The Design and Implementation of a Self-Healing Database System*

Peng Liu

School of Info Sciences and Technology
Pennsylvania State University
University Park, PA 16802
pliu@ist.psu.edu

Jiwu Jing, Pramote Luenam, Ying Wang
Lunquan Li, Supawadee Ingsriswang

Department of Information Systems
UMBC
Baltimore, MD 21250

Abstract

In this paper, we present the design and implementation of *ITDB*, a self-healing or intrusion-tolerant database prototype system. While traditional secure database systems rely on preventive controls and are very limited in surviving malicious attacks, ITDB can detect intrusions, isolate attacks, contain, assess, and repair the damage caused by intrusions in a timely manner such that sustained, self-stabilized levels of data *integrity* and *availability* can be provided to applications in the face of attacks. ITDB is implemented on top of a COTS DBMS. We have evaluated the cost-effectiveness of ITDB using several micro-benchmarks. Preliminary testing measurements suggest that when the accuracy of intrusion detection is satisfactory, ITDB can effectively locate and repair the damage on-the-fly with reasonable (database) performance penalty.

Keywords: Security, Survivability, Self-Healing Database Systems

1 Introduction

Despite authentication and access control, data contained in a database can be corrupted by authorized insiders due to operation mistakes or malicious intent, or outside attackers who have assumed an insider's identity. Data objects contained in a database are accessed by a transaction for correctness and reliability. A data object is corrupted or damaged if its value is changed from a correct one to a wrong one. Data corruption is a serious security problem in both commercial and military data applications, since corrupted data could lead to wrong decisions and disarsterous actions. Data corruption is not only a data integrity issue, but also a data

*This work is supported by DARPA (OASIS) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-00-2-0575, by DARPA and AFRL, AFMC, USAF, under award number F20602-02-1-0216, and by NSF CCR-TC-0233324.

availability issue. Many critical real world data applications shut down the whole database when they realize that the database could have been seriously corrupted. The execution of new transactions can be resumed only after the database is recovered for sure. In some cases, the real purpose of a data corruption attacker could be denail-of-service to good users.

Besides preventive controls, a *self-healing* or *intrusion tolerant* database system can detect such data corruption attacks (i.e., intrusions), isolate such attacks, contain, assess, and repair the damage caused by such intrusions. This approach contrasts with traditional database security techniques such as authorization [17, 44, 22], inference control [1], multilevel secure databases [52, 46], and multilevel secure transaction processing [4] where the focus is on *preventing* users from gaining access beyond their authorities. Such a preventive approach is very limited to handle successful data corruption attacks, which can seriously jeopardize the integrity and availability of databases. A self-healing database system, on the other hand, arms a prevention-centric secure database system with the ability to *survive* intrusions, hence it evolves a prevention-centric secure database system into an attack resilient database system.

Four factors motivate our work on intrusion tolerant database systems: the expanding cyber attacks, the security limitations of prevention-centric database systems, the pressure brought by the advance of *trusted* database systems and trusted computing, and the opportunity provided by intrusion detection technologies.

Experience with data-intensive applications such as credit card billing, banking, air traffic control, inventory tracking, and online stock trading, has shown that a variety of attacks do succeed to fool traditional database protection mechanisms. A database attack can be enforced at four possible levels: processor (or instruction) level, OS level, DBMS level, and transaction (or application) level. And in general, malicious *insiders* tend to attack at the transaction level and *outsiders* tend to attack at the other three levels. A study performed in [8] shows that most attacks are from insiders. As web-based applications (e.g., e-business) evolve, people have seen (and will see) more and more cyber attacks for a couple of reasons. For example, more critical and valuable information is now processed through the web, which is world-wide accessible. Expanding (successful) cyber attacks pressure applications to not only prevent unauthorized access, but also tolerate intrusions.

Unfortunately, current database security techniques are very limited in tolerating data corruption intrusions, 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. Concurrency control mechanisms cannot distinguish an attacker's transaction 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.

We are glad to see that the recent introduction of *trusted* platforms provides possible solutions to processor level, OS level, and DBMS level intrusion tolerance. Such a platform employs a hardware package containing a processor, memory, and tamper-detecting circuitry [49], or various techniques for software protection [5, 11]. For instance, in [40] a trusted database system built on untrusted storage is proposed where a trusted DBMS running in a trusted processing environment, and a small amount of trusted storage are used to protect a scalable amount of untrusted storage such that every unauthorized read is disabled (by encryption), and every unauthorized data modification can be detected (using hashes, signatures, and tamper-resistant counters) and recovered. These advances pressure database systems to provide transaction-level intrusion tolerance, which is the focus of this work. Note that trusted platforms cannot be used to tolerate transaction-level intrusions, which are instead authorized.

Moreover, the recent advances in intrusion detection (ID) technologies enable intrusion tolerance by providing the ability to effectively identify intrusions [39, 43, 26, 28, 27]. Although our self-healing database system is designed to live with not-so-good intrusion detectors, extremely poor intrusion detection could make a self-healing database system not cost-effective, and shorter detection latency and higher detection accuracy can certainly improve the overall cost-effectiveness of self-healing database systems. These advances make it more feasible than ever to build a cost-effective, self-healing database system on top of existing intrusion detection techniques. (Of course, we expect to see much better intrusion detection techniques in the near future, and such techniques will make our self-healing database systems more feasible and attractive.) For example, DARPA 1998 ID evaluation shows that cutting-edge ID techniques can achieve a detection rate around 70% for misuse detection [29]. Although the effectiveness of existing anomaly detection systems could be seriously compromised by high false alarm rate, we have seen more and more technology breakthroughs in anomaly detection [27, 47], and for some specific applications existing anomaly detection techniques are already good enough to build cost-effective intrusion-tolerant systems. For example, [51] shows that using typical classification rule induction tools such as RIPPER [10], fraud credit card transactions can be identified with a detection rate around 80% and a false alarm rate less than 17%.

In contrast to prevention-centric secure database systems, our objective is to build a self-healing database system - one that can survive attacks. In contrast to trusted database systems, our objective is to build a transaction-level intrusion-tolerant database systems using “off-the-shelf” components. We have designed and implemented ITDB, a prototype intrusion tolerant database system, to investigate this goal. ITDB illustrates intrusion tolerance design principles in four ways: (1) using multiple intrusion tolerance phases to achieve defense-in-depth; (2) using isolation and multi-phase damage containment to live with a not-so-good intrusion detector; (3) on-the-fly intrusion tolerance transparent to applications; (4) using adaptive reconfiguration to achieve self-stabilized levels of data integrity and availability. Although built upon “off-the-shelf” components, ITDB (currently) cannot defend against processor, OS, or DBMS-level attacks, when the lower-level attacks are not so serious and when most attacks are from malicious transactions, ITDB can still be very

effective. Moreover, ITDB can be easily extended to build itself upon a trusted database system such as TDB [40], or to integrate a set of lower-level intrusion-tolerance facilities into itself.

ITDB makes three sets of contributions. First, ITDB synthesizes a number of recent innovations that, taken together, provide a basis for transaction-level intrusion-tolerant database system design. ITDB relies on previous work in areas such as intrusion detection, isolation [33], and trusted recovery [2, 31]. Second, in addition to borrowing techniques developed in other projects, we have refined them to work well in our intrusion tolerant system. For instance, we have extended existing ID techniques to support application-aware, transaction-level ID. We have also developed novel techniques to implement the general isolation algorithm proposed in [33], and the general attack recovery algorithm proposed in [2]. Third, we have developed a set of new intrusion-tolerance techniques, such as the multiphase damage containment technique [32] and the rule-based adaptive intrusion-tolerance techniques [36].

We have built a prototype that demonstrates the key features of ITDB, including transaction proxying, intrusion detection, damage assessment, repair, attack isolation, damage containment, and self-stabilization (or adaptive reconfiguration). We have developed two micro-benchmarks to evaluate quantitatively the cost-effectiveness of ITDB. As Section 10 details, however, some ITDB components are not fully coded, and some components are still not quantitatively evaluated. The results in this paper should thus be viewed as evidence that the ITDB approach is promising, not as "proof" that it will succeed. We present a few preliminary measurements of the prototype. They show that when the effectiveness of the intrusion detector is satisfactory, ITDB can effectively locate and repair the damage on-the-fly with reasonable (database) performance overhead. A comprehensive evaluation of ITDB is now in progress.

The rest of this paper discusses these issues in more detail. Section 2 addresses some related work. Section 3 provides an overview of ITDB. Section 4 outlines ITDB's approach to detect intrusions. Section 5 explains how ITDB assesses and repairs damage. Section 6 explains how ITDB isolates suspicious users. Section 7 outlines ITDB's approach to contain damage. Section 8 outlines ITDB's approach to provide self-stabilized levels of data integrity and availability. Section 9 addresses the security and survivability of ITDB. We describe our prototype in Section 10, including initial testing measurements. Section 11 concludes the paper.

2 Related Work

ITDB builds itself upon several recent and ongoing research efforts to achieve its goal of transaction-level intrusion tolerance. First, ITDB exploits the recent advances in (database) intrusion detection. Second, several ideas of ITDB are motivated by the recent advances in survivability and intrusion tolerance. Finally, ITDB builds upon our previous work on database survivability.

Intrusion detection. One critical step towards intrusion-tolerant database systems is intrusion detection (ID), which has attracted many researchers [13, 37, 21, 18, 39, 43, 38, 26, 28]. The existing methodology of ID can be roughly classed as *anomaly detection*, which is based on *profiles* of normal behaviors [23, 45, 27, 47], and *misuse detection*, which is based on known patterns of attacks, called *signatures* [19, 14, 48]. However, current ID research focuses on identifying attacks on OS and computer networks. Although ITDB needs to apply the existing anomaly detection algorithms, existing ID systems for OS and networks cannot be directly used to detect malicious transactions. Although there are some works on database ID [9, 51], these methods are neither application aware nor at transaction-level, which are the two major design requirements of the ID component of ITDB.

Survivability and intrusion tolerance. The need for intrusion tolerance, or *survivability*, has been recognized by many researchers in such contexts as *information warfare* [15]. Recently, extensive research has been done in general principles of survivability [24], survivable software architectures [50], survivability of networks [42], survivable storage systems [53], etc. These research is helpful for database survivability, but the techniques cannot be directly applied to build intrusion tolerant database systems.

Some research has been done in database survivability. In [3], a fault tolerant approach is taken to survive database attacks where (a) several phases are suggested to be useful, such as attack detection, damage assessment and repair, and fault treatment, but no concrete mechanisms are proposed for these phases; (b) a color scheme for marking damage (and repair) and a notion of integrity suitable for partially damaged databases are used to develop a mechanism by which databases under attack could still be safely used. This scheme can be viewed as a special damage containment mechanism. However, it assumes that each data object has an (accurate) initial damage mark, our approach does not. In fact, our approach focuses on how to automatically mark (and contain) the damage, and how to deal with the negative impact of inaccurate damage marks.

There are also some work on OS-level database survivability. In [41] a technique is proposed to detect *storage jamming*, malicious modification of data, using a set of special *detect objects* which are indistinguishable to the jammer from normal objects. Modification on detect objects indicates a storage jamming attack. In [6], checksums are smartly used to detect data corruption. Similar to trusted database system technologies, both detect objects and checksums can be used to make ITDB more resistant to OS level attacks.

Our previous work. In [2, 31], we have proposed a set of transaction-level trusted recovery algorithms. ITDB has actually implemented the on-the-fly damage assessment and repair algorithm proposed in [2]. In [33], we have proposed a general isolation algorithm. However, this algorithm cannot be directly implemented on top of an "off-the-shelf" DBMS. ITDB has developed a novel SQL rewriting approach to implement the algorithm. Finally, although the design and implementation of almost every key ITDB component have been described in detail in our previous publications [32, 35, 30, 34, 36], this paper is the first paper to (a) present a comprehensive view of ITDB and the fundamental design and implementation issues of intrusion-tolerant database systems, and (b) describe (in detail) how the set of individual ITDB components interact with each

other, and why these components can be integrated into one intrusion-tolerant database system.

3 Transaction Level Intrusion Tolerance

3.1 ITDB Overview

ITDB focuses on the intrusions enforced by authorized but malicious transactions. ITDB views a database as a set of data *objects*. At a moment, the *state* of the database is determined by the values of these objects. The database is accessed by transactions for the ACID properties. A *transaction* is a partial order of *read* and *write* operations that either *commits* or *aborts*. The execution of a transaction usually transforms the database for one state to another. For simplicity, ITDB assumes that every object written by a transaction will be read first, although ITDB can be easily extended to handle *blind writes* in an optimized way. Moreover, ITDB models the (usually concurrent) execution of a set of transactions by a structure called a *history*. ITDB assumes that *strict two-phase locking* (2PL) is used to produce *serializable* histories where the commit order indicates the *serial order* among transactions [7].

ITDB focuses on the *damage* caused by malicious, committed transactions. Since an *active*, malicious transaction will not cause any damage before it commits (due to the *atomicity* property), it is theoretically true that if we can detect every malicious transaction before it commits, then we can roll back the transaction before it causes any damage. However, this "perfect" solution is not practical for two reasons. First, transaction execution is, in general, much quicker than detection, and slowing down transaction execution can cause very serious denial-of-service. For example, Microsoft SQL Server can execute over 1000 (TPC-C) transactions within one second, while the average anomaly *detection latency* is typically in the scale of minutes or seconds. Detection is much slower since (1) in may cases detection needs human intervention; (2) to reduce false alarms, in many cases a sequence of actions should be analyzed. For example, [27] shows that when using system call trails to identify *sendmail* attacks, synthesizing the anomaly scores of a sequence of system calls (longer than 6) can achieve much better accuracy than based on single system calls. Second, it is very difficult, if not impossible, for an anomaly detector to have a 100% detection rate with reasonable false alarm rate and detection latency, since some malicious transactions can (accidentally) behave just like a legitimate transaction.

Hence ITDB is motivated by the following practical goal: "After the database is damaged, locate the damaged part and repair it as soon as possible, so that the database can continue being useful in the face of attacks." In other words, ITDB wants to provide sustained levels of data integrity and availability to applications in the face of attacks. The major components of ITDB are shown in Figure 1(a). Note that all operations of ITDB are on-the-fly without blocking the execution of (most) normal user transactions. The job of the *Intrusion Detector* is to identify malicious transactions. In the rest of this section, we give an overview of the jobs that the other ITDB components do.

The complexity of ITDB is mainly caused by a phenomenon called *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 an object x updated by another transaction 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 object updated by a *good* transaction that is affected by B_i, directly or indirectly. The job of the *Damage Assessor* is to identify every affected good transaction. The job of the *Damage Repairer* is to recover the database from the damage caused on the objects updated by malicious transactions as well as affected good transactions. In particular, when an affected transaction is located, the *Damage Repairer* builds a specific *cleaning* transaction to *clean* each object updated by the transaction (and not cleaned yet). Cleaning an object is simply done by restoring the value of the object to its latest undamaged version. This job gets even more difficult as the execution of new transactions continues because the damage can spread to new transactions and cleaned objects can be re-damaged. Therefore, the main objective of ITDB is to guarantee that damage spreading is (dynamically) controlled in such a way that the database will not be damaged to a degree that is useless.

We believe the single most challenging problem in developing practical, cost-effective self-healing database systems (e.g., ITDB) is that during the detection latency, a tremendous amount of damage spreading can be caused. This is because of the fact that intrusion detection is in general much slower than transaction processing. So when a malicious transaction is detected, a lot of affected transactions may have already been committed. Therefore, a practical, cost-effective self-healing database system must be able to live with a very long detection latency relative to transaction processing.

A unique technical contribution of ITDB is that it can live with long detection latency without suffering serious damage spreading. Allowing long detection latency not only lowers ITDB's requirement on detection agility, but also indirectly lowers ITDB's requirements on detection rate and false alarm rate, because in many cases, longer detection latency can lead to higher detection rate and lower false alarm rate. Nevertheless, it should be noticed that in ITDB, a malicious transaction may not be detected and repaired (since the detection rate is usually not 100%), and a good transaction could be mistakenly identified as a "malicious" transaction and repaired in a wrong way (since the detection rate is usually not 0%), although allowing a longer detection latency usually can reduce the number of such problematic transactions.

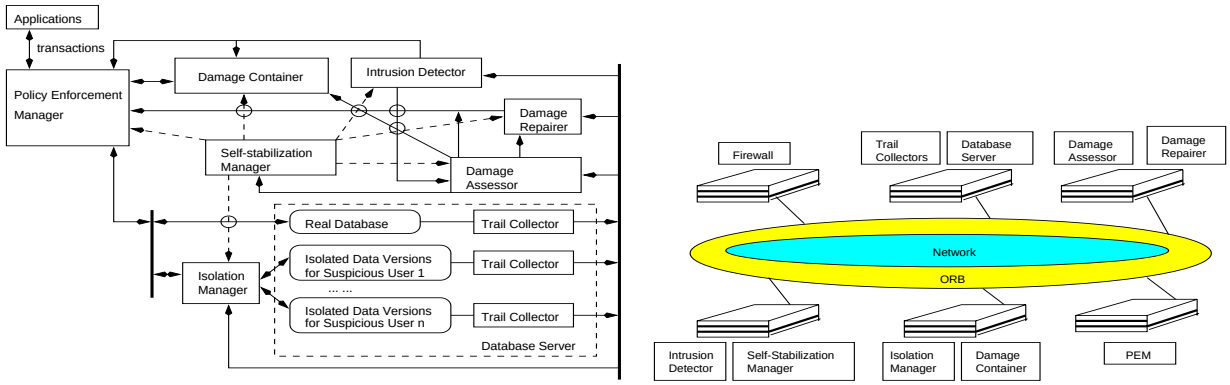
To live with a long detection latency is not an easy task. In ITDB, the impact of detection latency is three folds: (1) during the detection latency, the damage can spread to many objects; (2) a significant *assessment latency* can be caused, and during the assessment latency the damage can further spread to many more objects; (3) significant assessment latency can cause ineffective - to some degree at least - damage containment. These three aspects of impact can cause the database to be too damaged to be useful. The job of the Isolation Manager and the Damage Container is to mitigate this impact.

It is easy to see that if every malicious transaction can be identified just after it commits, then very little damage can spread, and damage assessment could be done quickly. However, with a significant detection latency, when a malicious transaction B_i is identified, in the history there can already be a lot of good transactions following B_i and many of them may have already been affected, directly or indirectly, by B_i. Damage assessment at this situation will certainly cause a significant delay, since as shown in [2], damage assessment itself can spend substantial computation time to scan a long sub-history log for identifying the affected transactions.

Significant assessment latency could cause ineffective damage confinement. At the first glance, it seems that in order to prevent the damage from spreading during damage repair, containing the damage that is located by the Damage Assessor is a good idea. However, in this approach damage will not be contained until an object is reported by the Damage Assessor as damaged. Hence damage containment depends on damage assessment. As a result, when there is a significant latency in locating a damaged object x, during the latency many new transactions may read x and spread the damage on x to the objects updated by them. As a result, when x is confined many other objects may have already been damaged, and the situation can feed upon itself and become worse because as the damage spreads the assessment latency could become even longer. This clearly contradicts with our original motivation of damage containment.

ITDB tackles the three challenges through two novel techniques: attack isolation and multiphase damage containment. Although working towards the same goal, the Isolation Manager and the Damage Container take two very different approaches. And these two approaches compensate each other. In particular, Damage Container takes a novel *multi-phase* damage containment approach which first instantly contains the damage that might have been caused by an intrusion as soon as the intrusion is identified, then tries to uncontain the objects that are previously contained by mistake. Multi-phase damage containment can ensure that no damage will spread during the assessment latency, although with some availability lost. However, Damage Container can do nothing to reduce the damage caused during the detection latency. In contrast, the Isolate Manager can reduce the damage caused during the detection latency (thus it indirectly reduces the damage caused during the assessment latency) by isolating the execution of a *suspicious* transaction that is very likely to cause damage later on. Isolation immunizes the database from the damage caused by the set of suspicious transactions without sacrificing substantial availability, since if an isolated user turns out to be innocent, most - if not all - of his or her updates can be merged back to the real database.

Finally, the job of the *Self-Stabilization Manager* (SSM) is to dynamically reconfigure the other ITDB components based on (a) the current attacks, (b) the current workload, (c) the current system state, and (d) the current defense *behavior* of ITDB, in such a way that stabilized levels of data integrity and availability can be provided to applications in a cost-effective way. We call factors (a), (b), and (c) the *environment* of ITDB. More details of the SSM can be found in Section 8. And the job of the *Policy Enforcement Manager* (PEM) is to (a) proxy user transactions; and (b) enforce system-wide intrusion tolerance policies. For example, a policy



(a) ITDB Architecture (b) One ITDB Installation

Figure 1: ITDB

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 executed by the user.

A real-world ITDB system is shown in Figure 1(b). For load balancing (or efficiency), self-resilience, and reducing the overhead on the database server, ITDB distributes its components across a high-speed switching LAN and uses an ORB to support network-transparent message interactions among these components, and to provide an open interface for (future) integration with third party intrusion-tolerance components. In general, any component can be located at any node except that the Trail Collectors should stay with the Database Server, and the Damage Repairer should stay with the Damage Assessor. Here the firewall is used to hide ITDB from applications and to protect ITDB from outside attacks.

3.2 Key Data Structures

Table 1 contains a summary of the major on-disk data structures used by ITDB to solve the above problems; the data structures are discussed in detail in later sections of the paper. In ITDB, all the data structures are just tables stored at the Database Server.

3.3 System Operation

ITDB is a *concurrent* system triggered by a set of specific *events*. The major events are as follows. (1) When a *user* transaction T is submitted by an application, the PEM will proxy each SQL statement and transaction processing *call* of T, and keep useful information about T and these SQL statements in the Transaction List table and the Statement List table. (2) When a SQL statement is executed, the Trail Collectors will log the corresponding writes in the Write Log, the corresponding reads will instead be extracted from the statement text into the Read Log, and the Intrusion Detector will assess the *suspicion level* of the corresponding

Data Structure	Purpose	Maintained by Who?	Section
Write Log	Logs every write operation	Trail Collectors	4
Read Log	Logs every read operation	The Read Extractor	5
Rule Base	Keeps the rules for ID	The Intrusion Detector	4
Function Table	Registers the functions used in a detection rule	The Intrusion Detector	4
Transaction List	Contains status information about transactions	The PEM	4
Statement List	Records each SQL statement of a transaction	The PEM	4
Transaction Type Table	Helps identify the type of a transaction	The PEM	5
Read and Write Set Templates Table	Helps extract read operations from SQL statement texts	The PEM	5
Isolated Transaction Table	Contains status information about suspicious transactions	The Isolation Manager	6
Time Stamp Columns	Keeps the time stamps for each write operation at the record level	The PEM and the Damage Container	7
Uncontainment Set	Maintains the set of objects that should be uncontained	The Damage Container and the Assessor	7

Table 1: Summary of the major data structures of ITDB

transaction and session using the trails kept in the Write Log (and possibly some other audit tables). (3) When the Intrusion Detector identifies a *suspicious* user, the PEM will notify the Isolation Manager to start isolating the user. (4) When the Intrusion Detector identifies a *malicious* transaction, the Damage Assessor will start to locate the damage caused by the transaction, and the PEM and the Damage Container will start the multi-phase damage containment process (if needed). (5) When the damage caused by a malicious or affected transaction is located, the Damage Repairer will compose and submit a specific *cleaning* transaction to repair the damage. (6) When the Damage Assessor finds an unaffected transaction, when the Damage Container identifies an undamaged object, or when a cleaning transaction commits, some objects will be reported to the PEM to do uncontainment. (7) When the Intrusion Detector finds that a suspicious user is malicious, the Isolation Manager will discard the isolated data versions maintained for the user. (8)When the Intrusion Detector finds that a suspicious user is actually innocent, the isolation manager will merge the work of the user back into the real database by composing and submitting some specific *back-out* and/or *update forwarding* transactions (to the PEM). (9) When the Self-Stabilization Manager receives a report about the environment some *reconfiguration* commands could be generated and sent to some other ITDB components.

4 Intrusion Detection

The ID component of ITDB distinguishes itself from other ID systems in four aspects. (a) It is a database intrusion detection system. (b) It is application aware. (c) It is at transaction level. (d) Instead of developing new ID techniques, ITDB applies existing anomaly detection techniques, and instead focuses on the system support for flexible application-aware transaction-level database ID, and seamless integration of the ID component into the ITDB prototype.

One big difference of database anomaly detection from OS or network anomaly detection is that application semantics are much more critical to the *accuracy* of ID. In many cases, application-semantics-independent metrics for transactions (and users), such as table access statistics, data file access statistics, session statistics, and database schema semantics (enforced by foreign key constraints) are not enough to identify abnormal behaviors, and application (and domain) semantics have to be incorporated and evaluated. For example, when a corrupted accountant raises the salary of a professor by $18,000 instead of $3,000, which is the right number, the application-semantics-independent metrics will find nothing abnormal. However, the amount of salary raise, namely $18,000, is clearly very suspicious for a university system, this kind of anomaly can only be captured when application semantics are used.

Application-aware ID systems are the ones that understand and exploit application semantics. An application-aware ID system should be able to *adapt* to application semantics. For each database application APP_i, an ideal application-aware database ID system should be able to do the following things. (1) Automatically extract (or mine) APP_i's *semantics* from the specification of APP_i, the specification of the database, the knowledge of the Application Manager, and the access history of the database. (2) Automatically compose and deploy a specific set of trail collectors based on the semantics (extracted in Step 1) in such a way that the trails collected can correctly capture the semantics. (3) Automatically use the extracted semantics to guide selecting or composing the right ID algorithm which can achieve the maximum ID effectiveness for APP_i.

ITDB takes the first steps towards application-aware ID, although at this stage, the ITDB's processes of semantics extraction, trail-collector deployment, and ID algorithm selection are still not completely automated. In particular, first, ITDB not only allows the Application Manager to manually specify application semantics using adhoc ID *rules*, but also allows the Application Manager to automatically *mine* semantics-based ID rules from transaction trails using such classification-rule induction tools as RIPPER [10]. Second, ITDB allows the Application Manager to flexibly deploy trail collectors. In particular, ITDB collects transaction trails at three levels (although the current implementation does not handle OS level trails): OS level trails are got from the OS audit file, SQL statement level trails are collected by the set of Trail Collectors and kept in the Write Log, and session level trails are mainly got from the DBMS session audit table. ITDB implements the Trail Collectors using triggers associated with each user table. Triggers can capture every write involved in a Delete, Insert, or Update statement. Based on the application semantics, the Application Manager can determine where to deploy triggers and what to audit by each trigger.

Third, ITDB uses a simple rule-based framework to enable the Application Manager to apply a variety of ID algorithms. In ITDB, an ID rule has four parts: (1) Event, which includes such events as the commit of a transaction; (2) Alarm, which indicates the suspicion level that should be reported if the *condition* is satisfied; (3) Condition, and (4) Priority, which is used to pick the right rule when multiple rules are satisfied. In ITDB, rules are not only used to plug in ID algorithms, but also used to capture application semantics. In order to apply a variety of ID algorithms, ITDB tries to support any kind of conditions. For this purpose, the condition clause of a rule is: $C = P_1 \wedge P_2 \wedge ... \wedge P_m$, and $P_i = function(v_1, v_2, ..., v_n)$. Here P_i is actually a C++ function, and v_j can be any parameter acceptable to the C++ function. In this way, since every existing anomaly technique can be implemented as a couple of C++ functions (given that the needed data are properly prepared), ITDB can support almost every existing ID algorithm. To illustrate, first, ITDB can easily support either adhoc threshold-based ID rules or RIPPER rules. Second, ITDB can support a statistical-profile-based ID algorithm by implementing the algorithm as a couple of condition functions, by storing the profile in a durable table, and by specifying the Alarm field of a rule also as a function.

To support the above rule-based framework, ITDB has a Rule Base, an interface for users to define and register the rules they want to use in addition to the set of default rules, and an interface for users to define and register the functions they need to use in the condition clauses of these user-defined rules. ITDB uses DLL libraries to support online integration of new functions and new rules. No recompilation is needed. We have implemented the ID component and tested it using some adhoc rules, and the results are encouraging [20].

Intrusion detection at the transaction level has several advantages. First, transactions are a good abstraction to model the behavior of users and to collect trails. They enable ITDB to easily synthesize the suspicion levels (or scores) of a sequence of SQL statements, a sequence of transactions, or a sequence of sessions for accuracy of ID. Second, the *atomicity* property of transactions can enable ITDB to prevent the attacks of some malicious long-duration transactions when they can be detect before they commit based on the SQL statements that they have already executed, by simply aborting them.

5 Damage Assessment and Repair

Database recovery mechanisms do not address malicious attacks, 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. Undo and redo operations can only be performed on uncommitted or active transactions, and the durability property ensures that traditional recovery mechanisms never undo committed transactions [7]. Although our approach is related to the notion of *cascading abort* [7], cascading aborts only capture the *read-from* relation between active transactions, and in standard recovery approaches cascading aborts are avoided by requiring transactions to read only committed data [25].

We trace damage spreading by capturing the affecting relationship among transactions. In a history, a

transaction T_i is *dependent upon* another transaction T_j, if there exists an object x such that T_i reads x after T_j updates it, and there is no committed transaction that updates x between the time T_j updates x and T_i reads x. The *dependent upon* relation indicates the path along which damage spreads. In particular, if a transaction which updates an object x is dependent upon a malicious transaction which updates an object y, we say the damage on y *spreads to* x, and we say x is *damaged*. Moreover, a transaction T_u *affects* transaction T_v if the ordered pair (T_v, T_u) is in the transitive closure of the *dependent upon* relation. Since the notation of 'affects' captures both *direct* and *indirect* affecting relationship among transactions, identifying the set of good transactions affected by the set of malicious transactions does the job of damage assessment. To illustrate, consider the following history over $(B_1, G_1, G_2, G_3, G_4)$ where B_1 is malicious and others are innocent. Note that G_2 and G_4 are affected by B_1, but G_1 and G_3 are not. When G_3 commits, $\{y, z\}$ are not damaged, but $\{x, u, v\}$ are. When G_4 commits, z is damaged. Note also that undoing G_4, G_2, are B_1 cleans the database and the work of G_1 and G_3 is saved.

$$H_1 : r_{B_1}[x] r_{B_1}[u] w_{B_1}[x] w_{B_1}[u] c_{B_1} r_{G_1}[z] w_{G_1}[z] r_{G_2}[x] r_{G_2}[v] c_{G_1} w_{G_2}[v]$$
$$r_{G_3}[z] r_{G_3}[y] c_{G_2} w_{G_3}[y] c_{G_3} r_{G_4}[u] r_{G_4}[z] r_{G_4}[v] w_{G_4}[v] w_{G_4}[z] c_{G_4}$$

To identify affected transactions, we need to know which transactions are committed, the serial orders among them, their writes, and their reads. The first three pieces of information can be got from the *log* used by (almost) every "off-the-shelf" DBMS. However, since the DBMS on top of which ITDB is built has a confidential log structure, ITDB uses triggers and the transaction proxy (a part of the PEM) to collect these information. Unfortunately, no "off-the-shelf" DBMS logs reads, and triggers are unable to capture reads either. Making a DBMS capture reads is very expensive. ITDB takes the approach of extracting reads from SQL statement texts. In particular, ITDB assumes each transaction belongs to a transaction *type*, and the *profile* (or source code) for each transaction type is known. For each transaction type ty, ITDB extracts a *read set template* from ty_i's profile. The template specifies the kind of objects that transactions of type ty could read. Later on when a transaction T_i is executed, the template for $type(T_i)$ will be *materialized* to produce the read set of T_i using the input arguments of T_i. We say a read set template *covers* a transaction type if any transaction T of that type, when executed, will have a real read set contained by the materialized read set template. ITDB currently supports only canned transactions. ITDB can be extended to support adhoc SQL statements using an on-the-fly statement analyzer, which is out of the scope of this paper.

To illustrate, consider a simple banking database that has two tables: the Account table that keeps the current balance of each customer account, and the Money_Transaction table that keeps the trails of every deposit or withdraw. The Account table has two fields: Account_ID that is the primary key, and Balance. The Money_Transaction table has four fields: (1) Money_Transaction_ID, the primary key; (2) Account_ID; (3) Amount; (4) Date; and (5) Teller_ID. A deposit transaction can contain the two SQl statements shown in the following. Statement List table. Here Seq_No indicates the execution order of statements.

Trans_ID	SQL_Type	Statement	Seq_No
4.91.6240	INSERT	INSERT INTO Money_Transaction (Money_Transaction_ID, Account_ID, Amount, Date, Teller_ID) VALUES (2833, 1591766, 500, '21-Mar-00', 'Teller-1');	1
4.91.6240	UPDATE	UPDATE Account SET Balance = Balance+500 WHERE Account_ID = 1591766;	2

Assume the read set template of Deposit transactions is shown in the following Read and Write Set Template Table.

Trans_Type	Template_Type	Table_Name	Record_ID	Field
Deposit	Read	Account	AccID	Balance

The template tells us which table or column is read, but does not tell us which records or fields are read. Hence we need materialization. In particular, ITDB gets the input arguments of the transaction also from SQL statements with the help from the Input Argument Extraction table, which is shown as follows.

Trans_Type	Identifier Var	Statement_Pattern	After	Before
Deposit	AccID	'UPDATE'	'Account_ID ='	';'

This table says that for a deposit transaction the value of the Record_ID variable AccID (an input argument) is the number after the string "Account_ID =" and before the char ';' within the SQL statement that starts with "SELECT". Therefore, here the value of AccID is 1591766. Note that since a transaction could execute different sets of statements based on different input arguments and database states, Seq_No sometimes can identify a wrong statement. Then based on the template table, ITDB knows the transaction has read only one record, which can be denoted as Account.1591766, and only one field, which can be denoted as Account.1591766.Balance. Then this read will be recorded in the Read Log. Moreover, ITDB uses the Transaction Type table to help identify the type of transaction 4.91.6240 by pattern matching.

In this paper, we justify the feasibility of this approach using TPC-C benchmark [16] which simulates a practical inventory management database application handling millions of records. The results of our study, namely the read (and write) set templates of TPC-C profiles, are shown in Appendix B. Our study shows that good templates can be got from real world applications such as TPC-C. Our study also shows that for statements with a simple structure, this approach works very well. However, for some complicated SQL statements that contain nested structures, extracting reads is not a easy job and sometimes it may only restore an approximate read set. In [2], some general guidelines are given for extracting reads from nested statements. In order to guarantee that approximate read sets will not affect the correctness of ITDB, we need to ensure that every read set templates covers the corresponding transaction type.

Since ITDB keeps reads and writes in two logs, ITDB needs to synchronize the Write Log and the Read Log in order to correctly identify every affected transaction. Since the commit order kept in the Write Log indicates an equivalent serial order of the history, ITDB needs not to know the exact order between reads and

writes, and approximate synchronization should work. In particular, ITDB scans the Write Log in sequential order (so a lot of searching time can be saved). Only when a commit record is scanned will ITDB retrieve the whole read set of the transaction from the Read Log, and use the reads to determine whether or not the transaction is affected.

After an affected transaction T_i is identified, ITDB repairs the damage caused by T_i using a simple cleaning transaction. Compared with the approach of cleaning each object damaged by T_i separately, a cleaning transaction cleans multiple objects within one transaction.

In contrast with traditional database undo operations which are performed backwardly, ITDB executes cleaning transactions *forwardly* for two reasons: (1) to reduce the repair time; (2) to not block new transactions which could spread damage. Then it is possible that an object cleaned by a previous cleaning transaction is damaged by a later on cleaning transaction if the cleaning transaction of every transaction just restores each object updated by the transaction to the value before the update. To avoid this problem, ITDB, when composing a cleaning transaction, only handles the objects that are damaged and are not under cleaning.

Another key challenge is termination detection. Since as the damaged objects are identified and cleaned new transactions can spread damage if they read a damaged but still unidentified object, so ITDB faces two critical questions. (1) Will the repair process terminate? (2) If the repair process terminates, can we detect the termination? However, as shown in Section 7 later on, the right answer is dependent on the mechanism of damage containment. In particular, if ITDB enforces multi-phase damage containment, then since multi-phase damage containment can guarantee that after the containment phase no damage will spread, so the answers to the above questions are simply “YES”. On the other hand, if ITDB enforces one-phase damage containment, then new transactions can continuously spread damage during damage assessment and repair. In this case, with respect to question 1, when the damage spreading speed is quicker than the repair speed, the repair may never terminate. (In this case, to ensure that the repair will terminate, ITDB can slow down the damage spreading speed by slowing down the execution of new transactions.) Otherwise, the repair process will terminate. With respect to question 2, under the following three conditions we can ensure that the repair terminates: (1) the cleaning transaction for every malicious transaction has been committed; (2) every identified damaged item is cleaned; (3) further scans will not identify any new damage. When conditions 1 and 2 are satisfied, in order to ensure that condition 3 is also satisfied, ITDB reports termination only after the records for every user operation that is executed before any damaged object is cleaned are scanned. Please refer to [2] for more details about termination detection.

Finally, it should be noticed that ITDB can easily handle each newly identified malicious transaction during the repair process of a set of (previously identified) malicious transactions. The details are omitted for space reason.

6 Isolation

The idea of isolation is very simple: redirecting the access of a user when he or she is found suspicious (i.e., very likely to cause damage later on). Later on, if the user is proved malicious, his or her updates can be discarded without harming the database. If the user turns out to be innocent, (usually most of) his or her updates can be merged back to the real database. Isolation immunized the database from the damage caused by suspicious users without sacrificing substantial availability.

Similar to detecting malicious users, ITDB uses a specific *suspicion level* to identify suspicious users. The suspicion level is determined based on several factors, for example, the probability that users reaching this suspicion level will evolve into an intruder, the number of users that need to be isolated, the corresponding isolation cost, etc. This suspicion level is also dynamically reconfigured by the Self-Stabilization Manager. See Section 8 for more details.

There are three major strategies to isolate a suspicious user: in *complete isolation*, no new writes of trustworthy users will be disclosed to the user, and vice versa; in *one-way isolation*, new writes of trustworthy users will be disclosed to the user, but not vice versa; in *partial isolation*, new writes of trustworthy users can be disclosed to the user, but vice versa. ITDB chooses one-way isolation for the merit of more availability and (potentially) less merging lost.

To save resources, ITDB does not use completely replicated databases to do isolation, instead, ITDB maintains extra *data versions* only for the object that are updated (including object creation) by an isolated transaction. In particular, when a suspicious user is isolated, for each (real database) table R_i that the user wants to write, ITDB maintains one extra table with the same structure, called a *suspicious version* of R_i (denoted S_i), only during the isolation period. S_i keeps all and only the writes of the user on R_i. Hence, the size of S_i should be much smaller than R_i in most cases.

Enforcing one-way isolation is not an easy task. When an isolated user issues a SQL statement to access a table R_i, according to one-way isolation, a record r in R_i can be read only if r does not have a version in S_i, and only S_i can be modified. To achieve this, changing the way SQL statements are executed, i.e., using templates, can cause substantial overhead, however, executing the SQL statement solely on S_i can generate invalid results, so the only practical way is to rewrite the SQL statement. ITDB rewrites every Select, Insert, Delete, and Update statement in a slightly different way. ITDB assumes each statement can access multiple tables, and can have nested subqueries. Each rewriting algorithm has three steps: (1) build the virtual database for the isolated user; (2) execute the statement; (3) restore the real database. To illustrate, the rewriting algorithm for Insert statements is shown in Figure 2. Note that the algorithm is with respect to an Oracle Server. Readers can refer to [30] for the other rewriting algorithms. The limitation of SQL statement rewriting is that for some complicated SQL statements the isolated user can suffer a significant delay.

When an isolated user is proved malicious, all the suspicious versions of the user are discarded. When an isolated user is proved innocent, we need to merge the updates of the user back into the real database. Since

Start

if the table (denoted R_i) the SQL statement wants to insert data into
has not a suspicious version (denoted S_i) for the suspicious user

▷ create S_i;

if the SQL statement has no subqueries in the *values_clause*

▷ rewrite the Insert statement in such a way that R_i is replaced by S_i;

▷ forward the rewritten Insert statement to the Oracle Server;

else assume the subquery of the SQL statement accesses tables R_{j1}, ..., R_{jn}

▷ for each R_{jk} that has some records that are deleted from S_{jk}, delete these records from R_{jk} and keeps these records in memory;

▷ for each S_{jk} that has some records that are not in R_{jk}, insert these records into R_{jk};

▷ rewrite the Insert statement in such a way that the string "INSERT INTO table_name" is removed;

▷ rewrite the Select statement generated from the previous step as (sel_{j1}, upd_{j1}, sel_{j2}, upd_{j2}, ..., sel_{jn}, upd_{jn}, $orig\_stat$, res_{j1}, res_{j2}, ..., res_{jn}). Here $orig\_stat$ is the Select statement. sel_{jk} is "SELECT * INTO : k FROM R_{jk} WHERE R_{jk}.primary_key IN (SELECT S_{jk}.primary_key FROM S_{jk});". Here : k is a bind array to tentatively keep all the rows that are selected. upd_{jk} is "UPDATE R_{jk} a SET (all fields) = (SELECT * FROM S_{jk} b WHERE a.primary_key = b.primary_key) WHERE R_{jk}.primary_key IN (SELECT S_{jk}.primary_key FROM S_{jk});". res_{jk} is "UPDATE R_{jk} SET (all fields) = (the corresponding record of : k) WHERE R_{jk}.primary_key IN : k;";

▷ forward the sequence to the Oracle server for execution, assume the result is denoted To_Insert;

▷ delete every record that has been inserted into R_{j1}, ..., R_{jn};

▷ for the records that have been deleted from R_{j1}, ..., R_{jn}, re-insert them into these tables;

▷ for each record in To_Insert, insert it into the corresponding S_{jk};

End

Figure 2: Rewriting Algorithm for Insert Statements

a data object can be independently updated by both a trustworthy transaction and a suspicious transaction, the real database and the isolated database can be inconsistent. In [33], a specific graph, denoted *precedence graph*, which illustrates the serial order that should be among the transactions in the merged history, is used to identify and resolve the inconsistencies. [33] shows that if the precedence graph is acyclic, then the real database and the isolated database are consistent. ITDB uses the same approach. For completeness, the conflict identification and resolution algorithm is summarized in Appendix C.

To build the precedence graph, ITDB needs to know the read and write sets of each transaction, and the read-from relationship between the trustworthy history and the suspicious history, which are denoted as *read edges* in the graph. ITDB gets writes from the Write Log, however, getting reads from the Read Log may not work because for an isolated user the reads are extracted from his or her original statements before rewriting. ITDB double-checks each object in the read set of an isolated transaction, if the object is in a suspicious table S_i, the identifier of the object (in the read set) will be changed. If not, ITDB captures the read edge by searching the write log after the corresponding isolated transaction commits to find the latest trustworthy transaction that updates the object. Note that this search should be done before any new trustworthy transaction is executed.

To resolve the inconsistencies, some transaction may have to be backed out. To reduce the impact of the backed out transactions on other being isolated transactions, ITDB backs out only (once) isolated transactions. The back-out cost is typically very small. For instance, the simulation done in [12] shows that for a database with 1000 objects, when the number of the transactions involved in the merging is 100, the percentage of the transactions backed out is around 5%. When the database becomes bigger, the percentage will usually be even smaller. After the inconsistencies are resolved, ITDB forwards the updates of the isolated user back into the real database using an *update forwarding transaction*. To prevent the updates of new transactions from being overwritten by the update forwarding transaction, ITDB denies the access of new transactions to the tables involved in the merging during the merging period.

7 Damage Containment

Traditional damage containment approaches are *one-phase*. An object x will not be contained until the Damage Assessor identifies it as damaged. However, significant assessment latency can cause the damage on x to spread to many other objects before x is contained. To overcome this limitation, ITDB uses a novel technique called *multi-phase* damage containment as an alternative. This approach has one containing phase, which instantly contains the damage that might have been caused by an intrusion as soon as the intrusion is identified, and one or more later on uncontaining phases, denoted *containment relaxation*, to uncontain the objects that are mistakenly contained during the containing phase.

ITDB models the containing phase as a simple estimation problem. When a malicious transaction B_i is

detected, the containing phase gets the set of objects to contain by 'estimating' which objects in the database may have been damaged. This set is called a *containment set*, denoted S_E. Accurate damage assessment is not possible in this phase, since our approach requires the containing phase be finished in a very short period of time so that the damage can be contained before any new transaction is executed (to possibly spread the damage), and the Damage Assessor usually needs a much longer period of time to do accurate assessment. Assume at this moment the set of objects that are really damaged is S_D, then the relation between S_E and S_D is of four types: (1) $S_E = S_D$ (exact estimation); (2) $S_E \supset S_D$ (over estimation); (3) $S_E \subset S_D$ (not enough estimation); (4) $S_E \cap S_D \neq S_E$, and $S_E \cap S_D \neq S_D$ (approximate estimation). Although some approximate estimation approaches are also investigated, ITDB currently performs *over-estimation* using time stamps. ITDB lets the transaction proxy tag each record with a time stamp, which is kept in the time stamp columns of each table, to time the latest write on the record. To maintain time stamps in a transparent manner, ITDB rewrites SQL statements. For example, an Insert statement "INSERT INTO tablename (...) VALUES (...)" will be rewritten to "INSERT INTO tablename (..., timestamp) VALUES (..., SYSDATE)". When B_i is identified, the containing phase is simply done by denying any new access to the objects that are updated between the time B_i starts and the time this containing control is enforced. The part of the history executed during this period of time is denoted the *contained subhistory*.

ITDB models each later on uncontaining phase as a process to transform one containment set to another. Hence the whole multi-phase containment process can be modeled by a sequence of confinement sets, denoted $S_E, S_2, S_3, ...S_n, S_D$. S_E indicates the result of the containing phase. S_i, $2 \leq i \leq n$, indicates the result of an uncontaining phase. The goal of the uncontaining phases is to converge the sequence to S_D. The sequence can be converged to S_D in many different ways. The way ITDB takes is that $S_E \supseteq S_1$, $S_i \supseteq S_j$ for $i < j$, and $S_n \supseteq S_D$. It should be noticed that the above discussion does not take into account the objects that are cleaned during each uncontaining phase. Typically the objects cleaned during uncontaining phase i should be removed from S_i.

ITDB has three uncontaining phases, which are processed one by one, but can have some overlaps. ITDB uncontains an object by inserting it into the Uncontainment Set table. The PEM will allow access to the objects kept in this set in spite of their time stamps. For this purpose, when a SQl statement s arrives, the PEM will first check s'reads, which are extracted from s on-the-fly, before allowing s to be executed. Uncontaining phase I exploits the dependency relationship among transaction types. A transaction type ty is *dependent upon* ty_j if the intersection of ty_j's write set template and ty_i's read set template is not empty. The rational is that in the corresponding type history of the contained subhistory, if $(type(T_j),\ type(B_i))$ is not in the transitive closure of the dependent upon relation, then T_j is not affected by B_i, and T_j's write set should be uncontained.

However, even if $(type(T_j),\ type(B_i))$ is in the transitive closure of the dependent upon relation, T_j could still be unaffected, since the granularity of an object (type) kept in a template is usually very large. To

uncontain the writes of such T_j, in uncontaining phase II, ITDB materializes the read and write set templates of each transaction in the contained subhistory and uses the materialized read and write sets to identify the transactions that are not affected. Their writes are then uncontained. In uncontaining phase III, the Damage Assessor puts the write sets of the contained transactions that are not affected into the Uncontainment Set table. Note that in general uncontaining phase I is much quicker than phase II, and phase II is much quicker than phase III. ITDB uses multiple uncontaining phases for quicker uncontainment and more availability.

Finally, it should be noticed that after a repair terminates, the time-stamp based containment control should be dismissed, and the Uncontainment Set table should be reset to be empty. Moreover, it should be noticed that when multiple malicious transactions are simultaneously contained, some previously uncontained objects may need to be contained again. ITDB has developed a technique which can guarantee the correctness of simultaneous damage containment. Readers can refer to [34] for more details about this technique.

Multi-phase damage containment causes no damage leakages during the assessment latency. So the work of the Damage Assessor and the Repairer can be bounded within the contained subhistory and be significantly simplified. The drawback is that since some undamaged objects can be mistakenly contained, some availability can be temporarily lost. This indicates that under the situations where damage spreading during the assessment latency is not serious, multi-phase damage containment may be too expensive. Therefore, in these situations ITDB should instead enforce one-phase damage containment.

8 Self-Stabilization

The ITDB components we have presented so far can behave in may different ways. For one example, the Intrusion Detector can use different suspicion levels to raise alarms. For another example, either one-phase or multiphase damage containment could be enforced. At one point of time, the *resilience* of an ITDB system is primarily affected by four factors: (a) the current attacks; (b) the current workload; (c) the current system state; and (d) the current defense *behavior* of the system. It is clear that on the same system state, attack pattern, and workload, two ITDB systems with different behaviors can yield two very different levels of resilience. To achieve the maximum amount of resilience, ITDB systems must *adapt* their behaviors to the *environment* (determined by factors a, b, and c).

The goal of self-stabilization is to adapt ITDB to the environment in such a way that the data integrity and availability levels can be *stabilized*. In particular, ITDB uses four *resilience metrics* to measure the effectiveness of adaptation operations: (1) level of data integrity, denoted LI, which is indicated by the percentage of the damaged data objects (among all the data objects); (2) level of data availability from the perspective of containment, denoted LDA, which is indicated by the percentage of the data objects contained by the Damage Container; (3) level of data availability from the perspective of false alarms, denoted LTA, which is indicated by the percentage of the good transactions that are mistakenly rolled back due to false alarms; (4) level of

data availability from the perspective of isolation, denoted LIA, which is indicated by the percentage of the innocent transactions that are backed-out during the merge processes.

To achieve this goal, the Self-Stabilization Manager (SSM) *monitors* the environment, especially the four resilience metrics, closely; analyzes the reasons for each significant degradation of data integrity or availability levels; enforces agile ITDB *reconfiguration* to *adjust* the behaviors of the ITDB components in a way such that the adjusted system behavior is more (cost) effective than the old system behavior in the changed environment. In addition to the four resilience metrics, ITDB has identified around 20 additional environment parameters such as NAT (Number of Affected Transactions) and TRT (Transaction Response Time).

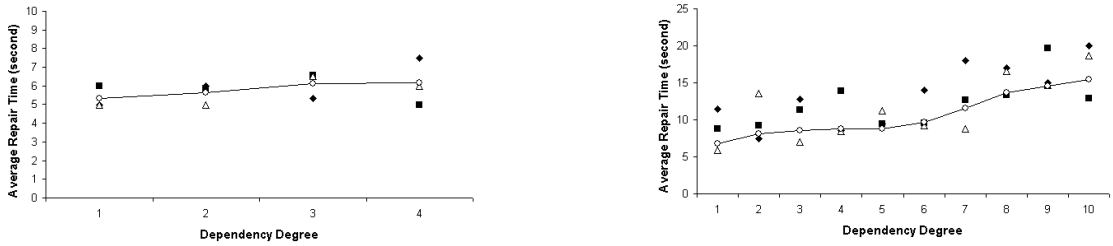
Almost every ITDB component is reconfigurable and the *behavior* of each such component is *controlled* by a set of a *parameters*. For example, the major control parameters for the Intrusion Detector are TH_m, the threshold suspicion-level for reporting malicious transactions, and TH_s, the threshold for suspicious transactions. The major control parameter for the *Damage Container* is the amount of allowed damage *leakage*, denoted DL. When $DL = 0$, multi-phase containment is enforced; when there is no restriction on DL, one-phase containment is enforced. The major control parameter for the PEM is the transaction delay time, denoted DT. When $DT = 0$, transactions are executed in full speed; when DT is not zero, transaction executions are slowed down. At time t, we call the set of control parameters (and the associated values) for an intrusion-tolerance component C_i, the *configuration* (vector) of C_i at time t, and the set of configurations for all the ITDB components, the *configuration* of ITDB at time t. In ITDB, each reconfiguration is done by adjusting the system from one configuration to another configuration.

To do optimal reconfiguration, we want to find the best configuration (vector) for each (new) environment. However, this is very difficult, if not impossible, since the *adaptation space* of ITDB systems contains an exponential number of configurations. To illustrate, the simplest configuration of an ITDB system could be $[TH_m, TH_s, DL, DT]$, then the size of the adaptation space is $domain(TH_m) \times domain(TH_s) \times domain(DL) \times domain(DT)$, which is actually huge. Moreover, we face conflicting reconfiguration criteria, that is, resilience and cost conflict with each other, and integrity and availability conflict with each other. Therefore, we envision the problem of finding the best system configuration under multiple conflicting criteria a NP-hard problem.

ITDB thus focuses on near optimal heuristic adaptation algorithms which can have much less complexity. In particular, the SSM uses normal ECA (Event-Condition-Action) rules to do heuristic adaptations. ITDB cares about such events as the arriving of a report from the Damage Assessor, the arriving of a report about the false alarm rate, and the arriving of a report about the effectiveness of isolation. ITDB cares about such conditions as LI changes and TRT changes. Actions in rules correspond to the reconfiguration operations. For example, a rule can be:

Event: a report from the Damage Assessor arrives;

Condition: the integrity level decreases to 80% & the new transaction submission rate goes



(a) Medium Transaction Size

(b) Large Transaction Size

Figure 3: Average Repair Time

beyond 3 per second & the current containment mechanism is one-phase;
Action - asks the PEM to slow down the execution of new transactions by 20% &
changes one-phase containment to multi-phase.

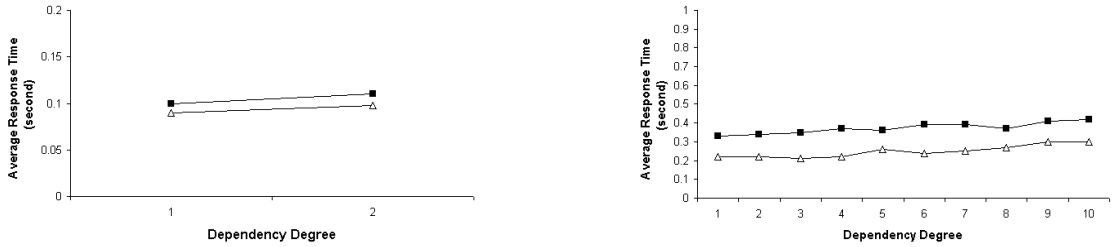
In some cases, cost should be taken into account for better cost-effectiveness. For example, a rule can be:

Event: a report from the Isolation Manager arrives;
Condition: more than 50% of isolated users are innocent &
the average back-out cost is beyond 4;
Action: asks the Intrusion Detector to increase the suspicion level threshold for
suspicious transactions by 25%.

The current design of the SSM is limited in several aspects. First, the approach is responsive but not proactive. Second, the SSM cannot do automatic quantitative, cost-effectiveness-analysis-based reconfiguration. Third, the SSM cannot learn reconfiguration rules by itself. Improving the SSM is one of the focuses of our ongoing research.

9 Security and Survivability of ITDB

ITDB components should certainly be secure and attack resilient by themselves, because otherwise the attacker can attack the database through attacking these components. For example, a faked cleaning transaction (message) can instruct the PEM to roll back the work of a benign, unaffected transaction. To secure ITDB, first, a firewall is used to prevent outsiders from accessing ITDB components. Second, software protection techniques can be used to protect ITDB code [5, 11]. Third, cryptography can be applied to protect ITDB data structures and messages.



(a) Small Transaction Size

(b) Large Transaction Size

Figure 4: Average Response Time

Making ITDB attack (and failure) resilient is more challenging. For this purpose, first, we distribute ITDB components to multiple nodes and try to minimize the dependency of one ITDB component upon another. For example, when the Damage Container is broken, the rest of ITDB can still work well by asking the PEM to do containment only based on the reports from the Damage Assessor, or by asking the PEM to transfer from multi-phase containment to one-phase. However, ITDB has single points of failures, i.e., the Intrusion Detector. Second, distributing the database to multiple sites could help. For example, with respect to trusted recovery, when one site is disabled, the rest of the database may still be able to be continuously repaired (maybe in a degraded manner). Third, some kinds of replication can help. For example, if we have two Intrusion Detectors working in parallel, then when one is down, the other can take over. Currently, none of the above techniques is implemented by ITDB except component distribution.

10 ITDB Prototype

This section describes the state of the ITDB prototype as of September 2002 and presents some preliminary testing results. Although these results are preliminary and although we expect future tuning to substantially improve absolute cost-effectiveness, they suggest that when the accuracy of the intrusion detector is satisfactory, ITDB can effectively locate and repair damage on-the-fly with reasonable performance penalty.

10.1 Prototype Status

The prototype implements all the ITDB components we have described so far. Moreover, we have implemented two database applications to evaluate ITDB: one for credit card transaction management, the other for inventory management (based on TPC-C). However, it should be noticed that the implementations of the Intrusion Detector and the SSM are still preliminary.

10.2 Test Environment

For our testbed, we use Oracle 8.1.6 Server to be the underlying DBMS (Note that ITDB is in general DBMS independent.) The Oracle Server, the Trail Collectors, and the Intrusion Detector are running on a DELL dual Pentium 800MHZ CPU PC with 512MB memory. The PEM, the Isolation Manager, the Damage Container, and the SSM are running on a Gateway PC with a 500MHZ Pentium CPU and 192MB memory. The Damage Assessor and Repairer and the Read Extractor are running on a Gateway PC with a 600MHZ Pentium CPU and 128MB memory. These three PCs run Windows NT 4.0 and are connected by a 10Mbps switched-LAN.

The current version of ITDB has around 30,000 lines of (multi-threaded) C++ code and Oracle PL/SQL code. Each ITDB component is implemented as a set of C++ objects that have a couple of CORBA calling interfaces through which other components can interact with the component and the reconfiguration can be done. We use ORBacus V4.0.3 as the ORB. Finally, ITDB assumes applications use OCI calls, a standard interface for Oracle, to access the database, and ITDB proxies transactions at the OCI call level. The reason that ITDB does not proxy transactions at the TCP/IP or the SQL*NET level, which are more general, is because the exact packet structure of SQL*NET is confidential.

10.3 Testing Results

This section presents a set of preliminary testing results for ITDB. In particular, our testing is three-folds: first, we test the functionality of ITDB using the two database applications we have implemented, and the results show that ITDB meets its design requirements. Second, we evaluate the effectiveness of ITDB using several micro-benchmarks launched by a set of specific transaction simulators. Third, we evaluate the (database) performance overhead of ITDB using another set of micro-benchmarks. Since currently we do not have real database intrusion data, the Intrusion Detector is not evaluated. Instead, we let the Intrusion Detector function as a "simulator" which can detect intrusions with different detection latency.

These testing results are preliminary. We focus on the damage assessment and repair aspect of ITDB and simply assume that (a) one-phase containment is enforced, (a) no user is isolated, and (c) the SSM does no adaptations. Although the impact of isolation, multiphase containment, and self-stabilization remains to be evaluated, we do not expect these additions to significantly impact the results presented here, since the percentage of isolated transactions is usually very small, the PEM enforces multiphase containment in a concurrent manner, and the SSM does not adjust the way of assessment and repair. However, thorough future investigation will be needed to evaluate the merits of the three ITDB components and their impact on the proxying delay, which is a major overhead of ITDB.

Also the current prototype implementation suffers from two inefficiencies, all of which we will tackle in the future. One is that ITDB is currently implemented on top of a commercial DBMS. This hurts performance because the log cannot be exploited and transactions have to be proxyed. To fix this limitation, we plan to

move ITDB into the DBMS kernel, for example, integrating ITDB into the kernel of TDB [40]. The other inefficiency is that we have done little tuning. As we do so, we expect to find and fix inefficiencies. As a result, the absolute performance is much less than we expect for the in-kernel ITDB. The in-kernel ITDB should have a throughput similar to a prevention-centric DBMS.

From the damage assessment and repair perspective, the effectiveness of ITDB is measured by the average repair time of a damaged object. The micro-benchmarks use a database (table) of 10,000 records (each record has 140 bytes), and assume that only one malicious transaction is involved in each repair. Note that our transaction simulators can easily *inject* malicious transactions into the micro-benchmarks. The repair time of a damaged object is defined by the duration between the time when the malicious transaction is identified and the time when the object is cleaned. In this way, the detection latency has no impact on the average repair time.

We simulate damage spreading by having a set of *dependency degrees* among transactions. When transaction T_i is dependent upon transaction T_j, the size of the intersection of T_j's write set and T_i's read set is called the dependency degree of T_i upon T_j. Each micro-benchmark is a sequence of transactions with a specific probabilistic dependency degree (between each pair of adjacent transactions) and a specific transaction size, and each micro-benchmark runs three times using different transactions. Figure 3 shows the impact of transaction size and dependency degree on the average repair time. In addition, for large-size transactions around 15 out of 200 transactions are affected. Here the dots on the lines indicate the average of the measurements of three runs. Here large size transactions have 15 reads and 10 writes, and medium size ones have 8 reads and 4 writes. It is clear that large size transactions are more sensitive to the dependency degree than medium size ones. The main reason is that more statements are needed to repair large size transactions. See that the repair time is reasonable even if the speed to the Database Server is only 10Mbps.

The performance penalty, which is mainly caused by the proxying latency, is evaluated by comparing the throughput or the average response time of a set of transactions when the ITDB is enforced with the throughput of the same set of transactions when ITDB is not enforced. Figure 4 shows that the average response time increases by 11-20% for small size transactions (with 5 reads and 2 writes) and by 37-68% for large size transactions. The main reason is that the micro-benchmarks use one statement for each read or write, thus large size transactions have much more statements to proxy. In real world applications where many reads and writes are usually done within one SQL statement, the response time delay should be much smaller than the results here.

11 Conclusion and Future Work

Self-healing database systems can detect data corruption intrusions, isolate attacks, contain, assess, and repair the damage caused by intrusions, in a timely manner such that the database will not be too damaged to be

useful. This approach has the potential to evolve a prevention-centric secure database system into a defense-in-depth resilient database system. The ITDB prototype demonstrates the viability of building such resilient database systems, and its initial performance results illustrate the potential of this approach.

We believe an important future work is to quantitatively investigate the influence of detection latency and accuracy on the cost-effectiveness of self-healing database systesms. Although we believe that the techniques proposed in this paper can build cost-effective, self-healing database systems under a variety of intrusion detection effectivenesses (in terms of detection latency, detection rate, and false alarm rate), it is very desirable to identify quantitatively the kind of intrusion detectors that can support cost-effective, self-healing data management.

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. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, September 2002.

[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. Aucsmith. Tamper resistant software: an implementation. In *Proc. International Workshop on Information Hiding*, pages 317–333, Cambridge, UK, 1996.

[6] 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.

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

[8] Carter and Katz. Computer Crime: An Emerging Challenge for Law Enforcement. *FBI Law Enforcement Bulletin*, 1(8), December 1996.

[9] C. Y. Chung, M. Gertz, and K. Levitt. Demids: A misuse detection system for database systems. In *14th IFIP WG11.3 Working Conference on Database and Application Security*, 2000.

[10] W. W. Cohan. Learning trees and rules with set-valued features. In *Proc. 13th National Conference on Artificial Intelligence*, 1996.

[11] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 184–196, San Deigo, CA, 1998.

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

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

[14] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

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

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

[17] 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.

[18] P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, 1993.

[19] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1993.

[20] S. Ingsriswang and P. Liu. Aaid: An application aware transaction-level database intrusion detection system. Technical report, Dept. of Information Systems, UMBC, 2001.

[21] R. Jagannathan and T. Lunt. System design document: Next generation intrusion detection expert system (nides). Technical report, SRI International, Menlo Park, California, 1993.

[22] 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.

[23] 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.

[24] J. Knight, K. Sullivan, M. Elder, and C. Wang. Survivability architectures: Issues and approaches. In *Proceedings of the 2000 DARPA Information Survivability Conference & Exposition*, pages 157–171, CA, June 2000.

[25] 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.

[26] T. Lane and C.E. Brodley. Temporal sequence learning and data reduction for anomaly detection. In *Proc. 5th ACM Conference on Computer and Communications Security*, San Francisco, CA, Nov 1998.

[27] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[28] Wenke Lee, Sal Stolfo, and Kui Mok. A data mining framework for building intrusion detection models. In *Proc. 1999 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[29] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *Proc. of 2000 DARPA Information Survivability Conference and Exposition*, Jan 2000.

[30] P. Liu. Dais: A real-time data attack isolation system for commercial database applications. In *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001.

[31] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovery from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[32] P. Liu and S. Jajodia. Multi-phase damage confinement in database systems for intrusion tolerance. In *Proc. 14th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 2001.

[33] P. Liu, S. Jajodia, and C.D. McCollum. Intrusion confinement by isolation in information systems. *Journal of Computer Security*, 8(4):243–279, 2000.

[34] P. Liu and Y. Wang. The design and implementation of a multiphase database damage confinement system. In *Proceedings of the 2002 IFIP WG 11.3 Working Conference on Data and Application Security*, 2002.

[35] P. Luenam and P. Liu. Odam: An on-the-fly damage assessment and repair system for commercial database applications. Technical report, Dept. of Information Systems, UMBC, 2001. This paper is submitted for publication.

[36] P. Luenam and P. Liu. The design of an adaptive intrusion tolerant database system. In *Proc. IEEE Workshop on Intrusion Tolerant Systems*, 2002.

[37] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H. S. Javitz, A. Valdes, P. G. Neumann, and T. D. Garvey. A real time intrusion detection expert system (ides). Technical report, SRI International, Menlo Park, California, 1992.

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

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

[40] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of 4th Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[41] 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.

[42] D. Medhi and D. Tipper. Multi-layered network survivability - models, analysis, architecture, framework and implementation: An overview. In *Proceedings of the 2000 DARPA Information Survivability Conference & Exposition*, pages 173–186, CA, June 2000.

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

[44] 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.

[45] D. Samfat and R. Molva. Idamn: An intrusion detection architecture for mibile networks. *IEEE Journal of Selected Areas in Communications*, 15(7):1373–1380, 1997.

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

[47] S. Sekar, M. Bendre, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[48] 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.

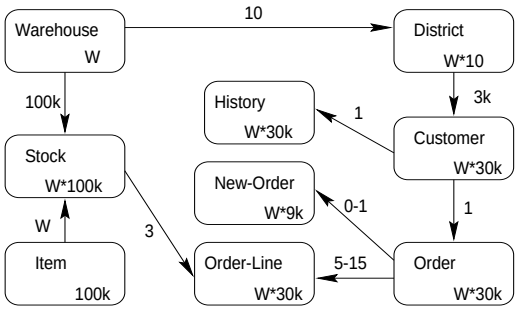


Figure 5: Entity-Relationship Diagram of the TPC-C Database

[49] S. Smith, E. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Proc. International Conference on Financial Cryptography*, Anguilla, British West Indies, 1998.

[50] V. Stavridou. Intrusion tolerant software architectures. In *Proceedings of the 2001 DARPA Information Survivability Conference & Exposition*, CA, June 2001.

[51] S. Stolfo, D. Fan, and W. Lee. Credit card fraud detection using meta-learning: Issues and initial results. In *Proc. AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

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

[53] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote, and P. K. Khosla. Survivable information storage systems. *IEEE Computer*, (8):61–68, August 2000.

A TPC-C Databases and Transactions

The benchmark portrays a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. As the Company's business expands, new warehouse and associated sales districts are created. Each regional warehouse covers 10 districts. Each district serves 3000 customers. All warehouse maintain stocks for the 100,000 items sold by the Company. Customers call the Company to place a new order or request the status of an existing order. Orders are composed of an average of 10 order items.

The database has nine tables whose structures are omitted here(See [16] for the structures of these tables). The entity-relationship diagram of the database is shown in Figure 5 where all numbers shown illustrate the minimal database population requirements. The numbers in the entity blocks represent the cardinality of the tables. The numbers next to the relationship arrows represent the cardinality of the relationships.

B Read and Write Set Templates of TPC-C Transactions

In TPC-C, the term **database transaction** as used in the specification refers to a unit of work on the database with full ACID properties. A **business transaction** is composed of one or more database transactions. In TPC-C a total of five types of business transactions are used to model the processing of an order (See [16] for the source codes of these transactions). The read and write Set templates of these transaction types are as follows.

- 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) }
  Write_Set= { x=District.(w_id+d_id).D_NEXT_O_ID;
  New-Order.(w_id+d_id+x);
  Order.(w_id+d_id+x);
  R_1= { ol_i_id };
  Order-Line.(w_id+d_id+x+R_1) }

- 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 templates for this type of transaction are:

  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, 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) **]** }

Write_Set= { Warehouse.w_id.W_YTD;
District.(w_id+d_id).D_YTD;
[Case 1, the input is customer number:
{ Customer.(c_w_id+c_d_id+c_id).(C_BALANCE,
C_YTD_PAYMENT, C_PAYMENT_CNT);
History.(c_id+c_d_id+c_w_id+d_id+w_id).* }
Case 2, the input is customer last name:
{ Customer.(c_w_id+c_d_id+c_last).(C_BALANCE,
C_YTD_PAYMENT, C_PAYMENT_CNT);
History.(c_d_id+c_w_id+d_id+w_id).* } **]** }

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

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) }

Write_Set= { }

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

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) }

Write_Set= {R_1 = New-Order.(w_id+d_id).NO_O_ID;

R_2 = Order.(w_id+d_id+x).O_C_ID;

Order.(w_id+d_id+R_1).O_CARRIER_ID;

Customer.(w_id+d_id+R_2).(C_BALANCE,

C_DELIVERY_CNT);

New-Order.(w_id+d_id+R_1);

Order-Line.(w_id+d_id+R_1).OL_DELIVERY_D }

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

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 }
Write_Set= { }

C Conflict Identification and Resolution for Isolation

The conflict identification and resolution algorithm is as follows.

- Assume the history of the isolated user is H_s; assume the suffix of the real database history after the user is isolated is H_m
- The precedence graph, denoted $G(H_m, H_s)$, is built as follows
  - Let T_i and T_j be two suspicious transactions or two trustworthy transactions that perform conflicting operations on a data item. There is a directed edge $T_i \rightarrow T_j$ if T_i precedes T_j.
  - If an update of a trustworthy transaction T_g was disclosed to a suspicious transaction T_s during the isolation, then there is a directed edge $T_g \rightarrow T_s$. This type of edge is called a *read edge*. We add read edges to the traditional precedence graph to support one-way isolation.
  - Let T_g be a trustworthy transaction that reads a data item that has been updated by a suspicious transaction T_s, and there is no path from T_g to T_s, then there is a directed edge $T_g \rightarrow T_s$.
  - Let T_s be a suspicious transaction that reads a data item that has been updated by a trustworthy transaction T_g, and there is no path from T_g and T_s that includes a read edge, then there is a directed edge $T_s \rightarrow T_g$.
- If $G(H_m, H_s)$ is acyclic, then the algorithm ends. If $G(H_m, H_s)$ has cycles, then first break all the cycle by backing out some transactions, then end the algorithm. Although it is shown in [12] that just finding the optimal back out strategy is NP-complete, the simulation results of [12] show that in many cases, several back out strategies, in particular *breaking two-cycles optimally*, can achieve good performance.
- For each transaction T_g that is backed out from H_m, locate every active suspicious history which has a read edge from T_g, and for each such read edge, denoted $T_g \rightarrow T_s$, back out T_s and every transaction that is affected by T_s.