Global Scheduling for Flexible Transactions in Heterogeneous Distributed Database Systems

Aidong Zhang
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260

Marian Nodine*
Department of Computer Science
Brown University
Providence, RI 02912

Bharat Bhargava
Department of Computer Science
Purdue University
West Lafayette, IN 47907

Abstract

A heterogeneous distributed database environment integrates a set of autonomous database systems to provide global database functions. A flexible transaction approach has been proposed for the heterogeneous distributed database environments. In such an environment, flexible transactions can increase the failure resilience of global transactions by allowing alternate (but in some sense equivalent) executions to be attempted when a local database system fails or some subtransactions of the global transaction abort. In this paper, we study the impact of compensation, retry, and switching to alternative executions on global concurrency control for the execution of flexible transactions. We propose a new concurrency control criterion for the execution of flexible and local transactions, termed F-serializability, in the error-prone heterogeneous distributed database environments. We then present a scheduling protocol that ensures F-serializability on global schedules. We also demonstrate that this scheduler avoids unnecessary aborts and compensation.

Index terms: Heterogeneous and autonomous database, transaction management, concurrency control, flexible transactions, serializability.

1 Introduction

A heterogeneous distributed database system (HDDBS) integrates a set of autonomous database systems to provide global database functions. In a HDDBS environment, transaction management is handled at both the global and local levels. As a confederation of pre-existing local

*Current address: MCC, 3500 West Balcones Center dr, Austin, TX 78759.

databases, the overriding concern of any HDDBS must be the preservation of local autonomy [Lit86, GMK88, BS88, Pu88, Vei90, VW92]. This is accomplished through the superimposition of a global transaction manager (GTM) upon a set of local database systems (LDBSs). Global transactions are submitted to the global transaction manager, where they are parsed into a set of global subtransactions to be individually submitted to local transaction management systems at local sites (LSs). At the same time, local transactions are directly submitted to the local transaction management systems. Each local transaction management system maintains the correct execution of both local and global subtransactions at its site. It is left to the global transaction manager to maintain the correct execution of global transactions.

The preservation of the atomicity and isolation of global transactions is fundamental in achieving the correct execution of global transactions. Preserving the atomicity or semantic atomicity [GM83] of global transactions in the HDDBS systems has been recognized as an open and difficult issue [SSU91]. The traditional two-phase commit protocol (2PC) developed in distributed database environments has been shown [LKS91a, SKS91, MR91, BZ94] to be inadequate to the preservation of the atomicity of global transactions in the HDDBS environment. For example, some local database systems may not support a visible prepare-to-commit state, in which a transaction has not yet been committed but is guaranteed the ability to commit. In such situations, a local database system that participates in a HDDBS environment may unilaterally abort a global subtransaction without agreement from the global level. Moreover, even if the local database systems are assumed to support a prepare-to-commit state (as in traditional distributed database systems), the potential blocking and long delays caused by such states severely degrade the performance. The concept of compensation, which was proposed [GM83] to address the semantic atomicity of long-running transactions, has been shown [LKS91a] to be useful in the HDDBS environment. Using this technique, the global subtransactions of a global transaction may commit unilaterally at local sites. Semantic atomicity guarantees that if all global subtransactions commit, then the global transaction commits; otherwise, all tentatively committed global subtransactions are compensated.

Mehrotra et al. [MRKS92] have identified the class of global transactions for which the semantic atomicity can be maintained in the HDDBS environment. Each global transaction contains a set of subtransactions which are either compensatable, retriable, or pivot, and at most one subtransaction can be pivot. In [ZNBB94], it was shown that this class can be extended by specifying global transactions as flexible transactions. Flexible transaction models, such as ConTracts, Flex Transactions, S-transactions, and others [DHL91, ELLR90, BDS$^+$93], increase the failure resilience of global transactions by allowing alternate subtransactions to be executed when an LDBS fails or a subtransaction aborts. In a non-flexible transaction, a global subtransaction abort is followed either by a global transaction abort decision or by a retry of the global subtransaction. With the flexible transaction model, there is an additional option of switching to an alternate global transaction execution. The following example is illustrative:

Example 1 *A client at bank b_1 wishes to withdraw \$50 from her savings account a_1 and deposit it in her friend's checking account a_2 in bank b_2. If this is not possible, she will deposit the \$50 in her own checking account a_3 in bank b_3. With flexible transactions, this is represented by the following set of subtransactions:*

- t_1: *Withdraw \$50 from savings account a_1 in bank b_1;*
- t_2: *Deposit \$50 in checking account a_2 in bank b_2;*
- t_3: *Deposit \$50 in checking account a_3 in bank b_3.*

In this global transaction, either $\{t_1, t_2\}$ or $\{t_1, t_3\}$ is acceptable, with $\{t_1, t_2\}$ preferred. If t_2 fails, t_3 may replace t_2. The entire global transaction thus may not have to be aborted even if t_2 fails. □

Flexibility allows a flexible transaction to adhere to a weaker form of atomicity, which we term *semi-atomicity*, while still maintaining its correct execution in the HDDBS. Semi-atomicity allows a flexible transaction to commit as long as a subset of its subtransactions that can represents the execution of the entire flexible transaction commit. By enforcing semi-atomicity on flexible transactions, the class of executable global transactions can be enlarged in a heterogeneous distributed database system [ZNBB94]. The effect of retrial and compensation methods were investigated to preserve semi-atomicity on flexible transactions. However, the design of scheduling approaches to global concurrency control for the execution of flexible transactions has not been carefully investigated.

Global concurrency control considering the effect of compensation with respect to traditional transaction model has been extensively studied. In [KLS90], a formal analysis is presented of those situations in which a transaction may see the partial effect of another transaction before these partial effects are compensated. It is then proposed in [LKS91a] that, to prevent an inconsistent database state from being seen in a distributed database environment, a global transaction should be unaffected by both aborted and committed subtransactions of another global transaction. A concurrency control correctness criterion, termed *serializability with respect to compensation (SRC)*, is further proposed in [MRKS92] to preserve database consistency in the HDDBS environment throughout the execution of global transactions possessing no value dependencies among their subtransactions. This criterion prohibits any global transaction that is serialized between a global transaction G_i and its compensating transaction CG_i from accessing the local sites at which G_i aborts. All these proposed approaches are inadequate to a situation in which value dependencies are present among the subtransactions of a global transaction. Value dependencies, which specify data flow among the global subtransactions of each global transaction, are important characteristics of flexible transactions.

In this paper, we will propose a concurrency control criterion for the execution of flexible and local transactions in the HDDBS environment. We will carefully analyze the effects of compensation, retry, and switching to alternate executions on global concurrency control. We will propose a specific correctness criterion for schedules of concurrent flexible and local transactions, called F-serializability, in the HDDBS environment. We will then demonstrate that an F-serializable execution maintains global database consistency. We will also present a graph-based scheduling protocol for flexible transactions that ensures F-serizalibility.

This paper is organized as follows. Section 2 introduces the system and flexible transaction models. In Section 3, we discuss the issues relevant to global concurrency control on the execution of flexible and local transactions. Section 4 proposes a global concurrency control criterion. In Section 5, we offer a scheduling protocol to implement the proposed criterion. Concluding remarks are presented in Section 6.

2 Preliminaries

In this section, we shall introduce the system and transaction models that will be used in the rest of the paper.

2.1 System Model

The system architecture under consideration is shown in Figure 1. A HDDBS consists of a set of $\{\mathrm{LDBS}_i$, for $1 \leq i \leq m\}$, where each LDBS_i is a pre-existing autonomous database management system on a set of data items D_i at a local site (LS_i), superimposed on which is a global transaction manager (GTM). We assume that there is no integrated schema provided and users know the existence of local database systems. The set of data items at a local site LS_i is partitioned into *local* data items, denoted LD_i, and *global* data items, denoted GD_i, such that $LD_i \cap GD_i = \emptyset$ and $D_i = LD_i \cup GD_i$. The set of all global data items is denoted GD, $GD = \bigcup_{i=1}^{m} GD_i$. Flexible transactions are submitted to the GTM and then divided into a set of subtransactions which are submitted to the LDBSs individually, while local transactions are directly submitted to LDBSs. We assume that the GTM submits flexible transaction operations to the local databases through servers that are associated with each LDBS.

In a HDDBS, *global consistency* means that no integrity constraints among the data in the different local databases are violated. As with transactions on a database, global and flexible transactions on a HDDBS should be defined so that, if executed in isolation, they would not violate the global consistency of the HDDBS. The concurrency control protocol schedules the concurrent execution of the flexible subtransactions in the HDDBS such that global consistency is maintained. However, unlike monolithic databases where all the data are strictly controlled by a single trans-

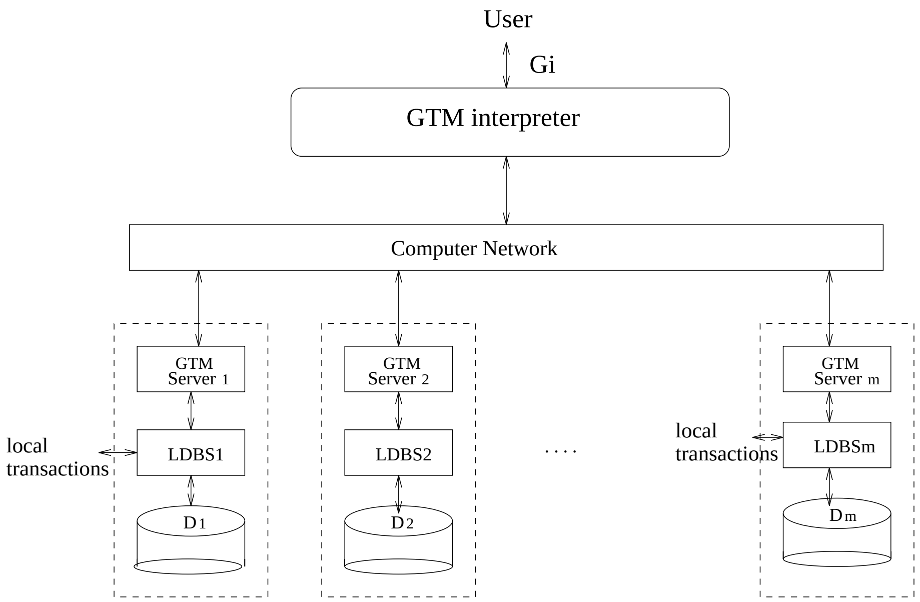

Figure 1: The HDDBS system model.

action manager, HDDBSs submit transactions to autonomous local databases. Thus, the HDDBS transaction manager cannot ensure that transactions submitted independently to local databases do not violate global integrity constraints. Following the previous research commonly proposed in the community [BST90], we assume that all local transactions do not modify the global data items in GD. Note that this will not prevent the local transactions to read global data items, as long as the execution of the local transaction maintains local integrity constraints. Based on this assumption, local transactions maintain both local and global integrity constraints.

2.2 Flexible Transaction Model

From a user's point of view, a *transaction* is a sequence of actions performed on data items in a database. In a HDDBS environment, a *global transaction* is a set of subtransactions, where each *subtransaction* is a transaction accessing the data items at a single local site. The flexible transaction model supports flexible execution control flow by specifying two types of dependencies among the subtransactions of a global transaction: (1) execution ordering dependencies between two subtransactions, and (2) alternative dependencies between two subsets of subtransactions. A formal model has been offered in [ZNBB94]. Below, we shall provide a brief introduction of this model.

Let $\mathcal{T} = \{t_1, t_2, ..., t_n\}$ be a repertoire of subtransactions and $\mathcal{P}(\mathcal{T})$ the collection of all subsets of $\mathcal{T}$. Let $t_i, t_j \in \mathcal{T}$ and $T_i, T_j \in \mathcal{P}(\mathcal{T})$. We assume two types of control flow relations to be defined on the subsets of $\mathcal{T}$ and on $\mathcal{P}(\mathcal{T})$, respectively: (1) (**precedence**) $t_i \prec t_j$ if t_i precedes t_j $(i \neq j)$; and (2) (**preference**) $T_i \rhd T_j$ if T_i is preferred to T_j $(i \neq j)$. If $T_i \rhd T_j$, we also say that T_j is an alternative to T_i.[1] Note that T_i and T_j may not be disjoint. Both precedence and preference relations are irreflexive and transitive. That is, they define partial order relations. In other words, for each $t_i \in \mathcal{T}$, $\neg(t_i \prec t_i)$; and for each $T_i \in \mathcal{P}(\mathcal{T})$, $\neg(T_i \rhd T_i)$. If $t_i \prec t_j$ and $t_j \prec t_k$, then $t_i \prec t_k$; if $T_i \rhd T_j$ and $T_j \rhd T_k$, then $T_i \rhd T_k$.

The precedence relation defines the correct parallel and sequential execution ordering dependencies among the subtransactions. The semantics of the precedence relation refers to the execution order of subtransactions. $t_1 \prec t_2$ implies that t_1 finishes its execution before t_2 does. Note that t_2 may start before or after t_1 finishes. The preference relation defines the priority dependencies among alternate sets of subtransactions for selection in completing the execution of $\mathcal{T}$. For instance, $\{t_i\} \rhd \{t_j, t_k\}$ implies that either t_j and t_k must abort when t_i commits or t_j and t_k should not be executed if t_i commits. In this situation, $\{t_i\}$ is of higher priority than $\{t_j, t_k\}$ to be chosen for execution.

A flexible transaction is defined as follows:

Definition 1 (Flexible transaction) *A flexible transaction $\mathcal{T}$ is a set of related subtransactions on which the precedence ($\prec$) and preference ($\rhd$) relations are defined.*

The execution of a flexible transaction may contain several alternatives. Let T_i be a subset of $\mathcal{T}$, with a precedence relation $\prec$ defined on T_i. It is defined that $(T_i, \prec)$ is a partial order of subtransactions. $(T_i, \prec)$ is a **representative partial order**, abbreviated as **$\prec$-rpo**, if the execution of subtransactions in T_i represents the committed execution of the entire flexible transaction $\mathcal{T}$. Clearly, if $(T_i, \prec)$ is a $\prec$-rpo, then there are no subsets T_{i1} and T_{i2} of T_i such that $T_{i1} \rhd T_{i2}$.

The structure of a flexible transaction $\mathcal{T}$ can thus be depicted as a set of $\prec$-rpos $\{(T_i, \prec), i = 1, ..., k\}$ of subtransactions, with $\bigcup_{i=1}^{k} T_i = \mathcal{T}$.[2] Note that $\mathcal{T}$ may contain more than one subtransaction at a local site. Let $(T, \prec)$ be a $\prec$-rpo of $\mathcal{T}$. A partial order $(T', \prec')$ is a *prefix* of $(T, \prec)$, denoted $(T', \prec') \leq (T, \prec)$, if:

- $T' \subseteq T$;
- for all $t_1, t_2 \in T', t_1 \prec' t_2$ in $(T', \prec')$ if and only if $t_1 \prec t_2$ in $(T, \prec)$; and
- for each $t \in T'$, all predecessors of t in T are in T'.

A partial order $(T', \prec')$ is the prefix of $(T, \prec)$ with respect to $t \in T$, denoted $(T', \prec') \leq (T, \prec)(t)$, if $(T', \prec')$ is a prefix of $(T, \prec)$ and T' contains only all predecessors of t in T. A partial order $(T', \prec')$ is the *suffix* of $(T, \prec)$ with respect to $t \in T$, denoted $(T', \prec') \geq (T, \prec)(t)$, if, for all $t_1, t_2 \in T', t_1 \prec' t_2$ in $(T', \prec')$ if and only if $t_1 \prec t_2$ in $(T, \prec)$ and T' contains only t and all successors of t in T.

[1] In general, the alternate relationship need not exist only between two individual subtransactions; one subtransaction may be a semantic alternative of several subtransactions.

[2] Note that when $k = 1$, a flexible transaction becomes a traditional global transaction.

We now use prefixes and suffixes to show how a flexible transaction can switch from executing one $\prec$-rpo to executing a lower-priority alternative. Intuitively, if $\prec$-rpos $(T_i, \prec_i)$ and $(T_j, \prec_j)$ share some prefix and the subtransactions $t_1, \cdots, t_k$ immediately following that prefix in the execution of $(T_i, \prec_i)$ fail, then $(T_j, \prec_j)$ can continue execution from the point where the shared prefix completed. In this case, the set of $\{t_1, \cdots, t_k\}$ forms a *switching set*, formally defined as follows:

Definition 2 (Switching set) *Let $t_1, \cdots, t_k$ be subtransactions in $\prec$-rpo $(T, \prec)$ of a flexible transaction $\mathcal{T}$, with respective suffixes $(T_i, \prec_i) \geq (T, \prec)(t_i)$ for $i = 1, \cdots, k$. $\{t_1, ..., t_k\}$ forms a* switching set *of $(T, \prec)$ if*

- *there is a $\prec$-rpo $(T', \prec')$ of $\mathcal{T}$ such that $(T - (T_1 \cup ... \cup T_k), \prec'')$ is a prefix of $(T', \prec')$, and*
- *$(T_1 \cup ... \cup T_k) \rhd (T' - (T - (T_1 \cup ... \cup T_k)))$.*

A *switching point* is a subtransaction in a switching set which relates one $\prec$-rpo to another $\prec$-rpo.

Let $p_1 = (T_1, \prec_1)$ and $p_2 = (T_2, \prec_2)$ be two $\prec$-rpos of flexible transaction $\mathcal{T}$. We say that p_1 has higher priority than p_2 in $\mathcal{T}$, denoted $p_1 \rightarrow p_2$, if there are $T_{1i} \subseteq T_1$ and $T_{2j} \subseteq T_2$ such that $T_{1i} \rhd T_{2j}$. The preference relation defines the preferred order over alternatives. We state that two subsets $T_j, T_k \subset \mathcal{T}$ have the same priority if there is a $T_i \subset \mathcal{T}$ such that $T_i \rhd T_j$ and $T_i \rhd T_k$, but $\neg(T_j \rhd T_k)$ and $\neg(T_k \rhd T_j)$.

The execution of a flexible transaction $\mathcal{T}$ at any moment must be uniquely determined. We say that a flexible transaction $\mathcal{T}$ is *unambiguous* if the following conditions are satisfied:

- For any switching set $\{t_1, ..., t_k\}$ in a $\prec$-rpo $(T, \prec)$ of $\mathcal{T}$, $(T_1 \cup ... \cup T_k)$ where $(T_i, \prec_i) \geq (T, \prec)(t_i)$, for $i = 1, ..., k$, has no two alternatives with the same priority.
- None of the $\prec$-rpos $p_1, ..., p_l$ of $\mathcal{T}$ are in a priority cycle such that $p_{i_1} \rightarrow ... \rightarrow p_{i_l} \rightarrow p_{i_1}$ for a permutation $i_1, ..., i_l$ of $1, ..., l$.

Note that the set of all $\prec$-rpos of a flexible transaction may not be clearly ranked, even if it is unambiguous. The aborting of subtransactions determines which alternative $\prec$-rpo will be chosen. In the remainder of this paper, we assume that all flexible transactions are unambiguous. The following example is given in [ZNBB94]:

Example 2 *Consider a travel agent information system arranging a travel schedule for a customer. Assume that a flexible transaction $\mathcal{T}$ has the following subtransactions:*

t_1*: withdraw the plane fare from account a_1;*

t_2*: withdraw the plane fare from account a_2;*

t_3*: reserve and pay for a non-refundable plane ticket;*

t_4: *rent a car from Avis;*

t_5: *book a limo seat to and from the hotel.*

The following $\prec$*-rpos are defined on the above subtransactions:*

$$p_1 = (\{t_1, t_3, t_4\}, \prec), \quad p_2 = (\{t_1, t_3, t_5\}, \prec), \quad p_3 = (\{t_2, t_3, t_4\}, \prec), \quad p_4 = (\{t_2, t_3, t_5\}, \prec),$$

where $\{t_1\}$ *is the switching set of* p_1 *and* $\{t_4\}$ *is the switching set of both* p_1 *and* p_3. *With these switching sets, we have* $\{t_1, t_3, t_4\} \rhd \{t_2, t_3, t_4\}$ *and* $\{t_4\} \rhd \{t_5\}$. *Clearly, the set of* $\prec$*-rpos in this flexible transaction is unambiguous. Note that* $p_1 \rightarrow p_2$ *and* $p_1 \rightarrow p_3$, *but* p_2 *and* p_3 *cannot be ranked in any preferred order.* □

In each $\prec$-rpo of subtransactions, the value dependencies among operations in different subtransactions define data flow among the subtransactions. Let $(T, \prec)$ be a $\prec$-rpo and T have subtransactions $t_1, t_2, \cdots, t_n$. We say that t_{j_t} is *value dependent* on $t_{j_1}, ..., t_{j_{t-1}}$ $(1 \leq j_1, ..., j_t \leq n)$ if the execution of one or more operations in t_{j_t} is determined by the values read by $t_{j_1}, ..., t_{j_{t-1}}$.

Each subtransaction is categorized as either *retriable*, *compensatable*, or *pivot*. We say that a subtransaction t_i is *retriable* if it is guaranteed to commit after a finite number of submissions when executed from any consistent database state. The retriability of subtransactions is highly determined by implicit or explicit integrity constraints. For instance, a bank account usually has no upper limit, so a deposit action is retriable. However, it usually does have a lower limit, so a withdrawal action is not retriable.

A subtransaction is *compensatable* if the effects of its execution can be semantically undone after commitment by executing a compensating subtransaction at its local site. We assume that a compensating subtransaction ct_i for a subtransaction t_i will commit successfully if persistently retried.[3] ct_i must also be independent of the transactions that execute between t_i and ct_i. Local database autonomy requires that arbitrary local transactions be executable between the time t_i is committed and the time ct_i is executed, and these local transactions must be able to both see and overwrite the effects of t_i during that time. For example, consider a HDDBS that has account a in LS_1 and account b in LS_2, with the integrity constraints $a \geq 0$ and $b \geq 0$. Suppose a transaction $\mathcal{T}_1$ transfers \$100 from a to b. The withdrawal subtransaction t_1 at LS_1 is compensatable, while the deposit subtransaction t_2 at LS_2 is not. The compensation of t_2 may violate the integrity constraint $b \geq 0$ if a local transaction which is executed between t_2 and its compensating subtransaction takes the amount of b. Note that both t_1 and t_2 are compensatable in the traditional distributed database environment, which ensures that the transactions that are executed between t_2 and its compensating subtransaction ct_2 are commutative with ct_2 [KLS90, BR92].

A subtransaction t_i is a *pivot* subtransaction if it is neither retriable nor compensatable. For

[3]This requirement, termed *persistence of compensation*, has been discussed in the literature [GM83].

example, consider a subtransaction which reserves and pays for a non-refundable plane ticket. Clearly, this subtransaction is not compensatable. This subtransaction is also not retriable, since such a ticket might never be available.

The concept of semi-atomicity was introduced in [ZNBB94] for the commitment of flexible transactions. The execution of a flexible transaction $\mathcal{T}$ is *committable* if the property of semi-atomicity is preserved, which requires one of the following two conditions to be satisfied:

- All its subtransactions in one $\prec$-rpo commit and all attempted subtransactions not in the committed $\prec$-rpo are either aborted or have their effects undone.
- No partial effects of its subtransactions remain permanent in local databases.

We will now define the *commit dependency* relationships between any two subtransactions of a flexible transaction that should be obeyed in the commitment of these subtransactions. We say that t_j is *commit dependent* on t_i, denoted $t_i \rightarrow_c t_j$, if the commitment of t_i must precede that of t_j to preserve semi-atomicity. Clearly, if $t_i \prec t_j$ in $(T_i, \prec)$ $(1 \leq i \leq k)$, then $t_i \rightarrow_c t_j$. These dependencies, which are determined by the execution control flow among subtransactions, are termed *e-commit dependencies*. To ensure that the execution of a $\prec$-rpo can terminate, the commitment of compensatable subtransactions should always precede that of pivot subtransactions, which in turn should precede the commitment of retriable subtransactions. These dependencies are termed *t-commit dependencies*. Also, for those subtransactions which are retriable, value dependencies must be considered in determining a commitment order. Each retriable subtransaction remains retriable without resulting in any database inconsistency, as long as all other subtransactions that are value dependent upon it have not committed. Such dependencies are termed *v-commit dependencies*.

We say that a flexible transaction is *well-formed* if it is committable. Well-formed flexible transactions have been identified in [ZNBB94] for which the semi-atomicity can be maintained. Well-formed global transactions have been identified in [MRKS92] for which the semantic atomicity [GM83] can be maintained. We assume, in this paper, that the flexible transactions are well-formed for the discussion of global concurrency control. We say that a database state is *consistent* if it preserves database integrity constraints. As defined for traditional transactions, the execution of a flexible transaction as a single unit should map one consistent HDDBS state to another. However, for flexible transactions, this definition of consistency requires that the execution of subtransactions in each $\prec$-rpo must map one consistent HDDBS state to another.

In the above discussion, we have used banking and travel agency as application examples. It has been recognized that the concept of flexible transactions can be extended to specify the activities involving the coordinated execution of multiple tasks performed by different processing entities [SR93, BDS$^+$93]. For instance, in manufacturing applications, flexible transactions are used to specify and control the data flow between agile partner applications. Typical inter-task dependencies include: (1) Ordering dependencies which define the parallel and sequential executions among tasks, (2) Trigger dependencies which define the contingency executions among tasks, and (3) Real-time dependencies which define real-time constraints on tasks; e.g., a chronological dependency is defined by specifying the start time and the expected completion time of tasks. These dependencies can be realized using the flexible transaction model. The extension of these dependencies have also been recognized in the the concept of workflow which has been used as a specification facility to separate control and data flows in a multi-system application from the rest of the application code [SR93, BDS+93].

3 Issues in Global Concurrency Control

In this section, we will discuss various inconsistent scenarios that may arise when compensation or retrial are allowed. These observations are important input for the establishment of a suitable global concurrency control correctness criterion.

3.1 Global Serializability

Global serializability [BS88, BGMS92] is an accepted correctness criterion for the execution of (non-flexible) global and local transactions in the HDDBS environment. A global schedule S is *globally serializable* if the committed projection from S of both global transactions in the HDDBS environment and transactions that run independently at local sites is conflict-equivalent to some serial execution of those transactions.[4] In the traditional transaction model, it has been shown that a global schedule S is globally serializable if and only if all S_k $(k = 1, ..., m)$ are serializable and there exists a total order O on global transactions in S such that, for each local site LS_k $(1 \leq k \leq m)$, the serialization order of global subtransactions in S_k is consistent with O [GRS91, MRB+92, ZE93]. Note that each global transaction can have more than one subtransactions at a local site, as long as their serialization order is consistent with O. That is, if global transaction G_2 precedes global transaction G_3 and follows global transaction G_1 in the serialization order, then the serialization order of all subtransactions of G_2 must precede that of all subtransactions of G_3 and follow that of all subtransactions of G_1 at each local site.

Following the definition of semi-atomicity on flexible transactions, a committed flexible transaction can be considered as a traditional global transaction that contains only the subtransactions in its committed $\prec$-rpo. Some subtransactions in the flexible transaction which do not belong to the committed $\prec$-rpo may have committed and their effects are compensated. In this discussion, we call such subtransactions *invalid subtransactions*. Invalid subtransactions and their compensating transactions are termed *surplus* transactions, as their effects are not visible in the HDDBS once

[4] See [BHG87] for the definitions of committed projection and conflict equivalence.

the flexible transaction has committed.

Let S be the global schedule containing the concurrent executions of both the subtransactions (and compensating subtransactions) of flexible transactions $\mathcal{T}_1, ..., \mathcal{T}_l$ and a set of local transactions. Let S^c be the projection from S of the committed local transactions and the subtransactions of the committed $\prec$-rpos of $\mathcal{T}_1, ..., \mathcal{T}_l$. We extend *global serializability* to treat surplus transactions as local transactions:

Definition 3 (Global serializability) *A global schedule S is globally serializable if the projection of committed local, flexible, and surplus transactions in S is conflict-equivalent to some serial execution of these transactions.*

In the rest of this paper, for the sake of similicity, we assume that each global transaction has at most one subtransaction at each local site. However, all the theorems can be directly applied to the general situation where multiple subtransactions are permitted in each global transaction. In the case of flexible transaction, we consider that each $\prec$-rpo of a flexible transaction has at most one subtransaction at a local site. Note that $\mathcal{T}$ may still contain more than one subtransaction at a local site, provided that they are in different $\prec$-rpos.

We denote $o_1 <_S o_2$ if operation o_1 is executed before operation o_2 in S. Let $t_{i1}, \ldots, t_{im}$ be the subtransactions at local sites $\mathrm{LS}_1 \ldots \mathrm{LS}_m$ in the committed $\prec$-rpo of flexible transaction $\mathcal{T}_i$, and $t_{j1}, \ldots, t_{jm}$ be the analogous subtransactions in the committed $\prec$-rpo of flexible transaction $\mathcal{T}_j$. t_{ip} and t_{jp} both are executed at local site LS_p. Let $\rightarrow_s$ be a serialization ordering on transactions, with $t_{ip} \rightarrow_s t_{jp}$ indicating that the execution of t_{ip} must be serialized before that of t_{jp} on the LDBS on which they executed. Applying the above global serializability theory in the execution of flexible transactions, we have the following theorem:

Theorem 1 *Let $\mathcal{T}_1, ..., \mathcal{T}_l$ be the well-formed flexible transactions in global schedule S. Assume that all LDBSs maintain serializability on the local transactions and subtransactions at their sites. The global serializability of S is preserved if, for S^c, there exists a permutation $\mathcal{T}_{i_1}, ..., \mathcal{T}_{i_l}$ of $\mathcal{T}_1, ..., \mathcal{T}_l$ such that, for any $\mathcal{T}_{i_{j_1}}, \mathcal{T}_{i_{j_2}} \in \{\mathcal{T}_{i_1}, ..., \mathcal{T}_{i_l}\}$, $t_{i_{j_1}p} \rightarrow_{S_p^c} t_{i_{j_2}p}$ for all local sites LS_p, where $j_1 < j_2$ and $1 \leq p \leq m$.*

Several solutions have been proposed to enforce the serializable execution of global transactions, including forced local conflicts [GRS91]. These approaches are readily applicable to flexible transactions. However, globally serializable schedules may no longer preserve database consistency in the execution of flexible and local transactions, because the global serializability criterion fails to consider the constraints on the committed subtransactions which are not a part of the committed $\prec$-rpos, and on their compensating subtransactions. In fact, every commit protocol which uses compensation during the execution of flexible transactions may face difficulties with the preservation of the consistency of globally serializable schedules.

3.2 Problematic Situations

We will now consider some scenarios that may arise if compensation or retrial are allowed. Before discussing these scenarios, we first introduce the concept of the *serialization point*, which is similar to the existing notion of the serialization event [ED90] and serialization function [MRB+92].

Definition 4 (Serialization point) *Let t_{ip} and t_{jp} be two subtransactions executed on local site LS_p. Operation o_{ip} of t_{ip} is a serialization point of t_{ip} in global schedule S if, for any subtransaction t_{jp}, there exists an operation o_{jp} of t_{jp} such that $t_{ip} \rightarrow_{S_p} t_{jp}$ if and only if $o_{ip} <_{S_p} o_{jp}$.*

The determination of serialization points depends on the concurrency control protocol of the LDBS. For instance, if the local database system uses strict two-phase locking (pessimistic concurrency control), then the serialization point can fall anywhere between the moment when the subtransaction takes its last lock and its commitment [MRB+92]. When local conflicts are forced, each subtransaction updates a shared data item at the local site. The order of these updates forms the serialization order. A detailed discussion of this procedure can be found in [MRB+92]. Note that, in general, any individual transaction may not have a serialization point in the schedule. This is the case when serialization graph testing [BHG87] is used as the concurrency control protocol. However, in enforcing globally serializable schedules, we must determine the serialization orders of subtransactions at local sites in order to deal with local indirect conflicts [GRS91, MRB+92, ZE93]. Thus, we assume that the serialization point of each subtransaction can be determined at the local site. We also assume that, with the help of forced local conflicts, we can ensure that the serialization point is reached after the transaction begins, and before it commits (the *bounded serialization point assumption*).

First, let us consider the following example:

Example 3 *Consider a HDDBS that has data items a, b at LS_p and data item c at LS_{p+1}. Let the integrity constraints be $a > c$ and $b > c$. Two flexible transactions $\mathcal{T}_1$ and $\mathcal{T}_2$ at local site LS_p are executed as follows:*

- *t_{1p}, which does $b := b - 1$, commits.*
- *t_{2p}, which does $a := b$, executes its serialization point, enforcing $t_{1p} \rightarrow_s t_{2p}$. t_{2p} has read data item b that was written by t_{1p}.*
- *t_{1p+1}, which does $c := c - 1$, aborts; $\mathcal{T}_1$ makes a global decision to abort.*

- *Compensating subtransaction ct_{1p}, which does $b := b + 1$, is executed.* □

In this example, all the effects of $\mathcal{T}_1$ are eventually removed from the execution, including the effects of t_{1p}. Global serializability is preserved, because flexible transaction $\mathcal{T}_1$ was aborted and correctly compensated for, even though its effects were read by flexible transaction $\mathcal{T}_2$. However, $\mathcal{T}_2$ proceeds based on the reading of the data item that was updated by t_{1p} and thus may be inconsistent. Consider an initial database state $a = 5, b = 5, c = 4$. The resulting database state after the execution of ct_{1p} would be $a = 4, b = 5, c = 4$, which is inconsistent.

The concept of *isolation of recovery* [LKS91b, LKS91a] states that a global transaction should be unaffected by both the aborted and the committed subtransactions of other global transactions. In addition, the above example shows that, if a global transaction is affected by a committed subtransaction which must later be compensated, the task of constructing the compensating subtransaction will be greatly complicated by the need to restore database consistency. Such compensating subtransactions must be capable of undoing any effects that may have been seen by other global transactions. For example, in the above example, ct_{1p} must restore not only data item b but also data item a. However, it is also undesirable for ct_{1p} to have to check for reads by other global subtransactions. Note that the effects of the compensated subtransactions on local transactions need not be considered if we assume that the execution of a subtransaction transfers the local database from one consistent state to another. This leads us to the following observation:

Observation 1 *A necessary condition for maintaining database consistency in an execution containing concurrent flexible transactions in which compensating subtransactions undo only the effects of their corresponding compensatable subtransactions is that, for each compensatable subtransaction t_{ip} of $\mathcal{T}_i$ at given local site LS_p $(1 \leq p \leq m)$, subtransactions that are subsequently serialized at the local site must not read the data items updated by t_{ip} until either $\mathcal{T}_i$ makes a global decision to commit the $\prec$-rpo containing t_{ip} or the compensating subtransaction for t_{ip} has executed its serialization point.*

As a further complication, a compensating subtransaction ct_{ip} may still unilaterally abort and need to be retried. Some conflicting subtransactions may have to be aborted to ensure that they are serialized after ct_{ip}. To avoid such undesirable cascading aborts, we must delay the execution of the serialization point of such conflicting subtransactions until one of the following conditions holds for t_{ip}:

- The flexible transaction containing t_{ip} has made a decision to commit the $\prec$-rpo which contains t_{ip}.
- ct_{ip} has committed.

The following example is also problematic:

Example 4 *Two flexible transactions $\mathcal{T}_1$ and $\mathcal{T}_2$ at local site LS_p are executed as follows:*

- *$\mathcal{T}_1$ makes a global decision to commit.*
- *t_{1p} executes its serialization point.*
- *t_{2p} executes its serialization point, enforcing $t_{1p} \rightarrow_s t_{2p}$.*
- *t_{1p}, which is retriable, unilaterally aborts.*
- *t_{1p} is resubmitted and re-executes its serialization point. Now we have $t_{2p} \rightarrow_s t_{1p}$, which contradicts the order $t_{1p} \rightarrow_s t_{2p}$ that was enforced.*

At this point, if the serialization order is not consistent with those at other local sites, t_{2p} must be aborted. □

To avoid cascading aborts, we make the following observation:

Observation 2 *A necessary condition for avoiding cascading aborts when a subtransaction is retried is to ensure that a subtransaction does not execute its serialization point until all retriable subtransactions that precede it in the serialization order of the execution and have not been aborted have successfully committed.*

If, in this situation, we do not avoid the cascading abort, we also can allow a situation where *compensation* must be cascaded:

- $\mathcal{T}_1$ makes a global decision to commit.
- t_{1p} executes its serialization point.
- t_{2p} executes its serialization point, enforcing $t_{1p} \rightarrow_s t_{2p}$. Then it commits.
- t_{1p}, which is retriable, unilaterally aborts.
- t_{1p} is resubmitted and re-executes its serialization point. Now we have $t_{2p} \rightarrow_s t_{1p}$, which contradicts the order $t_{1p} \rightarrow_s t_{2p}$ that was enforced.

In this case, $\mathcal{T}_2$ must backtrack to compensate for t_{2p}, so that the correct serialization order can be attained at LS_p. Unfortunately, if t_{2p} is not compensatable, this may be impossible.

Whether or not the global transaction is flexible, using either compensation or retrial to regain consistency leads to blocking. Furthermore, for flexible transactions, switching to an alternate

$\prec$-rpo can also create problems. Let us consider an alternate in flexible transaction $\mathcal{T}_1$. Let $t_{11} \prec t_{1i} \prec t_{1n}$ be the preferred $\prec$-rpo, and let $t_{11} \prec t_{1j}$ be the second alternate $\prec$-rpo. Note that t_{1j} is at a different local site than any subtransaction in the preferred $\prec$-rpo. We then have the following example, where $\mathcal{T}_1$ should be serialized before $\mathcal{T}_2$:

Example 5 *The executions of two flexible transactions $\mathcal{T}_1$ and $\mathcal{T}_2$ are given as follows:*

- *t_{11}, which is pivot, commits (therefore $\mathcal{T}_1$ makes a global decision to commit).*
- *t_{1i} executes its serialization point.*
- *t_{2j} executes its serialization point.*
- *t_{1i} unilaterally aborts.*
- *$\mathcal{T}_1$ chooses an alternate $\prec$-rpo and submits t_{1j}. It has yet to execute its serialization point, so we now have $t_{2j} \rightarrow_s t_{1j}$, which contradicts the serialization order $\mathcal{T}_1 \rightarrow_s \mathcal{T}_2$ that was enforced.*

At this point, t_{2j} must be aborted to maintain global serializability. □

Again, if the cascading abort is not avoided, we can also allow a cascading compensation situation:

- t_{1p}, which is compensatable, commits.
- t_{2p}, which is pivot, commits (and $\mathcal{T}_2$ makes a global decision to commit).
- $\mathcal{T}_1$ backtracks, issuing compensating subtransaction ct_{1p}.
- $\mathcal{T}_1$ tries a different $\prec$-rpo, issuing subtransaction t'_{1p}.

We now have $t_{2p} \rightarrow_s t'_{1p}$, which contradicts the serialization order $\mathcal{T}_1 \rightarrow_s \mathcal{T}_2$. Furthermore, since t_{2p} is not compensatable, the only way to regain global serializability is to abort the entire flexible transaction $\mathcal{T}_1$.

We observe and later prove that if the cascading abort is avoided, then the cascading compensation cannot occur. To avoid cascading aborts in this situation, we have the following observation:

Observation 3 *A necessary condition for avoiding cascading aborts when an alternate $\prec$-rpo is attempted is to ensure that, at any local site, an LDBS does not execute the serialization point of a subtransaction until, for all uncompleted flexible transactions that precede it in the global serialization order, no alternate subtransaction can possibly be initiated.*

Observations 1, 2 and 3 above indicate that some blocking of the execution of subtransactions which reach their serialization point early may be unavoidable with a concurrency control algorithm designed to avoid cascading aborts. This blocking will result in the delay of the commitment operations of these subtransactions. Observations 1 and 2 relate directly to delays which are caused by compensation or retrial and which cannot therefore be avoided by any global transaction model that uses these techniques to regain consistency after a subtransaction aborts.

Observation 3 concerns delays that arise only with flexible transactions. There are conflicting considerations here; the more flexible (and therefore more failure-resilient) global transactions will be prone to greater delays. Global transactions with less flexibility, which are less resilient to failure, will have fewer delays. Consequently, while the flexible transaction approach can indeed extend the scope of global transactions, it does cause more blocking than does the traditional transaction model.

The observations we have made concerning concurrency control will play a dominant role in the design of concurrency control algorithms for maintaining global serializability on the execution of flexible and local transactions. Following these observations, we see that the execution of a flexible transaction may be greatly affected by the concurrent execution of other flexible transactions.

4 A Global Scheduling Criterion

We will now propose a new concurrency control criterion for the execution of flexible and local transactions. This criterion, termed F-serializability, places restrictions on global serializability. Thus, the set of F-serializable schedules is a subset of globally serializable schedules. Following Observations 1-3, we see that only Observation 1 has an effect on the definition of such a criterion, while Observations 2 and 3 will impact on the design of a concurrency control protocol.

4.1 Serializability with Flexible Transactions

Clearly, the database inconsistency that may be caused by compensation can be prevented in a globally serializable execution simply by requiring that, for each compensatable subtransaction t_i of $\mathcal{T}_i$ at given local site LS_p $(1 \leq p \leq m)$, conflicting subtransactions that are subsequently serialized at the local site do not execute their serialization points until either $\mathcal{T}_i$ makes a global decision to commit the $\prec$-rpo containing t_i or the compensating subtransaction for t_i has executed its serialization point.

A less restrictive approach is also possible. By definition, a compensating subtransaction ct_i should be able to compensate the effect of t_i regardless of what transactions execute between t_i and ct_i. However, such transactions may propagate the effect of t_i to other transactions and such propagation cannot be statically considered in ct_i. For flexible transaction $\mathcal{T}_j$ following $\mathcal{T}_i$ in the global serialization order, if subtransaction t_j of $\mathcal{T}_j$ accesses (reads or writes) the data items not written by t_i of $\mathcal{T}_i$, t_j will definitely not propagate the effect of t_i and can then be serialized between t_i and ct_i in the same manner as any local transaction. Only when t_j accesses the data items written by t_i may database inconsistency result. Let $AC(t)$ denote the set of data items that t accesses and commits, $RC(t)$ denote the set of data items that t reads and commits, and $WC(t)$ denote the set of data items that t writes and commits. We define a *compensation-interference free* property on global schedules as follows:

Definition 5 (Compensation-interference free) *A global schedule S is compensation-interference free if, for any subtransaction t_j which is serialized between a subtransaction t_i and its compensating transaction ct_i in S, $WC(t_i) \cap AC(t_j) = \emptyset$.*

We now propose a new global concurrency control criterion as follows:

Definition 6 (F-serializability) *Let S be a global schedule of a set of well-formed flexible transactions and local transactions. S is F-serializable if it is globally serializable and compensation-interference free.*

Note that, in Definition 6, the execution of a well-formed flexible transaction may result in both a committed flexible transaction and some surplus transactions. Comparing the definition of F-serializable schedules given above with that of global serializable schedules given in Definition 3, we can easily see that the set of F-serializable schedules is a subset of globally serializable schedules. However, if we had followed the traditional definition of global serializability in which all subtransactions and their compensating subtransactions of a flexible transaction at a local site are treated as a logically atomic subtransaction, then the set of F-serializable schedules would be a superset of globally serializable schedules. For example, consider a committed flexible transaction $\mathcal{T}_1$ which has generated a surplus pair of subtransaction t_1 and its compensating subtransaction ct_1. Let another flexible transaction $\mathcal{T}_2$ contain data items such that $RC(t_1) \cap AC(\mathcal{T}_2) \neq \emptyset$, but $WC(t_1) \cap AC(\mathcal{T}_2) = \emptyset$. $\mathcal{T}_2$ can be F-serialized between t_1 and ct_1, but cannot be globally serialized between t_1 and ct_1. In addition, since the surplus pair of the subtransactions belong to a different representative partial order from the committed one, these subtransactions can be treated as a separate global transaction.

Theorem 2 given below demonstrates that F-serializability ensures global database consistency. We first show that the compensation-interference free property in an F-serializable global schedule is inherited in its conflict equivalent schedule.

Lemma 1 *Given an F-serializable global schedule S, any schedule S' that is conflict-equivalent to S is compensation-interference free.*

Proof: The proof proceeds by contradiction. Suppose we do have a schedule S' that is conflict-equivalent to S and is not compensation-interference free. There then exists a subtransaction t_j such that it is serialized between a subtransaction t_i and its compensating transaction ct_i in S', and t_j accesses some data item d written by t_i. Since S' is conflict equivalent to S, t_j must also access d written by t_i in S and is serialized between t_i and ct_i in S. Consequently, S is not compensation-interference free, contradicting the given condition. □

Without loss of generality, we assume that, in the below theorem, all local, flexible, or surplus transactions in S are committed. Thus, S is identical to S^c.

Theorem 2 *An F-serializable schedule S preserves global database consistency.*

Proof: Let S be an F-serializable schedule that transfers a consistent database state DS^0 to a new database state DS^1. Let $S' = T_1T_2...T_n$ be its equivalent global serial schedule, where T_i ($1 \leq i \leq n$) is a committed local, flexible, invalid or compensating (sub)transaction. By lemma 1 we know that S' is compensation-interference free. We now demonstrate that DS^1 is consistent.

We first prove that every local, flexible, or invalid (sub)transaction reads consistent database state. The proof proceeds by induction on the position of each transaction in S':

Basis: Obviously, T_1 can only be either local, flexible or invalid (sub)transaction. Since T_1 reads from DS^0, it reads consistent database state.

Induction: Assume that for all transactions T_i, $i = 1, ..., k-1$ ($k \leq n$), if T_i is not a compensating subtransaction, then T_i reads consistent state. Consider T_k. There are two cases:

- T_k is a local transaction.

  Since all transactions $T_1, ..., T_{k-1}$ preserve local database consistency, T_k thus reads locally consistent database state.

- T_k is an invalid subtransaction or a committed flexible transaction.

  Let D be the set of all data items existing in the global database. Let D' be $D-\{$the set of data items updated by those invalid subtransactions appeared in $T_1, ..., T_{k-1}$ but their compensating subtransactions appeared after $T_k\}$. Since S' is compensation-interference free, T_k reads only the state of D'. Thus, T_k reads only consistent database state.

By the semantics of compensation, the partial effects of invalid subtransactions in S' are semantically compensated by their compensating subtransactions. Since no effects of invalid subtransactions are seen by other transactions before they are compensated, any inconsistencies caused by these invalid subtransactions are restored by their compensating subtransactions. Let S'' be S' restricted to those transactions that are neither invalid subtransactions or their compensating subtransactions. Thus, S'' consists only the serial execution of atomic local and flexible transactions. Since each transaction in S'' sees a consistent database state, then S'' preserves the global database consistency. Therefore, DS^1 is consistent. □

4.2 Avoiding Cascading Aborts and Compensations

Lemma 2 *Let $\mathcal{T}_i \rightarrow_s \mathcal{T}_j$ be maintained in global schedule S. The following rules are necessary and sufficient for a flexible transaction scheduler to follow in order to avoid cascading aborts for serialization reasons:*

- *No subtransaction of flexible transaction $\mathcal{T}_j$ can execute its serialization point until all retriable subtransactions of flexible transactions $\mathcal{T}_i$ such that $\mathcal{T}_i \rightarrow_s \mathcal{T}_j$ have committed.*
- *No subtransaction of flexible transaction $\mathcal{T}_j$ can execute its serialization point until all alternative subtransactions of flexible transactions $\mathcal{T}_i$ such that $\mathcal{T}_i \rightarrow_s \mathcal{T}_j$ either have committed or can no longer participate in $\mathcal{T}_i$'s committed $\prec$-rpo.*

Proof: The necessary condition was shown in Observations 2 and 3.

The sufficient condition can be shown by the observation that cascading aborts really occur when some subtransaction t_j of some flexible transaction $\mathcal{T}_j$ is serialized before a subtransaction t_i of some flexible transaction $\mathcal{T}_i$ on the same local database, with $\mathcal{T}_i \rightarrow_s \mathcal{T}_j$. The above conditions ensure that the serialization point of a subtransaction, which fixes its place in the local database's serialization order, is not made until all subtransactions that could precede it in the global serialization order are either committed or decided against. Since no earlier subtransaction can attempt to execute a new subtransaction on the local database, no cascading abort can occur. □

Recall from Example 5 that cascading compensations can occur when a subtransaction t_2 reads from some uncommitted subtransaction t_1 and then commits. If t_1 later aborts, t_2 must be compensated for. We show the following theorem with respect to cascading compensations:

Theorem 3 *A flexible transaction scheduler avoids cascading compensations if it avoids cascading aborts.*

Proof: Assume that the scheduler avoids cascading aborts. This means that it never needs to force the abort of an uncommitted subtransaction t because of the violation of serialization. By Lemma 2, this is guaranteed by delaying the execution of the serialization point of t until all subtransactions which must be serialized previously have committed. Therefore, t cannot have committed before these subtransactions commit. Thus, t need never be compensated for. So cascading compensation is also avoided. □

5 A Scheduling Protocol

In this section, we present a GTM scheduling protocol that ensures F-serializability on the execution of local and flexible transactions, and avoids cascading aborts. This protocol is based on the assumption that if the concurrency control protocol of a local database does not allow the HDDBS to determine the serialization point for each subtransaction, a ticket scheme similar to [GRS91] can be implemented on the local database. Thus, the serialization point for a subtransaction is always reached between the time the subtransaction begins and the time it commits.

For the GTM scheduling protocol, we propose a execution graph testing method to avoid the high overhead of keeping track of serialization points, to ensure F-serializability, and to avoid aborts. For scheduling purposes, we maintain a stored subtransaction execution graph (SSEG) among subtransactions to be scheduled. The SSEG is defined as follows:

Definition 7 (Stored Subtransaction Execution Graph) *The* Stored Subtransaction Execution Graph (SSEG) *of a set of flexible transactions in global schedule S is a directed graph whose nodes are global subtransactions and compensating subtransactions for those flexible transactions, and whose edges $t_i \rightarrow t_j$ indicate that t_j must serialize before t_i due to preference, precedence, or conflict.*

Global subtransaction nodes are labeled t_{ip}^{m} for flexible transaction $\mathcal{T}_i$ running on local site p. If more than one global subtransaction is defined for $\mathcal{T}_i$ on LS_p, then the nodes can be ordered by the $\triangleright$ order of their $\prec$-rpos, and m indicates this node's position in that order. If t_{ip}^{m} is compensatable, its compensating subtransaction's node is ct_{ip}^{m}. We begin with a few definitions. A flexible transaction *commits* once its pivot subtransaction commits. We say that a flexible transaction *robustly terminates* once all subtransactions in the committed $\prec$-rpo have committed and all compensating subtransactions for committed subtransactions not in the committed $\prec$-rpo have also committed.

The GTM scheduling protocol assumes that each global subtransaction and compensating subtransaction predeclares its read- and write- sets. It includes node and edge insertion and deletion rules, and an operation submission rule. All nodes and edges associated with a flexible transaction are inserted as a unit. If some edge insertion fails for flexible transaction $\mathcal{T}_i$, no edges may be inserted for subsequent flexible transaction $\mathcal{T}_j$ until either the insertion succeeds or all edges for $\mathcal{T}_i$

have been deleted. Nodes and edges for flexible transaction $\mathcal{T}_i$ are inserted into the SSEG according to the following rules:

Node Insertion Rule: Insert a node for each subtransaction defined for $\mathcal{T}_i$. For each compensatable subtransaction insert a node ct_{ip}^m.

Edge Insertion Rule: For subtransaction t_{ip}^m, where edge insertion does not cause a cycle:

1. For each previously-scheduled t_{hp}^n, $h < i$, insert edge $t_{ip}^m \to t_{hp}^n$.
2. For each previously-scheduled ct_{hp}^n, $h < i$, if $WC(t_{hp}^n) \cap AC(t_{ip}^m) \neq \emptyset$, insert edge $t_{ip}^m \to ct_{hp}^m$.
3. If t_{ip}^m is compensatable, insert edge $ct_{ip}^m \to t_{ip}^m$.
4. If t_{ip}^m is (e-, v-, or t-) commit-dependent on t_{iq}^n insert edge $t_{ip}^m \to t_{iq}^n$.
5. For all $n < m$, insert edge $t_{ip}^m \to t_{ip}^n$. If t_{ip}^n is compensatable, insert edge $t_{ip}^m \to ct_{ip}^n$.

The first two edge insertion cases ensure F-serializability. The third rule ensures that in a surplus pair the invalid subtransaction precedes its compensating transaction. The rest of the cases ensure that, for all flexible transactions, the resulting schedule is commit-dependency preserving and all alternatives are attempted in $\rhd$ order.

Nodes and edges are deleted from the SSEG according to the following rules:

Node Deletion rule:

1. Upon completion of a flexible transaction backtrack, delete all nodes representing subtransactions in the current switching set or its successors, as well as the nodes representing their compensating subtransactions.
2. Upon commitment of a subtransaction or compensating subtransaction, delete its node.
3. Upon commitment of a pivot or a retriable subtransaction, delete all nodes representing its alternatives and their successors in the flexible transaction, as well as the nodes representing the compensating subtransactions of these deleted nodes.
4. Upon robust termination of a flexible transaction, delete its remaining nodes.

Edge Deletion Rule: Delete all edges incident on deleted nodes.

The operations of a global subtransaction of $\mathcal{T}_i$ are submitted to the local databases according to the following rule.

Operation Submission Rule: Submit operations of a subtransaction (including begin and commit) to its local database only if its node in the SSEG has no outgoing edges.

The SSEG algorithm is defined based on the above rules. Note that the implementation of the Operation Submission Rule can vary depending on what concurrency control mechanism is used at each local site. If each local DBMS uses the strict two-phase locking as its concurrency control mechanism, then the serialization point of a subtransaction can be controlled by the GTM [MRB+92]. As a result, some operations of a subtransaction t_i of a flexible transaction may be submitted before other subtransactions which are serialized before t_i reach their serialization points. However, for other concurrency control mechanisms, we generally cannot have such gain from the local DBMSs.

We now show that the SSEG algorithm maintains global consistency. We begin with a basic lemma on the restraints the SSEG places on the execution, then apply that to the scheduling algorithm.

Lemma 3 *If there is an edge $t_{jq} \rightarrow t_{ip}$ in the SSEG, then if both t_{ip} and t_{jq} execute and commit, t_{ip} must serialize before t_{jq}.*

Proof: By the operation submission rule, no operation of t_{jq} (including begin and commit) can be executed until the node t_{jq} has no outgoing edges. Therefore, t_{jq} cannot begin (and consequently by the bounded serialization point assumption cannot execute its serialization point) until the edge $t_{jq} \rightarrow t_{ip}$ is deleted. By the edge deletion rule, the edge is only deleted once the node t_{ip} is deleted. By the node deletion rule, the node t_{ip} is only deleted once the subtransaction commits. By the bounded serialization point assumption, t_{ip} must have executed its serialization point before it commits. Therefore, t_{ip} must have executed its serialization point before t_{jq} could possibly have executed its serialization point, so the two subtransactions must serialize in the order that t_{ip} precedes t_{jq}. □

Note also that if there is an edge $t_{jq} \rightarrow t_{ip}$ in the SSEG, but one of them does not commit, but either fails to execute or aborts, then at most one of them is present in the serialization order, so how they are actually executed is unimportant.

Theorem 4 *Consider two flexible transactions $\mathcal{T}_i$ and $\mathcal{T}_j$. $\mathcal{T}_i$ is F-serialized before $\mathcal{T}_j$ if the nodes and edges of $\mathcal{T}_i$ are inserted before those of $\mathcal{T}_j$.*

Proof: Since all nodes are inserted for $\mathcal{T}_i$ before any nodes are inserted for $\mathcal{T}_j$, we know by the edge insertion rule that all edges between subtransactions of $\mathcal{T}_i$ and subtransactions of $\mathcal{T}_j$ must be directed from some subtransaction of $\mathcal{T}_j$ to some subtransaction of $\mathcal{T}_i$ in the same local database.

Consider some local site LS_p at which $\mathcal{T}_i$ and $\mathcal{T}_j$ conflict. By the first edge insertion rule, there is an edge $t_{jp} \rightarrow t_{ip}$ in the SSEG. By Lemma 3 this means that if both subtransactions execute and commit, t_{ip} serializes before t_{jp} at LS_p. If one of the two subtransactions does not commit, then the two committed $\prec$-rpos of the flexible transactions may not both have subtransactions at LS_p. In such a case, LS_p will not have any effect on the serialization order of $\mathcal{T}_i$ and $\mathcal{T}_j$.

If t_{ip} is executed and is later compensated for, and t_{jp} is also executed, we have the following two cases:

Case 1: The compensation-interference free condition does not hold between the execution of t_{ip} and t_{jp}. By the second edge insertion rule, there is an edge $t_{jp} \rightarrow ct_{ip}$ in the SSEG, so the operations of t_{jp} could not be submitted until this edge is deleted. By Lemma 3, if both t_{jp} and ct_{ip} execute and commit, this means that ct_{ip} must serialize before t_{jp}.

Case 2: The compensation-interference free condition does hold. There is then no edge $t_{jp} \rightarrow ct_{ip}$, and t_{jp} may be interleaved between t_{ip} and ct_{ip}. However, the resulting schedule is still compensation-interference free.

Consequently, $\mathcal{T}_i$ is F-serialized before $\mathcal{T}_j$. □

Following Theorem 4, we can see that the SSEG algorithm maintains global consistency. We now also show that the SSEG algorithm has the additional desirable property of avoiding cascading aborts and cascading compensations. We know by Theorem 3 that if it avoids cascading aborts, then it avoids cascading compensations. Therefore, we show the following:

Theorem 5 *The SSEG protocol for scheduling flexible transactions avoids cascading aborts.*

Proof: By Lemma 3, we know that if, for subtransactions t_i and t_j, there is an edge $t_j \rightarrow t_i$ and t_i, t_j commit, then t_i must serialize before t_j. In fact, similar reasoning shows us that if such an edge exists in the SSEG, then t_i must either commit, abort, or be removed from consideration before t_j can begin. Thus, for this proof, we merely need to show that edges are inserted into the SSEG that are sufficient to prevent a concurrent execution that allows for a cascading abort.

The first condition for avoiding cascading aborts is that no subtransaction of flexible transaction $\mathcal{T}_j$ can execute its serialization point until all retriable subtransactions of any $\mathcal{T}_i$ such that $\mathcal{T}_i \rightarrow_s \mathcal{T}_j$ have committed. The first case in the edge insertion rule enforces this by inserting edges $t^m_{jp} \rightarrow t^n_{ip}$ for all previously-scheduled t^n_{ip}.

The second condition for avoiding cascading aborts is that no subtransaction of flexible transaction $\mathcal{T}_j$ can execute its serialization point until all alternative subtransactions of any flexible transaction $\mathcal{T}_i$ such that $\mathcal{T}_i \rightarrow_s \mathcal{T}_j$ either have committed or can no longer participate in $\mathcal{T}_i$'s committed $\prec$-rpo. The first case in the edge insertion rule also enforces this.

Since the SSEG protocol ensures that both conditions necessary for avoiding cascading aborts are enforced, the theorem holds. □

Thus, we have shown that the SSEG algorithm maintains global consistency. We have also shown that the SSEG algorithm has the additional desirable property of avoiding cascading aborts and cascading compensations. Based on the rules for insertion and deletion of nodes and edges in SSEG as well as the operation submission rule, the SSEG algorithm can be efficiently implemented in the HDDBS environment.

6 Conclusions

This paper has proposed a new correctness criterion on the execution of local and flexible transactions in the HDDBS environment. We have advanced a theory which facilitates the maintenance of F-serializability, a concurrency control criterion that is stricter than global serializability in that it prevents the flexible transactions which are serialized between a flexible transaction and its compensating subtransactions to affect any data items that have been updated by the flexible transaction. Consequently, no effect of a compensatable subtransaction is spread to other flexible transactions before it is compensated. In order to prevent cascading aborts, the effects of retrial and alternatives on concurrency control must also be considered. These factors generate unavoidable blocking on the execution of flexible transactions. Thus, trade-off between flexibility of specifying global transactions and high concurrency on the execution of flexible transactions remains.

References

[ANRS92] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proceedings of the 18th VLDB conference*, Vancouver, Canada, August 1992.

[BDS+93] Y. Breitbart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. *SIGMOD Record*, 22(3):23–30, September 1993.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *The VLDB Journal*, 1(2):181–239, October 1992.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems.* Addison-Wesley Publishing Co., 1987.

[BR92] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase Update Issues. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 135–142, Chicago, IL, June 1988.

[BST90] Y. Breitbart, A. Silberschatz, and G. Thompson. Reliable Transaction Management in a Multidatabase System. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 215–224, May 1990.

[BZ94] Bharat Bhargava and Aidong Zhang. Scheduling with Compensation in Multidatabase Systems. In *Proceedings of the Third International Conference on System Integration*, San Paulo, Brazil, 1994.

[DHL91] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. *17th VLDB Proceedings*, pages 113–122, 1991.

[ED90] 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.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–581, Brisbane, Australia, August 1990.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMK88] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed Systems. In *Proceedings of the First International Symposium on Databases for Parallel and Distributed Systems*, pages 158–166, Austin, Texas, December 1988.

[GRS91] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. In *Proceedings of the 7th Intl. Conf. on Data Engineering*, pages 314–323, Kobe, Japan, April 1991.

[KLS90] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[Lit86] W. Litwin. A multidatabase interoperability. *IEEE Computer*, 19(12):10–18, December 1986.

[LKS91a] E. Levy, H. Korth, and A. Silberschatz. A Theory of Relaxed Atomicity. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1991.

[LKS91b] E. Levy, H. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991.

[MR91] P. Muth and T.C. Rakow. Atomic commitment for integrated database systems. In *Proceedings of the 7th International Conference on Data Engineering*, pages 296–304, Kobe, Japan, April 1991.

[MRB+92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 288–297, 1992.

[MRKS92] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz. A transaction model for multidatabase systems. In *Proceedings of International Conference on Distributed Computing Systems*, June 1992.

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

[SKS91] Nandit Soparkar, Henry F. Korth, and Abraham Siberschatz. Failure-resilient transaction management in multidatabases. *IEEE Computer*, 24(12):28–36, December 1991.

[SR93] Amit Sheth and Marek Rusinkiewicz. On Transaction Workflows. *IEEE Data Engineering Bulletin*, 16(2), 1993.

[SSU91] A. Silberschatz, M. Stonebraker, and J. Ullman. Database systems: Achievements and opportunities. *Communication of ACM*, 34(10):110–120, 1991.

[Vei90] J. Veijalainen. *Transaction Concepts in Autonomous Database Environments.* R. Oldenbourg Verlag, Germany, 1990.

[VW92] J. Veijalainen and A. Wolski. Prepare and commit certification for decentralized transaction management in rigorous heterogeneous multidatabases. In *Proceedings of the International Conference On Data Engineering*, 1992.

[ZE93] Aidong Zhang and Ahmed Elmagarmid. A theory of global concurrency control in multidatabase systems. *The VLDB Journal*, 2(3):331–359, July 1993.

[ZNBB94] Aidong Zhang, Marian Nodine, Bharat Bhargava, and Omran Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, May 1994.