Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1989

Supporting Value Dependency for Nested Transactions in InterBase

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Weimin Du

Report Number:

89-885

Elmagarmid, Ahmed K. and Du, Weimin, "Supporting Value Dependency for Nested Transactions in InterBase" (1989). *Department of Computer Science Technical Reports.* Paper 753.
https://docs.lib.purdue.edu/cstech/753

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

SUPPORTING VALUE DEPENDENCY FOR NESTED TRANSACTIONS IN INTERBASE

Ahmed K. Elmagarmid
Weimin Du

CSD-TR-885
May 1989

Supporting Value Dependency for Nested Transactions in InterBase[1]

Ahmed K. Elmagarmid and Weimin Du
Computer Sciences Department
Purdue University
West Lafayette, IN 47907
ahmed@cs.purdue.edu
(317)-494-1998

[1]This work is supported by a PYI Award from NSF under grant IRI-8857952 and grants from AT&T Foundation, Tektronix, and Mobil Oil.

Abstract

A value dependency of a transaction is a relation between a value written by an operation and a value read by a previous operation. In a nested transaction, value dependencies among operations of different subtransactions have significant effects on its execution. It is our goal in this paper to study these effects. The main contributions of this paper are (1) a general approach to value dependency control, (2) conditions of nested transactions under which value dependency control can be performed, and (3) a sufficient condition of value dependency for quasi serializability, a correctness criterion for heterogeneous distributed database systems.

1 Introduction

Nested transactions are an extension of simple transactions [Gra81], and have been introduced as an approach to reliable distributed computing [Mos85]. They have been widely studied in distributed systems and transaction processing systems.

The fact that subtransactions of a nested transaction may be executed concurrently and in a distributed fashion has significant effects on the execution of nested transactions. Much research has been devoted to the related issues (e.g., concurrency control, commitment, recovery, and deadlock detection). In this paper, we discuss a new issue: value dependency within a nested transaction. Value Dependency can be loosely defined as a relation between a value written by an operation and a value read by a previous operation. The issue was raised in the study of InterBase, a project integrating pre-existing databases to support global applications accessing more than one database. The result presented in this paper, however, also applies to other database environments.

Value dependency of a nested transaction is a part of the semantics of the transaction. Traditionally (e.g., in classical serializability theory), it is assumed that no knowledge of transaction semantics is available[1]. For example, in a simple transaction, no information of value dependency is explicitly given at the system level. Instead, we assume that there is a value dependency between each pair of write and previous read operations. This assumption, however, does not apply to nested transactions. The reason is that operations of a nested transaction are no longer executed in any pre-specified linear order. As a result, certain information of value dependency have to be explicitly used in the execution of a nested transaction in order to preserve its semantics. Dependency must be given at the system level in order to execute nested transactions properly.

Value dependency has significant effects on the execution of nested transactions. For example, not all nested transactions are well formed in the sense that they will never be executed to completion correctly. One goal of this paper is to identify conditions of well formed nested transactions. This paper presents a list of subclasses of nested transactions that are well formed at various levels.

[1]Serializability theory has already been extended by augmenting with semantics to represent nested computations conveniently (see, e.g., [BBG89].)

Another issue we shall study in this paper is the effect of value dependency on transaction consistency of database systems. In [DE89], we studied this problem for heterogeneous distributed database environments, resulting in the theory of quasi serializability, a correctness criterion for global concurrency control. In this paper, we shall focus on how restrictions on value dependency simplify transaction consistency problems. We do so by giving a sufficient condition of value dependency for quasi serializability.

The rest of the paper is organized as follows. In section 2, we introduce the notion of value dependency within a nested transaction. We also discuss the general execution issues of nested transactions in terms of value dependency. In section 3, we discuss the effects of value dependency on executions by studying various subclasses of well formed nested transactions. Then, in section 4, we study the effects of value dependency on transaction consistency. Finally, we conclude the paper in section 5.

2 Value Dependency within a Nested Transaction

In this section, we introduce the notion of value dependency within a nested transaction. We also discuss how to support value dependency in distributed environments.

2.1 Terminology

A (simple) *transaction* is a sequence of primitive operations that transform a database from one state to another. Each primitive operation either reads or writes a data item. The value written by a write operation is potentially a function of all values previously read by the same transaction. This function defines a type of *value dependency* between a write operation and previous read operations. The function, however, is usually not explicitly known to the scheduler. A transaction, as defined above, is a pure syntactic object. No semantic information is specified in this simple transaction model.

Nested transactions are an extension of the traditional notion of transactions. The difference between transactions and nested transactions is that nested transactions have more internal structure. Each Nested transaction consists of either a group of primitive operations or a group of nested trans-

actions (called *subtransactions* of the containing nested transaction).

A *simple subtransaction* is a subtransaction consisting of primitive operations only. Simple subtransactions behave like simple transactions in many ways. For example, there is also a value dependency between each pair of write and previous read operations of a simple subtransaction. Like in simple transactions, value dependency within a simple subtransaction is usually not explicitly given at the system level (e.g., schedulers).

In a nested transaction, value dependency also exists between operations of different simple subtransactions, as the following example shows.

Example 2.1 Let us consider a distributed banking database that consists of two element databases, A and B, representing two branches of the bank. A customer wants to move his account from branch A to B. A possible transaction for this operation is as follows.

```
begin transaction T
    input(oldaccount, newaccount)
    begin subtransaction a_subtran
        balance = read(oldaccount)
        delete(oldaccount)
    end subtransaction a_subtran
    begin subtransaction b_subtran
        create(newaccount)
        write(newaccount, balance)
    end subtransaction b_subtran
    commit
end transaction T
```

The amount, *balance*, that *b_subtran* writes into *newaccount* must be the same as the amount, *balance*, that *a_subtran* read from *oldaccount*. In other words, the value written by *b_subtran* is a function of the value read by *a_subtran*. □

Usually, the value dependency is intrinsic to the application and cannot be avoided in the execution. In example 2.1, the value dependency between subtransactions *a_subtran* and *b_subtran* is a part of the semantics of transaction T. Every implementation of T has to support this dependency in some way.

In order to study the effects of value dependency, let us formalize the

notions of nested transactions and value dependency within a nested transaction.

A nested transaction can be viewed as a combination of a collection of operations, an internal structure, and a value dependency relation. Formally, a nested transaction T is a triple $< O, P_1, P_2 >$, where O is a collection of primitive operations. P_1 is a partial order over O, representing the internal structure of T and P_2 is a relation over O, representing the value dependency among operations of different subtransactions of T. By default, we assume that $(o_i, o_j) \in P_1$ if o_i and o_j belong to the same simple subtransaction, o_i is a read operation, o_j is a write operation, and o_i precedes o_j in T. Given $o_i, o_j \in O$, $(o_i, o_j) \in P_2$ if o_i is a read operation of one simple subtransaction, o_j is a write operation of another simple subtransaction, and the value written by o_j is a function of the value read by o_i.

Value dependency within a nested transaction can be viewed at different levels (e.g., operation level, subtransaction level, and site level). Given a transaction T, its operations can be grouped into *operation sets*, o_1, o_2, ..., o_n, according to a given level of abstraction. For example, at the subtransaction level, an operation set will consist of all operations in a subtransaction. We say that there is a value dependency between o_i and o_j if the value written by one of o_j's write operation is a function of the value (or values) read by (at least) one of o_i's read operation. In the above example, there is a value dependency between subtransactions *a_subtran* and *b_subtran*.

Value dependency among operation sets can be characterized by value dependency functions. Let T be a nested transaction and o_1, o_2, ..., o_n be the operation sets of T. The *dependent function* of o_i, denoted $D(o_i)$, is the set of all other operation sets to which there is a value dependency from o_i. The operation sets in $D(o_i)$ are called the dependents of o_i. The *antecedent function* of o_i, denoted $S(o_i)$, is the set of all other operation sets from which there is a value dependency to o_i. The operation sets in $S(o_i)$ are called the antecedents of o_i. Note that functions D and S are mutually redundant. They are chosen to simplify the forthcoming discussions.

Value dependencies can also be classified based on the location of the antecedent and dependent.

- *Local value dependency*: both the antecedent and dependent are at the same site;
- *Remote value dependency*: the antecedent and dependent are at dif-

ferent sites.

The value dependency between subtransactions *a_subtran* and *b_subtran* in example 2.1 is a remote one.

2.2 Value Dependency Control

One of the main advantages of nested transactions over simple transactions is that they allow subtransactions of a containing nested transaction to execute concurrently. The execution should not only be serializable with respect to subtransactions, but should also meet the requirements of value dependency of the nested transaction. In this subsection, we discuss general issues of value dependency control, i.e., how to support value dependency in the execution of a nested transaction.

Value dependency control is not a problem for simple transactions. In a simple transaction, execution order of operations is given in the transaction and is compatible with the value dependency relation. For nested transactions, however, the situation is quite different. Operations of a nested transaction might be executed concurrently, or even in a distributed fashion. For those operations of a nested transaction that have no value dependency, physical execution order is not important. For the operations with value dependency, however, the order is important. To enforce this order, the value dependency information must be given explicitly at the system level (e.g., to the schedulers) and used in the execution of the containing nested transaction.

In example 2.1, for instance, the operation **read**(*amount*) must be executed physically before the operation **write**(*amount*), even though they are at different sites. The order is required by the value dependency between the two operations.

As we have mentioned, a transaction processing mechanism that is to execute nested transactions correctly needs information regarding these dependencies. One alternative is to use specially designed schedulers. Such schedulers should produce schedules meeting not only the consistency constraints but also the value dependency requirements. For example, a two phase locking scheduler can be modified such that a write operation should not be granted a lock until all its antecedents have completed.

A second alternative, which we shall use in the forthcoming discussion,

is to use a separate protocol to enforce the appropriate execution order of operation sets of each nested transaction. This alternative is chosen for its simplicity and ease of understanding. Following is such a protocol which coordinates the submission of operation sets of a nested transaction according to their value dependency. Operation sets are submitted to this protocol according to the internal structure of their containing nested transaction. For example, operation sets in a simple subtransaction should be submitted sequentially in the order they appear .

1. For each $o \in S(o_i)$ do
   wait until o is done and the results arrive (if o is at a different site)
2. Submit o_i to scheduler
3. For each $o \in D(o_i)$ which is at different site from o_i do
   send the results of o_i to o (if o is at a different site)

The above protocol guarantees that the operation sets of a nested transaction will be executed in a proper order. However, even in conjunction with concurrency control protocols, it is still not sufficient to guarantee the progress of execution in the face of circular dependencies. To guarantee such progress, nested transactions themselves must be well formed in terms of value dependency.

3 Well Formed Nested Transactions

A nested transaction is *well formed* at an abstract level if, when it is executed alone, it can be executed to completion properly using the protocol given in the last section. By proper execution, we mean that the execution orders are compatible with the internal structure of the nested transaction. For example, operations of a simple subtransaction should be executed in the order specified by this subtransaction. The execution protocol guarantees that execution orders are also compatible with the value dependency relations of nested transactions. In this section, we focus on subclasses of nested transactions that are well formed at various abstract levels.

Not all nested transactions are well formed in any abstract level, as the following example shows.

Example 3.1 Let us consider a distributed database system consisting of two element databases, D_1 and D_2. Data items a and b are at D_1 and data items c and d are at D_2. Let T be a nested transaction of two subtransactions s_1 and s_2 running at D_1 and D_2 respectively.

$$s_1 : w(a)\ r(b)$$
$$s_2 : w(c)\ r(d)$$

Suppose that the value dependencies (at the operation level) between s_1 and s_2 are

$$D(r(b)) = \{w(a)\}\ S(w(a)) = \{r(b)\}, \text{ and}$$
$$D(r(d)) = \{w(c)\}\ S(w(c)) = \{r(d)\}$$

Then T is not well formed at any abstract level. The two subtransactions depend on each other and will be deadlocked. They will never be executed to completion using the protocol given in section 2. □

Value dependency has significant effects on the execution of nested transactions. First, nested transactions in different classes (of well formed nested transactions) have different system support needs, and hence different implementation performance. The lower the level at which a transaction is well formed, the more system support it requires. Therefore, it is expensive to support low level (e.g., operation level) value dependency. Second, a low level value dependency allows flexible control of nested transactions and therefore yields a high degree of "intra transaction concurrency". A nested transaction which is well formed at one level is also well formed at lower levels, but not necessarily at higher levels.

Since local value dependency is much easier and more efficient to implement than remote value dependency, we shall be interested only in remote value dependency whenever performance is analyzed.

3.1 Operation Level Value Dependency

In this subsection, we study the general case of well formed nested transactions. They are characterized by operation level acyclic value dependency relations. Let us first introduce the notion of operation dependency graphs for nested transactions.

Definition 3.1 (Operation Dependency Graph) *Let* $T = < O, P_1, P_2 >$ *be a nested transaction. The* operation dependency graph *of* T *is a graph* $ODG(T) = < V, E >$, *where* $V = O$ *is the set of all primitive operations of* T *and* E *is the set of all relations* (o_i, o_j) *such that either* $(o_i, o_j) \in P_1$ *when* o_i *and* o_j *belong to the same simple subtransaction or* $(o_i, o_j) \in P_2$ *when they belong to different simple subtransactions.*

The operation dependency graph of a nested transaction defines the requirements of execution order of its primitive operations. It consists of both the orders specified by its internal structure and the orders required by its value dependency relation.

Theorem 3.1 *A nested transaction* T *is well formed at the operation level if and only if its operation dependency graph* $ODG(T)$ *is acyclic.*

While the theorem is quite straightforward, let us sketch an informal proof as follows. Given a nested transaction T, if ODG(T) is acyclic, then it can be topologically sorted. The operations of T can be executed in the linear order specified by ODG(T) using the protocol given in section 2. On the other hand, suppose T can be executed properly using the protocol, the execution specifies a linear order of T's operations. Since the order is compatible with both value dependency relation P_2 and internal structure P_1, the operation dependency graph ODG(T) must be acyclic too.

Nested transactions that are well formed at the operation level make up the largest subclass of well formed nested transactions. This is because primitive operations are the smallest execution unit and are atomic at the system level in a database system. No linear execution order can be found for a nested transaction with a cyclic operation dependency graph.

It is worth noting that the bad-formedness of nested transactions is not intrinsic to their semantics. A badly formed nested transaction can be made well formed by rearranging the order of its operations without changing the semantics of the original transaction. One such rearrangement is to move all write operations to the end of their containing simple subtransactions. We shall come back to this issue later (see subsection 3.3).

To perform value dependency control at the operation level, the system should allow a kind of operation-to-operation communication. In other words, before each (write) operation is executed, the system must wait until

values from its antecedents (if any) arrive. Each time a (read) operation finishes execution, the result must be immediately sent to its dependents (if any). Generally, this implementation is very expensive because of the large number of messages. Let S be the average number of simple subtransactions of a nested transaction, O the average number of operations in a simple subtransaction that have remote dependents, and D_o the average number of remote dependents of such an operation. Then the average number of messages required to execute a nested transaction is $S * O * D_o$. Clearly, it is desirable to implement value dependency at the operation level only if $O * D_o$ is not very large.

Another problem with operation level value dependency control is recovery of failed subtransactions. Recall that a subtransaction may send its results to another subtransaction before it finishes all its operations. The failure of the antecedent subtransaction will cause its dependents to be rolled back.

3.2 Subtransaction Level Value Dependency

The smallest execution unit in a database system is a simple subtransaction. Since simple subtransactions bear many similarities to simple transactions, they are considered a good level at which to study value dependency. In this subsection, we focus on nested transactions that are well formed at the subtransaction level.

Definition 3.2 (Subtransaction Dependency Graph) *Let $T = < O, P_1, P_2 >$ be a nested transaction. The* subtransaction dependency graph *of T is a graph $SDG(T) = < V, E >$, where V is the set of all simple subtransactions of T and E is the set of all relations (o_i, o_j) such that there exist $o_1 \in o_i$ and $o_2 \in o_j$ such that $(o_1, o_2) \in P_2$.*

Unlike operation dependency graphs, a subtransaction dependency graph contains information about value dependency only. It does not contain the internal structure, which is embedded in the subtransactions. Like an operation dependency graph, a subtransaction dependency graph is useful when testing for well formedness of nested transactions at the subtransaction level.

Theorem 3.2 *A nested transaction T is well formed at the subtransaction level if and only if its subtransaction dependency graph $SDG(T)$ is acyclic.*

Subtransaction level value dependency allows a lower degree of intra transaction concurrency. The following is an example which is well formed at the operation level, but not at the subtransaction level.

Example 3.2 Consider a distributed database system consisting of two local databases, D_1 and D_2. Data items a and b are at D_1 and data items c and d are at D_2. Let G be a global transaction consisting of two subtransactions, G_1 and G_2, submitted to D_1 and D_2 respectively.

$$G_1 : r_1(a)w_1(b)$$
$$G_2 : w_2(c)r_2(d)$$

Suppose the remote value dependency of G is

$$D(r_1(a)) = \{w_2(c)\} \text{ and } D(r_2(d)) = \{w_1(b)\}$$

Or at the subtransaction level

$$D(G_1) = \{G_2\} \text{ and } D(G_2) = \{G_1\}$$

Clearly, G is well formed at the operation level, but not at the subtransaction level. □

Nested transactions which are well formed at the subtransaction level can be implemented more efficiently than those which are well formed at the operation level. A subtransaction will not be submitted to the scheduler until all its antecedents have completed. Once a subtransaction begins its execution, it can be executed to completion (as far as value dependency is concerned). The results are then sent to all its remote dependents.

The above execution has two advantages. First, the number of messages required in the execution is reduced because a subtransaction only needs to communicate (at most) once with another subtransaction. Let D_s be the average number of remote dependents of a subtransaction. Then the average number of messages required in the execution of a subtransaction is $S * D_s$. Clearly $D_s \leq O * D_o$. The second advantage is that there is no remote rollback within a nested transaction. A subtransaction will not send its results to remote dependents until it is done.

3.3 A Special Case: Two Phase Nested Transactions

We have discussed subclasses of nested transactions that are well formed at the operation and subtransaction levels, respectively. They both have advantages and disadvantages. In this subsection, we discuss another subclass of well formed nested transactions which has the advantages of both (previous) subclasses: a high degree of intra transaction concurrency and efficient implementation. The subclass is characterized by a special transaction structure which is defined as follows.

Definition 3.3 (Two phase transactions) *A (simple) transaction is two phase if all its write operations follow its read operations.*

Similarly, a nested transaction is two phase if all its simple subtransactions are two phase transactions.

The two phase model of nested transactions has significant effects on the behavior of value dependency.

Theorem 3.3 *A two phase nested transaction is always well formed at the operation level.*

Proof: Let T be a two phase nested transaction. Suppose that T is not well formed at the operation level. Then ODG(T) is cyclic. Recall that there are two types of arcs in a operation dependency graph: (1) the arcs between operations of different simple subtransactions, and (2) the arcs between operations of the same simple subtransaction. The first type of arcs are from read operations to write operations, while the second type of arcs are from write operations to read operations. Each cycle must contain arcs of both types. Since, in each simple subtransaction, write operations always follow read operations, arcs of the second type are impossible. Therefore, no cycle is possible in ODG(T). □

Given a two phase nested transaction, group the operations for each simple subtransaction into two sets: a *read set* consisting of all read operations and a *write set* consisting of all write operations.

Definition 3.4 (Semi Subtransaction Dependency Graph) *Let* $T = < O, P_1, P_2 >$ *be a nested transaction. The* semi subtransaction dependency

graph *of T is a graph $SSDG(T) =< V, E >$, where V is all the read and write sets of T as described above and E is the set of all relations (o_i, o_j) such that there exist $o_1 \in o_i$ and $o_2 \in o_j$ such that $(o_1, o_2) \in P_2$.*

The semi subtransaction dependency graph of a two phase nested transaction has a special structure. Each read set of T is a source of $SSDG(T)$, while each write set is a sink of $SSDG(T)$. There is no value dependency between read (or write) sets themselves. Therefore, the semi subtransaction dependency graphs are always acyclic. In other words,

Theorem 3.4 *A two phase nested transaction is always well formed at the semi subtransaction level.*

Due to the special structure of two phase nested transactions, the execution protocol given in section 2 can be simplified for value dependency control at the semi subtransaction level.

1. For each $o \in D(o_i)$ which is a write set do
   Wait until o is done and its results arrives
2. submit o_i to scheduler
3. For each $o \in D(o_i)$ which is a read set do
   send the result of o_i to o

It is not hard to see that value dependency control at the semi subtransaction level can be done as efficiently as at the subtransaction level.

4 Effects on Transaction Consistency

Another motivation for studying value dependency is its effects on transaction consistency. In this section, we study this problem. We first discuss the general effects of value dependency on transaction consistency. Then we give, as a special case, a sufficient condition for quasi serializability [DE89].

4.1 Transaction Consistency Problem

A basic requirement of transaction management is database consistency. A transaction (either simple or nested), when being executed alone, should transform a database from one consistent state to another. Similarly, when multiple transactions are executed concurrently, they should also transform a database from one consistent state to another.

One aspect of database consistency is transaction consistency. A database state is transaction consistent if it is obtained (1) from a transaction consistent state, and (2) through a transformation by an execution in which transactions interfere with each other properly. A transaction consistent database state should be a proper result of executed transactions. Transaction consistency can be understood and enforced by a set of constraints. For example, two transactions should not interfere with each other in any way (therefore, inconsistent retrieval is not allowed).

However, transaction consistency constraints are usually not explicitly given. One way of insuring transaction consistency is to execute transactions serializably. In other words, a database state is transaction consistent if it is obtained from a consistent state through a transformation by serializably executed transactions.

Value dependency has significant effects on transaction consistency in distributed database environments. In order to understand these effects, let us first study how transactions interfere with each other.

In a distributed database system, there are two kinds of transactions, local transactions that access only one element database and global transactions that access more than one element database. Transactions (both local and global) may interfere with each other either directly or indirectly (see e.g., [DELO89]. Transactions that access common data items interfere with each other directly by writing and reading data items. For example, a local transaction (or a global subtransaction) is influenced by another running at the same local site if the former reads the value of a data item last updated by the latter. Similarly, a global transaction may be influenced by another global transaction if they access common data items.

Transactions that do not access common data items might also interfere with each other indirectly through other transactions as the following example shows.

Example 4.1 Let us consider a database consisting of two data items, a and b. Let T_1, T_2, and T_3 be three transactions running at this database, and H be the history.

$T_1 : w_1(a)$
$T_2 : r_2(a)w_2(b)$
$T_3 : r_3(b)$
$H : w_1(a)r_2(a)w_2(b)r_3(b)$

T_1 directly influence T_2 through a and T_2, in turn, directly influence T_3. Since we assume (by default) that there is a value dependency between $r_2(a)$ and $w_2(b)$, T_3 is also indirectly influenced by T_1 through T_2. □

Indirect interference even exists between local transactions running at different local sites. In fact, remote local transactions may only interfere with each other indirectly.

Example 4.2 Consider a distributed database consisting of two databases D_1 and D_2. Data item a is at D_1 and data item b is at D_2. Let L_1 and L_2 be two local transactions running at D_1 and D_2, respectively, and G be a global transaction consisting of subtransaction G_1 and G_2.

$L_1 : w_1(a)$
$L_2 : r_2(b)$
$G_1 : r_g(a)$ and $G_2 : w_g(b)$

Let H_1 and H_2 be two local histories at D_1 and D_2, respectively.

$H_1 : w_1(a)r_g(a)$
$H_2 : w_g(b)r_2(b)$

Suppose the remote value dependency of G is $D(G_1) = G_2$, then L_1 indirectly influences L_2 through G. □

From the above examples, we have seen that value dependency plays a key role in indirect interference between transactions. The only way that a transaction might indirectly influence another transaction is through a third transaction. It is the value dependency in the third transaction that causes the interference.

One way to understand the effects of value dependency is to study the way that restrictions on value dependency simplify the transaction consistency problem. Let us consider example 4.2 again. Local transaction L_1 influences L_2 which is running at a different site. The global history

$H = \{H_1, H_2\}$ is correct only if

1. both local histories preserve local transaction consistency of their respective host sites, and
2. global history preserves the global transaction consistency of the distributed database.

If, however, no remote value dependency is allowed within global transactions, local transactions at different local sites run independently and no indirect interference is possible. Global transaction consistency is therefore automatically preserved because there is only one global transaction[2] In other words, the transaction consistency problem has been simplified by turning it into two local transaction consistency problems that are much easier to understand.

4.2 Quasi Serializability

In the last subsection, we have seen that restrictions on value dependency are very useful. In certain database environments, it is reasonable to make these restrictions. For example, in heterogeneous distributed database environments, remote value dependency is very unlikely due to the local autonomy. One way of making use of this knowledge is to develop new correctness criteria allowing those executions that are semantically correct but not serializable. Based on this observation, we introduced quasi serializability [DE89].

In this subsection, we shall focus on how the knowledge of restrictions on value dependency can be used in quasi serializability theory. We do so by giving a sufficient condition of remote value dependency for quasi serializability.

4.2.1 Background

A heterogeneous distributed database system, HDDBS, is a system integrating pre-existing database systems to support global applications accessing more than one database. Formally, an HDDBS consists of a set

[2] Generally, however, there might be more than one global transaction.

$D = \{D_1, D_2, ..., D_m\}$ of local databases systems (LDBSs), a set $G = \{G_1, G_2, ..., G_n\}$ of global transactions, and a set $L = \bigcup_{l=1}^{m} L_l$ of local transactions, where $L_l = \{L_{l,1}, L_{l,2}, ..., L_{l,j_l}\}$. A local transaction is a transaction that accesses only one LDBS, while a global transaction accesses more than one LDBS. A global history H over $G \cup L$ in an HDDBS is a set of local histories $H = \{H_1, H_2, ..., H_m\}$, where the local history H_l (at LDBS D_l) is defined over global subtransactions $G_{1,l}, G_{2,l}, ..., G_{n,l}$ and local transactions $L_{l,1}, L_{l,2}, ..., L_{l,j_l}$.

The basic idea of quasi serializability is that, in order to preserve the global database consistency, global transactions should be executed in a serializable way, with proper consideration of the effects of local transactions. Formally, quasi serializable histories are defined as follows.

Definition 4.1 *A global history is quasi serial if*

1. *all local histories are (conflict) serializable; and*
2. *there exists a total ordering of all global transactions such that, for every two global transactions G_i and G_j, if $G_i \rightarrow G_j$ then all G_i's operations precede G_j's operations in all local histories in which they both appear.*

Definition 4.2 *A history is quasi serializable if it is (conflict) equivalent to a quasi serial history.*

4.2.2 A Sufficient Condition

In this subsection, we try to develop a sufficient condition of value dependency for quasi serializability. The condition is sufficient in the sense that a quasi serializable history will preserve transaction consistency if all the involved transactions meet this requirement.

Let us first introduce the notion of site dependency graph for a global history.

Definition 4.3 (Site Dependency Graph) *Let H be a global history of an HDDBS. Suppose that $G_1, G_2, ..., G_n$ are the only committed global transactions in H, then the* site dependency graph *of H is $SG(H) = < V, E >$,*

where V is the set of all sites of the HDDBS and E is the set of all relations (s_i, s_j) *such that there exists a global transaction* G_k *such that* $(G_{k,i}, G_{k,j}) \in SDG(G_k)$*.*

Since a global transaction can have at most one subtransaction per site in an HDDBS [GP86], the site dependency graph of a global history is actually the sum of subtransaction dependency graphs of all global transactions in the history.

The acyclicity of the site dependency graph is sufficient to guarantee the transaction consistency of heterogeneous distributed database systems. Given a history H, let us consider any pair of transactions T_i and T_j in H. There are three cases.

case 1: Transactions T_i and T_j are local to the same site. T_i and T_j can be either local transactions or global subtransactions. They interfere with each other properly because of the serializability of local histories.

case 2: Both T_i and T_j are global transactions. Since H is equivalent to a quasi serial history in which global transactions are executed sequentially and therefore interfere properly. T_i and T_j also interfere with each other properly in H.

case 3: Transaction T_i is local to a site, D_i, while T_j is local to another site, D_j. Again, T_i and T_j can be either local transactions or global subtransactions. Since SG(H) is acyclic, it can be topologically sorted. Suppose $D_i \rightarrow D_j$ in the order, then T_i might influence T_j because there might be value dependency from subtransactions at D_i to subtransactions at D_j. The converse, however, does not hold.

Cases 1 and 3 imply that local transactions and global subtransactions at the local level interfere with each other in the order compatible with local serialization order and SG(H). The acyclicity of SG(H) guarantees that the interference is proper (i.e., equivalent to that of some serial execution). At the global level, global transactions themselves interfere properly in the order specified by a equivalent quasi serial history, as case 2 explained. Therefore, we have,

Theorem 4.1 *A quasi serializable history H preserves transaction consistency of an HDDBS if SG(H) is acyclic.*

The following example illustrates why the acyclicity of the site dependency graph is important.

Example 4.3 Consider an HDDBS consisting of two LDBSs D_1 and D_2. Data items a and b are at D_1 and data items c and d are at D_2. Let L_1 and L_2 be two local transactions running at D_1 and D_2 respectively.

$L_1 : w_{l_1}(a)r_{l_1}(b)$
$L_2 : r_{l_2}(c)w_{l_2}(d)$

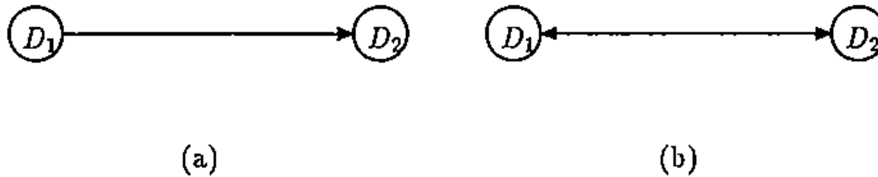
Let G_1 and G_2 be two global transactions.

$G_1 : r_{g_1}(a)w_{g_1}(c)$
$G_2 : r_{g_2}(d)w_{g_2}(b)$

Let H_1 and H_2 be two local histories at D_1 and D_2 respectively.

$H_1 : w_{l_1}(a)r_{g_1}(a)w_{g_2}(b)r_{l_1}(b)$
$H_2 : w_{g_1}(c)r_{l_2}(c)w_{l_2}(d)r_{g_2}(d)$

Suppose that the value dependency relation of G_1 is $D(G_{1,1}) = \{G_{1,2}\}$, and that there is no value dependency in G_2. Then the site dependency graph is acyclic (see figure 4.1.a). The global transaction preserves transaction consistency because L_1 will not be influenced by L_2. If, on the other hand, the value dependency relation of G_2 is $D(G_{2,2}) = \{G_{2,1}\}$, then the site dependency graph is cyclic (see figure 4.1.b). The global history will not preserve transaction consistency because L_1 and L_2 each interfere with each other. □



(a) (b)

Figure 1: Site dependency graphs of H

5 Conclusions

In this paper, we have studied a fundamental issue in nested transactions, that of value dependency. Specifically, we have discussed the effects of value dependency on executions of nested transactions (i.e., value dependency control) and effects on transaction consistency of database systems.

To guarantee the correct execution of nested transactions, we must ensure that they are not only executed properly, but also that they are well formed with respect to value dependency. We have studied various subclasses of well formed nested transactions, as well as related issues. It turns out that certain restrictions on the transaction model (e.g., two phase transaction model) are very helpful in the execution of nested transactions with various levels of value dependency.

Value dependency is a part of transaction semantics. Therefore, we usually assume that no knowledge of value dependency is available (e.g., in serializability theory). However, certain knowledge of and/or restrictions on value dependency are both possible and necessary. Also discussed in this paper is a way to make use of value dependency in heterogeneous distributed database systems to validate the appropriateness of a new correctness criterion, namely quasi serializability. Effects of value dependency on transaction consistency in other database environments are not studied in this paper.

The concept of value dependency is simple, but important. It has effects on many execution issues of nested transactions. We are unable to study all of them (e.g., concurrency control) in this paper due to the space limitation. For the same reason, we are also unable to go into detail about some of the issues we have discussed. For example, we have not discussed how to perform value dependency control efficiently, nor have we given any efficient value dependency control algorithms. Future work is still needed in this area.

Acknowledgements

The authors wish to thank Shawn Ostermann for his many helpful suggestions while this paper was in preparation.

References

[BBG89] C. Beeri, P. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[DE89] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in interbase. In *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[DELO89] W. Du, A. Elmagarmid, Y. Leu, and S. Ostermann. Effects of autonomy on global concurrency control in heterogeneous distributed database systems. In *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, Gaithersburg, MD, October 1989.

[GP86] V. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Informtion Systems*, 11(4):287–297, 1986.

[Gra81] J. Gray. The transaction concept: virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, Cannes, France, September 1981.

[Mos85] E. Moss. *Nested Transactions, an Approach to Reliable Distributed Computing.* The MIT Press, 1985.