactions to ensure the consistency of an HDDBS. Due to the hierarchical structure of HDDBSs, two types of consistency coexist: local and global consistencies. Local consistency defines constraints on relationships of data in a local database as well as on interactions among local transactions and global subtransactions executed at the site. Global consistency defines constraints on relationships of data at different local databases and on interactions among global transactions, as well as interactions among local transactions executed at different sites. Initially, only local concurrency controllers (LCCs) at each site exist. These LCCs must continue to operate independently after the HDDBS is built in order to preserve the autonomy and maintain the consistency of the LDBS. LCCs, however, are not capable of ensuring the global consistency of HDDBSs because global transactions may be scheduled inconsistently at different sites. A global concurrency controller (GCC) is therefore needed. The GCC is built on top of LCCs and maintains the global consistency by coordinating local executions.

The conventional way of maintaining database consistency is to execute transactions in a serializable fashion. An execution of a set of transactions is serializable if it is equivalent to a serial execution of the transactions. Serializability in HDDBSs represents the strongest type of consistency in that it makes no distinction between local and global transactions. Due to local autonomy of LDBSs [5] [4], it is difficult to maintain serializability in HDDBSs. The difficulties are discussed later in the paper and the reader is urged to consult the references for detailed examples.

The primary goal of this paper is to present a new approach to maintain HDDBS consistency by using a weaker correctness criterion called quasi serializability. A global execution of a set of local and global transactions is quasi serializable if local executions are all serializable and it
is equivalent to a quasi serial execution in which global transactions are executed sequentially. Quasi serializability focuses on the behavior of and interactions among global transactions which the GCC is able to control and therefore is maintainable at the global level without violating local autonomy. It is the LCCs’ responsibility to maintain serializability at each of the local sites. The main results of the paper are: 1) studying the quasi serializability theory; and 2) proposing a concurrency control mechanism based on quasi serializability.

The basic quasi serializability theory was introduced in [2]. The scheduler for quasi serializable executions in Section 5.1 was first presented in [3]. This paper extends the results of the previous two papers. It focuses on transaction aspect of HDDBS consistency. The ability of quasi serializable executions to preserve data integrity of HDDBSs is discussed in [6].

The problem of extending serializability in various environments has been investigated by other researchers, see [9] [10] [8] for example. In general, they require users to provide semantic information about transactions. In other words, users have the responsibility to specify, in terms of either transaction semantic types [9] or transaction atomic steps [10] [8], what kinds of non-serializable executions are allowed. Our work is different from theirs in that no additional responsibility is imposed on the users in order to allow non-serializable executions. This is important, especially in HDDBSs, because a user may not be aware of all other applications.

The rest of the paper is organized as follows. We first present, in Section 2, HDDBS applications illustrating the basic idea of maintaining the HDDBS consistency using quasi serializable executions. A formal model of HDDBS is introduced, along with other notations, in Section 3. We introduce quasi serializability and study its basic properties in Section 4. In Section 5, we discuss concurrency control based on quasi serializability by proposing a scheduler for quasi serializable executions and presenting a restriction on information flow in a global execution to prevent undesirable interactions. Section 6 contains a discussion and some concluding remarks.

2 Examples

As we have mentioned, the basic idea of quasi serializability is 1) to relax consistency requirements by taking advantage of consistency of local databases and independence of local transactions; and 2) to simplify concurrency control problem by controlling information flow in an HDDBS. In this section, we illustrate the idea by describing two simple applications. The objective of the first application is to show that a non-serializable execution may be correct under certain conditions. The observation has led to the study of quasi serializability. The second application shows possible anomalies in quasi serializable executions and how they can be prevented by
Flight reservation. Consider a simple flight reservation system. The system consists of databases of airline companies in different countries. Each local database system allows users to make reservations and cancellations in flights of the company. The global system allows users to make reservations and cancellations in flights of more than one company at a time. For example, a user who wants to go from Lafayette, Indiana to Beijing, China would like to reserve a seat from Lafayette to San Francisco on an American Airline flight and a seat from San Francisco to Beijing on a CAAC flight. The system also allows users to generate reports of reservations of one or more flights in one or more companies. In summary, the system supports transactions consisting of the following operations.

- **Local reservation** on flights of a single company
- **Local Cancellation** on flights of a single company
- **Local report** of reservations on flights of a single company
- **Global reservation** on flights of more than one company
- **Global cancellation** on flights of more than one company
- **Global report** of reservations on flights of more than one company

We assume that checking the number of seats reserved on a flight is an atomic operation, while reserving a seat on a flight consists of the following two atomic operations: checking the number of seats reserved and increasing them by one (if there are seats available).

Example 2.1 Consider a local transaction $L_1$ which makes reservations on flights $a_1$ and $a_2$ of airline company $A$, and a local transaction $L_2$ which makes a reservation on flight $b_1$ of another airline company $B$. Since $L_1$ and $L_2$ access different databases, they are independent. Consider also a global transaction $G_1$ which makes reservations on flights $a_1$ (of airline company $A$), $b_1$, $b_2$ (of airline company $B$) and $c_1$ (of airline company $C$), and a global transaction $G_2$ which makes a reservation on the flight $a_2$ and reports the reservations of flights $a_2$, $b_1$, $b_2$ and $c_1$. Suppose that the initial reservations of the flights are all 100, and the transactions are executed as follows.

Airline $A$:

- $L_1$ checks the number of seats reserved on $a_1$
- $L_1$ increases the number of seats reserved on $a_1$ by 1
- $G_1$ checks the number of seats reserved on $a_1$
- $G_1$ increases the number of seats reserved on $a_1$ by 1
- $G_2$ checks the number of seats reserved on $a_2$
$G_2$ increases the number of seats reserved on $a_2$ by 1
$G_2$ checks the number of seats reserved on $a_2$
$L_1$ checks the number of seats reserved on $a_2$
$L_1$ increases the number of seats reserved on $a_2$ by 1

Airline $B$:
$G_1$ checks the number of seats reserved on $b_1$
$G_1$ increases the number of seats reserved on $b_1$ by 1
$G_1$ checks the number of seats reserved on $b_2$
$G_1$ increases the number of seats reserved on $b_2$ by 1
$L_2$ checks the number of seats reserved on $b_1$
$L_2$ increases the number of seats reserved on $b_1$ by 1
$G_2$ checks the number of seats reserved on $b_1$
$G_2$ checks the number of seats reserved on $b_2$

Airline $C$:
$G_1$ checks the number of seats reserved on $c_1$
$G_1$ increases the number of seats reserved on $c_1$ by 1
$G_2$ checks the number of seats reserved on $c_1$

After the execution, the number of seats reserved on the flights are $a_1 = 102, a_2 = 102, b_1 = 102, b_2 = 101$ and $c_1 = 101$, respectively. So, seats are reserved correctly. In addition, the reservations of flights $a_2, b_1, b_2$ and $c_1$ read by $G_2$ are also correct and reflect the execution of transactions $G_1$ and $L_2$. Therefore, the execution is correct. However, it is not serializable.

The execution in Example 2.1 has the following properties. First, each local execution is serializable. Second, the global transactions are executed sequentially and in the same order at all local sites. Finally, local transactions at different sites do not interact with each other. To see this, notice that $L_1$ and $L_2$ do not access common data. Although global transactions may introduce indirect conflicts between them (for example, $L_1$ indirectly conflicts with $L_2$ because it conflicts with $G_1$ at Airline $A$ which in turn conflicts with $L_2$ at the site 2), such an indirect conflict (e.g., that between $L_1$ and $L_2$) does not imply interactions between their executions. As a result, each transaction appears as an indivisible step to every other transaction in the execution. A Global transaction is indivisible to other global transactions because they are executed sequentially. It is also indivisible to local transactions accessing the same site because of the serializability of the local execution. Similarly, a local transaction appears as an indivisible step to other local and global transactions accessing the same site because the local execution is serializable. It is indivisible to local transactions at other sites because they do not interact with each other. The
indivisibility of transactions implies correctness of the execution and consistency of the HDDBS.

On the other hand, the first two properties of the execution (i.e., serializability of local executions and sequential execution of global transactions) are also necessary to ensure indivisibility of the transactions. To see this, note that if the global transactions are not executed in the same order at all site, e.g., $G_2$ precedes $G_1$ at Airline C, the reservations read by $G_2$ would be incorrect because it reflects the result of execution of $G_1$ at Airline B, but not that at Airline C. In other words, $G_1$ does not appear as an indivisible step to $G_2$. Similarly, if local executions are not serializable, a local transaction may not be indivisible to local transactions and global subtransactions at the site. We call executions with the two properties quasi serializable executions (see Section 4). As we have seen from Example 2.1, quasi serializability ensures indivisibility of transactions if local transactions at different sites do not affect each other.

In some applications, however, local transactions at different sites do interact with each other indirectly via global transactions. The interactions could be prevented by controlling information flow in a global transaction, as illustrated in the following application.

**International banking.** Consider the HDDBS of an international banking federation consisting of local databases of member banks at each country. Each local database consists of individual accounts. A customer may have accounts at one or more banks. He can manipulate his accounts at a bank in the same way as he did before (local transactions). In addition, he can deposit money to and check the balance of his accounts at more than one bank (global transactions). He can also transfer money from an account at one bank to accounts at other banks. The following are operations supported by the system.

- **Local deposit:** deposit money to accounts at a single bank.
- **Local withdrawal:** withdraw money from accounts at a single bank.
- **Local transfer:** transfer money from an account to another account at the same bank.
- **Global deposit:** deposit money to accounts at more than one bank.
- **Global withdrawal:** withdraw money from accounts at more than one bank.
- **Global transfer:** transfer money from an account at one bank to accounts at other banks.

**Example 2.2** Consider an HDDBS consisting of LDBSs of two banks A and B. Let $L_1$ and $L_2$ be local transactions and $G_1$, $G_2$ and $G_3$ be global transactions, where $L_1$ checks the balance of accounts $x_1$ and $x_2$ at bank A and deposits $1,000 to $x_1$, $L_2$ transfers all the money from account $y_1$ to account $y_2$ at bank B, $G_1$ transfers all the money in $x_1$ to $y_1$, $G_2$ deposits $500 to $x_2$ and $y_2$, and $G_3$ transfers all the money in $y_2$ to $x_2$. Suppose that the initial values of the accounts are $x_1 = y_2 = 1,000 and y_1 = 0$. The transactions are executed as follows.
Bank A:

$L_1$ reads the balance of $x_1$
$L_1$ deposits $1,000 to $x_1$
$G_1$ withdraws the balance of $x_1 (= 2,000)$ and put it in $a$
$G_2$ deposits $500 to $x_2$
$G_3$ deposits $b (= 3,500)$ to $x_2$
$L_1$ reads the balance of $x_2$

Bank B:

$G_1$ deposits $a (= 2,000)$ to $y_1$
$L_2$ transfers the balance of $y_1 (= 2,000)$ to $y_2$
$G_2$ deposits $500$ to $y_2$
$G_3$ withdraws the balance of $y_2 (= 3,500)$ and put it in $b$

The execution is quasi serializable (i.e., each local execution is serializable and global transactions are executed sequentially). However, the balance of $x_1$ and $x_2$ read by $L_1$ is $6,000$ which is clearly not correct.

The problem with the execution is that both $L_1$ and $L_2$ affect each other. $L_2$ is affected by $L_1$ because the balance it reads includes the money deposited by $L_1$, while $L_1$ is affected by $L_2$ because the balance of $x_2$ it reads includes the money transferred by $L_2$. The interactions are due to information flows in global transactions (called remote value dependency, see Section 3). For example, $L_1$ affects $L_2$ because of the information flow ($2,000 from x_1 to y_1$) in $G_1$. Similarly, $L_2$ affects $L_1$ because of the information flow ($3,500 from y_2 to x_2$) in $G_2$. As a result, the balance of $x_1$ is read by $L_1$ twice (one from $x_1$ and another from $x_2$ which is transferred by $L_2$). To ensure that $L_1$ and $L_2$ interact with each other in a partial order, one of the information flows must be delayed.

Bank A:

$L_1$ reads the balance of $x_1$
$L_1$ deposits $1,000 to $x_1$
$G_1$ withdraws the balance of $x_1 (= 2,000)$ and put it in $a$
$G_2$ deposits $500 to $x_2$
$L_1$ reads the balance of $x_2$
$G_3$ deposits $b (= 3,500)$ to $x_2$

Now, only $G_1$ introduces an interaction between $L_1$ and $L_2$. Therefore, $L_1$ and $L_2$ appear indivisible to each other. It is not hard to see that the new execution is correct. It is worth noting that the execution is still not serializable.
In the following sections, we will formalize the idea of maintaining HDDBS consistency using quasi serializable executions. First, let us review and introduce some of the basic concepts that will be useful throughout the paper.

3 Preliminaries

3.1 An HDDBS Model

An HDDBS consists of a set $D^1$ of data items and a set $T$ of transactions. The data item set $D$ consists of $n$ subsets, $D_1, D_2, ..., D_n$, called local databases. In this paper, we assume that local databases are disjoint. In other words, there is no replication at the global level. The transaction set $T$ consists of $n + 1$ subsets, $G, L_1, L_2, ..., L_n$, where $L_i$ is a set of local transactions that access $D_i$ only, while $G$ is a set of global transactions that access more than one local database. A global transaction $G_i$ consists of a set of subtransactions $\{G_{i,1}, G_{i,2}, ..., G_{i,n}\}$, where the subtransaction $G_{i,j}$ accesses $D_j$ only. The data item set $D_i$, together with the transaction set $T_i = L_i \cup G_i$ where $G_i = \{G_{j,i} \mid G_j \in G\}$, forms the local database system $LDBS_i$.

3.2 Transactions and Value Dependency

A transaction $T_i$ is a finite set of operations. Each operation is either a read operation reading a data item $x$, denoted $r_i(x)$, or a write operation writing a data item $x$, denoted $w_i(x)$. We use $R(T_i)$ and $W(T_i)$ to denote the sets of read and write operations of $T_i$, respectively, and $O(T_i) = R(T_i) \cup W(T_i)$ the set of all operations in $T_i$.

Operations in a simple transaction (i.e., a local transaction or a global subtransaction) are linearly ordered (execution order). We assume that if a transaction both reads and writes a data item, the read operation precedes the write operation in the execution order. Execution orders between operations of different subtransactions of the same global transaction, however, are not specified$^2$.

Operations in a transaction (local or global) are also partially ordered according to their value dependency. Value dependency is a relation between a write operation and a read operation of the same transaction. More specifically, a write operation depends on a read operation if the value it writes is a function of the value read by the read operation. We assume that there is value

---

$^1$In the paper, we use italic letters to denote instances, in particular, lower case for data items and upper case for transactions, calligraphic letters to denote sets, and Roman letters to denote acronyms.

$^2$Specification and coordination of the execution order of operations of different subtransactions are discussed in [7].
dependency between each write operation and a read operation of the same simple transaction which precedes it in the execution order. A write operation in a subtransaction may also depend on a read operation in another subtransaction of the same global transaction. This kind of remote value dependency must be explicitly specified in order to execute transactions correctly.

Definition 3.1 (Simple transactions) A simple transaction $T$ is a pair $\langle \mathcal{O}(T), \prec_t^T \rangle$, where $\mathcal{O}(T)$ is the set of operations of $T$ and $\prec_t^T$ is a linear order (execution order) in which operations in $\mathcal{O}(T)$ are executed.

Given a simple transaction $T$, its value dependency is formally defined as

$$
\prec_{vd}^T = \{(o_i, o_j) \mid o_i \in \mathcal{R}(T), o_j \in \mathcal{W}(T) \text{ and } o_i \prec_t^T o_j\}
$$

Definition 3.2 (Global transactions) A global transaction $G_0$ is a pair $\langle TS(G_0), \prec_{vd}^{G_0} \rangle$, where $TS(G_0)$ is a set of simple transactions (called global subtransactions) and $\prec_{vd}^{G_0}$ is a binary relation over $\mathcal{O}(G_0) = \cup_{T \in TS(G_0)} \mathcal{O}(T)$ representing the remote value dependency of $G_0$,

$$
\prec_{vd}^{G_0} = \{(o_i, o_j) \mid \exists x, y \in \mathcal{D} \text{ and } G_{0,i}, G_{0,j} \in TS(G_0) \text{ such that } o_i = r_i(x) \in \mathcal{R}(G_{0,i}), o_j = w_j(y) \in \mathcal{W}(G_{0,j}) \text{ and } y = f(x) \text{ for some function } f\}.
$$

Remote value dependency is included in the definition of global transactions because it is both necessary for execution of transactions and useful in maintaining transaction consistency of HDDBSs (see Section 5.2).

Given a global transaction $G$, we define $\prec_{eo}^G = \cup_{T \in TS(G)} \prec_t^T$ to be its execution order and $\prec_{vd}^G = \prec_{vd}^{G_0} \cup (\cup_{T \in TS(G)} \prec_{vd}^T)$ to be its value dependency.

Example 3.1 (International banking) The global transaction $G_1$ in Example 2.2 which transfers money from accounts $x_1$ at bank $A$ to account $y_1$ at bank $B$ can be expressed using the formalism as follows.

$$
G_1 = \langle TS(G_1), \prec_{vd}^{G_1} \rangle \text{ where } TS(G_1) = \{G_{1,A}, G_{1,B}\} \text{ and } \prec_{vd}^{G_1} = \{(r_{g_1}(x_1), w_{g_1}(y_1))\}
$$

$$
G_{1,A} = \langle O(G_{1,A}), \prec_{eo}^{G_{1,A}} \rangle, \text{ where } O(G_{1,A}) = \{r_{g_1}(x_1), w_{g_1}(x_1)\} \text{ and } \prec_{eo}^{G_{1,A}} = \{(r_{g_1}(x_1), w_{g_1}(x_1))\}
$$

$$
G_{1,B} = \langle O(G_{1,B}), \prec_{eo}^{G_{1,B}} \rangle, \text{ where } O(G_{1,B}) = \{r_{g_1}(y_1), w_{g_1}(y_1)\} \text{ and } \prec_{eo}^{G_{1,B}} = \{(r_{g_1}(y_1), w_{g_1}(y_1))\}
$$

In cases where remote value dependency is not important, a global transaction can be simply expressed as a set of subtransactions. For example,

$$
G_1 = \{G_{1,A}, G_{1,B}\}, \text{ where } G_{1,A} : r_{g_1}(x_1)w_{g_1}(x_1) \text{ and } G_{1,B} : r_{g_1}(y_1)w_{g_1}(y_1).\]
3.3 Executions

Definition 3.3 (Local executions) A local execution $E_t$ in $LDDBS_t$ is an interleaved sequence of operations of transactions in $T_t$, with the following property: $\forall o_i, o_j \in O(T)$ where $T \in T_t$, if $o_i \prec_T o_j$ then $o_i$ precedes $o_j$ in $E_t$.

The order in which operations are executed in $E_t$ is called the execution order of $E_t$ and is denoted as $\prec_{E_t}$.

Definition 3.4 (Global executions) A global execution $E$ in an HDDBS consists of a set of local executions, $E = \{E_1, E_2, ..., E_n\}$, where $E_t$ is the local execution at $LDDBS_t$.

The execution order of a global execution $E$ is defined as $\prec_E = \cup_{t=1}^n \prec_{E_t}$.

Example 3.2 (Flight reservation) In Example 2.1, let $E$ be an execution of transactions $L_1, L_2, G_1$ and $G_2$, where

$L_1 : r_{l_1}(a_1)w_{l_1}(a_1)r_{l_2}(a_2)w_{l_1}(a_2)$
$L_2 : r_{l_2}(b_1)w_{l_2}(b_1)$
$G_1 = \{G_{1,1}, G_{1,2}, G_{1,3}\}$, where $G_{1,1} : r_{g_1}(a_1)w_{g_1}(a_1), G_{1,2} : r_{g_1}(b_1)w_{g_1}(b_1)r_{g_1}(b_2)w_{g_1}(b_2)$ and $G_{1,3} : r_{g_1}(c_1)w_{g_1}(c_1)$
$G_2 = \{G_{2,1}, G_{2,2}\}$, where $G_{2,1} : r_{g_2}(a_2)w_{g_2}(a_2)r_{g_2}(a_2), G_{2,2} : r_{g_2}(b_1)r_{g_2}(b_2)$ and $G_{2,3} : r_{g_2}(c_1)$.

Then $E = \{E_1, E_2, E_3\}$, where

$E_1 : r_{l_1}(a_1)w_{l_1}(a_1)r_{g_1}(a_2)w_{g_1}(a_2)r_{g_1}(a_1)w_{g_1}(a_1)r_{l_2}(a_2)w_{l_1}(a_2)$
$E_2 : r_{g_1}(b_1)w_{g_1}(b_1)r_{g_1}(b_2)w_{g_1}(b_2)r_{l_2}(b_1)w_{l_2}(b_1)r_{g_2}(b_1)r_{g_2}(b_2)$
$E_3 : r_{g_1}(c_1)w_{g_1}(c_1)r_{g_2}(c_1)$.

4 Quasi Serializability

In this section, we introduce quasi serializability, a correctness criterion for concurrency control in HDDBSs [2]. We first define quasi serializable executions. Then, we present a necessary and sufficient condition of quasi serializable executions which is useful in the study of quasi serializability. We also show, in this section, that quasi serializability is a weaker criterion than serializability.
4.1 Quasi Serializable Executions

The main objective of concurrency control is to maintain transaction consistency which is usually specified in terms of correct (or allowable) interactions among transactions. Although transactions may be interleaved, a correct execution should give users an illusion that they are executed alone. This virtual "executing alone" environment is guaranteed in serializability by controlling interleavings among transactions. The serializability theory is based on the observation that a transaction never affects previously executed transactions and the assumption that it is always possible for a transaction to affect following transactions. To prevent such possible interactions, transactions are required to be executed in a serializable way. Although necessary in general, the requirement is too strong in some HDDBS applications for the following reasons. First, local transactions at different sites are executed independently. There usually exists no precedence relation between them. Second, although global transactions may introduce indirect precedence (e.g., a local transaction precedes a global transaction at one site which in turn precedes a local transaction at another site), possible interactions could be prevented more easily by controlling executions of global subtransactions than by coordinating executions of the local transactions themselves. Based on this observation, the requirement of linear precedence relation is compromised in quasi serializability for local transactions at different sites. In other words, quasi serializability focuses on the behavior of global transactions and their interactions. Executions of local transactions at different sites are not coordinated.

Definition 4.1 (Quasi serial executions) A global execution \( E = \{E_1, E_2, ..., E_n\} \) is quasi serial if

- all local executions are serializable; and
- there exists a total ordering over \( G \) such that \( \forall G_i, G_j \in G \) and \( G_i \) preceding \( G_j \) in the ordering, \( o_i < o_j \) in \( E_i \) for all \( o_i \in O(G_i) \) and \( o_j \in O(G_j) \) (1 \( \leq l \leq n \)).

Definition 4.2 (Quasi serializable executions) A global execution is quasi serializable if it is equivalent to a quasi serial execution of the same set of transactions.

The order in which global transactions are executed in an equivalent quasi serial execution is called the quasi serialization order of the execution. The quasi serialization order of an execution is not unique.

Example 4.1 The global execution \( E \) in Example 3.2 is not serializable. However, it is quasi serializable. It is equivalent to the quasi serial execution \( E' = \{E_1', E_2', E_3'\} \), where
In a quasi serializable execution, each global transaction appears to all other global transactions as an indivisible step. For example, $G_1$ and $G_2$ in Example 3.2 appears to each other as indivisible steps because they execute sequentially in the equivalent execution $E'$. Global and local transactions accessing the same site (e.g., $G_1$, $G_2$, and $L_1$ in $E'_1$) also appears to each other as indivisible steps because of serializability of local executions. On the other hand, it is possible that local transactions at different sites do not appear to each other as indivisible steps. For example, local transaction $L_1$ in $E'_1$ is not indivisible to $L_2$ in $E'_2$ in the sense that they affect each other mutually. $L_2$ reads indirectly from $L_1$ which also reads indirectly from $L_2$. In Section 5.2, we will discuss how to prevent such undesirable interactions between local transactions at different sites.

The main advantage of quasi serializability over serializability is that quasi serialization order of global transactions is compatible with their execution order. For example, $G_1$ precedes $G_2$ in the quasi serialization order because it executes before $G_2$ does in both $E'_1$ and $E'_2$. The serialization order of $G_1$ and $G_2$ at $E_2$, however, is different from their execution order. The compatibility of quasi serialization order and execution order is useful in global concurrency control in HDDBSs. It allows the GCC to enforce specific orders by just controlling the submission and interleaving of global transactions, as we will see in Section 5.

4.2 Quasi Serializability Theorem

There is a convenient graph-theoretic characterization of quasi serializable executions which is described in the following theorem. Let us first introduce the notion of indirect conflict operations and quasi serialization graphs.

Let $o_i$ and $o_j$ be operations of two different transactions in a local execution $E_l$. We say that they directly conflict with each other if they access the same data item and at least one of them is a write operation. We say that $o_i$ indirectly conflicts with $o_j$ in $E_l$ if there exist operations $o_1$, $o_2$, ..., $o_k \in \mathcal{O}(E_l) = \bigcup_{T \in \mathcal{T}} \mathcal{O}(T)$ ($k \geq 1$) such that $o_i$ directly conflicts with and precedes $o_1$ in $E_l$, $o_1$ directly conflicts with and precedes $o_2$ in $E_l$, ..., and $o_k$ directly conflicts with and precedes $o_j$ in $E_l$. Let $G_i$ and $G_j$ be two global transactions in a global execution $E$. We say that $G_i$ indirectly conflicts with $G_j$ in $E$ if one of $G_i$'s operations indirectly conflicts with one of $G_j$'s operations in a local execution of $E$. 

12
Definition 4.3 (Quasi serialization Graphs) The quasi serialization graph of a global execution $E$, denoted $QSG(E)$, is a directed graph whose nodes are the global transactions in $E$, and whose edges are all the relations $(G_i, G_j)$ ($i \neq j$) such that $G_i$ either directly or indirectly conflicts with $G_j$.

The quasi serialization graph of the global execution $E$ in Example 3.2 is shown in Figure 1.

Theorem 4.1 (Quasi serializability theorem) A global execution $E$ is quasi serializable if and only if all local executions are serializable and $QSG(E)$ is acyclic.

Proof: See Appendix A. □

4.3 Relationship Between Serializability and Quasi Serializability

Quasi serializability is a weaker criterion than serializability. In other words, each serializable execution is also quasi serializable, but not vice versa. Let $QSR$ be the set of all quasi serializable executions and $SR$ the set of all serializable executions.

Theorem 4.2 $SR \subseteq QSR$.

Proof: (1) $SR \subseteq QSR$.
Given a global execution $E$. Suppose that $E \in SR$, but $E \notin QSR$. Since $E$ is not quasi serializable, there exists a cycle in $QSG(E)$. Let the cycle be $G_{i_1} \rightarrow G_{i_2} \rightarrow \ldots \rightarrow G_{i_k} \rightarrow G_{i_1}$ where $k \geq 2$. Since $G_{i_1} \rightarrow G_{i_2}$, there exist $o_{i_1} \in O(G_{i_1})$ and $o_{i_2} \in O(G_{i_2})$ such that $o_{i_1}$ indirectly conflicts with $o_{i_2}$ in one of the local executions $E_l$. In other words, there exist local operations $o_1, o_2, \ldots, o_j$ where $j \geq 0$ such that $o_{i_1}$ directly conflicts with $o_1$, $o_1$ directly conflicts with $o_2$, $o_2$ directly conflicts with $o_3$, and $o_3$ directly conflicts with $o_{i_2}$. Therefore there is a path from $G_{i_1}$ to $G_{i_2}$ in the serialization graph of $E$. Similarly, there are paths from $G_{i_2}$ to $G_{i_3}$, from $G_{i_3}$ to $G_{i_4}$, and from $G_{i_k}$ to $G_{i_1}$. In other words, there is a cycle in the serialization graph of $E$. $E$ is not serializable. A contradiction!

(2) $SR \neq QSR$.
In Example 3.2, $E \in QSR$ but $E \notin SR$. □
Concurrency Control Based on Quasi Serializability

Concurrency control based on quasi serializability consists of two parts: scheduling global transactions in a quasi serializable fashion and controlling possible remote interactions between local transactions at different sites. In this section, we study the two issues by proposing a scheduler for quasi serializable executions and presenting a restriction on information flow in a global execution which prevents undesirable remote interactions.

5.1 Maintaining Quasi Serializability

Theorem 4.1 gives a sufficient and necessary condition for quasi serializable executions. However, it is very hard to construct a scheduler based on the theorem. The reason is that it is difficult for the GCC to predict or detect indirect conflicts between global operations because they may be introduced by local operations [5]. On the other hand, it is possible, as we mentioned before, to guarantee quasi serializability of executions by only controlling the submission of global transactions. In this subsection, we study the feature of quasi serializable executions. We first introduce the notion of access graphs of global transactions and global executions. The notion is useful because it characterizes the interleavings of global transactions based on the information the GCC has. We then show how to ensure the quasi serializability of an execution by maintaining acyclicity of its access graph.

5.1.1 Access Graphs

Informally, the access graph of a global transaction is a linear link of all local databases it accesses, while the access graph of a global execution with respect to a global transaction is the union of the access graphs of all global transactions that interleave with the transaction. The access graphs of an execution characterize the way local databases are accessed by global transactions, which, as we will see in the following, is very useful in determining its quasi serializability.

Definition 5.1 (Access graphs of global transactions) The access graph $AG(G_i)$ of a global transaction $G_i$ is an undirected graph $< V, A >$, where $V = \{ D_{i_1}, D_{i_2}, ..., D_{i_k} \}$ is the set of all local databases $G_i$ accesses ($i_1 < i_2 < ... < i_k$ and $k \geq 2$), and $A = \{(D_{ij}, D_{ij+1}) \mid 1 \leq j < k \}$.

In order to extend the notion of access graphs to global executions, we need more notations.

\footnote{Actually, $A$ could be any set of arcs that connect $D$s in $V$ in a linear order. The increasing subscript ordering is chosen in the definition for the sake of simplicity.}
We say that a transaction \( T_i \) directly interleaves with another transaction \( T_j \) in an execution \( E \) if their operations are executed concurrently in \( E \). In other words, some of \( T_i \)'s operations precede those of \( T_j \) in \( E \), while others follow \( T_j \)'s operations in \( E \). We say that \( T_i \) indirectly interleaves with \( T_j \) in \( E \) if there exist transactions \( T_1, T_2, \ldots, T_k \) (\( k \geq 0 \)) such that \( T_i \) directly interleaves with \( T_1 \), \( T_1 \) directly interleaves with \( T_2 \), ..., and \( T_k \) directly interleaves with \( T_j \). We also say that two transactions interleave with each other meaning that they either directly or indirectly interleave with each other. We use \( I(T, E) \) to denote the set of all transactions that interleave with \( T \) in \( E \).

**Definition 5.2 (Access graphs of global executions)** The access graph of a global execution \( E \) with respect to global transaction \( G_0 \) is \( AG(E, G_0) = \bigcup_{G \in I(G_0, E)} AG(G) \).

The concept of access graph is similar to that of site graph in [1]. The difference is that the access graph of an execution concerns only those global transactions that interleave with each other.

**Example 5.1** Consider an HDDBS consisting of three LDBSs, where \( a, b \in D_1, c, d, e \in D_2 \) and \( f \in D_3 \). Let \( G_1, G_2, G_3 \) be global transactions submitted to the HDDBS:

- \( G_1 = \{G_{1,1}, G_{1,2}\} \), where \( G_{1,1} : w_{g1}(a) \) and \( G_{1,2} : r_{g1}(d) \)
- \( G_2 = \{G_{2,2}, G_{2,3}\} \), where \( G_{2,2} : w_{g2}(c) r_{g2}(e) \) and \( G_{2,3} : r_{g2}(f) \)
- \( G_3 = \{G_{3,1}, G_{3,2}\} \), where \( G_{3,1} : r_{g3}(b) \) and \( G_{3,2} : w_{g3}(e) \)

Let \( L_1 \) and \( L_2 \) be two local transactions submitted to LDBS1 and LDBS2, respectively:

- \( L_1 : r_{l1}(a) w_{l1}(b) \)
- \( L_2 : r_{l2}(c) w_{l2}(d) \)

Let \( E = \{E_1, E_2, E_3\} \) be a global execution of \( G_1, G_2, G_3, L_1 \) and \( L_2 \), where

- \( E_1 : w_{g1}(a) r_{l1}(a) w_{l1}(b) r_{g1}(b) \)
- \( E_2 : w_{g2}(c) r_{l2}(c) w_{l2}(d) r_{g2}(d) w_{g2}(e) r_{g2}(e) \)
- \( E_3 : r_{g2}(f) \)

\( G_2 \) directly interleaves with \( G_1 \) in \( E \) but \( G_3 \) does not. However, \( G_3 \) indirectly interleaves with \( G_1 \) because it directly interleaves with \( G_2 \).

The access graphs of \( G_1, G_2, G_3 \) and the access graph of \( E \) with respect to \( G_1 \) are shown in Figure 2.(a), (b), (c) and (d), respectively.
That two global transactions interleave with each other implies that there may exist a quasi serialization order between them. The order, however, may be different from their execution order if there are other global transactions executed concurrently with them. For example, $G_3$ executes after $G_1$ does in both $E_1$ and $E_2$. However, it precedes $G_1$ in the quasi serialization order. The order is introduced by $G_2$ which executes concurrently with both $G_1$ and $G_3$.

On the other hand, the quasi serialization order of two global transactions is compatible with their execution order if they do not interleave with each other. In addition, not all quasi serialization orders are important in maintaining quasi serializability. For example, the quasi serialization order between $G_2$ and $G_3$ in Example 5.1 will not effect the quasi serializability of the execution because they access only one common local database.

The above idea is formalized in the following theorem using access graphs.

**Theorem 5.1** A global execution $E$ is quasi serializable if $AG(E,G)$ is acyclic for all $G \in G$.

**Example 5.2** In Example 5.1 global execution $E$ is not quasi serializable because $AG(E,G_1) = AG(E,G_2) = AG(E,G_3)$ is cyclic. Let $E'$ be the execution resulted by taking all operations of $G_1$ away from $E$. Then $E'$ is quasi serializable. It is not hard to verify that $AG(E',G_2) = AG(E',G_3)$ is acyclic.

**Proof of the theorem:** Let $E$ be a global execution. Assume that $AG(E,G)$ is acyclic for all $G \in G$. We show that $E$ is quasi serializable by contradiction.
Suppose that $E$ is not quasi serializable. Then there exists a cycle in $QSG(E)$. Let the cycle be $G_{i_1} \rightarrow G_{i_2} \rightarrow \ldots \rightarrow G_{i_k} \rightarrow G_{i_1}$. The proof consists of the following two parts.

1. $G_{i_1}, G_{i_2}, \ldots, G_{i_k}$ interleave with each other in $E$

   We prove by induction on $k$, the number of global transactions in the cycle.

   **Basis step:** ($k = 2$) Since $G_{i_1} \rightarrow G_{i_2} \rightarrow G_{i_1}$, $G_{i_1}$ directly interleave with $G_{i_2}$.

   **Induction hypothesis:** Assume that it is true for cycles of less than $k$ global transactions.

   **Induction step:** There exist two global transactions $G_{i_p}$ and $G_{i_q}$ ($1 \leq p \neq q \leq k$) such that they directly interleave with each other. Too see this, notice that, otherwise, all $G_{i_1}$'s operations would precede those of $G_{i_2}$, all $G_{i_2}$'s operations would precede those of $G_{i_3}$, ..., all $G_{i_{k-1}}$'s operations would precede those of $G_{i_k}$. In other words, all $G_{i_1}$'s operations precede those of $G_{i_k}$, a contradiction to $G_{i_1} \rightarrow G_{i_2}$.

   Let us transform the cycle $G_{i_1} \rightarrow G_{i_2} \rightarrow \ldots \rightarrow G_{i_k} \rightarrow G_{i_1}$ into two smaller ones by combining $G_{i_p}$ and $G_{i_q}$ together into one node $G_{i_1(p,q)}$, as shown in Figure 3. This is possible because of the transitive property of interleave relation. In other words, a transaction interleaves with $G_{i_p}$ if and only if it interleaves with $G_{i_q}$. We now have two cycles: $G_{i_1(p,q)} \rightarrow G_{i_2(p,q)} \rightarrow \ldots \rightarrow G_{i_{k-1}(p,q)} \rightarrow G_{i_k}$. In each cycle, there are less than $k$ global transactions. According to the induction hypothesis, they interleave with each other. In specific, all transactions $(G_{i_1}, G_{i_2}, \ldots, G_{i_k})$ interleave with $G_{i_1(p,q)}$ (i.e., either $G_{i_p}$ or $G_{i_q}$). Therefore, they all interleave with each other in $E$.

2. $AG(E, G)$ is acyclic, where $G = G_{i_1}, G_{i_2}, \ldots, G_{i_k}$

   Since $G_{i_1}$ conflicts with $G_{i_2}$, they must access a common local database $D_{i_1}$. Similarly, $G_{i_j}$ and $G_{i_{j+1}}$ access a common local database $D_{i_j}$ for $j = 2, 3, \ldots, k$ and $G_{i_{k+1}} = G_{i_1}$. According to Definition 5.1, there is a path from $D_{i_1}$ to $D_{i_2}$ in $AG(G_{i_1})$. Similarly, there is a path from $D_{i_2}$ to $D_{i_{j+1}}$ in $AG(G_{i_j})$ for $j = 2, 3, \ldots, k$ and $D_{i_{k+1}} = D_{i_1}$. In other words, there is a cycle $D_{i_1} \rightarrow D_{i_2} \rightarrow \ldots \rightarrow D_{i_k} \rightarrow D_{i_1}$ in $AG(E, G_{i_1})$ (as well as in $AG(E, G_{i_2}), \ldots, AG(E, G_{i_k}))$. A contradiction! □

It is worth noting that acyclicity of access graphs does not guarantee serializability, as the following example shows.

**Example 5.3** Consider execution $E'$ in Example 4.1. Since $G_1$ and $G_2$ execute sequentially, they do not directly interleave with each other. Therefore, $AG(G_1, E') = AG(G_1)$ is acyclic. Similarly, $AG(G_2, E')$ is also acyclic. However, $E'$ is not serializable. □
5.1.3 Scheduling Global Transactions

According to Theorem 5.1, the quasi serializability of a global execution is assured if global transactions are submitted in such a way that the global transactions whose access graphs form a cyclic graph do not interleave with each other. For example, $G_1$ and $G_2$ in Example 4.1 access more than one common local database. Their quasi serialization order at these sites may be inconsistent (e.g., $G_1 \rightarrow G_2$ at $LDBS_1$ and $G_2 \rightarrow G_1$ at $LDBS_2$). To guarantee a specific quasi serialization order at all sites, they must be submitted and executed sequentially. In addition, no other global transactions should execute concurrently with both of them.

We now present such a scheduler that guarantees quasi serializability of executions by controlling submission of global transactions. The scheduler maintains the following data structures.

- $active_{xact}$: the set of currently active global transactions.
- $delayed_{xact}$: the set of global transactions that are delayed by the scheduler. A global transaction is delayed if its submission will create a cycle in the current access graph of the execution.
- $xact_{access_{graph}}$: the access graph of the global transaction being scheduled.
- $exec_{access_{graph}}$: the access graph of the current execution. It is the union of access graphs of all currently active global transactions and those transactions that interleave with them.

The function $ACCESS_{GRAPH}$ will be used in the procedures of the scheduler. It takes as an argument a global transaction and generates as output the access graph of that transaction.

The scheduler consists of two parts. The first procedure, $TRANSACTION_{SUBMISSION}$,
receives global transactions and either submits or delays them according to the current execution environments. The second procedure, TRANSACTION_TERMINATION, is activated when a global transaction terminates (either commits or aborts). It removes the global transaction from active_xact and (when no transaction is active) releases all global transactions from exec_access_graph. In the latter, it also tries to resubmit all delayed global transactions.

```
procedure TRANSACTION_SUBMISSION(Gi);
begin
    xact_access_graph ← ACCESS_GRAPH(Gi); (1)
    if exec_access_graph + xact_access_graph is cyclic
        then delayed_xact ← delayed_xact ∪ {Gi}; (3)
        else active_xact ← active_xact ∪ {Gi}; (4)
        exec_access_graph ← exec_access_graph + xact_access_graph; (5)
    endif
end;
```

```
procedure TRANSACTION_TERMINATION(Gi);
begin
    active_xact ← active_xact - {Gi}; (1)
    if active_xact = ∅ (2)
        then exec_access_graph ← ∅; (3)
        for Gj ∈ delayed_xact do
            TRANSACTION_SUBMISSION(Gj); (5)
        endfor
    endif
end;
```

The scheduler works in a stepwise manner. At first, it keeps receiving global transactions and submits them whenever possible (i.e., creating no cycle in the access graph of the current execution). Eventually, it will reach a point after which no more global transaction could be submitted without creating cycles. It waits until all active global transactions commit and repeats the process.

Example 5.4 Consider global transactions G1, G2 and G3 in Example 5.1. Suppose that their operations are submitted in the order shown in E. Since G1 and G2 access only one common database (LDBS2), the union of their access graphs is acyclic. Therefore, operations w91(a), w92(c), r91(f)
and $r_{g1}(d)$ are scheduled immediately. The operations of $G_3$, however, will be delayed because it access more than one common database ($LDBS_1$ and $LDBS_2$) with $G_1$. They will be scheduled after both $G_1$ and $G_2$ finish. Thus, the execution is

$$E_i : w_{g1}(a) r_{t1}(a) w_{t1}(b) r_{g2}(b)$$

$$E_2 : w_{g2}(c) r_{t2}(c) w_{t2}(d) r_{g2}(d) w_{g2}(c)$$

$$E_3 : r_{g2}(f)$$

It is not hard to see that the global execution $E' = \{E_1', E_2', E_3'\}$ is quasi serializable. Notice that $G_1$ and $G_2$ execute concurrently in $E'$.

The proposed scheduler generates quasi serializable executions only because it groups global transactions in such a way that transactions in the same group can interleave arbitrarily (i.e., their quasi serialization order at a specific site is not important). Since global transactions at different groups do not interleave with each other, their quasi serialization order is compatible with their execution order.

**Theorem 5.2** The scheduler generates quasi serializable executions only.

**Proof:** Let $E$ be a global execution generated by the scheduler. Let $t_1, t_2, ..., t_k, (k \geq 1)$, be the time when step (3) of the procedure TRANSACTION_TERMINATION is executed. Without loss of generality, let us assume $t_1 < t_2 < ... < t_k$. Let $E_1$ be the subexecution of $E$ from the beginning to $t_1$, $E_2$ be the subexecution of $E$ from $t_1$ to $t_2$, and so forth. Then $E$ is the concatenation of $E_1, E_2, ..., E_k$ in the order. Each global transaction is involved in exactly one subexecution. Let $G(E_i)$ be the set of global transactions involved in $E_i$ ($i = 1, 2, ..., k$). Then $G = \cup_{i=1}^{k} G(E_i)$.

Let us consider $E_i$ (1 ≤ $i$ ≤ $k$). All global transactions in $G(E_i)$ are submitted by procedure TRANSACTION_SUBMISSION. The step (2) in the procedure guarantees that the access graphs of $E_i$ with respect to these global transactions are acyclic. According to Theorem 5.1, $E_i$ is quasi serializable. Therefore, there exists a quasi serial execution $E_i'$ of the same set of operations such that $E_i$ is equivalent to $E_i'$. Let $E'$ be the concatenation of $E_1', E_2', ..., E_k'$ in the order. Then $E'$ is quasi serial and is equivalent to $E$. In other words, $E$ is quasi serializable. □

The following example shows that the scheduler produces both serializable and non-serializable executions.

**Example 5.5** Consider an HDDBS consisting of two LDBSs, where $x \in D_1$ and $y \in D_2$. The following global transactions are submitted to the HDDBS:

$$G_1 = \{G_{1,1}, G_{1,2}\}, \text{ where } G_{1,1} : w_{g1}(x) \text{ and } G_{1,2} : w_{g1}(y)$$
\[ G_2 = \{G_{2,1}, G_{2,2}\}, \text{ where } G_{2,1} : \tau_{g_2}(x) \text{ and } G_{2,2} : \tau_{g_2}(y) \]

Let \( L_1 \) be a local transaction submitted to \( LDBS_1 \):

\[ L_1 : \tau_{t_1}(x)w_{t_1}(x) \]

Let \( E_1 \) and \( E_2 \) be the local executions at \( LDBS_1 \) and \( LDBS_2 \), respectively:

\[ E_1 : \tau_{t_1}(x)w_{g_1}(x)\tau_{g_2}(x)w_{t_1}(x) \]
\[ E_2 : w_{g_1}(y)\tau_{g_2}(y) \]

The global execution \( E = \{E_1, E_2\} \) can be generated by the scheduler. To see this, notice that when \( G_1 \) commits, no global transaction is active. Therefore, the access graph of \( G_1 \) is released immediately after the commitment. Then \( G_2 \) is submitted immediately because it is the only active global transaction. However, \( E \) is not serializable.

5.2 Preventing Undesirable Remote Interactions

As we have mentioned before, local transactions at different sites may affect each other indirectly via global transactions. We now discuss how to prevent the undesirable interactions by controlling information flow in an execution. Let us first formalize the notions of indirect read from relation of transactions and information flow in a global execution.

Local transactions at different sites do not affect each other directly. They affect each other indirectly in the sense that the change made by one transaction at a site is propagated to other sites by global transactions which is then observed by local transactions at the sites. For example, in Example 2.2, the money deposited by \( L_1 \) at Bank A is transferred by global transaction \( G_1 \) to Bank B. The balance read by \( L_2 \) therefore includes the money deposited by \( L_1 \).

We say that a read operation directly reads from a previous write operation if they both access the same data item and there is no other write operation (to the same data item) between them. We say that a read operation \( o_i \) of local transaction \( L_i \) at a site indirectly reads from a write operation \( o_j \) of another local transaction \( L_j \) at another site if there exists \( o_1, o_2, \ldots, o_k \) (\( k \geq 2 \) and \( k \) is even) such that \( o_1 \) directly reads from \( o_2 \), \( o_1 \) value depends on \( o_2 \) (i.e., \( (o_2, o_1) \in \prec_{cd}^T \) for some \( T \in T \)), \( o_2 \) directly reads from \( o_3, \ldots, \) and \( o_k \) directly reads from \( o_j \).

**Definition 5.3 (Indirect read from relation of executions)** The indirect read from relation of a global execution \( E \) is a binary relation:

\[ \prec_{ij}^E = \{(T_i, T_j) | 1. \exists i \neq j \text{ such that } T_i \in T_i \text{ and } T_j \in T_j; \]
\[ 2. \exists o_i \in \mathcal{W}(T_i) \text{ and } o_j \in \mathcal{R}(T_j) \text{ such that } (o_i, o_j) \in (\prec_{ij}^E \cup (\cup_{T \in T} \prec_{cd}^T))^* \} \]
Given an execution $E$ of transactions $T$ and a time $t$ in its lifetime. Let $\delta_{\text{max}}$ be the maximum elapsed time of a single local transaction and $G(t)$ be the set of global transactions that are active in $(t - \delta_{\text{max}}, t + \delta_{\text{max}})$ in $E$.

**Definition 5.4 (Information flow graphs of executions)** The information flow graph of a global execution $E$ at time $t$, $\text{IFG}(E,t)$, is a directed graph $< V, A >$, where $V = \{ D, D_1, ..., D \}$ and $A = \{(D, D_i) \mid \exists \exists G_i \in G(t), o_i \in \mathcal{A}(G_i,i) \text{ and } o_j \in \mathcal{V}(G_j,i) \text{ such that } (o_i, o_j) \in \eta_{G(i,j)} \}$.

The information flow graph of a global execution characterizes the possible remote interactions between local transactions at different sites that are active in a specific period of time. In other words, $(D_i, D_j) \in \text{IFG}(E,t)$ means that local transactions active in $(t - \delta_{\text{max}}, t + \delta_{\text{max}})$ at $D_j$ may be affected by those at $D_i$. On the other hand, the absence of the arc implies that they are independent.

**Theorem 5.3** Given an execution $E$, $\text{IFG}(E,t)$ is acyclic if $\text{IFG}(E,t)$ is acyclic for all $t$ in $E$'s lifetime.

**Proof:** Suppose that $\text{IFG}(E,f)$ is cyclic: $\exists T_i \in L_i, T_j \in L_j (i \neq j)$ such that $(T_i, T_j) \in \text{IFG}(E,f)$ and $(T_j, T_i) \in \text{IFG}(E,f)$. Then, there exist $o_{p_1} \in \mathcal{V}(G_{p_1,i}), o_{p_2} \in \mathcal{V}(G_{p_2,i}), ..., o_{p_i} \in \mathcal{V}(G_{p_i,i}), o_{p_i} \in \mathcal{V}(G_{p_i,i})$ ($l \geq 1$), where $G_{p_1}, G_{p_2}, ..., G_{p_i} \in G$, such that $o_{p_1}$ indirectly depends on $o_{p_{l-1}}$, ..., $o_{p_2}$ indirectly depends on $o_{p_1}$. Similarly, there exist $o_{q_1} \in \mathcal{V}(G_{q_1,i}), o_{q_2} \in \mathcal{V}(G_{q_2,i}), ..., o_{q_{m-1}} \in \mathcal{V}(G_{q_{m-1},i}), o_{q_m} \in \mathcal{V}(G_{q_m,i})$ ($m \geq 1$), where $G_{q_1}, G_{q_2}, ..., G_{q_m} \in G$, such that $o_{q_1}$ indirectly depends on $o_{q_{m-1}}$, ..., $o_{q_2}$ indirectly depends on $o_{q_m}$. Therefore, there exist $t_1, t_2$ such that $T_i, G_{p_1}, G_{p_2}, ..., G_{p_i}$ and $T_j$ are all active at time $t_1$, and $T_i, G_{q_1}, G_{q_2}, ..., G_{q_m}$ and $T_j$ are all active at time $t_2$. Clearly, $t_1 \in (t_2, t_2 + \delta_{\text{max}})$ (assume that $t_1 > t_2$). In other words, $G_{p_1}, ..., G_{p_i}, G_{q_1}, ..., G_{q_m} \in G(t_1)$. Therefore, $\text{IFG}(E,t_1)$ is also cyclic. 

Therefore, the transaction consistency of an HDDBS can be maintained using quasi serializable executions if global transactions are scheduled in such a way that the information flow graph is always acyclic. This can be done by delaying those global transactions whose executions would introduce cycles in the current information flow graph, as illustrated in the following example.

**Example 5.6 (International banking)** In Example 2.2, let $t$ be the time when $w_{1}(X_1)$ is executed. Both $G_1$ and $G_3$ are active in $(t - \delta_{\text{max}}, t + \delta_{\text{max}})$. Therefore, $\text{IFG}(E,t) = \{(A, B), (B, A)\}$ is cyclic. To maintain acyclicity of the value dependency graph of $E$, $G_3$ is delayed until $t + \delta_{\text{max}}$. In other words, $G_3$ executes after $L_1$ at site $A$. 

22
\( E' = \{ E'_1, E'_2 \} \), where
\[
E'_1 : \tau_{t_1}(x_1)w_{t_1}(x_1)r_{g_1}(x_1)w_{g_1}(x_1)\tau_{t_2}(x_2)w_{t_2}(x_2)\tau_{g_2}(x_2)w_{g_2}(x_2)
\]
\[
E'_2 : \tau_{t_3}(y_1)w_{t_3}(y_1)\tau_{g_3}(y_1)w_{g_3}(y_1)\tau_{t_4}(y_2)w_{t_4}(y_2)\tau_{g_4}(y_2)w_{g_4}(y_2)
\]

It is not hard to see that \( L_1 \) and \( L_2 \) interact with each other in a partial order, \( L_1 \rightarrow L_2 \) in \( E' \).

6 Conclusions

In this paper, we proposed a new approach to maintain the HDDBS consistency based on quasi serializability theory. A global execution is quasi serializable if its local executions are serializable and it is equivalent to another execution of the same set of transactions in which global transactions are executed sequentially. The basic idea of the quasi serializability approach is to take advantage of existing local concurrency controllers and the independence of local transactions. The approach is novel in that it maintains transaction consistency not only by scheduling transactions, as in the serializability approach, but also by controlling other aspects of transaction execution. The approach is attractive in HDDBSs where undesirable interactions between local transactions at different sites are impossible or can be easily prevented. The quasi serializability approach differs from that of serializability in that it focuses on the behavior of global transactions only and therefore can be implemented without violating local autonomy.

Quasi serializability is a weaker correctness criterion than serializability. In other words, there are aspects of HDDBS consistency that are maintained by serializable executions but not by quasi serializable executions. We studied the strengths and weaknesses of quasi serializability as a correctness criterion for concurrency control in HDDBSs. Generally speaking, quasi serializability is good for those aspects of HDDBS consistency that are concerned with execution of transactions in a single site, as well as execution of global transactions. It is, however, not good for those aspects of consistency that are concerned with local executions at different sites.

There are two issues in maintaining HDDBS consistency using the quasi serializability approach: scheduling transactions in a quasi serializable fashion and controlling other execution aspects (information flow, in this case) to maintain those aspects of consistency that are not guaranteed by quasi serializable executions. We described a scheduler that produces quasi serializable executions only. The main advantage of the scheduler is that it does not violate local autonomy. In addition, it allows concurrent execution of some global transactions. We also presented the restriction on the information flow in a global execution that prevents undesirable interactions between local transactions at different sites.
There are several issues remaining for further investigation. First, the discussion in this paper is based on the assumption that no failure occurs in the execution of transactions. An important issue is therefore how to maintain quasi serializability of executions in the face of failure. Generally, it is easier to recover a quasi serializable execution than a serializable execution. Even though, it is still not clear whether it can be done without violating local autonomy. Another issue that remains to be investigated is whether the quasi serializability approach provides, in general, better performance than the serializability approach. Concurrency controllers based on quasi serializability provide a higher degree of concurrency than those based on serializability. However, there is additional overhead in maintaining the aspects of consistency that are not guaranteed by quasi serializable executions (e.g., controlling remote value dependency of global transactions). It is not clear under what conditions the quasi serializability approach provides better performance.

Appendix A. Proof of the Quasi Serializability Theorem

(if) Suppose \( E = \{ E_1, E_2, ..., E_m \} \) is a global execution over \( G \cup L \), where \( G \) is a set of global transactions and \( L \) is a set of local transactions. Since QSG(\( E \)) is acyclic, it may be topologically sorted. Let \( i_1, i_2, ..., i_n \) be a permutation of \( 1, 2, ..., n \) such that \( G_{i_1}, G_{i_2}, ..., G_{i_n} \) is a topological sort of QSG(\( E \)). For each local execution \( E_l \) (1 \( \leq l \leq m \)), assume that \( G_{i_{1,l}}, G_{i_{2,l}}, ..., G_{i_{n,l}} \) are the global subtransactions that appear in \( E_l \). We show, below, that there is another serializable local execution \( E'_l \), equivalent to \( E_l \), such that all of \( G_{i_{p,l}}'s \) operations precede \( G_{i_{r,l}}'s \) operations in \( E'_l \), all of \( G_{i_{q,l}}'s \) operations precede \( G_{i_{r,l}}'s \) operations in \( E'_l \), and so on.

In order to construct the equivalent execution \( E'_l \), let us group the operations in \( E_l \) into \( n \) operation sets based on global subtransactions.

For \( p = 1 \) to \( n - 1 \) do

\[ \mathcal{O}(i_{p,l}) = \{ o \mid o \in \mathcal{O}(E_l) - \bigcup_{k=1}^{p-1} \mathcal{O}(i_{k,l}) \text{ and } \}
\]


\[ \begin{align*}
\text{either } o & \in G_{i_{p,l}} \text{ or } o \text{ conflicts with one of } G_{i_{r,l}}'s \text{ operations} \\
\end{align*} \]

\[ \mathcal{O}(i_{n,l}) = \{ \text{everything left} \} \]

Informally, \( \mathcal{O}(i_{p,l}) \) consists of all \( G_{i_{p,l}}'s \) operations and those operations in \( E_l \) that must precede some of \( G_{i_{q,l}}'s \) operations in any execution which is equivalent to \( E_l \) (but not in previous \( \mathcal{O}(i_{p,l}) \)'s). \( \mathcal{O}(i_{p,l}) \)s are well defined and have the following properties:

1. \( \mathcal{O}(G_{i_{p,l}}) \subseteq \mathcal{O}(i_{p,l}) \), for \( 1 \leq p \leq n \).
2. \( \mathcal{O}(E_l) = \bigcup_{k=1}^{n} \mathcal{O}(i_{k,l}) \).
3. \( \mathcal{O}(i_{p,l}) \cap \mathcal{O}(i_{q,l}) = \emptyset \), for \( 1 \leq p < q \leq n \).
Let \( fse \) be the constructor that constructs from an \( OP \) a subexecution which has the same conflict relations as \( E_i \) as far as the operations in \( OP \) are concerned. This can be done by ordering all operations in \( OP \) in the same way as in \( E_i \).

Now \( E_i' \) can be constructed as follows:

\[
fse(\mathcal{OP}(i_1, l)) \circ \ldots \circ fse(\mathcal{OP}(i_n, l))
\]

where \( \circ \) stands for concatenation of subexecutions. We claim that:

1. \( E_i' \) involves the same transactions as \( E_i \);
2. \( E_i' \) is equivalent to \( E_i \); and
3. Global subtransactions in \( E_i \) are executed sequentially in \( E_i' \).

The correctness of the first and the last statements are clear. We now show that the second statement is also true.

Let \( o_i \) and \( o_j \) be two operations in \( E_i \) such that \( o_i \) conflicts with \( o_j \). There exists an integer \( p \) such that \( o_j \in \mathcal{OP}(i_p, l) \). If \( p < n \), then \( o_j \) either belongs to \( G_{i_p,i} \) or conflicts with one of \( G_{i_p,i} \)'s operations, and so is \( o_i \). Therefore, either \( o_i \in \mathcal{OP}(i_q, l) \), where \( q < p \), or \( o_i \in \mathcal{OP}(i_p, l) \) by the definition of \( \mathcal{OP}(i_p, l) \). In either case, \( o_i \prec_{E_i'} o_j \). This is also true when \( p = n \). So, \( o_i \) also conflicts with \( o_j \) in \( E_i' \).

Let \( E' = \{ E'_1, E'_2, \ldots, E'_m \} \), then \( E' \) is quasi serial and equivalent to \( E \). Therefore, \( E \) is quasi serializable.

(only if) Let \( E \) be a quasi serializable global execution. Again, we assume that \( G_1, G_2, \ldots, G_n \) are the global transactions in \( E \). Let \( E' \) be a quasi serial global execution which is equivalent to \( E \). Then \( \text{QSG}(E) = \text{QSG}(E') \). Since \( G_1, G_2, \ldots, G_n \) are executed sequentially in \( E' \) and one operation can only conflict with subsequent operations in an execution, \( \text{QSG}(E') \) is acyclic and so is \( \text{QSG}(E) \).

References


