Ensuring Semi-Atomicity 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

Contents

1 Introduction ...................................................... 3
  1.1 Contributions ............................................... 5
  1.2 Related Research .......................................... 5
  1.3 Structure of the Paper .................................... 6

2 A Formal Model of Flexible Transactions .................. 6

3 Semi-atomicity ................................................ 10

4 Constructing Committable Flexible Transactions .......... 12
  4.1 Commit Dependencies ....................................... 12
  4.2 Well-formed Flexible Transactions ....................... 13
  4.3 Semi-atomicity for Committability of Flexible Transactions .......... 17

5 The Flexible Commit Protocol .................................. 19
  5.1 System Model ............................................... 19
  5.2 The Commit Protocol ....................................... 20
  5.3 Persistent Transmission in the HDDBS Environment .......... 24
  5.4 Discussion .................................................. 26

6 Effects of Concurrency Control on Commitment Ordering .. 27

7 Application in Traditional Distributed Database Systems .. 32

8 Conclusions .................................................. 33

*This paper is an extended version of the paper that appeared in ACM SIGMOD'94 [ZNBB94].
Abstract

Global transaction management in a distributed database system requires cooperation from local sites to ensure the consistent and reliable execution of global transactions. In a heterogeneous distributed database environment, various local sites make conflicting assertions of autonomy over the execution of global transactions. The flexible transaction approach increases the failure resilience of global transactions by allowing alternative (but in some sense equivalent) executions to be attempted when a local database system fails or some subtransaction of the global transaction aborts.

In this paper, we define the concept of semi-atomicity, which is a correctness criterion related to semantic atomicity but which is applied specifically to flexible transactions. We offer a fundamental characterization of the flexible transaction model as a set of partial orders of alternatives, each of which consists of a set of interdependent subtransactions. We precisely define semi-atomicity with respect to these transactions and characterize the restrictions that must be placed on a flexible transaction for semi-atomicity to be ensured. We present an approach to global transaction commitment that preserves the semi-atomicity of flexible transactions allowing local sites to autonomously maintain serializability and recoverability. Our approach allows a subtransaction to commit before or after a commit/abort decision is made for its global transaction and uses compensation, retry and switching to alternative executions to regain correctness. Since the subtransactions in an execution may depend on one another, we use commit dependencies to control the commitment order of the subtransactions in a flexible transaction execution.

Semi-atomicity enlarges the class of executable global transactions in a heterogeneous distributed database system. We will first explore the flexible transaction model in the heterogeneous distributed database setting and then demonstrate the application of semi-atomicity in the traditional distributed database environment.
1 Introduction

A heterogeneous distributed database system (HDDBS) is a collection of heterogeneous and autonomous local database systems (LDBSs) which is viewed as a single unit. There are two types of transactions in an HDDBS. A local transaction is submitted directly to an LDBS and accesses only its information. In contrast, a global transaction may access several local databases. A global transaction is submitted to a global transaction manager (GTM) superimposed upon a set of autonomous local database systems, where it is parsed into a series of global subtransactions, each of which is submitted to a single LDBS.

Traditionally, the correctness criterion for executing a single local transaction on an LDBS is atomicity; each transaction executes either all or none of its operations. Serializability ensures that the database system appears to execute local transactions individually, in isolation. When a transaction aborts, a syntactic undo can be used, in which the original values of any changed data items is rewritten. Distributed database systems attempt to extend the techniques used for a single local database into a system where data is stored at multiple sites under a single transaction manager. The atomicity of the global transactions in traditional distributed database systems can be ensured using the well-known two-phase commit (2PC) protocol and its variants [BHGC87]. A prepare-to-commit state for global subtransactions, where the ability to commit is guaranteed, must be supported in each LDBS. Furthermore, the transaction manager can exploit its knowledge of the execution of all the operations performed on the different LDBSs to enhance concurrency while still maintaining transaction atomicity and isolation.

In an HDDBS, global transaction execution must be coordinated under a single transaction manager, assuming local database system autonomy. Among other stipulations, this states that LDBSs can unilaterally execute, commit, and abort both local transactions and global subtransactions. As a result, LDBSs cannot cooperate to execute and commit a global transaction atomically [SKS91]. Specifically, HDDBSs cannot assume that all participating LDBSs support a prepare-to-commit state that can be accessed by a global transaction manager in the process of committing its global subtransactions. Thus, when the decision is made to commit a global subtransaction, the LDBS cannot segregate the processing needed to guarantee a successful commit (pending a global transaction commit/abort decision) from the processing that allows transactions that conflict with the global subtransaction to proceed. In such situations, a local site (LS) that participates in an HDDBS environment may unilaterally abort a global subtransaction without agreement from the global level (termed a local unilateral abort). Consequently, the HDDBS cannot ensure that subtransactions in multiple local sites will commit consistently with a global commit decision. 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. Thus, the 2PC protocol has not been widely used.
Given these difficulties with preserving the atomicity of global transactions, the correctness criterion for global transactions is usually assumed to be semantic atomicity [GM83]. Semantic atomicity relies entirely on the possibility of either compensating for (semantically undo) or re-executing subtransactions, since the GTM has no knowledge of the execution of local transactions on the LDBS and therefore cannot implement a syntactic undo. With semantic atomicity, a global transaction commits its subtransactions independently. If a global decision is made to abort, all committed global subtransactions are compensated for. When a global decision is made to commit, all uncommitted global subtransactions must be retried until they commit. Consistency is compromised because a local transaction that follows some global subtransaction may see the effects of that global subtransaction, even if the global transaction eventually aborts. The requirements of local database system autonomy place the execution of the local transactions completely out of the control of the GTM.

Flexible transaction models, such as ConTracts, Flex Transactions, S-transactions, and others [DHL91, ELLR90, BDS+93], increase the failure resiliency 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 alternative 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.

Since the flexible transaction model was proposed, much research has been devoted to its application [ANRS92, KPE92]. Most of this work has assumed the availability of visible prepare-to-commit states in local database systems. In such a scenario, the preservation of either the atomicity
or the semi-atomicity of flexible transactions is relatively straightforward.

1.1 Contributions

In this paper, we propose a formal model of flexible transactions and precisely define semi-atomicity. We present an approach which preserves semi-atomicity in an HDDBS environment in which the local database systems are required to ensure only serializability and recoverability [BHGD87].

A flexible transaction is defined as a prioritized set of alternatives, each of which embodies one way of successfully executing the flexible transaction. Each alternative consists of a set of subtransactions and a set of dependencies over those subtransactions. This methodology differs from previous approaches in that no specific application semantics are involved in specifying the structure of flexible transactions.

We classify the set of committable flexible transactions that can be executed in an error-prone HDDBS environment. We combine the forward and backward recovery techniques defined in [MRKS92] with the ability to switch to an alternative global transaction execution to eliminate the requirement of local prepare-to-commit states. The proposed flexible transaction model substantially enhances the scope of global transaction management beyond that offered by the traditional global transaction model. It also presents an approach that can be used to improve performance in the traditional distributed database environment.

1.2 Related Research

In order to handle local unilateral aborts, approaches using forward recovery (redo and retry) and backward recovery (compensation) have been proposed in the literature. These approaches seek to ensure the semantic atomicity [GM83] of global transactions in HDDBSs. When a subtransaction of a global transaction aborts, the GTM may either re-execute it until commitment or undo the effects of the committed subtransactions. The strategies characterizing these approaches can be classified by the relative timing of the commitment of subtransactions in the local databases with respect to the global transaction commit/abort decision [MR91]. The work in [WV90, BST90] enforces a global decision on the subtransactions by redoing or retrying them as necessary. The work in [PRR91, LKS91b, NZ94] commits subtransactions locally before a global decision is made and rely on compensation when a global transaction is aborted. The work in [MRKS92, Nod93] combines these two approaches. With the forward approach, all subtransactions must be redoable or retrievable, while the backward approach requires that all subtransactions must be compensatable. With the combined approach, only one subtransaction of each global transaction can be neither retrievable nor compensatable, and the rest of its subtransactions must be either retrievable or compensatable. Consequently, the ability to specify global transactions becomes severely limited when
the traditional global transaction model is employed in HDDBSs.

1.3 Structure of the Paper

This paper is organized as follows. Section 2 introduces the fundamental flexible transaction model. Section 3 defines the property of semi-atomicity. In Section 4, we define those flexible transactions that can be executed in the error-prone HDDBS environment without requiring local prepare-to-commit states. In Section 5, we present the flexible transaction recovery protocol and demonstrate its effectiveness in preserving the semi-atomicity of flexible transactions. In Section 6, we discuss the effects of concurrency control on commitment ordering. Section 7 discusses the application of the proposed commit protocol in the conventional distributed database environment. Concluding remarks are presented in Section 8.

2 A Formal Model of Flexible Transactions

Following [ELLR90], the definition of flexible transactions takes the form of a high-level applications description. Various applications semantics are captured in the flexible transaction definition, including commit and abort dependencies and the acceptable set of successful subtransactions. Unfortunately, such a semantics-oriented formulation of flexible transactions may not prevent redundancy in the dependency specification, and the structure of flexible transactions cannot generally be effectively depicted. Delineating a generic structure for flexible transactions is thus necessary for the discussion of flexible transaction management. In this section, we define precisely a flexible transaction model for global transactions.

From a user's point of view, a transaction is a sequence of actions performed on data items in a database. In an 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. We assume that each global transaction has at most one subtransaction at each local site.\footnote{This is necessary for the concurrency control of global transactions [GPZ86].}

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. Below, we shall formally delineate the flexible execution control flow in the flexible transaction model.

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

The precedence relation defines the correct parallel and sequential execution ordering dependencies among the subtransactions, while the preference relation defines the priority dependencies among alternate sets of subtransactions for selection in completing the execution of \( T \).

A flexible transaction can be defined as follows:

**Definition 1 (Flexible transaction)** A flexible transaction \( T \) is a set of related subtransactions on which the precedence \((<)\) and preference \((>)\) relations are defined.

As this basic definition of flexible transactions provides only a vague picture of the structure of flexible transactions, we shall now seek a more precise delineation. Let \( T_i \) be a subset of \( T \), with a precedence relation \(<_i\) defined on \( T_i \). We then say that \( (T_i, <_i) \) is a partial order of subtransactions. \( (T_i, <_i) \) is a **representative partial order**, abbreviated as \(<_{-\text{rpo}}\), if the execution of subtransactions in \( T_i \) represents the execution of the entire flexible transaction \( T \). Clearly, if \( (T_i, <_i) \) is a \(<_{-\text{rpo}}\), then there are no subsets \( T_{i1} \) and \( T_{i2} \) of \( T_i \) such that \( T_{i1} \succ T_{i2} \). Since each global transaction has at most one subtransaction at a local site, each \(<_{-\text{rpo}}\) of a flexible transaction must have at most one subtransaction at a local site.

The structure of a flexible transaction \( T \) can thus be depicted as a set of \(<_{-\text{rpo}}\) \( \{(T_i, <_i), i = 1, \ldots, k\} \) of subtransactions, with \( \bigcup_{i=1}^{k} T_i = T \).\(^3\) Note that \( T \) may contain more than one subtransaction at a local site, provided that they are in different \(<_{-\text{rpo}}\). Let \( (T, <) \) be a \(<_{-\text{rpo}}\) of \( T \). A partial order \( (T', <') \) is a **prefix** of \( (T, <) \), denoted \( (T', <') \leq (T, <) \), if:

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

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

We now use prefixes and suffixes to show how a flexible transaction can switch from executing one \(<_{-\text{rpo}}\) to executing a lower-priority alternative. Intuitively, if \(<_{-\text{rpo}}\) \( (T_i, <_i) \) and \( (T_j, <_j) \) share

\(^2\)In general, the alternate relationship need not exist only between two individual subtransactions; one subtransaction may be a semantic alternative of several subtransactions.

\(^3\)Note that when \( k = 1 \), a flexible transaction becomes a traditional global transaction.
some prefix and the subtransactions \( t_1, \ldots, t_k \) 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 \( \{t_1, \ldots, t_k\} \) forms a switching set, formally defined as follows:

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

- there is a \( \prec\text{-rpo} (T', \prec') \) of \( T \) such that \((T - (T_1 \cup \ldots \cup T_k), \prec'') \) is a prefix of \((T', \prec')\), and
- \((T_1 \cup \ldots \cup T_k) \triangleright (T' - (T_1 \cup \ldots \cup T_k))\).

A switching point is a subtransaction in a switching set which relates one \( \prec\text{-rpo} \) to another \( \prec\text{-rpo} \).

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

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

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

Note that the set of all \( \prec\text{-rpos} \) of a flexible transaction may not be clearly ranked, even if it is unambiguous. The aborting of subtransactions determines which alternative \( \prec\text{-rpo} \) will be chosen.

In the remainder of this paper, we assume that all flexible transactions are unambiguous.

So far, we have syntactically specified a flexible transaction as a set of alternate \( \prec\text{-rpos} \) of subtransactions that is determined by the relations of precedence and preference. The semantics of the precedence relation refers to the execution order of subtransactions. For instance, \( t_1 \prec t_2 \) may imply that \( t_2 \) cannot start before \( t_1 \) finishes or that \( t_2 \) cannot finish before \( t_1 \) finishes. Similarly, the preference relation defines alternative choices and their priority. For instance, \( \{t_i\} \triangleright \{t_j, t_k\} \) may imply that \( t_j \) and \( t_k \) must abort when \( t_i \) commits or that \( t_j \) 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.

To make the structure of a flexible transaction more visible, we may describe the structure of flexible transaction \( T \) by an execution dependency graph, denoted \( EDG(T) \). This is a directed
graph whose nodes are all subtransactions of $T$ labeled with the $\prec$-rpos that execute them and whose edges are all $t_i \rightarrow t_j$ ($t_i, t_j \in T$), where $t_i$ precedes $t_j$ in $T$ and there is no other subtransaction $t_k$ which follows $t_i$ and precedes $t_j$ in that $\prec$-rpo. The following example is illustrative:

**Example 2** Consider a travel agent information system arranging a travel schedule for a customer. Assume that a flexible transaction $T$ 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:

- $p_1 = \{t_1, t_3, t_4\}$, $p_2 = \{t_1, t_3, t_5\}$,
- $p_3 = \{t_2, t_3, t_4\}$, $p_4 = \{t_2, t_3, t_5\}$,

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 $\{t_1, t_3, t_4\} \triangleright \{t_2, t_3, t_4\}$ and $\{t_4\} \triangleright \{t_3\}$. The $EDG(T)$ is shown in Figure 1. 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 $t_1, t_2, \ldots, t_n$. We say that $t_{j_i}$ is value dependent on $t_{j_1}, \ldots, t_{j_{i-1}}$ (1 ≤ $j_1, \ldots, j_i$ ≤ $n$), denoted $t_{j_1} \rightarrow_v t_{j_i}$, $t_{j_2} \rightarrow_v t_{j_i}$, ..., $t_{j_{i-1}} \rightarrow_v t_{j_i}$, if the execution of one or more operations in $t_{j_i}$ is determined by the values read by $t_{j_1}, \ldots, t_{j_{i-1}}$.

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 HDBBS state to another.

3 Semi-atomicity

We now discuss the execution of flexible transactions. We assume that the execution of a subtransaction can be viewed by the system as a sequence of read and write accessing operations followed by a termination operation (either a commit or an abort). We denote commit and abort operations as $c$ and $a$ (possibly subscripted).

Traditionally, a transaction must be executed atomically, requiring that either all or none of its actions be completed. In the HDBBS environment, this concept of atomicity has been relaxed to that of semantic atomicity [GM83]. Semantic atomicity differs from atomicity in that a global transaction is allowed to commit parts of its results at different times. If all subtransactions commit, then the entire global transaction commits; otherwise, the effects of all tentatively committed subtransactions are undone and the entire global transaction is aborted. The concept of semantic atomicity can be further relaxed in the execution of flexible transactions. Since a flexible transaction allows for the specification of multiple $\prec$-rpos and results in the successful execution and commitment of the subtransactions in one of those $\prec$-rpos (which is termed committed $\prec$-rpo), the execution of a flexible transaction can proceed in several different ways. The subtransactions in different $\prec$-rpos may be attempted simultaneously, as long as any attempted subtransactions not in the committed $\prec$-rpo can either be aborted or have their effects undone. The semi-atomicity of flexible transactions, which is an extension of semantic atomicity, is defined as follows:

**Definition 3 (Semi-atomicity)** The execution of a flexible transaction $T$ preserves the property of semi-atomicity if one of the following conditions is 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.

Such an extension of semantic atomicity allows different $\prec$-rpos of a flexible transaction to be attempted, possibly concurrently, while ensuring that the effects of either zero or exactly one $\prec$-rpo remain permanent in the HDBBS. Thus, even if a flexible transaction commits, some of its subtransactions may abort and the effects of some committed subtransactions may be undone. We say that the execution of a $\prec$-rpo $(T, \prec)$ can terminate if its execution can move either forward to the commitment of its subtransactions or backward to the removal of any partial effects of the committed subtransactions. Obviously, if the execution of a flexible transaction is semi-atomic, then each $\prec$-rpo of the flexible transaction must terminate.
As local prepare-to-commit states are not presumed in the HDDBS system, we now investigate the preservation of the semi-atomicity of flexible transactions through a unification of the retry and compensation approaches. As pointed out in [MRKS92], in contrast to the redo technique [BST90], the retry technique allows us to relax some of the restrictions on data items that global transactions can read and write.

Each subtransaction is categorized as either retrievable, compensatable, or pivot. We say that a subtransaction \( t_i \) is retrievable if it is guaranteed to commit after a finite number of submissions when executed from any consistent database state. The retriebability 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 retrievable. However, it usually does have a lower limit, so a withdrawal action is not retrievable.

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.\(^4\) \( ct_i \) 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 an 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 \( T_1 \) 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].

The set of compensating subtransactions that are executed for the subtransactions in each \(<\text{-rpo}\) can be considered as an independent global transaction. Consequently, the execution of these compensating subtransactions will not violate the assumption that there exists only one subtransaction for each global transaction at a local site.

A subtransaction \( t_i \) is a pivot subtransaction if it is neither retrievable nor compensatable. For 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 retrievable, since such a ticket might never be available.

[MRKS92] formulates each global transaction as the combination of a set of independent subtransactions, each of which is either compensatable, retrievable, or pivot. At most one subtransaction

\(^4\)This requirement, termed persistence of compensation, has been discussed in the literature [GM83].
can be pivot. In this commit protocol, the compensatable subtransactions must be committed before the commitment of the pivot subtransaction, which in turn must commit before the commitment of the retrievable subtransactions. The global commit/abort decision is determined by the outcome of the pivot subtransaction commit. If it aborts, all of the compensatable subtransactions are compensated for; otherwise, the retrievable subtransactions are attempted until they commit. With flexible transactions, we extend this protocol to allow the execution of a flexible transaction that does not follow this subtransaction commit order and which permits multiple pivot subtransactions in the flexible transaction. A detailed discussion of these concepts follows in the next section.

4 Constructing Committable Flexible Transactions

In this section, we present a formulation for flexible transactions that integrates the retry and compensation techniques to preserve the semi-atomicity of their execution.

4.1 Commit Dependencies

We first examine the commit dependency relationships between any two subtransactions of a flexible transaction that must be obeyed in the commitment of these subtransactions. Let \( T = \{(T_i, \prec_i), i = 1, \ldots, k\} \) be a flexible transaction with subtransactions \( t_1, t_2, \ldots, t_n \). Let \( t_i, t_j \in T \). 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 \( (T_i, \prec_i) \) \((1 \leq i \leq k)\), then \( t_i \rightarrow_c t_j \). These dependencies, which are determined by the execution control flow among subtransactions, are termed \( e\)-commit dependencies.

The type (compensatable, pivot, or retrievable) of subtransactions also determines their commitment order in the preservation of semi-atomicity. 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 retrievable subtransactions. These dependencies are termed \( t\)-commit dependencies.

For those subtransactions which are retrievable, value dependencies must be considered in determining a commitment order. For example, assume that a value written by subtransaction \( t_i \) at local site \( LS_1 \) is dependent on a value read by retrievable subtransaction \( t_j \) at local site \( LS_2 \). If \( t_i \) commits and \( t_j \) aborts, then \( t_j \) should be retried. However, local transactions may be executed after the abort of \( t_j \) but before it is retried at \( LS_2 \). This may result in inconsistencies between the data read from the original execution of \( t_j \) and from its retriangular. To ensure that the retriangular of \( t_j \) does not result in any database inconsistency, when a subtransaction \( t_i \) is value dependent on \( t_j \), the commitment of \( t_j \) must precede that of \( t_i \). Thus, if the retriangular of \( t_j \) leads to a result which
is different from that of its original execution, \( t_i \) can be aborted and re-executed. Consequently, each retrievable subtransaction remains retrievable 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 \( v\text{-commit dependencies}. \)

These three types of commit dependencies are closely related. Whenever a subtransaction \( t_2 \) is e-commit dependent, t-commit dependent, or v-commit dependent on \( t_1 \), we must have \( t_1 \rightarrow_c t_2 \). A set of subtransactions \( t_1, \ldots, t_k \) is in a \emph{commit dependency cycle} if there is a permutation \( i_1, \ldots, i_k \) of \( 1, \ldots, k \) such that \( t_{i_1} \rightarrow_c t_{i_2} \rightarrow_c \ldots \rightarrow_c t_{i_k} \rightarrow_c t_{i_1} \). The existence of a commit dependency cycle in a flexible transaction may cause its commitment to result in a cyclic commit situation, where each subtransaction has to wait for another subtransaction to commit before its commitment. In addition, the possession of two or more pivot subtransactions in a \( \prec \text{-rpo} \) may render it difficult to determine a commit order among them which ensures that the execution of the \( \prec \text{-rpo} \) can terminate. In the remainder of this section, we examine those restrictions that need to be placed on flexible transactions to ensure semi-atomicity.

### 4.2 Well-formed Flexible Transactions

Let \( (T, \prec) \) be a \( \prec \text{-rpo} \) in a flexible transaction. As e-commit dependencies must be obeyed in the commitment of subtransactions, the commitment of any pivot or retrievable subtransaction \( t \) in \( T \) will cause the effects of its predecessors in \( T' \), where \( (T', \prec') \leq (T, \prec)(t) \), to be no longer undoable. Among the pivot subtransactions\(^5\) of \( (T, \prec) \) for which all predecessors are compensatable, we identify one as the \emph{critical point} of the \( \prec \text{-rpo} \). The commitment of this subtransaction requires that the flexible transaction must commit, and the aborting of this subtransaction requires that either \( (T, \prec) \) be switched to an alternative \( \prec \text{-rpo} \) or any partial effects of \( (T, \prec) \) must be undone.

Let \( T' = \{t_1, \ldots, t_k\} \) be the set of all pivot subtransactions of \( (T, \prec) \) for which all predecessors are compensatable. The \emph{critical point} of \( (T, \prec) \) is determined through the following rules:

- if \( t \) is the only pivot subtransaction in \( T' \), then \( t \) is the critical point; otherwise
- if there is a \( t \in T' \) which has been chosen as the critical point for a higher-priority \( \prec \text{-rpo} \), then choose \( t \) as the critical point; otherwise
- if there is a \( t \in T' \) which is not a switching point, then choose \( t \) as the critical point; otherwise
- a subtransaction in \( T' \) is randomly chosen as the critical point.

Any compensatable subtransactions that do not follow pivot or retrievable subtransactions can commit before the critical point, and any retrievable subtransactions which are not ordered with the critical point can commit after the critical point. However, once the critical point commits,\(^5\)

---

\(^5\)If \( (T, \prec) \) has no such pivot subtransaction, then a dummy null pivot subtransaction, which is not ordered in \( \prec \) relation with any subtransaction of \( T \), is created.
the aborting of any pivot subtransaction which is not the critical point and of any compensatable subtransaction which follows a pivot or retrievable subtransaction in \( (T, \prec) \) may hamper \( (T, \prec) \) in termination. Such problematic subtransactions are termed abnormal subtransactions. More precisely:

**Definition 4 (Abnormal subtransaction)** Let \( (T, \prec) \) be a \( \prec \)-rpo in a flexible transaction. A subtransaction \( t \) in \( T \) is abnormal if one of the following conditions is satisfied:

- \( t \) is compensatable and there is a pivot or retrievable subtransaction \( t_1 \) in \( T \) such that \( t_1 \prec t \); or
- \( t \) is a pivot subtransaction but is not the critical point.

Note that only a compensatable or pivot subtransaction may be an abnormal subtransaction.

Clearly, no abnormal subtransactions can be permitted in a traditional global transaction. Otherwise, the semantic atomicity of the global transaction may not be preserved. As a result, each global transaction can only have one pivot subtransaction, and no compensatable subtransaction can commit after any pivot or retrievable subtransactions. This may be too restrictive for some applications, especially those complex global applications in an HDDBS environment which involve many local sites.

The use of flexible transactions permits the presence of abnormal subtransactions. In Example 2, it is obvious that \( t_1 \) and \( t_2 \) are compensatable, \( t_3 \) is pivot, and \( t_4 \) is compensatable. If we assume that a limo is available in the required time period, \( t_5 \) is retrievable. Note that \( t_4 \) is an abnormal subtransaction in both \( p_1 \) and \( p_3 \). If \( t_1 \) and \( t_3 \) have already committed and then \( t_4 \) aborts, the partial effects of \( p_1 \) cannot be undone. However, the execution of \( t_4 \) can be replaced by the execution of \( t_3 \). As \( t_5 \) is retrievable, \( T \) can be committed. Thus, abnormal subtransactions can be permitted in flexible transactions. However, if any abnormal subtransaction aborts, appropriate actions must ensue to continue the execution of the flexible transaction.

Let \( T \) be a flexible transaction and \( (T_i, \prec_i) \) be a \( \prec \)-rpo of \( T \). Let an immediate predecessor of a subtransaction \( t \) denote a subtransaction \( t_1 \) such that \( t_1 \prec t \) and no other subtransaction \( t_2 \) exists such that \( t_1 \prec t_2 \prec t \).

Intuitively, we define an abnormal subtransaction \( t \) to be a blocking point in \( (T_i, \prec_i) \) if it identifies a place at which the aborting or compensation of \( t \) must result in the switching of the execution of \( (T_i, \prec_i) \) to an alternative \( \prec \)-rpo. Formally:

**Definition 5 (Blocking point)** Let \( (T_i, \prec_i) \) be a \( \prec \)-rpo in a flexible transaction. An abnormal subtransaction \( t \) in \( T_i \) is a blocking point if it has no predecessors \( t_1, ..., t_k \) that have the following properties:

- \( t_1, ..., t_k \) are members of a switching set \( SW \), and
Figure 2: Execution dependency graph of Example 3.

- all members of SW and their successors that are not in $T_i'$, where $(T_i', <_i') \geq (T_i, <_i)(t)$, are compensatable, and
- all subtransactions in the alternative to the suffixes of $(T_i, <_i)$ with respect to members in SW are either retrievable or abnormal.

The following example is illustrative:

**Example 3** Assume that a flexible transaction $T$ is defined in Figure 2. Let $t_1, t_5,$ and $t_6$ be compensatable subtransactions, $t_2, t_3,$ and $t_7$ be pivot subtransactions, and $t_4$ and $t_8$ be retrievable subtransactions. There are three $<_p$-pos in $T$:

$p_1 = (\{t_1, t_2, t_3, t_5, t_6, t_7\}, <_1),$

$p_2 = (\{t_1, t_2, t_3, t_8\}, <_2),$

$p_3 = (\{t_1, t_2, t_4\}, <_3),$

where $\{t_5, t_6, t_7\} \succ \{t_8\}, \{t_3, t_8\} \succ \{t_4\}.$ and $\{t_3, t_5, t_6, t_7\} \succ \{t_4\}$. By the critical point determination rules, $t_2$ is the critical point for all three $<_p$-pos, because there is an alternative if pivot subtransaction $t_3$ fails. Thus, $t_3, t_5, t_6, t_7$ are abnormal subtransactions; among them, $t_3, t_5,$ and $t_6$ are the blocking points. Clearly, $\{t_3\}$ and $\{t_3, t_6\}$ are the switching sets of $p_1,$ and $\{t_3\}$ is the switching set of $p_2.$ □

Note that, in Example 3, $t_7$ is not a blocking point. When $t_7$ aborts, the execution of $p_1$ can switch to $p_2$ by compensating the effects of $t_5$ and $t_6.$ Now, let us consider a variation of Figure 2 in which another pivot subtransaction $t_9$ is the successor of $t_5$, as shown in the following figure:
In this situation, \( t_7 \) becomes a blocking point. If \( t_9 \) commits but \( t_7 \) aborts, it is then impossible for \( T \) to either switch from \( p_1 \) to another \( \prec \text{-rpo} \) or to have the partial effects of \( p_1 \) undone.

We now define well-formed flexible transactions as follows:

**Definition 6 (Well-formed flexible transaction)** A flexible transaction \( T \) is well-formed if, for each abnormal subtransaction \( t \) that is a blocking point in a \( \prec \text{-rpo} \) \( (T_i, \prec_i) \) in \( T \), the following conditions are satisfied:

- \( t \) is a member of some switching set \( SW \) such that the other members in \( SW \) are compensatable, and
- if a successor of a member in \( SW \) is not ordered by \( \prec_i \) relation with another member in \( SW \), then it is compensatable, and
- all subtransactions in the alternative to the suffixes of \( (T_i, \prec_i) \) with respect to members in \( SW \) are either retrievable or abnormal.

Thus, in a well-formed flexible transaction, for any \( \prec \text{-rpo} \) \( (T_i, \prec_i) \) which contains an abnormal subtransaction \( t \), there is at least one alternate \( \prec \text{-rpo} \) \( (T_j, \prec_j) \) which shares a prefix with \( (T_i, \prec_i) \) such that the aborting of \( t \) will lead the execution of \( T \) from \( (T_i, \prec_i) \) to \( (T_j, \prec_j) \) without resulting in any database inconsistency. Note that, following Definition 6, a switching set can have at most one blocking point that is a pivot subtransaction.

**Observation 1** If a flexible transaction is well-formed and contains a finite number of subtransactions, it has at least one \( \prec \text{-rpo} \) \( p \) that has no abnormal subtransactions, and no other \( \prec \text{-rpo} \) has lower priority than \( p \).

In Example 2, the flexible transaction \( T \) has two \( \prec \text{-rpos} \) \( p_2 \) and \( p_4 \) that contain no abnormal subtransactions. The only blocking point \( t_4 \) constitutes a switching set. Thus, \( T \) is well-formed. In Example 3, the flexible transaction \( T \) has one \( \prec \text{-rpo} \) \( p_3 \) that contains no abnormal subtransactions. Since the blocking point \( t_3 \) constitutes the switching set and the blocking points \( t_5 \) and \( t_6 \) also constitute the switching set, \( T \) is also well-formed.
4.3 Semi-atomicity for Committability of Flexible Transactions

We now discuss the preservation of semi-atomicity. Following [BHCG87], we define a schedule over a set of transactions as a partial order of the operations of those transactions which orders all conflicting operations and which respects the order of operations specified by the transactions. A global schedule $S$ is a schedule over both local and flexible transactions which are executed in an HDDSBS. We denote $o_1 <_S o_2$ if operation $o_1$ is executed before operation $o_2$ in global schedule $S$.

Clearly, to preserve semi-atomicity, the commit dependencies of flexible transactions must be correctly preserved in global schedules. We formulate the concept of a commit dependency graph that is defined on each well-formed flexible transaction, effectively incorporating all effects of e-commit dependencies, t-commit dependencies, and v-commit dependencies.

**Definition 7 (Commit dependency graph)** A commit dependency graph of a well-formed flexible transaction $T = \{(T_i, \prec_i), i = 1, ..., k\}$, denoted $CDG(T)$, is a directed graph whose nodes are all subtransactions of $T$ and whose edges are all $t_1 \rightarrow t_2 \ (t_1, t_2 \in T_i, \ for \ 1 \leq i \leq k \ and \ t_1 \neq t_2)$ such that:

- (e-commit dependency) $t_1 \prec_i t_2 \ (1 \leq i \leq k)$; or
- (v-commit dependency) $t_1 \rightarrow_o t_2$ and $t_1$ is retrievable; or
- (t-commit dependency) $t_1$ is compensatable but not abnormal and $t_2$ is the critical point; or
- (t-commit dependency) $t_1$ is the critical point and $t_2$ is either pivot or retrievable.

We define the concept of commit dependency preserving on global schedules as follows:

**Definition 8 (Commit dependency preserving)** Let $G$ be a set of well-formed flexible transactions. A global schedule $S$ is commit dependency preserving if, for any two subtransactions $t_1$ and $t_2$ of $T$ in $S$ such that $t_1 \rightarrow t_2$ in $CDG(T)$, $c_{t_1} \in S$ implies $c_{t_1} <_S c_{t_2}$.

**Lemma 1** Let $G$ be a set of flexible transactions that participate in global schedule $S$. If, for each $T$ in $G$, $CDG(T)$ is acyclic, then $S$ is commit dependency preserving.

**Proof:** Since, for each $T$ in $G$, $CDG(T, \prec)$ is acyclic for all $(T_i, \prec)$ in $T$, for any $(T_i, \prec)$ in $T$, $CDG(T_i, \prec)$ may be topologically sorted. Without loss of generality, let $t_1, ..., t_m$ be the nodes of $CDG(T_i, \prec)$ and $j_1, ..., j_m$ be a permutation of $1, 2, ..., m$ such that $t_{j_1}, t_{j_2}, ..., t_{j_m}$ is a topological sort of $CDG(T_i, \prec)$. This order ensures that the commitment orders of these subtransactions in global schedule $S$ conform to the definition of commit dependency preserving. To illustrate this, let $t_i$ and $t_k$ be subtransactions in $T_i$ such that $t_k \rightarrow t_i$. By the definition of $CDG(T_i, \prec)$, $t_k \rightarrow t_i$ is an edge in $CDG(T_i, \prec)$. Thus, $t_k$ must appear before $t_i$ in the topological sort $t_{j_1}, t_{j_2}, ..., t_{j_m}$. If the commitment order of all subtransactions in $T_i$ follows the order of $t_{j_1}, t_{j_2}, ..., t_{j_m}$ in global schedule.
$S$, then the commitment of $t_k$ precedes that of $t_i$ in $S$. Hence, $S$ is commit dependency preserving.

A well-formed flexible transaction in a commit dependency preserving global schedule may still not be semi-atomic. For example, if two alternative retriable subtransactions commit simultaneously, it will be impossible to undo the effects of one of those subtransactions. Since the effects of both will remain, the execution of that flexible transaction cannot be semi-atomic. We define the concept of $F$-recoverability on global schedules as follows:

**Definition 9 (F-recoverability)** Let $G$ be a set of well-formed flexible transactions. A global schedule $S$ is $F$-recoverable if, for each flexible transaction $T$ in $G$, no two pivot or retriable subtransactions which participate in different $\prec$-pos commit in $S$ simultaneously.

**Theorem 1** Let $T$ be a well-formed flexible transaction and $CDG(T)$ be acyclic. If global schedule $S$ is commit dependency preserving and $F$-recoverable, then the semi-atomicity of $T$ is preservable.

**Proof:** Without loss of generality, we assume that each $\prec$-partial order of $T$ contains at least one pivot subtransaction. Following Lemma 1, we can assume that $S$ is commit dependency preserving. The proof proceeds by induction on a number $n$ of $\prec$-partial orders of $T = \{(T_i, \prec), i=1,...,n\}$:

- **Basic step ($n=1$).** $T$ is a traditional global transaction. If every subtransaction of $T_1$ commits, then the atomicity of $(T_1, \prec)$ is obviously preserved. Suppose $t \in T_1$ is aborted. Assume that $t$ is either compensatable or pivot. Since $S$ is commit dependency preserving, by the definition of $\rightarrow$, all committed subtransactions of $T_1$ in $S$ must be compensatable. Hence, the committed partial effects of $T_1$ can be undone by the execution of the corresponding compensating subtransactions. Now assume that $t$ is retriable. Again, since $S$ is commit dependency preserving, by the definition of $\rightarrow$, any subtransactions of $T_1$ which are value dependent on $t$ must not have committed in $S$. Let $t' \in T_1$, which is value dependent on $t$, also be executed but not yet committed in $S$. $t'$ can be aborted and resubmitted for execution if the retrial of $t$ would result in inconsistency in the execution of $t'$. As our model assumes that value dependencies and $\prec$ are the only relationships in effect among the subtransactions of each $\prec$-partial order, $t$ can thus be retried without creating any HDBDS inconsistencies. Hence, the semi-atomicity of $T$ is preservable.

- **Induction.** Suppose for $n = k (\geq 1)$, the semi-atomicity of $T$ is preservable. Consider $n = k + 1$. Let $(T_i, \prec)$ be a $\prec$-partial order of $T$. Consider the following situations:

  1. If every subtransaction of $T_i$ commits, then, since $S$ is $F$-recoverable, no pivot or retriable subtransaction in another $\prec$-partial order of $T$ has committed. Thus, the effects of all committed subtransactions of $T$ which are not in $T_i$ can be undone by the execution of the corresponding compensating subtransactions. Consequently, the semi-atomicity of $T$ is preservable.

  2. Suppose now that $t \in T_i$ is aborted and $t$ is either compensatable and normal or a critical
point. Since \( S \) is commit dependency preserving, by the definition of \( \rightarrow_c \), all committed subtransactions of \( T_i \) in \( S \) must be compensatable. Any of the partial effects of \( T_i \) can be undone. The problem then is reduced to preserving the semi-atomicity of less than \( k + 1 \) \( \prec \)-partial orders of \( T \). Thus, by the induction hypothesis, the semi-atomicity of \( T \) is preservable.

(3) Suppose now that \( t \in T_i \) is aborted and \( t \) is retriable. Then, at least one pivot subtransaction of \( (T_i, \prec) \) has committed. By the F-recoverability of \( S \), we know that no pivot or retriable subtransaction in another \( \prec \)-partial order of \( T \) has committed. As with the proof in the basic step, since \( S \) is commit dependency preserving, \( t \) can be retried until commitment. The situation then resolves to either (1) or (4).

(4) Suppose now that \( t \in T_i \) is aborted and \( t \) is abnormal. Since \( T \) is well-formed, by Definition 6, the execution of \( (T_i, \prec) \) can be changed to another \( \prec \)-partial order \( (T_j, \prec) \) without resulting in any database inconsistency. The problem then is reduced to preserving the semi-atomicity of less than \( k + 1 \) \( \prec \)-partial orders of \( T \). Thus, by the induction hypothesis, the semi-atomicity of \( T \) is preservable.

We say that a flexible transaction is commitable if its semi-atomicity is guaranteed to be preserved. Based upon Theorem 1, if a flexible transaction is well-formed and its commit dependency graph is acyclic, then it is commitable.

Because abnormal subtransactions are permitted, the concept of a well-formed flexible transaction extends the scope of global transactions that can be specified in the HDDBS environment beyond that of the basic global transaction model. This was demonstrated in Example 2, where a compensatable transaction (\( t_4 \): reserve a rental car) could be attempted after the pivot (\( t_3 \): reserve and pay for a non-refundable plane ticket) because of the presence of an alternate retriable transaction (\( t_5 \): book the limo).

5 The Flexible Commit Protocol

We now present a commit protocol for flexible transactions that is based upon Theorem 1 and that maintains semi-atomicity as defined in Definition 3.

5.1 System Model

The system model employed for this protocol is shown in Figure 3. We assume that the GTM submits flexible transaction operations to the local databases through servers that are associated with each LDBS. This model assumes that, at any time, only one \( \prec \)-rpo of each flexible transaction is executing.\(^6\) The local databases may execute their own local transactions independently.

\(^6\)A multiple-threads approach which deals with multiple \( \prec \)-rpos simultaneously can be extended from this basis. Such an approach should not commit more than one pivot or retriable subtransaction simultaneously in different
Figure 3: The HDDBS system model.

In the flexible transaction definition, we assume that each flexible transaction contains a finite number of subtransactions and a specification mechanism is provided to allow users to identify to the system the type (compensatable, pivot, or retriable) of each subtransaction. The compensating subtransactions are submitted together with the corresponding compensatable subtransactions. The GTM ensures that flexible transactions are well-formed. Also, a commit dependency graph is built for each flexible transaction and is checked for cycles. If a flexible transaction is not well-formed or if its commit dependency graph is cyclic, it is rejected.

As each subtransaction begins to execute, the type of the subtransaction is sent to the local database server with the begin command. The operations belonging to the subtransaction are submitted to an individual local database by its server as a part of a single subtransaction. The completion of each submitted operation, as well as the begin and commit operations, are individually acknowledged by the local database server to the GTM.

5.2 The Commit Protocol

In our optimistic protocol, commitment of flexible transactions is approached in a dynamic manner. Each subtransaction is permitted to commit locally as soon as possible after it has finished execution. This is needed to deal with the possible presence of value dependencies among subtransactions. If \( t_1 \rightarrow_v t_2 \) is a value dependency between two subtransactions \( t_1 \) and \( t_2 \), then the abort of \( t_1 \) may force the abort of \( t_2 \), resulting in a cascading abort. To avoid such cascading aborts, we require that the commit dependency graph be acyclic.
restrict that if $t_1 \rightarrow t_2$ and $t_1$ is retriable, then $t_2$ is not submitted until $t_1$ commits.

We assume that the GTM maintains state information for each subtransaction in the commit dependency graph of its flexible transaction. This state may be one of the following cases:

- **inactive**: if the subtransaction has not started its execution.
- **active**: if the subtransaction has started but not completed its execution.
- **to_be_committed**: if the subtransaction has completed its execution.
- **committed**: if the subtransaction has committed.
- **aborted**: if the subtransaction has aborted.
- **committed-reversed**: if both the subtransaction and its compensating subtransaction have committed.

Figure 4 shows the state transition diagram for a subtransaction. In this figure, $op$ is an operation submitted by the GTM, and $ack$ is an acknowledgement from a local database server.

The execution and dynamic commitment of a flexible transaction is coordinated via message exchanges. The message exchanges that occur during the commitment of a single subtransaction $t$ are shown in Figure 5. In this figure, items in italics are messages that are passed. Other items indicate code that is executed when the message is received. Time proceeds from top to bottom.

When the last operation in a subtransaction is acknowledged, the GTM updates $t$’s subtrans-
Figure 5: Message passing sequences for committed and aborted subtransactions. (a) Commit succeeds in the local database. (b) Commit fails in the local database, subtransaction is not retrievable. (c) First commit fails in the local database, subtransaction is retrievable.
action state to "to_be_committed." The GTM can thus commit the subtransaction as soon as it is consistent with the maintenance of semi-atomicity. Once all the subtransactions that precede \( t \) in the commit dependency graph have committed, the GTM sends a \( commit(t) \) message to \( t \)'s server, which then tries to commit \( t \) in the local database. This commit is either successful (case (a) in Figure 5) or unsuccessful (case (b) or (c)). If it is successful, the final state of the subtransaction in the GTM is \( committed \). Otherwise, if it is retrievable, the subtransaction is retried by the local database server until it eventually commits, and the final state of the subtransaction becomes \( committed \). Otherwise, it is set to \( aborted \).

When an \( ack\_abort(t) \) is received, the GTM seeks an alternate \( \langle \rangle \)-rpo of the flexible transaction. Such an alternative may be obtained directly, or one may be constructed by finding alternatives of more remote predecessors and trying from an earlier point. In this paper, we assume that the next alternative attempted by the GTM requires as little backtracking as possible and that the alternatives must be tried in preference order. The following algorithm describes the backtracking method:

**Algorithm 1 (Backtracking Algorithm)** Input: \( t \), the aborted subtransaction.

1. If \( t \) is a switching point, find the switching set that includes \( t \) and has the fewest committed successors. Otherwise, find the closest predecessor \( t_1 \) in the \( \langle \rangle \)-rpo that is a switching point and then find the switching set inclusive of \( t_1 \) that has the fewest committed successors.

2. If a switching set is found, abort all subtransactions in the switching set, as well as all their successors that are in the active state. Compensate for all subtransactions in the switching set and their successors that are in the committed state. Attempt the next untried alternative in preference order.

3. If no switching set is found, abort the flexible transaction by aborting all subtransactions in the \( \langle \rangle \)-rpo that are in the active state. Compensate for all subtransactions in the \( \langle \rangle \)-rpo that are in the committed state.

When a subtransaction must be compensated for, the compensating subtransaction may need to be retried until it actually succeeds. When the compensating subtransaction commits, the state of the original subtransaction in the GTM changes to committed-reversed. Thus, the execution of the compensating subtransaction \( ct \) of subtransaction \( t \) is as shown in Figure 6.

When all subtransactions in the currently executing \( \langle \rangle \)-rpo of the flexible transaction enter the committed state, the flexible transaction commits. All its subtransactions which are not in the committed \( \langle \rangle \)-rpo should be in either inactive, aborted, or committed-reversed states. Otherwise, there is no committed \( \langle \rangle \)-rpo, and the flexible transaction aborts. In this case, all its subtransactions should be in either the inactive, aborted, or committed-reversed state.
(a) Compensating for a Committed Subtransaction during Backtracking

Global Transaction Manager

\[ \text{compensate}(t, ct) \]

state(t) \leftarrow \text{committed-reversed}

Local Database Server

\[ \text{execute } ct \text{ and commit in local database} \]

if aborted, retry the compensating subtransaction until it commits

\[ \text{ack\_commit}(ct) \]

(b) Aborting a Running Subtransaction during Backtracking

Global Transaction Manager

\[ \text{abort}(t) \]

state(t) \leftarrow \text{ aborted}

Local Database Server

\[ \text{abort } t \text{ in local database} \]

\[ \text{ack\_abort}(t) \]

Figure 6: Message passing sequences used during backtracking: (a) when compensation is required; (b) when aborting is possible.

5.3 Persistent Transmission in the HDDBS Environment

Finally, a system failure must never result in message loss when the proposed commit protocol is used. We consider the two major system failures, namely, communication and site failures [BHIG87]. Communication failures may cause a message sent through the communication link to be lost, and site failures may result in the loss of the knowledge of the commitment status. For example, if a site failure occurs after subtransaction \( t_j \) commits but before this commitment information is forwarded to its GTM server, the commitment record may be lost.

Hsu and Silberschatz [HIS91] propose a persistent transmission protocol for transaction processing in a distributed database environment. They implement persistent transmission through message logging, guaranteeing that a communication failure will not result in the loss of a message sent through the communication link. To prevent the loss of the commitment information pertaining to a subtransaction, a commit record is logged during the subtransaction commit. This is accomplished by revising the traditional commit operation to force the logging of this information prior to commitment.

In the HDDBS environment, their approach to avoiding message loss due to communication failures is still applicable. However, under the constraints of local design autonomy, it may not be possible to revise commit operations in local database systems. We therefore propose an approach which enforces persistent transmission through logging at the GTM server level. Each GTM server maintains its own commit-status table in stable storage with one entry for each submitted subtransaction at the site. Each entry contains the following information:

- The subtransaction identifier,
Figure 7: Data required in the GTM_server and local database for persistent transmission.

- The object identifier of its commit status record. The commit status record is stored independently in the local database, and has status *ACTIVE* if the subtransaction is still executing or *COMMIT* if it commits.

Figure 7 illustrates the relationship between the commit-status table and its commit status records. Note that the commit-status table may also be stored in the local database.

When GTM_server_i receives the submission of t_j, it first creates an entry for t_j in the commit-status table, with the object identifier of its commit status record set to *NULL*. The execution of t_j then proceeds as follows:

- Begin the subtransaction.

- Create a commit status record in the local database, with value *ACTIVE*.
  (At this point, GTM_server_i independently stores the identifier of the commit status record returned by the LDBS in the subtransaction’s entry in the commit-status table.)

- Execute the body of subtransaction t_j.

- Set the value in the commit status record to *COMMIT*.

- Commit the subtransaction.

When t_j commits, the LDBS sends the commit acknowledgement to GTM_server_i, which then notifies the GTM of this successful commit. In the case of a communication failure, GTM_server_i can still ascertain that the subtransaction committed by noting that its commit status is set to *COMMIT*, and retransmit the commit notification. In the case of a site failure between the time the LDBS
completed the commitment and the time the commit notification was conveyed to GTM\_server\_i, GTM\_server\_i can ascertain that the subtransaction committed in the same manner.

If the LDBS aborts the subtransaction, it notifies GTM\_server\_i, which in turn must notify the GTM of the abort. Note that, regardless of the timing of the abort, one of the effects of aborting the subtransaction is that its commit status record will be deleted from the database, though its object identifier will still be stored stably by GTM\_server\_i in its commit-status table. In the case of a communication failure, GTM\_server\_i can still ascertain that the subtransaction aborted by querying the local database and noting that the object identifier of its commit status record no longer is valid. Thus, the notification message can be retransmitted until it succeeds in reaching the GTM. In the case of a site failure between the time the LDBS completed the abort and the time it sends the abort notification to GTM\_server\_i, GTM\_server\_i can ascertain that the subtransaction aborted in the same manner.

In the case of a communication failure during the execution of a subtransaction, the GTM can ascertain the status of the subtransaction once the communication link is re-established by checking its commit status record. Once the GTM acknowledges receipt of a subtransaction commit or abort notification, GTM\_server\_i can delete the commit status record of the subtransaction from the commit-status table and remove the the commit status record from the local database.

In the case of a site failure, GTM\_server\_i can also notify the GTM of the abort status of each subtransaction in the commit-status table by checking for the validity of the object identifier of its commit status record in the local database. If the object identifier of the commit-status record is NULL, the subtransaction never started. If the commit-status record is invalid, the subtransaction aborted. If the commit-status record is valid and is set to COMMIT, the subtransaction committed.

### 5.4 Discussion

The flexible transaction commit protocol preserves the semi-atomicity of flexible transactions in a HDDBS. This is shown as follows:

**Theorem 2** The flexible transaction commit protocol ensures that the semi-atomicity of flexible transactions is preserved, provided that the flexible transactions are well-formed and have acyclic commit dependency graphs.

**Proof:** Let $\mathcal{T} = \{ (T_i, \prec), i = 1, ..., k \}$ be a well-formed flexible transaction and $CDG(\mathcal{T})$ is acyclic. Let $(T_i, \prec)$ of $\mathcal{T}$ be currently in execution. Because the GTM issues commits for the subtransactions of $T_i$ in the order that is defined in $CDG(\mathcal{T})$, the global schedule is commit dependency preserving. When the GTM receives an $ack\_abort(t)$, there are two cases to be considered. If $t$ is either compensatable or normal or a critical point, then any partial effects of $(T_i, \prec)$ can be undone. Based upon the backtracking algorithm, the GTM either changes the execution of $\mathcal{T}$ to an
alternative $\prec$-partial order $(T_j, \prec)$ by properly undoing some of the partial effects of $(T_i, \prec)$ or if no alternative exists, it undoes all partial effects of $(T_i, \prec)$ and aborts the entire flexible transaction. If $t$ is abnormal, then, since $T$ is well-formed, the backtracking algorithm automatically changes the execution of $T$ to an alternative $\prec$-partial order $(T_j, \prec)$. In either case, the backtracking algorithm guarantees that no partial effects of $(T_i, \prec)$ remain permanent. When the GTM receives commit acknowledgement for all subtransactions of $T_i$, then $T$ is determined to be committed. At this moment, no partial effects of any other $\prec$-partial orders remain permanent in the HDDBS.\footnote{Note that $F$-recoverability is also guaranteed in the global schedule.} Hence, the execution of $T$ satisfies Definition 3.

This commit protocol also is effective with local database systems that accept only transactions that are executed and committed as a unit. However, in such instances, the submission and execution of a subtransaction on the local database must occur at the time it is expected to commit. Thus, the inactive, active, and to.be.committed states are all collapsed into a single to.be.committed state, and the transition to the committed state occurs when the commit acknowledgment for the entire subtransaction is received. If the transaction aborts, then the transition is made to the aborted state.

In the absence of failures, the proposed protocol requires $2n$ messages that need to be exchanged to reach a commit decision and at least 6 rounds as compared to either $3n$ messages and 3 rounds needed by the 2PC protocol [BHIG87] or $2n$ messages and exactly 6 rounds needed by the GTM commit protocol proposed in [MRKS92], where $n$ is the number of local sites.

Both the GTM commit protocol described in [MRKS92] and the flexible transaction commit protocol outlined here prevent the severe blocking of local transactions that may be caused by the 2PC protocol. The flexible transaction commit protocol permits each subtransaction to commit dynamically without waiting for other subtransactions of the same flexible transaction to complete their execution. Thus, it generates even less blocking of local transactions than does the GTM commit protocol developed in [MRKS92]. However, to prevent cascading aborts, value dependencies may cause the execution of some subtransactions to be delayed by retrievable subtransactions upon which they are value dependent, a complication not present with the GTM commit protocol. As a result, the GTM commit protocol may achieve better performance for the more restricted class of input global transactions to which it applies.

6 Effects of Concurrency Control on Commitment Ordering

In this section, we briefly discuss the effect of concurrency control on the commitment ordering of flexible transactions.

In this study, we use the notion of global serializability [ZE93] as the applicable correctness
criterion. Global serializability states that an execution containing both the global (or flexible) transactions in the HDDBS environment and the transactions that run independently on local sites is conflict-equivalent to some serial execution of those transactions. If two flexible transactions execute on the same set of LDBSs, it is necessary that their subtransactions are serialized in a consistent order on the LDBSs for global serializability to be maintained. Thus, a correct concurrent execution must enforce serialization ordering constraints among the subtransactions of different flexible transactions.

Let \( t_{i1}, \ldots, t_{in} \) and \( t_{j1}, \ldots, t_{jn} \) be the subtransactions in some \( \prec \)-pos of flexible transactions \( T_i \) and \( T_j \), respectively, at local sites \( LS_1 \) to \( LS_n \). That is, \( t_{ik} \) and \( t_{jk} \) both are executed at local site \( LS_k \). Let \( \rightarrow_s \) be a serialization ordering, so \( t_{ik} \rightarrow_s t_{jk} \) means that \( t_{ik} \)'s execution must be serialized before that of \( t_{jk} \) on the LDBS it is executed. We then have the following observation:

**Observation 2** Let \( T_1, \ldots, T_l \) be a set of flexible transactions and LDBSs maintain serializability on the local transactions and subtransactions executed at their sites. Global serializability is preserved if there exists a permutation \( T_{i1}, \ldots, T_{in} \) of \( T_1, \ldots, T_l \) such that, for any \( T_{ij}, T_{ij'} \in \{ T_{i1}, \ldots, T_{in} \} \), \( t_{ij_1 k} \rightarrow_s t_{ij_2 k} \) for all local sites \( LS_k \), where \( j_1 < j_2 \) and \( 1 \leq k \leq n \).

This observation has been made by many researchers [GPZ86]. Several solutions have been proposed to enforce the serializable execution of global transactions, including forced local conflicts [GRS91]. However, if a commit protocol uses compensation and retrial, several issues may arise with respect to serializability. Consider the case of enforcing \( t_{ik} \rightarrow_s t_{jk} \) on local site \( LS_k \). We can assume that serialization order for any two subtransactions can be enforced by determining a serialization point (or serialization event) [ED90, MRB+92b, Pu88] for each subtransaction and ensuring that the subtransactions execute their serialization points in the order defined by \( \rightarrow_s \). This serialization point depends on the concurrency control protocol of the LDBS. For instance, if the local database uses 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. When local conflicts are forced, each subtransaction updates a shared data item at the local site. The order of the updates is the serialization order.

We will now consider some scenarios that may arise if compensation or retrial are allowed. First, let us consider the following example:

**Example 4** Two flexible transactions \( T_1 \) and \( T_2 \) at local site \( LS_k \) are executed as follows:

- \( t_{1k} \) executes its serialization point.
- \( t_{2k} \) executes its serialization point, enforcing \( t_{1k} \rightarrow_s t_{2k} \).

\(^9\)See [BHGG87] for the definition of conflict equivalence.
• $t_{2k}$ reads some data item that was written by $t_{1k}$.

• $t_{1k}$, which is compensatable, commits.

• $T_1$ makes a global decision to abort.

• A compensating subtransaction is executed for $t_{1k}$.

In this example, all of the effects of $T_1$ are eventually removed from the execution, including the effects of $t_{1k}$. Global serializability is preserved, because flexible transaction $T_1$ was aborted and correctly compensated for, even though its effects were read by flexible transaction $T_2$. Furthermore, $T_2$ proceeds based on the reading of the data item that was updated by $t_{1k}$ and may be inconsistent.

The concept of isolation of recovery [LKS91b, LKS91a] states that a global transaction should be unaffected by both aborted and committed subtransactions of other global transactions. In addition, it has also been observed [BZ94] that, if a global transaction is affected by a committed subtransaction which later must 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. Note that the effects of the compensated subtransactions on local transactions need not be considered, under the condition that the execution of subtransaction transfers local database from one consistent state to another. This leads us to the following observation:

**Observation 3** A necessary condition for isolation of recovery to be maintained in an execution containing concurrent flexible transactions is that, for each subtransaction $t_{ik}$ of $T_i$ at a given local site, conflicting subtransactions that are subsequently serialized at the local site must not execute their serialization points until either $T_i$ makes a global decision to commit or the compensating subtransaction for $t_{ik}$ has executed its serialization point.

In addition, the compensating subtransactions may still unilaterally abort and need be retried. Some conflicting subtransactions may have to be aborted to ensure that they are serialized after the retried compensating subtransactions. To avoid such undesirable cascading aborts, we must delay the execution of the serialization point of any conflicting subtransactions until one of the following conditions holds for the subtransaction:

• The subtransaction’s flexible transaction has made a decision to commit.

• The subtransaction’s compensating subtransaction has committed.

The following example is also problematic:

**Example 5** Two flexible transactions $T_1$ and $T_2$ at local site $LS_k$ are executed as follows:
- \( t_{1k} \) executes its serialization point.

- \( t_{2k} \) executes its serialization point, enforcing \( t_{1k} \rightarrow_s t_{2k} \).

- \( T_i \) makes a global decision to commit.

- \( t_{1k} \), which is retriable, unilaterally aborts.

- \( t_{1k} \) is resubmitted and re-executes its serialization point. Now we have \( t_{2k} \rightarrow_s t_{1k} \), which contradicts the order \( t_{1k} \rightarrow_s t_{2k} \) that was enforced.

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

Again, cascading aborts are not an acceptable means of regaining consistency. We make the following observation:

**Observation 4** 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.

We see then that using either compensation or retrial to regain consistency leads to blocking. These issues hold regardless of whether or not the global transaction is flexible. Furthermore, when using flexible transactions, switching to an alternate \( \prec \)-rpo can also create problems. Let us consider an alternative in flexible transaction \( T_i \). Let \( t_{1i} \prec t_{1i} \prec t_{1m} \) be the preferred \( \prec \)-rpo, and let \( t_{1j} \prec t_{1j} \) 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 \( T_i \) should be serialized before \( T_2 \):

**Example 6** Two flexible transactions \( T_1 \) and \( T_2 \) are given as follows:

- \( t_{1i} \) executes its serialization point.

- \( t_{2j} \) executes its serialization point.

- \( T_i \) makes a global decision to commit.

- \( t_{1i} \), which is retriable, unilaterally aborts.

- \( T_1 \) chooses another 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 \( T_1 \rightarrow_s T_2 \) that was enforced.
At this point, \( t_2 \) must be aborted to maintain global serializability.

To avoid cascading aborts in this situation, we have the following observation:

**Observation 5** A necessary condition for avoiding cascading aborts when an alternate <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.

These complicated orders of precedence can be tracked by maintaining for each flexible transaction a bag of all the potential subtransactions to be executed. It is initialized with the additive union of all of the alternatives. Once an alternative fails, its subtransactions are removed from the bag. We can avoid cascading aborts by prohibiting a subtransaction \( t \) from executing its serialization point until there are no subtransactions at the same local site in any of the bags for flexible transactions that precede \( t \) in the global serialization order.

Observations 3, 4, and 5 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 3 and 4 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 abort.

Observation 5 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. Following these observations, we see that the execution of a flexible transaction may be greatly affected by the concurrent execution of other flexible transactions. The investigation of issues related to the concurrent execution of a set of flexible transactions merits more detailed research. In particular, we believe that experimentation is needed to compare and contrast the behavior of different transaction models in a real HDDBS setting.
7 Application in Traditional Distributed Database Systems

Flexible transactions can be used to increase the failure resiliency of global transactions in the traditional distributed database environment, where local autonomy is not assumed and all transactions are submitted to the GTM. The proposed theory and protocol are readily applicable to preserving the semi-atomicity of flexible transactions. Since in this context there are no local transactions submitted separately to local sites, some of the restrictions required in the flexible commit protocol can be dropped. In particular, compensating subtransactions may be subject to fewer restrictions in such an environment. A compensating subtransaction $ct_i$ need not be independent of the transactions that execute between $t_i$ and $ct_i$ as long as the GTM ensures that the transactions that are executed between $t_i$ and its compensating subtransaction $ct_i$ are commutative with $ct_i$ [KLS90]. Thus, the scope of global transactions that can be specified in the traditional distributed database environment can be larger than that can be specified in the HDDBS environment. Also, if subtransaction $t_i$ of a flexible transaction commutes [BR92] with subtransaction $t_j$ of another flexible transaction that is serialized later, the blocking on $t_j$ caused by either compensation, re-trial, or alternatives in the HDDBS environment need not be considered. Thus, delays introduced to avoid cascading aborts in the HDDBS environment between these two flexible transactions may be reduced.

The 2PC protocol can also be used to ensure either all subtransactions in one $\prec$-rpo of the flexible transaction commit or none of its subtransactions commit. In this case once a single $\prec$-rpo completes its execution, a 2PC variant is started. The result of the first phase may be that some subtransactions are in the prepare-to-commit state and others have aborted. In this situation, rather than aborting the flexible transaction, we attempt to switch to an alternate $\prec$-rpo which has a prefix that contains only subtransactions in the prepare-to-commit state. If such a $\prec$-rpo exists, the new subtransactions are executed and attempt to enter the prepare-to-commit state. Once a $\prec$-rpo has all its subtransactions in the prepare-to-commit state, a global commit decision can be made. If no alternate $\prec$-rpos exist, a global abort decision is made. Semi-atomicity is therefore maintained.

The blocking effects caused by both the flexible commit protocol and the 2PC variant must be carefully compared to determine whether the former protocol offers any advantages over the latter. Blocking may take the form either of intra-blocking between the subtransactions of a single flexible transaction or inter-blocking between different flexible transactions. Intra-blocking among the subtransactions of a flexible transaction in the flexible commit protocol is caused by commit dependencies. Intra-blocking in the 2PC variant occurs because all subtransactions are blocked until they all enter the commit-to-prepare state. The intra-blocking in the flexible commit protocol is minor when compared with the intra-blocking in the 2PC variant. It is crucial to recognize that the intra-blocking which arises in the flexible commit protocol occurs generally between two individual
subtransactions, while the intra-blocking found in the 2PC variant involves all of the subtransactions in one <rpo of the flexible transaction and lasts until the several rounds of communication are complete.

Inter-blocking in the 2PC variant occurs because conflicting subtransactions cannot execute until the current flexible transaction both completes its execution and its commit protocol. Inter-blocking caused by conflicting compensatable and retrievable subtransactions in the flexible commit protocol is not found with the 2PC variant. However, inter-blocking in the flexible commit protocol caused by conflicting compensatable subtransactions only lasts until the commit decision is made, which is often before all subtransactions of the flexible transaction complete their execution. In the situations where inter-blocking in the flexible commit protocol is caused by conflicting retrievable subtransactions, the 2PC variant would have switched the flexible transaction to an alternative or aborted if no alternative is defined. Thus, the flexible commit protocol introduces a delay in the current subtransaction where the 2PC variant would have introduced a switch or a global abort.

Inter-blocking also may arise as a result of conflicting alternative subtransactions for untried alternatives in both commit protocols. These delays are between individual subtransactions and are in direct proportion to the number of potentially conflicting alternative subtransactions. We do not expect most flexible transactions to specify large numbers of alternative subtransactions, and thus we do not expect these delays to be prohibitive.

8 Conclusions

Global transaction management in an error-prone HDDBS environment has been recognized as a yet-to-be resolved issue in those instances in which the component local database systems do not support prepare-to-commit states [SSU91]. We have advanced a theory which facilitates the preservation of semi-atomicity, a form of atomicity related to semantic atomicity in heterogeneous databases that is weaker in that it encompasses the ability of the flexible transaction to execute alternatives. We feel that flexible transactions are more appropriate to an HDDBS environment. Our theory includes definitions of a fundamental transaction model and of the semi-atomicity property of flexible transactions, and classifies the flexible transactions that can be executed in the presence of failures.

The preservation of the weaker property of semi-atomicity renders flexible transactions more resilient to failures than traditional global transactions. This property is preserved through subtransaction compensation and retry, as well as the ability to switch to alternative executions. Local prepare-to-commit states are therefore not required. The construction of committable flexible transactions that are executable in the error-prone HDDBS environment demonstrates that the flexible transaction model indeed enhances the scope of global transaction management beyond that offered
by the traditional global transaction model.

This discussion has primarily focused on the issues relevant to the consistency and reliability of a single flexible transaction. Questions related to the concurrency control of flexible transactions have been raised only when germane to their effects on the commitment of flexible transactions. A more complete exploration of the concurrency control of flexible transactions should be pursued. Any such study must carefully analyze the effects of compensation, retrial, and alternatives on concurrency control, especially when global serializability is used as the correctness criterion.

Given the likely future preeminence of the HDBDBS environment, the evaluation of the proposed commit protocol assumes a high degree of urgency and importance. This evaluation must experimentally demonstrate the performance of the proposed protocol and comparatively examine the efficiencies of the various commitment protocols.

References


