Automating Compensation in a Multidatabase [†]

Marian H. Nodine and Stanley B. Zdonik
Brown University
Providence, RI 02906

Abstract

Compensation is the process by which a committed transaction in a database is undone by running the semantic inverse of that transaction on the database. Compensation has been proposed as a technique for undoing committed work in various situations where strict atomicity cannot be maintained [GS87, MR91].

In this paper, we discuss compensation in long-running multidatabase transactions. We define the step approach *to integrating local database schemas into a mulitdatabase. In the step approach, each local database is encapsulated by a set of procedures (*steps*). Steps can be grouped into atomic global transactions. Each step also has an associated compensating step, which is called if the compensating transaction is run.*

We examine two areas of multidatabase transaction management where compensation is required. The first is implementing compensation as a recovery technique when an open nested transaction is aborted. The second is in backing out the effects of an atomic multidatabase transaction when some local database transaction commits before a global abort decision is made.

1 Introduction

In situations where a transaction is required to commit some of its work before it actually makes a decision itself whether to commit or abort, compensation is required to semantically undo that work if that transaction eventually aborts. Longer transactions release resources early because holding them will severely impact the amount of concurrency in a database [Gra81]. In multidatabase environments, local transaction autonomy requires that the local databases commit their parts of a global transaction before a coordinated commit decision is finalized.

In this paper, we describe our approach for implementing compensation in the Mongrel prototype multidatabase system. We propose a multidatabase architecture and a way of implementing global transactions on that architecture that enables us to automatically generate compensating transactions. In this architecture, each local database presents a library of *steps* that the multidatabase can execute on it. If all local databases are only accessed through their step libraries, we can define multidatabase transactions flexibly while maintaining the ability to automatically compute their compensating transactions as needed.

The step model is a new variant of the standard restricted access multidatabase model [HM85], in that each step is not in itself a complete transaction. Rather, steps in different local databases can be grouped into a single atomic global transaction. At the other extreme, the unrestricted model for multidatabase access allows arbitrary global queries and updates on the local databases. Thus, the transaction definer must explicitly specify how to compensate. Our model provides some of the flexibility of the unrestricted model while not requiring the transaction definer to explicitly consider compensation.

Each step in a database's Step Library is statically associated with its compensating step. We assume that the actual call to execute a compensating step can be derived from the original step function name, arguments, and return value. A compensating transaction executes these steps, in the inverse of the order the original steps were executed. We specify a two-level logging mechanism for storing the information needed for compensation.

We also examine compensation in two areas of multidatabase transaction management. In each area, we provide an underlying theory concerning compensatability, or the ability to decide when compensation will unconditionally succeed (given a failure-free environment). We provide algorithms for using the step/compensating step pairing in the Step Library to generate the compensating transactions in a way that maintains consistency in the database.

Open nested transaction models [GS87, Nod93b] break up a long transaction into a set of shorter ones. If the longer transaction is aborted, compensation is required to undo any short transactions that have already committed. For open nested transactions, we show how to use compensation to undo this committed work. We give approaches for determining when compensation is straightforward to execute, and when it is not. We can place restrictions on the execution and commitment of atomic transactions within the open nested transaction to facilitate smooth operation of an open nested transaction abort. When exact compensation is not possible, we give alternative approaches.

The second area in which we examine compensation is when emulating a two-phase commit of a atomic global transaction. Compensation is required during two-phase commit when a local commit decision is finalized before a global abort decision is made [MR91].

[†]This work was supported in part by ARPA order 8225, ONR grant N00014-91-J-4052, and in part by an IBM Graduate Fellowship.

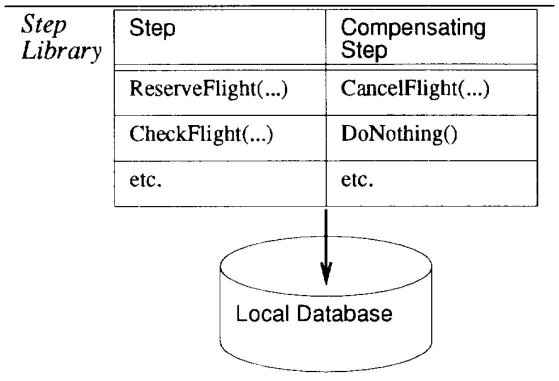

Figure 1: The local database interface to the multidatabase.

We show how to ensure that compensation eventually succeeds in this case, thus ensuring semantic atomicity of global transactions. In some cases, the multidatabase must interfere with the local databases to ensure that compensation will always succeed when required.

2 The Step Approach

In the Mongrel prototype multidatabase, we take a *step approach* to integrating the information in the local databases into the multidatabase. In this, each local database defines an interface of function calls, or *steps* that it is willing to provide for use by the multidatabase. These steps are collected together in the local database's own *Step Library*. This approach to integrating heterogeneous systems is similar to that proposed by Rusinkiewicz *et al.* [ROEL90] to support their Distributed Operation Language.

Figure 1 shows a local database and its Step Library. Note that each step in the Step Library is explicitly paired with its compensating step. As stated earlier, compensation is a necessary function in a multidatabase that attempts to preserve the autonomy of its local databases. This explicit pairing allows the multidatabase to determine, at the time a global transaction's step is executed on a local database, what the appropriate compensating step is. Since the step knows exactly what is run on the local database, it also encodes the ability to compute the values required for each parameter of the compensating step call. Thus, the step can log this information as it executes to ensure that it is available when compensation is required.

The step approach differs from the two existing approaches for integrating local databases, which are the restricted and the unrestricted approaches. The restricted approach (e.g., [HM85, AGS87]) states that each time a multidatabase application accesses a local database, it runs a complete, predefined transaction on that local database. The step approach differs from the restricted approach in that each step is not itself a separate local database transaction. Rather, different steps on the same local database can be grouped into a single atomic global subtransaction. Steps on different local databases can be grouped into a single atomic global transaction. Thus, the step model is more flexible than the restricted model.

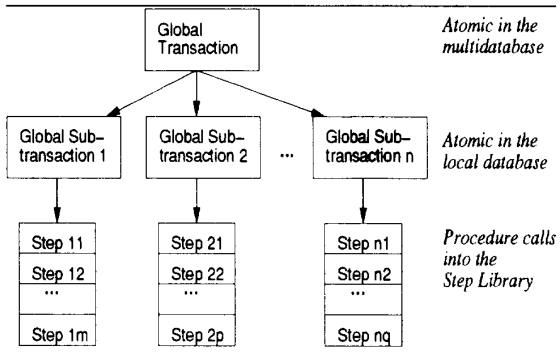

Figure 2: A global transaction execution.

The unrestricted model (e.g. [CBE93]) allows global transactions in the multidatabase to run arbitrary sets of queries in the local databases. The step model differs from the unrestricted model in that it only allows access to the local databases through the defined steps. This restriction is useful because it provides a uniform way of accessing the information in the multidatabase, independent of any local database's data manipulation language. It also allows the Step Library definer to associate a compensating step with each step, and to define how to derive the compensating step call during the execution of a step. When a global transaction must be semantically undone, these compensating step calls are used to generate compensating global transactions. This frees the user from having to specify compensating subtransactions himself.

3 Global Transaction Execution

In the step model, different steps on different local databases can be combined into a single atomic global transaction. Within a global transaction, the steps on each local database are grouped into global subtransactions. Thus, the global transactions encompass multiple global subtransactions and span multiple local databases. However, to help ensure that the global transactions are atomic, no two global subtransactions of the same transaction execute on the same local database. Each global subtransaction begins at the time the global transaction starts executing on the local database, and commits atomically with its global transaction commit. This ensures that no other transactions on any local database see partial results of the global transaction.

Figure 2 shows the basic structure of a global transaction execution. It also summarizes the atomicity properties of the different parts. The global transactions in a multidatabase are assumed to be atomic or

	Atomicity	Semantic Atomicity
No Effects	1. Before 2. After Abort	1. Before 2. After Abort (when nothing committed)
All Effects	1. After Commit	1. After Commit (when commit is final) 2. After Commit (when eventually undone)
Undone Effects	1. Never	1. After Abort (when something committed)

Figure 3: Summary of atomicity properties.

at least semantically atomic. If the global transaction is atomic, this means that any other transaction will see either no effects of the atomic transaction or all of its effects. It sees no effects if it precedes the atomic transaction or if the atomic transaction was aborted. It sees all the effects if it follows the atomic transaction and the atomic transaction was committed.

Semantic atomicity differs from atomicity in that it may be possible for another transaction to see the effects of a semantically atomic transaction even if that transaction eventually is aborted. With a semantically atomic global transaction, some or all of the subtransactions of the global transaction may be committed in the local database, and other transactions may read the effects of the committed subtransactions. Then, if the global transaction is eventually "aborted", those effects will have to be semantically undone. A syntactic undo is not possible because the data items written by the semantically atomic transaction may have been accessed and/or overwritten by some other transaction between the time the semantically atomic transaction committed its effects on the local database and the time the "abort" is processed. Instead, each subtransaction has an associated compensating subtransaction, which semantically undoes the effects of the original subtransaction. For example, a subtransaction that withdraws money from a bank account may have a compensating subtransaction that deposits that money back into the account. When a global transaction is "aborted", the compensating subtransactions for each of its committed subtransactions must be executed.

Figure 3 summarizes the visibility properties for atomic and semantically atomic global transactions.

4 Compensating Transactions

We can now show how a compensating transaction can be automatically executed given information derived during the original global transaction execution. First, we assume that all compensating transactions are atomic or semantically atomic, and have a similar structure to the global transactions. While less restrictive models of compensation have been described in certain contexts (e.g., [LKS91]), making compensating transactions atomic enables us to generate compensating transactions that are robust in any situation where compensation is needed. It also means that the commit protocol for the compensating transactions is identical to the one used for global transactions.

Now, let us consider the compensating steps that form a compensating subtransaction on a single local database. The original global subtransaction executed a sequence of steps, and then committed. Associated with each of these steps is a compensating step that semantically undoes the effects of the original step. Just like with a state-based undo, we need to back out of each step in the inverse of the order the original steps were executed. So the compensating subtransaction execution algorithm is as follows:

Algorithm 4.1 *Input: The identifier of the subtransaction to be compensated for.*

1. *Begin a transaction on the local database.*
2. *From the log that records compensating subtransaction information, extract the compensating step calls using the input subtransaction identifier.*
3. *Execute the compensating steps in the inverse of the order in which they are logged. The sequence in which they are logged reflects the execution order of the steps in the original transaction.*
4. *Commit the global subtransaction as directed by the global level of the multidatabase. A coordinated global commit such as two-phase commit is necessary to preserve the atomicity of the compensating transaction.*

This algorithm requires a log to maintain local-level compensation information for the subtransactions that execute as a part of the global transactions in the multidatabase. We maintain this information separately for each local database. The information logged for compensation during a global subtransaction execution includes the following:

1. The identifier of the global subtransaction on the local database.
2. The compensating step for each step that was executed in the global subtransactions, computed from the explicit pairing in the Step Library.
3. The arguments for all of the compensating steps, derived from the step's arguments, its return values, and possibly also from the results of supplementary queries concerning the local database state at the time the step executed.

Given that we can log enough information to be able to deduce the compensating subtransaction definition from the original subtransaction execution, the question remains of how to combine the compensating subtransactions into a global transaction. If we consider that the purpose of the compensating transaction is to undo (inverse) the effects of the original global transaction semantically, we can see that the different undos on the different local databases should be independent. In the original execution, subtransactions form dependencies because one subtransaction reads information from its local database, and that information is used as an argument to a step call in a different local database. However, the compensating transaction is working with changes that are fully specified before it executes, and therefore there cannot be dependencies among the subtransactions in a compensating transaction.

We therefore assume that the subtransactions of a compensating transaction can be executed concurrently (because they are independent). Given this, the log of global transaction compensation information at the global level of the multidatabase must contain only the local database and local identifier for each global subtransaction that comprised the original global transaction.

The algorithm for executing a compensating transaction is as follows:

Algorithm 4.2 *Input: The identifier of the global transaction to be compensated for.*

1. *Begin the global transaction.*
2. *For each committed subtransaction of the global transaction, do the following:*
   (a) *Using the global transaction identifier, retrieve from the global transaction log the subtransaction identifier and the local database that the subtransaction ran on.*
   (b) *Submit a request to the local database to run the compensating subtransaction, given the specified subtransaction identifier.*
3. *When all local databases have indicated that the global subtransaction execution succeeded, initiate a coordinated atomic commit protocol.*

Note here that we assume that the compensating transaction will commit. If some global subtransaction does not succeed, that subtransaction must be retried until it eventually succeeds. This requirement, called *persistence of compensation*, is recognized as necessary for maintaining complete database consistency in an environment that allows a task to partially commit before a comprehensive commit decision can be made (see, for instance [KLS90]). Naturally, we attempt to manipulate situations where compensation is needed to ensure that it succeeds the first time.

5 Compensatability

In this section we discuss *compensatability*, which defines the conditions under which compensation is possible, and when compensation can be expected to proceed smoothly. The compensatability property characterizes situations where persistence of compensation is certain, and where it is problematic to enforce. Whether or not a step, global subtransaction, or transaction is compensatable can be derived based on a simple syntactic analysis of the compensation code.

Definition 5.1 (Compensatable) *A step, global subtransaction, or global transaction is* compensatable *if its compensating subtransaction will always succeed, regardless of the state of the local database.*

Examples of compensatable steps include a read-only step (which has a null compensating subtransaction), and a withdrawal (because the success of the compensating deposit does not depend on the state of the particular bank account).

We define a step, global subtransaction, or transaction as *provisionally* compensatable if the success of the compensating step, global subtransaction, or global transaction does depend on the state of the underlying database. For example, a deposit is provisionally compensatable; as long as no one withdraws the money in the interim it can be compensated for.

Given that a step, global subtransaction, or global transaction has compensating code written or derived for it, we can also specify the following theorem that relates its structure to its compensatability:

Theorem 1 *A step, global subtransaction, or transaction is compensatable if compensating code can be specified and executed, and there are no conditional branches in the compensating code that depend on information read from the database.*

Intuitively, we see that this follows from the fact that, even if some other transaction serializes between the global transaction and its compensating transaction on some local database, that transaction cannot change that local database in a way that would affect the execution of the compensating transaction.

Theorem 2 *A step, global subtransaction, or global transaction is provisionally compensatable if compensating code can be specified, even if it has conditional branches in the compensating code that depend on information read from the database.*

Note that we assume here that if a compensating subtransaction can be executed, it has permission to execute as well. Also, we assume that code that has no conditional branches that depend on the database state also calls no subroutines that have conditional branches that depend on the database state. Proofs for these theorems are found in [Nod93a].

5.1 Computing the Properties

In the Mongrel multidatabase model, each step in a local database's step library has a compensating step defined for it. If the code for the step has conditional branches based on information in the local database, the step is labeled at step definition time as provisionally compensatable. Otherwise, it is labeled as compensatable. This label is associated with the step and the compensating step in the Step Library itself.

Given that all the steps executed as a part of a global subtransaction are labeled, we can then compute whether the global subtransaction itself is compensatable based on the following (trivial) theorem:

Theorem 3 *If all of the steps in a global subtransaction are compensatable, the global subtransaction is compensatable. Otherwise, the global subtransaction is provisionally compensatable.*

Similarly, we can compute the compensatability of a global transaction using the following theorem:

Theorem 4 *If all of the global subtransactions in a global transaction are compensatable, then the global transaction is compensatable. Otherwise, the global transaction is provisionally compensatable.*

Proofs for these theorems are given in [Nod93a].

5.2 Implications

Given that compensating code can be written for a global transaction or subtransaction, its compensatability properties define exactly when the persistence of compensation requirement becomes a problem. If a global transaction is compensatable, then only the failure of a process, system, or network can prevent the compensating transaction from completing immediately. Retrying the compensating transaction in the presence of such a failure should succeed as soon as the multidatabase becomes failure-free, so persistence of compensation is guaranteed.

If a global transaction is only provisionally compensatable, then problems can occur when persistence of compensation is also required. This is because a change in the state of some data item after the original transaction committed could prevent the compensating transaction from succeeding. The success of the compensating transaction would then depend entirely on some third agent changing the state of a specific data item in the local database, so the conditional test that depends on that value would succeed.

In this second situation, alternative measures can be taken to ensure that persistence of compensation holds, or at least does not create a problem. Examples of these alternative measures include the following:

1. Find some means to ensure that critical data items are not changed until we can be sure that compensation can no longer be invoked (for some specific purpose).

2. Avoid using compensation.

3. Find alternative means of compensation if the primary method fails.

In the following two sections we discuss compensation in open nested transactions and compensation during two-phase commit. We propose using different combinations of the above methods to facilitate the eventual success of compensation in the specific context.

6 Open Nested Transactions

An *open nested transaction* is a partial order of atomic global transactions, each of which runs over multiple local databases and commits when it completes its execution. An open nested transaction can be viewed as a long, non-atomic transaction that is executed as a partially-ordered set of independent, atomic pieces (the atomic global transactions).

In this section, we examine issues of how open nested transactions can use compensation to enforce their semantic atomicity. We first examine how dependencies form. We then discuss the basic compensation algorithm. We give some constraints on the original execution that allow for more efficient recoverability. We also give an approach to recovery when direct compensation fails.

6.1 Global Transaction Dependencies

The execution of an open nested transaction is specified as a partial order because we assume that unrelated global transactions can execute concurrently. We assume that the open nested transaction starts out single-threaded, but can fork new threads as it executes. A set of threads can join if none of them has an active global transaction.

Different kinds of dependencies form between global transactions in an open nested transaction. An *execution dependency* forms between two global transactions T_i and T_j under one of the following conditions:

1. T_i precedes T_j in the same execution thread.

2. T_j is the first transaction in a forked thread, and T_i is the last transaction that began on the forking thread before the fork occurred.

3. T_j is the first transaction after a join operation, and T_i is the last transaction that committed on one of the joining threads.

Execution dependencies reflect the specified order of global transactions in the open nested transaction execution.

A second form of dependency can occur among global transactions if the open nested transaction maintains some form of internal state. We call this type of dependency a *state dependency*. A state dependency occurs among two global transactions in an open nested transaction when they conflict on an internal variable. In this case, even though the two global transactions are specified in a way that allows them to be executed concurrently, dependencies form in the order in which conflicting operations on the internal variables are resolved.

Both state and execution dependencies can be computed and logged as the open nested transaction executes.

Formally, we define the partial order of global transactions in an open nested transaction execution O as $O = (T_O, <^T_O)$, where

1. T_O is the set of global transactions that are executed as a part of O, and

2. $<^T_O$ is a partial order of those global transactions as executed, where $T_a <^T_O T_b$ if T_b has an execution dependency or a state dependency on T_a.

Aborting an open nested transaction means aborting the uncommitted, but active global transactions

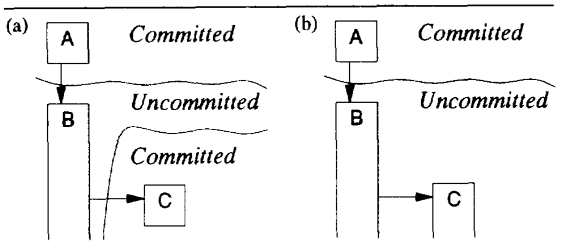

Figure 4: (a) Uncommitted Global Transaction B has forked a second execution thread, and has a successor committed Global Transaction C. (b) Same execution, but in this case Global Transaction C is restricted to not commit until Global Transaction B commits.

and then compensating for the committed global transactions. We follow principles derived from the Sagas work [GS87], and compensate for the committed global transactions in the inverse of the partial order defined by $<_O^T$ to maintain consistency in the multidatabase during compensation. This ensures, for instance, that the conflicts on the internal state are resolved appropriately during compensation. Following this order is also necessary to correctly inverse the effects of the open nested transaction on a data item in some local database that is accessed more than once by the open nested transaction.

Thus, we have the following algorithm for open nested transaction abort:

Algorithm 6.1 (Open Nested Abort)
Input: $<_O^T$.

1. *Abort all active, uncommitted global transactions in the open nested transaction.*
2. *Compensate for each committed global transaction once all of its successors in* $<_O^T$ *are either aborted or compensated for.*

6.2 Global Transaction Commits

In the multidatabase, we assume that for correctness the different global transactions in an open nested transaction are serialized in an order consistent with that open nested transaction's partial order $<_O^T$.

Ease of compensation is affected by the order in which those global transactions commit. When the new thread executes its first global transaction, care must be taken to ensure that that global transaction does not commit before the global transaction in the forking thread that precedes it. This could happen if the forking thread is in the middle of executing a global transaction when the fork statement is executed. If the global transaction in the forked thread were to commit first, we would have the situation such as the one shown in Figure 4(a). In this situation, if the forking global transaction B were to abort, then global transaction C would also have to be semantically undone. To avoid this cascading, we constrain the commit order of the global transactions to be consistent with the order imposed by its state and execution dependencies.

Thus, we define the following property for an open nested transaction:

Definition 6.1 (Open Nested Recoverability) *The* open nested recoverability property *for an open nested transaction states that no global transaction can commit until all global transactions that precede it in* $<_O^T$ *have committed.*

If this property is enforced, then in no case can the abort of some uncommitted global transaction cause some later global transaction in $<_O^T$ to be compensated for. This means that compensation can occur regularly and smoothly, with the execution of the compensating transactions following the inverse of the order $<_O^T$, as defined in the previous section.

6.3 Sloppy Compensation

Despite the precautions outlined in the previous section to ease compensation, the requirement for persistence of compensation does not hold in a general open nested transaction environment. Open nested transactions periodically release the resources they hold, allowing other database transactions to interleave with their own operation. This release of control by the open nested transaction allows other transactions to modify information required for compensation.

Consider as an example, a travel agent making a reservation for a customer on a Pan Am flight. The customer pays for the flight and gets his ticket. Then, before the customer actually takes the flight, Pan Am goes bankrupt. The customer may no longer be able to go on the trip, but fully canceling the Pan Am flight and getting a refund is impossible. The customer has relinquished control over his money.[1]

We propose that compensating steps be specified as prioritized sets of guarded code blocks. The top priority code block exactly semantically undoes the original step, provided no intervening transactions have changed the values of any data items that it reads. Lower-priority code blocks could take alternative, automated steps if certain conditions hold true ("sloppy compensation"). These blocks may not semantically undo the transaction at all, but instead take other options to back out of the original work as much as possible. For example, one compensating code block for buying the ticket could specify that if the airline was bankrupt, a letter should be written to claim the ticket money.

7 Emulated Two-Phase Commit

In this section, we discuss compensation and how it fits into emulating an atomic or semantically atomic two-phase commit protocol in a multidatabase. We shift our focus somewhat because the primary concern here is how to enforce the atomicity or at least semantic atomicity during the commit process itself.

[1]Ultimately, compensation may succeed once the bankruptcy proceedings have completed. However, this process is not timely with respect to getting the money back for use on the trip.

We begin by briefly summarizing existing work in two-phase commitment in distributed databases, and in emulating two-phase commitment in multidatabases. We then provide some theoretical background for understanding two-phase commit, including a discussion of when and how atomic and semantically atomic commit are possible. We describe a two-phase commit algorithm that uses compensation to ensure that semantic atomicity is preserved during global transaction commit.

7.1 Two-Phase Commit (Review)

The two-phase commit protocol [BHG87] is the standard protocol used to coordinate commitment in a homogeneous, distributed database. In two-phase commit, a commit coordinator polls the participant (local) databases of the transaction being committed to determine whether or not they are certain to commit. Each participant checks whether its part of the transaction executed successfully. If so, it returns a "yes" vote and enters the prepared state. If not, it returns a "no" vote and aborts its subtransaction.

Once the commit coordinator receives votes from all of its participants, it makes a global decision. If all votes were "yes", the global decision is "commit"; otherwise, it is "abort". The commit coordinator forwards the global decision to the participants, who commit or abort their subtransactions from the prepared state, according to the global decision.

The prepared state essentially allows the commit process to be split into two parts in the local database. First, it makes a local decision that can potentially be revoked (if the local decision is to commit). Later, it possibly revokes the decision. However, once the global decision is made it becomes irrevocable.

In a multidatabase, local database autonomy assumptions constrain us, in that we cannot ensure that the local databases support a prepared state that can be used by the multidatabase's commit coordinator (a "visible" prepared state). Thus, we are restricted to taking one of the following two tactics for making a global decision (from [MR91]):

1. Local commit before global decision: the local database commits its subtransaction when it forwards the "yes" vote to the commit coordinator. If the global decision is "abort", a compensating subtransaction must be run on the local database to undo the committed effects.

2. Global commit before local commit: the local database makes an educated guess concerning its vote, but commits nothing. If the global decision is "commit", the local database commits the subtransaction. If the actual local commit fails, the subtransaction must be redone.

In this work, we choose a local commit before global decision strategy.

7.2 Subtransaction Information Flow

To understand the commit properties of global transactions, we first need to examine information flow between the subtransactions in the global transaction being committed. Let the ***subtransaction information flow (SIF)*** be a relationship $(T, <_F)$, where

1. T is the set of subtransactions in the global transaction, and

2. $T_a <_F T_b$ if subtransaction T_a and T_b in T contain conflicting steps S_a and S_b, respectively, and S_b reads information local to the global transaction that S_a wrote.

For the purposes of this discussion we assume that the subtransaction information flow is acyclic.[2] We also assume that there is a begin marker B in the subtransaction information flow, and that every subtransaction is a descendant of B. Similarly, we have a commit marker C, where every subtransaction has C as its descendant.

Let G be the directed graph representation of $<_F$, with nodes for each subtransaction, as well as for B and C. Given the assumptions in the previous paragraph, we can see that G is a directed acyclic graph with a single source at B and a single sink at C. Define B and C to be compensatable. Label each subtransaction in the global transaction with a c if it is always compensatable or if it executes on a local database that supports a visible prepared state, and a c_p if it is only provisionally compensatable.

We assume that any subtransaction in a local database that supports a visible prepared state can be considered to be compensatable. This is because we can emulate compensatability using the visible prepared state. When a local commit decision is made, we put the subtransaction in the prepared state. This guarantees that the subtransaction will eventually commit. Once the global decision is made, the subtransaction can commit or abort from the prepared state.

Now, we make the following definition with respect to the graph G:

Definition 7.1 (Max Compensatable Subgraph) *The* maximum compensatable subgraph *is the largest subgraph of G containing the node B in which all the nodes have c in their label (are compensatable) and whose ancestors are also all in the maximum compensatable subgraph.*

Figure 5 illustrates the maximum compensatable subgraph for an example global transaction.

Global transaction commit processing is constrained somewhat by the order $<_F$. This is because if we ever decided to retry a subtransaction, all subtransactions that follow it in $<_F$ would also have to be retried to ensure that they read the correct information.[3] Starting at the source, intuitively we want to commit each global transaction once all of its predecessors in $<_F$ have committed. Since the maximum compensatable subgraph contains the largest portion of the

[2]Further details concerning cycles in the subtransaction information flow can be found in [Nod93a].

[3]Forward recovery techniques are not discussed in this paper, but a detailed approach can be found in [Nod93a].

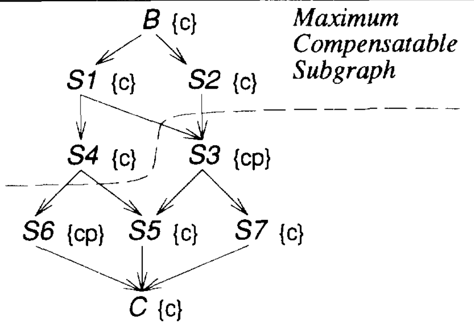

Figure 5: Example maximum compensatable subgraph.

global transaction execution that can always be compensated for, it also represents the set of global subtransactions that can always be committed locally before a global decision is made. Thus, we have the intuition behind following theorem, proved in [Nod93b]:

Theorem 5 *The maximum compensatable subgraph contains the maximum set of subtransactions that can unconditionally be locally committed before a global decision is made, if semantic atomicity is to be maintained.*

7.3 Algorithm

Given this background, we can now examine two-phase commit in a multidatabase. From Theorem 5, we see that if a global transaction has at most one subtransaction that is not in the maximum compensatable subgraph, we can define a simple strategy that, given the labeled subtransaction information flow graph can commit its global transaction. First, commit all of the the subtransactions of the global transaction in the maximum compensatable subgraph. If any return a "no" vote, compensate for the committed subtransactions and make a global "abort" decision. Then, a global decision can be made. If a single uncommitted subtransaction remains, the global decision is made during the process of committing it, according to the outcome of that specific local commit. Otherwise, a global "commit" decision is made. This algorithm succeeds only because there is at most one global subtransaction where persistence of compensation is questionable.

This algorithm is impractical in that we cannot guarantee that all global transactions will have a subtransaction information flow graph that fits its criteria. Unfortunately, if more than one global subtransaction is not in the maximum compensatable subgraph, then all such subtransactions must cooperate to make a unanimous global decision. This cannot be done if the local databases are fully autonomous. In the following section, we describe ways that others have proposed to help the multidatabase make a coordinated decision while violating the autonomy of the local databases as little as possible.

7.3.1 Blocking Protocols

Because autonomous local databases in a multidatabase do not necessarily provide a visible prepared state, the multidatabase itself must provide other protocols (*blocking protocols*) to block other transactions from interfering with its subtransactions during voting and commit phases of the commit process. These protocols intervene in the local databases for a period to block the execution of conflicting transactions, and thus all violate local database autonomy assumptions.

Both active and passive blocking protocols have been proposed for multidatabases. An active blocking protocol is one that needs to be "turned on" and "turned off". A passive blocking protocol implements a low-level filtering at all times. The most widely-proposed passive blocking protocols partition the multidatabase according to what can be accessed in what ways by the global and local transactions [BST90, MRKS91, MRB$^+$92].

Several examples of active blocking protocols have been proposed. For example, in Hydro [PRR91], no transactions are allowed to pass operations to the local database while a global commit decision is being made. This type of blocking protocol requires that the multidatabase to be able to delay the local transactions. Mullen*et al.* [MJS] propose a reservation protocol where the global transactions use additional data in the database to reserve the resources they need. Denied local updates, proposed for the 2PC agent method [WV90], require modifying the local databases to prevent updates while a global commit decision is being made.

7.3.2 The Generic Algorithm

We can now formalize a generic algorithm that can successfully implement an emulated two-phase commit in a multidatabase, provided that all local databases support a visible prepared state or a blocking protocol:

Algorithm 7.1 (Emulated Two-Phase Commit)
Input: Labeled subtransaction information flow graph for the global transaction.

1. *Commit all of the global subtransactions in the maximum compensatable subgraph. If one aborts, make a global abort decision, compensate for the committed subtransactions, and return abort. Note that the commits can all proceed concurrently, because the compensating subtransactions should be independent of one another.*

2. *For the local databases that participated in the global transaction and whose subtransactions did not commit in the first step: If one or fewer subtransactions remains to commit, go to the next step directly. Otherwise, for each such local database, if it supports an active blocking protocol, turn blocking on.*

3. *Commit all of the remaining subtransactions. If any of these abort (a "no" decision is returned),*

make a global decision to abort. Compensate for all of the committed transactions, then turn off the blocking protocols on all local databases where it was turned on in the previous step.

4. *Make a global decision to commit.*

Provided that only one global transaction is in the commit process at a time, this algorithm succeeds and maintains the semantic atomicity of the global transaction. It succeeds because the blocking protocol effectively protects the data that the provisionally compensatable transactions depend on from changing.

This algorithm only preserves semantic atomicity because, at least for the transactions committed in the first step, their effects become visible and are accessible by other transactions on the local database. Even if the blocking protocol is run on all local databases for the entire duration of the algorithm, this algorithm still does maintain full atomicity. This is because there may be small side effects in the original global transaction that cannot be compensated for easily, but whose presence is deemed not to be significant by the application designer. If no such side effects are present, then full atomicity can be guaranteed.[4]

8 Related Work

Open nested transaction models such as Sagas [GS87], Flex Transactions [ELLR90], and ConTracts [WR91] use compensation during recovery. Compensation was first defined in detail by Garcia-Molina and Salem [GS87]. In Sagas, each transaction has a corresponding compensating transaction. A correct execution of a Saga looks like a sequence of transactions: $T_1, T_2, \cdots, T_n$. However, the Saga may also be undone in midstream, causing the compensating transactions to be run in the inverse order: $T_1, T_2, \cdots, T_i, C_i, \cdots, C_2, C_1$. This approach requires a Saga's code and compensating code to be stored persistently.

Multilevel transactions also require compensation-based recovery strategies [WHBM90]. Multilevel transaction recovery has been implemented in the DASDBS project. [Wei91] notes several other approaches for multilevel transactions, with different strategies for maintaining (semantic) undo and redo information at the various levels.

Nodine's thesis [Nod93a] expands on compensation and other approaches to maintaining semantic atomicity during two-phase commit. This thesis presents a solid underlying theory concerning global transaction properties and how they affect the ability to enforce different kinds of atomicity in a multidatabase.

The Optimistic Commit Protocol [LKS91] also supports semantic atomicity. However, Levy *et al.* observe that the compensating subtransactions do not need to be coordinated as strongly because they are largely independent.

Korth *et al.* [KLS90] studied compensation with respect to their entitywise 2PL correctness specification for transaction execution. This paper provides a good characterization of some of the stickier problems encountered when implementing any scheme that uses compensation.

9 Conclusions

As more and more complex applications rely on databases to persistently store information, new and advanced transaction models are emerging. Multidatabases provide one situation where traditional guarantees such as atomicity cannot be easily enforced for all transactions, simply because local database autonomy is incompatible with coordinating multidatabase decisions. Also, emerging classes of applications such as work flow applications map naturally to an open nested transaction model, and may access multiple databases.

Compensation is used to recover database consistency semantically when committed work must be undone. Work gets committed early in open nested transactions because their length precludes using techniques such as locking for holding the resources the transaction touches. Open nesting encourages concurrency among long-lived applications, but also relaxes atomicity guarantees. Work also gets committed early during a multidatabase two-phase commit, because local database autonomy precludes a commit coordinator from being able to ensure that a commit of a subtransaction will succeed without actually committing that subtransaction and releasing its resources. If the global decision is ultimately to abort, compensation must be used to recover database consistency.

We have discussed issues of compensation in both of these situations, within the context of our multidatabase transaction model. This model is based on a notion of *steps*, or procedures that encapsulate the actual information in the local databases. Global transactions in the multidatabase can only access the local databases via the steps. The step interface also provides adequate information to determine if a step is easily compensatable, or likely to cause trouble in specific situations. It also encodes additional processing in the function definition for the step to be able to derive and log information that would be required to compensate for the step. At the multidatabase level, we can also independently compute dependency information to ensure that compensation maintains the consistency of the multidatabase at all times. Thus, because of the step library interface, the multidatabase can use the information logged from an open nested transaction execution to generate compensating transactions and subtransactions when needed.

Problems with compensation occur when interference from other transactions prevents compensation from succeeding. We provided a theory for determining when this interference is possible. In open nested transactions, this interference cannot be prevented, but we gave approaches to dealing with such conflicts when they occur. In emulating two-phase commit, we presented protocols to prevent such interference.

This multidatabase architecture and automated compensation approach have been successfully implemented in our prototype multidatabase, Mongrel.

[4]The presence or absence of side effects is a property of the step code, and can thus be controlled by the step code definer.

References

[AGS87] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. *IEEE Data Engineering Bulletin*, 10(3), 1987.

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

[BST90] Yuri Breitbart, Avi Silberschatz, and Glenn R. Thompson. Reliable transaction management in a multidatabase system. In *SIGMOD Proceedings*, pages 215–224, 1990.

[CBE93] Jiansan Chen, Omran Bukhres, and Ahmed K. Elmagarmid. IPL: A multidatabase transaction specification language. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *VLDB Proceedings*, pages 507–518, 1990.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB Proceedings*, pages 144–154, 1981.

[GS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD Proceedings*, pages 249–259. ACM, 1987.

[HM85] Dennis Heimbinger and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Automation Systems*, 3(3):253–278, July 1985.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *VLDB Proceedings*, pages 95–106, 1990.

[LKS91] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An optimistic commit protocol for distributed transaction management. In *1991 ACM SIGMOD Proceedings*, pages 88–97, 1991.

[MJS] James G. Mullen, Jin Jing, and Jamshid Sharif-Askary. Reservation commitment and its use in multidatabase systems. (submitted to DEXA 1993).

[MR91] Peter Muth and Thomas G. Rakow. Atomic commitment for integrated database systems. In *1991 Data Engineering Proceedings*, pages 296–304, 1991.

[MRB$^+$92] Sharad Mehrotra, Rajeev Rastogi, Yuri Breitbart, Henry F. Korth, and Avi Silberschatz. Ensuring transaction atomicity in multidatabase systems. In *PODS 1992 Proceedings*, pages 164–175, 1992.

[MRKS91] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, 1991.

[Nod93a] Marian H. Nodine. *Interactions: Multidatabase Support for Planning Applications.* PhD thesis, Brown University, 1993. (Also Brown University Computer Scinece Department Technical Report CS-93-17).

[Nod93b] Marian H. Nodine. Supporting long-running tasks on an evolving multidatabase using Interactions and events. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 125–132, 1993.

[PRR91] William Perrizo, Joseph Rajkumar, and Prabhu Ram. Hydro: A heterogeneous distributed database system. In *SIGMOD Proceedings*, pages 32–39, 1991.

[ROEL90] M. Rusinkiewicz, S. Ostermann, A. Elmagarmid, and K. Loa. The distributed operation language for specifying multisystem applications. In *Proceedings of the First International Conference on System Integration*, pages 337–345, April 1990.

[Wei91] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[WHBM90] Gerhard Weikum, Christof Hasse, Peter Broessler, and Peter Muth. Multi-level recovery. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 109–123, 1990.

[WR91] Helmut Waechter and Andreas Reuter. The ConTract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan-Kauffman, 1991.

[WV90] Antoni Wolski and Jari Veijalainen. 2PC agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase. In *Proceedings of PARBASE*, pages 321–330, 1990.