AuditGuard: A System for Database Auditing Under Retention Restrictions

Wentian Lu and Gerome Miklau
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{wen,miklau}@cs.umass.edu

1. INTRODUCTION

Auditing the changes to a database is critical for identifying malicious behavior, maintaining data quality, and improving system performance. But an accurate audit log is a historical record of the past that can also pose a serious threat to privacy. In many domains, retention policies govern how long data can be preserved by an institution. Regulations like FERPA and HIPAA (in the U.S.) or the Directive of Data Protection (in the EU), require strict retention periods to be observed, mandating the disposal of past data. In addition, institutions often adopt their own retention policies, choosing to remove sensitive data after a period of time to avoid its unintended release, or to avoid disclosure that could be forced by subpeona.

Policies that limit data retention conflict with the goal of accurate auditing, and data owners have to carefully balance the need for accurate auditing with the privacy goals of retention policies. Unfortunately, existing technologies make balancing these goals difficult. Most database systems include audit logs, but there are few mechanisms for limiting access to logs beyond wholesale destruction of the log for a given time period. Some proposed database systems support persistence, in which past states are retained. These can be used for some (but not all) audit functions, and also lack effective protection mechanisms for past versions of data.

The AuditGuard system is a flexible database audit mechanism which retains history *and* enforces retention policy. It allows an auditor to selectively remove data from a log of the past, while still retaining non-sensitive aspects of history that can be vital to an audit. The main components of the AuditGuard system are the following:

Historical data model AuditGuard implements a transaction time data model that stores past states of the database, along with an operational log which contains additional metadata. This data model enables the user to perform powerful audit queries over the past.[1]

Retention policies AuditGuard's distinctive feature is that it supports retention policies, which remove or protect past data from disclosure. For example, a typical policy might require that an employee's salary be removed from the database prior to 5 years ago, or that the records of employees working on Project X be removed from the history.

Incomplete history Applying such retention policies causes the (physical or logical) removal of individual data values or entire tuples from the historical record. Once retention policies have been applied, the AuditGuard system stores a partially *incomplete* record of history that respects the privacy goals of the policy.

Querying incompleteness A key feature of the AuditGuard system is the capability to evaluate audit queries over the protected, incomplete audit history that results from policy application. Naturally, some audit queries cannot be answered as accurately while respecting the retention policies. AuditGuard presents partial answers whenever possible, representing directly the uncertainty introduced by the retention policy.

The main benefit of AuditGuard is the enforcement of retention policies along with the ability to get audit query answers where none would be possible before. In particular, AuditGuard allows sensitive *values* to be removed, while retaining the *history of changes* to values for auditing. Such a sanitized audit log can meet many of the goals of the auditor by reporting on who made changes, when changes were made, and in what context they were made. At the same time disclosure of sensitive information can still be avoided.

2. AUDITING HISTORY

In this section we present the main components of the AuditGuard system by reference to an example scenario.

2.1 Historical Data Model

The AuditGuard data model stores, for each table of the user schema, a *transaction-time* table [8] and an *operational history* table. The transaction-time table records a complete history of tuples' state. The active time period of a tuple is indicated by the transaction time of creation (i.e. after insert or update) and the transaction time of deletion (or the special value `now` for active tuples). User modifications to the database are applied to the current snapshot only, and

[1]Note that the goal of AuditGuard is to audit *modifications* to the database. We do not audit read-only queries.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

name	dept	sal	from	to
Bob	Mgmt	10k	0	100
Chris	HR	15k	50	400
Bob	Sales	10k	100	200
Bob	Sales	12k	200	500
Chris	Sales	15k	400	450
Chris	Sales	12k	450	700
Bob	Sales	15k	500	now

(a) Transaction-time table T

tid	name	type	attr	user	ttime
1	Bob	ins	all	Jack	0
2	Bob	upd	dept	Chris	100
3	Bob	upd	sal	Chris	200
4	Bob	upd	sal	Chris	500
5	Chris	ins	all	Jack	50
6	Chris	upd	dept	Jack	400
7	Chris	upd	sal	Jack	450
8	Chris	del	all	Jack	700

(b) Operational history table H

Table 1: The logical schema for a sample Employee history.

never modify history. The operational history table records metadata about the update transactions that have been applied to the database. This table stores, for each transaction, descriptive fields which may include the authorization ID of the user issuing the transaction, the IP address of user, etc.

As a running example we consider auditing a simple Employee database with an original schema `Emp(name, dept, salary)`. Table 1a shows T, the transaction-time table recording the department and salary history for employees `Bob` and `Chris`. The columns `from` and `to` are added to the original schema. Table 1b shows H, the operational history table. We assume for this example that `name` is the primary key of T, that each operation is assigned a unique transaction time, `ttime`, and that each update operation in H changes exactly one tuple in the original table. The `attr` column in H indicates the attribute changed by an operation if it is an update query, or `all` if it is an insert or delete operation.

2.2 A Language for Auditing Queries

AG-SQL is an extension to SQL designed to support queries over the transaction-time and operational history tables. The formal semantics of our language (omitted due to space constraints) is based on extended relational algebra operators for temporal relations [4, 8, 11]. The following are examples of simple auditing queries that can be expressed over the Employee history described by tables T and H:

A1: *Return all users who have updated Bob's salary, along with the time of update.*

```
SELECT user, ttime
FROM H
WHERE name='Bob' AND type='upd' AND attr='sal'
```

A2: *Return the history of all employees no longer employed by the company.*

```
SELECT name, dept, sal, from, to
FROM T
WHERE name NOT IN(SELECT name FROM T(now))
```

A3: *Return the name and time of modifications for all employees whose salary was updated by Chris while Chris was not in the HR department.*

```
SELECT H.name, H.ttime
FROM TEMPORAL T, H
WHERE H.user=T.name AND H.attr='sal'
      AND T.name='Chris' AND T.dept!='HR'
```

A1, in fact, includes no temporal features, consisting of standard SQL. In such situations, the transaction-time tables listed in `FROM` are treated as traditional tables with additional time attributes. In **A2**, `T(now)` is the current snapshot of table T. Attributes `from` and `to` in the `SELECT` clause indicate that the result will be a transaction time table. In **A3**, the `TEMPORAL` identifier in the `FROM` clause declares a temporal join of tables.

2.3 Data Retention Policies

A retention policy specifies time periods for which data or documents should be retained, and limits on retention. The AuditGuard prototype supports two kinds of retention policies which are used to limit the lifetime of data in our historical data model.

The first is called **redaction**. When redaction is applied to an attribute value it removes the value, but does not hide its existence. Redaction can be applied to any combination of attributes, an entire tuple, or sets of tuples. The second operation is called **expunction**. When a tuple is expunged, its values are removed, along with a record of its existence. These two operators serve fundamentally different purposes as they enact *value* removal in one case and *existence* removal in the other. Our example scenario will focus on the redaction policy operator, as illustrated in these simple policies:

P1: Hide Bob's salary prior to time 200.

P2: Hide Chris's salary and dept prior to time 400.

Policies can be specified by the following SQL-like language. A policy can be evaluated like a `SELECT` query in SQL, but with the addition of a `WHEN` clause indicates the retention period. The language enables flexibility and expressiveness of policy customization, and policies can be freely combined.

```
EXPUNGE | REDACT attribute-list
FROM table-list
WHERE condition-list
WHEN time-condition-list
```

The meaning of an individual redaction policy rule on relation R, which is denoted $p_R(k, a, t)$, is as follows: for the tuple $\tau \in r$ with k as its value of key, all values of attribute a before time t should be removed from the database. It should not be possible for a user to infer these values from the database instance accessible after policy application.

2.4 Applying Retention Policies

Applying a retention policy to a database changes history, removing sensitive information. Removal can be physical (history is deleted from physical storage) or logical (history is inaccessible to some class of users, but still stored physically). The AuditGuard prototype currently implements physical removal, but logical removal is clearly important in order to support conflicting retention policies for multiple parties.

Whether removal is logical or physical, application of a retention policy results in an incomplete database, representing a set of possible worlds (or more precisely, a set of possible histories). The set of possible worlds is introduced by attribute variables and a tuple status indicator. The variables represent an unknown value from a given domain. The tuple status specifies whether a tuple exists in all the possible histories.

Applying this policy set $P = \{P1, P2\}$ to the database consisting of T and H, we get an incomplete transaction-time table T^P (shown as Table 2). The operational table H is unchanged[2].

In T^P, redacted values have been replaced by a set of variables $\{sx, sy, dx\}$. The domain of sx and sy is dom(`sal`); the domain of dx is dom(`dept`). Distinct variables in the same column are assumed to take distinct values. An additional column `status`, with domain $\{\texttt{p}, \texttt{c}\}$, has been added in the representation of T^P. A `status` of `c` means the tuple is certain to be present in the relation; `p` means the existence of the tuple is merely possible.

name	dept	sal	from	to	status
Bob	Mgmt	*sx*	0	100	c
Chris	*dx*	*sy*	50	400	c
Bob	Sales	*sx*	100	200	c
Bob	Sales	12k	200	500	c
Chris	Sales	*sy*	400	450	c
Chris	Sales	12k	450	700	c
Bob	Sales	15k	500	now	c

Table 2: Table after redaction T^P

In general, each policy set P will cause a series of removals on T, H (or both) over the specified time period. Policy application results in incomplete information tables [7, 1], T^P and H^P, which represent a set of possible worlds. A mapping of each variable present in T^P into an element of its domain defines a possible world in the expected way: all certain tuples must be in present, and any subset of the possible tuple may optionally be present. It is not hard to show that, for any policy P, the application of P is truth-preserving: the original database (T, H) is one of the possible worlds described by (T^P, H^P).

Policy application causes an unavoidable loss of information, but much of the information about the sequence of changes applied to the database may remain. The AuditGuard system can evaluate audit queries over the incomplete history and still return useful answers.

[2]In AuditGuard, there are other policies whose application will change both T and P, namely when a retention policy splits the active period of a tuple version.

3. AUDITING INCOMPLETENESS

AuditGuard is designed to support audit queries over the incomplete history resulting from retention policy application. Answers to our audit queries on an incomplete history have the expected semantics, representing the set of possible answers on the possible worlds. We can show that for the kind of incompleteness introduced by our policies, AG-SQL query answers can be represented concisely in our data model [5].

Returning to the three auditing queries discussed previously, Table 3 compares the answers to **A1, A2, A3** on the original and incomplete tables.

A1. Return all users who have updated Bob's salary, along with the time of update. Although policy $P1$ removes the salary information of Bob before time 200, we can still get the full answer to this audit query: Chris updated Bob's salary at time 200 and 500 respectively. Bob's private values are protected, but the record of changes to his data can still be audited.

A2. Return the history of all employees no longer employed by the company. This query answer represents certainty about the existence of employee Chris in the database at time 50, as well as a change to his salary at time 400. His department between time 50 and 400 is unknown – it may have been Sales, or it may have been changed to Sales at time 400. It is again possible to present an accurate history of Chris, without revealing sensitive values.

A3. Return the name and time of update for all employees whose salary was updated by Chris while Chris was not in the HR department. This query includes one certain tuple and one possible tuple. From the incomplete history it is clear that Bob's salary was updated by Chris at time 500 when Chris was not in HR. However, Bob's salary was also updated at time 200 by Chris, but Chris's department is unknown at that time. Therefore, (Bob, 200) is a possible answer for this query. The audit query contains some uncertainty, but provides a superset of the true answer.

4. IMPLEMENTATION

We implement AuditGuard by translating our historical data model into standard relations in Postgres. The prototype implementation includes the following:

Managing historical data. Each modification (insert, update and delete) on a user-defined table triggers a series of operations on the corresponding transaction-time table and operational log. The original persistence features of Postgres have been deprecated in recent software releases and are not used in our implementation.

Managing retention policies. AuditGuard provides a simple but powerful language to specify retention policies. As we discussed above, policies can be implemented logically or physically. AuditGuard currently supports physical removal which means expired history will disappear from the database after execution of the policy. AuditGuard translates the policy into update queries which will delete or modify some parts of history, introducing tuple-variables.

Implementing Query Evaluation. Query evaluation is implemented by transforming AQ-SQL queries into standard relational operators that can be optimized and executed

Original Results

name	ttime
Chris	200
Chris	500

After Redaction

name	ttime	status
Chris	200	c
Chris	500	c

(**A1**)

Original Results

name	dept	sal	from	to
Chris	HR	15k	50	400
Chris	Sales	15k	400	450
Chris	Sales	12k	450	700

Original Results

name	ttime
Bob	500

After Redaction

name	ttime	status
Bob	200	p
Bob	500	c

(**A3**)

After Redaction

name	dept	sal	from	to	status
Chris	*dx*	*sy*	50	400	c
Chris	Sales	*sy*	400	450	c
Chris	Sales	12k	450	700	c

(**A2**)

Table 3: The result of three audit queries, evaluated over the original database and the redacted database.

in Postgres. Standard transformations for accommodating temporal features in a relational database are adopted [11]. The extended relational algebra operators that are the basis of AG-SQL are evaluated using special comparisons that take into account tuple-variables in the instances. For example, conditions in the `WHERE` clause return TRUE or FALSE or POSSIBLE on an incomplete tuple. We follow existing work on maybe-operations [3] and incomplete temporal models [5] and tune the performance of our relational representation through query-rewriting and index selection for efficient evaluation in Postgres.

Demonstration Outline & Significance

The AuditGuard demonstration will present an example scenario in which the historical record of an institutional database is audited under retention policy constraints. The complete demonstration will: (i) Illustrate audit queries expressed as AG-SQL expressions, as well as redaction and expunction policies expressed as policy rules; (ii) Demonstrate the feasibility of query execution over the incomplete historical data model, and evaluate scalability with respect to database size, history size, and number of retention policies applied; (iii) Display incomplete answers to users.

As a result, the AuditGuard demonstration will provide insight into the interplay between retention policies and the accuracy of audit query answers, illuminating fundamental conflicts that exist between auditing and privacy as well cases where auditing can be performed without threatening unwanted disclosure.

5. RELATED WORK

Garcia-Molina et al. considered expiring tuples from materialized views in a data warehouse [6]. An administrator can declaratively request to remove tuples from a view, and the system will remove as much information as possible as long as it does not impact of views referencing the original view. Toman proposed techniques for automatically expiring data in a historical data warehouse while preserving answers to a fixed set of queries [12]. Skyt et al. consider vacuuming a temporal database [10]. Policies remove entire tuples, and the authors are concerned with the correctness of vacuum specifications, and mitigating actions to handle queries referencing missing information. The above works differ from ours because they do not consider cell-level removal, do not view the resulting database as an incomplete history from which possible answers can be derived, and do not consider an audit log accompanying the history. Recently, Ataullah et al. [2] have considered retention restrictions on complex business records that are defined, in their framework, by logical views in a database. A complete description of related work is included in our technical report [9].

Acknowledgements. This work was supported by NSF CAREER Award IIS-0738244.

6. REFERENCES

[1] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):159–187, 1991.

[2] A. Ataullah. A framework for records management in relational database systems. Master's thesis, University of Waterloo, 2008.

[3] J. Biskup. A foundation of codd's relational maybe-operations. *ACM Trans. Database Syst.*, 8:608–636, 1983.

[4] S. K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Trans. Database Syst.*, 13:418–448, 1988.

[5] S. K. Gadia, S. S. Nair, and Y.-C. Poon. Incomplete information in relational temporal databases. In L.-Y. Yuan, editor, *VLDB Conference*, pages 395–406, 1992.

[6] H. Garcia-Molina, W. Labio, and J. Yang. Expiring data in a warehouse. In *VLDB Conference*, pages 500–511, San Francisco, CA, USA, 1998.

[7] T. Imieliński and J. W. Lipski. Incomplete information in relational databases. *J. ACM*, 31:761–791, 1984.

[8] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental implementation model for relational databases with transaction time. *IEEE TKDE*, 3:461–473, 1991.

[9] W. Lu and G. Miklau. Log sanitization: Auditing a database under retention restrictions. Technical Report 22, Univ. of Massachusetts Amherst, June 2008.

[10] J. Skyt, C. S. Jensen, and L. Mark. A foundation for vacuuming temporal databases. *Data Knowl. Eng.*, 44(1):1–29, 2003.

[11] R. T. Snodgrass. *Developing time-oriented database applications in SQL.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1999.

[12] D. Toman. Expiration of historical databases. In *Symposium on Temporal Representation and Reasoning (TIME)*, pages 128–135, 2001.