Dynamic Damage Recovery for Web Databases*

[1]Hong Zhu (朱虹), [1]Ge Fu (付戈), [2]Kevin Lü
[1]Huazhong University of Science and Technology, 430074, Wuhan, China
[2]Brunel University, Uxbridge, UK UB8 3PH
gefu@smu.edu.sg, zhuhong@public.wh.hb.cn, kevin.lu@brunel.ac.uk

Abstract In web context, a self-healing database system which has the ability to automatically locate and undo the set of transactions that are corrupted by malicious attacks is in urgent need. The metrics of survivability and availability require the database provide continuous services in the period of recovery, which is referred to as dynamic recovery. In this paper, we present a *Fine Grained Transaction Log* to serve for the damage assessment. The log records all the data items of the read only and update-involved operations for the committed transactions, and even extracts data items read by the subqueries in the SQL statements. Based on the log, we propose a *Dynamic Recovery System* to implement the damage repair. The system retains the execution results for blind write transactions and gives a complete solution to the issues of recovery conflicts caused by *Forward Recovery*. Moreover, a confinement activity is imposed on the in-repairing data to prevent a further damage propagation while the data recovery is processing. The integrity measurement and performance evaluation in our experiments show that the system is reliable and high-efficient.

Keywords: Dynamic recovery, survivability, availability.

1. Introduction

In traditional database based systems, applications were run from the users' client. Users' identities were fixed and known in advance for database systems. Therefore, the database system could easily track the users' activities and prevent malicious intrusion behaviors. Thus the sensitive information could be well protected by using traditional security mechanisms such as authentication & authorization, access control etc. With the development of the network techniques, the architecture of the applications based on database is becoming more and more complicated. The three or even multiple layers architectures are widely applied to the enterprise resolution towards data-sensitive applications. Meanwhile, various interfaces are provided for applications to access the database systems. These mechanisms significantly facilitate the

* The work presented in this paper is supported by 863 hi-tech research and development program of China, granted number: 2006AA01Z430.

application design, implementation and maintenance. However, new security risks are consequently brought to web-based database systems. For example, the connection pool techniques which are widely used in the web server make it difficult for database systems to differentiate between transactions of different users. As a result, on the view of security, each data requester is recognized as a "super-user" [Roi07]. It brings tremendous difficulties for the database to verify the identity of the data requester and to track the user's behaviors. Therefore, the inherent authorization & authentication, access control, and intrusion detection mechanisms in DBMSs appear inapplicable in the context of connection pool. Furthermore, the wide range of attack activities such as SQL injection and cross-site-scrip attacks still can penetrate the security-preserve system and cause data corruption and damage propagation to web databases. It is generally recognized that the inherent preventative-based security mechanisms in DBMSs are no longer capable to prevent all malicious intrusion activities. We have to admit that no matter what a rigorous safety protection activity has been taken, there still exists a possibility that sensitive data are exposed to the hackers in the contexts of web applications [Amm97, Kan02, Zho05]. Therefore, the recovery technique towards corrupted data in the post-intrusion phase becomes a dire necessity for web databases.

Inherent data recovery mechanisms in DBMSs focus on the transaction level and system level recovery: transaction level recovery uses keywords "commit" and "rollback" to guarantee logical integrity of transactions while system level recovery undoes and redoes relative transactions between the nearest checkpoint and the crash point. For system level recovery, the DBMSs rollback both malicious and benign transactions, but cannot process user requests during the recovery period. In fact, to some extent, to prevent database systems from providing normal services is the purpose of some attackers [Liu02a]. In the web era, the survivability and availability require the database system to provide a fault tolerant mechanism: when damage occurs, the database system can automatically recover itself and meanwhile, provide continuous, but maybe degraded service while the damage is being repaired. The database system that provides the fault tolerant ability is regarded as intrusion-tolerant, attack-resistant or self-healing database system [Amm02, Lue01, Yum04]. In this paper, we focus on the recovery techniques for malicious transactions in the scenario of web applications. The complexity of the recovery is mainly caused by a phenomenon denoted damage spreading (or damage propagation) [Amm02]: in a database system, the results of a malicious transaction may affect the execution of some other transactions. For example: in a transaction history *H: …T1, …, T2,…*, suppose that transaction *T2* reads the results modified by transaction *T1* (we say transaction *T2* is read-write dependent upon *T1*), after that *T2* writes a data *x* to the database system. Then transaction *T2* is affected by *T1* and data *x* is corrupted. The goal of damage recovery is to locate each affected transaction (damage

assessment) and recover the database from all malicious or affected transactions (data recovery) [Pan98]. At the same time, we should guarantee that the database system is capable to provide continues services in the repair process.

There are mainly two solutions for the damage recovery: 1), *on-the-fly* method. [Amm02] proposed this method firstly. It established a middleware between user applications and the database system, and do damage assessment and data recovery on the top of the database layer. Succeeding work on the functionality measurement and performance evaluation is presented in [Wan00, Wan06, Wan07]. 2), *build-in* approach. This method maintained a *Transaction Dependency Graph* in the execution period, and built the damage tracking, quarantine, and recovery into the kernel of a DBMS [Bai08, Chi05, Chi08, Xie08]. The on-the-fly method maintains a transaction log to track the users' behaviors and repairs the corrupted data in a compensation way. Compared with the *build-in* approach, the characteristic of independence from DBMS for on-the-fly method facilitates the database system to provide damage assessment and damage recovery to any database-relative applications. The defect of the on-the-fly method is that it brings heavier performance overhead. Although the two categories of solutions are capable to provide services of data recovery to web databases, they still have some problems unresolved, described in Section 1.1.

1.1. The Problems Description

Approaches for damage assessment in existing recovery approaches [Amm02, Liu00a, Pil02] are based on read-write dependencies between transactions: suppose a transaction *T* reads a data item which is modified by a malicious transaction before, new damages may be produced while *T* writes some other data items into the database. So the damage assessment regards *T* as an affected transaction and *T* would be undone in the period of recovery [Amm02]. Especially, if *T* executes a statement with subqueries, the data items involved in the subqueries will be read impliedly (in this paper, we denote it *Extended Read*). If a data item involved in the subqueries of the statement was modified by another malicious transaction before, the read operation in the subqueries for *T* would also cause new damages if *T* writes some other data items. Therefore, for each SELECT statement in a transaction, it is indicated that not only the data items involved in the outer SELECT sub-clause but also the data items involved in the nested subqueries should be recorded. In addition, the data items involved in subqueries of UPDATE, DELETE, or INSERT statements should also be recorded. Unfortunately, existing auditing approaches never consider the "implied read" inside a subquery. Therefore, the damage spreading caused by read operations in the subqueries cannot be captured and controlled by damage assessment. (see *Example 1* below).

Example 1*[*]*.

A transaction history *H* consists of malicious transaction *B* and benign transaction *G*: transaction *B*: *UPDATE item SET i_id = 15 WHERE i_cost = 100*; transaction *G*: *SELECT ol_i_id FROM order_line WHERE ol_i_id IN (SELECT i_id FROM items WHERE i_cost =100)* (here, the results of the query are stored in variable *$oliid*); *UPDATE order_line SET ol_qty =200 WHERE ol_i_id = $oliid*. The benign transaction *G* would like to modify the quantities of the products whose price is *100*, and set the quantities to *200* in the table *order_line*. Now before *G* is executed, malicious transaction *B* modified *i_id* of the products whose price is *100*, and then *G* modifies *ol_qty* for relative products. After *G* is executed, damage spreads from *i_id* of table *item* to *ol_qty* of table *order_line*. Existing methods cannot record the data item *i_id* read by the subquery "*SELECT i_id FROM items WHERE i_cost =100*", so the damage spread cannot be captured by damage assessment.

Furthermore, current damage assessment algorithms [Amm02, Liu00a, Pil02, Raj04] do not consider a special kind of transaction: *Blind Write Transaction*. A blind write transaction refers to a transaction whose operations are all update-involved [Amm02]. Just like common transactions, there are also two kinds of blind write transactions: benign and malicious (e.g. a blind write transaction can be regarded as benign if it consists of a batch of update-involved operations used to maintain database by DBA; otherwise, it can also be identified as malicious if it suffers from a SQL injection attack). Because a blind write transaction never reads from other transactions, so it never has read-write dependency relationships with other transactions and thus can not be affected by malicious transactions. On this point, whether a blind write transaction is benign is fully determined by the intrusion detection technology (commonly used intrusion detection methods are listed in [Chi05, Liu00a, Lue01]). In dynamic recovery, the results produced by benign blind write transactions may turn dirty data items clean and thus prevent damage from spreading. For example, suppose some attributes of *i_id* for table *item* are corrupted. Now a benign blind write transaction *T*: *UPDATE item SET i_id = 5* is executed. We can see that benign blind write transaction *T* turn the compromised attributes of *i_id* clean. If another transaction *T'* reads the attribute *i_id* latter, the damage will not propagate any further. Therefore, in the context of dynamic recovery, it is indicated that the results produced by benign blind write transactions should be kept in database persistently. Whereas, existing recovery methods do not consider blind write operations. Consequently, in some cases, the results for a benign blind write transaction are lost.

As we mentioned above, the fault-tolerant database system requires a continuous service while the data recovery is processing. In this paper, we refer this service as dynamic recovery. In dynamic recovery, two problems should be paid special attention to: 1) recovery conflict for the

[*] There are two tables used in the examples in this paper: table *order_line* (attributes of *ol_id*, *ol_i_id*, *ol_qty* are used) and table *item* (attributes of *i_id*, *i_name*, *i_cost* are used). The tables come from *TPC-W Benchmark*.

undo transactions schedule; and 2) data access confinement during recovery period. Problem 1) derives from the Forward Recovery schedule principle [Amm02] for undo transactions. Traditional Backward Recovery [Amm97] waits for the moment when all affected transactions are found by damage assessment and then undoes the affected transactions backward from the last one. In dynamic recovery, we must run the relative undo transaction when a affected transaction is located for consideration of efficiency. So the sequence of undo process has to start from the first affected transaction forward to the end. This specific recovery is denoted Forward Recovery. Forward Recovery may produce inconsistent data and make the recovery results incorrect. Existing methods ignore these problems for forward recovery. We will describe it in Section 3.3. Problem 2) derives from durative data access while recovery is processing. While data recovery is processing, once a damaged data item is read by a new coming user transaction, new damages may be written to the database, and the recovery process must do additional operations to deal with the damage propagation. Existing methods of confinement is cell-grained. Namely, during the recovery period, before a SQL statement of the new coming user transaction is submitted to the database system, the confinement system exacts the data items which the statement attempts to access to check whether these data items are damaged [Amm02]. The confinement system discards requests that attempt to read corrupted data. However, the cell-grained confinement approach has great defects in efficiency. A high-efficient damage confinement algorithm is in urgent need.

1.2. Our Contributions

In this paper, we extended the on-the-fly architecture [Amm02], and established an applied dynamic recovery system. The system captures read-write dependency relationship between transactions which caused by subqueries in the executing period of transactions based on a fine grained transaction log. Moreover, the dynamic recovery system assesses the damage and repairs compromised data using the fine grained transaction log. The contributions of this paper are described as follows:

1) We propose a concept of *Extended Read Operation* and explained how the Extended Read Operations causes damage spreading. Based on the concept, an extended read-write dependency between transactions is added to the recovery model.

2) We propose a *Fine Grained Transaction Log (FGTL)* which recorded data items read or written by transactions, including data items read by subqueries in SQL statements. The FGTL is the data source for damage assessment. Furthermore, a prototype system *FGTL Generator* is implemented to generate the FGTL. The experiment results showed that the *FGTL Generator* is effective.

3) We extend the on-the-fly damage assessment algorithm, and build a component *Damage Assessment Manager* (*DAM*) into the dynamic recovery system. The *DAM* captured the affected transactions compromised by extended read-write dependency relationships between transactions. Moreover, the *DAM* generates compensation transactions to redo the blind write operations so that the results that are created by blind write can be retained.

4) We analyze the conflict issues during the recovery period, and proposed a schedule rule for transactions which were submitted in the damage repair phase. A component *Transaction Scheduler (TS)* is developed to schedule the transactions according to the schedule rule, so that the inconsistent state for dynamic recovery can be avoided.

5) We build a *Data Access Confinement Controller* (*DACC*) into the dynamic recovery system to prevent damage spreading after corrupted data items are identified. A *Two-tier Data Access Confinement Algorithm* was proposed to improve the confinement efficiency. The confinement granularity was for table-level and column-level separately. Experiments results show a great performance improvement while the *DACC* is startup.

1.3. Organization of the Paper

The rest of the paper is organized as follows: Section 2 was mainly devoted to the FGTL. Section 3 proposed the dynamic recovery algorithm based on the FGTL. Section 4 presents the experiments and results and analysis. The related work and conclusions were given separately in Section 5 and Section 6.

2. Fine Grained Transaction Log

As described above, data recovery locates transactions according to read-write dependency relationships between transactions. A log that records transactions' operations must be maintained. A Fine Grained Transaction Log (FGTL) records all the operations in a transaction history with a cell granularity. The FGTL is the main data source for analysis of dependency relationship between transactions. For the structure of transactions can be quite complicated (e.g. nested transactions) and the types of concurrency control mechanisms can be various (Two phase locking, Conflict (serializability, precedence) graph checking, etc.), a transaction execution history can be too complicated for the analyzing process. To simplify the analyzing process, we impose following assumptions to assure the availability for the recovery system.

Assumption 1. *The transactions we process are all flat transactions (a flat transaction is a transaction without containing any sub transactions).*

Assumption 2. *The scheduling for transactions in the database system obeys the two phase locking principle (2PL).*

Assumption 3. *The fine grained transaction log is under protection, and cannot be modified by the users.*

Assumption 1 is proposed to simplify the transaction model for generating the FGTL. In real world application systems, most transactions are flat transactions, so the assumption is available; in dynamic recovery, we have to rearrange the transaction log and put log entries that belong to the same transaction together. Assumption 2 guarantees serializability schedule for transactions. That is to say the final transaction schedule is equivalent to one of the serialized schedules. So we can put log entries which belong to the same transaction together and the post-rearrangement transaction history is equivalent with the pre-rearrangement history. The assumption of 2PL facilitates the damage assessment algorithm. Assumption 3 guarantees the security of transaction history. Namely, the transaction history is trusted.

The transaction log is "fine-grained" means the log has to be recorded in a cell granularity, and the implied read operation in subqueries must be recorded, too. In order to described the semantics of "fine-grained", we propose some definitions as follows.

Definition 1. Write Operation. *Given a SQL statement t, if t is a statement for update-involved operation, such as INSERT, UPDATE, and DELETE, then t is a Write Operation.*

Definition 2. Extended Read Operation. *Given a SQL statement t, if t satisfies one of the following recursive conditions, then t is an Extended Read Operation:*

1) t is a SQL statement SELECT;

2) t is a subquery statement of the WHERE sub-clause in a write operation;

3) t is a subquery statement in an Extended Read Operation.

Definition 3. Association Degree for a SQL statement. *Given a SQL statement t, the Association Degree of t is the nested level of the Extended Read Operations involved in t.*

Definition 1 and 2 describe the operations of FGTL need to consider: write and read operations including subqueries. Formally, a FGTL should record the following information: which transaction, at what time, what type of operation, and which data item the operation operates on. Following is the definition of FGTL.

Definition 4. Fine Grained Transaction Log. *A Fine Grained Transaction Log is a set of log entries and satisfies the following conditions:*

1) a log entry is a six-tuple: [TRASATIONID, OPTYPE, ITEM, BEFOREIMG, AFTERIMG, TIME][*]*, and each log entry is built according to a data item which a transaction extended reads or writes;*

[*] *TRANSATIONID* denotes the transaction that reads or writes the data item. *OPTYPE* is an item of *{SELECT, UPDATE, INSERT, DELETE}*: for Extended Read Operation, *OPTYPE = SELECT*. *ITEM* involves *tablename*, *rownumber*, and *column*. For each user table, we modify the table structure and add an automatic increase field as rownumber to identify one tuple of the table. *BEFOREIMG* represents the value before the data item is modified, and *AFTERIMG* denotes the value after modified. Especially, if *OPTYPE = DELETE*, *AFTERIMG = null*; if *OPTYPE=INSERT*, *BEFOREIMG = null*. *TIME* is the timestamp when the operation on the data item executes.

2) *when a transaction is committed, entries for write and Extended Read Operations involved in the transaction are added to the log.*

The FGTL must be generated in the executing period of transactions. In this paper, we use a system Fine *Grained Transaction Log Generator* (*FGTL Generator*) to create the FGTL. The *FGTL Generator* is a middleware between user clients and backend database system. We introduce it in the following subsections.

2.1. Architecture of the FGTL Generator

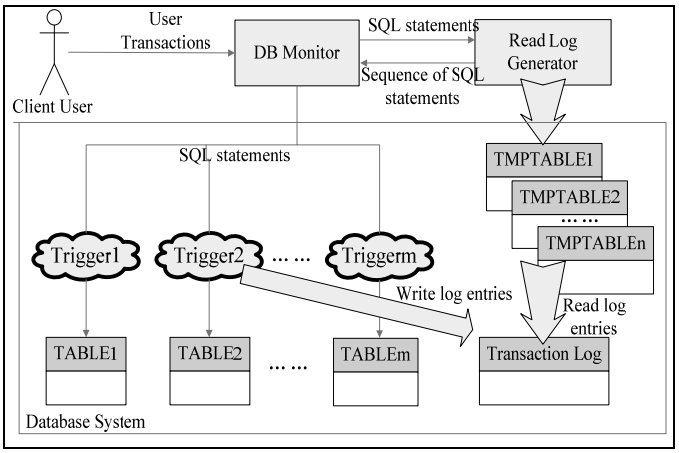


Fig.1. Architecture of *Fine Grained Transaction Log Generator*

As mentioned above, for an operation *op*, the FGTL should record at what time, which data items *op* operates, the before image & after image of these data items, what is the type of the *op*, and the transaction which *op* belongs to. To obtain the FGTL, we have to establish a middleware between client users and backend database systems to record all the SQL statements submitted by user clients. We refer this middleware as the *FGTL Generator*. The architecture of the *FGTL Generator* is illustrated in Fig.1. There are several components in the *FGTL Generator*: *DB Monitor*, *Read Log Generator* and some database-involved components (including *Transaction Log table*, *triggers* and *temporary tables)*. *Transaction Log table* records all log entries for the Extended Read and Write Operations. The *DB Monitor* is an intermediary between *Client User* and backend databases. It is introduced to intercept SQL statements submitted by the *Client user* and response user queries. The tasks for *DB Monitor* are: 1) extract a SQL statement and send it to the *Read Log Generator*; 2) after the statement is processed by the *Read Log Generator*, the *DB Monitor* would receive the statements from *Read Log Generator* and sends them to the database (a single SQL statement may be divided into several statements); 3) if there are return results from the database, the *DB Monitor* would send the results to *Client User*.

The *Read Log Generator* generates read log entries. Based on the *DB Monitor*, the *Read Log Generator* stores data items read by transactions into the *temporary tables* by using a Divide and

Combination Algorithm (the Divide and Combination Algorithm is mainly devoted to audit the behaviors of subqueries. It divides a nested SQL statement to several simple SQL statements according to the nested level and combines them together and submits to the backend database. We will discuss it in Section 2.2 in detail), and then the read log entries are written to the *Transaction Log table*. *Triggers* are used to generate write log entries: for each user table, a trigger is created. If a Write Operation happens on a data item for a user table, the trigger writes a log entry into the *Transaction Log table*. There is a special kind of Write Operation: "self-modification" operation. For example: *UPDATE item SET i_cost = 1.2 * i_cost*; A self-modification operation implicitly reads some data items and modifies them based on their original values. For a self-modification operation, the triggers would generate two categories of log entries: the first category is the SELECT log entries for the data items on the original values; the other is log entries for real Write Operations. The above example SQL statement is divided into a read and a write operations: *SELECT i_cost FROM item*; *UPDATE item SET i_cost = 1.2 * i_cost*; The division policy is consist with the semantics of self-modification operation.

2.2. Logging Extended Read Operations for a SQL Statement

The tasks for the *Read Log Generator* is to log data items involved in all Extended Read Operations of the SQL statements sent from the *DB Monitor*. These statements include SELECT statements (may have subqueries) and UPDATE/ INSERT/ DELETE statements with subqueries. It's always difficult to log the Extend Read Operation which is a subquery. In order to solve this problem, a Divide and Combination Algorithm is proposed. In this algorithm, a SQL statement is divided into several SELECT…INTO… statements to record read data items in temporary tables and then combines the generated statements into a sequence of SQL statements. The generated statements have the same execution results with the initial SQL statement. The process is described as follows: 1) For a SELECT statement whose Association Degree is *n*, *n+1* statements are created: the former *n* statements are SELECT…INTO… statements. They are created according to the order from the inner to the outer subqueries. A SELECT…INTO…statement stores several records in a temporary table. One record represents a data item read by a related subquery. The last SELECT statement is created to return results to the *Client User*. In the whole process, if the outer subquery uses results of inner subquery, it reads from the temporary table which records the intermediate results for the inner subquery; 2) If the statement is an (a) UPDATE/ INSERT/ DELETE statement with subqueries (*Association Degree* is *n*), (*n+1*) statements are created. For each subquery, a SELECT…INTO… statement is created to store read data items in a temporary table using methods in 1), and the *(n+1)th* statement is an (a) UPDATE/ INSERT/ DELETE statement for modification. After all generated SQL statements are executed

successfully in the databases, temporary tables which record all data items read by the initial SQL statement are created. 3) According to data in the temporary tables, log entries are generated and added to the *Transaction Log table*. The Divide and Combination Algorithm is described as follows:

Algorithm 1. *Divide and Combination Algorithm*

Input: a SQL statement (Association Degree = *n*) that is sent from *DB Monitor*.

Output: a sequence of SQL statements that are sent back to *DB Monitor* which logs all read operations for the input SQL statement.

STEP1: if it is a SELECT statement (all subqueries are marked with a *Layer Number*, from the inner to the outer denoted *1, 2, …, n*. Suppose the statement is "*SELECT column1, …, column… FROM…*")

from the most inner subquery to outer (*Layer Number* from *1* to *n*), for each subquery *i*, create a SELECT statement *SELECT … INTO TMPTABLEi….* The temporary table *TMPTABLEi* records the intermediate results for subquery *i* (each row of *TMPTABLEi* denotes one data item read by the subquery, and involves the value of the primary key for the data item. The primary key identifies the rownumber of the data item). If the outer subquery *i+1* uses the results subquery *i* returns, let it read from *TMPTABLEi* directly;

after *n SELECT … INTO…*statements are generated, create a statement *SELECT column1,…, columnn FROM TMPTABLEn…*to return results to the *Client user*;

send (*n+1*) SELECT statements to *DB Monitor*;

STEP2: if it is a Write Operation with subqueries (all subqueries are marked with a *Layer Number*, from the inner to the outer denoted *1,2,…n*)

for subqueries *1* to *n*, create *n SELECT…INTO…* statements using the method for a SELECT statement with subqueries mentioned in STEP 1;

after *n SELECT … INTO…*statements are created, create a Write Operation according to initial SQL statement for modification. If it needs to use results of subquery *n* returns, let it read from *TMPTABLEn* directly;

send (*n+1*) SQL statements to *DB Monitor*;

We give two examples to illustrate the Divide and Combination process.

Example 2. For the statement: *SELECT ol_qty FROM order_line WHERE ol_i_id IN (SELECT i_id FROM item WHERE i_cost >100)*, the Association Degree is 2. According to the "Divide and Combination" algorithm, 3 SELECT statements are generated: 1) *SELECT i_id INTO TMPTABLE1 FROM item WHERE i_cost>100*; 2) *SELECT ol_qty INTO TMPTABLE2 FROM order_line where ol_i_id IN (SELECT i_id from TMPTABLE1*; 3) *SELECT ol_qty from TMPTABLE2*. The three SQL statements are sent to the database orderly.

Example 3. For the statement: *UPDATE order_line SET ol_qty =200 WHERE ol_i_id IN (SELECT i_id FROM items WHERE i_cost >100)*, the Association Degree is 2. According to the "*Divide and Combination*" algorithm, 2 SQL statements are generated: 1) *SELECT i_id INTO TMPTABLE1 FROM item WHERE i_cost > 100*; 2) *UPDATE order_line SET ol_qty = 200 WHERE ol_i_id IN (SELECT i_id from TMPTABLE1)*. The SQL statements are sent to the database orderly.

The correctness of Algorithm 1 is proved in the following.

Lemma 1. *For an input SELECT statement with Association Degree=1, the sequence of SQL statements generated by Algorithm 1 has the same execution results with the input SQL statement.*

Proof. We denote the input SQL statement as *SELECT <TargetList> FROM <TableList> WHERE <Condition>*. Algorithm 1 divides the statement into 2 statements:

1) *SELECT <TargetList_modified> INTO TMPTABLE FROM <TableList> WHERE <Condition>*;

2) *SELECT <TargetList> FROM < TMPTABLE>*.

Here *<TargetList>* denotes the set of target columns for the SQL statement, and *<TargetList_modified>* denotes the set of target columns as well as the column of primary keys. Obviously, $TargetList \subset TargetList\_modified$.

Supposing the execution results set for the input statement is *u_input*, the execution results set for the sequence of 1) and 2) which are generated by Algorithm 1 is *u_division*, and the intermediate results set for *TMPTABLE* in divided statement 1) is *u_TMPTABLE*. In the following we prove that *u_input = u_division.*

According to the relational algebra, we have:

1) $u\_input = \pi_{TargetList}(\delta_{Condition}(TableList))$;

2) $u\_TMPTABLE = \pi_{TargetList\_modified}(\delta_{Condition}(TableList))$; and

3) $u\_division = \pi_{TargetList}(TMPTABLE)$.

From 2) and 3),we conclude that:

$u\_division = \pi_{TargetList}(\pi_{TargetList\_modified}(\delta_{Condition}(TableList)))$.

According to *Algorithm 1*,we have $TargetList \subset TargetList\_modified$, so

$u\_division = \pi_{TargetList}(\delta_{Condition}(TableList)) = u\_input$

□

Theorem 1. *For an input SQL statement with Association Degree=n, the sequence of SQL statements generated by Algorithm 1 has the same execution results with the input SQL statement.*

Proof. 1) if the input statement is a SELECT statement, we denote the statement as:

$SELECT < TargetList_n > FROM < TableList_n > WHERE < Condition_n >$

$SELECT < TargetList_{(n-1)} > FROM < TableList_{(n-1)} > WHERE < Condition_{(n-1)} >$

$\ldots\ldots SELECT < TargetList_1 > FROM < TableList_1 > WHERE < Condition_1 >$;

According to Algorithm 1, the SQL statement is divided into *(n+1)* statements:

(1) *SELECT <TargetList_modified$_1$> INTO TMPTABLE$_1$ FROM <TableList$_1$> WHERE <Condition$_1$>*;

(2) *SELECT <TargetList_modified$_2$> INTO TMPTABLE$_2$ FROM <TMPTABLE$_1$> WHERE <Condition$_2$>*;

…… …… ……

(i) *SELECT <TargetList_modified$_i$> INTO TMPTABLE$_i$ FROM <TMPTABLE$_{(i-1)}$> WHERE <Condition$_i$>*;

…… …… ……

(n) *SELECT <TargetList_modified$_n$> INTO TMPTABLE$_n$ FROM <TMPTABLE$_{(n-1)}$> WHERE <Condition$_n$>*;

(n+1) *SELECT <TargetList$_n$> FROM <TMPTABLE $_{(n-1)}$>*.

The divided SQL statement *(i)* is corresponding to a subquery with the *Layer Number = i*. Some properties of a nested SELECT statement are described as follows:

(1) It could be considered that all subqueries of the statement are executed independently in the order from 1 to n for the *Layer Number*, namely, the *ith* subquery should be executed after the *(i-1)th* subquery;

(2) The subquery with the *Layer Number i* returns results to the outer subquery with the *Layer Number (i+1)* (here, *i=1,2,…, n-1*). The subquery with the *Layer Number n* returns results to the *Client User*.

According to the properties above and Lemma 1, we can conclude that *TMPTABLE$_i$* records the intermediate execution results of the subquery with the *Layer Number i* for the input statement (*i=1,2,…, n*). The *(n+1)th* statement in the sequence of statements generated by Algorithm 1 returns the same execution results with the input SQL statement.

2) if the statement is a INSERT/ UPDATE/ DELETE statement with Association Degree *n*, similarly to 1), we conclude that for each subquery with the *Layer Number i*, the *ith* statement in the sequence of statements generated by Algorithm 1 keep the results in *TMPTABLE$_i$* which involves the intermediate execution results of the subquery with the *Layer Number i* for the input statement (*i=1,2,…, n*), and the same results are affected by the *(n+1)th* statement with the input statement.

From 1) and 2), we can conclude that the sequence of SQL statements generated by Algorithm 1 has the same execution results with the input SQL statement

□

2.3. Fine Grained Transaction Log Generation Algorithm

The Divide and Combination Algorithm generates read log entries for Extended Read Operations. Based on that, an algorithm to generate the FGTL is proposed: The *DB Monitor* captures statements of user transactions. For each SQL statement with Extended Read Operations, the statement is sent to the *Read Log Generator*. The *Read Log Generator* divides the statement into several statements and combines the statements into a sequence of SQL statements, and then sends the sequence back to the *DB Monitor*. The *DB Monitor* submits the sequence of SQL statements to the database. After all statements in the sequence are executed, read log entries based on the temporary tables are written into the *Transaction Log table*. For the Write

Operations, triggers capture the modification on data items and then write log entries are written into the *Transaction Log table*. Therefore, the *Transaction Log table* logs all Extended Read and Write Operations for user transactions. The Fine Grained Transaction Log Generation Algorithm is as follows:

Algorithm 2. *Fine Grained Transaction Log Generation Algorithm*

Input: user transactions which consist of SQL statements.

Output: the Fine Grained Transaction Log.

DB Monitor captures statements of user transactions;

for each statement

if the statement is a Write Operation without subqueries

DB monitor submits it to database;

else

send it to *Read Log Generator*;

using *Algorithm 1*, *Read Log Generator* produces a sequence of SQL statements and sends them back to *DB Monitor*;

DB monitor submits the SQL statements to the database. After the statements are executed successfully, temporary tables which record data items for read operations are created;

based on the temporary tables, store records with fields *TRASATIONID, OPTYPE, ITEM,* and *TIME* in *Transaction Log table*;

the triggers capture the Write Operations (including self modification operations) and records with fields *TRASATIONID*, *OPTYPE*, *ITEM*, *BEFOREIMG*, *AFTERIMG*, and *TIME* are stored in *Transaction Log table* in database;

Correctness of Algorithm 2 is given in Theorem 2.

Theorem 2. *Algorithm 2 logs each write and read operation of user transactions.*

Proof. 1) For the Write Operations in transactions, triggers capture the modifications on data items and write log entries to the *Transaction Log table*. 2) According to Definition 2, the Extended Read Operations involve two categories: subqueries in SELECT statements and subqueries in Write Operations. For a SQL statement with *Association Degree* equals *n*, after the *n* SELECT…INTO… is executed in Algorihm 1, *TMPTABLE1* to *TMPTABLEn* are created which record all data items read by the input statement. So all read operations for the statement are logged. From 1) and 2), we conclude that all write and read operations for user transactions are logged in the *Transaction Log table*

□

3. Dynamic Recovery based on Fine Grained Transaction Log

In this section, a dynamic recovery system is proposed. The system locates the set of affected transactions according to the dependency relationships between transactions; undo the

malicious as well as the affected transactions; and retain the results of other transactions except for malicious or affected transactions.

3.1 Architecture of the Dynamic Recovery System

The architecture of the dynamic recovery system is described in Fig.2. The system includes four parts: *Data Access Confinement Controller* (*DACC*), *Damage Assessment Manager* (*DAM*), *Transaction Scheduler* (*TS*) and *the database system*. As we mentioned in Section 2, the transaction log entry is record as *[TRASATIONID, OPTYPE, ITEM, BEFOREIMG, AFTERIMG, TIME]*. To simplify the notation, in the following sections, we use *w[T, x, v1, v2, TIME]* to denote the Write Operation on data item *x* for transaction *T*, and *r[T, x, v, TIME]* to represent the Extended Read Operation on *x* for transaction *T*.

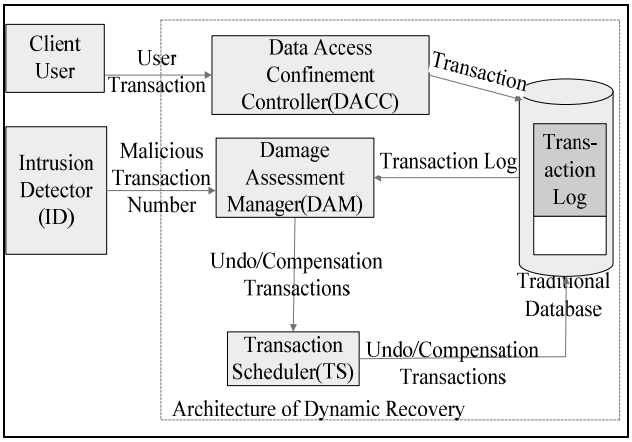


Fig.2. Architecture of the Dynamic Recovery System based on FGTL

To a certain extent, the Dynamic Recovery System is based on the techniques of Intrusion Detection [Amm02, Chi05, Liu00a, Lue01]. Here we only focus on the recovery and neglect the intrusion detection. We assume that *Intrusion Detector* (see Fig.2) provides *Malicious Transaction Number* to the recovery system. A *Malicious Transaction Number* identifies a malicious transaction, and corresponds to the field of *TRANSACTIONID* in the transaction log entries.

The *DAM* is the core component of the recovery system. It is applied to locate every malicious or affected transactions based on the *Transaction Log table*. For every malicious or affected transaction, the *DAM* builds an undo transaction. In order to retain the results produced by a blind write transaction (the blind write transaction is discussed in Section 1.1), the *DAM* also creates a compensation transaction which redoes the blind write operations when the records are overwritten by the undo transaction. The *DAM* submits the undo and compensation transactions to the *TS* and then the *TS* sends the transactions to *the database system*. The *TS* also schedules the undo transaction to prevent conflict while recovery is processing.

The *DACC* is applied to prevent damage spreading after a data item is confirmed dirty. During the recovery period, the *DACC* checks all data requests and discards the transactions which attempt to access dirty data items. A Two-tier Confinement Algorithm is proposed to improve the efficiency of database interaction (we will discuss it in Section 3.4).

3.2. Damage Assessment Manager

In this sub-section, the *DAM* related problems and solutions are introduced. We firstly discuss termination condition in dynamic recovery and introduce some data structure applied in *DAM*. Then we illustrate the methods for retaining the results for Blind Write Transactions and approaches for constructing undo and compensation transactions. Finally we give the damage assessment algorithm and prove its correctness.

3.2.1 Termination condition

A repair process in terms of a set of malicious transactions *B* terminates when all the damage that has already been caused by *B* is cleaned and no damage can be indirectly caused by *B* in the future [Amm02]. According to the criteria, [Amm02] proposed a termination condition. It maintains a *dirty_list* to keep every data item in the "marked dirty" state. A *cleaned_list* is used to keep every data item that has been cleaned. Associating with each item *x* in the *cleaned_list* there is a number *x.CLEANTIME*, which denotes the time when *x* is cleaned. An *undo_list* is used to keep affected transactions. If transaction *T* reads a data item *x* in the *dirty_list*, or *T* has read an item *x* in the *cleaned_list* where *r.TIME* < *x.CLEANTIME*. (here, *r* denotes the read entry *r[T,x,v,TIME]*), *T* is added to *undo_list*. We keep the termination condition and introduce it to our algorithm.

Termination Condition:

DAM scans the log from the entry which is relative to the first transaction in B. The repair process terminates if all of the conditions following are satisfied:

1) every transaction in B has been undone;

2) $dirty\_list = \varnothing$ *;*

3) $undo\_list = \varnothing$ *; and*

4) $\forall x \in clean\_list, x.CLEANTIME < n.TIME$*. Here, n denotes the next entry DAM will scan.*

The repair process terminates when all the data items have been cleaned, and the clean time of each corrupted data is earlier than the log entry that the *DAM* will scan. The correctness of this condition has been proved in [Amm02]. Here we do not introduce the prove process.

3.2.2 Retaining results for blind write transactions

A blind write transaction is a transaction whose operations are all write involved and include no read involved operations. A blind write transaction can be found in the transaction log: A

transaction *T* is a blind write transaction iff ∀ log entry *e* of transaction *T*, *e.OPTYPE* ≠ *SELECT*. Since blind write transactions never cause damage spreading, the results of benign blind write transactions should be retained in the database and never rollback. Suppose that *x* is a data item written by a blind write transaction *T*. In the contexts of dynamic recovery, if any of the following two conditions for *x* is satisfied, the results of blind write transaction *T* for *x* is retained in the database and never affected by the process of damage repair: 1) *x* is never dirty (means never compromised by a malicious or affected transaction) or 2) *x* was once dirty but bas been cleaned, and the clean time is before the moment when *x* is written (namely *x.CLEANTIME* < *w.TIME*, here *w* denotes the log entry writes *x* in transaction *T*). Besides, in some cases, the results for blind write transaction *T* can not be retained in dynamic recovery. To describe the condition that results for a blind write transaction do not be retained, we propose the concept of "Losing Modification Conditions" as follows:

Losing Modification Conditions for a blind write transaction T

1) ∃ log entry w[T,x,v1,v2,TIME] for transaction T, x is in dirty_list; or

2) ∃ log entry w[T,x,v1,v2,TIME] for transaction T, x is in cleaned_list and w.TIME < x.CLEANTIME.

If any of the above conditions is satisfied, we say the results for the blind transaction *T* are lost. *Losing Modification Conditions for a blind write transaction T* could be explained that if *x* is turned dirty, and then *T* blind writes *x* and turns *x* correct. So at this time, *x* is cleaned by the blind write transaction *T* and will not cause damage spreading any further. So *x* should be deleted from the *dirty_list*. However, existing damage assessment treats blind write transactions as normal benign transactions so that the log entries of *T* are skipped by the *DAM*, and *x* will not be deleted from the *dirty_list*. Namely *x* is still treated as a dirty data item, and is recovered to the before image. Therefore, the modifications for the blind write transaction *T* are lost. For example, Fig.3, at time t1 *T'* makes *x* dirty and modifies the value of *x* from *v1* to *v2*; at time t2 log entry *w[T',x,v1,v2,t1]* is scanned by the *DAM*; at time t3 *T* blind writes *x* and modifies the value of *x* from *v2* to *v3*; at time *t4* log entry *w[T,x,v2,v3,t3]* is scanned but skipped by the *DAM* (the log entry is treated as a normal benign operations); at time *t5 x* is cleaned from *v2* to *v1*. So the modification for *T* which modifies *x* from *v2* to *v3* is lost).

In order to retain the results for a blind write transaction which satisfies the Losing Modification Conditions, a compensation transaction is created to redo the blind write transaction. The compensation transaction is just a "copy" of the blind write transaction. After it is created, the compensation transaction is submitted to *Transaction Scheduler*. When the compensation transaction is executed successfully in the database system, the results modified by the blind write transaction can be retained (shown in Fig.3, at time *t6*, a compensation transaction for the blind

write transaction *T* is created which modifies the value of *x* to *v3*). At last, we reset the *CLEANTIME* of corresponding data item *x* operated by blind write transaction. The detailed process is discussed in Section 3.2.4.

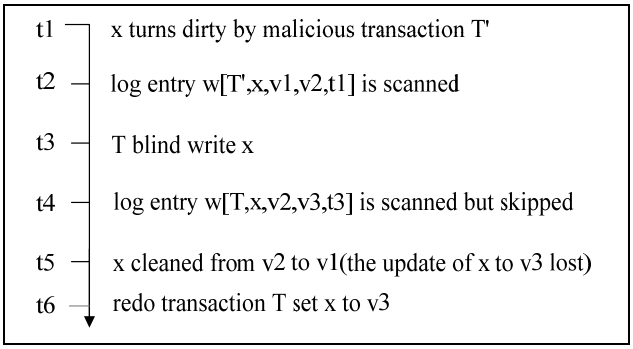


Fig.3. Using a compensation transaction to retain results of a blind write transaction *T*

3.2.3 Constructing undo and compensation transactions

As is mentioned in Section 3.1, the *DAM* constructs undo transactions for the malicious as well as affected user transactions. A typical method for assembling the undo transaction *UTi* for transaction *Ti* is as follows: scan the log backwards from the end, for each Write Operations of *Ti*, create an inverse operation to rollback the relative data item to the before image [Amm97, Amm02]. In this approach, for each Write Operation, a corresponding inverse operation must be constructed, which brings performance degradation. On the point of efficiency, a light-weight set of inverse operations becomes a dire necessity. It implies that the redundant operations should be removed or merged.

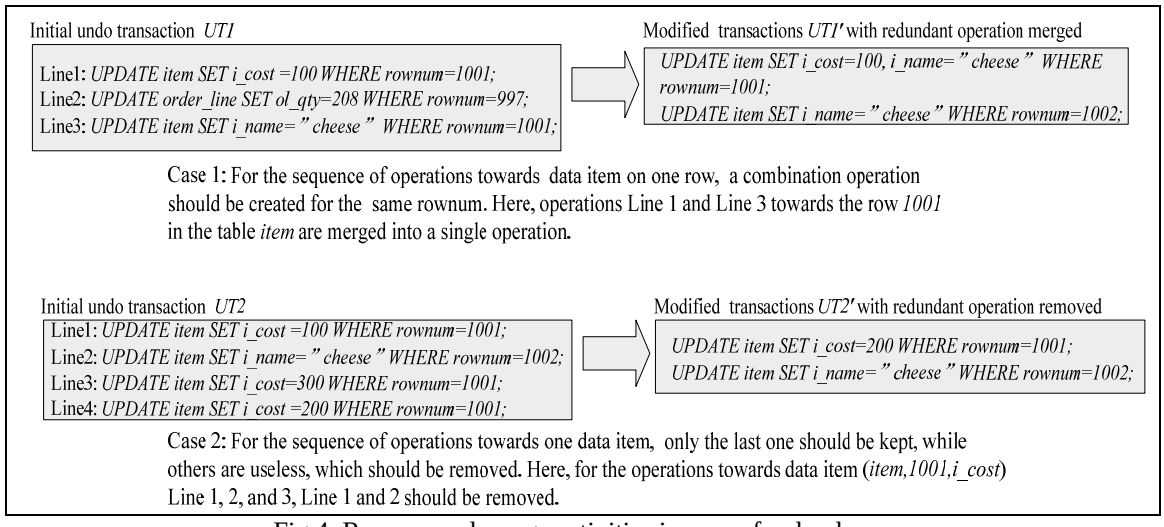


Fig.4. Remove and merge activities in case of redundancy

In common cases, the recovery for one data item correlates one recovery activity, except for the following two situations: 1), ∃ operations *Oi*: *w[T, x, v1, v2]* and *Oj*: *w[T, y, u1, u2]* satisfy that *x.rownumber= y.rownumber* and *Oi.OPTYPE= Oj.OPTYPE*; 2), ∃ operations *Oi*: *w[T, x, v1, v2]*

and *Oj*: *w[T, x, u1, u2]*, and *Oi.OPTYPE= Oj.OPTYPE= UPDATE*. For Situation 1), *Oi* and *Oj* operates on the same row and a combination operation could be generated as a substitution (see Case 1 of Fig.4); for Situation 2), *Oi* and *Oj* modify the same data item, and only the last data version should be retained while the operations on the former versions can be removed (see Case 2 of Fig.4). It has been shown that the remove and merge actions have no impacts on the correctness of the undo transactions, while significantly speed up the recovery process. The Undo Transaction Construction Algorithm is described as follows:

Algorithm 3. *The Undo Transaction Construction Algorithm*

Input: a sequence of log entries for transaction *T*.

Output: undo transaction *UT* for transaction *T* with all redundancy elimination.

Data structure:

1, *u_queue*: Temporal container for the undo operations, one element in *u_queue* corresponds a SQL statement for the undo transaction *UT*, including following fields: *OPTYPE*, *tablename*, *ass_list*, *rownum*. *OPTYPE*$\in$*{INSERT, DELETE, UPDATE}*, *ass_list* is a list involve the fields of *column* and *beforimg*, which separately represent column name and the before image of the data items. *tablename* and *rownum* indicates the tuple the SQL statement operates on. A SQL statement could be assembled according to an element in *u_queue*. For example: for the node *("UPDATE", "item", (<i_cost, 100>,<i_name, "cheese">), 1001)*, the constructed SQL statement is: *UPDATE item SET i_cost=100, iname="cheese" WHERE rownum=1001*.

2, *UT_queue*: a container for SQL statements of the undo transaction *UT*.

Algorithm Description:

three steps are described for the process. In Step 1, the *u_queue* is constructed, which merges operations on the same row into one node of *u_queue*; Step2 is mainly devoted to deal with the redundancy in case2, Fig.4; *UT* is constructed and returned in Step3.

Initialization:

u_queue*:={}, UT_queue:={}.*

STEP1. scan from the end of log entry backwards to the front of transaction *T*

for Write Operation *w[T,x,v1,v2]*

if ($\exists$ node *o* in *u_queue*, *x.rownum= o.rownum && w.OPTYPE==o.OPTYPE*)

add (*x.column*, *x.BEOREIMG*) to *o.ass_list*;

else *u_queue*.Enqueue($\overline{w.OPTYPE}$, *x.tablename, (x.column, x.BEFOREIMG),*

x.rownum); **/*here** $\overline{w.OPTYPE}$ **means the inverse type of the operation for *w* */**

STEP2. scan from the end node backwards to the front for *u_queue*

for node *o* in *u_queue*

if ($\exists$ node *p* in front of *o*, $\exists$ *item_p* in *p.ass_list*, *item_o* in *o.ass_list* such that *p.OPTYPE==o.OPTYPE==UPDATE && p.rownum=o.rownum && item_p.column=item_o.column*)

remove *item_p* from *p.ass_list*;

if (*p.ass_list* is empty)

Dequeue(*p*);

STEP3. assemble a SQL statement for each node in *u_queue* (the operation type takes the inverse value of the field *OPTYPE*), and Enqueue the statement in *UT_queue*;

return *UT_queue*;

The remove and merge method could be applied to the redundancy elimination for the compensation transactions. While some modifications should be done in Step1: 1) while constructing the data structure *u_queue*, the order of the scanning process should be straight forward from the first log entry; 2) The *OPTYPE* is the same value as *w.OPTYPE* (for undo transaction, the *OPTYPE* is the inverse type of *w*).

3.2.4 The Damage Assessment Algorithm

As it is described above, the *Damage Assessment Manager* (*DAM*) is the core component in the whole damage recovery system. Cooperating with the *TS* and *DACC*, the *DAM* analyzes the log kept in *Transaction Log table*. With a malicious transactions set *B* detected by *Intrusion Detector* as input, the *DAM* discovers the affected transactions based on the read-write dependency relationship between transactions using a Damage Assessment Algorithm. Meanwhile, create undo and compensation transactions. Specifically, The *DAM* scans the transaction log entries from the first malicious transaction in *B*. Until Termination Condition is satisfied the recovery process terminates (Termination Condition regards that all damage has been repaired and no further data will be compromised any further). Log analysis is the main task for the *DAM*. The Damage Assessment Algorithm is mainly devoted to three kinds of log entries: 1) a write entry for a transaction in *B*. In this case, the data item for the log entry is dirty and should be added to *dirty_list*; 2) the log entry for a blind write transaction. If Losing Modification Conditions (see Section 3.2.2) are satisfied, the entry would be added to *compensation_Ti*. When the last log entry for the blind write transaction is scanned, a compensation transaction is created to redo the blind write operations which satisfy Losing Modification Conditions; 3) the log entry for a normal transaction. For this kind of log entry, the goal of the algorithm is to locate the affected transactions, and then create undo transactions and to submit them to *Transaction Scheduler*. The details of the algorithm are described as follows.

Algorithm 4. *The Damage Assessment Algorithm*

Input: the *FGTL*, the set *B* of malicious transactions.

Output: undo and compensation transactions which recover database to the correct state.

Data Structure:

Besides the data structures introduced in Section 3.2.1, we maintain another two data structures: *tmp_item_list* and *compensation_Ti*. The *tmp_item_list* temporarily records data items written by transactions, and is used to generate the undo transactions. The records in *tmp_item_list* include *ITEM* and *BEFOREIMG* which are collected from the log entry of the transaction log. The *compensation_Ti* records the log entries which satisfy the Losing Modification Conditions

for transaction *Ti*.

Initialization:

undo_list :={}, cleaned_list :={}, dirty_list :={}, tmp_item_list :={}.

STEP1. locate the log entry where the first malicious transaction starts;

STEP2. while (Termination Condition does not hold)

Scan next entry;

if (the entry is for a transaction *Ti* in *B* **/*entry for malicious transactions*/**

add *x* to *dirty_list*;

if (the entry is the last entry for transaction *Ti*)

build an undo transaction *UTi* for *Ti* using Algorithm3. Submit *UTi* to the *TS*;

elseif (the entry *w[Ti,x,v1,v2,TIME]* is for a blind write transaction)

/*entry for blind write transactions*/

if (Losing Modification Condition is satisfied)

add *w* to *compensation_Ti*;

elseif ((*x* in *cleaned_list* and *w.TIME>x.CLEANTIME*) or *x* is never dirty)

skip the entry;

if (*w* is the last entry for transaction *Ti*)

create the compensation transaction *CTi* for *T*, in the method discussed in Section 3.2.3, with *compensation_Ti* as input. Submit *CTi* to the *TS*;

else **/*entry for normal transactions*/**

case (the entry is a write record *w[Ti,x,v1,v2,TIME]*)

if (*x* is not in *cleaned_list*)

add *x* to the *tmp_item_list*;

case (the entry is a record *r[Ti,x,v,TIME]* for Extend Read Operation)

/* capture the Extended Read-Write Dependency */

if (*x* is in *dirty_list* or (*x* is in *cleaned_list* and $r.TIME \leq x.CLEANTIME$))

add *Ti* to the *undo_list*;

case the entry is the last entry for transaction *Ti*

if (*Ti* is in *undo_list*)

create *UTi* for *Ti* using Algorithm3. Submit *UTi* to the *TS*;

move all records of *Ti* from the *tmp_item_list* to *dirty_list*;

else

delete all records from *tmp_item_list*;

STEP3. exit;

At the same time:

1) if undo transaction *UTi* finishes, delete data items which are repaired by *UTi* from *dirty_list*, and write the data item and its *CLEANTIME* to *cleaned_list*; Record the timestamp *CLEANTIME_Ti* when *UTi* finishes;

2) if compensation transaction *CTi* finishes, modify *CLEANTIME* for each data item which has

been modified by *CTi* in *cleaned_list*. Record the the timestamp *CLEANTIME_Tj* when *CTj* finishes.

The compensation transaction method for blind write is first proposed in this paper, and here we point out that the method retains the results of blind write transactions as follows:

Theorem 3. *Algorithm 4 retains modifications for blind write transactions which satisfy Losing Modification Condition.*

Proof. Assume that the case of losing modification of *v3* on *x* in Fig.3 occurs. It is indicated that the entry of blind write *w[T,x,v2,v3,t3]* must be skipped in /*entry for blind write transactions*/ of STEP2. In this case, the condition "*x in cleaned_list and w.TIME > x.CLEANTIME) or x is never dirty*" must hold. But in fact, Fig.3 shows that x is not "never dirty" since malicious transaction *T'* operates on *x*, and *w.TIME < x.CLEANTIME*. So a contradiction is inferred. The assumption does not hold

□

3.3. Transaction Scheduler

There are four states for the undo transactions or compensation transactions (shown in Fig.5). The *DAM* creates an undo transactions or a compensation transaction (denoted as *T*) and submitted it to the *TS*, and then the state of *T*' is changed into "submitted". When *T* is scheduled by DBMS, *T*'s state is transferred into "scheduled". After *T* is executed successfully, the state becomes "finished". *TS* would guarantee that the undo and compensation transactions are scheduled by DBMS according to the sequence that they are created, so that a correct transaction log is generated. We give the schedule rule as follows.

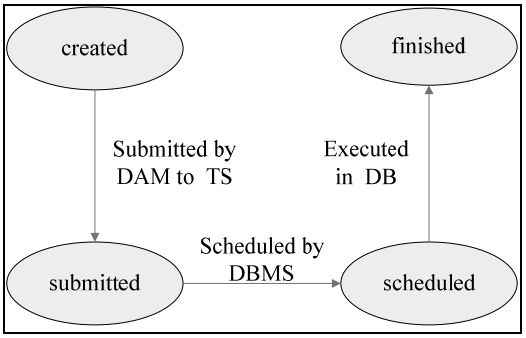


Fig.5. State transition of undo/compensation transactions

Schedule Rule for TS:

for ∀ conflict undo/ compensation transactions Ti and Tj (suppose Ti is the undo/ compensation transaction of Si, and Tj is the undo/ compensation transaction of Sj), if Si and Sj satisfy the condition that $Si <_H Sj$, *then* $Ti <_H Tj$ *(For two transactions Qi, Qj, if* $Qi <_H Qj$, *then Qi is logged before Qj in the transaction history. Namely, Qi is scheduled before Qj by DBMS).*

The *TS* schedules the transactions strictly according to the Schedule Rule to make sure an undo transaction is scheduled according to the executing order of their original transactions from which they are derived. Whereas, the rule cannot guarantee the database be recovered to a consistent state in the context of dynamic recovery: The *TS* schedules the undo transactions in the order of the timestamp when they are submitted (first come, first service), rather than schedules from the end transaction backwards when all undo transactions are submitted. This process is referred as Forward Recovery, which is a specific schedule policy for dynamic recovery [Amm02]. If no extra restrictions are imposed on the Forward Recovery, errors may arise when two conflict undo transactions are scheduled.

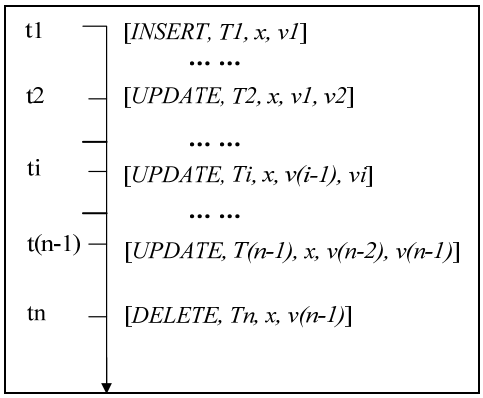
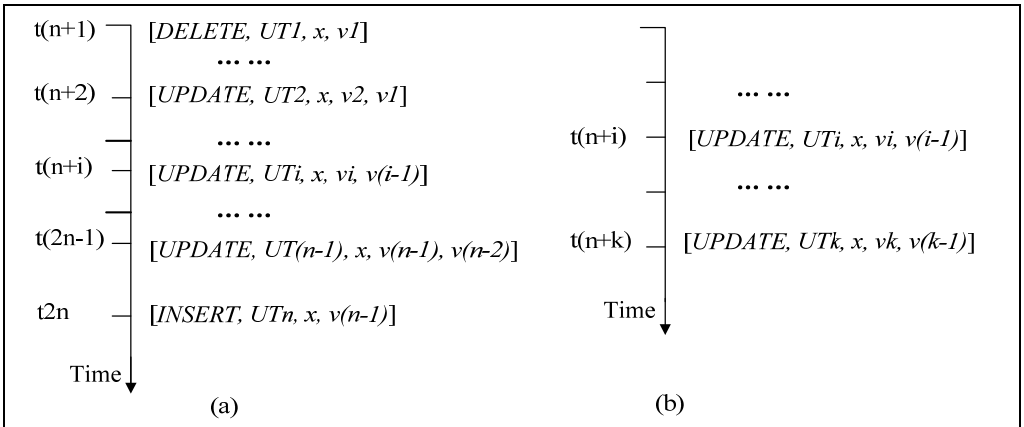


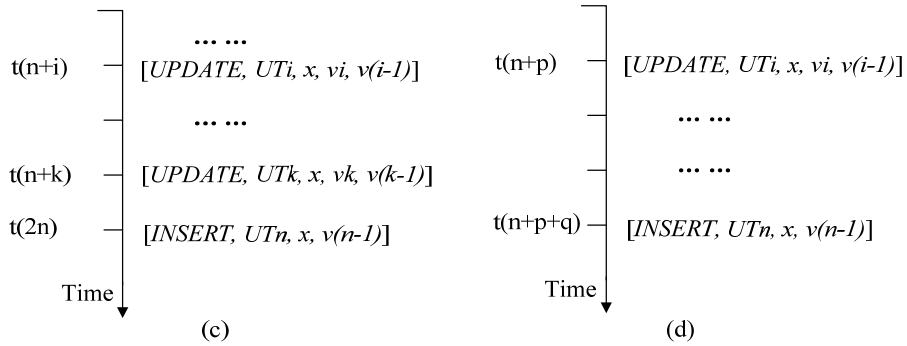
Fig.6. The whole lifecycle for data item *x*

To describe the errors clearly, we start from analyzing the lifecycle for a single data item. The lifecycle for a single data item *x* can be divided with three key activities: create, modify, and release, which are individually correlated to the operations INSERT, UPDATE, and DELETE. The whole lifecycle for a data item *x* is shown in Fig.6.

In Forward Recovery, Inconsistent state may be caused by the execution of undo conflict transactions on x. We summarize three cases in Fig.7. In the lifecycle of a single data item x: 1) in case that x is created by a malicious or an affected transaction, the following conflict undo transactions would not be scheduled, otherwise, inconsistent state may arise for data item *x* (shown in Fig.7(a)); 2) In case that conflict UPDATE operations on *x* are malicious or affected, the first UPDATE operation is scheduled, while the following undo operations of UPDATE should be skipped, or else, data item *x* would be rewritten to an inconsistent value (shown in Fig.7(b)); 3) last case, if UPDATE as well as DELETE operations make *x* dirty, The undo operation for the first UPDATE statement should be scheduled, and the schedule should be chosen for the time slot when the undo operation for DELETE has finished. Otherwise, the undo operation for UPDATE would write a “NULL item”, which causes an error for the recovery process (shown in Fig.7(c),(d)).



Case 1: If the INSERT operation is malicious or affected, only the INSERT operation should be undone, while all the following conflict undo transactions on *x* should not be scheduled.
e.g. In Fig.6, Assume that transactions *T1* to *Tn* are malicious or affected transactions, only *T1* needs to be undone, while transactions *UT2* to *UTn* should be unscheduled. Otherwise, transaction *UT2* to *UT(n-1)* operates on a "NULL item" and returns a message that 0 rows affected; transaction *UTn* rewrites *x* to an inconsistent value *v(n-1)* (shown in (a)).



Case 2: If only the UPDATE operations are malicious or affected, only the first UPDATE operation should be undone, while all the following conflict undo transactions on *x* should not be scheduled.
e.g. In Fig.6, Assume that transactions *Ti* to *Tk* are malicious or affected transactions, only *Ti* needs to be undone, while transactions *UT(i+1)* to *UTk* should be unscheduled. Otherwise, transaction *UTp (i<p<=k)* rewrites *x* to an inconsistent value *v(p-1)*. (show in (b))

Case 3: If the UPDATE and the last DELETE operations are malicious or affected, only the first UPDATE and the DELETE operations should be undone, while the undo time must be chosen when the undo transaction for DELETE finishes. Otherwise, the undo UPDATE transaction will operate on a "NULL item", and the result for recovery on x is not correct.
e.g. In Fig.6, Assume that transactions *Ti* to *Tk as well as Tn* are malicious or affected, only *Ti* and *Tn* need to be undone, while transactions *UT(i+1)* to *UTk* should be unscheduled. Otherwise, transaction *UTp (i<p<=k)* will operate on a "NULL item" (shown in (c)). The rearranged schedule for the undo process is shown in (d).

Fig.7. Inconsistent results for Forward Recovery and the solution

In order to eliminate the inconsistency for the Forward Recovery, we maintain a data structure *submitted_list* to keep every data item *x* whose undo operation has been submitted to database but has not been cleaned. Before an undo transaction is scheduled, we should firstly check whether there is an operation that operates data items in *submitted_list*. If so, the *TS* discards the operation; otherwise submits the operation to the database. This method can prevent Case1 and Case 2 in Fig.7., while the inconsistency for recovery for Case 3 in Fig.7 still can not be avoided. For the

undo operations with the *OPTYPE=UPDATE*, the return message must be captured by the *TS*, if it returns "0 rows affected", then it could be inferred that the data item operated by the undo operation has been deleted. So the operation must be redone while the INSERT undo operation towards the same data item accomplishes. A container *temp_con* is used to record these futile operations needed to be redone. The schedule algorithm is described as follows:

Algorithm 5. *Transaction Schedule Algorithm*

Input: undo transactions submitted by the *DAM*

Output: the correct database state with all malicious and affected transaction undone and modification by blind write transaction retained.

Initialization: *submitted_list:={}; tra_queue:={};*

STEP1. add every transaction into *tra_queue* according to submitting order from *DAM*;

STEP2. while (*tra _queue* $\neq \varnothing$)

get next transaction *T* from *tra_queue*;

if *T* is a compensation transaction **/**T* is a compensation transaction*/**

submit *T* to the database;

else **/**T* is an undo transaction*/**

for each operation *w[x,v]* in transaction *T*

if *x* is not in *cleaned_list* or *submitted_list*

submit *w[x,v]* to the database and add *x* to *submitted_list*;

if *w[x,v]* returns the message "0 rows affected"

add operation *w[x,v]* to *tem_con*;

STEP3. when submitted operation *w[x,v]* has executed successfully in database, delete *x* from *submitted_list*;

when the Termination Condition is satisfied, submit the operations in *tem_con* to the database system

3.4. Data Access Confinement Controller (DACC)

The *DACC* is proposed to speed up the recovery process. In this subsection, we start form the analysis for factors that affect the recovery efficiency, and then the necessity of introducing the *DACC* is deduced. Finally we present a Two-tier Data Access Confinement Algorithm to accelerate the recovery process.

3.4.1. Mean Time for Dynamic Recovery (MTDR)

MTDR is one of the most important performance metrics in the recovery system. It represents mean time of recovery for a transaction in malicious transaction set *B*. The definition is as follows.

Definition 5. Mean Time for Dynamic Recovery: *given a malicious transaction set B = {b1,b2,…,bn}; suppose the time is t (in milliseconds) when B is inputted into Algorithm 1 till the algorithm exits, then the Mean Time for Dynamic Recovery on B equals t/n.*

MTDR concerns on the efficiency for damage assessment, undo/redo transaction generation and execution. MTDR is an available metrics for a fault-tolerate system.

3.4.2. The time cost for dynamic recovery

The time cost for dynamic recovery is discussed to analyze factors that affect the MTDR. As mentioned above, the process of dynamic recovery is to locate and repair every dirty and affected data item. The factors which affect the recovery for a single data item will probably affect the MTDR. In the following, we discuss the recovery time for a single data item to introduce the necessity of the *DACC*.

There are four states for a data item in dynamic recovery algorithm: “clean”, “dirty”, “marked dirty” and “cleaning”. When data item *x* is written by malicious or affected transactions, *x* turns dirty. However, there is a time interval between *x* turns dirty and *x* is identified dirty. The time interval may be explained as the cost of intrusion detection and damage assessment. When *x* is added to *dirty_list* by *DAM*, it turns “marked dirty” state. When *x* is added to *submitted_list* while it is submitted to database by *TS*, it becomes “cleaning” state. When undo or compensation transactions are executed successfully, *x* transits to “clean” state again. The state transition diagram of a data item is shown in Fig.8.

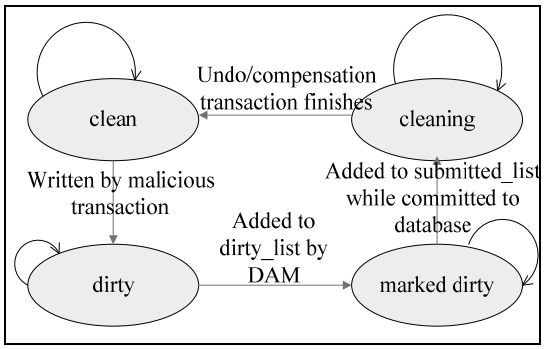


Fig.8. The state transition diagram of a data item in dynamic recovery

According to Fig.8, we divide recovery time for a data item *x* into four periods: 1) *x* from clean to dirty; 2) *x* turns from dirty to marked dirty; 3) from *x* is marked dirty to dirty data items affected by *x* (including *x*) is repairing; 4) dirty data items affected by *x* (including *x*) from repairing to the state that are all cleaned (see Fig.9).

We define the “recovery time” for a single data item *x* as the time from *x* is marked dirty to dirty items affected by *x* are all cleaned. It is indicated that the recovery time for dirty data items modified by transactions in malicious transaction set *B* affect MTDR. For period 1, the efficiency issue is corresponding to nature of user transaction (benign or malicious). For periods 2, the efficiency issue mainly depends on *Intrusion Detector* and the *DAM*. We have taken measures such as removing and merging redundant operations while undo and compensation transactions are created to improve efficiency of the *DAM*. Intrusion detection technology is beyond

consideration of this paper. For period 4, the efficiency issue is mainly determined by the throughput database system while repair is executing. In period 3, while dirty item *x* is marked dirty, a measure can be taken to confine the spreading of damage affected by *x*. The following subsection is devoted to the confinement measure.

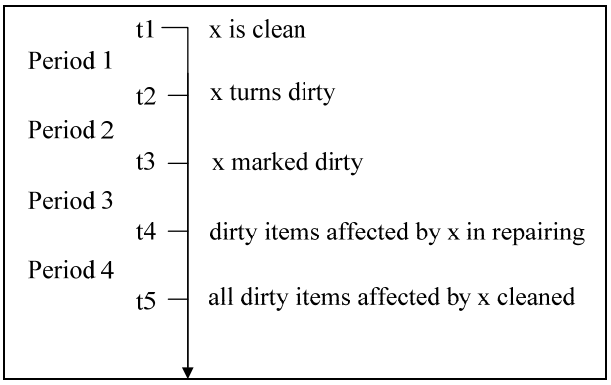


Fig.9. Recovery time for a single data item

In dynamic recovery, while damage is being repaired, user transactions can keep on accessing the database. If a data item read by a user transaction is dirty, the damage will spread. When the database is not busy, for a data item in the state "marked dirty", repairing it takes a short period of time. Because once an (a) undo/compensation transaction is submitted, it would be scheduled in a short time. However, when the database is busy, the time for recovery would increase sharply. Existing confinement mechanism in [Amm02] is cell-grained. For every user transaction submitted to the database during the recovery, the data items that the transactions attempt to access should be checked. There are defects for this approach: 1) To extract data items which is accessed by transactions is considerably time consuming (the method is similar to the Divide and Combination Algorithm), and the efficiency is decreased; 2) In normal application scenario, most of the new coming user transactions during the recovery period do not access dirty data items, so that it is not necessary to check all data items which the new coming user transactions attempt to access. In order to improve efficiency for the confinement, we introduced the *DACC* to the recovery system. Experiments show that the MTDR is greatly reduced with *DACC*.

3.4.3. Two-tier Data Access Confinement Algorithm

While the database is busy, we shall start the *DACC*. The *DACC* is designed with a two-tier algorithm. While user transactions access database during the recovery period, *DACC* checks the SQL statements submitted to the database (with the help of *DB Monnitor*). If a SQL statement attempts to access a data item in *dirty_list*, then the statement *is* discarded; Normal user requests are submitted to the database. The checking process is divided into two tiers: table tier and column tier. Firstly tables which would be accessed by a SQL statement are checked: if these tables' names do not appear in the *ITEM.tablename* fields for all records in *dirty_list*, *DACC*

regards the SQL statement is benign and would not cause damage spreading; if one of these tables' names appears in the *ITEM.tablename* fields, the second tier check is activated. In the second tier, columns which would be accessed by the SQL statement are checked. If these columns' names do not appear in the *ITEM.column* fields for all records in *dirty_list*, the *DACC* consider the SQL statement is benign and continues to execute it. If one of these columns' names appears in the *ITEM.column* fields, *DACC* predicate the SQL statement may cause damage spreading, and then discard it. The algorithm is shown as follows.

Algorithm 6. *Two-Tier Data Access Confinement Algorithm*

Input: SQL statements that would be submitted to the database.
Output: filtered SQL statement that can not cause damage spreading.
Initialization: *table_list:={};column_list:={}; dirty_table:={};dirty_column:={};*
STEP1.if there is a SQL statement submitted from a *Client user*, analysis it, and put names of tables the statement would accesse in *table_list*;
STEP2.put all distinct *ITEM.tablename* from *dirty_list* in *dirty_table;***/*the 1st confinement*/**
if *table_list* ∩*dirty_table*= ∅
submit the statement to database and execute it;
clean *table_list* and *dirty_table*;
exit;
else go to **STEP 3**;
STEP3.put names of columns the statement would accessed in *column_list*; **/*the 2nd confinement*/**
put all distinct *ITEM.column* from *dirty_list* in *dirty_column*;
if *column_list* ∩*dirty_column*= ∅
submit the statement to database and execute it;
else discard the statement;
clean *table_list*, *dirty_table*, *column_list*, and *dirty_column*;
exit;

The two-tier data access confinement makes a tradeoff between accurateness and efficiency of confinement. The queries which attempt to access data that has the same table name and column name with a dirty data are confined. The availability decreases while introducing the *DACC*. However, when database is busy, the confinement approach is efficient. The experiment results show that the tradeoff is valuable in a busy database system.

4. Experiment results and analysis

In this section, four experiments are proposed for the functionality measurement and performance evaluation of our dynamic recovery system. The first experiment examines the performance for *FGTL Generator*. The *FGTL Generator* audits the behaviors of transactions by

keeping a fine grained transaction log. The process for generating reading log is quite time consuming, and it may become the bottleneck of the recovery system. So to study and improve the performance of *FGTL Generator* is an important work; the efficiency for data recovery also relies on the test data set. If there are more transactions in the test data set have read-write dependency relationships, the recovery process will cause a longer time. So we construct different transactions sets which have various degrees of dependency relationship to study the efficiency for recovery system in experiment 2. In experiment 3, we examine the function of *DACC* in speeding up the recovery process when database is busy. In experiment 4, we use the TPC-W system to examine the integrity and performance of the recovery system, the purpose is to study whether the system is available in an applied e-commerce system.

4.1. Test Environment

We use *SQL Server 2000* as the underlying DBMS. The recovery system, *TPC-W* [*] platform and the SQL Server database system are deployed in distributed PCs. The database is installed on a PC with *Windows NT*, *Pentium R 2.8GHZ* CPU, *2GB* main memory. The recovery system and *TPC-W* platform is running on a PC with *Windows NT*, *Pentium R 2.8GHZ* CPU, *1GB* main memory. These two PCs are connected by a *10/100Mbps* switch LAN. The recovery system is implemented using *Eclipse* on *JAVA* platform.

4.2. NUM of Transactions per Second vs. Association Degree

A higher Association Degree will cause more time costs for auditing the Extended Read Operations. In this experiment, we qualify the throughput of *FGTL Generator* as Numbers of Transactions per Second, and our objective is to study the throughputs of *FGTL Generator* under different amounts of Extended Read Operations. We construct 5 sets of transactions to simulate different amounts of Association Degrees. The sets' Association Degrees varies from 1 to 5 separately (for a set of transactions *C*, its *Association Degree* equals *i* means that for each SQL statement *s* of each transaction in *C*, the *Association Degree* of *s* equals *i*). Each set has 1,000 short transactions, and each transaction consists of a *SELECT* and an *UPDATE* statement with the same *Association Degree*. *The number of Concurrent User* is *1*, *Database Size (records)* is 5,000, and *Record Size* (bytes) is 160. We compare throughputs with and without *FGTL Generator* (see Fig.10).

[*] *TPC-W* is a transactional web e-commerce benchmark introduced by the *Transaction Processing Performance Council*. *TPC-W* specifies an e-commerce workload that simulates the activities of a retail website which produces workload on the backend databases. *WIPS*, *WIPSb*, and *WIPSo* measure the number of web interactions per second under three different modes. *WIPSb* measures throughput under a test set with *5%* write interactions and *95%* interactions; *WIPS* measures throughput under a test set with *20%* write interactions and *80%* read interactions; and *WIPSo* measures throughput under a test set with *50%* write interactions and *50%* read interactions.

From Fig.10 we can see that throughput decreases by 20.7%, 35.4%, 54.5%, 69.5%, and 81.7% when Association Degrees increase from 1 to 5 when *FGTL Generator* runs. We conclude that a high *Association Degree* leads to a sharp decrease of throughput for *FGTL Generator*. This can be explained that high Association Degree indicates that a large amount of Extended Read Operations is generated, and a large amount of SQL statements are sent to *Read Log Generator*. For a test set of transactions with the Association Degree *i*, each SQL statement of a transaction inside the set is divided into (*i+1*) statements by the *Read Log Generator*. Additional *i* statements would be submitted to the database for each SQL statement with subqueries. So the total number of SQL statements executed for the test set increases sharply, and the dividing as well as the executing process for such a large amount of additional statements are quite time consuming. Therefore, the higher Association Degree would cause the lower throughput for the *FGTL Generator*.

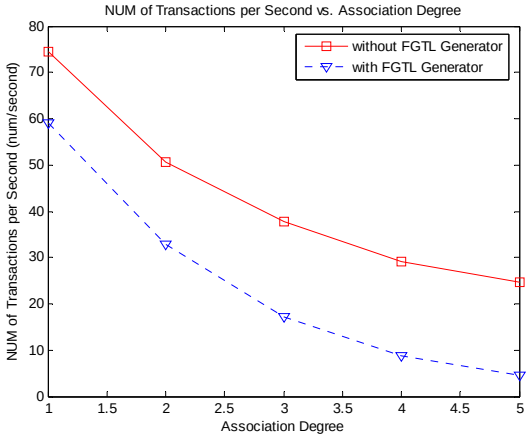


Fig.10. NUM of Trans. Per Second vs. Association Degree

4.3. MTDR vs. Dependency Degree (DD)

More dependency relationship between transactions in test sets will bring more time costs for damage repair. The objective of this experiment is to study the MTDRs under different situations of damage spreading. Damage spreading is caused by read-write dependency between transactions. In order to quantify the dependency relationships for a sequence of transactions, the Dependency Degree (DD) is introduced. The DD denotes the proportion of the transactions with dependency relationships in the sequence of transactions.

Definition 6. Dependency Degree (DD) for a sequence of transactions:

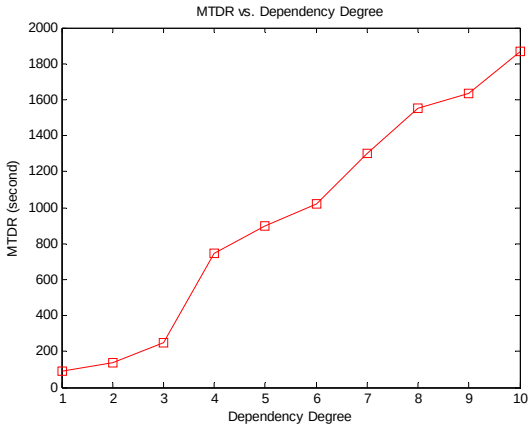
Suppose the number of transactions in the sequence is n, from beginning to the end of the sequence, for each transaction T, if there is a dependency from T to the former transactions (no matter how many transactions that T is dependent on), the Degree (initialized as 1) is increased by 1. Finally,

$$DD = \begin{cases} 0\ ; & Degree = 1 \\ Degree / n. & Degree \neq 1 \end{cases}$$

For Example 4, according to the definition of Dependency Degree, for *S*, *Degree=4*, and *DD*=4/5=80%.

Example 4. A sequence of transactions *S* is described as follows: *T1: a=a+1*; *T2: b=a+2*; *T3: c=b+3*; *T4: d=c+4*; *T5: e=x+5*.

In the experiment, DDs are set from 1 to 10, which represent 10%, 20%,…, 100% of test transactions are with dependency relationships. The *number of Concurrent User* is 10, *Database size (records)* is 10,000, and *Record Size* (bytes) is 160. One concurrent user executes 1,000 short transactions orderly, and each transaction consists of one SELECT and an UPDATE statement. The number of malicious transactions is set at 1000 for each amount of DDs. The results are shown in Fig.11.



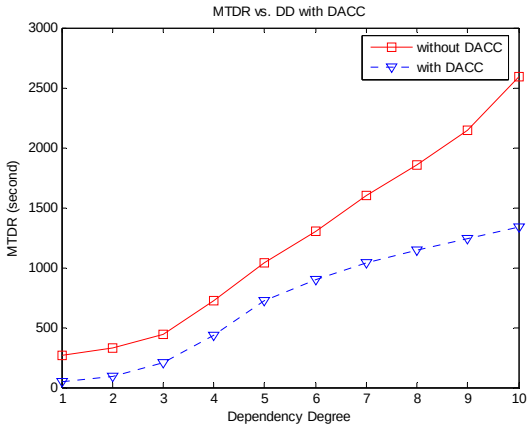


Fig.11. *MTDR* vs. *Dependency Degree*

Fig.12. *MTDR* vs. *DD* with *DACC*

Results from Fig.11 show that dependency degree affects the performance of the system. High dependency degree leads to high MTDR. This can be explained that the high dependency degree indicates that a large amount of dependency relationships exists between transactions in the test set. That means for the same amount of malicious transactions, the higher Dependency Degree is, the larger amount of affected transactions would be discovered by *DAM*, so that the whole process of dada recovery would consume more time, and consequently, the MTDR decreases. To sum up, we conclude that high Dependency Degree would lead to high MTDR.

4.4. MTDR vs. DD with DACC

The objective of this experiment is to study the varying of MTDR with and without *DACC* under different Dependency Degrees. In the experiment, we use the same experiment environment with the experiment discussed in Section 4.3, and only increase the number of concurrent users to

simulate the busy state for the database system. We compare MTDRs with and without *DACC* under different amount of DDs when the database is busy. The results are shown in Fig.12.

The results in Fig.12 show that with *DACC*, the *MTDR*s decrease by *48.4%* to *81.4%* when the database is busy. It is obvious that *DACC* is available in improving efficiency when the database is busy. This can be explained that when the database is busy, a large amount of current user is accessing the database, and a large amount of user transactions continuously access the database during the recovery period. The damages spread sharply. The higher the Dependency Degree is, the damage spreading is more obvious. With *DACC*, transactions which attempt to access damaged data is filtered, and the damage spreading is confined, so that the termination condition is satisfied in a short time, and the total recovery time decreases.

4.5. Integrity Measurement and Performance Evaluation under the Platform of TPC-W Benchmark

We choose the TPC-W benchmark as our test platform. The platform provides a scenario of an applied commercial database application which simulates various concurrent users' online-purchasing activities. In the context of TPC-W, the quantitative metrics provided by TPC-W benchmark could measure the functionality and performance for dynamic recovery. In the context for TPC-W platform, the data integrity metrics are proposed to measure the effectiveness for the dynamic recovery system, while the metrics for throughputs of the TPC-W benchmark are introduced to the performance evaluation. The value of main TPC-W parameters for our experiments are listed in Table 1:

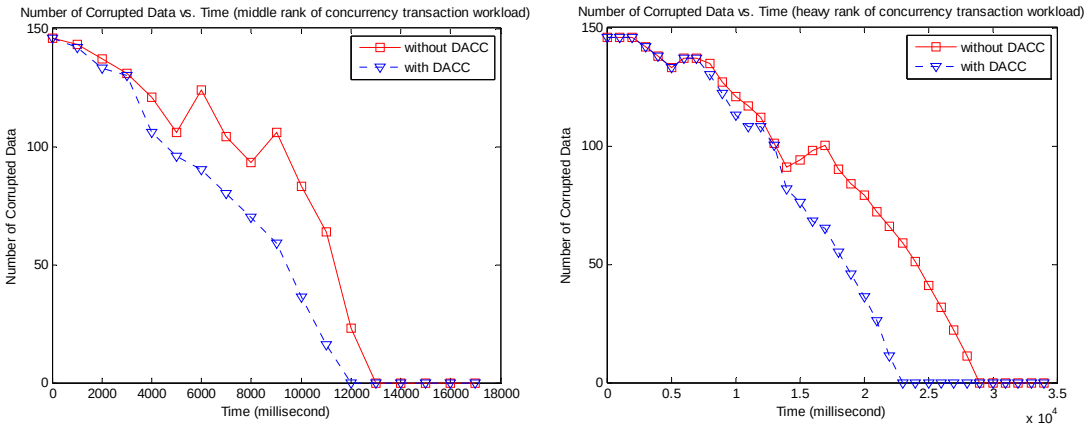
Table1. Parameters in *TPC-W* Platform

Parameter	value	Comments
NUM_EBS	*1K/10K*	number of concurrent user, which simulates various user transactions.
NUM_ITEM	*10K*	cardinality of table *item* (in rows), and also has relationship with the size of database

4.5.1 Integrity Measurement

On the point of security, we measure the effectiveness of the dynamic recovery system from the aspects of data integrity. The integrity metrics measure the amount of corrupted data items, and the damage propagation scales during the recovery period. In our experiment, we use the metrics Number of Corrupted Data vs. Time to indicate the varying of integrity for the database system under different time periods in face of malicious attacks. We focus on the varying of data integrity during the period when damages are being repaired. We compare the effects of data recovery in cases that the *DACC* runs and terminates, with a middle rank and a heavy rank of concurrent transaction workload under WIPS mode. (Here, the rank of concurrent transaction has a large impact on the scale of

damage propagation. The heavy concurrent transaction workload leads to a serious damage spreading. In TPC-W, the rank of concurrent transaction can be controlled by the parameter NUM_EBS. In our experiment, we indicate that for the middle rank of concurrent transaction workload, NUM_EBS=1K, while for heavy workload, NUM_EBS=10K). The results for the experiment are shown in Fig.13.



(a) (b)

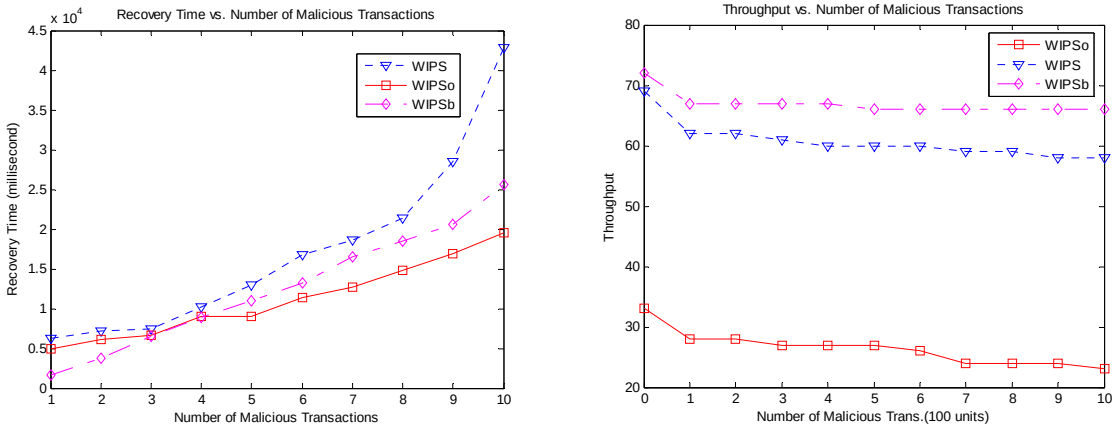
Fig.13. Integrity Measurement under TPC-W Benchmark

In the period of data recovery, The *DAM* assesses the scales of corrupted data items and creates undo transactions to restore the database to a consistent state. In order to gain a higher availability, user transaction can keep on accessing the databases in the repairing period. If the *DACC* is not activated, during the recovery period, the data access for user transactions may propagates damage and cause extra corrupted data items. So the curve of Number of Corrupted Data when the *DACC* is terminated declines slowly, even a slight rebound appears in the recovery period. On the contrary, the curve of Number of Corrupted Data when the *DACC* is activated declines quickly and no fluctuations occur. Comparing the repair time of (a) and (b) in Fig.13, we also conclude that the heavy workload of concurrency transactions may cause a large scale of damage spreading and lead to a longer recovery time.

4.5.2 Performance Evaluation

The objective of this experiment is to study the recovery time and throughputs under different amounts of malicious transactions. In TPC-W platform, the throughput is described using WIPS, WIPSo, and WIPSb. With the amount of malicious transactions increasing from 0 to 1,000, we calculate the recovery time and throughputs under the modes of WIPS, WIPSo, and WIPSb for the dynamic recovery system (specially, the number of malicious transactions equals 0 means that the dynamic recovery does not run, namely the throughput is obtained when only SQL server and TPC-W platform runs). The results are shown in Fig.14.

Results from Fig.14 (a) show that the curve of Recovery Time rises with the number of malicious transactions increase. This phenomenon is easy to explain: The large amount of malicious transactions is bound to cause a large scale of damage propagation, and the corresponding repair process should be extended. Meanwhile, there is another interesting phenomenon: the mode WIPS costs longest repair time while the mode WIPSo costs the shortest. It can be explained that in the mode WIPSb and WIPSo, the proportion for read and write transactions is in a state of imbalance (WIPSb has a proportion of Write transactions 20%, while WIPSo has the proportion 50%). For the damage propagation is caused by the read-write dependency, so just a balanced proportion for read and write transactions could cause a large scale for damage spreading. Maybe the WIPS mode satisfies the balance condition.



(a) (b)

Fig.14. Performance Evaluation under TPC-W Benchmark

Fig.14 (b) shows that with the number of malicious transactions increasing from 100 to 1,000, throughputs decrease by 6.94% to 8.33% for WIPSb, 10.14% to 15.94% for WIPS, and 15.15% to 30.30% for WIPSo. We could conclude that: 1) a large amount of malicious transactions would lead to a low throughput. This can be explained that a large amount of malicious transactions might cause a large number of affected transactions. For each of the malicious and affected transactions *DAM* would generate an undo transaction and the number of undo transactions increased. 2) The throughput decreases the most for WIPSo and the least for WIPSb. This can be explained that only when the Write Operations are executed, new damages would be generated, and the Extended Read Operations never cause damage spreading. WIPSo measures the throughput with the largest number of Write Operations (about 50%), so that a large amount of damages would be caused and a large number of undo transaction would be created for dynamic recovery system to repair the damages. Therefore, the throughput decreases the largest in WIPSo.

In contrast, for WIPSb, most operations (about 95%) are for read, so it is indicated that the damage spreading would not be obvious. Consequently, the number of WIPSb is the highest.

5. Related Work

In the field of damage recovery, [Amm02] first introduced the read-write dependency method in database systems, and proposed a data recovery model based on the transaction dependency. This model laid foundation for the damage recovery. [Liu00a] presented a model which rewrites transaction history to retain as many as benign transitions in the period of data recovery. With this model, only the suffix of the transaction history needed to be undone. However, the process of rewriting is very time consuming so that the model could not be applied to real application systems. [Pil02] proposed a novel thinking. It maintained a transaction dependency graph in the executing period of transactions to substitute the transaction log. Due to the mechanism of maintaining transaction dependency graph in the executing period, the process for recovery could be quite high-efficient. A prototype system *Phoenix* based on this model was also developed using *PostgreSQL* DBMS [Chi05, Chi08]. However, the model depends on the mechanism of *Multi-Version Concurrency Control (MVCC)* of the *PostgreSQL* DBMS and could not be applied to common database systems. [Amm02, Raj04] presented an “on-the-fly” recovery method, and firstly introduced the semantics of “dynamic” for data recovery to database system. [Liu02b, Val03] described the architecture of the model and [Liu02b, Val03, Wan07] illustrated a prototype system *ITDB* which provides recovery for commercial DBMSs. Evaluations towards performance of *ITDB* were given in [Wan00, Wann06, Wan07]. The *ITDB* could not capture the read-write dependency formed by Extended Read Operation, and the blind write operations are not considered yet.

Intrusion detection was one of the most important techniques upon which the recovery system was dependent [Amm02, Chi05, Liu00a, Lue01]. The data recovery required an exact and high-efficient intrusion detection mechanism so that the malicious and affected transactions could be located as soon as possible. In the area of intrusion detection, Christina [Chu99] and Lee [Lee02] proposed a transaction profile to excavate regular user activities, and abnormal transactions which were not consistent with the profile would be identified as attack-prone. This transaction profile based approach was proved available in detecting malicious transactions, and was introduced to provide intrusion detection for many fault-torrent systems [Lue01, Wan07]. Moreover, methods for detecting SQL injection attacks [Ber05, Jos07, Jos08] and stored procedure relative attacks [Xio08] were also proved effective and efficient. Based on these methods, we established an intrusion detection system to serve our Dynamic Recovery System.

In the area of transaction log, [Sob04] proposed a Read-Write Set approach which regards transactions as a sequence consisting of statements and IF-ELSE predicates. The approach records relative information according to predefined Read Set or Write Set. The flaw of this approach is that the IF-ELSE predicates and the execution sequence could only be extracted from the application level, and the database system could not get the application logic in the web context, so the method could not be applied in web databases. [Amm02, Lue01] proposed a Templates Materialization method to record the read-write information for transactions. Due to the transaction template being predefined, and only simple SQL statements could be recorded, the method could only be applied to a certain application scenario. On the contrary, the FGTL could be adapted to universal web applications with acceptable performance degradations, and meanwhile, possess the ability to record the data read by the subqueries inside SQL statements.

Towards the schedule rule for dynamic recovery, [Amm02] concisely described the conflict issues for the Forward Recovery approach. But certain inconsistency issues for damage recovery results are not considered. The Transaction Schedule Algorithm proposed in this paper supplies a complete solution to the Forward Recovery relative issues.

In the aspects of damage spreading confinement, [Liu00b, Liu01] proposed a Multi-Phase Damage Confinement approach. This approach implemented intrusion tolerance for database systems. However, the effect of Multi-Phase Damage Confinement depended on the accuracy rate and time duration for intrusion detection. It required that the damage confinement algorithm must adapt to the various means of intrusion detection so that the Multi-Phase Damage Confinement could only be applied to certain applications. [Liu06] gave a Multi-Version approach to confine damage spreading after attacks. When a transaction was identified abnormal, a new version of data was created to retain modifications of the transaction, and if the transaction was found malicious latter, the old version of the data would be applied to substitute current version (denoted Version Combination). The Multi-Version approach provided a new consideration for the damage recovery, but the multiple versions of the data occupied a large amount of spaces and the process of Version Combination was quite complicated. It brought significant performance degradation for large scale web applications. The DACC provided a compact and efficient algorithm for damage confinement. The performance evaluation in the experiments indicated that the confinement algorithm reached a balance between efficiency and effectiveness.

6. Conclusions and Future Work

In this paper, we presented a dynamic recovery technique which restores a database system from malicious attacks. This technique deals with the problem of the service disruption in the period of recovery. We proposed a concept of *Extended Read Operation*s and explained how the *Extended Read Operation*s would cause the damage spreading. Furthermore, a *Fine Grained Transaction Log* (*FGTL*) which could record data items read by subqueries inside SQL statements is introduced. Based on the *FGTL*, we implemented a *Dynamic Recovery System* which could provide continuous but maybe degraded database access while the recovery is processing. The system also retains the results for the blind write transactions and provides efficient damage confinement in the period of recovery using a *Two-tier Data Access Confinement Algorithm*. Meanwhile, the issue on recovery conflict for the *Forward Recovery* is solved. The system could be implemented on top of the commercial relational database systems and provide the functions of dynamic recovery for the database applications. Experiments based on TPC-W Benchmark show that the throughput decreased by 6.94% to 30.30% for introducing the *Dynamic Recovery System*. For common applications, the decrease is acceptable. The work for the optimization of the performance is on-going, and the impacts of integrity constraints on damage recovery are being considered.

References

[Amm97] P.Ammann, S.Jajodia, and C.D.McCollumet, "Surviving information warfare attacks on databases", In: Proc. of the IEEE Symposium on Security and Privacy, 1997.

[Amm02] P.Ammann, S.Jajodia, and P.Liu, "Recovery from malicious transactions", IEEE Transactions on Knowledge and Data Engineering, 2002.

[Bai08] K. Bai, P. Liu, and M. Yu, "TRACE: Zero-down-time Database Damage Tracking, Quarantine, and Cleansing with Negligible Run-time Overhead", In Proc. European Symposium on Research in Computer Security (ESORICS), pp.161-176, 2008.

[Ber05] E.Bertino, et al, "Intrusion detection in RBAC-administered databases", ACSAC, 2005.

[Chi05] T.Chiueh and D.Pilania, "Design, Implementation, and Evaluation of a Repairable Database Management System", IEEE International Conference on Data Engineering (ICDE), 2005.

[Chi08] T.Chiueh and S.Bajpai, "Accurate and Efficient Inter- Transaction Dependency Tracking", IEEE 24th International Conference on Data Engineering (ICDE 2008), pp.1209-1218, April 7-12, 2008.

[Chu99] C.Chung and L.Gertz, "DEMIDS: A Misuse Detection System for Database Systems", In: proc. of 3rd International IFIP Working Conference on Integrity and Internal Control in Information Systems, Kluwer Academic Publishers, 1999.

[Jos07] F.Jose, V.Marco, and H.Madeira, "Detecting malicious SQL", 4th International Conference on Trust, Privacy & Security in Digital Business (in conjunction with the 18th International Conference on Database and Expert Systems Applications), 2007.

[Jos08] F.Jose, V.Marco, and H.Madeira, "Online Detection of Malicious Data Access Using DBMS Auditing", 23rd Annual ACM Symposium on Applied Computing, 2008.

[Kan02] K.D.Kang, S.H.Son, and J.A.Stankovic, "STAR: Secure Real-Time Transaction Processing with Timeliness Guarantees", In: Proc. of the 23rd IEEE Real Time Systems Symposium, 2002.

[Lee02] S.Y.Lee, W.L.Low, and P.Y.Wong, "Learning Fingerprints For A Database Intrusion Detection System", ESORICS, 2002.

[Liu00a] P.Liu, P.Ammann, and S.Jajodia, "Rewriting histories: recovering from malicious transactions", *Distributed and Parallel Databases*, 2000.

[Liu00b] P.Liu, S.Jajodia, and C.D.McCollum, "Intrusion Confinement by Isolation in Information Systems", *Journal of Computer Security*, 2000.

[Liu01] P.Liu and S.Jajodia, "Multi-Phase Damage Confinement in Database Systems for Intrusion Tolerance", 14th IEEE Computer Security Foundations Workshop, 2001.

[Liu02a] P.Liu, et al, "Architectures for Intrusion Tolerant Database Systems", In: Proc. of 18th Annual Computer Security Applications Conf., 2002.

[Liu02b] P.Liu, et al, "Architectures for Intrusion Tolerant Database Systems", In: Proc. 18th Annual Computer Security Applications Conference (ACSAC), 2002.

[Liu06] P.Liu, H.Wang, and L.Liu, "Real-time data attack isolation for commercial database applications", *Journal of Network and Computer Applications*, 2006.

[Lue01] P.Luenam, and P.Liu, "ODAM: An On-the-fly Damage Assessment and Repair System for Commercial Database Applications", In: Proc. 15th IFIPWG Working Conf. on Database and Application Security, 2001.

[Pan98] B.Panda and J.Giordano, "An Overview of Post Information Warfare Data Recovery", In: Proc. of the 1998 ACM Symposium on Applied Computing, 1998.

[Pil02] D.Pilania and T.Chiueh, "Design, Implementation, and Evaluation of an Intrusion Resilient Database System", Experimental Computer Systems Lab, 2002.

[Raj04] Y.Rajesh and B.Panda, "Transaction Fusion: A Model for Data Recovery from Information Attacks", *Journal of Intelligent Information Systems Attacks*, 2004.

[Roi07] A.Roichman and E.Gudes, "Fine Grained Access Control to Web Databases", In: ACM Symposium on Access Control Models and Technologies (SACMAT), France, June 20-22, 2007.

[Sob04] R. Sobhan and B. Panda, "Reorganization of Database Log for Information Warfare Data Recovery", *ACM Transactions on Database System (TODS)*, 29(2):4-13, 2004.

[Val03] A.Valsangkar and B.Panda, "An Architecture for Making Data Available Ceaselessly During Recovery", In Proc. of the 2003 IEEE Workshop on Information Assurance, 2003.

[Vie05] M.Vieira and H.Madeira, "Detection of malicious transactions in DBMS", 11th IEEE International Symposium Pacific Rim Dependable Computing, (PRDC), 2005.

[Wan00] H.Wang, and P.Liu, "Modeling and Evaluating the Survivability of an Intrusion Tolerant DatabaseSystem", 11th European Symposium on Research in Computer Security (ESORICS), 2006.

[Wan06] H. Wang and P. Liu, "Modeling and Evaluating the Survivability of an Intrusion Tolerant Database System", In Proc. European Symposium on Research in Computer Security (ESORICS), 2006.

[Wan07] H. Wang, P. Liu and L. Li, "Evaluating the Survivability of Intrusion Tolerant Database Systems and the Impact of Intrusion Detection Deficiencies", *International Journal of Information and Computer Security*, Vol. 1, No. 3, pp. 315-340. 2007.

[Xie08] Meiyi Xie, Hong Zhu, Yucai Feng, "Tracking and Repairing Damaged Database Using Before Image Table", The Japan-China Joint Workshop on Frontier of Computer Science and Technology (FCST), 2008.

[Xio08] J.Xiong, Z.Cao, H.Zhu, and Y.Feng, "Detecting SQL Injection Attacks in Stored Procedure Based on Path-Construction", *Jaurnal of Computer Research and Development*, Vol.41, Suppl., 2008.

[Yum04] M.Yu, P.Liu, and W.Zang, "Self Healing Workflow Systems under Attacks", In: Proc. 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04), 2004.

[Zho05] Y.Zhong and X.L.Qin, "Database Intrusion Containment and Recovery Techniques under Information Warfare", *Chinese Journal of Computers*, 2005.

Hong Zhu received her M.S and Ph.D degrees from Huazhong University of Science and Technology in 1990 and 2001 separately. She is now a professor in School of Computer Science, Huazhong University of Science and Technology. Her research interests include database security, trusted database system, database system implementation etc.

Ge Fu received his B.S degree in software engineering from Xidian University in 2002. He is currently working toward the Ph.D. degree with the School of Computer Science, Huazhong University of Science and Technology, China. His research interests include database security, fault-tolerant database system, RFID security, etc.