Fine Grained Transaction Log for Data Recovery in Database Systems

Ge Fu, Min Chen, Hong Zhu, Yucai Feng, Yi Zhu , Jie Shi
Database and Multimedia Technology Research Institute
Huazhong University of Science and Technology
430074, Wuhan, Hubei, P.R.China
{fuge2006, shijie1123}@gmail.com, zhuhong@public.wh.hb.cn
{cm_ctgpc, fengyucai}@126.com, judy@mail.ccnu.edu.cn

Abstract

Data recovery for malicious committed transactions after attacks increasingly becomes an important issue. Damage assessment for data recovery requires a transaction log which record data items read or written by all malicious and benign transactions. Unfortunately, conventional undo/redo log could not record read operations for transactions; and existing auditing mechanisms in DBMS could not capture operations for data items. In this paper, we introduce a concept of "Extended Read Operations" and illustrate how the Extended Read Operations would cause the damage spreading, and then a Fine Grained Transaction Log (FGTL) is proposed. The log records all the data items of the read only and update-involved operations (read and write) for the committed transactions, and even extracts data items read by the subqueries in the SQL statements. A prototype system denoted FGTL Generator is developed to generate the FGTL. Experiments based on TPC-W Benchmark show the availability for FGTL Generator.

1. Introduction

Traditional security mechanisms such as authentication & authorization, access control, and data encryption in modern database systems are mainly preventive. These measures sometimes are penetrated by malicious attacks and may cause damages in database systems [2, 3]. So data recovery after damage becomes an important issue [16]. In modern database systems, the survivability and availability are becoming key criterions. More and more database systems are devoted to protect themselves against syntactically correct and semantically damaging transactions, which could arise because of malicious attacks (for example, SQL injection, cross-site script attack). The complexity for recovery from malicious attacks is caused by damage spread: the results of malicious transactions can affect the execution of some other transactions [8, 9]. Informally, when a transaction T_i reads a data item x updated by another transaction T_j (we say T_i reads x from T_j), T_i is directly dependent on T_j (T_i is an affected transaction). If a third transaction T_k is dependent on T_i, but not directly dependent on T_j, then T_k is indirectly dependent on T_j. It is easy to see that, when a (relatively old) transaction B_i that updates x is identified malicious, the damage on x can spread to every data item updated by a transaction that is dependent on B_i directly or indirectly. The goal of damage recovery is to locate each affected transaction and recover the database from the damage caused on the data items updated by every malicious or affected transaction (damage assessment and data recovery).

Approaches for damage assessment in existing recovery models [1, 4, 12]are based on dependencies between transactions. In order to obtain the dependencies, we should generate a log which records data items read or written by transactions. For each data item that are manipulated by transactions, the row number, column number, before image (the value before modified) and after image (the value after modified), and operate time should be recorded in the log. Existing mechanisms in DBMS could not provide the functions:

1) Conventional undo/redo log is only for write operations, and can not capture read operations for transactions;

2) Existing auditing mechanisms are designed to audit database statements, privileges, or schema objects. The audit is over the "table level". Namely, when an operation is executed on a table, information about user, privilege, operate time, SQL statement etc. for the operation can be logged. Unfortunately, the audit can not obtain data items that are manipulated by the operation.

In order to record data items that are read or written by a transaction, we must record the data items that are operated by every SQL statements inside the transaction. That is not an easy work:

1) In common cases, we could not calculate data items

that are read or written by a SQL statement until the statement is executed. We should record the data items in the executing period of the statement;

2) The damage assessment locates damage data items based on read-write dependencies between transactions: Suppose a transaction T reads a data item which is modified by a malicious transaction before, damage may spread while T writes some other data items latter [1, 4, 12]. Especially, if T executes a statement with subqueries, it could be inferred that T would read the data items involved in subqueries impliedly. If a data item involved in subqueries of the statement was modified by another malicious transaction before, the read operation in the subqueries for T would also cause the spread of damages if T writes some other data items latter. Therefore, for each SQL statement in a transaction, it is indicated that not only the outer SELECT sub-clause but also the nested subqueries in the SELECT statements should be recorded as read operations, so should subqueries in the UPDATE, DELETE, or INSERT statements. Unfortunately, existing approaches could not record data items read by subqueries inside a SQL statement. So damage spread caused by read operations in the subqueries of the statement could not be captured by damage assessment (see **Example 1**).

In this paper: 1) We propose a concept of "Extended Read" operation which could represent subqueries inside a SQL statement. It is proved that extend read may cause damage spread. 2) Based on the concept of extend read, a "Fine Grained Transaction Log" (FGTL) is proposed which could record data items manipulated by all write and extended read operations for transactions. A prototype system FGTL Generator is established to obtain correct FGTL efficiently. 3) In the prototype system FGTL Generator, FGTL is put in the database system (denoted Transaction Log table). For security, it is indicated that the Transaction Log table should not be modified by user transactions. Therefore, the problem of protection for FGTL is discussed.

The rest of this paper is organized as follows. In Section 2 we discuss related work. Section 3 is devoted to FGTL. Section 4 provides the results of a detailed performance study of the prototype system FGTL Generator based on the TPC-W Benchmark. Section 5 summarizes this paper with its main contributions and an outline of future work.

Example 1 malicious transaction B: *UPDATE item SET i_id = 15 WHERE i_cost = 100*; benign transaction G: *UPDATE order_line SET ol_qty =200 WHERE ol_i_id IN (SELECT i_id FROM items WHERE i_cost =100)*; 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 these products. After G is executed, damage spreads from *i_id* of table item to *ol_qty* of table *order_line*. Existing methods could not record the data item *i_id* read by the subquery "*SELECT i_id FROM items WHERE i_cost =100*", so the damage spread could not be captured by damage assessment.

2. Related Work

There are several models for data recovery: Ammann [1] first introduced the read-write dependency method to database systems, and proposed an "on-the-fly" recovery model based on the transaction dependency. The model logged transaction history in executing period of transactions, and established the dependencies of transactions based on the history, after that undoes all the malicious and affected transactions. [5, 13] described the architecture of the "on-the-fly" recovery model and [7, 15] illustrated a prototype system ODAR which provides recovery for commercial database systems. Evaluations towards the performance of ODAR are given in [14].

P.Liu adopted the approach of extracting read information from the transaction profiles [1]. The approach analyzes the source code of each type of transaction off line and gets the read set template of that type. When a transaction T is submitted to the database, the read set template is materialized for T's type. The approach of transaction profiles has some limitations: 1) It could only be applied to specific applications and can only capture read data items for certain SELECT statements in the read set template. 2) Subqueries in SELECT and write operations could not be recorded, either. [10] proposed an extended data dependency log to classify read and write operations. A new definition of transaction is given to represent "*if-else*" statement. With this log, data recovery could be robust. This log is proposed in the level of application, but in the level of databases, we can not obtain the semantics such as *if-else* clause presented in application level. The log could not be applied to real application systems. [6, 11] illustrated a transaction log which rewrites the transaction history. The prefix of the rewritten history serializes exactly the set of benign transactions, and the suffix of the rewritten history includes special state information to describe affected transactions as well as malicious transactions. In data recovery, only the suffix of the transaction history needs to be undone. The log is applied to keep more benign operations, and constructing the log is also based on recording data items for read and write operations. The problem on how to construct the transaction log is not mentioned.

3. Fine Grained Transaction Log (FGTL)

In order to define the fine grained transaction log, 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 number of the extended read operations involved in t.
In the following discussion, for simplification, we use "read operation" to substitute "extended read operation". The fine grained transaction log is defined as follows.
Definition 4 Fine Grained Transaction Log
A Fine Grained Transaction Log is a set of log entries and it satisfies the following conditions:
1) a log entry is a five-tuple: [TRASATIONID, OPTYPE, ITEM, BEFOREIMG, AFTERIMG, TIME] [1] , and each log entry is built according to a data item which a transaction reads or writes;
2) when a transaction is committed, entries for write and read operations involved in the transaction are added to the log.

3.1. Architecture of the Fine Grained Transaction Log Generator

In this section, a prototype system *Fine Grained Transaction Log Generator* (*FGTL Generator*) is introduced, which generates the FGTL. The architecture of the *FGTL Generator* is illustrated in **Figure 1**. There are several components in the *FGTL Generator*: *DB Monitor*, *Read Log Generator*, and a *database* (including *Transaction Log table*, *Triggers* and *Temporary Tables*). *Transaction Log table* records all log entries for read and write operations. In order to generate log entries for read operations, firstly we should capture the SQL statements submitted by a *Client user* before they are sent to the database, and then analyze and rewrite the statements to obtain the read log entries. *DB Monitor* is introduced to obtain SQL statements submitted by the *Client user*. The tasks for *DB Monitor* are: 1) extract a SQL statement and send it to *Read Log Generator*; 2) after the statement is processed by *Read Log Generator*, 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, sends the results to the *Client user*.

[1] TRANSATIONID denotes the transaction that reads or writes the data item. OPTYPE is an item of SELECT, UPDATE, INSERT, DELETE: for read operation, OPTYPE = SELECT. ITEM involves tablename, rownumber, and column. 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.

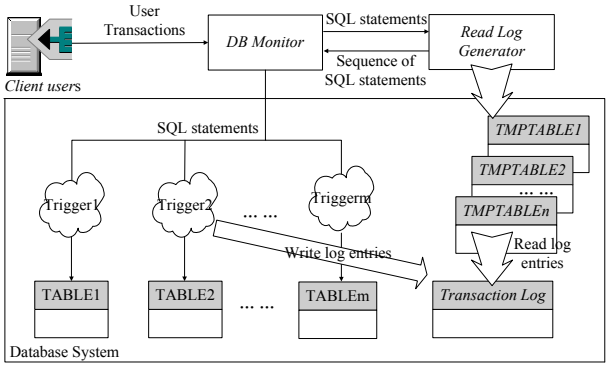

Figure 1. Architecture of FGTL Generator

Read Log Generator generates read log entries. Based on the *DB Monitor*, *Read Log Generator* records data items read by transactions in the *Temporary Tables* by using a "Divide and Combination algorithm", and then 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 write a log entry to *Transaction Log table* in the database.

3.2. Logging Read Operations for a SQL Statement

The work for *Read Log Generator* is to log all read operations in SQL statements sent from the *DB Monitor*. These statements include SELECT statements (may have subqueries) and UPDATE/ INSERT/ DELETE statements with subqueries. It's difficult to log the read operation which is a subquery of a SQL statement. In order to solve this problem, a "Divide and Combination algorithm" is proposed to log the read operations. The algorithm divides a statement into several SELECT ...INTO... statements to record read data items in temporary tables and then combines the generated statements into a SQL sequence, which has 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 relative subquery. The last SELECT statement is created

Input: a SQL statement (Association Degree = *n*) sent from *DB Monitor*.
Output: a sequence of SQL statements 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 column$_1$, ..., column$_n$... FROM...*")
from the most inner subquery to outer (Layer Number from *1* to *n*), for each subquery *i*, create *SELECT ... INTO TMPTABLE$_i$...*. The temporary table *TMPTABLE$_i$* records the intermediate results for subquery *i* (each row of *TMPTABLE$_i$* 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 *TMPTABLE$_i$* derectly;
after *n SELECT ... INTO...* statements are created, create a statement *SELECT column$_1$,..., column$_n$ FROM TMPTABLE$_n$...* 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 the **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 *i* returns, let it read from *TMPTABLE$_n$* directly;
send (*n+1*) SQL statements to *DB Monitor*;

Figure 2. Divide and Combination Algorithm

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 $= n$). $(n + 1)$ 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 database, temporary tables which record all data items read by the initial SQL statement are created. The "Divide and Combination" algorithm is described in **Figure 2**.

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* $TMPTABLE_1$ *FROM item WHERE i_cost>100*; 2) *SELECT ol_id, ol_qty INTO* $TMPTABLE_2$ *FROM order_line where ol_i_id IN (SELECT i_id from* $TMPTABLE_1$*)*; 3) *SELECT ol_qty from* $TMPTABLE_2$; The three SQL statements are sent to 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, two SQL statements are generated: 1) *SELECT i_id INTO* $TMPTABLE_1$ *FROM item WHERE i_cost>100*; 2) *UPDATE order_line SET ol_qty = 200 WHERE ol_i_id IN (SELECT i_id from* $TMPTABLE_1$*)*; The two SQL statements are sent to the database orderly.

The correctness of the Divide and Combination Algorithm is proved in the following.

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

Proof We denote the input SQL statement as *SELECT TargetList FROM TableList WHERE Condition*. the Divide and Combination Algorithm 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 enquiry, and *TargetList_modified* denotes the set of target columns as well as the column of primary key. Obviously: $TargetList \subset TargetList\_modified$.

Suppose the execution results set for the input statement is *u_input*, the execution results set for the sequence of 1) 2) which are generated by the Divide and Combination Algorithm 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)$u\_TMPTABLE = \pi_{TargetList\_modofied}\ (\delta_{Condition}(TableList))$;
3)$u\_division = \pi_{TargetList}(TMPTABLE)$.
From2) and 3), we conclude that:
$u\_division = \pi_{TargetList}(\pi_{TargetList\_modified}(\delta_{Condition}(TableList)))$.
According to the Divide and Combination Algorithm,we have $TargetList \subset TargetList\_modified$. So $u\_division = \pi_{TargetList}(\delta_{Condition}(TableList)) = u\_input$.

Theorem 2 *For an input SQL statement with Association Degree=n, the sequence of SQL statements generated by the Divide and Combination Algorithm 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:
$SELECT\ TargetList_n\ FROM\ TableList_n\ WHERE\ Codition_n\ SELECT\ TargetList_{n-1}\ FROM\ TableList_{(n-1)}\ WHERE\ Condition_{(n-1)} \ldots SELECT\ TargetList_1\ FROM\ TableList_1\ WHERE\ Condition_1$

According to the Divide and Combination Algorithm, 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 is 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, $i = 1, 2, \ldots, n-1$).The subquery with the Layer Number n returns results to the *Client user*.

According to the properties above and *Theorem 1*, we can conclude that for each subquery with the Layer Numer i, the ith statement in the sequence of statements generated by the Divide and Combination Algorithm keep the results in $TMPTABLE_i$ which involves the intermediate execution results of the subquery with the Layer Numer i for the input statement ($i = 1, 2, \ldots, n$). The $(n+1)th$ statement in the sequence of statements generated by the Divide and Combination Algorithm 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 Numer i, the ith statement in the sequence of statements generated by the Divide and Combination Algorithm keep the results in $TMPTABLE_i$ which involves the intermediate execution results of the subquery with the Layer Numer 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 the Divide and Combination Algorithm has the same execution results with the input SQL statement.

Theorem 3 *the Divide and Combination Algorithm logs all read operations for a SQL statement.*

Proof According to **Definition 2**, the 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 the Divide and Combination Algorithm, $TMPTABLE_1$ to $TMPTABLE_n$ are created which record all data items read by the input statement. So all read operations for the statement are logged.

3.3. Fine Grained Transaction Log Generation Algorithm

The Divide and Combination Algorithm generates read log entries for read operations. Based on *Algorithm 1*, the Fine Grained Transaction Log Generation Algorithm is proposed. The algorithm works as follows: *DB Monitor* captures statements of user transactions. For each SQL statement with read operations, the statement is sent to Read Log Generator. The *Read Log Generator* divides the statement into several statements and combines the statements into a sequence, and then sends the sequence back to *DB Monitor*. *DB Monitor* submits the sequence of SQL statements to the database. After all statements in the sequence are executed, read log entries based on temporary tables are generated. For write operations, triggers capture the modification on data items and then write log entries are generated. The Fine Grained Transaction Log Generation

Input: user transactions 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 the Divide and Combination Algorithm , *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 and records with fields TRASATIONID, OPTYPE, ITEM, BEFOREIMG, AFTERIMG, and TIME are stored in *Transaction Log table* in database;

Figure 3. Fine Grained Transaction Log Generation Algorithm

Algorithm is as **Figure 3**:

Theorem 4 *The Fine Grained Transaction Log Generation Algorithm logs each write and read operation for 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) the Divide and Combination Algorithm captures all read operations in a SQL statement. Using the Divide and Combination Algorithm, the Fine Grained Transaction Log Generation Algorithm logs read operations for all transactions, and log entries are written to the Transaction Log table, too.

From 1) and 2), we conclude that all write and read operations for user transactions are logged in the *Transaction Log Table*.

3.4. Protection for the Transaction Log

The transaction log is generated to record data items which are read or written by user transactions, and it is the data source for damage assessment. It is indicated that the transaction log should not be modified by any user transaction. In *FGTL Generator*, *DB Monitor* could protect transaction log from being modified by user transactions: Before a SQL statement of a user transaction is sent to the database system, *DB Monitor* captures the statement and checks if the statement would attempt to access the *Transaction Log table*. If so, *DB Monitor* would discard the statement and do not submit it to the database. Therefore, only statements which do not attempt to access the *Transaction Log table* can be executed for user transactions. In this way, the security of the transaction log could be guaranteed.

4. Experiment results and analysis

In this section, we give two experiments to examine the performance of *FGTL Generator*. In the first experiment, we build a test environment to examine the throughputs varying under several Association Degrees. The Association Degrees is used to quantify read operations for test transactions. In the second experiment, we use TPC-W platform to compare throughputs of with and without *FGTL Generator*. 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. An EB (Emulated Broswer) simulates a concurrent Client user. In the second experiment, the numbers of EBs in TPC-W platform are set from 100 to 500. For each number of EBs, we test the values of WIPS, WIPSb, WIPSo with and without *FGTL Generator*. WIPSb measures throughput under a test collection with 5% write transactions and 95% read transactions; WIPS measures throughput under a test collection with 20% write transactions and 80% read transactions; and WIPSo measures throughput under a test collection with 50% write transactions and 50% read transactions. Overheads for *FGTL Generator* for each EBs are also analyzed. In TPC-W, number of EBs and parameter NUM_ITEM affect the size of the database.

4.1. Test Environment

We use SQL Server 2000 as the underlying DBMS. *FGTL Generator*, TPC-W platform (used in the second experiment) and the database system are deployed in two distributed PCs. The database is running on a PC with Windows NT, Pentium R 2.8GHZ CPU, 2GB main memory. The *FGTL Generator* and TPC-W platform is run-

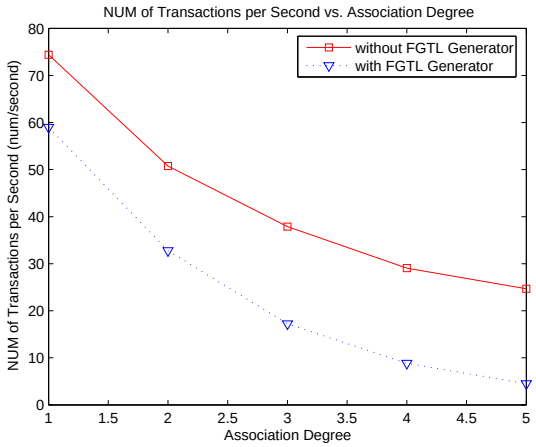

Figure 4. NUM of Transactions per Second vs. Association Degree

ning 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. FGTL Generator is implemented using Eclipse on JAVA platform.

4.2. NUM of Transactions per Second vs. Association Degree

The objective of this experiment is to study throughput of *FGTL Generator* under different amounts of read operations (quantified with Association Degree). To show the varying of the throughput (measured with numbers of transactions per second) under different Association Degrees, we construct 5 collections of transactions. The collections' Association Degrees equals 1 to 5 separately. (a collection C's Association Degree equals i means that for each SQL statement s of each transaction in collection C, the Association Degree of s equals i). Each collection has 1,000 short transactions, and each transaction consists of a SELECT and an UPDATE statements 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 **Figure 4**).

From **Figure 4** we can see that throughput decreases by 20.7%, 5.4%, 54.5%, 69.5%, 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 read operations is generated, and a large amount of SQL statements are sent to *Read Log Generator*. For a test collection of transactions with the Association Degree equals i, each SQL statement of a transaction inside the collection is divided into $(i+1)$ statements by *Algorithm 1*. So additional i statements would be submitted to the database for each SQL statement with subqueries. The total number of SQL statements executed for the test collection increases sharply. The dividing process and the executing process for so large amount of additional statements are time consuming. Therefore, the higher Association Degree would cause lower throughput for *FGTL Generator*.

4.3. Throughput in TPC-W Benchmark

The reason we choose TPC-W Benchmark for our experiment is: 1) TPC-W simulates a e-commerce workload which produces workload on the backend database, and the availability of *FGTL Generator* can be shown; 2) TPC-W provide three web interaction patterns and use WIPSb, WIPS, WIPSo to measure performances under different web interaction patterns. In the three patterns, amounts of read transactions and write transactions can be controlled. So it is possible for us to calculate which operations would lead to larger decrease of performance.

The objective of this experiment is to study throughput of *FGTL Generator* under different amounts of EBs. The numbers of EBs are set 100, 200, 300, 400, and 500. In TPC-W platform, the size of database and test collection of transactions could considerable large. So if each read or write operation is logged, the size of Transaction Log table should be large and the test would consume a very long time. In order to simplify the test, we set some "sensitive columns" for user tables in the database for TPC-W platform (if a column is set "sensitive", it means that only read and write operations on that column would be logged in the Transaction Log table, and operations on other column would not be logged). In the experiment, we set the column *ol_qty* in table *order_line*, the column *i_cost* in table *item*, the column *addr_id* in table *address* and column *scl_qty* in table *shopping_cast_line* as sensitive columns. So that the size of *Transaction Log table* could be reduced. The parameter NUM_ITEM is set 10K for the test. For each number of EBs, we compare the throughputs of WIPSo, WIPS, WIPSb of with and without *FGTL Generator*. The overheads of throughputs for *FGTL Generator* are listed in **Figure 5**.

From **Figure 5**, we can see that: 1) With the number of EBs increasing, the overhead of *FGTL Generator* become large. This could be explained that a large amount of EBs indicates that a large amount of user transactions is being processed in the database system. Therefore, the amount of read and write operations which are being processed by *FGTL Generator* is also large, so that the Fine Grained Transaction Log Generation Algorithm would generate a large amount of log entries in the *Transaction Log table*. Creating such a large size transaction log is considerable time consuming. We conclude that larger size of the

Throughput(WIPSo)	EBs=100	EBs=200	EBs=300	EBs=400	EBs=500
without FGTL Generator	14	28	35	39	42
with FGTL Generator	13	23	27	30	32
Overhead	7.14%	17.86%	22.86%	23.08%	23.81%
Throughput(WIPS)	EBs=100	EBs=200	EBs=300	EBs=400	EBs=500
without FGTL Generator	14	30	41	55	70
with FGTL Generator	13	24	32	40	49
Overhead	7.14%	20.00%	21.95%	27.27%	30.00%
Throughput(WIPSb)	EBs=100	EBs=200	EBs=300	EBs=400	EBs=500
without FGTL Generator	14	32	42	56	71
with FGTL Generator	13	25	32	40	49
Overhead	7.14%	21.88%	23.81%	28.57%	30.99%

Figure 5. Throughput vs. NUM of EBs

transaction log leads to lower throughput for FGTL Generator. The decrease of throughput for *FGTL Generator* is mainly due to the time cost of logging read and write operations. For write operations, log entries are generated by triggers, and for read operations, log entries are temporarily stored in temporary tables and then written to the Transaction Log table. Either of the two processes is time consuming. 2) In WIPSo, the overhead of throughput for *FGTL Generator* is the lowest and in WIPSb is the highest. This can be explained that WIPSb measures throughput under a test set with 5% write transactions and 95% read transactions. The proportion of read operations is the largest, and in contrast, proportion of read operations for WIPSo is the smallest (50%). Compared the read with the write operations, logging read operations consumes more time. The cause may be due to the complexity for logging read operations: for each SQL statement with Association Degree equals i, additional i statements are generated, and i temporary tables are created to store read data items. The process of division for SQL statements and writing read log entries from temporary tables to Transaction Log table would consume much time. In contrast, for write operations, triggers only capture modifications on data items and log the write operations in the transaction log directly. This process consumes much less time.

5. Conclusions and Future Work

In this paper, we propose a concept of Extended Read Operations and illustrate how the Extended Read Operations would cause the damage spreading in database systems. Then a Fine Grained Transaction Log (FGTL) which logs all data items read or written by transactions is introduced. FGTL makes up for the limitations of conventional undo /redo log and auditing mechanisms, and provides a correct and fine-grained transaction history for the data recovery. Compared with existing transaction logs, FGTL could keep record of the data items read by subqueries inside SQL statements. A prototype system *FGTL Generator* is also established to generate the fine grained transaction log. *FGTL Generator* also provides a protection mechanism for the transaction log. Experiment based on TPC-W Benchmark indicates that the throughput decreases by 7.14% to 30.99% with *FGTL Generator* (the number of EBs from 100 to 500). Now based on *FGTL Generator*, a dynamic data recovery system is being developed, and the recovery would provide continues but maybe degraded services while damage is being repaired. The constraint of data integrity in database system is considered in the dynamic recovery system.

6. Acknowledgement

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

References

[1] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Trans. Knowl. Data Eng.*, 14(5):1167–1185, 2002.

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

[3] K. Kang, S. Son, and J. Stankovic. Star: Secure real-time transaction processing with timeliness guarantees. In *23rd IEEE Real-Time Systems Symposium*, pages 303–314, Dec. 2002.

[4] C. Lala and B. Panda. Evaluating damage from cyber attacks: a model and analysis. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 31(4):300–310, 2001.

[5] P. Liu. Architectures for intrusion tolerant database systems. In *18th Annual Computer Security Applications Conference*

(ACSAC 2002), pages 311–320, Las Vegas, NV, USA, Dec. 2002.

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

[7] P. Luenam and P. Liu. Odar: An on-the-fly damage assessment and repair system for commercial database applications. In *Database and Application Security XV, IFIP TC11/WG11.3 Fifteenth Annual Working Conference on Database and Application Security*, pages 239–252, Niagara on the Lake, Ontario, Canada, July 2001.

[8] B. Panda and J. Giordano. An overview of post information warfare data recovery. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 164–174, Feb. 1998.

[9] B. Panda and J. Giordano. Reconstructing the database after electronic attacks. In *Proceedings of the 12th IFIP 11.3 Working Conference on Database Security*, pages 143–156, July 1998.

[10] B. Panda and K. A. Haque. Extended data dependency approach: a robust way of rebuilding database. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, pages 446–452, Madrid, Spain, Mar. 2002.

[11] B. Panda and S. Tripathy. Data dependency based logging for defensive information warfare. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC)*, pages 361–365, 2000.

[12] B. Panda and R. Yalamanchili. Transaction fusion in the wake of information warfare. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC)*, pages 242–247, Las Vegas, NV, USA, Mar. 2001.

[13] A. Valsangkar and B. Panda. An architecture for making data available ceaselessly during recovery. In *IEEE Systems, Man and Cybernetics Society Information Assurance Workshop*, pages 196–202, West Point, New York, USA, June. 2003.

[14] H. Wang and P. Liu. Modeling and evaluating the survivability of an intrusion tolerant database system. In *Proceedings of 11th European Symposium on Research in Computer Security*, pages 207–224, Hamburg, Germany, Sep. 2006.

[15] M. Yu, P. Liu, and W. Zang. Self-healing workflow systems under attacks. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 418–425, Hachioji, Tokyo, Japan, Mar. 2004.

[16] Y. Zhong and X. Qin. Database intrusion containment and recovery techniques under information warfare. *Chinese Jouinal of Computers*, 32(7):78–82, 2005.