1990

Maintaining Quasi Serializability in HDDBSs

Weimin Du

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

Won Kim

Report Number:
90-971

MAINTAINING QUASI SERIALIZABILITY IN HDDBSs

Weimin Du
Ahmed Elmagarmid
Won Kim

CSD-TR 971
March 1990
(Revised June 1990)

Maintaining Quasi Serializability in HDDBSs*

Weimin Du, Ahmed Elmagarmid and Won Kim†
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907
{du,ake}@cs.purdue.edu

† MCC
3500 W. Balcones Center Dr.
Austin, TX 78759
kim@mcc.com

Abstract

We present, in this paper, a scheduler that generates quasi serializable executions. Quasi serializability is a new correctness criterion for concurrency control in heterogeneous distributed database systems (HDDBSs). The scheduler differs from the previous ones in that it controls the submission and interleaving of global transactions only. Therefore, it does not violate local autonomy. In addition, it provides a high degree of concurrency and is global deadlock free. All the features make it very attractive for HDDBSs.

1 Introduction

A heterogeneous distributed database system (HDDBS) is a federation of pre-existing databases (called local database systems, or LDBSs). LDBSs are integrated to support global applications accessing data at more than one LDBS.

An important issue in supporting global applications is controlling concurrent execution of transactions. The traditional criterion for concurrency control is serializability. Serializability in HDDBSs represents the strongest type of consistency because it treats an HDDBS as a set of

*This work is supported by a PYI Award from NSF under grant IRI-8857952, a David Ross Fellowship from Purdue Research Foundation, and grants from AT&T Foundation, Mobil Oil, SERC and Tektronix.

strongly coupled LDBSs. For example, there is no distinction between local and global transactions. The problem with serializability is that it is very hard to maintain in HDDBSs because LDBSs are autonomous. For example, it is very difficult (if not impossible) to detect at the global level conflicts between global transactions. It is also very difficult to resolve the conflicts [DELO89].

In this paper, we present a scheduler based on quasi serializability, a new correctness criterion for concurrency control in HDDBSs [DE89]. Quasi serializability represents a weaker type of HDDBS consistency because it is based primarily on the behaviour of global transactions. Quasi serializability is meaningful for HDDBSs because it can be effectively maintained at the global level without violating local autonomy. The aspects of HDDBS consistency that are maintained by serializable executions but not quasi serializable executions (e.g., interference between local transactions at different sites) can be easily maintained explicitly at the global level [DE90]. The scheduler guarantees quasi serializability of executions by controlling the submission and interleaving of global transactions only. Therefore, it does not violate local autonomy.

The rest of the paper is organized as follows. In Section 2, we review the basic quasi serializability theory. Then, we discuss, in Section 3, the feature of quasi serializability that makes it maintainable at the global level without violating local autonomy. The scheduler and its correctness proof are given in Section 4. A detailed discussion of various features of the scheduler is given in Section 5. Finally, some concluding remarks are given in Section 6.

2 Quasi Serializability

2.1 Notations

An HDDBS consists of a set $\mathcal{D}$[1] of data items and a set $\mathcal{T}$ of transactions. The data item set $\mathcal{D}$ consists of n subsets, $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n$, 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 $\mathcal{T}$ consists of $n+1$ subsets, $\mathcal{G}, \mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n$, where $\mathcal{L}_i$ is a set of local transactions that access $\mathcal{D}_i$ only, while $\mathcal{G}$ 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 $\mathcal{D}_j$ only. The set of all operations in a transaction T (either a local or a global transaction) is called operation set and denoted $\mathcal{O}(T)$. The data item set $\mathcal{D}_i$, together with the transaction set $\mathcal{T}_i = \mathcal{L}_i \cup \mathcal{G}_i$ where $\mathcal{G}_i = \{G_{j,i} \mid G_j \in \mathcal{G}\}$, forms the local database system $LDBS_i$.

[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.

A local execution E_l in $LDBS_l$ is an interleaved sequence of operations of transactions in $\mathcal{T}_l$. Operations of the same transaction are executed in the same order in E_l as they appear in the transaction. A global execution E in an HDDBS consists of a set of local executions, $E = \{E_1, E_2, ... E_n\}$, where E_l is the local execution at $LDBS_l$. The execution order of local execution E_l and global execution E are denoted $\prec_{E_l}$ and $\prec_E$, respectively.

2.2 Quasi Serializable Executions

Quasi serializability was introduced in [DE89] as a correctness criterion for concurrency control in HDDBSs. The basic idea of quasi serializability is to focus on those aspects of an execution that are the global transaction manager (GTM) is able to control (i.e., the behaviour of and interaction among global transactions). It is local transaction managers' responsibility to guarantee the consistency of local databases. Interaction between local transactions at different sites is managed explicitly by controlling information flow in a global transaction [DE90].

Definition 2.1 (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 $\mathcal{G}$ such that $\forall G_i, G_j \in \mathcal{G}$ and G_i preceding G_j in the ordering, $o_i \prec_{E_l} o_j$ in E_l for all $o_i \in \mathcal{O}(G_i)$ and $o_j \in \mathcal{O}(G_j)$ $(1 \leq l \leq n)$.*

Definition 2.2 (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 2.1 *Consider an HDDBS consisting of two LDBSs, where $a, b \in \mathcal{D}_1$ and $c, d \in \mathcal{D}_2$. The following global transactions are submitted to the HDDBS:*

$G_1 = \{G_{1,1}, G_{1,2}\}$, *where* $G_{1,1} : w_{g_1}(a)$ *and* $G_{1,2} : r_{g_1}(c)$

$G_2 = \{G_{2,1}, G_{2,2}\}$, *where* $G_{2,1} : r_{g_2}(b)$ *and* $G_{2,2} : w_{g_2}(d)$

Let L_1 and L_2 be two local transactions submitted to $LDBS_1$ and $LDBS_2$, respectively:

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

Let E_1 and E_2 be the local executions at $LDBS_1$ and $LDBS_2$, respectively:

$E_1 : w_{g_1}(a)r_{l_1}(a)w_{l_1}(b)r_{g_2}(b)$

$E_2 : w_{g_2}(d)w_{l_2}(c)r_{l_2}(d)r_{g_1}(c)$

Then $E = \{E_1, E_2\}$ is quasi serializable (but not serializable). It is equivalent to the quasi serial execution $E' = \{E_1', E_2'\}$, where

$E_1' : w_{g_1}(a)r_{l_1}(a)w_{l_1}(b)r_{g_2}(b)$

$E_2' : w_{l_2}(c)r_{g_1}(c)w_{g_2}(d)r_{l_2}(d)$

The main difference between serializability and quasi serializability is that quasi serialization order of global transactions in the latter is compatible with their execution order. For example, G_1 precedes G_2 in the order because it executes before G_2 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 important in concurrency control. It allows the GTM to enforce specific orders by just controlling the submission and interleaving of global transactions, as we will see in Section 4.

2.3 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, ..., $o_k \in \mathcal{O}(E_l) = \bigcup_{T \in T_l} \mathcal{O}(T)$ $(k \geq 0)$ 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.

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 indirectly conflicts with G_j.

Theorem 2.1 (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 [DE89] □

3 Maintaining Quasi Serializability

Theorem 2.1 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 GTM to predict or detect indirect conflicts between global operations because they may be introduced by local operations [DELO89]. 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 section, we study the feature of quasi serializable executions. We first introduce the notion of access graphs of global transactions and global executions. Then, we present a sufficient condition on access graphs for quasi serializable executions. The scheduler based on this condition will be described in the next section.

3.1 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 next subsection, is very useful in determining its quasi serializability.

Definition 3.1 (Access graphs of global transactions) *The access graph of a global transaction G_i is an undirected graph $AG =< \mathcal{V}, \mathcal{A} >$, where $\mathcal{V} = \{\mathcal{D}_{i_1}, \mathcal{D}_{i_2}, ..., \mathcal{D}_{i_k}\}$ is the set of all local databases G_i accesses ($i_1 < i_2 < ... < i_k$ and $k \geq 1$), and $\mathcal{A} = \{(\mathcal{D}_{i_j}, \mathcal{D}_{i_{j+1}}) \mid 1 \leq j < k\}$*[2].

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

We say that a transaction T_i *directly interleaves* with another transaction T_j in an execution E if their operations are interleaved 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 $T_1, T_2, ..., T_k$ ($k \geq 0$) 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 simply say that two transactions interleave with each other meaning that they either directly or indirectly interleave with each other. We use $\mathcal{I}(T, E)$ to denote the set of all transactions that interleave with T in E.

Definition 3.2 (Access graphs of global executions) *The access graph of a global execution E with respect to global transaction G_0 is $AG(E, G_0) = \cup_{G \in \mathcal{I}(G_0, E)} AG(G)$.*

[2] Actually, $\mathcal{A}$ could be any set of arcs that connect $\mathcal{D}$s in $\mathcal{V}$ in a linear order. The increasing subscript ordering is chosen in the definition for the sake of simplicity.

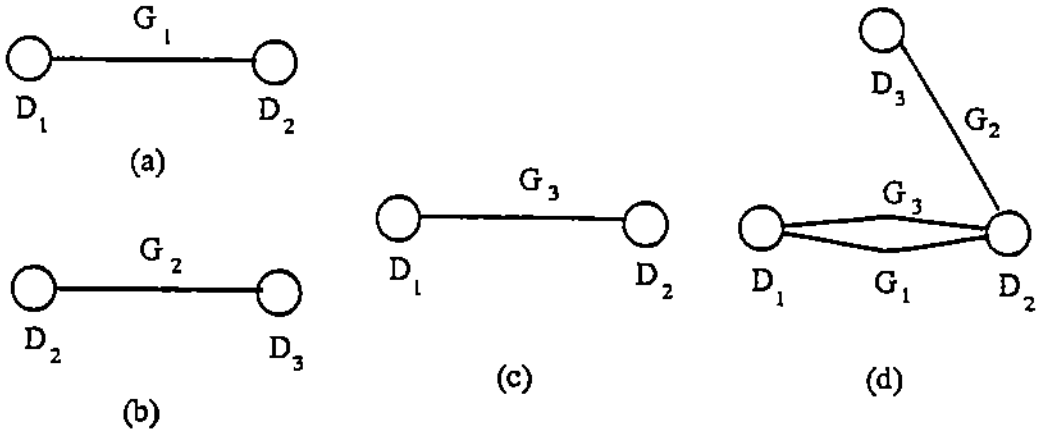

Figure 1: Access graphs of G_1, G_2, G_3 and E

The concept of access graph is similar to that of site graph in [BS88b]. The difference is that the access graph of an execution concerns only those global transactions that interleave with each other.

Example 3.1 *Consider an HDDBS consisting of three LDBSs, where* $a, b \in \mathcal{D}_1$, $c, d, e \in \mathcal{D}_2$ *and* $f \in \mathcal{D}_3$. *Let* G_1, G_2, G_3 *be global transactions submitted to the HDDBS:*

$G_1 = \{G_{1,1}, G_{1,2}\}$, *where* $G_{1,1} : w_{g_1}(a)$ *and* $G_{1,2} : r_{g_1}(d)$

$G_2 = \{G_{2,2}, G_{2,3}\}$, *where* $G_{2,2} : w_{g_2}(c) r_{g_2}(e)$ *and* $G_{2,3} : r_{g_2}(f)$

$G_3 = \{G_{3,1}, G_{3,2}\}$, *where* $G_{3,1} : r_{g_3}(b)$ *and* $G_{3,2} : w_{g_3}(e)$

Let L_1 *and* L_2 *be two local transactions submitted to* $LDBS_1$ *and* $LDBS_2$, *respectively:*

$L_1 : r_{l_1}(a) w_{l_1}(b)$

$L_2 : r_{l_2}(c) w_{l_2}(d)$

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*

$E_1 : w_{g_1}(a) r_{l_1}(a) w_{l_1}(b) r_{g_3}(b)$

$E_2 : w_{g_2}(c) r_{l_2}(c) w_{l_2}(d) r_{g_1}(d) w_{g_3}(e) r_{g_2}(e)$

$E_3 : r_{g_2}(f)$

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 1.(a), (b), (c) and (d), respectively.*

3.2 A Sufficient Condition

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 3.1 will not effect the quasi serializability of the execution because they access only one common local database.

The following theorem gives a sufficient condition for quasi serializable executions.

Theorem 3.1 *A global execution E is quasi serializable if $AG(E,G)$ is acyclic for all $G \in \mathcal{G}$.*

Example 3.2 *In Example 3.1, 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 $G \in \mathcal{G}$. We prove 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 $G_{i_1} \rightarrow G_{i_2} \rightarrow ... \rightarrow G_{i_k} \rightarrow G_{i_1}$. The proof consists of the following two parts.

(1) **$G_{i_1}, G_{i_2}, ..., G_{i_k}$ 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 $G_{i_k} \rightarrow G_{i_1}$.

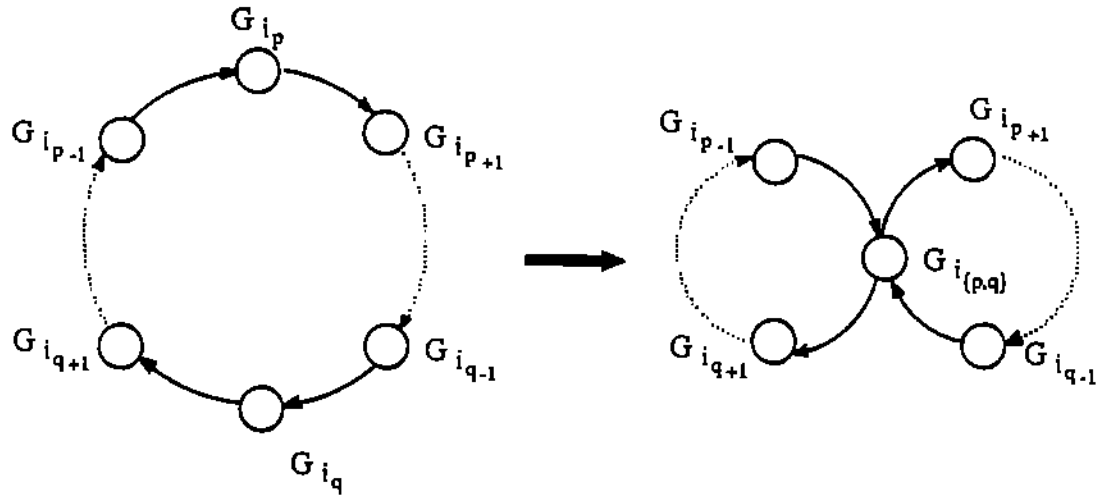

Figure 2: Transformation of cycles in QSG(E)

Let us transform the cycle $G_{i_1} \rightarrow G_{i_2} ... G_{i_k} \rightarrow G_{i_1}$ to two smaller ones by combining G_{i_p} and G_{i_q} together into one global transaction $G_{i_{(p,q)}}$, as show in Figure 2. 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: $G_{i_{(p,q)}} \rightarrow G_{i_{q+1}} \rightarrow ... \rightarrow G_{i_{p-1}} \rightarrow G_{i_{(p,q)}}$ and $G_{i_{(p,q)}} \rightarrow G_{i_{p+1}} \rightarrow ... \rightarrow G_{i_{q-1}} \rightarrow G_{i_{(p,q)}}$. In each cycle, there are less than k global transactions. According to the induction hypothesis, they interleave with each other. In specific, all transactions $(G_{i_1}, G_{i_2}, ..., G_{i_k})$ interleave with $G_{i_{(p,q)}}$ (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 $G = G_{i_1}, G_{i_2}, ..., G_{i_k}$**

Since G_{i_1} conflicts with G_{i_2}, they must access a common local database $\mathcal{D}_{i_1}$. Similarly, G_{i_j} and $G_{i_{j+1}}$ access a common local database $\mathcal{D}_{i_j}$ for $j = 2,3,..,k$ and $G_{i_{k+1}} = G_{i_1}$. According to Definition 3.1, there is a path from $\mathcal{D}_{i_1}$ to $\mathcal{D}_{i_2}$ in AG(G_{i_1}). Similarly, there is a path from $\mathcal{D}_{i_j}$ to $\mathcal{D}_{i_{j+1}}$ in AG(G_{i_j}) for $j = 2,3,...,k$ and $\mathcal{D}_{i_{k+1}} = \mathcal{D}_{i_1}$. In other words, there is a cycle $\mathcal{D}_{i_1} \rightarrow \mathcal{D}_{i_2} \rightarrow ... \rightarrow \mathcal{D}_{i_k} \rightarrow \mathcal{D}_{i_1}$ in AG(E, G_1) (as well as in AG(E, G_2), ..., AG(E, G_k)). A contradiction! □

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

Example 3.3 *Consider execution E' in Example 2.1. 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.* □

4 A Scheduler

According to Theorem 3.1, the quasi serializability of a global execution is assured if global transactions are submitted in such a way that global transactions whose access graphs form a cyclic graph do not interleave with each other. For example, G_1 and G_2 in Example 2.1 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.

In this section, we present such a scheduler which guarantees quasi serializability of executions by controlling submission of global transactions.

4.1 The Algorithm

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(G_i);
begin
    xact_access_graph ← ACCESS_GRAPH(G_i); (1)

```
    if exec_access_graph + xact_access_graph is cyclic                    (2)
      then delayed_xact ← delayed_xact ∪ {G_i};                           (3)
      else active_xact ← active_xact ∪ {G_i};                             (4)
           exec_access_graph ← exec_access_graph + xact_access_graph;     (5)
    endif
end;

procedure TRANSACTION_TERMINATION(G_i);
begin
    active_xact ← active_xact - {G_i};                                    (1)
    if active_xact = ∅                                                    (2)
      then exec_access_graph ← ∅;                                         (3)
           for G_j ∈ delayed_xact do                                      (4)
               TRANSACTION_SUBMISSION(G_j);                               (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 4.1 *Consider global transactions G_1, G_2 and G_3 in Example 3.1. Suppose that their operations are submitted in the order shown in E. Since G_1 and G_2 access only one common database ($LDBS_2$), the union of their access graphs is acyclic. Therefore, operations $w_{g_1}(a), w_{g_2}(c), r_{g_2}(f)$ and $r_{g_1}(d)$ 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*

$E'_1 : w_{g_1}(a) r_{l_1}(a) w_{l_1}(b) r_{g_3}(b)$

$E'_2 : w_{g_2}(c) r_{l_2}(c) w_{l_2}(d) r_{g_1}(d) r_{g_2}(e) w_{g_3}(e)$

$E'_3 : r_{g_2}(f)$

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'.

4.2 Correctness Proof

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. The argument is formalized in the following theorem.

Theorem 4.1 *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 $\mathcal{G}(E_i)$ be the set of global transactions involved in E_i $(i = 1, 2, ..., k)$. Then $\mathcal{G} = \cup_{i=1}^{k} \mathcal{G}(E_i)$.

Let us consider E_i $(1 \leq i \leq k)$. All global transactions in $\mathcal{G}(E_i)$ 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 3.1, 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. □

We have shown in [DE90] [ED90a] that a quasi serializable execution maintains the HDDBS consistency to a satisfactory degree. More specifically, it preserves all local integrity constraints, as well as those global constraints that involve data updatable by global transactions only. In addition, all global transactions, as well as local transactions at a site interfere with each other in a partial order in a quasi serializable execution. Although local transactions at different sites may interfere with each other mutually, such an undesirable interference can be easily prevented at the global level by controlling the information flow in a global transaction (see [DE90]).

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

Example 4.2 *Consider an HDDBS consisting of two LDBSs, where $x \in \mathcal{D}_1$ and $y \in \mathcal{D}_2$. The following global transactions are submitted to the HDDBS:*

$G_1 = \{G_{1,1}, G_{1,2}\}$, *where* $G_{1,1} : w_{g_1}(x)$ *and* $G_{1,2} : w_{g_1}(y)$

$G_2 = \{G_{2,1}, G_{2,2}\}$, *where* $G_{2,1} : r_{g_2}(x)$ *and* $G_{2,2} : r_{g_2}(y)$

Let L_1 be a local transaction submitted to $LDBS_1$.

$L_1 : r_{l_1}(x) w_{l_1}(x)$

Let E_1 and E_2 be the local executions at $LDBS_1$ and $LDBS_2$, respectively:

$E_1 : r_{l_1}(x) w_{g_1}(x) r_{g_2}(x) w_{l_1}(x)$

$E_2 : w_{g_1}(y) r_{g_2}(y)$

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 Discussion

In this section, we discuss features of the proposed scheduler with respect to various HDDBS issues. We also briefly compare our results with other related work.

Local Autonomy

Preservation of local autonomy is one of the main objectives of schedulers in HDDBSs. Informally, local autonomy defines the right of an LDBS to control the access to its data by other LDBSs and the right to access and manipulate its data independently [DEK90]. Local autonomy makes scheduling very difficult. Most other schedulers in the literature violate local autonomy by either requiring LDBSs to provide information and/or making assumptions about LDBSs. For example, the scheduler proposed in [BS88a] assumes that each LDBS uses two-phase locking protocol for concurrency control, while the scheduler in [Pu88] requires LDBSs to report serialization order of global transactions to the GTM.

The scheduler we proposed in this paper requires no information from and makes no assumption about LDBSs (except serializability of local executions). Therefore, LDBSs may use any concurrency control protocols (e.g., two-phase locking, timestamp ordering, etc.). They may also schedule local transactions and global subtransactions in any way they want, as long as the execution is serializable, and have no obligation to communicate with the GTM regarding the global concurrency control.

Global Deadlock

The basic idea of our scheduler is to control the submission of global transactions so that undesirable situations will not occur (see [DELO89]). Therefore, like the scheduler in [ED90b],

our scheduler is global deadlock-free, as the following theorem shows.

Theorem 5.1 *Global transactions do not deadlock in executions produced by the scheduler.*

Proof: Let E be a global execution produced by the scheduler. Suppose that there is a deadlock in E which involves global transactions $G_{i_1}, G_{i_2}, ..., G_{i_p}$. Without loss of generality, let assume that $G_{i_1} \rightarrow G_{i_2} \rightarrow ... G_{i_p} \rightarrow G_{i_1}$ be a cycle in the wait-for graph of E. Let t be the time when the deadlock occurs. Then $G_{i_1}, G_{i_2}, ..., G_{i_p}$ are all active at time t. Since G_{i_1} is waiting for G_{i_2}, they must access a common local database. Let it be $\mathcal{D}j_1$. Similarly, G_{i_2} and G_{i_3} access $\mathcal{D}j_2$, ... G_{i_p} and G_{i_1} access $\mathcal{D}j_p$. Then, there is a path from $\mathcal{D}j_p$ to $\mathcal{D}j_1$ in AG(G_{i_1}), a path from $\mathcal{D}j_1$ to $\mathcal{D}j_2$ in AG(G_{i_2}), ..., and a path from $\mathcal{D}j_{p-1}$ to $\mathcal{D}j_p$ in AG(G_{i_p}). In other words, there is a cycle in AG(E, G_{i_1}) $\cup ... \cup$ AG(E, G_{i_p}), $\mathcal{D}j_1 \rightarrow \mathcal{D}j_2 \rightarrow ... \mathcal{D}j_p \rightarrow \mathcal{D}j_1$. This is, however, impossible because the global transactions are submitted by the scheduler concurrently. □

However, it is possible that a global transaction gets into a livelock. Generally, the more sites a global transaction accesses, the more likely it will be delayed. A global transaction accessing all local sites is always delayed unless it is the only active global transaction in the phase. The problem can be solved using ordinary techniques, e.g., always try the oldest transaction first.

Concurrency

Schedulers based on quasi serializability provide a higher degree of concurrency than those based on serializability because they allow both serializable and non-serializable executions. Given the same information about local transactions and executions, schedules produced by schedulers based on serializability can also be produced by some schedulers based on quasi serializability, but not vice verse (e.g., E in Example 2.1). The main difference between schedulers based on serializability and quasi serializability is, however, that the latter works even if no information about local executions is available, while the former do not[3].

Two schedulers based on quasi serializability have been proposed [AGMS87][4] [ED90b]. They both allow a very low degree of concurrency among global transactions. Global transactions are basically submitted and executed sequentially. The scheduler we proposed in this paper is better in that it allows concurrent execution of some global transactions. Global transactions may interleave with local transactions in any way allowed by local schedulers. Global transactions may

[3]The scheduler in [BS88b] is an exception. It is however not realistic because it requires acyclicity of the site graph of the whole execution. If, for example, a global transaction accessed all local sites, then no other global transaction is allowed by the scheduler.

[4]The scheduler was designed to maintain serializability. However, as shown in [DE89], it maintains quasi serializability instead.

also interleave with each other in a limited way (i.e., the access graph of the global execution is acyclic).

In the following theorem, we show that our scheduler provides the highest degree of concurrency among global transactions if no information about local transactions and executions is available.

Theorem 5.2 *Given a set of global transactions $\mathcal{G}$ such that $\cup_{G \in \mathcal{G}} AG(G)$ is cyclic. There exists an execution E of $\mathcal{G} \cup \mathcal{L}$ where $\mathcal{L}$ is a set of local transactions such that E is not quasi serializable.*

We need the following lemma to prove the theorem.

Lemma 5.1 *Given two global transactions G_1 and G_2 accessing a common local database $\mathcal{D}_l$. There exists a local execution E_l such that G_1 indirectly conflicts with G_2 in E_l.*

Proof: Let $o_1 \in \mathcal{O}(G_1)$ and $o_2 \in \mathcal{O}(G_2)$ access data $a, b \in \mathcal{D}_l$, respectively. They directly conflict with each other if $a = b$ and one of them is a write operation. Otherwise, we have the following five cases:

- $o_1 = r_1(a), o_2 = r_2(b)$ **and** $a = b$: o_1 indirectly conflicts with o_2 in local execution $E_l : r_1(a) w_{l_1}(a) r_2(b)$, where $w_{l_1}(a)$ is a local operation.
- $o_1 = r_1(a), o_2 = r_2(b)$ **and** $a \neq b$: o_1 indirectly conflicts with o_2 in local execution $E_l : r_1(a) w_{l_1}(a)\ r_{l_2}(a) w_{l_2}(b) r_2(b)$, where $w_{l_1}(a), r_{l_2}(a)$ and $w_{l_2}(b)$ are local operations.
- $o_1 = w_1(a), o_2 = r_2(b)$ **and** $a \neq b$: o_1 indirectly conflicts with o_2 in local execution $E_l : w_1(a) r_{l_2}(a)\ w_{l_2}(b) r_2(b)$, where $r_{l_2}(a)$ and $w_{l_2}(b)$ are local operations.
- $o_1 = w_1(a), o_2 = w_2(b)$ **and** $a \neq b$: o_1 indirectly conflicts with o_2 in local execution $E_l : w_1(a) r_{l_1}(a)\ w_{l_1}(b) r_{l_2}(b) w_2(b)$, where $r_{l_1}(a), w_{l_1}(b)$ and $r_{l_2}(b)$ are local operations.
- $o_1 = r_1(a), o_2 = w_2(b)$ **and** $a \neq b$: o_1 indirectly conflicts with o_2 in local execution $E_l : r_1(a) w_{l_1}(a)\ r_{l_2}(a) w_{l_2}(b) w_2(b)$, where $w_{l_1}(a), r_{l_2}(a)$ and $w_{l_2}(b)$ are local operations. □

Proof of the theorem: Let $\mathcal{G} = \{G_1, G_2, ..., G_k\}$. Without loss of generality, let us assume that G_1, G_2 access local database $\mathcal{D}_{i_1}$, G_2, G_3 access $\mathcal{D}_{i_2}$, ..., and G_k, G_1 access $\mathcal{D}_{i_k}$. According to the lemma, there exist a local execution E_{i_1} in which G_1 indirectly conflicts with G_2, a local execution E_{i_2} in which G_2 indirectly conflicts with G_3, ..., and a local execution E_{i_k} in which G_k indirectly conflicts with G_1. Then there is a cycle in QSG(E): $G_1 \rightarrow G_2 \rightarrow ... \rightarrow G_k \rightarrow G_1$. According to Theorem 2.1, E is not quasi serializable. □

Transaction Abortion

Another advantage of the scheduler we proposed in Section 4 is that it aborts no global transactions for inconsistency among local executions. The reason is that the global transactions that may cause inconsistency are always submitted and executed sequentially. Transactions that are submitted concurrently never cause inconsistency because of acyclicity of the access graph.

6 Conclusion

In this paper, we presented a scheduler producing quasi serializable executions. The scheduler is based on the following property of quasi serializable executions. The quasi serialization order of two global transactions is compatible with their submission order if (1) they do not directly interleave with each other; and (2) no other global transaction interleaves with both of them. The scheduler guarantees quasi serializability of executions by controlling the submission and interleaving of global transactions. It is attractive for the following reasons.

- It does not violate local autonomy;
- It is global deadlock-free;
- It aborts few global transactions; and
- It allows concurrent execution of global transactions.

The scheduler also differs from others (based on serializability theory) in that it alone does not maintain the HDDBS consistency. The aspects of HDDBS consistency that are not guaranteed by quasi serializable executions (e.g., interference between local transactions at different sites) are difficult to maintain by simply scheduling transactions at the global level. However, they can be easily maintained by controlling the information flow in a global transaction. Thus, the scheduler, in conjunction with the controller for information flow, maintains HDDBS consistency.

Acknowledgements

The authors are very grateful to R. Martin, the research coordinator of the Software Engineering Research Center at Purdue University, for many helpful comments and proofreading.

References

[AGMS87] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. In *IEEE Data Engineering Bulletin*, pages 5–11, September 1987.

[BS88a] Y. Breitbart and A. Silberschatz. Multibdatabase systems with a decentralized concurrency control scheme. *Distributed Processing Technical Committee Newsletter*, 10(2):35–41, November 1988.

[BS88b] Y. Breitbart and A. Silberschatz. Multibdatabase update issues. In *Proceedings of the International Conference on Management of Data*, pages 135–142, June 1988.

[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*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[DE90] W. Du and A. Elmagarmid. Maintaining transaction consistency in HDDBSs using quasi serializable executions. Technical Report CSD-TR-969, Purdue University, March 1990.

[DEK90] W. Du, A. Elmagarmid, and W. Kim. Effects of local autonomy on heterogeneous distributed database systems. Technical Report ACT-OODS-EI-059-90, MCC, February 1990.

[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*, pages 113–120, Gaithersburg, Maryland, October 1989.

[ED90a] A. Elmagarmid and W. Du. Preserving data integrity in HDDBSs using quasi serializable executions. Technical Report CSD-TR-970, Purdue University, March 1990.

[ED90b] A. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the International Conference on Data Engineering*, pages 548–555. February 1988.