# Rewriting Histories: Recovering from Malicious Transactions *

Peng Liu[1] Paul Ammann[2] Sushil Jajodia[2]

[1]Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD 21250
*pliu@umbc.edu*

[2]Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
{*pammann,jajodia*}*@isse.gmu.edu*

**Abstract**

We consider recovery from malicious but committed transactions. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of good transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. We develop an algorithm that rewrites execution histories for the purpose of backing out malicious transactions. Good transactions that are affected, directly or indirectly, by malicious transactions complicate the process of backing out undesirable transactions. We show that the prefix of a rewritten history produced by the algorithm serializes exactly the set of unaffected good transactions. The suffix of the rewritten history includes special state information to describe affected good transactions as well as malicious transactions. We describe techniques that can extract additional good transactions from this latter part of a rewritten history. The latter processing saves more good transactions than is possible with a dependency-graph based approach to recovery.

*Liu, Ammann, Jajodia were partially supported by Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-97-1-0139.

# 1 Introduction

Preventive measures sometimes fail. In the database context, some transactions that shouldn't commit do anyway. Undesirable committed transactions can arise from malicious activity by a well-equipped attacker in many circumstances, these transactions are referred as *malicious* transactions. In tightly integrated networks, the damage caused by malicious transactions can spread quickly from the initial source. In this paper, we focus on one aspect of planning for and responding to such damage; specifically, we develop a family of algorithms for rewriting execution histories that back out a set of malicious (but committed) transactions while preserving the work of *good* transactions, namely committed transactions that arise from legitimate activity. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of malicious transactions as well as good ones, and compensating transactions, whose utility depends on application semantics. We show that our approach is strictly better at saving good transactions than a dependency-graph based approach.

## 1.1 Information Warfare

Experience with traditional information systems security practices (INFOSEC) has shown that it is very difficult to adequately anticipate the abuse and misuse to which an information system will be subjected in the field. The focus of INFOSEC is prevention: security controls aim to prevent malicious activity that interferes with either confidentiality, integrity, or availability. However, outsiders (hackers) have proved many times that security controls can be breached in imaginative and unanticipated ways. Further, insiders have significant privileges by necessity, and so are in a position to inflict damage. Finally, the dramatic increase in internetworking has led to a corresponding increase in the opportunities for outsiders to masquerade as insiders. Network-based attacks on many systems can now be carried out from anywhere in the world. Although mechanisms such as firewalls reduce the threat of outside attack, in practice such mechanisms cannot eliminate the threat without blocking legitimate use as well. In brief, strong prevention is clearly necessary, but less and less sufficient, to protect information resources.

In response to problems with the INFOSEC approach, a complementary approach with an emphasis on survivability has emerged.[‡] This 'information warfare' (IW) perspective is that not only should vigorous INFOSEC measures be taken to defend a system against attack, but that some attacks

[‡]For a recent summary with an emphasis on the database context, see [AJMB97].

should be assumed to succeed, and that countermeasures to these successful attacks should be planned in advance. The IW perspective emphasizes the ability to live through and recover from attacks.

The timeline in an IW scenario includes traditional preventive measures to harden a system against attack, intelligence gathering by the adversary to detect weaknesses in the resulting system, attack by the adversary, and finally countermeasures to the attack. Typical countermeasure phases are attack detection, damage confinement and assessment, reconfiguration, damage repair, and fault treatment to prevent future similar attacks. In this paper, we focus on one specific countermeasure phase to an information attack, namely the damage repair phase.

Although the IW adversary may find many weaknesses in the diverse components of an information system, databases provide a particularly inviting target. There are several reasons for this. First, databases are widely used, so the scope for attack is large. Second, information in databases can often be changed in subtle ways that are beyond the detection capabilities of the typical database mechanisms such as range and integrity constraints. For example, repricing merchandise is an important and desirable management function, but it can easily be exploited for fraudulent purposes. Finally, unlike most system components, many databases are explicitly optimized to accommodate frequent updates. The interface provides the outside attacker with built in functions to implement an attack; all that is necessary is to acquire sufficient privileges, a goal experience has shown is readily achievable. Advanced authorization services can reduce such a threat, but never eliminate it, since insider attacks are always possible, and also since system administrators are only human, and hence prove to making mistakes in configuring and managing authorization services.

Integrity, availability, and (to a lesser degree) confidentiality have always been key database issues, and commercial databases include diverse set of mechanisms towards these ends. For example, access controls, integrity constraints, concurrency control, replication, active databases, and recovery mechanisms deal well with many kinds of mistakes and errors. However, the IW attacker can easily evade some of these mechanisms and exploit others to further the attack. For example, access controls can be subverted by the inside attacker or the outside attacker who has assumed an insider's identity. Integrity constraints are weak at prohibiting plausible but incorrect data; classic examples are changes to dollar amounts in billing records or salary figures. To a concurrency control mechanism, an attacker's transaction is indistinguishable from any other transaction. Automatic replication facilities and active database triggers can serve to spread the damage introduced by an attacker at one site to many sites. Recovery mechanisms ensure that commit-

ted transactions appear in stable storage and provide means of rolling back a database, but no attention is given to distinguishing legitimate activity from malicious activity. In brief, by themselves, existing database mechanisms for managing integrity, availability, and confidentiality are inadequate for detecting, confining, and recovering from IW attacks.

### 1.2 Contribution

The specific problem we address in this paper is 'how can one repair a database, given that a set of malicious transactions has been identified.' The identification of the set of malicious transactions is outside the scope of this paper. In an IW context, such identification takes place in an earlier countermeasure phase. For example, identification of an attacker may lead directly to the identification of a set of malicious transactions.

Our contribution is to provide an algorithm that rewrites an execution history so that malicious transactions are as near the end of the history as possible, given the read-write dependencies between transactions. The prefix of the rewritten history consists solely of good transactions; we show that this prefix is equivalent to using a write-read dependency graph to unwind malicious transactions and those good transactions that depend, directly or indirectly, on the malicious transactions. We then show how to use the latter part of the rewritten history to save additional good transactions.

Although we develop these algorithms to repair a database when some malicious activity happens, our methods can be easily extended to other applications where some committed transactions may also be identified undesirable, thus have to be backed out. For example

- In [JLM98], the use of isolation is proposed to protect systems from the damage caused by authorized but malicious users, masqueraders, and misfeasors, where the capacity of intrusion detection techniques is limited. In the database context, the basic idea is when a user is found suspicious, his transactions are redirected to an isolated database version, and if the user turns out to be innocent later, the isolated database version will be merged into the main database version. Since these two versions may be inconsistent, some committed transactions may have to be backed out to ensure the consistency of the database.
- During upgrades to existing systems, particularly upgrades to software. Despite efforts for planning and testing of upgrades, upgrade disasters occur with distressing regularity.[§] If a system communicates with the

[§]For some more spectacular examples, see Peter Neumann's RISKS digest in the news-

outside world, bringing the upgrade online with a hot standby running the old software isn't complete protection. Problems with an upgrade by one organization can easily affect separate, but cooperating organizations. Thus an incorrect upgrade at a given organization may result in an erroneous set of transactions at one or more cooperating organizations. In many cases, it is not possible simply to defer activity, and so during the period between the introduction of an upgrade and the recognition of an upgrade problem, erroneous transactions at these cooperating organizations commit. As a result, backing out these committed erroneous transactions is necessary.

- In partitioned distributed database systems, Davidson's optimistic protocol [Dav84] allows transactions to be executed within each partitioned group independently with communication failures existing between partitioned groups. As a result, serial history $H_i$ consisting of all transactions executing within group $P_i$ is generated. When two partitioned groups $P_1$ and $P_2$ are reconnected, $H_1$ and $H_2$ may conflict with each other. Therefore, some committed transactions may have to be backed out to resolve the conflicts and ensure the consistency of the database.

- In [GHOS96], J. Gray et al. state that update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up. To reduce this problem, a two-tier replication algorithm is proposed that allows mobile applications to propose tentative update transactions that are later applied to a master copy. The drawback of the protocol is that every tentative transaction must be reexecuted on the base node, thus some sensitive transactions may have given users inaccurate information and the work of tentative transactions is lost. In this situation, the strategy that when a mobile node is connected to the base node merges the mobile copy into the master copy may be better, however, in order to ensure the consistency of the master copy after the mergence, some committed transactions may have to be backed out.

### 1.3 Organization

The outline of the paper is as follows. In section 2 we give our model for rewriting and repairing histories after first describing the dependencies relevant to repair. In section 3 we give an algorithm to rewrite histories and show that it is equivalent to using a dependency-graph based approach. We turn to methods to save additional good transactions in section 4. In section

group `news:comp.risks` or the archive `ftp://ftp.sri.com/risks`.

5, we show how to prune a rewritten history so that a repaired history can be generated. We examine the relationships among the possible rewriting algorithms in section 6. In section 7, we show how to implement our rewriting methods in a realistic transaction processing system which is based on the Saga model[GMS87]. Section 8 describes related work, and we conclude in section 9.

# 2 Model

## 2.1 Assumptions

We assume that the histories to be repaired are serializable histories generated by some mechanism that implements a classical transaction processing model [BHG87]. We denote (committed) malicious or *bad* transactions in a history by the set $\mathbf{B} = \{B_{i1}, B_{i2}, ..., B_{im}\}$. We denote (committed) *good* transactions in a history by the set $\mathbf{G} = \{G_{j1}, G_{j2}, ..., G_{jn}\}$. Since recovery of uncommitted transactions is addressed by standard mechanisms, we consider a history $H$ over $\mathbf{B} \cup \mathbf{G}$. We define $<_H$ to be the usual partial order on $\mathbf{B} \cup \mathbf{G}$ for such a history $H$, namely, $T_i <_H T_j$ if $<_H$ orders operations of $T_i$ before conflicting operations of $T_j$ [BHG87].

We assume that the concurrency control mechanism provides an explicit serial history $H^s$ of history $H$. For example, the order of first lock release provides a serialization order for transactions scheduled by a strict two-phase locking mechanism. We denote the total order on the transactions in a serial history $H^s$ by $<^s_H$.

We assume the availability of read information for transactions in $H$. Only write information is kept in logs for traditional recovery purposes, but, as later discussion makes clear, read information is also necessary to unwind committed transactions. Read information can be captured in several ways, these approaches are discussed in section 7.

We assume that transactions do not issue blind writes. That is, if a transaction writes some data, the transaction is assumed to read the value first. Although the approach in this paper can be adapted to blind writes, doing so complicates the presentation. Also, we compare the results in this paper to those obtained by a dependency-graph based approach to recovery that also assumes no blind writes.

## 2.2 Syntactic Dependencies

One simple repair is to roll back a history $H^s$ until at least the first transaction in **B** and then try to reexecute transactions in **G** that were undone during the rollback. The drawback of this approach is that many good transactions may be unnecessarily undone and reexecuted. Consider the history $H_1$ over $(B_1, G_2)$ where

$B_1 : r[x]w[x]$
$G_2 : r[y]w[y]$
$H_1 : B_1\ G_2$

It is clear that $G_2$ need not be undone and reexecuted since it does not conflict with $B_1$. We formalize the notion that some - but not all - good transactions need to be undone and reexecuted in the usual way:

**Definition 1** Transaction $T_j$ is *dependent upon* transaction $T_i$ in a history $H$ if there exists a data item $x$ such that:

1. $T_j$ reads $x$ after $T_i$ has updated $x$; and
2. there are no transactions that update $x$ between the time $T_i$ updates $x$ and $T_j$ reads $x$.

Every good transaction that is dependent upon some bad transaction potentially needs to be undone and reexecuted. There are also other good transactions that also need be undone and reexecuted. Consider the history $H_2$ over $(B_1, G_2, G_3)$:

$B_1 : r[x]w[x]$
$G_2 : r[x]r[y]w[y]$
$G_3 : r[y]w[y]$
$H_2 : B_1\ G_2\ G_3$

$G_3$ is not dependent upon $B_1$, but it should be undone and reexecuted, because the value of $x$ which $G_2$ reads from $B_1$ may affect the value of $y$ which $G_3$ reads from $G_2$. This relation between $G_3$ and $B_1$ is captured by the transitive closure of the dependent upon relation:

**Definition 2** In a history, transaction $T_i$ *affects* transaction $T_j$ if the ordered pair $(T_j,\ T_i)$ is in the transitive closure of the *dependent upon* relation.

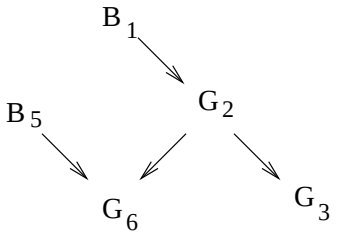


Figure 1: Dependency Graph for History $H_3$

It is convenient to define the *dependency graph* for a set of transactions $S$ in a history as $DG(S) = (V, E)$ in which $V$ is the union of $S$ and the set of transactions that are affected by $S$. There is an edge, $T_i \rightarrow T_j$, in $E$ if $T_i \in V$, $T_j \in (V - S)$, and $T_j$ is dependent upon $T_i$. Notice that there are no edges that terminate at elements of $S$; such edges are specifically excluded by the definition. As a result, every source node in $DG(\mathbf{B})$ is a bad transaction, and every non-source node in $DG(\mathbf{B})$ is a good transaction that reads some data, directly or indirectly, from at least one bad transaction. We refer to the transactions associated with the non-source nodes in $DG(\mathbf{B})$ as the *affected good transactions* or, more briefly, as the *affected transactions*. We denote the set of affected transactions as $\mathbf{AG}$.

As an example, consider the history $H_3$ over $(B_1, G_2, G_3, G_4, B_5, G_6)$:

$B_1 : r[x]w[x]$
$G_2 : r[x]w[x]r[y]w[y]$
$G_3 : r[y]w[y]$
$G_4 : r[z]w[z]$
$B_5 : r[z]w[z]$
$G_6 : r[y]w[y]r[z]w[z]$
$H_3 : B_1\ G_2\ G_3\ G_4\ B_5\ G_6$

$DG(\mathbf{B})$ is shown in Figure 1.

If a good transaction is not affected by any bad transaction (for example, $G_4$ in $H_3$), then the good transaction need not be undone and reexecuted. In other words, only the transactions in $DG(\mathbf{B})$ potentially need be undone, and only transactions in $\mathbf{AG}$ potentially need to be reexecuted. From the recovery perspective, the goal of a dependency-graph based approach to recovery is to first get $DG(\mathbf{B})$, then undo all these transactions.[¶]

[¶]Note to the referees: The dependency-graph based algorithm is part of a different paper that is being considered for

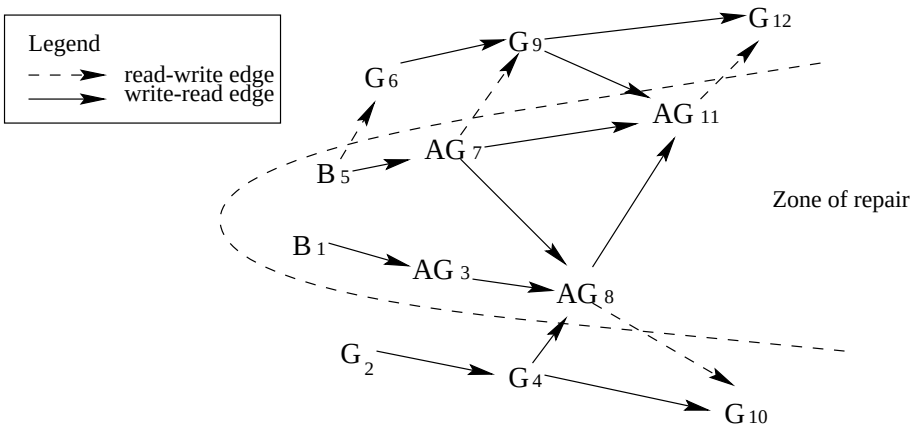


Figure 2: Zone of Repair

Figure 2 illustrates the dependency-graph based approach to backing out bad and affected transactions. In particular, it illustrates the importance of distinguishing between read-write and write-read dependencies during recovery. A read-write edge can leave the 'zone of repair' without causing the zone to expand. On the other hand a write-read edge potentially expands the zone. Note that due to the assumption of no blind writes, there are no write-write edges in the graph.

In this example, a possible history $H_4$ is

$$H_4 = B_1\ G_2\ AG_3\ G_4\ B_5\ G_6\ AG_7\ AG_8\ G_9\ G_{10}\ AG_{11}\ G_{12}$$

the set $\mathbf{AG} = \{AG_3,\ AG_7,\ AG_8,\ AG_{11}\}$, and the dependency-graph based recovery algorithm restores the before values for all data items written by transactions in the set $\mathbf{B} \cup \mathbf{AG}$. The result is a serializable history over $\mathbf{G} - \mathbf{AG}$:

$$H_{4r} = G_2\ G_4\ G_6\ G_9\ G_{10}\ G_{12}$$

The approach of rewriting histories developed in this paper has the advantage that it preserves ordering information for transactions in $\mathbf{B} \cup \mathbf{AG}$, thereby providing a basis for saving additional transactions in $\mathbf{AG}$.

publication. A description of the algorithm for recovery may be downloaded from the URL `http://www.isse.gmu.edu/faculty/pammann/papers/recovery.ps`

## 2.3 Rewriting Histories

For a serial history $H^s$, we *augment* $H^s$ with explicit database states so that the result is a sequence of interleaved transactions and database states. The sequence begins and ends with a state. The state that immediately precedes a transaction in $H^s$ is called the *before state*; the state that immediately follows a transaction in $H^s$ is called the *after state*. For an example, consider the augmented history

$$H_5^s = s_0\ B_1\ s_1\ G_2\ s_2$$

where

$$B_1 : \texttt{if x} > \texttt{0 then y} := \texttt{y} + \texttt{z} + \texttt{3}$$
$$G_2 : \texttt{x} := \texttt{x} - \texttt{1}$$

The states associated with $H_5^s$ are:

$$s_0 = \{\texttt{x} = \texttt{1};\ \texttt{y} = \texttt{7};\ \texttt{z} = \texttt{2}\}$$
$$s_1 = \{\texttt{x} = \texttt{1};\ \texttt{y} = \texttt{12};\ \texttt{z} = \texttt{2}\}$$
$$s_2 = \{\texttt{x} = \texttt{0};\ \texttt{y} = \texttt{12};\ \texttt{z} = \texttt{2}\}$$

In rewriting histories, the general goal is either to move bad transactions towards the end of a history or to move good transactions towards the beginning of a history. It turns out that the transformations do not necessarily result in a serializable history which is conflict-equivalent or view-equivalent to the original history[BHG87]. The lack of serializability is justified by the observation that bad transactions ultimately must be backed out anyway along with some or all of the affected transactions. Hence the serializability of such transactions is not a requirement.

The example above helps to clarify this point. The serial history $H_5^s$ is clearly not conflict-equivalent to the serial history $G_2B_1$ since there is a read-write dependency from $B_1$ to $G_2$. However, $G_2$ is not affected by $B_1$, and simply restoring $y$ with the appropriate value from the log not only repairs the damage caused by $B_1$, but preserves the effects of the good transaction $G_2$.

However, It turns out that rewriting histories for recovery purposes requires some care with respect to state-equivalence of histories. Two augmented histories $H_1^s$ and $H_2^s$ are *equivalent* if they are over the same set of transactions and the final states are identical.

To clarify this point, consider the above example again. After we make the transformation of exchanging the order of $G_2$ and $B_1$, $H_1^s$ is clearly not equivalent to the serial history $G_2 B_1$ since they result in different final states. At this situation, if $H_1^s$ has more transactions following $B_1 G_2$, i.e., $G_3 G_4 ... G_n$, then this transformation changes the before state of $G_3$. As a result, after the transformation the rewritten history may not be consistent any longer because the precondition of some $G_i$, $3 \leq i \leq n$, may not be satisfied any more. Even if the rewritten history is still consistent, the behaviors and effects of $G_3$, $G_4$, ..., and $G_n$ may have changed a lot, thus the original execution log may turn out to be useless. Moreover, the rewritten history usually can not result in the same final state, and the new final state is usually very difficult to get, thus semantics-based compensation is disabled. Therefore, keeping the equivalence of rewritten histories during a rewrite is essential to the success of the rewrite.

We approach this problem by decorating each transaction $T$ in an augmented history $H^s$ with special values for read purposes by $T$. The decoration is facilitated by the notation *fix* which is specified below.

**Definition 3** A *fix* for transaction $T_i$ in history $H^s$, denoted $F_i$, is a set of variables read by $T$ given values as in the original position of $T$ in $H^s$. That is, $F_i = \{(x_1, v_1), ..., (x_n, v_n)\}$, and $v_i$ is what $T_i$ read for $x_i$ in the original history.

The notation ${T_i}^{F_i}$ indicates that the values read by $T_i$ for variables in $F_i$ should not come from the before state of $T_i$, but from $F_i$.

To reduce notational clutter, we show just the variable names in $F_i$ and omit the associated values.

Consider the augmented history $H_5^s = s_0\ B_1\ s_1\ G_2\ s_2$ above. As discussed, the history

$$H_6^s = s_0\ G_2\ s_3\ B_1\ s_3$$

with

$$s_3 = \{\mathtt{x = 0;\ \ y = 7;\ \ z = 2}\}$$

results in a different value of $y$ in the final state, but the history

$$H_7^s = s_0\ G_2\ s_3\ B_1^{F_1}\ s_2$$

ends in final state $s_2$ for $F_1 = \{x\}$. States $s_1$ and $s_3$ differ in the value of $x$; this discrepancy is captured by $F_1$, where $x$ is associated with the value 1, which is the value $B_1$ read for $x$ in the original history $H_5^s$.

In what follows, each transaction $T_i$ is assumed to have an associated fix $F_i$. For ordinary serializable execution histories, each such fix $F_i = \emptyset$, the empty fix. In the example above, the two histories

$$H_5^s = s_0\ B_1^{\emptyset}\ s_1\ G_2^{\emptyset}\ s_2$$
$$H_7^s = s_0\ G_2^{\emptyset}\ s_3\ B_1^{\{x\}}\ s_2$$

are equivalent.

## 2.4 Repaired Histories

**Definition 4** Given a history $H^s$ over $\mathbf{B} \cup \mathbf{G}$, $H_r^s$ is a *repaired* history of $H^s$ if

1. $H_r^s$ is over some subset of $\mathbf{G}$, and
2. There exists some history $H_e^s$ over $\mathbf{B} \cup \mathbf{G}$ such that
   (a) $H_r^s$ is a prefix of $H_e^s$ and
   (b) $H_e^s$ and $H^s$ are equivalent.

Our notion of a repaired history is that only good transactions remain (condition 1) and further that there is some extension to the repair that captures exactly the same transformation to the database state as the original history (condition 2).

We note that the dependency-graph based approach satisfies the first part of the definition of a repaired history where the subset of $\mathbf{G}$ is $\mathbf{G} - \mathbf{AG}$. As an example, in figure 2 history $H_{4r}^s$ is a repair of $H_4^s$ since $H_{4r}^s$ is over $\{G_2, G_4, G_6, G_9, G_{10}, G_{12}\}$ which is a subset of $\mathbf{G}$ and the necessary history $H_{4e}^s$ exists:

$$H_{4e}^s = G_2\ G_4\ G_6\ G_9\ G_{10}\ G_{12}\ B_1^{F_1}\ AG_3^{F_3}\ B_5^{F_5}\ AG_7^{F_7}\ AG_8^{F_8}\ AG_{11}^{F_{11}}$$

for appropriate fixes $F_1, F_3, F_5, F_7, F_8$ and $F_{11}$. Details of how to construct fixes are discussed later in the paper.

Armed with a definition of repairs to histories, we are now ready to consider algorithms to construct them.

# 3 Basic Algorithm to Rewrite a History

## 3.1 Can-Follow Relation

We denote the set of items read or written by a transaction $T$ as $T.readset$ or $T.writeset$, and the set of items read or written by a sequence of transactions $R = T_1T_2...T_n$ as $R.readset$ or $R.writeset$. Due to our assumption of no blind writes, $R.writeset \subseteq R.readset$.

**Definition 5** Transaction $T$ *can follow* a sequence of transactions $R$ if

$$T.writeset \cap R.readset = \emptyset$$

There are some properties of can follow:

1. If $T_i.writeset$ is not empty, then transaction $T_i$ can not follow itself.
2. The fact that $T_i$ can follow transaction $T_j$ and $T_j$ can follow transaction $T_k$ does not imply that $T_i$ can follow $T_k$.
3. Read-only transactions can follow any transaction.

The can follow relation captures the notion that a transaction $T$ can be moved to the right past a sequence of transactions $R$ if no transaction in $R$ reads from $T$. The can follow relation ensures then the cumulative effects of the transactions in $R$ on the database state are identical both before and after $T$ is moved. The following lemma shows that the can follow relation can be repeatedly used to rewrite a history.

**Lemma 1** Transaction $T$ can follow a sequence of transactions $R$ iff $T$ can follow every transaction in $R$.

**Proof:** *if:* For every transaction $T_i$ in $R$, $T.writeset \cap T_i.readset = \emptyset$ because $T$ can follow $T_i$. Therefore $T.writeset \cap R.readset = \emptyset$, so $T$ can follow $R$.

*only if:* By contradiction, assume there is a transaction $T_i$ in $R$ such that $T$ cannot follow $T_i$, then $T.writeset \cap T_i.readset \neq \emptyset$. Therefore, $T.writeset \cap R.readset \neq \emptyset$, which contradicts the assumption that $T$ can follow $R$. □

### 3.2 Can-Follow Rewriting

The can follow relation can be used to rewrite a history to move transactions in $\mathbf{G} - \mathbf{AG}$ to the beginning of the history, namely, move transactions in $\mathbf{B} \cup \mathbf{AG}$ backwards.

**Algorithm 1** Can-Follow Rewriting
**Input:** the serial history $H^s$ to be rewritten and the set $\mathbf{B}$ of bad transactions.
**Output:** a rewritten history with transactions in $\mathbf{G} - \mathbf{AG}$ preceding transactions in $\mathbf{B} \cup \mathbf{AG}$.
**Method:** Scan forward from the first good transaction after $B_1$ until the end of $H^s$, for each transaction $T$

**case** $T \in \mathbf{B}$ skip it;
**case** $T \in \mathbf{G}$
**if** each transaction between $B_1$ and $T$ (including $B_1$) can follow $T$, then move $T$ to the position immediately preceding $B_1$.

Algorithm 1 does not describe how to compute the *fix* with any transaction which has some transaction being moved to the left of it. The reason is that repair can simply be accomplished by undo. However, if we want to save some of the transactions in $\mathbf{AG}$ then we need to maintain the *fix* information for these transactions. Fixes are computed as follows:

**Lemma 2** Suppose transaction $T$ can follow sequence $R$ in history $H_1^s = s_0\ T^{F_1}\ s_1\ R\ s_2$. Then for fix

$$F_2 = F_1 \cup (T.readset \cap R.writeset)$$

history $H_2^s = s_0\ R\ s_3\ T^{F_2}\ s_2$ is equivalent to $H_1^s$. The values associated with each data item in the fixes are those originally read by $T$.

**Proof:** Consider some database item $x \in s_2$. $x$ is not an element of both $R.writeset$ and $T.writeset$ since otherwise the relation $T$ can follow $R$ would not hold. If $x$ is an element of $R.writeset$, then the value computed by $R$ for $x$ is the same in both $H_1^s$ and $H_2^s$ since $R$ does not read from $T$. If $x$ is an element of $T.writeset$, then the value computed by $T$ for $x$ is the same in both $H_1^s$ and $H_2^s$ since $T$ reads identical values for elements in $T.readset$ in both histories, courtesy of fixes $F_1$ and $F_2$, respectively. If $x$ is not an element of either $T.writeset$ or $R.writeset$, then the order of $T$ and $R$ is irrelevant to the value of $x$. □

The correctness of Algorithm 1 is specified as follows.

**Theorem 1** Given a history $H^s$, Algorithm 1 produces a history $H_e^s$ with a prefix $H_r^s$ such that:

1. All and only transactions in $\mathbf{G} - \mathbf{AG}$ appear in $H_r^s$.
2. $H_e^s$ and $H^s$ order transactions in $\mathbf{G} - \mathbf{AG}$ identically. And they order transactions in $\mathbf{B} \cup \mathbf{AG}$ identically.
3. The fix associated with each transaction in $H_r^s$ is empty.
4. $H^s$ and $H_e^s$ are equivalent. And $H_r^s$ is a repaired history of $H^s$.

**Proof:** (1) We first show that when a transaction $T_1 \in \mathbf{G} - \mathbf{AG}$ is scanned every transaction between $B_1$ and $T_1$ is in $\mathbf{B} \cup \mathbf{AG}$. Assume this is not the situation and $T_2$ is the first one between $B_1$ and $T_1$ which belongs to $\mathbf{G} - \mathbf{AG}$. According to the algorithm when $T_2$ was scanned it should be moved to the left of $B_1$, which is a contradiction. We second show that no transactions in $\mathbf{AG}$ will be moved to the left of $B_1$ at the end of the algorithm. Assume this is not the situation and $T_2$ is the first one in $\mathbf{AG}$. According to the definition of $\mathbf{AG}$, when $T_2$ is scanned there is at least one transaction between $B_1$ and $T_2$ which can not follow $T_2$, which is a contradiction. We last show that no transactions in $\mathbf{B}$ will be moved to the left of $B_1$ because they will never be moved at all. Therefore, after the rewrite all and only transactions in $\mathbf{G} - \mathbf{AG}$ are moved to the left of $B_1$.
(2) Since Algorithm 1 moves transactions in $\mathbf{G} - \mathbf{AG}$ to the left of $B_1$ according to their orders in $H^s$, so they are ordered by $H_r^s$ and $H^s$ identically. Since transactions in $\mathbf{B} \cup \mathbf{AG}$ are never moved in Algorithm 1, so they are ordered by $H_r^s$ and $H^s$ identically.
(3) Since there are no transactions which are moved to the left of any transaction in $\mathbf{G} - \mathbf{AG}$ in Algorithm 1, transactions in $\mathbf{G} - \mathbf{AG}$ will have empty fixes.
(4) Follows from Lemma 2 and Definition 4. □

In realistic applications, although Lemma 2 gives users a sound approach to capture fixes in Algorithm 1, it is not efficient in many cases since whenever a transaction $T_i$ is moved to the left of another transaction $T_j$, $F_j$ may need be augmented. A better way to compute fixes is as follows:

**Lemma 3** For any history $H^s$, assume rewriting $H^s$ using Algorithm 1 generates a history $H_e^s$ with a prefix $H_r^s$ ($H_e^s$ typically looks like: $G_{j1}...G_{jn}\ B_{i1}^{F_{i1}}\ AG_{k1}^{F_{k1}}...B_{im}^{F_{im}}...AG_{kp}^{F_{kp}}$. The subhistory before $B_{i1}^{F_{i1}}$ is $H_r^s$ ),

and assume all the fixes are computed according to Lemma 2 during the rewriting, then the history $H_e^{s\prime}$, generated by replacing each non-empty fix $F_i$ in $H_e^s$ with $F_i' = T_i.readset - T_i.writeset$, is equivalent to $H_e^s$.

**Proof:** According to Theorem 1, the fix associated with each transaction in $H_r^s$ is empty. Given a transaction $T_i$ in $\mathbf{B} \cup \mathbf{AG}$, for each item $x$ in $F_i' - F_i$, showing that the value of $x$ in the before state of $T_i$ in $H_e^s$ is the same as that in $H^s$ gives the proof. Assume $G_j$ is the first transaction which was moved to the left of $T_i$, then before $G_j$ was moved, the before state of $T_i$ in the rewritten history is the same as that in $H^s$ because at this point, according to Lemma 2, the subhistory of the rewritten history which ends with the transaction immediately preceding $T_i$ is equivalent to the corresponding subhistory of $H^s$. After $G_j$ is moved to the left of $T_i$, the value of $x$ would not be changed since otherwise $x$ must be in $F_i$. Although $G_j$ might be further pushed through some other transactions in $\mathbf{B} \cup \mathbf{AG}$ to the beginning of the history, the value of $x$ in the before state of $T_i$ will not be changed. The reason follows from Lemma 2. □

Lemma 3 enables us to separate computing fixed from transforming histories. Fixes can be computed after all of the transformations. Based on Lemma 3, the fix of transaction $T_i$ can be captured in two ways: one is to first get the read and write sets of $T_i$, then compute $T_i.readset - T_i.writeset$; the other is to let each transaction $T_i$ write the set $T_i.readset - T_i.writeset$ as a record to the database when it is executed, then when we rewrite $H^s$ all of the fixes can be gotten directly from the database.

### 3.3 Significance of Algorithm 1

The major result of this section is an equivalence theorem between the effect of a dependency-graph based algorithm and the history produced by Algorithm 1. The dependency-graph based algorithm computes the set $\mathbf{B}.writeset \cup \mathbf{AG}.writeset$ and restores the values of all elements in this set. In particular, the theorem shows that the optimizations in the following section are strict improvements over the dependency-graph based algorithm.

**Theorem 2** Given $H^s$, let $H_r^s$ be the serial history produced by eliminating all transactions in $\mathbf{B} \cup \mathbf{AG}$ as in the dependency-graph based algorithm. Given $H^s$, let $H_e^s$ be the result of Algorithm 1. Then $H_r^s$ is a prefix of $H_e^s$.

**Proof:** Direct corollary of Theorem 1. □

# 4 Saving Additional Good Transactions

In this section, we show how to integrate the notion of commutativity with Algorithm 1 to save not only the transactions in $\mathbf{G} - \mathbf{AG}$, but potentially transactions in $\mathbf{AG}$ as well.

## 4.1 Motivating Example

Consider the following history:

$H_8 : B_1 G_2 G_3$
$B_1$: if $u > 10$ then $x := x + 100, y := y - 20$
$G_2$: $u := u - 20$
$G_3$: $x := x + 10, z := z + 30$

According to Algorithm 1, which rewrites based on can follow, $G_3$ needs to be undone since it reads from $B_1$ and hence is an element of $\mathbf{AG}$. The result of Algorithm 1 is the history $H_e^s = G_2 B_1^{\{u\}} G_3$. Note that $G_3$ *commutes backward through* $B_1^{\{u\}}$ for any value of $u$[‖], and so an equivalent history is $G_2 G_3 B_1^{\{u\}}$. Compensation for $B_1^{\{u\}}$ can be applied directly to this history, but an undo approach requires more care. Suppose we decide to undo $B_1$ by restoring the before values for $x$ and $y$ from the log entries for $B$. After $B$ is undone the value of $u$ is unchanged because only $G_1$ updates $u$. The value of $z$ is unchanged because only $G_3$ updates $z$. The effect of $G_3$ on $x$ is wiped out because both $G_3$ and $B$ update $x$, and after $B$ is undone $x$ no longer reflects the effects of $G_3$. However $x$ can be repaired by re-executing the corresponding part of $G_3$'s code, that is, $x = x + 10$, and the cumulative effect is that of history $G_2 G_3$. We call this last step an *undo-repair* action. Both the undo approach and the compensation approach to repair are discussed in detail in section 5.

The presence of fixes for transactions limits the extent to which commutativity can be applied. We illustrate this point with an example, and then define a more restrictive notion of commutativity called *can precede* that takes fixes into account.

$H_9 : \ s_0 \ T_1 \ s_1 \ T_2 \ s_2 \ T_3 \ s_3$

[‖]We adapt the notation of commutativity from [LMWF94, Wei88]. Transaction $T_2$ *commutes backward through* transaction $T_1$ if for any state $s$ on which $T_1 T_2$ is defined, $T_2(T_1(s)) = T_1(T_2(s))$; $T_1$ and $T_2$ *commute* if each commutes backward through the other. Note that one-sided commutativity (i.e., commutes backward through) is enough for our purpose.

$T_1$: if $y > 200$ then $x := x + 100$ else $x := x * 2$
$T_2$: $y := y + 100$
$T_3$: if $y > 200$ then $x := x - 10$ else $x := x/2$

$T_1$ can follow $T_2$ with fix $F_1 = \{y\}$ for $T_1$. Although $T_3$ commutes backward through $T_1$, $T_3$ does not commute backward through $T_1^{F_1}$, because the value of $x$ produced by $T_1^{F_1}$ depends on the value of y in the fix $F_1$. For example, if the initial value of $x$ is 100 and fix value of $y$ is 150, then the final value of $x$ in history $T_2T_1^{F_1}T_3$ is 190, but the final value of $x$ in history $T_2T_3T_1^{F_1}$ is 180.

The example shows that sometimes a fix can interfere with the commutativity of transactions. This motivates our definition of *can precede*:

**Definition 6** A transaction $T_2$ *can precede* a transaction $T_1$ for fix $F$ if for any assignment of values to the variables in $F$ and for any state $s_0 \in \mathbf{S}$ on which $T_1^F T_2$ is defined,

1. $T_2T_1^F$ is defined on $s_0$, and
2. The same final state is produced by $T_1^F T_2$ and $T_2T_1^F$.

## 4.2 Can-Follow and Can-Precede Rewriting

We present a repair algorithm which integrates both can-follow and can-precede.

**Algorithm 2** Can-Follow and Can-Precede Rewriting
**Input:** the history $H^s$ to be repaired.
**Output:** the repaired history $H_r^s$.
**Method:** Scan $H^s$ forward from the first good transaction after $B_1$ until the end of $H^s$, for each transaction $T$
**case** $T \in \mathbf{B}$ skip it;
**case** $T \in \mathbf{G}$
**if** for each transaction $T'$ between $B_1$ and $T$(including $B_1$), either $T'$ can follow $T$ or $T$ can precede $T'$, then move $T$ to the position immediately preceding $B_1$.
As $T$ is pushed through each such $T'$ between $B_1$ and $T$ to the left of $B_1$
**if** $T'$ can follow $T$, then push $T$ to the left of $T'$ and
modulate the fix of $T'$ correspondingly according to Lemma 2;
**else** push $T$ to the left of $T'$.

The correctness of Algorithm 2 is specified as follows.

**Theorem 3** Given a history $H^s$, Algorithm 2 produces a history $H_e^s$ with a prefix $H_r^s$ such that:

1. Every transaction in $\mathbf{G} - \mathbf{AG}$ appears in $H_r^s$.
2. $H_e^s$ and $H^s$ order transactions in $H_r^s$ identically. And they order transactions in $H_e^s - H_r^s$ identically.
3. The fix associated with each transaction in $H_r^s$ is empty.
4. $H^s$ and $H_e^s$ are equivalent. And $H_r^s$ is a repaired history of $H^s$.

**Proof:** The proof of statements (1), (2), and (3) is similar to that of Theorem 1.
(4) follows from Lemma 2, Definition 6 and Definition 4. □

In Algorithm 1, Lemma 3 provides an efficient way to compute fixes. However, Lemma 3 may not hold for Algorithm 2 if the system does not have the following property.

**Property 1** Transaction $T_j$ can precede transaction $T_i$ for a fix $F_i$ only if $(T_i.readset - T_i.writeset - F_i) \cap T_j.writeset = \emptyset$ and $(T_j.readset - T_j.writeset) \cap T_i.writeset = \emptyset$.

It should be noticed that Property 1 is not a strict requirement, and it usually holds for most of the transaction processing systems. The reason is: if $T_j$ writes an item $x$ in $T_i.readset - T_i.writeset - F_i$, then $x$ can have different values in the before states of $T_i$ in sequences $T_i^{F_i} T_j$ and $T_j T_i^{F_i}$ respectively. Since $x$ is not in $F_i$, $T_i$ can read different values of $x$ in the two sequences. Since the value of $x$ typically affects the values of some other items updated by $T_i$, the two sequences usually can not generate the same final state. For similar reasons, if $(T_j.readset - T_j.writeset) \cap T_i.writeset \neq \emptyset$, then $T_i^{F_i} T_j$ and $T_j T_i^{F_i}$ usually can not generate the same final state.

**Lemma 4** Lemma 3 holds for Algorithm 2 if the system has Property 1.

**Proof:** The proof is similar to that of Lemma 3 except the situation when $T_j$ is moved to the left of $T_i$ based on the relation that $T_j$ can precede $T_i$. At this point, for each item $x$ in $F_i' - F_i$, since the system has Property 1, $T_j$ will not write $x$, so the value of $x$ in $F_i'$ is still the same as that in the before state of $T_i$ after the rewrite. This completes the proof. □

### 4.3 Invert and Cover

In this section, we introduce two semantic relationships between transactions, namely, *Invert* and *Cover*, and show how they can be exploited to enhance repair.

If transaction $T_2$ *inverts* $T_1$, then any history of the form: $s_0...T_1\ T_2$ ... is equivalent to the same history with $T_1T_2$ omitted; if $T_2$ *covers* $T_1$, then any history of the form: $s_0...T_1\ T_2$ ... is equivalent to the same history with $T_1$ omitted. If $T_2$ covers $T_1$, then $T_2$ covers $T_1^{F_1}$ for any $F_1$, but this is not the case for invert.

**Definition 7** Let $P$ and $Q$ be two sequences of transactions. $Q$ *inverts* $P$ if for any state $s_0$ such that history $s_0\ P\ Q$ is feasible, $Q(P(s_0)) = s_0$.

**Definition 8** Let $P$ and $Q$ be two sequences of transactions. $Q$ *covers* $P$ if for any state $s_0$ such that history $s_0\ P\ Q$ is feasible, $Q(P(s_0)) = Q(s_0)$.

The rewriting algorithm which exploits these two relations is described below.

**Algorithm 3** Can-Follow, Can-Precede, Cover, and Invert Rewriting
**Input:** the history $H^s$ to be repaired.
**Output:** the repaired history $H_r^s$.
**Method:** Scan $H^s$ forward from the first good transaction after $B_1$ until the end of $H^s$, for each transaction $T$

**case** $T \in \mathbf{B}$ skip it;
**case** $T \in \mathbf{G}$
**if** for each transaction $T'$ between $B_1$ and $T$(including $B_1$), either $T'$ can follow $T$, or $T$ can precede $T'$, or $T$ inverts $T'$, or $T$ covers $T'$, then move $T$ to the position immediately preceding $B_1$. As $T$ is pushed through each $T'$ between $B_1$ and $T$ to the left of $B_1$
**if** $T$ covers $T'$, then remove $T'$ from the history;
**elseif** $T'$ can follow $T$, then push $T$ to the left of $T'$ and modulate the fix of $T'$ correspondingly according to Lemma 2;
**elseif** $T$ can precede $T'$, then push $T$ to the left of $T'$;
**else** remove both $T$ and $T'$ from the history.

For similar reasons, Lemma 3 can also be exploited to capture fixes in Algorithm 3 if the system has Property 1. The correctness of Algorithm 3 is specified as follows. The proof is similar to Theorem 1 and Theorem 3, thus omitted.

**Theorem 4** Given a history $H^s$, Algorithm 3 produces a history $H_e^s$ with a prefix $H_r^s$ such that:

1. $H_e^s$ and $H^s$ order transactions in $H_r^s$ identically. And they order transactions in $H_e^s - H_r^s$ identically.

2. The fix associated with each transaction in $H_r^s$ is empty.

3. Every transaction in $H_e^s$ is in $H^s$.

4. The final states of $H^s$ and $H_e^s$ are identical. And $H_r^s$ is a repaired history of $H^s$.

# 5 Pruning Rewritten Histories

After a rewritten history $H_e^s$ with a prefix $H_r^s$ , which is the repaired history, is generated from $H^s$, we need to prune $H_e^s$ such that the effects of all the transactions in $H_e^s - H_r^s$ are removed. Pruning $H_e^s$ generates $H_r^s$. If $H_e^s$ is produced by Algorithm 1, then the pruning can be easily done by undoing each transaction in $H_e^s - H_r^s$. However, if $H_e^s$ is produced by Algorithm 2 or Algorithm 3, undo does not give the pruning in most cases.

In this section, two pruning approaches are presented. The compensation approach removes the effect of each transaction $T_i^{F_i}$ in $H_e^s - H_r^s$ by executing the *fixed* compensating transaction of $T_i$, however, compensating transactions may not be specified in some systems. The undo approach prunes $H_e^s$ by building and executing a specific undo-repair action for each affected transaction in $H_r^s$. It is a syntactic approach, but it imposes some restrictions on transaction programs.

## 5.1 The Compensation Approach

We denote the compensating transaction of transaction $T_i$ as $T_i^{-1}$ [GM83, GMS87, KLS90]. $T_i^{-1}$ semantically undoes the effect of $T_i$. It is reasonable to assume that $T_i^{-1}.writeset \subseteq T_i.writeset$, and for simplicity we further assume that every transaction $T_i$ has a compensating transaction.

After Algorithm 2 or Algorithm 3, a typical rewritten history $H_e^s$ with a prefix $H_r^s$ looks like (note that $B_{i1}$ could be covered or inverted, and $H_e^s$ can also end with a bad one): $G_{j1}...AG_{h1}...G_{jq}...AG_{hk}\ B_{i1}^{F_{i1}}\ AG_{h(k+1)}^{F_{h(k+1)}}...B_{im}^{F_{im}}...AG_{hp}^{F_{hp}}$. The subhistory before $B_{i1}^{F_{i1}}$ is $H_r^s$. Based on $H_e^s$, compensation is a simple way to get the repaired history $H_r^s$. However, executing the compensating transaction sequence $AG_{hp}^{-1}...B_{im}^{-1}...AG_{h(k+1)}^{-1}\ B_{i1}^{-1}$ on the final state of $H^s$

can not generates $H_r^s$ in most cases because the transactions we need to compensate are usually associated with a non-empty fix. Fixes must be taken into account for the compensation to be correct.

**Definition 9** The *fixed compensating transaction* of $T_i^{F_i}$, denoted $T_i^{(-1,F_i)}$, is the regular compensating transaction of $T_i$ (denoted $T_i^{-1}$) associated with the same fix $F_i$.

The effects of $T_i^{F_i}$ can be removed by executing $T_i^{(-1,F_i)}$, this is justified by the following lemma.

**Lemma 5** Transaction $T_i^{F_i}$ can be *fix compensated*, that is, for every consistent state $s_1$ on which $T_i^{F_i}$ is defined, $T_i^{(-1,F_i)}(T_i^{F_i}(s_1)) = s_1$, if $F_i \cap T_i.writeset = \emptyset$.

**Proof:** Since $F_i \cap T_i.writeset = \emptyset$, $T_i^{-1}.writeset \subseteq T_i.writeset$, so $F_i \cap T_i^{-1}.writeset = \emptyset$. Therefore, neither $T_i$ nor $T_i^{-1}$ will update any item in $F_i$. Let $s_2 = T_i^{F_i}(s_1)$. For each item $x$ in $F_i$ we replace the values of $x$ in states $s_1$ and $s_2$ with the value of $x$ in $F$, thus two new states are generated (denoted $s_1'$ and $s_2'$ respectively). It is clear that $T_i^{-1}(s_2') = s_1'$. Since the differences between $T_i^{-1}(s_2')$ and $T_i^{(-1,F_i)}(s_2)$ are only with the values of the items in $F_i$ which are neither updated by $T_i^{-1}$, nor updated by $T_i^{(-1,F_i)}$, so $T_i^{(-1,F_i)}(s_2) = s_1$. This completes the proof. □

A rewritten history $H_e^s$ can be *fix compensated* if every transaction in $H_e^s$ can be fix compensated. Lemma 5 shows that every $H_e^s$ produced by Algorithm 2 or Algorithm 3 can be fix compensated because for each transaction $T_i$ in $H_e^s$ which is associated with a non-empty fix $F_i$, $F_i \cap T_i.writeset = \emptyset$ always holds. The pruning algorithm by compensation therefore is straightforward: based on the final state of $H^s$, executing the fixed compensating transaction for each transaction in $H_e^s - H_r^s$ in the reverse order as they are in $H^s$.

## 5.2 The Undo Approach

As stated above, after Algorithm 2 or Algorithm 3, a typical rewritten history $H_e^s$ looks like: $G_{j1}...AG_{h1}...G_{jq}...AG_{hk}\ B_{i1}^{F_{i1}}\ AG_{h(k+1)}^{F_{h(k+1)}}...B_{im}^{F_{im}}...AG_{hp}^{F_{hp}}$. As shown in $H_8$, undoing transactions in $H_e^s - H_r^s$ can not generate $H_r^s$ in most cases. However, building and executing the undo-repair actions for the

affected transactions in $H_r^s$, namely $AG_{h1}, ..., AG_{hk}$, after these undo operations can generate $H_r^s$. For example, in $H_8$, executing the undo-repair action, $x = x + 10$, for $G_3$ after $B$ is undone can produce the effect of history $G_2G_3$.

To build the undo-repair actions for $AG_{h1}, ..., AG_{hk}$, we need to do two things:

1. Abstract the code for each undo-repair action from the source code of the corresponding affected transaction.
2. Assign appropriate values for some specific data items accessed by these undo-repair actions.

Our algorithm described below is based on the following assumptions about transactions:

- a transaction is composed of a sequence of statements, each of which is either:
  - An operation;
  - A conditional statement of the form: **if** $c$ **then** $SS1$ **else** $SS2$, where $SS1$ and $SS2$ are sequences of statements, and $c$ is a predicate;
- each statement can update at most one data item;
- each data item is updated only once in a transaction;

**Algorithm 4** Build Undo-repair Actions
**Input:** an affected transaction $AG_k$.
**Output:** the undo-repair action $URA_k$ for $AG_k$.
**Method:**
1. Copy the codes of $AG_k$ to $URA_k$. Assign $URA_k$ with the same input parameters and the same values associated with them as $AG_k$.
2. Parse $URA_k$. For each statement to be scanned
    **case** it is a read statement, keep it;
    **case** it is an update statement of the form: $x := f(x, y_1, y_2, ...y_n)$ where $f$ specifies the function of the statement, $y_1, ..., y_n$ are the data items used in the statement. Some input parameters may also be used in the statement, but they are not explicitly stated here.
        **if** $x$ has not been updated by any other transaction in $\mathbf{B} \cup \mathbf{AG}$
            Remove the statement from $URA_k$;
        **elseif** $x$ has not been updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes $AG_k$ in $H^s$
            Replace the statement with: $x := AG_k.afterstate.x$,
            that is, get the value of $x$ from the after state of $AG_k$ in $H^s$;
        **else** for each $y_i$ (including $x$)

**if** $y_i$ has not been updated by any preceding statement and has not been updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes $AG_k$ in $H^s$
Bind $y_i$ with $AG_k.beforestate.y$;

3. Reparse $URA_k$. Remove every read statement which reads some item never used in an update statement of $URA_k$, or reads some item $y$ used in one or more update statements but $y$ is bound with a value in these statements.

It should be noticed that when we execute an undo-repair action $URA_k$, for each update statement $x = f(x, y_1, y_2, ...y_n)$ of $URA_k$, if $y_i$ is not bound then we get the value of $y_i$ from the current database state, otherwise, the bound value should be used.

The correctness of the undo approach is specified as follows.

**Theorem 5** For any rewritten history $H_e^s$ generated by Algorithm 2 or Algorithm 3, after all transactions in $H_e^s - H_r^s$ are undone, executing the undo-repair actions which are generated by Algorithm 4 for the affected transactions in $H_r^s$, in the same order as their corresponding affected transactions are in $H_r^s$, produces the same effect of $H_r^s$.

**Proof:** Showing that each item $x$ updated by an transaction in $H_e^s$ is restored to the value as generated by $H_r^s$ after the repair gives the proof.

If $x$ has never been updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$, then the value of $x$ will be correctly restored because an unaffected transaction $G_i$ can only read items from other unaffected transactions thus $G_i$'s updates will not be affected by transactions in $\mathbf{B} \cup \mathbf{AG}$.

Otherwise, assume $x$ has been updated by $k$ transactions in $\mathbf{B} \cup \mathbf{AG}$, that is, $T_{i1}, ..., T_{ik}$, $T_{ip} <_H^s T_{iq}$ if $p < q$. Note that after $x$ has been updated by $T_{i1}$, $x$ will not then be updated by any unaffected transaction. If $k = 1$, that is, there is only one such transaction. At this point, if $T_{i1}$ is in $H_e^s - H_r^s$ then after the undoes the value of $x$ will be correctly restored; otherwise, $T_{i1}$ is in $H_r^s$. Since $H_e^s$ is equivalent to $H^s$, so the value of $x$ in the final state of $H_r^s$ is the same as that in the after state of $T_{i1}$ in $H^s$. Hence in Algorithm 4 the corresponding update statement is removed.

When $k > 1$, if no such transaction is in $H_r^s$ then after the undoes the value of $x$ will be correctly restored. Otherwise, assume $T_{j1}$ is in $H_r^s$, then when $URA_{j1}$ is executed, $x := T_{j1}.afterstate.x$, according to Algorithm 4. This restores the value of $x$ to that generated by the subhistory of $H_r^s$ which ends with $T_{j1}$, because in rewriting when $T_{j1}$ is moved into $H_r^s$, the subhistory $H_1$ of $H^s$ which ends with $T_{j1}$ is equivalent to the subhistory $H_2$ of the rewritten history at that time which ends with the transaction immediately preceding $T_{j1}$ before the move, and $T_{j1}$ is the last transaction in $H_2$ that updates $x$.

Assume $T_{jl}$ ($l > 1$) is in $H_r^s$, if there is another such transaction $T_{jm}$ in $H_r^s$ such that $1 \leq m < l$ and no other such transactions stay between $T_{jm}$ and $T_{jl}$, then in the update statement $x := f(x, y_1, y_2, ...y_n)$ of $URA_{jl}$, the value of $x$ for read purpose should be got from the state after $URA_{jm}$ is executed; otherwise, there is no such $T_{jm}$, thus transactions $T_{j1}, ..., T_{j(l-1)}$ will all be undone, hence the value of $x$ in the above statement should be got from the state after $T_{j1}$ is undone.

As for $y_i$ in the above update statement, if $y_i$ has been updated by a preceding statement in $URA_{jl}$, then the updated value should be used. Otherwise, if $y_i$ has been updated by some transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes $T_{jl}$ in $H^s$, then according to the above discussion, the value of $y_i$ should be got from the state before $URA_{jl}$ is executed; Otherwise, the value of $y_i$ should be got from the before state of $T_{jl}$ in $H^s$. At this situation, getting the value of $y_i$ from the state before $URA_{jl}$ is executed can not ensure the correctness because it is possible that there is a transaction $T_i$ such that $T_i$ updates $y_i$, $T_i$ follows $T_{jl}$ in $H^s$, $T_i$ is in $\mathbf{B} \cup \mathbf{AG}$ and $T_i$ is in $H_r^s$. At this point, the value of $y_i$ updated by $T_i$ will not be undone.

Since the values of $x, y_1, y_2, ...y_n$ in the above statement are correctly captured, so the above statement can correctly restore the value of $x$ to that generated by the subhistory of $H_r^s$ which ends with $T_{jl}$. By induction on $l$, $1 \leq l \leq k$, the above claim holds. □

# 6 Relationships between Rewriting Algorithms

Rewriting can save more good transactions than is possible with a dependency-graph based approach to recovery. For a history $H^s$ to be repaired, we will let DGR($H^s$) and CFR($H^s$) represent the sets of saved transactions after $H^s$ is repaired using a dependency-graph based approach and can-follow rewriting (Algorithm 1), respectively. FPR($H^s$) and FPCI($H^s$) will be used to represent the sets of saved transactions after $H^s$ is repaired using can-follow and can-precede rewriting (Algorithm 2) and can-follow, can-precede, cover and invert rewriting (Algorithm 3), respectively.

Theorem 2 shows that for any history $H^s$, DGR($H^s$) = CFR($H^s$).

**Theorem 6** For any history $H^s$, CFR($H^s$) $\subseteq$ FPR($H^s$) $\subseteq$ FPCI($H^s$). The converse is not, generally, true.

**Proof:** Follows from Algorithm 1, Algorithm 2, and Algorithm 3. □

Commutativity can be directly used to rewrite histories without being integrated with can-follow rewriting. Let CR($H^s$) and CBTR($H^s$) represent the sets of saved transactions after $H^s$ is repaired using the two rewriting algorithms which are based on the *commute* relation and the *commutes backward through* relation between transactions, respectively. These two algorithms can be easily adapted from Algorithm 1 by checking the *commute* and *commutes-backward-through* relation between transactions respectively, instead of *can-follow*.

**Theorem 7** For any history $H^s$, CR($H^s$) $\subseteq$ CBTR($H^s$). The converse is not, generally, true.

**Proof:** Follows from the definitions of *commute* and *commutes backward through*. □

**Theorem 8** $\exists$ $H^s$, CFR($H^s$) $\cap$ CBTR($H^s$) $\neq \emptyset$ *and* each is not included in the other; $\exists$ $H^s$, CFR($H^s$) $\cap$ CR($H^s$) $\neq \emptyset$ *and* each is not included in the other;

**Proof:** Consider the history

$H_{10}$ : $s_0$ $B_1$ $s_1$ $G_2$ $s_2$ $G_3$ $s_3$
$B_1$: if $y > 200$ then $x := x + 10$
$G_2$: if $y > 200$ then $x := x + 30$
$G_3$: $y := y + 100$

It is clear that CFR($H^s_{10}$)= $\{G_3\}$; CBTR($H^s_{10}$)=CR($H^s_{10}$)= $\{G_2\}$. This completes the proof. □

**Theorem 9** If the system has Property 1, then

1. $\forall$ $H^s$, CBTR($H^s$) $\subseteq$ FPR($H^s$)
2. $\exists$ $H^s$, CBTR($H^s$) $\subset$ FPR($H^s$)

**Proof:** Given a history $H^s$, showing that $T_i \in$ FPR($H^s$) holds for each transaction $T_i \in$ CBTR($H^s$) gives the proof. We prove this by induction on $k$ where $T_k$ is the $k$st transaction moved into CBTR($H^s$).

*Induction base:* ($k = 1$) We want to show that $T_1 \in \text{FPR}(H^s)$. If there are no transactions between $B_1$ and $T_1$ which are in $\text{FPR}(H^s)$, then $T_1$ will be moved into $\text{FPR}(H^s)$ according to Algorithm 2 because $T_1$ can precede every transaction $T_j^{\emptyset}$ between $B_1$ and $T_1$ owing to the fact that $T_1$ commutes backward through $T_j$. Otherwise, there must be some transaction $T_j$ with a non-empty fix $F_j$ staying between $B_1$ and $T_1$ (including $B_1$) in the rewritten history when $T_1$ is scanned in Algorithm 2. Here we assume that $F_j$ is captured by Lemma 2. At this point, assume $T_1$ cannot precede $T_j^{F_j}$, then $F_j \cap (T_1.readset - T_1.writeset) \neq \emptyset$ because otherwise $T_1$ can precede $T_j^{F_j}$ (The reason is: for every state $s_0$ on which $T_j^{F_j}T_1$ is defined, replacing $s_0$ with another state $s_1$ where the value of each item $x$ in $s_0 \cap F_j$ is replaced with $x$'s value in $F_j$. Then $T_j^{\emptyset}T_1$ is defined on $s_1$. According to Property 1, since $T_1$ can precede $T_j^{\emptyset}$ (Note that $T_1$ commutes backward through $T_j$), so $(T_j.readset - T_j.writeset) \cap T_1.writeset = \emptyset$. Since $F_j \subseteq (T_j.readset - T_j.writeset)$ according to Lemma 2, so $F_j \cap T_1.writeset = \emptyset$. So $T_1$ will not read or update any item in $F_j$. Therefore, $T_j^{F_j}T_1(s_0) = T_j^{\emptyset}T_1(s_1)$, and $T_1T_j^{F_j}(s_0) = T_1T_j^{\emptyset}(s_1)$. Since $T_1$ commutes backward through $T_j$, so $T_j^{\emptyset}T_1(s_1) = T_1T_j^{\emptyset}(s_1)$. Therefore, $T_j^{F_j}T_1(s_0) = T_1T_j^{F_j}(s_0)$, so $T_1$ can precede $T_j^{F_j}$). Therefore, $\exists x$, such that, $x \in F_j \cap (T_1.readset - T_1.writeset)$. Since $x \in F_j$, so according to Algorithm 2 there must be a transaction $T_p$, such that $T_p$ is now in $\text{FPR}(H^s)$, and $x \in T_p.writeset$. Otherwise, $x$ will not be put into $F_j$ by Lemma 2. Hence $T_p.writeset \cap (T_1.readset - T_1.writeset) \neq \emptyset$. This conflicts with Property 1 since $T_1$ commutes backward through $T_p$ thus $T_1$ can precede $T_p^{\emptyset}$. So the assumption that $T_1$ cannot precede $T_j^{F_j}$ does not hold. Therefore, $T_1$ can precede $T_j^{F_j}$. So $T_1$ can precede every transaction between $B_1$ and $T_1$ which has a non-empty fix. Since $T_1$ commutes backward through all the other transactions between $B_1$ and $T_1$, so $T_1$ will be moved into $\text{FPR}(H^s)$.

*Induction hypothesis:* for each $1 \leq k \leq n$, if $T_k \in \text{CBTR}(H^s)$, then $T_k \in \text{FPR}(H^s)$.

*Induction Step:* Let $k = n+1$, then when $T_k$ is scanned in both algorithms, every transaction $T_j$, which is between $B_1$ and $T_k$ in the rewritten history generated by Algorithm 2 at that time, is between $B_1$ and $T_k$ in the rewritten history generated by the commutes-backward-through rewriting algorithm. Therefore, $T_k$ commutes backward through every such $T_j$. For the same reason as in the *induction base* step, we know that $T_k$ will be moved into $\text{FPR}(H^s)$.

Therefore, statement 1 holds. Consider history $H_{10}$, it is clear that $\text{FPR}(H_{10}^s) = \{G_2, G_3\}$; $\text{CBTR}(H_{10}^s) = \{G_2\}$. So statement 2 holds. □

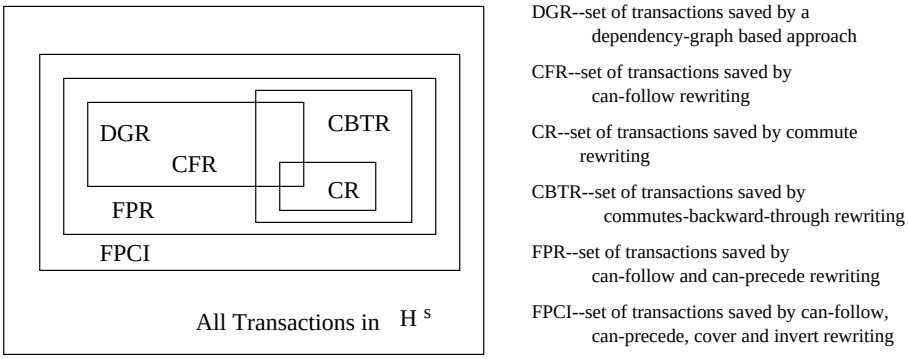


Figure 3: Relationships among Repair Approaches

In summary, after a history $H^s$ is repaired, the relationships among DGR($H^s$), CFR($H^s$), FPR($H^s$), FPCI($H^s$), CR($H^s$) and CBTR($H^s$) are shown in Figure 3. Here we assume that the system has Property 1.

# 7 Implementing the Repair Model on Top of Sagas

In this section, we will evaluate the feasibility of our repair model by integrating it with the Saga model [GMS87].

## 7.1 The Saga Model

The Saga Model is a practical transaction processing model addressing long duration transactions which can be implemented on top of an existing DBMS without modifying the DBMS internals at all. A saga consists of a collection of saga transactions (or *steps*), each of which maintains database consistency. However any partial execution of the saga is undesirable; either all the transactions in a saga complete successfully or compensating transactions should be run to amend for the partial execution of the saga. Thus corresponding to every transaction in the saga, except the last one, a compensating transaction is specified. The compensating transaction semantically undoes the effect of the corresponding transaction.

The Saga Model is suitable for our repair model to be implemented on top of it because it supports compensation inherently. For example, a com-

pensating transaction is specified for each transaction, except the last one, in a saga; and when a saga transaction $T_{ij}$ ends, the *end-transaction call* will include the identification of the compensating transaction of $T_{ij}$ which includes the name and entry point of the compensating program, plus any parameters that the compensating transaction may need.

By viewing each normal duration transaction and each long duration transaction which can not be specified as a multi-step saga, as a specific saga that consists of only one saga transaction, we can get an unified view of transactions in the systems where the saga model is implemented. By adding the compensating transaction for the last step in each saga, we can get all the necessary compensating transactions to do repair.

In addition, the saga model has the following two features which allow for optimization in rewriting a history.

**Consistency Property :** the execution of each saga transaction (step) maintains database consistency.

**Compensation Property :** during the lifetime of a saga **, no matter how the saga is interleaved with other sagas, any step in the saga which is successfully executed, if having not been compensated, can be compensated by executing the corresponding compensating transaction at the end of the growing history.

## 7.2 Repair a History of Sagas

The Compensation Property implies that in a history to be repaired whenever a saga is identified as a bad one, we can rewrite the history to move only the last step, instead of every step, of the saga to the end of the history. In this way, substantial rewriting and pruning work can be saved. The optimization based on can-follow rewriting (Algorithm 1) is specified in the the following algorithm.

**Algorithm 5** Rewrite a history of sagas by can-follow rewriting
**Input:** the serial history $H^s$ to be rewritten and the set **B** of bad sagas.
**Output:** a rewritten history $H_e^s$ with a prefix $H_r^s$ which consists of only good saga transactions.
**Method:** Scan forward from the first good saga transaction after $B_{11}$ until the end of $H^s$, for each step $T_{ij}$ (of saga $S_i$)
**case** $S_i \in$ **B** skip it;
**case** $S_i \in$ **G**

**The *lifetime* of a saga begins when the saga is initiated, and ends when the saga terminates (commits or aborts).

**if** there is a step of $S_i$ which stays between $B_1$ and $T_{ij}$
Skip $T_{ij}$;
**elseif** the final step $T_{pn}$ of every saga $S_p$ which stays between $B_1$ and $T_{ij}$ (including $B_1$) can follow $T_{ij}$
Move $T_{ij}$ to the position which immediately precedes $B_1$. As $T_{ij}$ is pushed through each such $T_{pn}$, augment $F_{pn}$ according to Lemma 2.

The integrated repair algorithm using Algorithm 5 to rewrite a history and the compensation approach to prune the rewritten history is specified as follows.

**Algorithm 6** Repair a history of sagas by can-follow rewriting
**Input:** the serial history $H^s$ to be repaired
**Output:** a repaired history $H^s_r$ which consists of only good saga transactions.
**Method:**
1. Rewrite $H^s$ using Algorithm 5 [††].
2. Do compensation from the end to the beginning of $H^s_e$ until $B_1$ is compensated. When the final step $T_{pn}$ of a saga $S_p$ is to be compensated

- First, execute $T_{pn}^{(-1,F_{pn})}$ to compensate $T_{pn}^{F_{pn}}$. The codes and input parameters of $T_{pn}^{(-1,F_{pn})}$ are got from the identification of the compensating transaction of $T_{pn}$ included in the *end-transaction call* of $T_{pn}$. Note that $F_{pn}$ has already been computed after $H^s$ was rewritten.
- Second, execute the sequence $T_{p(n-1)}^{(-1,\emptyset)}$, ..., $T_{pq}^{(-1,\emptyset)}$ to compensate all the other steps of $S_p$ which are not in $H^s_r$. The codes and input parameters of these compensating transactions are got in the same way as of $T_{pn}^{(-1,F_{pn})}$.

The correctness of Algorithm 6 is specified as follows.

**Theorem 10** Algorithm 6 is correct in the sense that $H^s_r$ is consistent after step 1, and the repair results in the same state as generated by re-executing $H^s_r$.

**Proof:** It is clear that in Algorithm 5 a step $T_{ij}$ will not be moved into $H^s_r$ unless every step between $T_{i1}$ and $T_{ij}$ can be moved into $H^s_r$. For a step $T_{ij}$, if $S_i$ is a good saga, and each step between $T_{i1}$ and $T_{ij}$ (including $T_{ij}$) can be moved into $H^s_r$, then we say $T_{ij}$ is an *unaffected step*, otherwise, we say $T_{ij}$ is an *affected step*.

We propose another approach to repair $H^s$ which is clearly correct. It works as follows: Scan $H^s$ backward from the end to the beginning:

[††]It is possible that in $H^s_e$ one part of a saga $S_i$ is in $H^s_r$ and the other part of $S_i$ is in $H^s_e - H^s_r$.

- If a bad final step $B_{in}$ is met, execute the sequence $B_{in}^{(-1,F_{in})}$, $B_{i(n-1)}^{(-1,\emptyset)}$, ..., $B_{i1}^{(-1,\emptyset)}$ on the final state of the current history. This can remove the effects of $B_i$ from the current history because at this point all the steps to the right of $B_{in}$ are unaffected steps. All the bad or affected steps to the right of $B_{in}$ have already been compensated. So $B_{in}$ can follow every step following it, thus $B_{in}$ can be moved to the end of the current history without changing the final state of the current history if $F_{in}$ is computed according to Lemma 2, therefore, according to the Compensation Property, after $B_{in}^{F_{in}}$ is compensated executing $B_{i(n-1)}^{(-1,\emptyset)}$, ..., $B_{i1}^{(-1,\emptyset)}$ can compensate the other steps.
- If an affected final step $T_{in}$ is met, assume $T_{ip}$ is the last unaffected step in $S_i$, execute the sequence $T_{in}^{(-1,F_{in})}$, $T_{i(n-1)}^{(-1,\emptyset)}$, ..., $T_{i(p+1)}^{(-1,\emptyset)}$. For similar reasons to the above case, this can remove the effects of all the affected steps of $S_i$.

It is clear that the above approach results in $H_r^s$. So $H_r^s$ is consistent. Since the above approach executes the same set of fixed compensating transactions on the final state of $H^s$ as Algorithm 6, and it executes these fixed compensating transactions in the same order, so Algorithm 6 results in the same state as generated by re-executing $H_r^s$. □

Algorithm 2 and Algorithm 3 can also be adapted to rewrite a history of sagas. The adapted algorithms are specified as follows. For brevity, and to highlight the differences between these algorithms, we describe only the modifications to Algorithm 2 and to Algorithm 3, respectively.

**Algorithm 7** Rewrite a history of sagas by can-follow and can-precede rewriting
**Method:** Scan $H^s$ forward from the first good step after $B_{11}$ until the end of $H^s$, for each step $T_{ij}$
**case** $S_i \in \mathbf{G}$
**if** there is a step of $S_i$ which stays between $B_1$ and $T_{ij}$
Skip $T_{ij}$;
**elseif** the final step $T_{pn}$ of every saga $S_p$ which stays between $B_1$ and $T_{ij}$ (including $B_1$) can follow $T_{ij}$, or $T_{ij}$ can precede $T_{pn}^{F_{pn}}$
Move $T_{ij}$ to the position which immediately precedes $B_1$.

**Algorithm 8** Rewrite a history of sagas by can-follow, can-precede, cover and invert rewriting
**Method:** Scan $H^s$ forward from the first good step after $B_{11}$ until the end of $H^s$, for each step $T_{ij}$
**case** $S_i \in \mathbf{G}$
**if** there is a step of $S_i$ which stays between $B_1$ and $T_{ij}$

Skip $T_{ij}$;
**elseif** the final step $T_{pn}$ of every saga $S_p$ which stays between $B_1$ and $T_{ij}$ (including $B_1$) can follow $T_{ij}$, or $T_{ij}$ can precede $T_{pn}^{F_{pn}}$, or $T_{ij}$ covers $T_{pn}^{F_{pn}}$, or $T_{ij}$ inverts $T_{pn}^{F_{pn}}$
Move $T_{ij}$ to the position which immediately precedes $B_1$.

The correctness of the repair based on Algorithm 7 or Algorithm 8 is specified in the following theorem. The proof is similar to that of Theorem 10, thus omitted.

**Theorem 11** The repair based on Algorithm 7 is correct in the sense that Theorem 10 still holds even if the rewriting step (step 1) of Algorithm 6 is done by Algorithm 7. The repair based on Algorithm 8 is correct in the sense that Theorem 10 still holds after Algorithm 6 is modified as follows:

- The rewriting step (step 1) is done by Algorithm 8;
- In step 2, when an affected step $T_{i(n-1)}$ of a saga $S_i$ is scanned [‡‡], assume $T_{ip}$ is the last unaffected step in $S_i$, execute the sequence $T_{i(n-1)}^{(-1,\emptyset)}$, ..., $T_{i(p+1)}^{(-1,\emptyset)}$.

## 7.3 Detecting Can-Follow, Can-Precede, Cover and Invert Relationships between Transactions

In Section 7.2, the repair based on Algorithm 5, Algorithm 7 and Algorithm 8 cannot be enforced without first capturing the *can-follow*, *can-precede*, *cover*, and *invert* relationships between saga transactions.

Given a history of sagas, the can-follow relationships between the saga transactions in the history depend on the readset-writeset relationships between these transactions. The write set of a transaction $T_i$ can be got from the traditional log where every write operation is recorded. However, the read information of $T_i$ we can get from the logs for traditional recovery purposes such as physical logs, physiological logs, and logical logs [GR93], is usually not enough to generate the read set. Therefore, the efficient maintenance of read information is a critical issue. In particular, there is a tradeoff between the extra cost we need to pay besides that of traditional recovery facilities and the guaranteed availability of read information. The read information can be captured in several ways, for example

[‡‡]This may happen because $T_{in}$ may have already been covered or inverted.

- Augment the write log to incorporate read information. There are basically two ways: one is appending the read record $[T_i, x]$ to the log every time when $T_i$ reads an item $x$. The other way is first keeping the set of items read by $T_i$ in another place until the time when $T_i$ is going to commit. At this point, the read set of $T_i$ can be forced to the log as one record.

  Although keeping read information in the log will not cause more forced I/O, it does consume more storage. Another problem with the approach lies in the fact that almost all present database systems keep only update(write) information in the log. Thus adding read records to the log may cause the redesign of the current recovery mechanisms.

- Extract read sets from the profiles and input arguments of transactions. Compared with the read log approach, when transaction profiles (or codes) are available, each transaction just needs to store its input parameters, which are often much smaller in size than the read set. More important, instead of putting the input parameters in the log, each transaction can store the parameters in a specific user database, thus the repair module can be completely isolated from the traditional recovery module. In this way, our repair model can be implemented on top of the Saga model without modifying the internals of the DBMS on which the Saga model is implemented.

  This approach captures read information without the need to modify DBMS internals. However, it usually can only achieve a complete repair, but not an exact repair. That is, the effects of all bad transactions will be removed, but the effects of some unaffected good transactions may sometimes be removed also since in many situations the approach can only get an approximate read set. Interested readers can refer to [AJL98] for more details of this approach.

- Although traditional logging only keeps write information, more and more read information can be extracted from the log, particularly when more operation semantics are kept in the logs. Traditional physical(value) logging keeps the before and after images of physical database objects(i.e., pages), so we only know that some page is read. In addition, a page is normally too large a unit to achieve a fine repair. Physiological logging keeps only the update to a record(tuple) within one and only one page, so we know that this record should be in the read set, which is much finer than physical logging. Logical logging keeps more operation semantics than the other two logging approaches. Conceptually logical logs can keep track of all the read information of a transaction, though this is not supported by current database systems.

However, logical logging attracts substantial industrial and research interests. In system R, SQL statements are put into the log as logical records; In [LT98], logical logs can be a function, like `x=sum(x,y)`, and `swap(x,y)` etc.. In both situations, we get more read information than other logging methods.

In long duration transaction models([GMS87], [WR91]), or in multilevel transaction models ([WHBM90], [Lom92]), it is possible to extract the read information of transaction (subtransaction) $T$ from its compensation log records, where the action of $T$'s compensating transaction is recorded.

The *can-precede*, *cover*, and *invert* relationships between transactions are based on the semantics of transactions, and they can be captured in a similar way to commutativity[LMWF94, Wei88, Kor83, SKPO88], and recoverability[BK92]. In order to capture these relationships, the profile (or code) and input arguments of each transaction must be available. In the Saga model, several possible solutions to the problem of saving code reliably are proposed[GMS87], therefore, these relationships can be reliably captured in the Saga model.

For a *canned* system with limited number of transaction classes and fixed code for each transaction class, the *can-follow*, *can-precede*, *cover*, and *invert* relationships between saga transactions can be detected according to the corresponding relationships between transaction classes. Although detecting these relationships between two transaction classes usually needs more effort than detecting these relationships between two transactions, after this is done with all the transaction classes, detecting these relationships between transactions of these classes can be much easier in many situations.

For example, in a bank a deposit transaction (denoted $dep(a_i, m)$) which deposits $m$ amount of money into account $a_i$ can follow a withdraw transaction (denoted $wit(a_j, n)$) which withdraws $n$ amount of money from account $a_j$ only if they access different accounts, that is, $a_i \neq a_j$. Therefore, given the can-follow relationship between the deposit transaction class and the withdraw transaction class, the can-follow relationship between a deposit transaction and a withdraw transaction can be detected without the need to check the readset-writeset relationship between the two transactions, checking their input parameters is enough.

### 7.4 Fix Information Maintenance

It is clear that Lemma 3 can be used in Algorithm 5, Algorithm 7 and Algorithm 8, to capture fixes. For a transaction $T_i$, there are two methods to get $T_i.readset - T_i.writeset$: one is to first get the readset and writeset of $T_i$ after an execution history is generated using the approaches proposed in Section 7.3, then compute $T_i.readset - T_i.writeset$; the other is what we have proposed in Section 3, that is, let each transaction $T_i$ write the set $T_i.readset - T_i.writeset$ as a record to the database when it is executed, then when we rewrite $H^s$ all the fixes can be directly got from the database.

It should be noticed that in the situations where the read and write sets of $T_i$ have to be firstly captured in order to detect the can-follow relationships between $T_i$ and some other transactions, the first method is more efficient; In contrast, when all the necessary can-follow relationships between $T_i$ and other transactions can be detected without the need to check the readset-writeset relationships between $T_i$ and these transactions, for example, when these relationships can be directly got from the can-follow relationships between the corresponding transaction classes, the second method is more efficient.

## 8 Related Work

Database recovery mechanisms are not designed to deal with the recovery from undesirable but committed transactions. Traditional recovery mechanisms [BHG87] based on physical or logical logs guarantee the ACID properties of transactions – Atomicity, Consistency, Isolation, and Durability – in the face of process, transaction, system and media failures. In particular, the last of these properties ensure that traditional recovery mechanisms never undo committed transactions. However, the fact that a transaction commits does not guarantee that its effects are desirable. Specifically, a committed transaction may reflect inappropriate and/or malicious activity.

There are two common approaches to handling the problem of undoing committed transactions: rollback and compensation. The rollback approach is simply to roll back all activity – desirable as well as undesirable – to a point believed to be free of damage. Such an approach may be used to recover from inadvertent as well as malicious damage. For example, users typically restore files with backup copies in the event of either a disk crash or a virus attack. In the database context, checkpoints serve a similar function of providing stable, consistent snapshots of the database. The rollback approach is effective, but expensive, in that all of the desirable work between the time of the backup and the time of recovery is lost. Keeping this window of vulnerability acceptably

low incurs a substantial cost in maintaining frequent backups or checkpoints, although there are algorithms for efficiently establishing snapshots on-the-fly [AJM95, MPL92, Pu86].

The compensation approach [GM83, GMS87] seeks to undo either committed transactions or committed steps in long-duration or nested transactions [KLS90]. There are two kinds of compensation: action-oriented and effect-oriented [KLS90, Lom92, WHBM90, WS92]. Action-oriented compensation for a transaction or step $T_i$ compensates only the actions of $T_i$. Effect-oriented compensation for a transaction or step $T_i$ compensates not only the actions of $T_i$, but also the actions that are affected by $T_i$. For example, consider a database system that deals with transactions that represent purchasing of goods. The effects of a purchasing transaction $T_1$ might have triggered a dependent transaction $T_2$ that issued an order to the supplier in an attempt to replenish the inventory of the sold goods. In this situation, the action-oriented compensating transaction for $T_1$ will just cancel the purchasing; but the effect-oriented compensating transaction for $T_1$ will cancel the order from the supplier as well. Although a variety of types of compensation are possible, all of them require semantic knowledge of the application.

The notion of commutativity, either of operations [LMWF94, Wei88, Kor83] or of transactions [SKPO88], has been well exploited to enhance concurrency in semantics-driven concurrency control. There are several types of commutativity. In operation level, for example, two operations $O_1$ and $O_2$ *commute forward* [Wei88] if for any state $s$ in which $O_1$ and $O_2$ are both defined, $O_2(O_1(s)) = O_1(O_2(s))$; $O_2$ *commutes backward through* [LMWF94] $O_1$ if for any state $s$ in which $O_1O_2$ is defined, $O_2(O_1(s)) = O_1(O_2(s))$; $O_1$ and $O_2$ *commute backward* [LMWF94, Wei88] if each commutes backward through the other. In transaction level, for example, two transactions *commute* [SKPO88] if any interleaving of the actions of the two transactions for which both transaction commit yields the same final state; Two transactions *failure commute*[SKPO88] if they commute, and if they can both succeed then a unilateral abort by either transaction cannot cause the other to abort. Our notation *can precede* is adapted from the *commutes backward through* notation for the purpose of taking advantage of transaction level commutativity.

In [BK92], semantics of operations on abstract data types are used to define *recoverability*, which is a weaker notion than commutativity. *recoverability* is a more general notion than *can follow* in capturing the semantics between two operations or transactions, but *can follow* is more suitable for rewriting histories. *recoverability* is applied to operations on abstract data types but *can follow* is applied to transactions. *recoverability* is defined based on the return value of operations, and thus a purely semantic notion; but *can*

*follow* is defined based on the intersections of read and write sets between two transactions.

Korth, Levy, and Silberschatz [KLS90] address the recovery from undesirable but committed transaction. The authors build a formal specification model for compensating transactions which they show can be effectively used for recovery. In their model, a variety of types of correct compensation can be defined. A compensating transaction, whose type ranging from traditional undo, at one extreme, to application-dependent, special-purpose compensating transactions, at the other extreme, is specified by some constraints which every compensating transaction must adhere. Different types of compensation are identified by the notion of compensation soundness. A history $X$ consisting of $T$, the compensating-for transaction; $CT$, the compensating transaction; and $dep(T)$, a set of transactions dependent upon $T$, is *sound* if it is equivalent to some history of only the transactions in $dep(T)$.

Though a compensating transaction in our model can be specified by their model, our notion of a *repaired* history is more suitable for rewriting histories than the notion of *sound* history, since the constraint that compensating transactions can only be applied to the final state of a history greatly decreases the possibility of finding a sound history, even if commutativity is fully exploited. We can get a feasible history by rewriting the original history based on *can follow*, *can precede*, *invert* and *cover*. The resulting history augmented with the corresponding undo-repair actions or fixed compensating transactions yields the desired repair.

# 9 Discussion and Conclusion

## 9.1 Discussion

### 9.1.1 Relevant Security Contexts

Our repair model can be applied to many kinds of secure database systems to enhance their survivability. However, the main factors on which the applicability of our model to a secure database system is dependent, such as (1) the characteristics of the database, i.e., whether it is single-version or multiversion, (2) the concurrency control protocol and the characteristics of the histories produced by it, and (3) the recovery protocol and the characteristics of the logs produced by it, are closely relevant to the security model and architecture of the system.

For a single-level secure database system where every subject (trans-

action) and object (data item) are within the same security class, traditional concurrency control protocols such as two-phase locking (2PL), and recovery protocols such as write-ahead logging (WAL), can be directly used without causing any security policy violations, no matter which kind of security model (i.e., access-matrix model[Lam74], role-based access control model[SCFY96], type-based access control model[San92], or flexible access-control model[JSS97]) is enforced. Since serializable histories are generated by most of the current single-level systems, so our repair model can be directly applied to single-level systems in most cases. However, there are some systems where each data item has multiple versions, and one-copy serializable histories are generated instead. Since an one-copy serializable history is view equivalent to a serial single-version history[BHG87], our model can be used to repair the one-copy serializable history by rewriting the equivalent serial history. However, it should be noticed that pruning a rewritten history in multiversion databases is usually more complicated because during pruning we need to decide for a (dirty) data item which version should be read, which version should be updated, and which version should be discarded (i.e., the versions created by bad transactions can just be discarded). Detailed pruning algorithms are out of the scope of the paper.

For a multilevel secure (MLS) database system, traditional concurrency control and recovery protocols, however, are usually not enough to satisfy security requirements[AJB97], especially, they can cause signaling channels from high level processes to low level processes. Therefore, secure transaction processing is required. Most of the recent research and development in secure concurrency control can be categorized into two different areas: one based on *kernelized* architecture and the other based on *replicated* architecture. These two are among the number of architectures proposed by the Woods Hole study group[oMDMSBC83] to build multilevel secure DBMSs with existing DBMS technology instead of building a trusted DBMS from scratch.

For kernelized architecture, several kinds of secure concurrency control protocols are proposed: (1) In [MJ93, JMR97], several secure lock-based protocols are proposed. Although they do not always produce serializable schedules, our repair model can be directly applied to every serializable history generated by them. Extending our model to repair those non-serializable schedules is out of the scope of the paper. (2) In [AJ92], two secure timestamp-based protocols are proposed. Although they produce only serializable histories to which our model can be directly applied, they are prone to starvation. In [JA92], a single-level timestamp-based scheduler is proposed which is secure and free of starvation. Although it produces one-copy serializable histories, our model can still be directly used to rewrite these histories (the reason is mentioned above). (3) In [AJB96, JA92, AJB97], three weaker notions of

correctness, namely, *levelwise* serializability, *one-item read* serializability, and *pairwise* serializability, are proposed to be used as alternative for one-copy serializability such that the nature of integrity constraints in MLS databases can be exploited to improve the amount of concurrency. Extending our model to repair levelwise, one-item read, and/or pairwise serializable histories is out of the scope of the paper.

For replicated architecture, several secure concurrency control protocols are proposed in [JK90, MJS91, Cos92, CM92]. Since they all produce one-copy serializable histories, so our model can be directly applied to rewrite these histories.

In [KT90], a scheduler is proposed which is secure and produces one-copy serializable histories to which our model can be applied. However, it uses a multilevel scheduler which, therefore, has to be trusted, thus it is only suitable for the *trusted subject* architecture.

Since in our repair model serial orders among transactions are captured from the log, so the applicability of our model is affected by logging protocols. In [PKP97], a multilevel secure log manager is proposed to eliminate such covert channels as *insert* channels and *flush* channels which are caused by traditional logging protocols. Although *Logical Log Sequence Numbers* (LLSN) instead of physical *Log Sequence Numbers* (LSN) are provided in [PKP97] to eliminate insert channels, we can still extract serial orders from the log because records of transactions within different security classes are still kept in the same log, and LLSNs can be translated to physical LSNs internally by the log manager. Moreover, since the mechanisms proposed to eliminate flush channels will not change the structure of the log, so our model can be directly applied to a system with such a log manager.

### 9.1.2 Other Issues

One criticism of the applicability of the method may be that if a bad transaction $B_i$ is detected too late, that is, if the *latency time* of $B_i$ is too long, then there can be too many affected good transactions to deal with, especially when they have caused further effects to the real world. For example, some real world decisions could be based on these affected transactions. At this situation, 'manual' recovery actions may be necessary.

We counter this augment by noting that the latency time of $B_i$ is usually related to the amount of transactions affected by $B_i$. The more transactions affected by $B_i$, the more proofs of $B_i$'s malicious actions can be collected by the intrusion detector, hence the shorter the latency time of $B_i$. Therefore,

even if the *latency time* of $B_i$ is very long, the amount of transactions affected by $B_i$ may not be too large in many circumstances. At this situation, the algorithm may need more time since it needs to scan a long history, but the pruning may still be a short process if most of the transactions in the history are unaffected. Although the compensation approach may not be practical when the history is very long and the codes for compensating transactions have to be kept in the log, it can be used in almost all canned systems, which are very general in real world where the codes for transactions and compensating transactions are fixed for each transaction class. As the techniques of intrusion detection are advanced, the latency time of a bad transaction should become shorter, so our repair model will apply to more situations.

As to the criticism that manual recovery actions can be necessary, note that when damage has been caused, the effects of these affected transactions to the real world are already there. No matter whether the history is repaired or not, some action to compensate these undesirable effects is required. In the real world, such manual recovery actions are basically unavoidable. Therefore, repairing the database such that a consistent database state where no effects of bad transactions are there could be generated can be viewed as a separate issue from manual recovery. In addition, our rewriting methods can help users to assess the degree of damages because $\mathbf{B} \cup \mathbf{AG}$ can be identified. Therefore, the security administrator can know on which transactions (or on which customers) such manual recovery actions should be enforced.

## 9.2 Conclusion

In an IW scenario it is necessary to undo committed malicious transactions for the purpose of damage repair. Traditional recovery methods have the disadvantage of wiping out much good work along with the bad, and compensation methods are heavily dependent on application semantics.

In this paper we developed the notion of rewriting execution histories for the purpose of removing the effects of a set of *bad* transactions and the *affected good* transactions that depend on the bad transactions. The fact that the transactions being moved during the rewriting are subsequently unwound greatly increases the flexibility of the rewriting methods. We developed a basic rewriting method that can unwind exactly the set $\mathbf{B} \cup \mathbf{AG}$, and then developed additional rewrites incorporating commutativity, inverses, and covers to save further transactions in $\mathbf{AG}$. It is shown that our approach is strictly better at saving good transactions than a dependency-graph based approach. And in most situations, our approach is strictly better at saving good transactions than an approach which is based on commutativity only. It is also shown

that besides recovery from malicious transactions, our approach can also be extended to may other applications such as malicious user isolation, system upgrades, optimistic replication protocols, and replicated mobile databases.

## References


[AJ92] P. Ammann and S. Jajodia. A timestamp ordering algorithm for secure, single-version, multi-level databases. In C. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, pages 23–25. Amsterdam: North Holland, 1992.

[AJB96] V. Atluri, S. Jajodia, and E. Bertino. Alternative Correctness Criteria for Concurrent Execution of Transactions in Multilevel Secure Databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):839–854, October 1996.

[AJB97] V. Atluri, S. Jajodia, and E. Bertino. Transaction Processing in Multilevel Secure Databases with Kernelized Architecture: Challenges and Solutions. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):697–708, 1997.

[AJL98] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. Technical report, George Mason University, 1998. http://www.isse.gmu.edu/~pliu/papers/dynamic.ps.

[AJM95] P. Ammann, S. Jajodia, and P. Mavuluri. On the fly reading of entire databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):834–838, October 1995.

[AJMB97] P. Ammann, S. Jajodia, C.D. McCollum, and B.T. Blaustein. Surviving information warfare attacks on databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–174, Oakland, CA, May 1997.

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

[BK92] B.R. Badrinath and Ramamritham Krithi. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[CM92] O. Costich and J. McDermott. A multilevel transaction problem for multilevel secure database systems and its solution for the replicated architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 192–203, Oakland, CA, 1992.

[Cos92] O. Costich. Transaction processing using an untrusted scheduler in a multilevel secure database with replicated architecture. In C. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, pages 173–189. Amsterdam: North Holland, 1992.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–581, September 1984.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, 1996.

[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 International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[JA92] S. Jajodia and V. Atluri. Alternative correctness criteria for concurrent execution of transactions in multilevel secure databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 216–224, Oakland, CA, 1992.

[JK90] S. Jajodia and B. Kogan. Transaction processing in multilevel secure databases using replicated architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 360–368, Oakland, CA, 1990.

[JLM98] S. Jajodia, P. Liu, and C.D. McCollum. Application-level isolation to cope with malicious database users. In *Proceedings of the 14th Annual Computer Security Application Conference*, Phoenix, AZ, December 1998. To appear.

[JMR97] S. Jajodia, L. Mancini, and I. Ray. Secure locking protocols for multilevel database management systems. In P. Samarati and R. Sandhu, editors, *Database Security X: Status and Prospects*. London: Chapman & Hall, 1997.

[JSS97] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, 1997.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

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

[KT90] T. F. Keefe and W. T. Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 369–383, Oakland, CA, 1990.

[Lam74] B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[Lom92] D.B. Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 185–194, San Diego, CA, June 1992.

[LT98] D. Lomet and M. R. Tuttle. Redo recovery after system crashes. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 6. Prentice Hall PTR, 1998.

[MJ93] J. McDermott and S. Jajodia. Orange locking: Channel-free database concurrency control. In B. M. Thuraisingham and C. E. Landwehr, editors, *Database Security, VI: Status and Prospects*, pages 267–284. Amsterdam: North Holland, 1993.

[MJS91] J. McDermott, S. Jajodia, and R. Sandhu. A single-level scheduler for replicated architecture for multilevel secure

databases. In *Proceedings of the 7th Annual Computer Security Applications Conference*, pages 2–11, San Antonio, TX, 1991.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 124–133, San Diego, CA, June 1992.

[oMDMSBC83] Committee on Multilevel Data Management Security, Air Force Studies Board, and National Research Council. *Multilevel Data Management Security*. National Academy Press, Washington, D.C., March 1983.

[PKP97] V. R. Pesati, T. F. Keefe, and S. Pal. The design and implementation of a multilevel secure log manager. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 55–64, Oakland, CA, 1997.

[Pu86] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, October 1986.

[San92] R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, Los Alamitos, CA, 1992.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, (2):38–47, February 1996.

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

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

[WHBM90] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multilevel recovery. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium of Principles of Database Systems*, pages 109–123, Nashville, Tenn, April 1990.

[WR91] H. Wachter and A. Reuter. The contract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann Publishers, 1991.

[WS92] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, Inc., 1992.