# Recovery From Malicious Transactions *

Paul Ammann    Sushil Jajodia
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
{pammann, jajodia}@gmu.edu

Peng Liu
Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD 21250
pliu@umbc.edu

## Abstract

Preventive measures sometimes fail to deflect malicious attacks. In this paper, we adopt an information warfare perspective, which assumes success by the attacker in achieving partial, but not complete, damage. In particular, we work in the database context and consider recovery from malicious but committed transactions. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of benign transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. Recovery is complicated by the presence of benign transactions that depend, directly or indirectly, on the malicious transactions. We present algorithms to restore only the damaged part of the database. We identify the information that needs to be maintained for such algorithms. The initial algorithms repair damage to quiescent databases; subsequent algorithms increase availability by allowing new transactions to execute concurrently with the repair process. Also, via a study of benchmarks, we show practical examples of how offline analysis can efficiently provide the necessary data to repair the damage of malicious transactions.

Index terms: Security, Database Recovery, Transaction Processing, Assurance.

## 1 Introduction

Database security concerns the confidentiality, integrity, and availability of data stored in a database. A broad span of research from authorization [13, 30, 15], to inference control [1], to multilevel secure databases [36, 31], and to multilevel secure transaction processing [4], addresses primarily how to protect the security of a database, especially its confidentiality. However, very limited research has been done on how to survive successful database attacks, which can seriously impair the integrity and availability of a database. Experience with data-intensive applications such as credit card billing, banking, air traffic control, logistics management, inventory tracking, and online stock trading, has shown that a variety of attacks do succeed to fool traditional database protection mechanisms. In fact, we must recognize that not all attacks – even obvious ones – can be averted at their outset. Attacks that succeed, to some degree at least, are unavoidable. With cyber attacks on data-intensive internet applications, i.e., e-commerce systems, becoming an ever more serious threat to our economy, society, and everyday lives, attack resilient database systems that can survive malicious attacks are a significant concern.
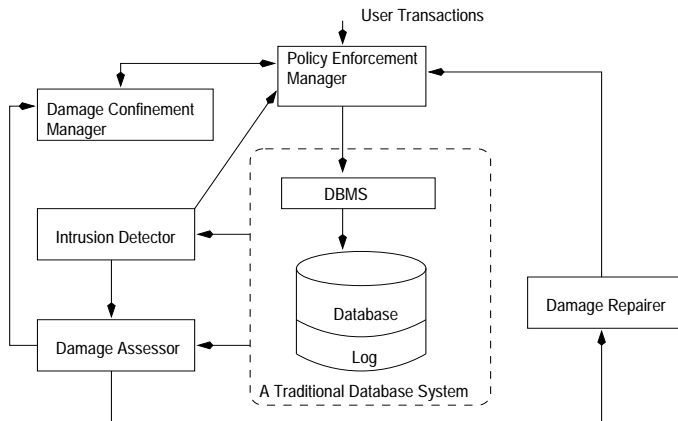
Figure 1: An Intrusion Tolerant Database System Architecture

One critical step towards attack resilient database systems is intrusion detection, which has attracted many researchers [7, 20, 27]. Intrusion detection systems monitor system or network activity to discover attempts to disrupt or gain illicit access to systems. The methodology of intrusion detection can be roughly classed as being either based on *statistical profiles* [16] or on known patterns of attacks, called *signatures* [14, 32]. Intrusion detection can supplement protection of database systems by rejecting the future access of detected attackers and by providing useful hints on how to strengthen the defense. However, intrusion detection makes the system attack-aware but not attack-resilient, that is, intrusion detection itself cannot maintain the integrity and availability of the database in face of attacks.

To overcome the inherent limitation of intrusion detection, a broader perspective is introduced, saying that in addition to detecting attacks, countermeasures to these successful attacks should be planned and deployed in advance. In the literature, this is referred to as *survivability* or *intrusion tolerance*. In this paper, we will study a critical database intrusion tolerance problem beyond intrusion detection, namely *attack recovery*, and present a set of innovative algorithms to solve the problem.

## The Problem

The attack recovery problem can be better explained in the context of an intrusion tolerant database system. Database intrusion tolerance can typically be enforced at two levels: *operating system (OS) level* and *transaction level*. Although transaction level methods cannot handle OS level attacks, it is shown that in many applications where attacks are enforced mainly through malicious transactions transaction level methods can tolerate intrusions in a much more effective and efficient way. Moreover, it is shown that OS level intrusion tolerance techniques such as those proposed in [20, 21, 23, 24, 5], can be directly integrated into a transaction level intrusion tolerance framework to complement it with the ability to tolerate OS level attacks. This paper will focus on transaction level intrusion tolerance, and our presentation will be based on the intrusion tolerant database system architecture shown in Figure 1.

The architecture is built on top of a traditional "off-the-shelf" DBMS. Within the framework, the *Intrusion Detector* identifies malicious transactions based on the history kept (mainly) in the log. The *Damage Assessor* locates the damage caused by the detected transactions. The *Damage Re-*

2

*pairer* repairs the located damage using some specific cleaning operations. The *Damage Confinement Manager* restricts the access to the data items that have been identified by the Damage Assessor as damaged, and unconfines a data item after it is cleaned. The *Policy Enforcement Manager* (PEM) (a) functions as a proxy for normal user transactions and those cleaning operations, and (b) is responsible for enforcing system-wide intrusion tolerant policies. For example, a policy may require the PEM to reject every new transaction submitted by an user as soon as the Intrusion Detector finds that a malicious transaction is submitted by the user.

We need this architecture because current database systems are relatively easy to attack (especially for malicious insiders) and very limited in surviving attacks, although access controls, integrity constraints, concurrency control, replication, active databases, and recovery mechanisms deal well with many kinds of mistakes and errors. 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 committed 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.

The attack recovery problem has two aspects: *damage assessment* and *damage repair*. The complexity of attack recovery is mainly caused by a phenomenon denoted *damage spreading*. In a database, the results of one transaction can affect the execution of some other transactions. Informally, when a transaction $T_i$ reads a data item $x$ updated by another transaction $T_j$ (We say $T_i$ reads $x$ from $T_j$), $T_i$ is directly *affected* by $T_j$. If a third transaction $T_k$ is affected by $T_i$, but not directly affected by $T_j$, $T_k$ is indirectly affected by $T_j$. It is easy to see that when a (relatively old) transaction $B_i$ that updates $x$ is identified malicious, the damage on $x$ can spread to every data item updated by a transaction that is affected by $B_i$, directly or indirectly. The goal of attack recovery is to locate each affected transaction and recover the database from the damage caused on the data items updated by every malicious or affected transaction.

In some cases, the attacker's goal may be to reduce availability by attacking integrity. In these cases the attacker's goal not only introduces damage to certain data items and uncertainty about which good transactions can be trusted, but also achieves the goal of bringing the system down while repair efforts are being made. 'Coldstart' semantics for recovery mean that system activity is brought to a halt while damage is being repaired. To address the availability threat, recovery mechanisms with 'warmstart' or 'hotstart' semantics are needed. Warmstart semantics for recovery allow continuous, but degraded, use of the database while information warfare damage is being repaired. Hotstart semantics make recovery transparent to the users. It is clear that the job of attack recovery gets even more difficult as use of the database continues because the damage can spread to new transactions and cleaned objects can be re-damaged by new transactions.

**Our Contribution**

Our contribution is to provide recovery algorithms that, given a specification of malicious, committed transactions (from the Intrusion Detector), unwinds the effects of each malicious transaction, along with the effects of any benign transaction that is affected, directly or indirectly by a malicious transaction. Significantly, the work of the remaining benign transactions is saved. Our recovery algorithms can be broken down into two categories: one category yields coldstart semantics; the database is unavailable during repair. The other category yields warmstart semantics; normal use

may continue during repair, although some degradation of service may be experienced by some transactions.

We outline various possibilities for maintaining read-from dependency information. Although direct logging of transaction reads has the virtue of simplicity, the performance degradation of such an approach may be too severe in some cases. For this reason, we show that offline analysis can efficiently meet the need for establishing read-from dependency information. We illustrate the practicality of such an approach via a study on standard benchmarks.

The remainder of the paper is organized as follows. Section 2 discusses related work. In Section 3, we present our transaction model for attack recovery. Section 4 presents our repair model. Section 5 develops algorithms with coldstart recovery semantics. Section 6 develops algorithms with warmstart recovery semantics. Section 7 uses benchmark applications to show how offline analysis can mitigate performance degradation during normal operation. Section 8 addresses some implementation issues. Section 9 concludes the paper.

## 2   Related Work

Database recovery mechanisms are not designed to deal with malicious attacks. Traditional recovery mechanisms [6] 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.

Although our repair model is related to the notion of *cascading abort* [6], cascading aborts only capture the *read-from* relation between active transactions. However, it may be necessary to capture the read-from relation between two committed transactions, even if the second transaction began long after the first one committed. In addition, in standard recovery approaches cascading aborts are avoided by requiring transactions to read only committed data [17].

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 [2, 25, 28].

The compensation approach [8, 9] seeks to undo either committed transactions or committed steps in long-duration or nested transactions [17] without necessarily restoring the data state to appear as if the malicious transactions or steps had never executed. There are two kinds of compensation: action-oriented and effect-oriented [17, 19, 34, 35]. 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 $T_i$. Although a variety of types of compensation are possible, all of them require semantic knowledge of the application. In this paper, we do not rely on semantic information, but rather use the syntactic information of read-write dependencies. Although the semantic approach can be be very powerful, our goal here

4

is to develop methods that integrate well with mainstream commercial systems, which currently do not support semantic models.

Survivability has received attention in the database context. Graubert, Schlipper, and McCollum identified database management aspects that determine the vulnerability to information warfare attacks [10]. McDermott and Goldschlag [23, 24] developed storage jamming, which can be used to seed a database with dummy values, access to which indicates the presence of an intruder. Ammann et al. [3] used a color scheme for marking damage and repair in databases and a notion of integrity suitable for databases that are partially damaged to develop a mechanism by which databases under attack could still be safely used. The present paper differs in that it focuses on repair, as opposed to management, detection, or availability, as cited above.

## 3 Transaction Models for Attack Recovery

At first glance, attack recovery, which aims to efficiently remove the effects of malicious or affected, committed transactions by exploiting traditional database recovery facilities as much as possible, seems to violate durability which implies that committed transactions should not be undone. This suggests that we need to bridge the theoretical gap between classical database recovery theory and attack recovery practice before addressing concrete recovery algorithms.

A straightforward approach to bridge the gap is using a *flat-transaction* recovery model where transactions are *flat* without containing any subtransactions, and committed transactions are 'undone' by building and executing a specific type of transactions, denoted *undo* transactions. Undo transactions *semantically* revoke the effect of a committed transaction without really undoing it, so this model keeps durability. In particular, to 'undo' a committed transaction $T_i$, the corresponding undo transaction $U_i$ is built as follows: for each update (write) operation of $T_i$, a write operation is appended to the program of $U_i$ (in reverse order) which writes the before value of the item updated. Undo transactions comprise only write operations. It should be noticed that using this model to 'undo' a committed transaction can be quite different from a traditional undo operation. Undoing a committed transaction $T_i$ can only remove its direct effects, but cannot remove the indirect effects (if there are any) of $T_i$ which are caused by the transactions affected by $T_i$. In contrast, undoing an active transaction $T_j$ can remove all the effects of $T_j$ because the isolation property ensures that $T_j$ can be backed out without affecting other transactions.

Although undo transactions are easy to build, the flat transaction recovery model cannot exploit traditional undo mechanisms. It is desirable that we can directly use existing undo facilities (to do attack recovery) without sacrificing performance objectives. This goal can be achieved by a *nested-transaction* recovery model shown in Figure 2.

Consider a commercial database system where a history is composed of a set of (committed) flat transactions [‡], for a history to repair, we build the model by introducing a specific virtual transaction, called *malicious activity recovery transaction* (MART), on top of the history, and letting the MART be the parent of all the flat transactions in the history. As a result, the history is evolved into a nested structure where the MART is the top-level transaction and each flat transaction turns out to be a subtransaction whose execution is controlled by the MART. Since subtransactions can theoretically be undone or compensated at any time before the corresponding top-level transaction commits [26, 22, 12], the model inherently supports undoing flat (commercial) transactions. This is also one of the reasons why we use the word 'undo' to denote one of our basic repair operations.

One interesting question about the model is 'Can a MART commit or abort?'. It is clear that aborting the MART is equivalent to rolling back the history to its initial state. However, how to

---

[‡]Note that the recovery model can be easily extended to incorporate histories of multilevel or nested transactions.

Malicious Activity
Recovery Transaction

B 1   C 1   G 2   C 2   G 3   C 3   G 4   C 4

Decision to
accept history
up to given point

Commit

History          Commercial Database

commit work in database
up to certain point
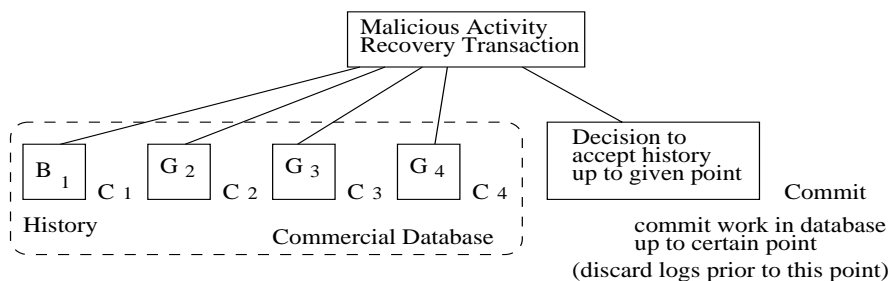(discard logs prior to this point)

Figure 2: Using Nested Transactions to Model Attack Recovery

commit the MART is tricky. In fact, the MART should be able to commit, because as the system keeps on executing new transactions the history can get tremendous long and the MART needs to maintain too much information for the purpose of attack recovery if the MART never commits. In practice, such information may no longer be available for a transaction $T_i$ after $T_i$ is committed for a long period of time. However, if we commit a MART at the end of the current history and start another new MART, then the work of some malicious transactions in the history supervised by the old MART could be committed before they are recovered. Hence we need to commit the work of good transactions while keeping the ability to recover from bad transactions. This goal is achieved by the following MART splitting protocol which is motivated by [29].

- When the history is recovered to a specific point $p_i$, that is, it is believed that the effects of every bad transaction prior to $p_i$ are removed, we can commit the work of all the transactions prior to $p_i$ by splitting the MART into two MARTs: one supervising all the transactions prior to $p_i$, the other supervising the latter part of the history. Interested readers can refer to [29] for a concrete process of transaction splitting, which is omitted here.

- We commit the MART which supervises the part of the history prior to $p_i$. From the perspective of attack recovery, the corresponding log records prior to $p_i$ can be discarded to alleviate the system's resource consumption.

- We keep the other MART active so it can still be repaired.

The advantage of the nested transaction recovery model is that it fits in current commercial database systems very well thus attack recovery need not be designed from scratch. First, flat transactions can be undone by directly applying traditional undo operations. In fact, in this model a savepoint is generated after each subtransaction commits so that the MART can rollback its execution to the beginning of any flat transaction. Second, this model causes very little performance penalty. The drawback of this model is that after a MART commits there is no automatic ways to undo a flat transaction supervised by the MART even if the transaction is later on identified malicious. Therefore, decisions to commit a MART should be made carefully.

Finally, it should be noticed that both the flat-transaction and the nested-transaction recovery models are used in our attack recovery algorithms. In particular, the nested-transaction recovery model is used by the coldstart repair algorithms presented in Section 5, and the flat-transaction model is used by the warmstart repair algorithms presented in Section 6 and Section 7.

# 4 The Repair Model

## Assumptions

We assume that the histories to be repaired are serializable histories generated by some mechanism that implements a classical transaction processing model [6]. We denote committed undesirable or *bad* transactions in a history by the set $\mathbf{B} = \{B_1, B_2, ..., B_m\}$. We denote committed desirable or *good* transactions in a history by the set $\mathbf{G} = \{G_1, G_2, ..., G_n\}$. 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 $T_j$. Two operations *conflict* if they are on the same data item and one is write. Two transactions *conflict* if they have conflicting operations.

## Transaction Dependencies

One simple repair is to roll back the history until at least the first bad transaction, and then try to reexecute all of the undone good transactions. The drawback of this approach is that many good transactions may be unnecessarily undone and reexecuted. Consider the following history over $(B_1, G_1, G_2)$:

$$H_1 : r_{B_1}[x]w_{B_1}[x]c_{B_1}r_{G_1}[y]r_{G_2}[x]w_{G_1}[y]c_{G_1}w_{G_2}[x]c_{G_2}$$

It is clear that $G_1$ 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 if there exists a data item $x$ such that:

1. $T_j$ reads $x$ after $T_i$ has updated $x$;

2. $T_i$ does not abort before $T_j$ reads $x$; and

3. every transaction (if any) that updates $x$ between the time $T_i$ updates $x$ and $T_j$ reads $x$ is aborted before $T_j$ reads $x$.

Every good transaction that is dependent upon some bad transaction needs to be undone and re-executed. There are also other good transactions that also need be undone and reexecuted. Consider the following history over $(B_1, G_1, G_2)$:

$$H_2 : r_{B_1}[x]w_{B_1}[x]c_{B_1}r_{G_1}[x]w_{G_1}[x]r_{G_1}[y]w_{G_1}[y]c_{G_1}r_{G_2}[y]w_{G_2}[y]c_{G_2}$$

$G_2$ is not dependent upon $B_1$, but it should be undone and reexecuted, because the value of $x$ which $G_1$ reads from $B_1$ may affect the value of $y$ which $G_2$ reads from $G_1$. This relation between $G_2$ and $B_1$ is captured by the transitive closure of the dependent upon relation:

**Definition 2** In a history, transaction $T_1$ *affects* transaction $T_2$ if the ordered pair $(T_1, T_2)$ is in the transitive closure of the *dependent upon* relation. A good transaction $G_1$ is *suspect* if some bad transaction $B_1$ affects $G_1$.
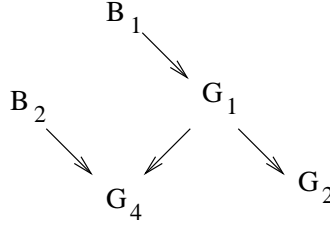
Figure 3: Dependency Graph for History $H_3$

It is convenient to define the *dependency graph* for a (any) 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 suspect transaction. Note that one bad transaction can be dependent upon another bad transaction. Note also that a bad transaction can have no transactions dependent upon itself.

As an example, consider the following history over $(B_1, B_2, G_1, G_2, G_3, G_4)$:

$$H_3 : r_{B_1}[x]w_{B_1}[x]c_{B_1}r_{G_1}[x]w_{G_1}[x]r_{G_3}[z]w_{G_3}[z]c_{G_3}r_{G_1}[y]w_{G_1}[y]c_{G_1}$$
$$r_{G_2}[y]w_{G_2}[y]r_{B_2}[z]w_{B_2}[z]c_{B_2}r_{G_2}[v]w_{G_2}[v]c_{G_2}r_{G_4}[z]w_{G_4}[z]r_{G_4}[y]w_{G_4}[y]c_{G_4}$$

$DG(\mathbf{B})$ is shown in Figure 3.

If a good transaction is not affected by any bad transaction (for example, $G_3$ in $H_3$), then the good transaction need not be undone and reexecuted. In other words, only the transactions in $DG(\mathbf{B})$ need be undone, and only the suspect transactions in $DG(\mathbf{B})$ need to be reexecuted. From the recovery perspective, the goal is to first get $DG(\mathbf{B})$, then undo all these transactions.

Before we continue, we modify our model with respect to blind writes.[§] We developed the model as is because it captures exactly the set of suspect transactions that must be undone, assuming that further information about the transactions – such as data flow or semantic information – is unavailable. Specifically, the model includes an optimization for blind writes. Suppose a transaction in $\mathbf{B}$ writes $x$ and subsequently a good transaction blindly overwrites $x$. Then the *dependent upon* chain is broken, and other good transactions that subsequently read $x$ will not necessarily appear in $DG(\mathbf{B})$.

From the perspective of the recovery algorithms developed in this paper, we view this optimization as counterproductive for two reasons. First, blind writes are relatively infrequent in many applications. Second, accommodating blind writes would complicate the recovery algorithms we present. We make the design decision that the optimizations of blind writes are not worth the additional storage and processing time that would be required in the algorithms. To accommodate this decision, for the remainder of this paper 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.

We say a data item $x$ is *dirty* if $x$ is in the write set of any bad or suspect transaction. From the data perspective, the goal is to restore each dirty data item to the value it had before the first transaction in $DG(\mathbf{B})$ wrote it. The resulting state will appear as if the bad and suspect transactions never executed.

---

[§]A transaction blindly writes a data item if it writes a value without first reading the item.

It is clear that the dependency graph of history $H$ can not be built without the corresponding read information for transactions in $H$. Unfortunately, the read information we can get from the logs for traditional recovery purposes such as physical logs, physiological logs, and logical logs [12], is usually not enough for constructing $DG(\mathbf{B})$. Therefore, 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.

There are several possible ways to maintain and capture read information. For example,

- Augment the write log to incorporate read information.

- Extract read sets from the profiles of transactions.

- Extract read information from physiological or logical logs.

- Build an online dependency graph.

Based on the amount of available read information provided by these methods, we can achieve several types of repair:

- A repair is *complete*, if the effects of every bad or suspect transaction are repaired.

- A repair is *exact*, if the effects of all and only bad or suspect transactions are repaired.

Since the specification and the properties of our repair algorithms are closely related to the approach which is selected to maintain the read information, we present the algorithms in a way which is based on read information capturing methods, although the basic ideas of these algorithms are very similar. The coldstart as well as the warmstart repair algorithms based on In-Log read information are introduced in Section 5 and Section 6 respectively. The repair algorithms based on the read information extracted from transaction profiles are specified in Section 7.

## 5   Static Repair Based on In-Log Read Information

Our basic repair algorithm is based on traditional recovery mechanisms [6]. One advantage of this approach is that we need not develop the repair algorithm from scratch. In addition, the standard recovery mechanisms need not be modified greatly to accommodate the repair algorithm.

We use the same physical log as used in traditional recovery mechanisms [6] except that we define a new type of log record to document every read operation. These records are used to construct the dependent upon relation between transactions. The read log record $[T_i, x]$ denotes that the data item $x$ is read by transaction $T_i$. An algorithm that does not modify the log, but instead maintains the read log separately, is discussed in section 5.2. We use the same TM-Scheduler-DM model of centralized database systems as used in [6]. Here TM denotes Transaction Manager, DM denotes Data Manager. We add one action, which appends $[T_i, x]$, a read record, to the log, to the RM-Read$(T_i, x)$ procedure. Here RM denotes Recovery Manager, a part of the DM. We assume that the scheduler invokes RM operations in an order that produces a serializable and strict execution.

The basic idea of static repair is that we halt the processing of transactions periodically after a set of bad transactions $B$ is identified, and then we build the $DG(\mathbf{B})$, based on the log and/or other available read information, to identify the bad as well as the suspect transactions.

## 5.1 Two Pass Repair Algorithm

The algorithm described below is composed of two passes. Pass one scans the log forward from the entry where the first bad transaction commits to locate every bad and suspect transaction. Pass two goes backward from the end of the log to undo all bad and suspect transactions.

**Algorithm 1** Two Pass Repair Algorithm
**Input:**  the log, the set **B** of bad transactions.
**Output:**  a consistent database state in which all bad and suspect transactions are undone.
**Initialization.**
Let $commit\_list := \{\}$, $undo\_list := \mathbf{B}$, $write\_set := \{\}$, $tmp\_write\_set := \{\}$, $tmp\_undo\_list$ /* See Comment A */
**Pass 1.**
1. Locate the log entry where the first bad transaction $B_1$ starts.
2. Scan forward until the end of the log. For each log entry,
2.1 **if** the entry is for a transaction $T_i$ in **B**
    **if** the entry is a write record $[T_i, x, v]$
      $write\_set := write\_set \cup \{x\}$;
2.2 **else**
    **case** the entry is a write record $[T_i, x, v]$
      $tmp\_write\_set := tmp\_write\_set \cup \{(T_i, x)\}$; /* See Comment B */
    **case** the entry is a read record $[T_i, x]$
      **if** $T_i$ is in the $tmp\_undo\_list$
        skip the entry;
      **if** $x$ is in the $write\_set$
        $tmp\_undo\_list := tmp\_undo\_list \cup \{T_i\}$; /* See Comment C */
    **case** the entry is an abort record $[T_i, abort]$
      delete all the data items of $T_i$ from the $tmp\_write\_set$;
      **if** $T_i$ is in the $tmp\_undo\_list$
        delete $T_i$ from the $tmp\_undo\_list$;
    **case** the entry is a commit record $[T_i, commit]$
      **if** $T_i$ is in the $tmp\_undo\_list$
        move $T_i$ from the $tmp\_undo\_list$ to the the $undo\_list$;
        move all the data items of $T_i$ from the $tmp\_write\_set$ to the $write\_set$;
      **else** delete all the data items of $T_i$ from the $tmp\_write\_set$;
**Pass 2.**
Scan backward from the end of the log to undo all the transactions in the $undo\_list$.

**Comments**

**A.**  The $commit\_list$ consists of the transactions which commit after the first bad transaction. The $undo\_list$ consists of the bad and suspect transactions that should be undone. The $tmp\_undo\_list$ is used to capture the set of in-repair good transactions that have read some dirty data. A transaction $T$ is *in-repair* between the time we scan the record $[T, begin]$ and the time we scan the record $[T, commit]$ or the record $[T, abort]$. The $write\_set$ consists of the dirty data items. The use of $tmp\_write\_set$ is explained in comment B.

**B.**  We need to keep track of the data items written by each in-repair good transaction because the transaction may be later on found suspect, and at that moment we need to add these data

10

items to the *write_set*. There are basically two approaches to solve this problem. One, which is used in the algorithm, is to keep the write items in a temporary memory structure (namely, *tmp_write_set*); the other is to scan the log backward to figure out the write items later on when the transaction is found suspect (the backward scan can be efficient since all the write log entries of a transaction are chained together in the log). The first approach costs more memory space but is faster. The second approach costs less memory space but is slower since it may cause disk operations.

Also, since we assume that the history to be repaired is strict, the following scenario, which happens in a history that is recoverable but not strict, will not occur:

$$r_{G_1}[x_1]w_{G_1}[x_1]r_{G_2}[x_1]w_{G_2}[x_1]r_{G_1}[y_1]w_{G_1}[y_1]c_{G_1}c_{G_2}$$

Suppose $y_1$ is in the *write_set*, $x_1$ and $x_2$ are not. When we encounter the entry $r_{G_2}[x_1]$, though $G_2$ is dependent upon $G_1$ we skip it according to the algorithm since $x_1$ is not in the *write_set*. Later when we encounter the entry $r_{G_1}[y_1]$, we will add $G_1$ to the *undo_list* since it reads an item in the *write_set*. But at this point, $G_2$ will not be added to the *undo_list* though it has been affected by $G_1$.

**C.** We cannot put $T_i$ into the *undo_list* because $T_i$ may be later on found aborted.

**Theorem 1** Given the state produced by history $H$ over $\mathbf{B} \cup \mathbf{G}$, Algorithm 1 constructs the state that would have been produced by $H'$, where $H'$ is $H$ with all transactions in $DG(\mathbf{B})$ removed.

**Proof:** Given the relationship between *dirty* data, the *bad* and *suspect* transactions, this theorem amounts to showing that each dirty data item is restored to the latest value before it turns dirty. The following three claims are sufficient to show this.
*Claim 1.* Every bad and suspect transaction is added to the *undo_list* in Pass 1. It is clear that every bad transaction is added to *undo_list* in step 2.1. Suppose there are some suspect transactions which have not been added to the *undo_list* and $T_i$ is the first one. Then according to the algorithm, it happens that when $T_i$ reads a dirty item $x_i$ in step 2.2 $x_i$ is still not in the *write_set*. Since the execution is strict, when $T_i$ reads $x_i$ every bad or suspect transaction that writes $x_i$ before the read operation has already committed, and therefore $x_i$ is already added to the *write_set* in step 2.1 or 2.2, which contradicts with the assumption.
*Claim 2.* Only bad and suspect transactions are added to the *undo_list* in Pass 1. It is clear that every aborted transaction will not be added to the the *undo_list* in Pass 1. Suppose some non suspect good transactions have been added to the *undo_list* and $T_i$ is the first one. Then according to the algorithm, it happens that when $T_i$ reads an item $x_i$ in step 2.2 $x_i$ is in the *write_set*. Therefore $T_i$ is indeed suspect, which contradicts the assumption.
*Claim 3.* Every dirty data item is restored to the latest value before it turns dirty. Suppose $x_i$ is a dirty data item and $T_i$ is the first transaction which makes $x_i$ dirty, then $T_i$ must be either bad or suspect, and $x_i$ will be cleaned after $T_i$ is undone in pass 2 because (1) $T_i$ will be put into the *undo_list* in pass 1; (2) no bad or suspect transaction that commits before $T_i$ has updated $x_i$; (3) no non suspect good transaction will be undone.  □

## 5.2   Repair Algorithm Based on Separate Read Log

The two pass repair algorithm is based on the log to which read records are added. Sometimes, it is desirable to use a separate log to document the read operations rather than change the traditional log. We call the separate log *read log*, and we call the traditional log *update log*. Using a read log

to repair a history has the advantage that the traditional recovery mechanisms do not have to be modified to take a different data structure for the log into account.

There is only one type of entry of the form $[T_i, x]$ in the read log, identifying the data item $x$ which is read by transaction $T_i$.

Conceptually, a two pass repair algorithm based on the read log as well as the update log can be designed using the same memory data structure and algorithm as used in Algorithm 1 if we can transform the serial scan operations in Algorithm 1 over one log to some equivalent interleaved scan operations over the read log and the update log. Thus, one important issue in using a read log to do repair is to synchronize the scan operations over the update log and the read log.

The order by which we interleave the scan operations over the update log and the read log is critical to the correctness of the repair algorithm. If an entry $[T_i, x]$ in the read log is scanned earlier than an entry in the update log which denotes an operation happening before the read, then $T_i$ may not be added to the *undo_list* though it is a suspect transaction. Look at the following scan sequence:

$$r_{G_1}[x_1]r_{G_1}[y_1]w_{G_1}[y_1]r_{G_2}[y_1]w_{G_1}[x_1]c_{G_1}w_{G_2}[y_1]c_{G_2}$$

Suppose $x_1$ is in the *write_set*. When we scan the entry $r_{G_2}[y_1]$, we cannot find that $G_2$ reads a dirty item since $y_1$ is not in the *write_set* yet. Later on, when we scan the entry $c_{G_1}$, we will add $y_1$ to the *write_set* but $G_2$ will be found not to be suspect since it will not read $y_1$ again. The point is that $r_{G_2}[y_1]$ happens after $c_{G_1}$ (since the execution is strict) but it is scanned before $c_{G_1}$.

If an entry $[T_i, x]$ in the read log is scanned later than an entry in the update log which denotes an operation happening after the read, then we may not find the write items of $T_i$ in the *tmp_write_set* when we find $T_i$ suspect since all the write items of $T_i$ may have been deleted. Look at the following scan sequence:

$$w_{G_1}[x_1]w_{G_1}[y_1]c_{G_1}r_{G_1}[x_1]r_{G_1}[y_1]$$

Suppose $x_1$ is in the *write_set*. When we scan the entry $c_{G_1}$, we will delete all the write items of $G_1(x_1, y_1)$ from the *tmp_write_set* since $G_1$ has not read any dirty data. Later on, when we encounter the entry $r_{G_1}[x_1]$, we find $G_1$ is suspect but we cannot find the items written by $G_1$ from the *tmp_write_set*.

So we must synchronize the scan operations in a way which can ensure the correctness of the algorithm. The requirement implied by this can be conveniently stated as two design rules that every two pass repair algorithm which uses read logs must observe.

**Rule 1:** Before a read entry $[T_i, x]$ is scanned in the read log, any write record for $x$ which denotes an operation happening before the read must have been scanned.

**Rule 2:** Before we scan a commit record $[T_i, commit]$ in the update log, all the read records for $T_i$ must have been scanned.

Our synchronizing mechanism is specified as follows.

**Mechanism 1** When a read entry is added to the read log, the largest $LSN$ (Log Serial Number) [6] of the update log will be recorded in the read entry. The LSN of a log entry is an ascending serial number which indicates the chronological order of logged operations. We scan the update log using a pointer $p_u$. We scan the read log using another pointer $p_r$. Let $p_u.LSN$ denote the LSN of the log entry pointed by $p_u$, let $p_r.read\_LSN$ denote the LSN recorded in the log entry pointed by $p_r$. We scan the update log and the read log as follows.

1. We start with $p_u$ pointed to the first update log entry of $B_1$, and $p_r$ pointed to the first read log entry after $B_1$ commits. We can locate $p_r$ by searching for an entry with its $read\_LSN$ larger than the LSN of the commit log entry of $B_1$.

2. If $p_r.read\_LSN < p_u.LSN$, then we scan the entry pointed by $p_r$ and then increase $p_r$ by 1. Otherwise, we scan the entry pointed by $p_u$ and then increase $p_u$ by 1.

3. We repeat the previous step until the ends of both the update log and the read log are scanned.

**Lemma 1** Mechanism 1 ensures that the scan order of the update log and the read log is the order in which the corresponding operations happen. Moreover, Mechanism 1 satisfies the two design rules.

**Proof:** For an entry $u_1$ in the update log and an entry $r_2$ in the read log, let $u_1.LSN$ denote the LSN of $u_1$, let $r_2.read\_LSN$ denote the LSN recorded in $r_2$. Let $o_1$ be the operation recorded by $u_1$ ($o_1$ may be a commit, abort, or update operation); let $o_2$ be the read operation recorded by $r_2$. If $r_2.read\_LSN \geq u_1.LSN$, then $o_1$ happens before $o_2$ because according to the way we maintain the $read\_LSN$ field it is easy to see that when $o_2$ happens $o_1$ is already in the update log. If $r_2.read\_LSN < u_1.LSN$, then $o_1$ happens after $o_2$ because if not then $r_2.read\_LSN \geq u_1.LSN$, which contradicts the assumption. $\square$

The repair algorithm based on separate read log is described as follows. Its correctness can be ensured by Lemma 1 and Theorem 1. Note that Algorithm 2 is actually a combination of Algorithm 1 and Mechanism 1. Algorithm 2 conceptually uses Mechanism 1 to provide the right input to Algorithm 1.

**Algorithm 2** Repair Algorithm Based on Separate Read Log
Use Mechanism 1 to schedule the order in scanning the update log as well as the read log. For every kind of log entry, do the same thing as Algorithm 1.

# 6 On-the-Fly Repair Based on In-Log Read Information

The two pass algorithm and the repair algorithm based on separate read log which we have presented in Section 5 are all static repair, or coldstart, methods. New transactions are blocked during the repair process. In some database applications, availability requirements dictate that new transactions be able to execute concurrently with the repair process, that is, the application requires warmstart semantics for recovery. The cost of on-the-fly repair is that some new transactions may inadvertently access and subsequently spread damaged data.

The traditional transaction management architecture is adequate to accommodate on-the-fly repair (see Figure 4). The *Repair Manager* is applied to the growing logs of on-the-fly histories to mark any bad as well as suspect transactions. For every bad or suspect transaction, the Repair Manager builds an undo transaction and submits it to the *Scheduler*. The undo transaction is only composed of write operations. The *Scheduler* schedules the operations submitted either by user transactions or by undo transactions to generate a correct on-the-fly history. Suspect transactions that are undone can be resubmitted to the Scheduler either by users or by the Repair Manager. The *Recovery Manager* executes the operations submitted by the Scheduler and logs them. It keeps the read information of transactions either in a traditional log modified to store read operations or in a separate read log.

For simplicity, in the rest of this section we first assume that each new transaction is good. We address the issues on repairing the damage caused by bad new transactions at the end of this section.
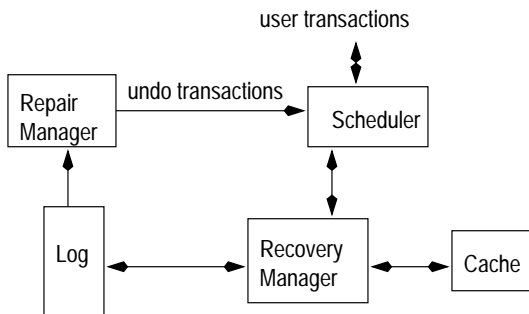
13

Figure 4: Architecture of the On-the-fly Repair System



Figure 5: A Snapshot of Repair on the Log

On-the-fly repair is different from static repair in several aspects. First, since user transactions keep on being submitted and executed, on-the-fly repair faces a damage confinement problem, that is, damaged data items should not be accessed until being repaired. Therefore, in addition to identifying suspect transactions, on-the-fly repair has to *mark* or identify damaged data items so that they can be confined. Before a data item is marked dirty, we do not know it is dirty and it will not be confined. Second, in order to give more availability we should clean the items that are confined as soon as possible. Therefore, we cannot start to undo bad and suspect transactions backwards until all the suspect transactions are identified. Instead, we should undo bad and suspect transactions forwards as soon as they are identified. Third, on-the-fly repair needs to perform concurrency control among undo as well as user transactions.

## 6.1   Termination Detection

New transactions are continuously submitted to the Scheduler, and as a result, the log keeps growing. A key question is 'Does repair terminate, and if so, is termination detectable?'

Suppose at some point the Repair Manager has repaired the history up to record $a$ on the log (See Figure 5). That is, every bad or suspect transaction which commits before $a$ is logged has been undone, its dirty data items have been marked and cleaned. Suppose record $b$ is the current bottom of the log. It is possible that a newly submitted read operation reads a dirty item which has not been marked, because the item can be made dirty by some write operation which happened between $a$ and $b$. Since neither the Scheduler nor the Repair Manager can detect this, the read operation is not rejected. In this way, some newly submitted good transaction may become suspect. As an example, consider the following operation sequence:

$$r_{G_{i1}}[x_1]w_{G_{i1}}[x_2]c_{G_{i1}}r_{G_{i2}}[x_2]w_{G_{i2}}[x_3]c_{G_{i2}}...r_{G_{ik}}[x_k]w_{G_{ik}}[x_{k+1}]c_{G_{ik}}...$$

Even if $x_1$ is the only dirty data item when the sequence begins, repair may not terminate until the submission terminates because when $x_i$ is cleaned, $x_{i+1}$ may have already become dirty.

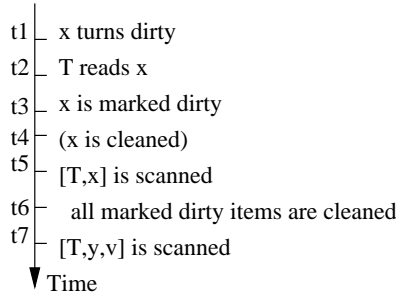Consider another operation sequence:

```
t1 ⌐  x turns dirty
t2 ⌐  T reads x
t3 ⌐  x is marked dirty
t4 ⌐  (x is cleaned)
t5 ⌐  [T,x] is scanned
t6 ⌐    all marked dirty items are cleaned
t7 ⌐  [T,y,v] is scanned
   ▼ Time
```

Figure 6: Transactions which have been found suspect may generate new dirty items

$$G_{i1}G_{i2}...G_{i(k-1)}r_{G_{ik}}[x_1]w_{G_{ik}}[x_2]c_{G_{ik}}...$$

Assume that only $x_1$ is dirty when the sequence begins and none of the transactions between $G_{i1}$ and $G_{i(k-1)}$ read $x_1$. Then it is possible that when $G_{ik}$ reads $x_1$, every bad or suspect transaction has already been repaired. Thus, $x_1$ is clean, and the repair terminates.

A repair in terms of a set of bad transactions **B** terminates when all the damage that has already been caused by **B** is cleaned and no damage can be (indirectly) caused by **B** in the future. Whether or not a repair terminates depends on the repair speed, the arrival rate of new transactions, and the nature of the new transactions. So, in general, termination of repair cannot be guaranteed without taking additional measures, which are discussed later. In other words, if we do not take extra measures, in some cases the repair process terminates while in some other cases the repair process may never terminate (especially when the database system is always busy). Fortunately, we found that in the cases when a repair terminates we can detect the termination. We turn to termination detection next.

Checking if every marked dirty data item has been cleaned to determine if repair is complete is not sufficient for two reasons. First, some transaction $T$ which has been found suspect may write dirty data items later on (see Figure 6): at time $t_5$ the read record $[T, x]$ is scanned and $T$ is found suspect since $x$ was dirty when T read $x$ (Notice that when $[T, x]$ is scanned $x$ may not be dirty since $x$ may already be cleaned at $t_4$); at time $t_6$ every dirty item that is marked before $t_6$ has been cleaned, but the repair does not terminate since at time $t_7$, $T$ writes an item $y$ and $y$ becomes dirty. Second, some transaction which has not yet been identified as suspect may generate dirty data (See Figure 7): $[T, begin]$ record is scanned after time $t_4$ when no data is dirty, we can not stop repair at time $t_4$ since at time $t_6$ we find $T$ is suspect and at time $t_7$ item $y$ is marked dirty.

From another perspective, when data item $x$ is read or written, $x$ may be in one of the three kinds of states shown in Figure 8. Before $x$ turns dirty, $x$ is in the 'clean' state. $x$ is in the 'pseudo clean' state between the time $x$ turns dirty and the time $x$ is marked dirty. $x$ is in the 'marked dirty' state between the time $x$ is marked dirty and the time $x$ is cleaned. $x$ is dirty when it is in either the 'pseudo clean' state or the 'marked dirty' state. When $x$ is cleaned, $x$ returns to the 'clean' state. Note that cleaned items can be damaged again, thus they can turn dirty and be cleaned again.

Mechanism 2 described below can capture the two situations shown in Figure 6 and Figure 7, thus it can detect the termination of on-the-fly repair processes.

**Mechanism 2** In the process of repair:

- Maintain a *dirty_item_set* to keep every data item in the 'marked dirty' state; maintain a

15

t1 ⌊ x turns dirty
t2 ⊢ T reads x
t3 ⊢ x is cleaned
t4 ⊢ all marked dirty items are cleaned
t5 ⊢ [T, begin] is scanned
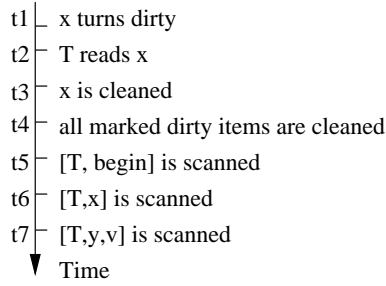t6 ⊢ [T,x] is scanned
t7 ⊢ [T,y,v] is scanned
↓ Time

Figure 7: Transactions which will later be found suspect may generate new dirty items
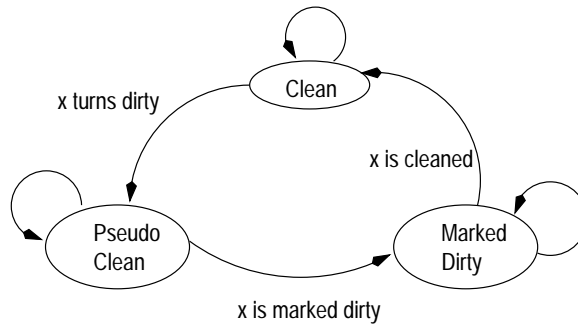


Figure 8: Item State Transition Diagram

*cleaned_item_set* to keep every data item that has been cleaned. We show how to capture these items in Section 6.4.

- Associate each item $x$ in the *cleaned_item_set* with a number, $x.LSN$, which denotes the log serial number of the bottom record of the log at the time when $x$ is cleaned.

- Maintain a *tmp_undo_list* to keep every in-repair transaction that has read some data item in the *dirty_item_set*, or has read an item $x$ in the *cleaned_item_set* where $r.LSN \leq x.LSN$. Here $r.LSN$ is the log serial number of the read record.

- We report that the repair terminates if

  - every bad transaction in **B** has been undone, and
  - *dirty_item_set* $= \emptyset$, and
  - *tmp_undo_list* $= \emptyset$, and
  - $\forall x \in$ *cleaned_item_set*, $x.LSN < l.LSN$. Here, $l.LSN$ denotes the log serial number of the next log record for the Repair Manager to scan.

**Theorem 2** Mechanism 2 reports termination iff the repair process, in fact, terminates.

**Proof:** Repair terminates iff all the marked dirty items have been cleaned and it is not possible for any item to turn dirty later on. When Mechanism 2 reports termination every marked dirty item
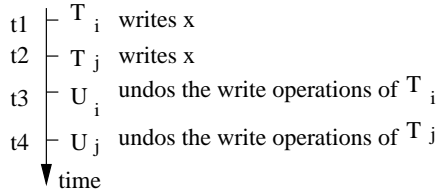
16

Figure 9: Flaw of the Staightforward Undo Method

has been cleaned since $dirty\_item\_set = \emptyset$. At this time, since every bad transaction in **B** has been undone, an item $x$ may turn dirty later on only if $x$ is written by a suspect transaction which has been detected or by a suspect transaction which will be detected later on.

A transaction $T$ can be found suspect only if there is an item $x$ such that $T$ read $x$ when $x$ was dirty. When $[T, x]$ is scanned, $x$ may still be dirty or may have been cleaned, but $x$ can not be first cleaned and then marked dirty for the following reason. Suppose the transaction that makes $x$ dirty again is $T'$. Then the write record $[T', x, v]$ can only be scanned after $[T, x]$ since it is appended to the log after $[T, x]$. Therefore, when $[T, x]$ is scanned $x$ is still dirty, and so $x$ must be in the $dirty\_item\_set$. If $x$ has been cleaned, then $r.LSN \leq x.LSN$. So every transaction that has been found suspect will be in the $tmp\_undo\_list$. Therefore, when $tmp\_undo\_list = \emptyset$ no such transaction exist.

When $dirty\_item\_set = \emptyset$ an item $x$ will be written by a transaction $T$ which will be found suspect later on only if $T$ had read $x$ before $x$ is cleaned, but when $T$ reads $x$, $x$ is still dirty. (This situation is shown in Figure 7.) When Mechanism 2 reports termination, $\forall x \in cleaned\_item\_set$, $x.LSN < l.LSN$, that is, every dirty item is cleaned before the operation denoted by the next log record for the Repair Manager to scan. Therefore, every read operation denoted by a record that the Repair Manager is going to scan will not read any dirty item, so the situation will not happen.

$\square$

## 6.2 Building Undo Transactions

On-the-fly repair requires the Repair Manager build and submit the undo transactions for every bad or suspect transaction, that is, the Repair Manager starts to built the undo transaction for a transaction $T_i$ as soon as $T_i$ is found bad or suspect. Since the log keeps on growing, the undo can only be done from the beginning to the end of the history, which is different from the methods presented in Section 5. Note that here we still assume that every bad or suspect transaction is committed.

The straightforward way to build undo transactions for bad or suspect $T_i$ is to scan backward along the log from the point where $T_i$ commits, and for every write record of $T_i$, add a corresponding write operation to the undo transaction $U_i$. The write operation in $U_i$ restores the item to its old value. This approach does not work if new transactions execute concurrently with repair. Consider the event sequence shown in Figure 9. If $x$ is clean before $T_i$ writes x, $T_i$'s undo transaction $U_i$ undos this write operation at time $t_3$ and $x$ is cleaned. However, the undo transaction $U_j$ of another suspect transaction $T_j$ undos the write operation of $T_j$ on $x$ at time $t_4$ and $x$ turns dirty again, which is not correct.

Algorithm 3 described below fills the hole of the straightforward method.

**Algorithm 3** Building Undo Transactions

1. Maintain a *submitted_item_set* to keep every item $x$ whose undo operation has been submitted to the Scheduler, but $x$ still has not been cleaned.

2. When building an undo transaction, for every write record which is scanned, if the record is on an item $x$ which is in the *cleaned_item_set* or in the *submitted_item_set* then omit the record; if $x$ is in the *write_item_set* but not in the *submitted_item_set*, then add the corresponding undo operation to the undo transaction and add $x$ to the *submitted_item_set*.

**Theorem 3** In Algorithm 3, when $U_i$ is built, every dirty data item $x$ of $T_i$ will either be repaired in an operation of $U_i$ or in an operation of another undo transaction, and $x$ will be restored to the value $x$ had before it turned dirty.

**Proof:** If $x$ is clean or cleaned before $T_i$ writes $x$, then the undo operation $w_{U_i}[x]$ will restore $x$ to the latest value before $x$ turned dirty. If $x$ is dirty before $T_i$ writes $x$, suppose $T_j$ is the transaction which makes $x$ dirty, then when $[T_i, x, v]$ is scanned, $x$ is either cleaned, so in the *cleaned_item_set*, or is submitted by the undo transaction which is built for $T_j$, so in the *submitted_item_set*, therefore, $U_i$ will not repair $x$. $\square$

## 6.3    On-the-fly Concurrency Control

Before introducing the On-the-fly repair algorithm, we need to first analyze how the Scheduler should schedule the user operations as well as the undo operations to achieve repair.

To define the acceptable histories generated by the Scheduler, we associate the read and undo operations in histories with appropriate states of the Repair Manager (i.e., the state of the *dirty_item_set*) when the operations are scheduled to execute, and use the states to indicate the correctness of repair histories.

**Definition 3** History $H$ is a *correct on-the-fly history* if

1. $H$ is serializable and strict,

2. There are no abort records for undo transactions,

3. For any read operation $r_{T_i}[x]$, the predicate $x \notin dirty\_item\_set$ holds,

4. For any conflicting undo transaction pair $U_i$ and $U_j$, if $T_i <_H T_j$ then $U_i <_H U_j$, and

5. For any undo operation $w_U[x]$, the predicate $(x \in dirty\_item\_set) \cap (x \in submitted\_item\_set)$ holds.

Statement 3 says that when a read operation $r_{T_i}[x]$ is scheduled $x$ must be clean. Statement 4 says that conflicting undo transactions should be scheduled in the same order in which they are submitted by the Repair Manager (As shown in Section 6.2, the order is critical to the correctness of repair.). Statement 5 says that when an undo operation $w_U[x]$ is scheduled, $x$ must be dirty.

The scheduling algorithm is described as follows:

**Algorithm 4** Scheduling Algorithm
The algorithm is based on strict two-phase-locking (2PL) where a transaction will not release any locks it gets until it commits. The modification lies in:

- Never abort undo transactions;

- When a read operation $r_{T_i}[x]$ arrives, if $x$ is in the *dirty_item_set*, then reject this read operation and rollback $T_i$.

We show in next section that if the Scheduler executes Algorithm 4, then together with the Repair Manager, and the Recovery Manager, the Scheduler generates correct on-the-fly histories.

An important task of the Scheduler is to control the submitting speed of user operations so that the repair can eventually terminate. Informally, the Scheduler can slow down the submitting speed of user operations when the repair process fails to terminate in an expected period of time. An automatic way to control the speed is as follows. Periodically the Scheduler evaluates the trend of the size of the *dirty_item_set*. The trend can be captured with some time series analysis techniques. If the trend is up, then the submitting speed can be reduced. Otherwise, termination is on the track.

## 6.4 On-the-fly Repair Algorithm

The integrated On-the-fly repair algorithm consists of three parts which are executed by the Repair Manager, the Scheduler, and the Recovery Manager, respectively.

**Algorithm 5** On-the-fly Repair Algorithm
**Input:** the log, the set **B** of bad transactions.
**Output:** if the repair terminates at the middle of the history, then any prefix $H_p$ of the history including the point where the repair terminates results in the state that would have been produced by $H_p'$, where $H_p'$ is $H_p$ with all transactions in $DG(\mathbf{B})$ removed. If the repair terminates at the end of the history $H$, then $H$ will result in the state that would have been produced by $H'$, where $H'$ is $H$ with all transactions in $DG(\mathbf{B})$ removed.
**Initialization:**
Let *tmp_undo_list* := {}, *cleaned_item_set* := {}, *dirty_item_set* := {}, *tmp_item_set* := {}.
**At the Repair Manager:**
1. Locate the log entry where the first bad transaction $B_1$ starts.
2. **while** (the termination conditions do not hold ¶)
      Scan next log entry:
         **if** the entry is for an undo transaction
            skip it;
         **elseif** the entry is for a transaction $T_i$ in **B**
            **if** the entry is a write record $[T_i, x, v]$ and $x$ is not in the *cleaned_item_set*
               add $x$ to the *dirty_item_set*;
            **if** the entry is a commit record $[T_i, commit]$
               build the undo transaction for $T_i$ using Algorithm 3 and
               submit it to the Scheduler;
         **else**
            **case** the entry is a write record $[T_i, x, v]$
               **if** $x$ is not in the *cleaned_item_set*
                  add $x$ to the *tmp_item_set*;
               **elseif** $w.LSN > x.LSN$ /* See comment A */
                  move $x$ from the *cleaned_item_set* to the *tmp_item_set*;
            **case** the entry is a read record $[T_i, x]$
               **if** $x$ is in the *dirty_item_set* or
               $x$ is in the *cleaned_item_set* and $r.LSN \leq x.LSN$

---

¶The termination conditions are stated in Mechanism 2.

19

add $T_i$ to the *tmp_undo_list*;
  **case** the entry is an abort record $[T_i, abort]$
   delete all the data items of $T_i$ from the *tmp_item_set*;
   **if** $T_i$ is in the *tmp_undo_list*, remove it;
  **case** the entry is a commit record $[T_i, commit]$
   **if** $T_i$ is in the *tmp_undo_list*
    move all the items of $T_i$ from the *tmp_item_set* to the *dirty_item_set*;
    build the undo transaction for $T_i$ using Algorithm 3 and submit it
    to the Scheduler;
   **else** delete all the items of $T_i$ from the *tmp_item_set*;
3. report termination; exit;
**At the Scheduler:**
Schedule the user operations as well as the undo operations using Algorithm 4.
**At the Recovery Manager:**
When an undo operation $w_{U_i}[x]$ is done, delete item $x$ from both the *dirty_item_set* and the *submitted_item_set*, then add $(x, x.LSN)$ to the *cleaned_item_set*.

## Comments

**A.** $w.LSN$ denotes the log serial number of the write record. Notice that when $w.LSN > x.LSN$ $x$ is cleaned before the write operation, therefore, the write operation may make $x$ dirty again. Otherwise, $x$ is cleaned after the write operation, so $x$ will not be made dirty again by this operation, therefore $x$ need not be cleaned anymore.

**Theorem 4** Algorithm 5 meets its specification.

**Proof:** Given the relationship between *dirty* data, the *bad* and *suspect* transactions, this theorem amounts to showing that at the time when the repair terminates each dirty data item is restored to the latest value before the data item turns dirty.
*Claim 1.* The Scheduler generates only correct on-the-fly histories. From the definition of 2PL and Algorithm 4, we know that the first three statements of Definition 3 hold. The Repair Manager builds and submits undo transactions in the scanning order, and before an undo operation $w_{U_i}[x]$ is executed and $x$ is cleaned any conflicting undo operation $w_{U_j}[x]$ will not be submitted to the Scheduler. This is because between the time $w_{U_i}[x]$ is submitted and the time it is executed any newly submitted user transaction which reads $x$ will be rejected, and the Repair Manager will not build any other undo operation to repair $x$. Therefore, statements 4 and 5 hold.
*Claim 2.* Algorithm 5 realizes Mechanism 2, and thus reports termination correctly.
*Claim 3.* In the Repair Manager, at any point of time every dirty data item $x$ in the part of the history having been scanned by the Repair Manager has been marked and the corresponding undo operation, which can restore the value of $x$ to the latest value before $x$ turns dirty, has been built and submitted to the Scheduler. Since in the part of history, an item $x$ can be first made dirty, then cleaned, and then made dirty again, we associate a dirty item $x$ with the period of time when it remains dirty(denoted $p$). Thus $(x, p_1)$ and $(x, p_2)$ denote two different dirty items. As shown in Algorithm 5, for every dirty data item $(x, p)$ an undo operation and only one undo operation will be built to repair it at the very beginning of $p$. See Theorem 1 and Theorem 2 for the reason that every dirty data item is marked. □

In the above presentation, we assume every new transaction is good. However, in some cases, a new transaction can be bad. In static repair, due to the delay of intrusion detection, a bad transaction may be identified during the repair. Similarly in on-the-fly repair, new bad transactions

can be identified at any time during the repair. For simplicity of presentation, we assume that the malicious transaction list will not change in the process of both static and on-the-fly repairs. Nevertheless, new attacks can be easily incorporated into our algorithms. In static repair, newly detected bad transactions can be repaired by re-scanning the log. In on-the-fly repair, when a new bad transaction is identified, we can stop the repair, skip to the place where the first un-repaired bad or suspect transaction begins, and apply the dynamic repair algorithm again. In the new round, the new bad and suspect transactions can be repaired.

# 7   Extracting Read Information from Transaction Profiles

Sections 5 and 6 detail recovery algorithms that, given a specification of malicious, committed transactions, unwind the effects of each malicious transaction, along with the effects of any benign transaction that depends, directly or indirectly on a malicious transaction. The significance is that the work of the remaining benign transactions is saved. However the assumption that read information are kept in the log may incur substantial performance penalties due to the significant storage and processing cost of maintaining read information.

There are basically two ways to keep read information in the write log or in another read log. One way is what we assumed in Sections 5 and 6, that is, let the RM-Read$(T_i, x)$ procedure append the read record $[T_i, x]$ to the log every time when $T_i$ reads an item $x$. The other way is to let the RM-Read$(T_i, x)$ procedure keep 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 put into the log as one record. Compared with the first approach, the second approach saves some storage since the identifier of $T_i$ need not be put into the log repeatedly, however, it may require the database to store relatively large data objects because read sets can be very big. In addition, it may delay termination detection during a warmstart repair process.

Although keeping read information in the log will not cause more forced I/O, it does consume more storage. Though the previous two approaches need to keep only the identifier instead of the value of each read item in the log, the size of a read set can still be very big. For example, in a bank a transaction which generates the monthly statement of a customer needs to read the information of every transaction submitted by the customer during the last month.

Another problem with keeping read information in logs lies in the fact that almost all present database systems keep only update (write) information in the log. Adding read records to the log may cause the redesign of the current recovery mechanisms, including both the data structure and the algorithms.

Any way of maintaining read information should keep the malicious transaction recovery module isolated from the traditional recovery module as much as possible. Such an approach avoids degrading the performance of the traditional recovery module and also makes it easier to build the malicious transaction recovery module on the top of the existing database systems.

In this section, we adopt the approach of extracting read information from the profiles and input arguments of transactions. Compared with the read log approach, 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 attack recovery module can be completely isolated from the traditional recovery module. In this way, our repair model can be easily implemented on the top of current database systems without any change to the DBMSs. The approach is not exact, and as a result, it may back out some non-suspect good transactions and/or delay termination detection during a warmstart repair process.

## 7.1   The Model

We start with the transaction profile of TPC-A, a well known database benchmark [11], as an example. TPC-A is stated in terms of a hypothetical bank. The bank has one or more branches. Each branch has multiple tellers. The bank has many customers, each with an account. The database represents the cash position of each entity (branch, teller, and account) and a history of recent transactions run by the bank. The transaction represents the work done when a customer makes a deposit or a withdrawal against his account. The transaction profile is specified as follows:

```
Input: Aid, Tid, Bid, Delta
BEGIN TRANSACTION
    Update Account_Balance where Account_ID = Aid:
        Read Account_Balance from Account
        Set Account_Balance = Account_Balance + Delta
        Write Account_Balance to Account
    Write to History:
        Aid, Tid, Bid, Delta, Time_stamp
    Update Teller where Teller_ID = Tid:
        Set Teller_Balance =  Teller_Balance + Delta
        Write Teller_Balance to Teller
    Update Branch where Branch_ID = Bid:
        Set Branch_Balance =  Branch_Balance + Delta
        Write Branch_Balance to Branch
COMMIT TRANSACTION
```

Here, Account_ID, Teller_ID, and Branch_ID are the primary keys to the relevant tables. The read sets of this transaction can be specified in both tuple level and field level as follows. In tuple level, each item in the read set is a primary key that denotes a tuple or a record. In field level, each item denotes a field of a record. Field level read sets are usually more accurate because in many cases not every field is useful when a tuple is read. Note that here primary keys are not put into field level read sets although they are sometimes used for locating a data item, since we assume that the primary key of a record is not updated unless the record is deleted.

In tuple level:   Read_Set = { Account.Aid, Teller.Tid, Branch.Bid }
In field level:   Read_Set = { Account.Aid.Account_Balance,
                      Teller.Tid.Teller_Balance, Branch.Bid.Branch_Balance }

## 7.2   Read Set Templates

As shown in the example above, given the source code and the input arguments, it is possible to extract exact or approximate read sets from transactions. However extracting read sets on the fly, that is, analyzing transaction source code during execution, may not meet the requirement of current online transaction processing systems. The reason is that extracting read sets can cause an unacceptable processing delay.

An efficient method of getting read sets is required. Since every transaction running in an OLTP system typically belongs to some category, we assume that a transaction type is associated with every transaction, which identifies the nature of the transaction. Transactions of the same type have the same program code, though they typically execute with different input arguments.

The *read set template* for a transaction type is a representation of the data items that will be read by transactions of the type. Since read set templates are generated based on only transaction

profiles, there are no real input arguments available and each data item in a read set template can only be specified as a function of the input variables.

An efficient way to extract read information from transaction profiles based on read set templates is as follows: first, analyze the source code of each type of transaction off line and get the read set template of that type. Second, when a transaction $T$ is submitted to the Scheduler, *materialize* the read set template for $T$'s type with the input arguments of $T$. The process of materializing is done by substituting each variable in the read set template with its corresponding real value. As a result, the materialized read set template is the read set for $T$. For example, for a TPC-A transaction instance with the input `Aid='A1591749'`, `Tid='T0002'`, `Bid='BGMU001'`, `Delta=$1000`, the read set (in field level) for the transaction after materialization is:

Read_Set = { Account.'A1591746'.Account_Balance, Teller.'T0002'.Teller_Balance,
       Branch.'BGMU001'.Branch_Balance }

As shown in the above example, for any TPC-A transaction instance and for any database state on which the transaction is executed, we can get the *exact* read set using the template, that is, the materialized template will indicate all and only the data items which are read by the transaction, either in tuple level or in field level.

However, in some other circumstances based on a template we may only be able to get an approximate read set. For example, consider the **Stock-Level** transaction profile of TPC-C [11], another well known benchmark. TPC-C portrays a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. Each regional warehouse covers 10 districts. Each district serves about 3000 customers. All warehouses maintain stocks for around 100,000 items sold by the supplier. Customers call the supplier to place a new order or request the status of an existing order. Orders are composed of an average of 10 order items.

The **Stock-Level** transaction retrieves the stock level of the last 20 orders of a district. Taking a district identifier (`w_id`, `d_id` ) as the input (Here `w_id` identifies a warehouse, `d_id` identifies a district covered by the warehouse), the transaction first queries the next available order number for the district, which is recorded in the `D_NEXT_O_ID` field of the `DISTRICT` table. Second, it uses the number to get the identifiers of the last 20 orders. Note that orders are assigned a sequential identifier number. Third, for each of these orders, it queries the `ORDER-LINE` table to get all the items on the order. The items are identified by the `OL_I_ID` field. Finally, it queries the `STOCK` table to get the stock level for each of these items, which is specified by the `S_QUANTITY` field.

The read set template of this transaction type can be specified as follows. Here, $x$ is a variable that denotes the next available order number, $R_1$ is the set of the identifiers of the last 20 orders, $R_2$ is the set of the identifiers of the items on these 20 orders, and the `OL_NUMBER` field keeps a sequential number of order lines.

Template =  { $x$ = District.(w_id+d_id).D_NEXT_O_ID;
      $R_1$ ={ x-1, ..., x-19, x-20 };
      $R_2$ = Order-Line.(w_id+d_id+$R_1$+OL_NUMBER).OL_I_ID;
      Stock.(w_id + $R_2$).S_QUANTITY }

Based on the transaction profile, we can trace the `D_NEXT_O_ID` field from the input, however we can not trace further from $R_1$ to the last 20 orders because the value of $x$ depends on the concrete database state when the transaction is executed. In this scenario, there are two approaches to materialize the template. One is *generalizing*, that is, to view $R_1$ as the set of all order numbers, thus the template can be materialized by only the input arguments. The other is *tracing*, that is, to materialize $R_1$ based on the database state, for example, when doing repair we can scan back from

the point of the log where the transaction was executed to get the value of $x$. Although the second approach can achieve finer repair, it may cause substantial extra costs, especially in dynamic repair scenarios.

Besides *exact* read sets, *potential* read sets maintain approximate read items for transactions. That is, for each item in the potential read set of transaction $T$, there exists a database state under which the item will be read by $T$ when it is executed. It is clear that the potential read set for a transaction is the union of all the possible exact read sets of the transaction. Since we materialize read set templates before transactions are executed, and since we do not predict control flows within transactions, in some cases we may get potential read sets instead of exact read sets. For example, using the 'generalizing' approach we can only get the potential read set of a **Stock-Level** transaction.

Since only database objects can be put into read set templates, we focus on the DML aspect of SQL statements, the interface between transactions and databases. In particular, we consider only **Select** and **Update** statements and view a **Delete** or an **Insert** statement as a special **Update** statement. Note that an **Insert** statement can contain subqueries, and a **Delete** statement can have conditions.

For a **Select** or **Update** statement $s$ of a transaction $T$, the input of $s$ is the values (may be denoted as variables) which are used in the **Where** or **Set** clauses of $s$. The input may come directly from the input of $T$, or indirectly from some previous query or program statement. It is clear that every template extraction operation must satisfy the following properties:

- For each **Select** statement, the template can not be larger than the union of all the relations in its **From** clauses. For each **Update** statement, the template can not be larger than the union of all the relations in its **Update** and **From** clauses. Here the term 'union' means the `union` operation of relational algebra.

- For each transaction, the template can not be larger than the union of all the templates for every **Select** or **Update** statement.

- The data items in the template for transaction $T$ depend only on the transaction program, and not on any particular database state.

Read set templates of transactions can be extracted in three steps:

1. Extract the template for each **Select** or **Update** statement separately.

2. Combine the templates for each **Select** or **Update** statement to get the template for the transaction.

3. Generalize the template as appropriate.

For example, there are two **Select** statements in the **Stock-Level** transaction. In step one, the template for the first statement is:

$TP_1 = \{$ `District.(w_id+d_id).D_NEXT_O_ID` $\}$

The template for the second statement is:

$TP_2 = \quad \{$ $R_1 = \{$ `o_id-1, ..., o_id-19, o_id-20` $\}$;
$\qquad\qquad R_2 = $ `Order-Line.(w_id+d_id+`$R_1$`).OL_I_ID`;
$\qquad\qquad$ `Stock.(w_id+`$R_2$`).S_QUANTITY` $\}$

| Rule | Step | Description |
|------|------|-------------|
| Affecting rule | 1 | If data item $d_1$ affects $d_2$, then $d_1$ should be put into the template so long as $d_2$ is in the template. |
| Set rule | 1 | If the result of a statement is based on a funtion of a set of database objects, then the whole set should be put into the template. |
| Join rule | 1 | If the result of a statement is got from the join of several relations, then every join key, except primary keys, should be in the template. |
| Mapping rule | 1 | Map the aggregate functions in a statement to the corresponding set operations in the template. Map nested **Select** statements to nested templates. Map **Exist** clauses to the emptiness judgement of nested templates. Map **Views** to the corresponding SQL statements of the views. |
| Branching rule | 2 | For branching program units, such as **if-else** and **case**, with the standard form **if** $c$ **then** $SS1$ **else** $SS2$, assume the read set templates for $SS1$ and $SS2$ are $RS_1$ and $RS_2$ respectively, then the read set template for the unit is: **if** $c$ **then** $RS_1$ **else** $RS_2$. |
| Loop rule | 2 | Viewing a loop structure as a limited set of program blocks, the read set template of the loop structure is the union of the templates of all its member blocks. |
| Container rule | 3 | For any data item $x$ which is read by transaction $T$, if $x$ can be directly specified by the input of $T$, that is, no database state is needed in the specification, then add $x$ into the template. Otherwise, add the least set of data items which includes $x$ and can be directly specified by the input into the template. The least set of items is called the *container* of $x$. |

Table 1: Rules for Extracting Read Set Templates

In step two, based on the relation between $TP_1$ and $TP_2$ that `o_id=District.(w_id+d_id).D_NEXT_O_ID`, we get the combined template which is specified in the above example.

In situations where tracing through the log for the value of some variable in the template does not justify the corresponding cost, a simpler template materialized from only the input is preferred. This is done in Step three. For this example, the generalized template is:

```
Template =  { District.(w_id+d_id).D_NEXT_O_ID;
              Order-Line.(w_id+d_id+OL_O_ID+OL_NUMBER).OL_I_ID;
              Stock.(w_id+OL_I_ID).S_QUANTITY }
```

Based on the different possible structures of a **Select** or **Update** statement, and the different possible control flows within a transaction program, some rules can be followed in these steps, which are summarized in Table 1.

In this paper, we justify the feasibility of this approach using a practical inventory management database application which handles millions of records. In particular, we investigate how to extract read set templates from TPC-C transaction profiles. TPC-C benchmark simulates a practical inventory management database application. The results of our study, namely the read set templates of TPC-C profiles, are shown in Appendix A. Our study shows that good templates can be got from real world applications such as TPC-C.

## 7.3   Static Repair

Using the read information extracted from transaction profiles, the static repair algorithms proposed in Section 5 can be adapted correspondingly to achieve the goal. In particular, the adapted two pass repair algorithm can work as follows. It should be noticed that the adapted algorithm is based on the assumption that the scanning order of transactions is a serial order, because the write-read dependency is based on the serial order. Fortunately, strict two phase locking, used in most commercial systems, ensures that the commit order is the serial order.

- In pass 1, scan the log from the beginning to the end; for each transaction in **B** add its write items to the set *dirty_set*; for each good transaction keep its write items until the *commit* or *abort* record is scanned; if it commits, and the intersection of its *read set* and the *dirty_set* is not empty, then add its write items to the set *dirty_set* and mark it 'suspect'.

## 7.4   Dynamic Repair

Algorithm 5 can be adapted as follows to support read sets extracted from transaction profiles.

- Although the concurrency control algorithm is the same as Algorithm 4, we need to use a slightly different notion of correct on-the-fly history. In particular,

  - A history is *read-strict* if it is strict, and if whenever $r_j[x] < o_i[x](i \neq j)$, either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$. That is, no data item may be read or overwritten until the transaction that previously read or wrote into it terminates, either by aborting or by committing.

  - History $H$ is a *correct on-the-fly history* if (1) $H$ is serializable and read-strict, and (2) there is no abort records for undo transactions, and (3) for any read operation $r_{T_i}[x]$, the predicate $x \notin dirty\_item\_set$ holds, and (4) for any conflicting undo transaction pair $U_i$ and $U_j$, if $T_i <_H T_j$ then $U_i <_H U_j$, and (5) for any undo operation $w_U[x]$, the predicate $(x \in dirty\_item\_set) \cap (x \in submitted\_item\_set)$ holds.

- The termination detection mechanism is the same as Mechanism 2 except that we maintain the *tmp_undo_list* as follows. Note that here the conditions which are used to detect termination are only adequate, but not necessary. That is, when the conditions are satisfied, the repair terminates; but when the repair terminates, these conditions may not be satisfied.

  - For each in-repair transaction $T$, if $T.Read\_Set \cap dirty\_item\_set \neq \emptyset$, then put $T$ into the list.

  - For each in-repair transaction $T$, if $\exists x \in T.Read\_Set \cap cleaned\_item\_set$ such that $T.Begin.LSN \leq x.LSN$, then put $T$ into the list.

- Undo transactions are built in the same way as Algorithm 3.

- At the Repair Manager,

  - When a commit record $[T_i, commit]$ is scanned, if $T_i.Read\_Set \cap dirty\_item\_set \neq \emptyset$, then we (a) build the undo transaction for $T_i$ and submit it, and (b) put all items of $T_i$ into the *dirty_item_set*. Otherwise, if $\exists x \in T_i.Read\_Set \cap cleaned\_item\_set$ such that $T_i.Begin.LSN \leq x.LSN$, then we (a) build the undo transaction for $T_i$ and submit it, and (b) put all items of $T_i$ into the *dirty_item_set*.

26

## 7.5 Other Methods of Getting Read Sets

As shown in the previous sections, keeping read information in the log can achieve an exact repair, but it may incur substantial performance penalties due to the significant storage and processing cost of maintaining read information. Extracting read sets from transaction profiles cuts the extra cost significantly, but it usually can only achieve a complete repair, and not an exact repair. That is, some non-suspect good transactions may have to be undone.

Maintaining a special purpose graph with transaction dependency information has many attractions: the graph is immediately available for backing out undesirable transactions, the frequency with which read information is put into stable storage can be dynamically adjusted as appropriate, and the graph can be targeted to cover those transactions most likely to be marked undesirable.

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 tuple within one and only one page, so we know that this tuple 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. Although logical logs are not supported by current database systems, logical logging attracts substantial industrial and research interests. In system R, SQL statements are put into the log as logical records; In [18], 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 [9, 33], or in multilevel transaction models [34, 19], it is possible to extract the read information of transaction (subtransaction) $T$ from its compensation log records, where the action of $T$'s compensation transaction is recorded. In these models, compensation may be more appropriate than undo.

# 8 Implementation Issues

### System Design

A prototype of the attack recovery subsystem is now under development. The main data structures (denoted by rectangles) and execution components (denoted by eclipses) are shown in Figure 10. The prototype is implemented on top of an Oracle server and a Windows NT platform. Since Oracle redo log structure is confidential and difficult to handle, we maintain read and write information by ourselves. In particular, we developed a *Proxy* to mediate every user transaction and some specific triggers to log write operations. The proxy does two things: (1) help collect read information; (2) make the system be aware of transaction status. A trigger is associated with each user table to log the write operations on that table. All the write operations are recorded in the `raw log` table. The *Intrusion Detector* uses the trails kept in the `raw log` and some other relevant proofs to identify bad transactions. If a bad transaction is active when being identified, the proxy will abort it. If a bad transaction is already committed when being identified, it's identifier will be put into the `bad transaction list`. Since we want our recovery subsystem to be portable to a variety of commercial DBMSs, we developed a *Log Generator* to produce both a `write log` and a `read log`, with a structure independent of specific DBMS. Our repair algorithms are based on these two logs.

There are two main challenges to develop the subsystem. One is that Oracle triggers cannot capture every read operation. Oracle triggers can capture the read operations on a data item that
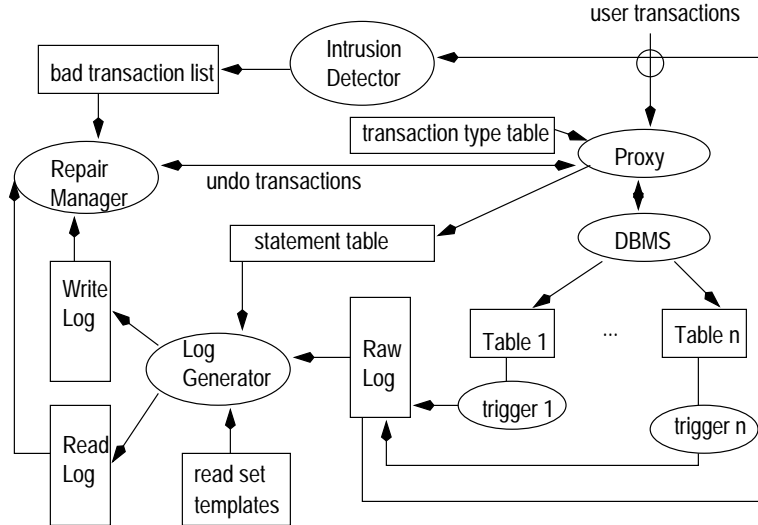
Figure 10: Design of the Attack Recovery Subsystem

is updated or deleted but cannot capture read-only operations. To capture every read operation, we take the approach of extracting read sets from SQL statement texts. To support this, first, we let the proxy identify the type of each in-coming transaction. This can be done by matching the transaction's statements with the transaction patterns kept in the `transaction type table`. Second, we let the proxy record each DML statement and the relevant arguments in the `statement table`. Third, we let the Log Generator use the transaction type information and the arguments to materialize the relevant templates maintained in the `read set templates` table.

The other challenge is how to activate the processes of intrusion detection, log generation, damage assessment, and repair instantly when a relevant event happens. For this purpose, we use a specific Oracle 'alert' mechanism which can send 'alert' messages to a component when a major event happens. We use the 'alert' mechanism in three situations: (1) when a new event is recorded in the `raw log`, an 'alert' is sent to the Intrusion Detector; (2) when a new bad transaction is recorded in the `bad transaction list`, an 'alert' is sent to the Repair Manager; (3) when an user transaction commits, an 'alert' is sent to the Log Generator which will then produce the `write log` and `read log` entries for the transaction.

The design differs from the algorithm presented in Section 7 mainly in two aspects: (1) since we cannot control the Oracle Scheduler, the scheduling algorithms is enforced by the proxy; (2) since we cannot control the Oracle Recovery Manager, the job of the Recovery Manager is now done by the Repair Manager after a 'success' return code is sent back from the proxy about an undo transaction.

## Performance Issues

Performance of the attack recovery subsystem can be measured by two measures: (1) the *average repair time*, which indicates how efficient the subsystem is at repairing damage; (2) the average response time (or throughput) *degradation* of user transactions, which indicates the negative impact of the subsystem on the execution of user transactions.

The average repair time denotes the average time used by the subsystem to clean a damaged data item, which can be measured by the total repair time divided by the number of damaged data items.

28

This measure is mainly affected by (1) the efficiency of the repair algorithms; (2) the access patterns of user transactions, or the dependency among user transactions; and (3) the load of the DBMS. The average response time degradation of user transactions is mainly affected by (1) the execution of undo transactions, and (2) the proxying delay. The impact of undo transactions on this measure is usually small when the number of undo transactions is only a small part of the total number of transactions being executed, or when the lock contention between undo and user transactions is not serious. A bigger impact could instead come from the proxying delay, especially when transactions are short. It is clear that integrating the functionalities of the proxy into the DBMS kernel can dramatically decrease the overhead of attack recovery.

# 9   Conclusion

In this paper, we have identified the problem of attack recovery and developed a family of trusted repair algorithms. Each repair algorithm has two versions: static algorithm and dynamic algorithm. It is shown that the more read information we maintain for repair purpose, the better result we get, although the more cost we may need to pay. And we show that dynamic repair algorithms, compared with static repair algorithms, may back out more good transactions, but they give users more availability and less service delay.

The approach taken in this paper is tailored to the mechanisms used in commercial database systems. It relies on purely syntactic information about the interleaving of read and write operations. In future work, we believe that the incorporation of semantics-based dependency information might result in dramatic reductions in the amount of rework required for recovery.

# A   Read Set Templates of TPC-C Transactions

- The **New-Order** transaction consists of entering a complete order through a single database transaction. The template for this type of transaction is ('+' denotes string concatenation):

| | |
|---|---|
| Input= | warehouse number(w_id), district number(d_id), |
| | customer number(c_id); a set of items(ol_i_id), |
| | supplying warehouses(ol_supply_w_id), and quantities(ol_quantity). |
| Read_Set= | { Warehouse.w_id.W_TAX; |
| | District.(w_id+d_id).(D_TAX, D_NEXT_O_ID); |
| | Customer.(w_id+d_id+c_id).(C_DISCOUNT, C_LAST, C_CREDIT); |
| | Item.ol_i_id.(I_PRICE, I_NAME, I_DATA); |
| | Stock.(ol_supply_w_id+ol_i_id).(S_QUANTITY, S_DIST_xx, S_DATA, |
| | S_YTD, S_ORDER_CNT, S_REMOTE_CNT) } |

- The **Payment** transaction updates the customer's balance, and the payment is reflected in the district's and warehouse's sales statistics, all within a single database transaction. The template for this type of transaction is:

| | |
|---|---|
| Input= | warehouse number(w_id), district number(d_id), |
| | customer number(c_w_id, c_d_id, c_id) or customer last name(c_last), |
| | and payment amount(h_amount) |
| Read_Set= | { Warehouse.w_id.(W_NAME, W_STREET_1, W_STREET_2, W_STATE, W_YTD); |
| | District.(w_id+d_id).(D_NAME, D_STREET_1, D_STREET_2, D_CITY, |

29

D_STATE, D_ZIP, D_YTD);
[ **Case 1,**  the input is customer number:
Customer.(c_w_id+c_d_id+c_id).(C_FIRST, C_LAST,C_STREET_1,
C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_SINCE,
C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE,
C_YTD_PAYMENT, C_PAYMENT_CNT, C_DATA);
**Case 2,**  the input is customer last name:
Customer.(c_w_id+c_d_id+c_last).(C_FIRST, C_LAST,C_STREET_1,
C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_SINCE,
C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE,
C_YTD_PAYMENT, C_PAYMENT_CNT, C_DATA) ]  }

- The **Order-Status** transaction queries the status of a customer's most recent order within a single database transaction. The template for this type of transaction is:

Input=    customer number(w_id+d_id+c_id) or
    customer last name(w_id+d_id+c_last)
Read_Set=  { [ **Case 1,**  the input is customer number:
    Customer.(w_id+d_id+c_id).(C_BALANCE,C_FIRST, C_LAST, C_MIDDLE);
    **Case 2,**  the input is customer last name:
    Customer.(w_id+d_id+c_last).(C_BALANCE,C_FIRST, C_LAST, C_MIDDLE)] ;
    $x$=Order.(w_id+d_id+c_id).O_ID;
    Order.(w_id+d_id+c_id).(O_ENTRY_D, O_CARRIER_ID);
    Order-line.(w_id+d_id+$x$).(OL_I_ID, OL_SUPPLY_W_ID,
        OL_QUANTITY, OL_AMOUNT, OL_DELIVERY_D) }

- The **Delivery** transaction processes ten new (not yet delivered) orders within one or more database transactions. The template for this type of transaction is:

Input=    warehouse number(w_id), district number(d_id),
    and carrier number(o_carrier_id)
Read_Set=  { $R_1$ = New-Order.(w_id+d_id).NO_O_ID;
    $R_2$ = Order.(w_id+d_id+$R_1$).O_C_ID;
    Order.(w_id+d_id+$R_1$).(O_CARRIER_ID,
        OL_DELIVERY_D, OL_AMOUNT);
    Customer.(w_id+d_id+$R_2$).(C_BALANCE, C_DELIVERY_CNT) }

- The **Stock-Level** transaction retrieves the stock level of the last 20 orders of a district. The template for this type of transaction is:

Input=    warehouse number(w_id), district number(d_id),
Read_Set =  { $x$ = District.(w_id+d_id).D_NEXT_O_ID;
    $R_1$ = $\{x-1, ..., x-19, x-20\}$;
    $R_2$ = Order-Line.(w_id+d_id+$R_1$+OL_NUMBER).OL_I_ID;
    Stock.(w_id+$R_2$).S_QUANTITY }

# References

[1] M. R. Adam. Security-Control Methods for Statistical Database: A Comparative Study. *ACM Computing Surveys*, 21(4), 1989.

[2] 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.

[3] 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.

[4] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[5] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Proceedings of the 2000 International Conference on Extending Data Base Technology*, Mar 2000.

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

[7] D. E. Denning. An intrusion-detection model. *IEEE Trans. on Software Engineering*, SE-13:222–232, February 1987.

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

[9] 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.

[10] R. Graubart, L. Schlipper, and C. McCollum. Defending database management systems against information warfare attacks. Technical report, The MITRE Corporation, 1996.

[11] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, Inc., 2 edition, 1993.

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

[13] P. P. Griffiths and B. W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.

[14] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[15] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 474–485, May 1997.

[16] H. S. Javitz and A. Valdes. The sri ides statistical anomaly detector. In *Proceedings IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 1991.

[17] 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.

[18] 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.

[19] 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.

[20] T. F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[21] Teresa Lunt and Catherine McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corporation, McLean, VA, 1998.

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

[23] J. McDermott and D. Goldschlag. Storage jamming. In D. L. Spooner, S. A. Demurjian, and J. E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 365–381. Chapman & Hall, London, 1996.

[24] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.

[25] 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.

[26] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[27] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.

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

[29] C. Pu, G. Kaiser, and N. Hutchinson. Split transactions for open-ended activities. In *Proceedings of the International Conference on Very Large Databases*, August 1988.

[30] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, 1994.

[31] R. Sandhu and F. Chen. The multilevel relational (mlr) data model. *ACM Transactions on Information and Systems Security*, 1(1), 1998.

[32] S.-P. Shieh and V. D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):661–667, 1997.

[33] 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.

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

[35] 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.

[36] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, 1994.

**Paul Ammann** is an Associate Professor in the Department of Information and Software Engineering. He earned an AB degree in Computer Science from Dartmouth College, and MS and PhD degrees in Computer Science from the University of Virginia. His basic research interest might best be described as "Why do things go wrong and what can we do about it?" Current research topics include secure information systems, software testing, semantic-based transaction processing, software for critical systems, and formal methods. Particularly fruitful of late have been cross disciplinary studies of the intersection between formal methods and databases and of the intersection between formal methods and testing. Dr. Ammann has published over fifty research papers in refereed journals and conferences.

**Sushil Jajodia** is BDM Professor and Chairman of the Department of Information and Software Engineering and Director of Center for Secure Information Systems at the George Mason University, Fairfax, Virginia. He joined GMU after serving as the director of the Database and Expert Systems Program at the National Science Foundation. Before that he was the head of the Database and Distributed Systems Section at the Naval Research Laboratory, Washington. He has also been a visiting professor at the University of Milan and University of Rome "La Sapienza", Italy and at the Isaac Newton Institute for Mathematical Sciences, Cambridge University, England.

Dr. Jajodia received his PhD from the University of Oregon, Eugene. His research interests include information security, temporal databases, and replicated databases. He has authored three books including Time Granularities in Databases, Data Mining, and Temporal Reasoning (Springer-Verlag, 2000) and Information Hiding: Steganography and Watermarking - Attacks and Countermeasures (Kluwer, 2001), edited seventeen books, and published more than 250 technical papers in the refereed journals and conference proceedings. He received the 1996 Kristian Beckman award from IFIP TC 11 for his contributions to the discipline of Information Security, and the 2000 Outstanding Research Faculty Award from GMU's School of Information Technology and Engineering.

Dr. Jajodia has served in different capacities for various journals and conferences. He is the founding editor-in-chief of the Journal of Computer Security. He is on the editorial boards of ACM Transactions on Information and Systems Security and International Journal of Cooperative Information Systems. He is the consulting editor of the Kluwer International Series on Advances in Information Security. He also serves as the chair of the IFIP WG 11.5 on Systems Integrity and Control, and on the Board of Directors of the National Colloquium for Information Security Education (NCISSE). He has been named a Golden Core member for his service to the IEEE Computer Society. He is a senior member of the IEEE and a member of IEEE Computer Society and Association for Computing Machinery. The URL for his web page is http://isse.gmu.edu/~csis/faculty/jajodia.html.

**Peng Liu** received his B.S. degree in Information Science and M.S. degree in Computer Science both from the University of Science and Technology of China. He received his Ph.D. degree in Information Technology from George Mason University.

He is an assistant professor of the Department of Information Systems and Director of the Laboratory for Information and System Security at the University of Maryland, Baltimore County. His current research interests are in information security, distributed systems, and databases.

Dr. Liu has served on the program committees of the ACM International Conference on Computer and Communications Security (CCS), and the International Conference on Conceptual Modeling (ER). He is an IEEE member and an ACM member.

The URL for his web page is http://www.research.umbc.edu/∼pliu.