A FORMAL APPROACH TO RECOVERY BY COMPENSATING TRANSACTIONS*

Henry F. Korth, Eliezer Levy,
and Abraham Silberschatz

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-90-14 May 1990

* Work partially supported by a grant from the IBM Corporation, TARP grant 4355 and NSF grant IRI-8805215.

A Formal Approach to Recovery by Compensating Transactions [*]

Henry F. Korth
Eliezer Levy
Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Abstract

Compensating transactions are intended to handle situations where it is required to undo either committed or uncommitted transactions that affect other transactions, without resorting to cascading aborts. This stands in sharp contrast to the standard approach to transaction recovery where cascading aborts are avoided by requiring transactions to read only committed data, and where committed transactions are treated as permanent and irreversible. We argue that this standard approach to recovery is not suitable for a wide range of advanced database applications, in particular those applications that incorporate long-duration or nested transactions. We show how compensating transactions can be effectively used to handle these types of applications. We present a model that allows the definition of a variety of types of correct compensation. These types of compensation range from traditional undo, at one extreme, to application-dependent, special-purpose compensating transactions, at the other extreme.

1 Introduction

The concept of transaction atomicity is the cornerstone of today's transaction management systems. Atomicity requires that an aborted transaction will have no effect on the state of the database. The most common method for achieving this is to maintain a recovery log and provide the *undo*(T_i) operation which restores the data items updated by T_i to the value they had just prior to the execution of T_i. However, if some other transaction, T_j, has read data values written by T_i, undoing T_i is not sufficient. The (indirect) effects of T_i must be removed by aborting T_j. Aborting the affected transaction may trigger further aborts. This undesirable phenomenon, called *cascading aborts*, can result in uncontrollably many transactions being forced to abort because some other transaction happened to abort.

Since a committed transaction, by definition, cannot abort, it is required that if transaction T_j reads the values of data items written by transaction T_i, then T_j does not commit before T_i commits. A system that ensures this property is said to be *recoverable* [BHG87]. One way of avoiding cascading aborts and ensuring recoverability is to prohibit transactions from reading *uncommitted* data values — those produced by transactions that have

*Work partially supported by a grant from the IBM Corporation, TARP grant 4355 and NSF grant IRI-8805215.

not committed yet. This principle has formed the basis for standard recovery in most contemporary database systems.

Unfortunately, there is a large range of database applications for which the standard recovery approach is excessively restrictive and even not appropriate. The common denominator of such applications is the need to allow transactions to read ***uncommitted data*** values.

In general, as indicated by Gray [Gra81], early exposure of uncommitted data is essential in the realm of long-duration and/or nested transactions. Applications incorporating transactions of that nature cannot be accommodated by the standard recovery approach since their executions entail cascading aborts and some of them are even non-recoverable. For example, consider a database system that uses locks to enforce concurrency control. Externalization of uncommitted data amounts to releasing exclusive locks before the end of the transaction. When long-duration transactions are used, early release of locks is a necessity, since otherwise the delay they incur by retaining locks for long periods of time is intolerable [Gra81, GMS87]. Shortening the time periods transactions hold locks is a worthwhile goal by its own right. Releasing locks early enhances concurrency, thereby increasing the system's throughput.

Nested transactions are often used in applications where a set of subtransactions are assigned a single coherent task that requires interaction and cooperation among the subtransactions [HR87]. For instance, nested transactions may represent long-duration design activities [KKB88, KLMP84]. Since design efforts are usually collaborative in nature, it is essential in certain cases for a design subtransaction to see uncommitted data written by another design subtransaction.

An additional restriction imposed by standard recovery is the inability to undo an already committed transaction. Suppose that a transaction was committed "erroneously." By committed erroneously, we mean that from the system's point of view there was nothing wrong with the committed transaction. However, external reasons, that were discovered later, rendered the decision to commit the transaction erroneous. Under the standard recovery approach there is no support for undoing such transactions. The need for a mechanism which support undoing of committed transactions is established in [Lag88].

This paper presents the method of a ***compensating transactions*** as a recovery mechanism in applications where exposure of uncommitted data and undoing of committed transactions must be facilitated. Our goals are to develop a better understanding of what compensation really is, when it is possible to employ it, and what the implications are on correctness of executions when compensation is used.

The remainder of this paper is organized as follows. We give an informal introduction to compensating transactions in Section 2. In Section 3, we present a transaction model suitable for the study of compensation. We then use this model in Section 4 to define criteria for "reasonable" compensation. After illustrating our definitions with examples in Section 5, we examine the theoretical consequences of our model in Section 6. Implementation issues are discussed in Section 7, and related work is described in Section 8.

2 An Overview of Compensating Transactions

When the updates of a (committed or uncommitted) transaction T are read by some other transaction, we say that T has been *externalized*. The sole purpose of *compensation* is to handle situations where we want to undo an externalized transaction T. We refer to T as the *compensated-for transaction*. The transactions that are affected by (reading) the data values written by T are referred to as *dependent transactions*. The key point of our recovery paradigm is that we would like to *leave the effects of the dependent transactions intact* when undoing the compensated-for transaction. Thus, compensation broadens the scope of recovery to encompass undoing of externalized transactions without resorting to cascading aborts. Moreover, since the compensated-for transaction may be a committed one, compensation allows the undoing of committed transactions, which stands in sharp contrast to the standard approach to recovery.

Similarly to the use of the traditional transaction undo as the means for automatically undoing a non-externalized transaction, we propose compensation as the method for automatically undoing externalized transactions. In both cases, the goal is to restore the database to a state that would have been obtained had the undone transaction, T, never taken place. Traditional transaction undo applies to the simple special case where T is not externalized; hence, all that needs to be done is to restore the database to the state just prior to T's beginning. Compensation is applicable in the more general case where T may be externalized; hence, it does not guarantee erasing all of T's direct and indirect effects.

We propose the notion of *compensating transactions* as the vehicle for carrying out compensation. A compensating transaction has the fundamental properties of a transaction along with some special characteristics. It appears atomic to concurrently executing transactions (that is, transactions do not observe partially compensated states); it conforms to consistency constraints; and its effects are durable. However, a compensating transaction is a very special type of transaction. Under certain circumstances, it is required to *restore* consistency, rather than merely preserve it. It is durable in the strong sense that once a decision is made to initiate compensation, the compensating transaction must complete (since it does not make any sense to abort it). There are other special characteristics. Above all, a compensating transaction does not exist by its own right; it is always regarded within the context of the compensated-for transaction. It is always executed after the compensated-for transaction. Its actions are derivative of the actions of the compensated-for transaction and the nature of the execution so far. In some situations, the actions of a compensating transaction can be extracted automatically from the program of the compensated-for transaction, the current state of the database, and the current state of the log. In other situations, it is the system programmer's responsibility to pre-define a compensating transaction. In either case, what a compensating transaction needs to accomplish is a function of the execution of the compensated-for transaction. A user may invoke a compensating transaction explicitly (in order to cancel the effects of an externalized transaction) in the same manner as regular transactions are invoked. Alternatively, a compensating transaction can be invoked internally by the recovery manager as a consequence of the abortion of an externalized uncommitted transaction.

A mundane example taken from "real life" exemplifies some of the characteristics of compensation. Consider a database system that deals with transactions that represent purchasing of goods. Consider the act of a customer returning goods after they have been sold. The compensated-for transaction in that case is a particular purchase, and the compensating transaction encompasses the activity caused by the cancellation of the purchase. The compensating transaction is bound to the compensated-for transaction by the details of the particular sale (e.g., price, method of payment, date of purchase). The effects of purchasing transaction might have been externalized in different ways. For instance, it might have triggered a dependent transaction that issued an order to the supplier in an attempt to replenish the inventory of the sold goods. Furthermore, the customer might have been added to the store's mailing list as a result of that particular sale. The actual compensation depends on the relevant policy. For example, the customer may be given store credit, or full refund. Whether to cancel the order from the supplier and whether to retain the customer in the mailing list are other application-dependent issues with which the compensating transaction must deal.

It is crucial to understand that compensation is a semantically-rich recovery activity. Defining what a compensating transaction needs to achieve concretely, depends heavily on the semantics of the application at hand. This observation will guide us in the construction of a model that can capture semantics of applications easily. The model allows the definition of a variety of types of correct compensation. These types of compensation range from traditional undo, at one extreme, to application-dependent, special-purpose compensating transactions, at the other extreme. We return to study the characteristics of compensating transactions in more depth after we introduce the model in the next section.

3 A Transaction Model

In the classical transaction model [Pap86, BHG87] transactions are viewed as sequences of read and write operations that map consistent database states to consistent states when executed in isolation. The correctness criterion of this model is called *serializability*. A concurrent execution of a set of transactions is represented as an interleaved sequence of read and write operations, and is said to be serializable if it is equivalent to a serial (non-concurrent) execution.

This approach poses severe limitations on the use of compensation. First, sequences of uninterpreted reads and writes are of little use when the semantically-rich activity of compensation is considered. Second, serializability isolates a transaction from concurrently executing transactions, whereas compensation is relevant especially when transactions are allowed to interact and cooperate.

In Section 3.1 we describe a semantics-based transaction model, and in Section 3.2 we define what executions are in our model and discuss the corresponding correctness criteria.

3.1 Transactions and Programs

A transaction is a sequence of operations that are generated as a result of the execution of some program. The exact sequence that the program generates depends on the database state "seen" by the program [Gra80]. In the classical transaction model only the sequences are dealt with, whereas the programs are abstracted and are of little use. Given a concurrent execution of a set of transactions (i.e., an interleaved sequence of operations) compensation for one of the transactions, T, can be modeled as an attempt to cancel the operations of T while leaving the rest of the sequence intact. The validity of what remains from that execution is now in serious doubt, since originally transactions read data items updated by T and acted accordingly, whereas now T's operations have vanished but its indirect impact on its dependent transactions is still apparent. The only formal way to examine a compensated execution is by comparing it to a hypothetical execution that does not include the compensated-for transaction. We use the comparison of the compensated execution with the hypothetical execution that does not include the compensated-for transaction, as a key criterion in our exposition. However, generating this hypothetical execution and studying it requires the introduction the *transactions' programs* which are, therefore, indispensable for our purposes.

A *transaction program* can be defined in any high-level programming language. We restrict our attention to Pascal-like assignments and conditional statements. Programs have local (i.e., private) variables. In order to support the private (i.e., non-database) state space of programs we define the concept of an *augmented state*. The augmented state space is the database state space unioned with the private state spaces of the transactions' programs. The provision of an augmented state allows one to treat reading and updating the database state in a similar manner. Reading the database state is translated to an update of the augmented state, thereby modeling the storage of the value read in a local variable.

Thus, a *database*, denoted as db, is a set of data *entities*. The *augmented database*, denoted as adb, is a set of entities that is a superset of the database; that is, $db \subset adb$. An entity in the set $(adb - db)$ is called a *private entity*. Entities have identifying *names* and corresponding *values*. A *state* is a mapping of entity names to entity values. We distinguish between the *database state* and the state of the augmented database, which is referred to as the ***augmented state.*** We use the notation $S(e)$, to denote the value of entity e in a state S. The symbols S and e (and their primed versions, S', e', etc.) are used, hereafter, to denote a state and an entity, respectively.

Another deviation from the classical transaction model is the use of semantically-richer operations instead of the primitive read and write. For example, object-oriented databases use abstract data type techniques to define data objects which support specific and rather complex operations (see, e.g., [ZM90]). Having semantically-richer operations allows refining the notion of conflicting versus commutative operations [BR87, Wei88]. That is, it is possible to examine whether two operations commute (i.e., do not conflict) and hence can be executed concurrently. By contrast, in the classical model, there is not much scope for such considerations since a write operation conflicts with any other operation on the same entity. Moreover, as was stressed earlier, compensation is made possible only when the semantics of the database application at hand are explicit. Therefore, one of the features of our semantics-based model is the ability to deal with a rich set of complex operations.

An ***operation*** is a function from augmented states to augmented states that is restricted as follows:

- An operation updates at most one entity (either a private or a database entity) ;
- an operation reads at most one ***database*** entity, but it may read an arbitrary number of private entities;
- an operation can both update and read only the same database entity.

We use the following shorthand notation for a single operation f: $e_0 \ := \ f(e_1, \ldots, e_k)$. We say that f ***updates*** entity e_0, and ***reads*** entities $e_1, \ldots, e_k$. The ***arguments*** of an operation are all the entities it reads. There are two special termination operations, ***commit***, and ***abort***, that have no effect on the augmented state. Operations are assumed to be executed ***atomically.***

It is implicitly assumed that all the arguments of an operation are meaningful; that is, a change in their value cause a change in the value computed by the operation. Observe that in defining operations, we have two contradictory goals. First, we want to capture complex and semantically-rich operations. Second, we want to adhere to principles of practicality and use operations as primitives of fine granularity. The operations in our model reconcile these two contradictory goals. On the one hand, operations are functions from augmented states to augmented states, thereby giving the flexibility to define complex operations. On the other hand, the mappings are restricted so that at most one database entity is accessed in the same operation, thereby making it feasible to allow atomic execution of an operation. Although only one database entity may be accessed by an operation, as many local variables (i.e., private entities) as needed may be used as arguments for the mapping associated with the operation. Having private entities as arguments to operations adds more semantics to operations. Having functions for operations allows us to conveniently compose operations by functional composition, thereby making sequences of operations functions too.

Entities in our scheme can be of arbitrary granularity and complexity. Examples for entities are pages of data and index files, or abstract data types like stacks and queues. Accordingly, sample reading operations are read a page, stack top, is-empty queue, and sample updating operations are write a page, stack push and pop, and insertion into a queue. Notice that the above sample reading operations only read the database state. On the other hand, a blind write only updates the database state but does not read it. Finally, assuming integer-type entities, an increment operation both reads and updates an integer entity.

We are in a position now to introduce the notion of a transaction as a program. A *transaction program* is a sequence of *program statements*, each of which is either:

- An operation.
- A *condition* of the form: **if** b **then** $S1$ **else** $S2$, where $S1$ and $S2$ are sequences of program statements and b is a predicate that mentions only private entities and constants.

We impose the the following restrictions on the operations that are specified in the statements:

- The set of private entities is *partitioned* among the transaction programs. An operation in a program cannot read nor update a private entity that is not in its own partition;
- private entities are updated only once;
- An operation reads a private entity only after another operation has updated that entity.

Example 1. Consider the following sets of entities: $db = \{a, b, c\}$, and $adb = db \cup \{u, v, w\}$, and the following two transaction programs, T_1 and T_2:

```
T1     begin                                      |  T2     begin
             u:=a;                                |               a:=0;
             v:=b;                                |               b:=1
             if u > v then c:= f(c,v)             |         end
                       else  begin                |
                                  w:=c;           |
                                  b:= g(u,w)      |
                             end                  |
       end                                        |
```

Observe that operation f both updates and reads entity c. T_2 demonstrates operations that read no entities. ◇

3.2 Histories and Correctness

The use of serializability as the correctness criterion for applications that demand interaction and cooperation among possibly long-duration transactions was questioned by the work on concurrency control in [KS88, KKB88]. Since we target compensation as a recovery mechanism for these kind of applications, our model does not rely on serializability as the correctness notion.

We use the framework for alternative correctness notions set forth in [KS88]. Explicit *input* and *output predicates* over the database state are associated with transactions. The input predicate is a pre-condition of transaction execution and must hold on the state that the transaction reads. The output condition is a post-condition which the transaction guarantees on the database state at the end of the transaction provided that there is no concurrency and the database state seen by the transaction satisfies the input condition. Thus, as in the standard model, transactions are assumed to be generated by correct programs, and responsibility for correct concurrent execution lies with the concurrency control protocol.

Observe that the input and output predicates are excellent means for capturing the semantics of a database system. We use the convention that predicates (and hence semantics) can be associated with a set of transactions, similarly to the way predicates are associated with nested transactions in [KS88]. That is, a set of transactions is supposed to collectively establish some desirable property, or complete a coherent task. This convention is most useful in domains where long-duration nested transactions are assigned a single complex task.

We do not elaborate on the generation of interleaved or concurrent executions of sets of transaction programs, since this is not central to understanding our results. However, the notion of a history, the result of this interleaving, is a central concept in our model. A *history* is a sequence of operations, defining both a total order among the operations, as well as a function from augmented states to augmented states that is the functional composition of the operations. We use the notation $X =< f_1, \ldots, f_n >$ to denote a history X in which operation f_i precedes f_{i+1}, $1 \leq i < n$. Alternatively, we use the functional composition symbol '$\circ$' to compose operations as functions. That is, $X = f_1 \circ \ldots \circ f_n$ denotes the function from augmented states to augmented states defined by the same history X. We use the upper case letters at the end of the alphabet, e.g., X, Y, Z, to denote both the sequence and the function a history defines.

The equivalence symbol '$\equiv$' is used to denote equality of histories as functions. That is, if X and Y are histories, then $X \equiv Y$ means that for all augmented states S, $X(S) = Y(S)$. Observe that since histories and operations alike are functions, the function composition symbol '$\circ$' is used to compose histories as well as operations.

When a (concurrent) execution of a set of transaction programs A is initiated on a state S and generates a history X, we say that X is a *history of* A whose *initial state* is S.

Example 2. Consider the transaction program T_1 of Example 1. Since T_1 has a condition there are two histories, X and Y, which can be generated when T_1 is executed in isolation. We list the histories as sequences of operations:

$$\begin{aligned} X &= \ < u := a, v := b, c := f(c, v) > , \\ Y &= \ < u := a, v := b, w := c, b := g(u, w) > \end{aligned}$$

Let $S = \{ a = 1 , b = 0 , c = 2 \}$ be database state, then S is an initial state for X. $X(S) = S'$, where $S'(c) = f(2, 0)$. Now, consider a *concurrent* execution of T_1 and T_2 of the previous example. There are many possible histories. We show the histories, Z and W, whose initial state is S given above. Each operation is prefixed with the name of the transaction that issued it.

$$\begin{aligned} Z &= \ < T_2 : a := 0,\ T_1 : u := a,\ T_2 : b := 1,\ T_1 : v := b,\ T_1 : w := c,\ T_1 : b := g(u, w) >, \\ W &= \ < T_2 : a := 0,\ T_2 : b := 1,\ T_1 : u := a,\ T_1 : v := b,\ T_1 : w := c,\ T_1 : b := g(u, w) > \end{aligned}$$

Observe that $Z(S) = W(S) = S''$, where $S'' = \{ a = 0 , b = g(0, 2) , c = 2 \}$. Observe that $Z \equiv W$. $\diamond$

A key notion in the treatment of compensation is *commutativity*. We say that two sequences of operations, X and Y, *commute*, if $(X \circ Y) \equiv (Y \circ X)$. Two operations *conflict* if they do not commute. Observe that defining operations as functions, regardless to whether they read or update the database, leads to a very simple definition of the key concept of commutativity. (Compare our definition to those of [Wei88, BR87] for example).

Typically, an operation that updates an entity and an operation that reads it do not commute. Part of the orderings implied by the total order in which operations are composed to form a history are arbitrary, since only conflicting operations must be totally ordered. In essence, our equivalence notion (when restricted to database state) is similar to final-state equivalence [Pap86]. However, in what follows, we shall need to equate histories that are not necessarily over the same set of transactions, which is in contrast to final-state equivalence (and actually to all familiar equivalence notions).

A *projection* of a history X on an entity e is is a subsequence of X, that consists of the operations in X that updated e. We denote the projection of X on e as X_e. The same notation is used for a projection on a set of entities.

We impose very weak constraints on concurrent executions in order to exclude as few executions as possible from consideration. In this paper we consider the following types of histories:

- A history X is *serial* if for every two transactions T_i and T_j that appear in X, either all operations of T_i appear before all operations of T_j or vice versa.

- A history X is *serializable* (SR) if there exists a serial history Y such that $X \equiv Y$.
- Let $C = c_1 \wedge \ldots \wedge c_n$ be a predicate over the database state. For each conjunct c_i let d_i denote the set of database entities mentioned in c_i. A history X is *predicate-wise serializable* with respect to a predicate C (PWSR$_C$) if for every set of entities d_i there exists a serial history Y such that $X_{d_i} \equiv Y_{d_i}$.
- A history X is *entity-wise serializable* (EWSR) if for every entity e there exists a serial history Y such that $X_e \equiv Y_e$.

The definition of PWSR histories is adapted from [KKB88]. As we shall see shortly, EWSR histories are going to be quite useful in our work. The following lemma is given without proof.

Lemma 1. *Let C be a predicate that mentions all database entities, and Let $ewsr$, $pwsr_C$, sr denote the set of EWSR histories, PWSR$_C$ histories, and SR histories, respectively. Then,* $sr \subset pwsr_C \subset ewsr$. □

We denote by X_T the sequence of operations of a transaction T in a history X, involving possibly other transactions. The same notation is used for sets of transactions. When X_T is projected on entity e the resulting sequence is denoted $X_{T,e}$.

4 Compensating Transactions

With the aid of the tools developed in the last section, we are in a position to define compensation more formally.

4.1 Guidelines for Defining Compensating Transactions

Although compensation is an application-dependent activity, there are certain guidelines to which every compensating transaction must adhere. After introducing some notation and conventions we present three specification constraints for defining compensating transactions. These constraints provide a very broad framework for defining concrete compensating transactions for concrete applications, and can be thought of as a generic specification for all compensating transactions.

We say that transaction T_j is *dependent upon* transaction T_i in a history if there exists an entity e such that

- T_j reads e after T_i has updated e;
- T_i does not abort before T_j reads e; and
- every transaction (if any) that updates e between the time T_i updates e and T_j reads e, is aborted before T_j reads e.

The above definition is adapted from [BHG87].

A transaction T_i, which is the depended-upon transaction may be either a committed transaction, or an active transaction. In either case, if we want to support the undo of T_i, then the corresponding compensating transaction, CT_i, must be pre-defined. The key point is that admitting non-recoverable histories and supporting the undo of committed transactions, is predicated on the existence of the compensatory mechanisms needed to handle undoing externalized transactions. In the rest of the paper, T denotes a compensated-for transaction, CT denotes the corresponding compensating transaction, and $dep(T)$ denotes a set of transactions dependent upon T. This set of dependent transactions can be regarded as a set of related (sub)transactions that perform some coherent task.

Constraint 1. *For all histories X, if $X_{T,e} \circ X_{CT,e}$ is a contiguous subsequence of X_e, then $(X_{T,e} \circ X_{CT,e}) \equiv I$, where I is the identity mapping.* □

The simplest interpretation of Constraint 1 is that for all entities e that were updated by T but read by no other transaction (since $X_{CT,e}$ follows $X_{T,e}$ in the history), CT amounts simply to undoing T. Consequently, if there are no transactions that depend on T, (i.e., no transaction reads T's updated data entities), then CT is just the traditional *undo*(T). The fact that CT does not always just undo T is crucial, since the effects of compensation depend on the span of history from the execution of the compensated-for transaction till its own initiation. If such a span exists, and T has dependent transactions, the effects of compensation may vary and can be very different from undoing T. For instance, compensation may include additional activity that is not directly related to undoing. A good example here, is a cancellation of reservation in an airline reservation system which is handled as a compensating transaction that causes the transfer of pending reservation from a waiting list to the confirmed list.

There are certain operations on certain entities that cannot be undone, or even compensated-for, in the form of inverting the state. In [Gra81] these type of operations and entities are termed *real* (e.g., dispensing money, firing a missile). For simplicity's sake, we omit discussion of such entities.

Constraint 2. *Given a history X involving T and CT, there must exist X' and X'' subsequences of X, such that no transaction has operations both in X' and in X'', and $X \equiv X' \circ X_{CT} \circ X''$.* □

This constraint represents the atomicity of compensation. That is, a transaction should either see a database state affected by T, or see a state following CT's termination. More precisely, transactions should not have operations that conflict with CT's operations scheduled both before and after CT's operations, or in between CT's first and last operations. It is the responsibility of the concurrency control protocol to implement this constraint (see

Section 6 for implementation discussion).

In what follows, we use the notation O_T and I_T to denote the output and input predicate of transaction T, respectively. The same notation is used for a set of transactions. These predicates are predicates over the database state.

Constraint 3. *Let Q be a predicate defined over the database state, if $(O_{dep(T)} \Rightarrow Q) \wedge (I_T \Rightarrow Q)$ then $O_{CT} \Rightarrow Q$.* $\square$

Constraint 3 is appropriate when Q is a either general consistency constraint, or a specific predicate that is established by *dep*(T) (that is, one of the collective tasks of the transactions in *dep*(T) was to make Q true). It constraints the semantics of compensation by imposing restrictions on the output predicate of the compensating transaction. Observe that the assumption that Q holds initially (i.e., $I_T \Rightarrow Q$) is crucial since T's effects are undone by CT, and hence, predicates established by T and preserved by *dep*(T) do not persist after the compensation. It is the responsibility of whoever defines CT to enforce Constraint 3.

Constraints 1 and 2 will be assumed to hold for all compensating transactions, hereafter. Constraint 3, which is more intricate and captures more of the semantics of compensation, will be discussed further in Section 6.

4.2 Types of Compensation

As was mentioned earlier, compensation is really an application-dependent activity. Therefore, there are many ways to define what compensation is supposed to accomplish. An important criterion exists, however, that dichotomizes the range of possibilities. For some applications, it is acceptable that an execution of the dependent transaction, without the compensated-for and the compensating transactions, would produce different results than those produced by the execution with the compensation. On the other hand, other applications might forbid compensation unless the outcome of these two executions is the same. Next we make explicit the above criterion that distinguishes among types of compensation by defining the notion of compensation soundness.

Definition 1. *Let X be the history of T, CT, and dep(T) whose initial state is S. Let Y be some history of only the transactions in dep(T) whose initial state is also S. The history X is sound, if $X(S) = Y(S)$.* $\square$

The history Y can be any history of *dep*(T). As far as the definition goes, different sets of (sub)transactions of *dep*(T) may commit in X and in Y, and conflicting operations may be ordered differently. The key point is that $X(S) = Y(S)$. If a history is sound then compensation does not disturb the outcome of the dependent transactions. The database state after compensation is the same as the state after an execution of only the dependent transactions, *dep*(T). All direct and indirect effects of the compensated-for transaction, T, have been erased by the compensation.

Transactions in $dep(T)$ see different database states when T and CT are not executed, and therefore generate a history Y which can be totally different than the history X. This distinction between the histories X and Y, which is the essence of the important notion of soundness, would not have been possible had we viewed a transaction merely as sequence of operations rather than a program.

A delicate point arises with regard to soundness when S does not satisfy $I_{dep(T)}$. Such situations may occur when T establishes $I_{dep(T)}$ for $dep(T)$ in such a manner that $dep(T)$ *must* follow T in any history. Hence, if T is compensated-for, there is no history of $dep(T)$, Y, that can satisfy the soundness requirement. We model such situations by postulating that if $I_{dep(T)}(S)$ does not hold, then $Y(S)$ results in a special state that is not equal to any other state (the ***undefined*** state), and hence X is indeed not sound.

We illustrate Definition 1 by considering the following two histories over read and write operations (the notation $r_i[e]$ denotes reading e by T_i, and similarly $w_i[e]$ for write, and c_i for commit):

$$\begin{aligned} W &= \ < w_j[e]\ ,\ r_i[e]\ ,\ c_j\ ,\ c_i >, \\ Z &= \ < w_j[e]\ ,\ r_i[e]\ ,\ w_i[e']\ ,\ c_i > \end{aligned}$$

The history W is recoverable. History Z is not recoverable. If however, CT_j is defined, T_j can still be aborted. Let us extend Z with the operations of CT_j and call the extended history Z'. Z' is sound provided that Z'_{T_i} would have been generated by T_i's program, and the same value would have been written to e', had T_i run in isolation starting with the same initial state as in Z'.

The key notion in the context of compensation, as we defined it, is ***commutativity*** of compensating operations with operations of dependent transactions. Significant attention has been devoted to the effects of commutative operations on concurrency control [Kor83, Wei88, BR87]. Our work parallels these results as it exploits commutativity with respect to recovery. In all of our theorems we prefer to impose commutativity requirements on CT rather than on T, since CT is less exposed to users, and hence constraining it, rather than constraining T, is preferable. Predicated on commutativity, the operations of the compensated-for transaction and the corresponding compensatory operations can be 'brought together', and then cancel each other's effects (by the enforcement of Constraint 1), thereby ensuring sound histories. The following theorem formalizes this idea.

Theorem 1. *Let X be a history involving $T, dep(T)$ and CT. If each of the operations in $X_{dep(T)}$ commutes with each of the operations in X_{CT}, then X is sound.* □

We omit the proof of the theorem since it is not central to our exposition. However we illustrate it by the following simple example:

Example 3. Let T_i, T_j and CT_i be a compensated-for transaction, a dependent transaction and the compensating transaction, respectively. Let the programs of all these transactions include no condition statements (i.e., they are sequences of operations). We give a history X, in which each operation is prefixed by the name of the issuing transaction. $X =< T_i : a := a+2,\ \ T_j : u := b,\ \ T_j : a := a+u,\ \ CT_i : a := a+2) >$. Clearly, every operation of T_j commutes with every operation of CT_i in X. Hence, X is sound, and the history that demonstrates soundness is simply $Y = X_{T_j} =< T_j : u := b,\ \ T_j : a := a+u >$. As will become clear in Section 6, the fact that no condition statements appear in T_j is important. ◇

Our main emphasis in this paper is on more liberal forms of compensation soundness, where the results of executing the dependent transactions in isolation may be different from their results in the presence of the compensated-for, and the compensating transactions. One way of characterizing these weaker forms of soundness is by qualifying the set of entities for which the equality in Definition 1 holds. In Section 5.1, we define a type of compensating transaction that ensures sound compensation with respect to some set of entities. Alternatively, in Section 6 we investigate other weak forms of soundness that approximate (pure) soundness.

5 Compensation Examples and Applications

In this section, we present several examples to illustrate the various concept we have introduced so far. Throughout this section we use the symbols T, $dep(T)$, CT, X, and S to denote a compensated-for transaction, its compensating transaction, the corresponding set of dependent transactions, the history of all these transactions, and the history's initial state, respectively.

5.1 A Generic Compensating Transaction

In this example we present a generic compensation definition. Let $update(T, X)$ denote the set of database entities that were updated by T in history X. The same notation is used for a set of transactions.

Definition 2. *Let $X(S) = S'$, and $X \equiv X' \circ X_{CT}$ (by Constraint 2). We define the generic compensating transaction CT, by characterizing S' for all entities e:*

$$S'(e) = \begin{cases} S(e) & \text{if } e \notin update(dep(T), X) \\ (X'(S))(e) & \text{if } e \in update(dep(T), X) \wedge e \notin update(T, X) \\ X_{dep(T),e}(S) & \text{if } e \in update(dep(T), X) \wedge e \in update(T, X) \end{cases}$$

□

Before we proceed, we informally explain the meaning of this type of compensation.

- If no dependent transaction updates an entity that T updates, CT undoes T's updates on that entity.
- The value of entities that were updated only by dependent transactions is left intact.

- The value of entities updated by both T and its dependents should reflect only the dependents' updates.

There is a certain subtlety in the second case of the definition which is illustrated next. Assume that T updated e. The modified e is read by a transaction in $dep(T)$ and the value read determines how this transaction updates e'. After compensation, even though the initial value of e is restored (by the first case of the definition), the indirect effect it had on e' is left intact (by the second case of the definition). We use the above definition as a precise specification of what CT should accomplish.

To further illustrate the type of compensation just described, we give a concrete example. Consider an airline reservation system with the entity *seats* that denotes the total number of seats in a particular flight, entity *rs* that denotes the number of already reserved seats in that flight, and entity *reject* that counts the number of transactions whose reservations for that flight have been rejected. Let **reserve(x)** be a simplified seat reservation transaction for x seats defined as:

```
if    (rs + x) <= seats   then rs := rs + x
                          else reject := reject + 1
```

The consistency constraint Q in this case is: $Q(S)\ \ iff\ \ S(rs) \leq S(seats)$. Assume:

$$S = \{seats = 100, rs = 95, rejects = 10\},\ T = \textbf{reserve(5)},\ dep(T) = \{\textbf{reserve(3)}\}$$

Let the history be $X \equiv X_T \circ X_{dep(T)} \circ X_{CT}$ where CT is defined by Definition 2. We would like to have after X: $S' = \{rs = 95, rejects = 11\}$, that is, T's reservations were made and later canceled by running CT, and $dep(T)$'s reservations were rejected. And that is exactly what we get by our definition. Observe how T's reservations were canceled, but still its indirect impact on *rejects* persists (since T caused $dep(T)$'s reservations to be rejected).

Hence, this example demonstrates a history that is not sound but is nevertheless intuitively acceptable. Had the transaction in $dep(T)$ been executed alone, it would result in successful reservations. Notice how in this example the operation of CT can be implemented as inverse of T's operation (addition and subtraction). The less interesting case, where there are enough seats to accommodate both T and $dep(T)$, also fits nicely. In this case CT's subtraction on the entity *seats* commutes with $dep(T)$'s addition to this entity.

5.2 Storage Management Examples and Applications

The following example is from [MGG86], though the notion of compensation is not used there. Consider transactions T_1 and T_2, each of which adds a new tuple to a relation in a relational database. Assume the tuples added have different keys. A tuple addition is processed by first allocating and filling in a slot in the relation's tuple file, and then adding the key and slot number to a separate index. Assume that T_i's slot updating (S_i) and index insertion (I_i) steps can each be implemented by a single page read followed by a single page write (written $r_i[tp]$, $w_i[tp]$ for a tuple file page p, and $r_i[ip]$, $w_i[ip]$ for an index file page p).

Consider the following history of T_1 and T_2 regarding the tuple pages tq, tr and the index page ip:

$$< r_1[tq] \, , \, w_1[tq] \, , \, r_2[tr] \, , \, w_2[tr] \, , \, r_2[ip] \, , \, w_2[ip] \, , \, r_1[ip] \, , \, w_1[ip] >$$

This is a serial execution of $< S_1 , S_2 , I_2 , I_1 >$, which is equivalent to the serial history of executing T_1 and then T_2. Assume, now, that we want to abort T_2. The index insertion I_1 has seen and used page p, which was written by T_2 in its index insertion step. The only way to abort T_2, without aborting T_1 is to compensate for T_2. Fortunately, we have a very natural compensation, CT_2, which is a delete key operation. Observe that a delete operation as compensation, satisfies Constraint 1, commutes with insertion of a tuple with a different key, and encapsulates composite compensation for the slot updating and index insertion. The resulting history is sound.

An entire class of applications for compensation (similar to the above example) can be found in the context of storage management in a database system. It is difficult to isolate the effects of an operation at the storage management level. Therefore, these effects are exposed to all the transactions. We list several specific examples as an illustration: The compensated-for transaction extends a file, or is allocated storage, and the additional space is used by other transactions; the compensated-for transaction frees space that is later allocated for other transactions; the compensated-for transaction inserts a record to a B-tree that causes a split of a node, and other transactions use the new nodes; the compensated-for transaction updates the free space information mechanism of the storage manager (percentage of occupied space in a page, etc.) and other transactions update the same information. We note that, in all the above storage management examples, although effects are exposed to transactions, they are not exposed to users.

Undoing such operations is referred to as *logical undoing* and is supported by *logical logging* [HR83, MHL+89]. The transaction is undone in a logical manner, from a semantic point of view. On the physical level it might leave traces. Clearly, use of physical logging, where before-images are restored blindly for every object affected by the undone transaction is out of the question in such cases. We propose to define compensatory actions for these type of storage management actions. Often, the compensation amounts to simply leaving the effects as they are. The point is, however, that traditional (physical) undoing does not work properly in this context.

6 Approximating Compensation Soundness

In this section we introduce weak forms of compensation soundness, where the results of an execution that includes compensation only *approximate* the results of executing the dependent transactions in isolation. We state several theorems that formalize the interplay among the approximated soundness notion, concurrency control constraints, restrictions on programs of dependent transactions, and commutativity. Each theorem is followed by a simplified example that serves to illustrate at least part of the theorem's premises and consequences. Proofs of the theorems can be found in the Appendix. Throughout this section, we assume that a compensating transaction complies with Constraints 1 and 2 of Section 4. We use these constraints in the proofs without explicitly mentioning them in the premises. We start with definitions of weaker forms of commutativity and weaker forms of compensation soundness.

Definition 3. *Two sequences of operations, X and Y, commute with respect to a relation $\mathcal{R}$ on augmented states (in short, R-commute), if for all augmented states S, $(X \circ Y)(S)\ \mathcal{R}\ (Y \circ X)(S)$.* □

Observe that when $\mathcal{R}$ is the equality relation we have regular commutativity.

Definition 4. *Let X be a history of T, $dep(T)$, and CT whose initial state is S, and let $\mathcal{R}$ be a reflexive relation on augmented states. The history X is sound with respect to R (in short R-sound), if there exists a history Y of $dep(T)$ whose initial state is S such that $Y(S)\ \mathcal{R}\ X(S)$.* □

Observe that regular soundness is a special case of R-soundness when $\mathcal{R}$ is the equality relation. Since $\mathcal{R}$ is reflexive, the empty history is always R-sound, regardless of the choice of R.

We motivate the above definitions by considering adequate relations $\mathcal{R}$ in the context of R-commutativity and R-soundness. Let Q be a predicate on database states such that $O_{dep(T)} \Rightarrow Q$. Q can be regarded as either a consistency constraint, or a desired predicate that is established by $dep(T)$ (similarly to the predicate Q in Constraint 3). Therefore, we would like to guarantee that compensation does not violate Q. Define $\mathcal{R}$ (in the context of X, Y and S) as follows:

$$Y(S)\ \mathcal{R}\ X(S)\ \ iff\ \ (Q(Y(S)) \Rightarrow Q(X(S)))$$

An R-sound history with such $\mathcal{R}$ has the advantageous property that predicates like Q are not violated by the compensation. Such R-sound histories yield states that approximate states yielded by sound histories in the sense that both states satisfy some desirable predicates. That definition of $\mathcal{R}$ is in the spirit of CAD systems where the final result of an execution is of importance, more than the order and the recovery-related actions of the execution [KLMP84, KKB88]. In the examples that follow the theorems, we use relations $\mathcal{R}$ of that form.

Definition 5. *Let $\mathcal{R}$ be a relation on states, and let v_e and v'_e denote values of an arbitrary entity e. We define the relations R_e on values of e for every entity e as follows:*

$v_e\ \mathcal{R}_e\ v'_e\ \ iff\ \ (\exists S', S'':\ S'(e) = v_1\ \wedge\ S''(e) = v_2\ \wedge\ S'\ \mathcal{R}\ S'')$ □

Definition 6. *Let X be a history of T, $dep(T)$ and CT whose initial state is S, and let $\mathcal{R}$ be a reflexive* *relation on augmented states. The history X is partially R-sound if there exists a history Y of $dep(T)$ whose initial state is S such that $(\forall e \in db : (Y(S))(e)\ \mathcal{R}_e\ (X(S))(e))$.* □

Definition 7. *A program of a transaction is fixed if it is a sequence of operations that use no private entities as arguments.* □

If T's program is fixed then it has no conditional branches. Moreover, T cannot use local variables to store values for subsequent referencing. A sequence of operations, where each operation reads and updates a single database entity (without storing values in local variables) is a fixed transaction. A transaction that uses a single operation to give a raise to a certain employee recorded in a salary management database is an example for a fixed transaction.

Theorem 2. *Let X be a history of $T, dep(T)$ and CT whose initial state is S. If the histories $X_{dep(T)}$ and X_{CT} R-commute, X is EWSR, and all programs of transactions in $dep(T)$ are fixed, then X is partially R-sound.*
□

Example 4. Consider a database system with the following entities, parametric operations, and reflexive relation:

$$db = \{a : integer,\ b : integer\}\ ,\ f(e) :: \ \textbf{if}\ e > 2\ \ \textbf{then}\ \ e := e - 2\ ,\ g(e) :: \ \textbf{if}\ e > 10\ \textbf{then}\ \ e := e - 10$$

$$S'\ \mathcal{R}\ S''\ iff\ (((S'(b) \geq 0 \wedge S'(a) \geq 10) \vee (S'(a) = 4)) \Rightarrow ((S''(b) \geq 0 \wedge S''(a) \geq 10) \vee (S''(a) = 4)))$$

(The predicates on a are present only to demonstrate partial R-soundness). We emphasize that f and g are (atomic) operations. The history X is as follows (there is no need to give the program of $dep(T)$ since it is fixed):

$$X =< dep(T) : A := a + 2,\ \ T : f(a),\ \ T : g(b),\ \ dep(T) : g(b),\ \ CT : a := a + 2,\ \ CT : b := b + 10 >$$

Observe that $X_{dep(T)}$ and X_{CT} do not commute but they do R-commute for the given relation $\mathcal{R}$. Let the initial state be $S = \{a = 2,\ b = 15\}$. We have that $X(S) = \{a = 4,\ b = 15\}$, whereas $Y(S) = \{a = 4,\ b = 5\}$, and indeed X is *partially* R-sound. ◇

Theorem 2 is quite weak since it restricts the programs of $dep(T)$ severely, and guarantees only partial R-soundness. The inherent problem with (the proofs of) compensation soundness is the fact that they equate two histories that are *not* over the same set of transactions, which is in contrast to all the equivalence notions in the traditional theory of concurrency control. The obstacle is that the history Y may be generated by different executions of the programs of $dep(T)$, and may be totally different from $X_{dep(T)}$, which is just a syntactic derivative of the history X. In Theorem 2, this problem was solved only because $dep(T)$ was fixed (see the proof in the Appendix). This obstacle can be removed by posing more assumptions, as is done next.

Definition 8. *A transaction T is a* <u>*serialization point*</u> *in a history X if $X \equiv X' \circ X_T \circ X''$.* □

Observe that no restrictions are imposed on X' and X''. Also notice that a compensating transaction is a serialization point, as implied by Constraint 2.

Theorem 3. *Let X be a history of $T, dep(T)$ and CT. Let Z be a history of the transactions in $dep(T)$ and CT such that $Z \equiv Z_{dep(T)} \circ Z_{CT}$. If for all states S and for all histories Z, there exists a history Y of $dep(T)$ such that $(Z_{CT} \circ Y)(S)\ \mathcal{R}\ (Z_{dep(T)} \circ Z_{CT})(S)$, then every history X where T is a serialization point is R-sound.*
□

Note that it is required that $dep(T)$'s programs be such that executing CT before $dep(T)$ would result in a state that is related by $\mathcal{R}$ to the state resulting when executing $dep(T)$ first and then CT. Observe that this requirement is stronger than R-commutativity.

This theorem is quite useful since it specifies a concurrency control policy that guarantees R-soundness. Namely, we need to ensure that every potential compensated-for transaction be isolated (i.e., T is a serialization point) in order to guarantee R-soundness in case of compensation.

Example 5. Consider the set entities of Example 4, with the addition of a private entity u that belongs to some transaction in $dep(T)$. Let the programs of T, $dep(T)$, CT, and the relation $\mathcal{R}$ be defined as follows:

$$T = a := a + 1 \quad , \quad dep(T) = \{\, u := a;\ \ \textbf{if}\ u \geq 5\ \textbf{then}\ f(b)\ \ \textbf{else}\ \ g(b)\,\}$$
$$CT = a := a - 1 \quad , \quad S'\ \mathcal{R}\ S''\ \ iff\ \ (S'(b) \geq 0 \Rightarrow S''(b) \geq 0)$$

Even though $dep(T)$'s history can branch differently when run alone and in the presence of T and CT, the two different histories produce final states that are related by $\mathcal{R}$. ◇

Definition 9. *A program of a transaction is* <u>*linear*</u> *if it is a sequence of operations.* □

Programs are sequences, but we allow operations to read multiple entities, that is, use local variables. Therefore, programs may not be fixed. An example for a linear transaction program is a program that gives a raise to all employees, where the raise based on some aggregated computation (for instance 10% of the minimum salary).

Definition 10. *Let $\mathcal{R}$ be a reflexive relation on augmented states. An operation f that updates e* <u>*preserves $\mathcal{R}$*</u>*, if $(\forall e' \in adb : (S(e')\ \mathcal{R}_{e'}\ S'(e')) \Rightarrow (f(S)\ \mathcal{R}_e\ f(S'))$* □

Theorem 4. *Let X be a history of $T, dep(T)$ and CT whose initial state is S. If the histories $X_{dep(T)}$ and X_{CT} R-commute, X is EWSR, the programs of all transactions in $dep(T)$ are linear, R is transitive, and the operations of $dep(T)$ preserve $\mathcal{R}$, then X is partially R-sound.* □

Example 6. Consider the set entities of Example 4, with the addition of a private entity u that belongs to some transaction in $dep(T)$. We use the relation $S' \mathcal{R} S''$ iff $((S'(b) \geq S'(a)) \Rightarrow (S''(b) \geq S''(a)))$. The history X is as follows:

$$X = < T : a := a + 1,\ \ dep(T) : u := a,\ \ dep(T) : b := u + 10,\ CT : a := a - 1 >$$

Observe that X_{CT} and $X_{dep(T)}$ R-commute (but do not commute), $dep(T)$ is linear (but not fixed), and X is (partially) R-sound. ◇

Finally, based on Lemma 1 from Section 2, we derive the following corollary.

Corollary 1. *Theorems 2 and 4 hold when X is $PWSR_C$ or SR instead of EWSR.* □

The requirements from the dependent transactions in Theorems 2, 3, and 4 are quite severe. Besides the R-commutativity requirement imposed on the operations of the dependent transactions, there are restrictions on the shape of the programs (e.g., fixed or linear programs) in each of the theorems' premises. Clearly, in practical systems, there are many transactions that do not stand up to any of these criteria. The practical ramification of this observation is that externalization of uncommitted data items should be done in a controlled manner if a degree of soundness is of importance. That is, uncommitted data should be externalized only to transactions that do satisfy the requirements specified in the premises of the theorems. In the context of locks, locks should be released only to qualified transactions, that is, those transactions that do satisfy the requirements. Other transactions must be delayed and are subject to the standard concurrency control and recovery policies.

7 On the Implementation of Compensating Transactions

In this section we discuss several implementation issues that need to be considered in order for compensation to be of practical use. A very useful component in a recovery system that supports compensation is the *log*. The log is the source for crucial information such as the actual sequence of operations generated by the compensated-for transaction, the identity of the dependent transactions, etc. It is likely that a logical logging scheme on an operation level [HR83] will be used in conjunction with compensation. Each log record in this scheme contains the operation name, the issuing transaction name, the name of the entity the operation updates, and the values of its arguments. Therefore, this scheme provides more semantic information than physical logging [HR83]. We envision that a compensating transaction would be driven by a scan of the log starting from the first record of the compensated-for transaction and up to its own begin-transaction log record. It is important to provide convenient on-line access to the log information for these purposes. Without a suitable logging architecture, these accesses might translate to I/O traffic that would interrupt the sequential log I/O that is performed on behalf of executing transactions. A related problem is concerned with supporting compensation for long-duration transactions whose log records span a lengthy log interval, thereby causing difficulties in terms of reusing log space and efficient access to the distant log records. We are currently investigating the issue of efficient log access for recovery purposes.

There are some subtle ramifications on concurrency control which are discussed next in the context of locking. We have required that CT's execution is serializable with respect to other concurrent transactions. (Constraint 2). Also, if it is reasonably assumed that $update(CT, X) \subseteq update(T, X)$, then this leads to the conclusion that the compensating transaction and the dependent transactions should follow a 2-Phase Locking protocol [BHG87] with respect to entities in $update(T, X)$. Otherwise, it possible to violate Constraint 2. A viable strategy that might simplify matters for the implementation can be as follows. Once CT is invoked, the entities in $update(T, X)$ should be identified by analyzing the log and then CT should exclusively lock all entities in this set. After performing the necessary updates, CT can release these locks.

The recovery issues of compensating transactions themselves must be also considered. As was noted earlier, we should disallow a compensating transaction to be aborted voluntarily. The choice of either to abort or to commit is present for the original transaction. A compensating transaction offers the ability to reverse this choice, but we do not go any further by providing the capability to abort the compensation. Other forms of transaction failures (as opposed to system failures) should be avoided too. A compensating transaction should not be chosen as the victim in a deadlock resolution scenario. Running one compensating transaction at a time, thereby avoiding a cycle of compensating transactions, might solve the problem (similarly to *golden transactions* in System R [GM+81]). Alternatively, the chances of selecting a compensating transaction as a victim, can be minimized by assigning them high priority, and choosing victims (of deadlock resolution) with lowest priority. Still, there is the problem of system failures. We think that the preferred way to handle this problem is to resume uncompleted compensating transactions rather than undoing them. To accomplish this, we need to resume a compensating transaction from a point where its internal state was saved along with the necessary concurrency control information, (refer to [GM+81, KKB88] for discussion of *save points* and saving locking status). We emphasize that the principle for recovery of compensating transaction is that once a begin-transaction record of CT appears in stable storage, CT must be completed.

The technique of logging undo activity is discussed in length in [MHL+89]. An implementation along the lines of the ARIES system [MHL+89] can support the persistence of compensating transactions across system crashes. In ARIES, undo activity is logged using Compensating Log Records (CLRs). Each CLR points (directly or indirectly) to the next regular log record to be undone. This technique guarantees that actions are not undone more than once and that undo actions are not undone, even if the undo of a transaction is interrupted by a system crash.

8 Related Work

The idea of compensating transactions as a semantically-rich recovery mechanism is mentioned, or at least referred to, in several papers. However, to the best of our knowledge, a formal and comprehensive treatment of the issue and its ramifications is lacking.

Strong motivation for our work can be found in Gray's early paper [Gra81]. There, compensating transactions are mentioned informally as 'post facto' transactions that are the only means to alter committed effects. Also, compensation is mentioned as a possible remedy to the limitations of the current transaction model. Another early reference is the DB/DC database system [Bjo73, Dav73], where the idea of semantic undoing is used.

The notion of compensation (countersteps) is mentioned in the context of histories that preserve consistency without being serializable in [GM83]. It is noted there that running countersteps (to undo steps) does not necessarily return the database to its initial state, an observation on which we elaborate in our work. The difficulty of designing countersteps is raised as a drawback of compensation, which is another problem we address.

Compensating transactions are also mentioned in the context of a *saga*, a long-duration transaction that can be broken into a collection of subtransactions that can be interleaved in any way with other transactions [GMS87]. A saga must execute all its subtransactions, hence compensating transactions are used to amend partial execution of sagas. In [GMS87] and in [GM83] the idea that a compensating transaction cannot voluntarily abort itself is introduced. Low-level details of how to store reliably the code of compensating transactions, and record their identity in the log records of the saga's subtransactions are also discussed there.

A noteworthy approach, which can be classified as a simple type of compensation, is employed in the XPRS system [SKPO88]. There, a notion of *failure commutativity* is defined for complete transactions. Two transactions failure commute if they commute; and if they can both succeed then a unilateral abort by either transaction cannot cause the other to abort. Transactions that are classified as failure commutative can run concurrently without any conflicts. Handling the abort of such a transaction is done by a log-based special undo function, which is a special case of compensation as we define it.

A more theoretic class of related work is based on commutativity-like properties of abstract data type operations. In [BR87], semantics of operations on abstract data types are used to define *recoverability*, which is a weaker notion than commutativity. Conflict relations are based on recoverability rather than commutativity. Consequently, concurrency is enhanced since the potential for conflicts is reduced. When an operation is recoverable with respect to an uncommitted operation, the former operation can be executed; however a commit dependency is forced between the two operations. This dependency affects the order in which the operations should commit, if they both commit. If either operation aborts, the other can still commit, thereby avoiding cascading aborts.

This type of work is more conservative than ours in the sense that it narrows the domain of interest to serializable histories. Our results offer several notions that are even weaker than standard recoverability, and hence applicable in the wider domain that includes non-serializable and non-recoverable histories. Moreover, our work explicitly allows and handles situations of exposed dirty data, and offers the extra flexibility of redressing such cases when the need arises.

9 Conclusions

In this paper we have discussed the consequences of early exposure of uncommitted data. We have argued that this concept is very useful for many database applications employing long-duration and nested transactions. Compensating transactions are proposed as the means for recovery management in the presence of early externalization. A framework for the understanding and design of the semantics of these application-dependent compensatory activities is established. Several types of compensation soundness criteria are introduced and are found to be predicated on notions of commutativity. Even the approximated forms of soundness can be used to guarantee that compensation results in desirable consequences and does not abrogate dependent transactions' outcome. A semantically-rich model that is adequate for dealing with non-serializable and non-recoverable histories is set up, and is offered as a viable tool for the understanding of these intricate histories and compensation issues.

We believe that future database applications will require the rethinking of the traditional transaction model that is founded on serializability and permanence of commitment. Contemporary applications in the domains of CAD, CASE, CIM (and other domains that require long-durations and nested transactions) exemplify our belief. The work presented in this paper is a step towards the establishment of this new model.

References

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

[Bjo73] L. A. Bjork. Recovery scenario for a DB/DC system. In *Proceedings of the ACM Annual Conference, Atlanta*, pages 142–146, 1973.

[BR87] B. R. Badrinath and K. Ramamirtham. Semantic-based concurrency control: Beyond commutativity. In *Proceedings of the Third International Conference on Data Engineering, Los Angeles*, 1987.

[Dav73] C. T. Davies. Recovery semantics for a DB/DC system. In *Proceedings of the ACM Annual Conference, Atlanta*, pages 136– 141, 1973.

[GM$^+$81] J. N. Gray, P. McJones, et al. The recovery manager of the system R database manager. *ACM Computing Surveys*, 13(2):223–242, 1981.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco*, pages 249–259, 1987.

[Gra80] J. N. Gray. A transaction model. In *Lecture Notes in Computer Science: Automata Languages and Programming*, pages 282–298. Springer-Verlag, Berlin, 1980.

[Gra81] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Databases, Cannes*, pages 144–154, 1981.

[HR83] T. Haerder and A. Reuter. Principles of transaction oriented database recovery — a taxonomy. *ACM Computing Surveys*, 15(4):289–317, December 1983.

[HR87] T. Haerder and K. Rothermel. Concepts for transaction recovery in nested transactions. In *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco*, pages 239–248, 1987.

[KKB88] H. F. Korth, W. Kim, and F. Bancilhon. On long duration CAD transactions. *Information Sciences*, 46:73–107, October 1988.

[KLMP84] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. Nested transactions for engineering design databases. In *Proceedings of the Tenth International Conference on Very Large Databases, Singapore*, pages 355–362, 1984.

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[KS88] H. F. Korth and G. Speegle. Formal model of correctness without serializability. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago*, pages 379–388, June 1988.

[Lag88] Future directions in DBMS research, 1988. A report that summarizes the Laguna Beach workshop.

[MGG86] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstractions in recovery management. In *Proceedings of ACM-SIGMOD 1986 International Conference on Management of Data, Washington*, pages 72–83, 1986.

[MHL+89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Technical Report RJ 6649 (63960), IBM Research, 1989. To appear in ACM Transactions on Database Systems.

[Pap86] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rickville, Maryland, 1986.

[SKPO88] M. R. Stonebraker, R. H. Katz, D. A. Patterson, and J. K. Ousterhout. The design of XPRS. In *Proceedings of the Fourteenth International Conference on Very Large Databases, Los Angeles*, pages 318–330, 1988.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, C-37(12):1488–1505, December 1988.

[ZM90] S. B. Zodnik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, California, 1990.

A More on the Model

We present some definitions and notation that are going to be used in the formal proofs. When a projection on an entity is applied to a state, we are interested in the resulting value of that particular entity. Therefore, we use $X_e(S)$, as a shorthand for $(X_e(S))(e)$.

The astute reader may have noticed that $X_e(S)$ is not well defined, and in particular it is not necessarily equal to $(X(S))(e)$. Since X_e includes only operations that update e, and since private entities are updated only once, the value of all private entities is undefined when histories that are projected on database entities are applied to a state. To rectify this anomaly we define the function ex from database states and histories to augmented states as follows. Let S be a database state, and X a history, then:

$$(ex(S,X))(e) = \begin{cases} S(e) & \text{if } e \in db \\ (X(S))(e) & \text{if } e \in adb - db \end{cases}$$

It should be noted that in the third case of Definition 1, we used $X_{dep(T),e}(S)$ instead of $X_{dep(T),e}(ex(S,x))$ for the sake of simplicity. We illustrate the function ex and its use in the following example:

Example 7. Let $u \in (adb - db)$, and $\{a,b\} \subseteq db$. Consider the following history $X =< u := a,\ b := f(u) >$. Let $S(u) = \emptyset$ (undefined value), and $S(a) = 1$. Then, $X_b(S) = f(S(u)) = f(\emptyset)$, whereas $(X(S))(b) = f(1)$. However, $(ex(S,X))(u) = (X(S))(u) = 1$, and then $X_b(ex(S,X)) = f(1)$. ◇

Essentially, the augmented state $ex(S,X)$ represents the *view* [Pap86] operations have on the database in history X applied to state S. The following lemma can be easily proven by induction on the length of X and with the aid of the restriction that private entities are updated only once.

Lemma 2. *For all histories X, $(\forall e \in adb : (X(S))(e) = X_e(ex(S,X)))$.* □

B Proofs

In this section we present the proofs of Theorems 2 through 4. To do so we need to first state two lemmas, whose proofs are omitted for brevity.

Lemma 3. *If two histories, X and Y, R-commute, then for all augmented states S, there exists an augmented state S' such that:*

- $(\forall e : (X_e \circ Y_e)(S)\ \mathcal{R}_e\ (Y_e \circ X_e)(S'))$, *and*
- $(\forall e \in db : S(e) = S'(e))$, *i.e., the restrictions of S and S' to database states coincide.*

□

Lemma 4. *Let X be a history of $T, dep(T)$ and CT whose initial state is S. If X is EWSR, then for all entities e that are updated by both T and $dep(T)$ in X, either*

1. $X_e \equiv X_{dep(T),e} \circ X_{T,e} \circ X_{CT,e}$, *or*

2. $X_e \equiv X_{T,e} \circ X_{dep(T),e} \circ X_{CT,e}$.

□

B.1 Proof of Theorem 2

Proof. Let Y be a history of the transactions in $dep(T)$ that includes the same operations as in $X_{dep(T)}$ and in the same order (such a Y is a legitimate history since $dep(T)$'s programs are fixed). We prove a stronger claim then our proof obligation; we show that for some entities, $(Y(S))(e)\ R_e\ (X(S))(e)$, and for the rest $(Y(S))(e) = (X(S))(e)$. Since $dep(T)$ is fixed, for all database states S, $X_{dep(T)}(S) = Y(S)$. We proceed by examining equivalences (1) and (2) established by Lemma 4. (The case of an entity that is updated by only one of T and $dep(T)$ is trivial). In case equivalence (1) holds, by Constraint 1, $X_{T,e}$ and $X_{CT,e}$ cancel each other, and hence $(X(S))(e) = X_e(ex(S,X)) = X_{dep(T),e}(ex(S,X))$. Now, since $dep(T)$'s programs are fixed $X_{dep(T),e}(ex(S,X)) = Y_e(ex(S,Y)) = (Y(S))(e)$, and our claim is proved for this case. In case equivalence (2) holds, observe that $(X(S))(e) = (X_{dep(T),e} \circ X_{CT,e})(X_{T,e}(ex(S,X)))$. We can apply the R-commutativity assumption and Lemma 3 and observe that $(X_{CT,e} \circ X_{dep(T),e})(X_{T,e}(S'')) = S'(e)$ and $S'(e)\ R_e\ (X(S))(e)$. S'' coincides with S on database entities by Lemma 3. However, $S'(e) = X_{dep(T),e}(S'')$ by enforcing Constraint 1. Since $dep(T)$'s transactions have fixed programs, and S and S'' coincide on database entities the following holds, $S'(e) = Y_e(ex(S,Y)) = (Y(S))(e)$. Thus, $(Y(S))(e)\ \mathcal{R}_e\ (X(S))(e)$ in this case. □

B.2 Proof of Theorem 3

Proof. Let X be a history where T is a serialization point, and let S be its initial state. Since both T and CT are serialization points $X(S) = (X_T \circ X_{dep(T)} \circ X_{CT})(S) = (X_{dep(T)} \circ X_{CT})(X_T(S))$. By assumption, there exists a history Y of $dep(T)$ such that $(X_{CT} \circ Y)(X_T(S)) = Y(S)$ (the last equality is by Constraint 1), and $Y(S)\ \mathcal{R}\ X(S)$. Observe that $X_T \circ X_{CT} \circ Y$ is indeed a history since Y involves only $dep(T)$. □

B.3 Proof of Theorem 4

Proof. By Lemma 4, the R-commutativity assumption, and the reflexivity of $\mathcal{R}$, we can show that $(\forall e \in db : X_{dep(T),e}(S')\ \mathcal{R}_e\ (X(S))(e))$, where S' coincides with S on the database state. (This is similar to the proof of Theorem 2). Let Y be a history of the transactions in $dep(T)$ that includes the same operations as in $X_{dep(T)}$ and in the same order (such a Y is a legitimate history since $dep(T)$'s programs are linear). If we can show that $(\forall e \in db : Y_e(ex(S,Y))\ \mathcal{R}_e\ X_{dep(T),e}(S'))$, we can then use the transitivity of $\mathcal{R}$ to complete the proof.

Since all programs in $dep(T)$ are linear we can treat Y merely as a sequence of operations, regardless of the issuing transactions. Let $f_1 \circ \ldots \circ f_k$ be the sequence of all the operations of $dep(T)$ in the order of their appearance in X (and hence also in Y).

We show that $(\forall e \in adb : Y_e(ex(S,Y))\ \mathcal{R}_e\ X_{dep(T),e}(S'))$, where S' coincides with S on the database state by induction on k.

$\underline{k=0}$: $(\forall e \in adb : X_{dep(T),e}(S') = Y_e(ex(S,Y)) = S(e))$.

<u>Inductive step</u>: Let f_{n+1} transform $e \in adb$. The final value of database entities other then e is computed by a sequence of at most n operations. Therefore, we can apply the hypothesis of induction and get the following $(\forall e' \in adb : (e' \neq e) \Rightarrow (Y_{e'}(ex(S,Y))\ \mathcal{R}_e\ X_{dep(T),e'}(S')))$. Let us focus on e itself. We can say that $Y_e(ex(S,Y)) = f_{n+1}(S'')$ and $X_{dep(T),e}(S') = f_{n+1}(S''')$. Since each argument of f_{n+1} is computed using less than $n+1$ operations in both X and Y, we can apply the hypothesis of induction and get $(\forall e \in adb : S''(e)\ \mathcal{R}_e\ S'''(e))$. Since f_{n+1} preserves $\mathcal{R}$ we have completed the proof. □