Design, Implementation, and Evaluation of A Repairable Database Management System

Tzi-cker Chiueh Dhruv Pilania

Rether Networks Inc.
99 Mark Tree Road, Suite 301, Centereach, NY 11720
{**chiueh@rether.com**

Abstract

Although conventional database management systems are designed to tolerate hardware and to a lesser extent even software errors, they cannot protect themselves against syntactically correct and semantically damaging transactions, which could arise because of malicious attacks or honest mistakes. The lack of fast post-intrusion or post-error damage repair in modern DBMSs results in a longer Mean Time to Repair (MTTR) and sometimes permanent data loss that could have been saved by more intelligent repair mechanisms. In this paper, we describe the design and implementation of *Phoenix* - a system that significantly improves the efficiency and precision of a database damage repair process after an intrusion or operator error and thus, increases the overall database system availability. The two key ideas underlying *Phoenix* are (1) maintaining persistent inter-transaction dependency information at run time to allow selective undo of database transactions that are considered "infected" by the intrusion or error in question and (2) exploiting information present in standard database logs for fast selective undo. Performance measurements on a fully operational *Phoenix* prototype, which is based on the PostgreSQL DBMS, demonstrate that *Phoenix* incurs a response time and a throughput penalty of less than 5% and 8%, respectively, under the TPC-C benchmark, but it can speed up the post-intrusion database repair process by at least an order of magnitude when compared with a manual repair process.

1 Introduction

Database management systems are an indispensable component of modern Internet services and are typically responsible for storing and providing access to mission-critical data. As DBMSs become more accessible over the network, they are also more susceptible to malicious attacks from remote sites, e.g., SQL Slammer [22]. On the other hand, because DBMS holds information that is critical to the continuous functioning of Internet services, high DBMS availability is absolutely essential. Although the original motivation for this work is survivability, i.e., ability to quickly recover from malicious security break-ins, the same mechanism is equally effective for repairing database damage caused by human errors. However, for the rest of this paper, we will focus only on intrusion damage.

There is a fundamental difference between a security breach and a hardware failure: Unlike a hardware failure, a security breach cannot always be detected immediately. the interval between when an intrusion takes place The *detection window* is the interval between when an intrusion takes place and when it is detected. Data entered during the detection window and not infected by the intrusion should not be affected by the post-intrusion repair process. We use the word "repair" rather than "recover" to emphasize the additional, often manual, efforts required to preserve useful data in the process of restoring the database back to normal operation.

Today, system administrators have two choices when it comes to repairing a database corrupted by an intrusion. First, they can restore the entire database back to the state before the intrusion took place. This approach is simple and fast, but could lead to loss of non-infected data created during the detection window. Alternatively, they can attempt to preserve the non-infected data as much as possible by manually removing all the side effects of the intrusion. However, this approach is labor-intensive, and thus time-consuming and error-prone. A repairable DBMS can minimize the post-intrusion repair time while preserving as much non-infected data as possible, thus achieving the best of both worlds.

Availability is defined as the ratio between mean time to failure (MTTF) and the sum of MTTF and mean time to repair (MTTR) ($availability = MTTF/MTTR$), where failure in this case corresponds to a successful attack. To maximize the availability metric, one can either increase

the MTTF to infinity or decrease the MTTR to zero. In contrast to most traditional database security mechanisms, which aim at maximizing the MTTF, the approach described in this paper attempts to minimize the MTTR, or fast repair of database damage left by security breaches, thus improving a DBMS's dependability by increasing it's availability.

There are two justifications for the MTTR minimization approach toward highly available DBMS. First, the effectiveness of existing database security mechanisms, such as integrity constraints, embedded rules in transactions, SQL and OS-based access control etc., is reaching a plateau. Consequently, it is becoming more and more difficult and expensive to further improve the MTTF. Comparatively, the MTTR minimization approach receives relatively less attention. Second, it is generally believed that there is no such thing as an unbreakable system. For example, attacks based on social engineering, password stealing, or insider information, are almost impossible to prevent. When these attacks occur, the best one can hope for is to restore the system back to normal operation as quickly as possible.

After an intrusion, currently the only option that database administrators have is to manually determine the "damage perimeter" and erase all the identified corrupted data. Unfortunately, given the highly complex interactions among database transactions, especially in a high-volume E-service site, it is next to impossible to determine how the transactions issued by intruders impact the database before the intrusion is detected. As a result, a common approach that database administrators take to repair intrusion damage is to roll the database back to where it was before the start of the first intruder transaction. While this approach is conceptually simple and does eliminate all damage caused by malicious transactions, it also throws away all the useful work produced between the occurrence of an attack and its detection.

A more desirable alternative is to determine the exact extent of intrusion damage, and undo only those transactions that are considered corrupted by the attack. *Phoenix* takes exactly this approach both to increase the amount of useful work preserved across intrusion and to speed up the repair process. More specifically, *Phoenix* tracks and maintains inter-transaction dependency at run time to determine the exact extent of damage caused by an intrusion, and exploits standard database logs to support single-assignment update semantics and thus the ability to roll back committed transactions.

We have built a fully operational *Phoenix* prototype based on PostgreSQL, which was chosen because *Phoenix* requires modifications to the DBMS kernel. PostgreSQL is currently the most advanced open-source DBMS with transaction support and is now a part of Red Hat's Linux distribution. It supports all the features of a fully functional commercial DBMS including ACID guarantees for transactions, write ahead logging, complex object types, ODBC/JDBC interfaces, etc.

The rest of this paper is organized as follows. In Section 2 we review previous research in the area of intrusion-resilient data systems in general, and intrusion-resilient DBMS in particular. Section 3 describes the system architecture of *Phoenix*. Section 4 presents the software architecture and implementation details of the current *Phoenix* prototype. Section 5 provides the results of a detailed performance study of the *Phoenix* prototype based on the TPC-C workload. Section 6 summarizes this paper with its main contributions and an outline of on-going work.

2 Related Work

Oracle 9i supports a feature called Flashback query [1] that allows end users to post a query against a database state at a particular point in time in the past. This feature is built upon the WAL log and can be used for building self-correcting applications that need to undo effects of certain transactions without point-in-time rollback. The Ingres database system from Computer Associates Inc. supports point-in-time rollback and roll-forward using journal-based recovery [2]. In this approach, journal files containing changes to database tables are maintained and analyzed to reconstruct the database state at the time point in question. Although these two commercial database features provide nice "history query" capabilities, they themselves do not allow intrusion tolerance for two reasons. First, they do not support *selective* undo of database transactions that take place between the current time and the time point of interest. Second, they do not solve the problem of determining which transactions to undo, or the extent of database damage due to an attack.

There have been several research projects on post-intrusion database damage repair. Ammann et al. [4] proposed a transaction model and associated protocols that allow normal transactions to proceed against a database whose portions are known to be damaged as a result of an intrusion. The proposal is largely a theoretical exercise without detailed system-level considerations. Peng Liu et al. [5] [6] described a concrete intrusion tolerant database system called ODAM, which can continue its transaction processing service even in the presence of active attacks. ODAM logs database updates in terms of SQL-based transactions. Instead of keeping track of inter-transaction dependencies at run time as *Phoenix* does, ODAM identifies them at the repair time by analyzing the SQL log. To support continuous operation, ODAM incorporated several concurrency control schemes to detect, assess and repair damaged databases on the fly without completely stopping the processing of new incoming transactions. However, during the repair time, the effective throughput of the DBMS is degraded. ODAM requires both a write log and a read log. Although a write log is quite common in modern DBMS, maintaining a read log poses a serious performance over-

head and therefore is not supported in existing DBMS. ODAM obtains transaction read information by using predefined templates of reads for each transaction. These read templates are instantiated at the time a transaction runs by parsing the SQL statements associated with the transaction. In contrast, *Phoenix* maintains an inter-transaction dependency graph and thus does not need any read log for recovery purpose. Consequently, our approach does not require any prior knowledge of the transactions and is more accurate. At present, *Phoenix* only supports off-line post-intrusion database repair, but can be extended to support on-the-fly database repair with proper concurrency control as in ODAM.

The Repairable File System (RFS) project [14] aims to improve the speed and precision of post-intrusion damage repair for NFS servers. Traditionally, file system recovery uses signatures generated by systems such as Tripwire [17] to determine corrupted system files or complete point in time restoration from backups. Instead, RFS maintains file system operation logs and carries out dependency analysis to provide fast and accurate repair of damage caused by NFS operations issued by attackers. *Phoenix* applies the same set of principles used in RFS to build intrusion-resilient database systems.

Finally, it should be noted that *Phoenix* is different from standard database back-up systems in that it backs up every database update and keeps track of inter-transaction dependencies at run time. *Phoenix* is also different from data mirroring/replication systems because the latter simply make the same mistake twice in the presence of intrusion or errors, and do nothing to improve the accuracy or performance of post-intrusion damage repair.

The Recovery Oriented Computing (ROC) project [18] [19] advocates a radical shift from a performance-dominated research focus to the focus of improving system availability by reducing MTTR and eventually the overall system ownership cost. One of the applications that is currently being pursued, an undoable email system [18], shares a similar approach with *Phoenix*, but focuses specifically on email message protection rather than general databases.

The Dali system [21] from Bell Labs took a similar inter-transaction dependency tracking approach to eliminate transactions that have been corrupted by an application which runs in the same address space as the database. The Dali approach keeps track of the read-dependency set of each transaction and is mainly for main memory database management.

3 The Phoenix Approach

The ultimate goal of the *Phoenix* project is to develop a modular system that can be plugged into an existing DBMS and makes it intrusion-resilient without any modification to the host DBMS. Whether this goal is attainable depends on the availability and flexibility of the "hooks" existing DBMSs support. Fundamentally, to convert a standard DBMS into one that is intrusion-resilient, one needs to add to the DBMS the following capabilities:

- The ability to maintain the *before images* of all database updates so that each database update is undoable, and
- The ability to keep track of the dependencies among transactions, so that it is possible to determine precisely the extent of database damage induced by one or a set of attack transactions.

3.1 Maintaining Before Image

Every modern DBMS supports database transactions, and maintains some sort of before image on disk for every database update, at least after the associated transaction is committed. In write-ahead logging, for example, the undo/redo log records associated with a transaction must be written to disk before the transaction can be committed. Theoretically *Phoenix* can use these "undo" log records as before images in the damage repair process. However, there are two issues that make the picture less than ideal. First, undo log records are usually kept for a shorter period of time than *Phoenix* needs. For example, if the *protection window* of *Phoenix*, which is the maximal interval between an attack and its detection that the system allows and still is able to support lossless repair, is set to one month, undo log records need to be kept for one month. Although these undo records could be kept in the archive log to survive disk failure, increasing use of disk mirroring and replication renders archive logs less and less popular in practice.

The second issue is that there is no standardized programming interface to access the transaction log and to apply the log records therein to undo already committed transactions. One possibility is to automatically derive a compensating transaction for a transaction that needs to be undone, and submit the compensating transaction to the DBMS through its standard access interface. This way no modification to the DBMS is required to support transaction rollback.

3.2 Tracking Inter-Transaction Dependency

Today, to repair intrusion damage on a database, administrators have no choice but to *manually* go through the set of transactions that take place between when an attack occurs and when it is detected, if they need to eliminate the effects of those and only those transactions that are affected by the attack. This is obviously a labor-intensive, error-prone, and time-consuming process, and is also why MTTR for database damage cannot be significantly reduced. *Phoenix* addresses this problem by keeping track of inter-transaction dependency at run time; therefore at repair time, *Phoenix* can accurately and automatically deter-

mine the set of transactions that are potentially corrupted by the intruder's transaction(s). Presented with these transactions identified by *Phoenix*, database administrators can either choose to refine them further or to rollback their effects directly.

The notion of inter-transaction dependency is far from obvious. The most sophisticated definition of transactional dependency calls for the semantic analysis of the applications in which transactions are embedded. For example, suppose Transaction A is embedded in Application 1 and Transaction B is embedded in Application 2. At run time Transaction A retrieves a record that is last updated by Transaction B. In this case, although there is an inter-transaction dependency between Transaction A and Transaction B, there may not be a real dependency between Application 1 and Application 2, because Application 1 never uses the record that Transaction A retrieves in its application logic. Therefore, this inter-transaction dependency can be safely ignored.

Conversely, even when two transactions do not access common records, it does not necessarily mean that there is no dependency between them, because it is possible for an application to access Record X through Transaction A, compute a value from Record X, and use the resulting value to update Record Y through Transaction B. In this case, although Transaction A and B do not share any data, there is an implicit inter-transaction dependency between Transaction A and Transaction B through the common embedding application. In the most extreme case, Transaction A is embedded in a different application than Transactions B, but these two embedding applications have data dependencies through inter-process communication.

The inter-transaction dependency that the current *Phoenix* prototype supports is based strictly on read-write relationships between transactions: There is a dependency from Transaction A to Transaction B if Transaction B reads a data item last updated by Transaction A. The assumption underlying this definition is that reading a particular item changes the state of a transaction and affects its execution results.

To compute inter-transaction dependencies, one can maintain for each transaction a read/write log, which records the set of records that are read and written by that transaction at run time, and perform dependency analysis at repair time. While conceptually simple, it has a major flaw: most DBMSs do not support read logs because keeping read logs incurs serious performance penalty. Recognizing this deficiency, *Phoenix* chose to build up inter-transaction dependency directly at run time, thus eliminating the need of read logging completely.

For each data item, *Phoenix* maintains an additional field that records the last transaction that updates the data item. Let's call this field of data item X as *Prev(x)*. When X is read by Transaction T, then *Phoenix* establishes a dependency relationship between T and the transaction indicated in *Prev(x)*. This algorithm is illustrated in Figure 1. For example, after the first transaction, Prev(1) = 0; once the second transaction completes, transaction 1 is dependent on transaction 0, as shown in the dependency graph that *Phoenix* builds on the fly.

Because inter-transaction dependencies need to survive intrusion, they should be kept persistent and preferably in a separate disk than other database disks. Specifically the dependencies associated with a transaction should be written to disk when the transaction commits.

Concurrent execution of transactions complicates the maintenance of inter-transaction dependency. First, if T just updates x, changing *Prev(x)* to T should happen before another transaction reads x. *Prev(x)* update should be atomic with respect to the actual update of x. Secondly, if a transaction T writes x and later aborts, *Prev(x)* should be reset to its previous value before T's update. This can be accomplished by considering *Prev(x)* information to be a part of the before image of an item x and then carrying out necessary changes at transaction rollback time. Thirdly, if transactions against a database system are allowed to execute at *read uncommitted* isolation level, cascaded aborts are possible due to dirty reads. In this case *Phoenix* must carefully remove any dependency information associated with the aborted transactions and also restore *Prev(x)* to it's earlier values. Handling aborts becomes considerably easier if all transactions are guaranteed to run at at least *read uncommitted* isolation level. We discuss these issues further in Section **??**.

The choice of the granularity of a data item for which the *Prev()* field is maintained affects the run-time performance overhead and the accuracy of dependency tracking. Coarser granularity results in lower performance overhead but a higher probability of *false sharing*, i.e., two transactions that read and write different portions of the same data item. From an implementation's standpoint, the definition of a data item must satisfy the following two properties:

- The read/write of a data item must be atomic so that the *Prev()* field update is atomic, and
- There should be a unique way to refer to an item so that undo can be done at the same granularity.

A table row satisfies both properties and thus is chosen as the basic unit for tracking inter-transaction dependencies. Accordingly a data item read operation is defined as either of the following: (1) read access to a row satisfying WHERE or HAVING clause in a select operation, or (2) read access to a row satisfying WHERE clause in update/delete operation. To a given transaction, a row is read only when the select statement returns a row or when a row is chosen to be updated/deleted by an update/delete statement. Note that this definition excludes those rows that are read *during* the processing of a select operation or a delete/update operation. That is, if the DBMS reads a set

OPERATION : r(x) / w(x) = read/write of item x

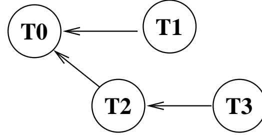
T3 : r(3) commit

T2 : w(3) r(1) w(4) commit

T1 : r(1) r(2) commit

T0 : w(1) commit

DEPENDENCIES :




```
OnRead (Item x, Transaction t) {
    LastToUpdate = prev(x);
    if ( t does not depend on LastToUpdate)
        AddDependency (t, LastToUpdate);
}

OnWrite(Item x, Transaction x) {
    Set prev(x) = x;
}
```

Figure 1: *Creating dependency graphs from transaction history. Transaction 1 becomes dependent on Transaction 0 after reading item 1 that is last updated by Transaction 0. The algorithm on the right describes how the graph on the left is created and maintained.*

of rows in order to service a select query, only the rows that satisfy the query are considered read by the transaction that issues the query. Similarly a write to a table row is defined as an update (insert) statement, which modifies (inserts) that row.

4 Prototype Implementation

The implementation of *Phoenix* adheres to the following principles. Firstly, the addition of intrusion resilience should be safe in the sense that it does not affect normal transaction execution semantics even upon system failures. Secondly, the performance and space overhead of *Phoenix* should be kept to the minimum, preferably to a level that is completely transparent to database users. Finally, database administrators should be able to enable/disable *Phoenix* easily.

Logically, *Phoenix* consists of two components: a run-time component that builds up inter-transaction dependencies during normal operation, and a repair-time component that computes the set of corrupting transactions and performs undo of their effects after an intrusion is detected. These two components will be described in more detail in the following subsections.

The current *Phoenix* prototype is built on PostgreSQL, an open-source database management system derived from Berkeley's Postgres project. We chose PostgreSQL as the underlying platform because it is open-source, it provides industry-strength transaction support, and most importantly the fact that its multi-version concurrency control and no-overwrite storage management are a perfect fit with *Phoenix*. In the next subsection, we will briefly describe these two PostgreSQL features to set the stage for subsequent discussion.

4.1 Multi-Versioning Record Structure in PostgreSQL

PostgreSQL uses a no-overwrite storage management policy to speed up the transaction processing. In this scheme, every update to an existing table row creates a new version of the row. Under this policy, both the before image and after image of each database update are implicitly stored in the database, and consequently there is no need for storing undo information in a separate log. This essentially means that a part of the standard write-ahead log (before images) is implicitly stored as part of the database records. This results in faster transaction abort handling and recovery as there is no undo log to process.

With this database record structure, PostgreSQL chooses to use a multi-version concurrency control (MVCC) scheme to improve the transaction concurrency. In MVCC, the concurrency control granularity is a table row; a read access and a write access to the same table row do not block each other; only a write access can block another write access to the same table row.

In PostgreSQL, a table row contains one or more tuples, each representing a distinct version of the table row. At run time, when a transaction accesses a table row, the tuple it actually sees depends on the *visibility* rule. Each tuple of a table row comes with a metadata that consists of three fields: *Xmin* contains the ID of the transaction that creates this tuple, *Xmax* contains the ID of the transaction that updates or deletes this tuple, *Link* points to the next tuple of the same table row. When a transaction with an ID of N accesses a table row, PostgreSQL returns the tuple of this table row that has a valid Xmin and invalid Xmax with respect to N.

In *read committed* isolation level, the ID in Xmin or Xmax is valid with respect to N if and only if it is N or it denotes a transaction that is already committed. In *serializable* isolation level, the ID in Xmin or Xmax is valid with respect to N if and only if it is N or it denotes a transaction that is already committed before Transaction N starts.

In visibility rule processing, PostgreSQL needs to know whether a transaction is already committed and when. For this purpose, it keeps a separate transaction status log, each entry of which corresponds to a transaction and indicates whether the transaction is in progress, committed, or aborted. To ensure correctness, the tuples updated by a transaction must be written to disk before its entry in the transaction status log can be changed to "committed."

Multiple tuples per table row combined with visibility

rule processing result in both a powerful framework for both concurrency control and error recovery. For example, during recovery, to abort a previously in-progress transaction, all that is needed is to convert the transaction's corresponding entry in the transaction status log to "aborted". Once this is done, none of the tuples it creates will be visible to any transactions according to the visibility rule. There is no need for any redo, because if a transaction is considered "committed" in the transaction status log, all the tuples it creates must be written to disk already.

From the standpoint of *Phoenix* implementation, PostgreSQL provides two key benefits. First, the *Xmin* field of every tuple provides the same information as *Prev()*. Second, the extremely simple way of aborting a transaction allows *Phoenix* to undo a committed transaction without making any modification to PostgreSQL's source code.

4.2 Run-Time Logic

4.2.1 Recording Last Update Transaction

To keep track of inter-transaction dependencies at run time, PostgreSQL needs to record the ID of the transaction that last updates each database record. There are two ways to implement this functionality. The first approach is tied with PostgreSQL's multi-versioning record structure. When a transaction accesses a table row, PostgreSQL returns one of the row's tuples according to the visibility rule; the *Xmin* field of this tuple gives the ID of the transaction that creates the instance of the table row that the current transaction is accessing. The additional space usage is only 4 bytes (size of a unique transaction ID) per row and the time cost is that of updating this 4-byte field each time a row is written. By exploiting the MVCC data structure inherent in PostgreSQL, no additional code is required for *Phoenix* to keep track of the last transaction that updates a given table row.

If there is no per-record *Xmin* field, then an alternative approach for maintaining *Prev()* information is based on triggers. More specifically, we create a system table called *CreateTran* that contains two columns : row OID and transaction ID. A tuple (o,x) in *CreateTran* indicates that the row with OID o is last updated by a transaction with ID x. To take control when a table row is updated, one can set up a create/delete/update trigger for every database table. The handler associated with a table's trigger will be invoked whenever a row in the table is created, deleted, or modified. Moreover, the handler can access the row in question. Using the trigger mechanism, we can modify the transaction ID field of the *CreateTran* entry corresponding to the row in question to point to the current transaction. The *CreateTran* table is implemented as an in-memory hash table with row OID as the key. On both read and write access to a table row, the system consults with the hash table with the row's OID, and updates the entry or builds up inter-transaction dependency accordingly. Because the hash table is memory-resident, the cost to access it is much smaller.

4.2.2 Capturing Inter-Transaction Dependency

In *Phoenix*, an inter-transaction dependency is constructed whenever a transaction reads a table row that is last updated by another transaction. Conceptually one can set up a read trigger for each database table and implement this dependency tracking logic in the associated trigger handler. Unfortunately, unlike write triggers, most DMBSes do not support read triggers. Even when they do, the read trigger semantics may not be an exact fit of what *Phoenix* needs and the associated performance overhead may be excessive. As a result, *Phoenix* has no choice but to modify PostgreSQL's internals to construct inter-transaction dependencies at run time.

Ideally, *Phoenix* needs to intercept the query execution at the point when a table row satisfying the WHERE or HAVING clause in the query is returned for further processing. This requirement relates to our definition of read given in section 3.2. The execution plan tree for a query is a binary tree comprising nodes that perform certain tuple producing operations. For example, an index scan node can use a key and a BTree index to return matching rows, a join node can merge-join the tuples produced by it's left and right subtrees, etc. Query execution proceeds in an iterative manner by a parent node asking it's child nodes to produce a tuple for it to operate on. Each returned tuple is operated upon in a node-specific manner. This process continues until child nodes indicate that there are no more tuples. The execution of a query plan is initiated by the root node requesting a tuple from it's children and ends when the children return a *null* tuple. For SELECT statements the root node will return the tuples it receives to the PostgreSQL front end. For UPDATE statements, the root node will perform the required update on the received tuples. For DELETE statements, the root node will expire the received tuples. The tuples that are operated upon by an execution plan tree are in fact copies of the actual on-disk tuples, instantiated at the lower levels in the plan tree. We refer to them as *cooked tuples*. The data portion in cooked tuples may change as they pass through projection or join nodes (cooking) while the meta-data portion usually remains the same. Not every cooked tuple present at lower levels reaches the root node as some of them are discarded during join operations, intersect operations etc.

In order to add a dependency, *Phoenix* intercepts the query execution plan when it identifies the occurrence of a read. At this point, it calls a hook function to add a dependency between the *current transaction* and the *last update transaction* for the tuple in question, before allowing query execution to proceed as normal. The point of interception in query execution is the code associated with processing the root node. Intercepting at the root node suffices as all rows returned to the root node for SELECT/UPDATE

statements will match the WHERE (or HAVING) clauses. Thus, we can call our dependency-adding hook function each time the root node's children return a tuple. However, prior to adding a dependency during interception, we also need to obtain the row ID(s) of the tuple as they are needed for determining *last update transaction*. The row ID information needed at the root level is maintained as part of the cooked tuples. It is possible for a cooked tuple to be based upon more than one original disk tuples. For instance, if the SQL query involves a join of 5 tables, the resulting tuple is a linear composition of some rows from these 5 tables. Due to this we need to maintain a list of row IDs for each cooked tuple. This list starts off as a single element when the cooked tuple is instantiated (row ID of disk based tuple) and will grow by one element at each join node. As no deletions or traversal of this list (except at the root node) are required we can maintain it with little overhead. It should be noted that size of this row ID list is usually quite small but such lists exist as meta-data for each of the cooked tuples in the final result of a query. Once the row ID information is available at the root node, *Phoenix* uses the *Prev()* information to find out what transaction(s) last updated the row in question and adds one or multiple dependencies.

4.2.3 Keeping Inter-Transaction Dependency Graph Persistent

Maintaining the inter-transaction dependency graph is a potential performance bottleneck for *Phoenix* as one needs to update and query the graph on each read during query execution. Further, this graph is maintained as a shared data structure operated upon by multiple transactions and thus efficiency is even more critical. We exploit two properties of the operations against the inter-transaction dependency graph to improve its run-time efficiency. Firstly, traversal of the graph is not required while *Phoenix* is operating. It is only during the post-intrusion phase that we need to analyze the dependency graph in detail. Secondly, deletion of edges from the graph can be carried out in an asynchronous manner and even deferred to the post intrusion phase if space is not a concern (see Section 5). This is because deletions need to be performed only when a transaction aborts. We can process a transaction abort event by simply setting a flag in the corresponding node of the graph and taking appropriate action at analysis time.

We implement the inter-transaction dependency graph as a hash table of hash tables. This is stored in the DBMSs shared memory area and thus is accessible to all backend server processes, each of which services a SQL-based transaction. The top-level hash table (Table-A) is indexed by a transaction ID. Each entry in Table-A is associated with one transaction and is a hash table itself (Table-B). Table-A is a store of dependencies for all transactions whereas Table-B is for dependencies associated with a particular transaction.

To record the dependency from Transaction X to Transaction Y, the Table-B associated with X is retrieved through a hashed access to Table-A using X as the key. Next, Y is added to the Table-B of Transaction X through another hashed access to Table-B using Y as the key. To check if Transaction X depends on Transaction Z, we retrieve the Table-B associated with X and verify if Z is present in that Table-B.

The part of the inter-transaction graph that describes what a transaction depends on is made persistent when the transaction is committed, at which time the Table-B associated with this committed transaction is written to a disk-based log file. This log file contains one record per transaction. Each record starts with three fields: *TransactionID, numberDepends, and needsUndo*, and is followed by *numberDepends* transaction IDs. The *needsUndo* flag is set if the transaction does not consist of only read queries; transactions that just read are not recorded in the write log and do not need undo. Finally, once a transaction's Table-B is written to disk, the in-memory version is discarded.

4.3 Repair-Time Logic

After an intrusion is detected, *Phoenix*'s repair-time logic is invoked, which consists of two phases: identification of the undo set and erasing the effects of the transactions in the undo set on the database. *Phoenix* first re-builds the inter-transaction dependency graph from the persistent dependency log file. Given the transactions that are identified as initiated by the attacker, *Phoenix* traverses the inter-transaction dependency graph to find all subsequent transactions that directly or indirectly depend on the attacker transaction(s). These transactions form the initial undo set.

Recognizing that the transaction dependency definition currently supported may not be the most appropriate one, *Phoenix* provides a tool for database administrators to refine the initial undo set into the final undo set. The basic operation supported by the tool is called *Depends(T)* which returns the set of transactions depending on T. *Depends(T)* is simply the result of the reachability algorithm with T as the start node. A DBA can delete certain transactions from the undo set, combine two undo sets, add elements to an undo set, etc. Therefore the DBA has considerable flexibility in finalizing the eventual undo set. The dependency graph query tool allows a DBA to mix and match his intuition and knowledge about transaction semantics along with the automated derived undo set. This interactive refinement facility is essential because it provides database administrators a sense of control and flexibility in the database damage control process.

Assume the set of transactions that take place between the time when an intrusion occurs and the time when it is detected is T_{total} and the final undo set is T_{undo}. Then logically what *Phoenix* needs to do to repair the intrusion

damage is to rollback all transactions in T_{undo}, and to roll forward those transactions that are in T_{total} - T_{undo}. The roll-forward step is necessary because it is what distinguishes *Phoenix* from other backup-based damage repair approaches: the ability to selectively undo only corrupting transactions and thus preserve as much useful work as possible.

Owing to the visibility rule logic in PostgreSQL, aborting a transaction is extremely simple: Just change the status of the transaction in question to "aborted" and all the tuples that this transaction creates immediately become invisible. With this powerful primitive, all *Phoenix* needs to do to repair the damage caused by an intrusion is to convert the status of all the transactions in T_{undo} to "aborted" in the transaction status log. There is no need to explicitly roll forward transactions in T_{total} - T_{undo}, because the effects of these transactions will become visible automatically through the visibility rule logic.

Recall that the protection window of *Phoenix* is the longest interval between an intrusion and its detection that *Phoenix* can tolerate and still provide perfect repair. Suppose the protection window is set to one month. This means that PostgreSQL needs to maintain table row versions and transactions status log entries that are younger than a month. PostgreSQL provides a vacuum command that can be run periodically to reclaim storage allocated to tuples that are outdated by other committed transactions. To ensure that tuples and transaction status log entries are properly preserved for recovery, it is required that vacuum should not be run during the protection window.

For other DBMSs that support standard write-ahead logging, both the rollback and roll-forward steps are needed. In rollback, one scans the write-ahead log backwards one record at a time. If the current record belongs to a transaction in T_{undo} and is not a start transaction record, it is undone. If the record belongs to a transaction in T_{undo} and is a start transaction record, the transaction is deleted from T_{undo}. This process continues until T_{undo} becomes empty and there are no more transactions to be undone. Next, in roll-forward, one scans ahead through the write-ahead log until the end of the log. If a record belongs to T_{undo} it is skipped. Otherwise, it is applied to the database.

5 Performance Evaluation

In this section we present the result of a performance evaluation study of a fully operational *Phoenix* prototype. The testbed machine for all the following experiments is a Dell Dimension Machine with a 1.8Ghz Intel Pentium 4 CPU, a 512K L2 cache, 1GB RAM and two 60GB hard-drives and running Red-Hat Linux version 7.1. Our experiments are based on the TPC-C benchmark with a warehouse factor of 8, unless specified otherwise. The TPC-C benchmark is designed to reflect real world OLTP processing activity and models a wholesaler business. The benchmark specifies five different types of transactions that are executed against a populated database. Each transaction type has a specific profile in terms of the number of reads/writes performed and the frequency of execution. We implement the TPC-C transactions in C using the libpq library provided by PostgreSQL.

Although the current *Phoenix*prototype successfully incorporates intrusion resilience, it is important to demonstrate that *Phoenix* does this without introducing undue performance overhead at run time. Logically, *Phoenix* adds additional code to keep track of the last update transaction for each table row, to build up inter-transaction dependency upon read access to a table row, and to put the inter-transaction dependency graph to disk at transaction commit time. To demonstrate that the run-time overhead of *Phoenix* is quite reasonable, we performed two experiments to evaluate its impact on transaction response time and throughput.

In the first experiment we measure the average transaction response times of *Phoenix* and PostgreSQL. The database cache was warmed up initially by executing all TPC-C transactions 10 times. The execution time of each transaction is recorded and averaged over 100 runs. The current *Phoenix* prototype implements two ways to keep track of inter-transaction dependencies: the MVCC approach and the Trigger approach (Section 4.2.1). The average per-transaction execution latencies of the five types of transactions in the TPC-C benchmark with and without inter-transaction dependency tracking are shown in Table 1. The Trigger and MVCC columns correspond to the trigger-based and the MVCC-based approach toward dependency tracking, respectively, whereas the PostgreSQL column corresponds to the case of no dependency tracking. The *Phoenix* Overhead column shows the percentage overhead difference between the MVCC version of *Phoenix* and PostgreSQL.

The main additional performance overhead of the MVCC approach with respect to generic PostgreSQL is due to the construction and storage of inter-transaction dependency graph upon read accesses, because PostgreSQL already records the ID of the last update transaction for each table row. Therefore, the performance difference between PostgreSQL and the MVCC approach should increase with the number of read accesses in the transactions. This explains why the execution latency difference between the MVCC approach and PostgreSQL is the largest for Stock Level transactions, which are read-only transactions, then Order Status, medium read-only transactions, and the smallest for Payment, light read/write transactions. Although New Order transactions also involve heavy read accesses, the relative performance difference between PostgreSQL and the MVCC approach is smaller compared to that of Stock Level transactions, because the writes in read/write transactions dilutes the cost of dependency tracking by significantly increasing the overall trans-

Transaction Type	Access Characteristics	Trigger based (ms)	PostgreSQL (ms)	MVCC (ms)	Phoenix Overhead
Payment	Light Read/Write	179.1	128.6	129.7	0.83%
Delivery	Medium Read/Write	4321	1787	1820.9	1.89%
New Order	Heavy Read/Write	30123	5840.9	5970	2.21%
Order Status	Medium Read Only	273	165.4	172.2	4.11%
Stock Level	Heavy Read Only	589.4	217.8	228.4	4.89%

Table 1: *Average execution latencies in milliseconds of different types of transactions in the TPC-C benchmark when executed with/without inter-transaction dependency tracking added by Phoenix. The Trigger and MVCC columns correspond to the trigger-based and the version-based approach, respectively, to dependency tracking, whereas the PostgreSQL column corresponds to the case of no dependency tracking. The Phoenix Overhead column shows the percentage overhead difference between MVCC and PostgreSQL.*

Throughput	W=4	W=8	W=16	W=32
Phoenix	911	776	735	681
PostgreSQL	993	850	793	740
Overhead	8.25%	8.7%	7.31%	8.0%

Table 2: *Overall transaction throughput comparison between the MVCC version of* Phoenix *and PostgreSQL under the TPC-C benchmark in terms of number of transactions executed per second. W is the number of warehouses.*

action latency without adding any extra dependency tracking overhead. This is also why a four-fold increase in latency from Delivery to New Order, only results in a corresponding increase in overhead of 0.32%.

The latency measurements in Table 1 demonstrate that the penalty in transaction response time that *Phoenix* introduced for the TPC-C benchmark is less than 5%. Table 1 also shows that the performance difference between the Trigger approach and the MVCC approach could be up to a factor of five, e.g., Delivery transactions. This is mainly due to the additional cost associated with access to the *CreateTran* table at run time.

In the next experiment we investigate how *Phoenix* affects the transaction throughput under the TPC-C benchmark. We simulated 400 simultaneous users that execute a transaction mix consisting of *45% New Order transactions, 43% Payment transactions and 4% Order Status, Delivery and Stock Level transactions*. Each user submits this transaction mix to the database server over a period of two hours. Table 2 shows the comparison in the transaction throughput between *Phoenix*-MVCC and PostgreSQL, in terms of number of transactions processed in one minute, which is the standard throughput metric for TPC-C benchmarks. The TPC-C specification models a warehousing business and allows scaling the database size by increasing the number of warehouses (W). These scaling rules are explicitly defined and addition of one row to the warehouse table results in a addition of 475,000 rows across the database.

Although the transaction throughput decreases as the database size is increased from W=2 to W=32, the performance overhead of *Phoenix* compared to PostgreSQL remains almost the same. Overall, *Phoenix* incurs a throughput penalty of around 8% for all database size when compared to PostgreSQL. This is because scaling has no effect on the number of dependencies that are being generated over the test run. Each transaction reads the same number of rows for a database with one warehouse as for a database with five warehouses. Thus the overhead associated with updating the dependency graph remains almost constant. Consequently, the resulting dependency graph has the same number of edges for all warehouse factors but its density may vary. The marginal drop in throughput performance can be attributed to the need to access a common disk based log file. The shared memory-resident dependency graph *does not* affect the overall transaction throughput, as no locking is required while accessing this graph and each transaction accesses it's own unique hash table. Further, allocating extra buffers in the shared memory region based on an estimate of the graph size ensures that the graph is always memory resident.

Finally, in terms of space overhead, *Phoenix* requires 4 bytes for each table row to record the ID of the last update transaction. Assuming at the average table row size is 50 bytes, this represents an 8% overhead. In the MVCC approach, this space overhead is 0% with respect to PostgreSQL because PostgreSQL maintains the Xmin field for free. The space overhead of the Trigger approach, on the other hand, is 16% because it uses a separate system table that requires an additional 4-byte row OID field. As for the inter-transaction dependency graph, assume that each transaction on the average adds 10 edges to the graph, and the DBMS executes 200 transactions per second, then the size of the inter-transaction dependency graph is $10 * 100 * 3600 * 24 * 4 = 3.456$ Gbytes per day, assuming each dependency graph edge costs 4 bytes. For a protection window of two weeks, this translates to less than 49 Gbytes, which can be easily accommodated by a 60-Gbyte IDE drive that costs under $150 as of July 2002.

6 Conclusion

As DBMSs become an integral component of Internet services, they are subject to the same type of attacks that have plagued their front-end servers. Although access control and authentication can a large extent fend off direct attacks,

they can do very little with respect to indirect attacks that go through compromised Web server or application servers, which DBMSs typically trust. Even for databases that are not visible in the Internet, they are still vulnerable to insider threats and other attacks that are based on non-technical means such as social engineering. While most database security research focuses on attack prevention, we take a different approach by borrowing ideas from the fault-tolerant computing community: To maximize the overall system availability, one can either minimize MTTR or maximize MTTF. As research on MTTF maximization starts to reach a point of diminishing return, it is essential to investigate the MTTR minimization approach for additional performance gain.

This paper describes the design, implementation, and evaluation of an intrusion-resilient database system called *Phoenix*, which both facilitates the post-intrusion system restoration process and improves the accuracy of database damage repair. It features a novel run-time inter-transaction dependency tracking mechanism that generates a data structure that allows *Phoenix* to quickly and automatically identify the corrupted transaction set whose effects on the database should be undone. To incorporate database administrators' inputs, *Phoenix* also provides an interactive exploration interface for them to further refine this undo set. One the final undo set is determined, *Phoenix* repairs the database by selectively rollback the transactions in the undo set. The current *Phoenix* prototype is built on PostgreSQL and intelligently exploits its no-overwrite storage structure and multi-version concurrency control to minimize the performance cost associated with inter-transaction dependency tracking and selective transaction undo. Performance measurements of the TPC-C benchmark on the fully operational *Phoenix* prototype show that the run-time overhead for inter-transaction dependency tracking is less than 5%, and the selective transaction undo is almost instantaneous. With this small run-time overhead, *Phoenix* can speed up the repair process by at least an order of magnitude compared to manual repair. Moreover, this fast repair advantage also carries over to database damages due to incorrect input entries or operational errors.

The proposed fast database damage repair technique closes a critical gap in the fault tolerance support of existing DBMS, in the sense that it provides a mechanism for a DBMS to quickly recover from application or operator errors. In contrast, existing DBMSi, even with support for mirroring and snapshot-based backup, can only protect the database from hardware failures and to a lesser extent software errors in the DBMS.

References

[1] Oracle 9i Flash Back query. Oracle Technet. Available at, http://technet.oracle.com/products/oracle9i/daily/ Aug13.html

[2] Caribou Lake, Journal Based recovery tool for Ingres. Available at, http://www.cariboulake.com/techinfo/irep_white paper.html

[3] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". TODS. 17(1): 94-162(1992)

[4] Peng Liu, Paul Ammann, Sushil Jajodia. "Rewriting histories: Recovering from malicious transactions". The International Journal of Distributed and Parallel Databases. 8(1):7-40, January 2000

[5] P. Luenam, P. Liu. "ODAM: An On-the-fly Damage Assessment and Repair System for Commercial Database Applications". Proc. 15th IFIP WG 11.3 Working Conference on Database and Application Security.

[6] P. Ammann, S. Jajodia, P. Liu. "Recovering from Malicious Transactions", IEEE Trans. on Knowledge and Data Engineering. To appear.

[7] Tripathy, Sani and Panda, Brajendra. "Post-Intrusion Recovery Using Data Dependency Approach". In Proceedings of the 2nd Annual IEEE Systems, Man, and Cybernetics Information Assurance Workshop, West Point, NY.

[8] T.F. Lunt "A Survey of Intrusion Detection Techniques". Computers & Security. 12(4):405 ? 418. June 1993

[9] D.E.Denning. "An intrusion detection model". IEEE Trans. On Software Engineering, SE-13:222-232. February 1987

[10] Dean Povey. "Enforcing well-formed and partially-formed transactions for UNIX". In Proceedings of the 8th USENIX Security Symposium. USENIX Association, August 1999.

[11] Dean Povey. "Optimistic security: A new access control paradigm". In Proceedings of the 1999 New Security Paradigms Workshop, September 1999.

[12] D. S. S et al. "Deciding when to forget in the Elephant File System" In Proceedings of the Seventeenth ACM Symposium on Operating System Principles, pages 110-123, 2002.

[13] J.D.S et al. "Self Securing storage: Protecting data in compromised systems". In Proceedings of the 2000 OSDI Conference, October 2000.

[14] Ningning Zhu, Tzi-cker Chiueh, "Design, Implementation, and Evaluation of Repairable File Service," in Proceedings of International Conference on Dependable Systems and Networks,San Francisco, CA, June 22nd - 25th, 2003.

[15] J.W et al. "Survivable Information storage systems". IEEE Computer, 2(1):61-68, August 2000.

[16] S. Quinlan and S. Dorward. "Venti: a new approach to archival storage". In USENIX conference on File and Storage Technologies, January 2002

[17] Home of the "tripwire open source project". http://www.tripwire.org/.

[18] David Patterson et al. "Recovery Oriented Computing ROC: Motivation, Definition, Techniques, and Case Studies", UC Berkeley Computer Science technical report, UCB//CSD-02-1175, March 2002.

[19] A. Brown and D. A. Patterson. "Embracing Failiure: A case for recovery-oriented computing (roc)". In 2001 High Performance Transaction Processing Symposium, October 2001.

[20] Kifer, Lewis, Bernstein. *Databases and Transaction Processing : An application oriented approach*. Addison-Wesley. 2002

[21] Philip Bohannon, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, S. Sudarshan, "Using Codewords to Protect Database Data from a Class of Software Errors," ICDE, p 276-285, 1999.

[22] CERT Coordination Center, "CERT Advisory CA-2003-04 MS-SQL Server Worm," http://www.cert.org/advisories/CA-2003-04.html