Auditing a Database Under Retention Restrictions

Wentian Lu [1] and Gerome Miklau [2]

Department of Computer Science, University of Massachusetts
140 Governors Drive, Amherst, MA USA
[1]wen@cs.umass.edu
[2]miklau@cs.umass.edu

***Abstract*— 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. Policies which limit data retention conflict with the goal of accurate auditing, and data owners have to carefully balance the need for policy compliance with the goal of accurate auditing.**

In this paper, we provide a framework for auditing the changes to a database system while respecting data retention policies. Our framework includes a historical data model that supports flexible audit queries, along with a language for retention policies that hide individual attribute values or remove entire tuples from history. Under retention policies, the audit history is partially incomplete. We formalize the meaning of audit queries on the protected history, which can include imprecise results. We implement policy application and query answering efficiently in a standard relational system, and characterize (both theoretically and experimentally) the cases where accurate auditing can be achieved under retention restrictions.

I. 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 mandate the disposal of past data and require strict retention periods to be observed. For example, the Fair Credit Reporting Act limits the retention, by credit reporting agencies, of personal financial records. In addition, institutions and companies 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. Tons of examples show that failure in disposal of expired data might result in serious consequences such as discovery abuse in judgment and impact the bottom line of the business [1].

Retention restrictions conflict with the goal of accurate auditing, and data owners therefore have to carefully balance the need for accurate auditing with the privacy goals of retention policies. Unfortunately, current mechanisms for auditing and managing historical records have few capabilities for managing the balance between the two objectives. Obeying a retention policy often means the wholesale destruction of the audit log.

In this paper we propose a framework for auditing the changes to a database system in the presence of retention restrictions. We consider a historical data model and propose two kinds of rules for selectively removing or obscuring sensitive data from the record of the past. Despite the removal of information, it is often still possible for an auditor to monitor the record of actions taken on the database. We provide an overview of the motivation and contributions of this work through the following detailed example.

A. Example Scenario

We begin with a database storing tables belonging to a *client schema*. *Clients* interact with the database by submitting queries and updates, always on the current snapshot. In the running example used throughout this paper, the client schema consists of a single table, S, describing employees:

$$S(\underline{eid}, name, department, salary)$$

The *auditor* is responsible for monitoring access to the database and tracking down malicious actions after they have occurred. Auditors typically inquire about *what* happened to the database, *when* it happened, and *who* did it.[1] To enable the auditor to query the state of the database over time, the system maintains an audit log table, L_S, for each table S in the client schema. Each modifying operation, issued by a client on S, is recorded in L_S along with additional *audit fields* describing the **time** of modification, the **type** of modification (insert, update, delete), and any other fields possibly of interest to the auditor. Table I shows an audit log table including audit fields recording the name of the issuing **client** and their **IP** address.

The audit log can easily be converted to an alternative transaction-time representation. Table II shows such a table, denoted T_S. It represents the complete data history of the table, recording, in the **from** and **to** columns, the active period of each tuple in the database. Throughout the paper we will use both the log-based and transaction-time representations as they each have benefits for expressing queries and defining concepts.

These historical tables can support a variety of queries of interest to the auditor. Some simple examples include:

A1. *Return all employees who earned a salary of 10k at some point in time.*

A2. *Return the clients who updated Bob's salary, and the time of update.*

[1] We are concerned here with auditing *modifications* only. We do not audit queries that read from the database.

TABLE I

THE AUDIT LOG DESCRIBING THE HISTORY OF OPERATIONS PERFORMED ON A CLIENT TABLE WITH SCHEMA $S(\underline{eid}, name, dept, salary)$. COLUMNS **client** AND **IP** ARE AUDIT FIELDS.

client	**IP**	**time**	**type**	`eid`	`name`	`dept`	`sal`
Jack	1.1.1	0	ins	101	Bob	Sales	10
Jack	2.1.1	100	upd	101	-	-	12
Kate	3.1.1	200	upd	101	-	Mgmt	-
Kate	4.1.1	300	upd	101	-	-	15
Jack	1.1.1	0	ins	201	Chris	HR	8
Jack	2.1.1	300	upd	201	-	Mgmt	10
Kate	4.1.1	500	del	201	-	-	-

TABLE II

THE TRANSACTION-TIME TABLE DESCRIBING THE DATA HISTORY OF THE CLIENT TABLE. IT IS DERIVED FROM THE AUDIT LOG IN TABLE I

`eid`	`name`	`dept`	`sal`	**from**	**to**
101	Bob	Sales	10	0	100
101	Bob	Sales	12	100	200
101	Bob	Mgmt	12	200	300
101	Bob	Mgmt	15	300	now
201	Chris	HR	8	0	300
201	Chris	Mgmt	10	300	500

A3. *Return the clients who updated any employee's dept, and the time of update.*

Some audit queries are conventional queries over a transaction-time data model (such as A1). Others ask specifically about changes, and reference the special audit fields contained in the audit log (such as A2, A3).

The *compliance officer* is responsible for enforcing data retention restrictions arising from privacy regulations or institutional policies. These policies are typically non-negotiable – they must be respected by all users of the system, including the auditor. We propose two kinds of declarative retention rules for limiting the lifetime of data. Notably, these retention policies are expressed in terms of T_S, the transaction time table describing the data history. This is the most natural choice because retention policies refer only to the client schema, and to the notion of time.

Our first retention rule is called **redaction**. When redaction is applied to an attribute value, it removes the value but does not hide its existence. For example, a redaction rule may say: *Hide Bob's salary between time 0 and 250*. The second operation, called **expunction**, is more extreme. When a tuple is expunged, it is completely removed, along with all evidence of its existence. For example, an expunction rule may say: *Remove the record of all employees in the HR department between time 0 and 300.*

Applying a set of retention rules transforms the stored history of the database.[2] Table III shows a new transaction-time table, the result of applying the retention rules to the table T_S. In applying the redaction rule, salary values have been replaced with variables (**sx, sy**). We use variables as an alternative to NULLs in order to support more accurate auditing. Also note that there is an extra row in Table III because the time interval [200,300] in the original data has been split into two intervals: [200,250], in which Bob's salary is hidden, and [250,300], in which Bob's salary can be revealed to be 12k. In applying the expunction rule, Chris's membership in the HR department has been removed from the history: he is now only in the Mgmt department from time 300 to 500. For illustration purposes, the expunged row is included in Table III, but displayed with a gray background.

TABLE III

THE TRANSACTION TIME TABLE, TRANSFORMED UNDER THE FOLLOWING RETENTION POLICIES: REDACT$_S(name = Bob, \{salary\}, [0, 250])$ AND EXPUNGE$_S(dept = HR, [0, 300])$. (THE GRAY ROW HAS BEEN DELETED.)

`eid`	`name`	`dept`	`sal`	`from`	`to`
101	Bob	Sales	**sx**	0	100
101	Bob	Sales	**sy**	100	200
101	Bob	Mgmt	**sy**	200	250
101	Bob	Mgmt	12	250	300
101	Bob	Mgmt	15	300	now
201	Chris	HR	8	0	300
201	Chris	Mgmt	10	300	500

A main goal of this paper is provide a proper semantics for audit queries in the presence of retention policies. Because the transformed history has tuples removed by expunction and values obscured by redaction, the answers to audit queries may be uncertain or, in some cases, provide false information. We reconsider the previous audit queries under retention restrictions:

A1. *Return all employees who earned a salary of 10k at some point in time.*
This query is a straightforward selection on the transaction-time table. On the original data in Table II the answer to this query is {Bob, Chris}. On Table III, under the retention policy, the answer to this query includes Chris as a *certain* answer. However, Bob is only a *possible* answer because the predicate depends on the unknown value of variables **sx** and **sy**. Our implemented system returns both answers, labeled appropriately as possible or certain.

A2. *Return the clients who updated Bob's salary, and the time of update.*
The answer to this query on the original data is {(Jack, 100), (Kate, 300)}. The transformed history in Table III shows that Bob's salary definitely changed at time 100 (from **sx** to **sy**) and at time 300 (from 12k to 15k). In addition, it *may* have changed at time 250 (from **sy** to 12k), depending on the unknown value of variable **sy**. (Note that the uncertainty about this change is crucial – if it is possible to deduce that the change did not occur, then it is clear that Bob's salary was indeed 12k between 250 and 300, and the retention policy is violated.)
In order to fully answer the query, we must use the audit log to get the names of the clients who issued the update. Jack and Kate performed the updates at time 100 and 300, respectively, so the certain answers to this query are: {(Jack, 100), (Kate, 300)}. A subtlety here is how

[2]As a practical matter, retention rules may be applied physically, altering storage of the table, or logically, in which access is restricted but hidden data is still physically stored.

to return the possible answer for the update at 250, since there is no known client that performed that update. The possible answer that could be returned is: (NULL,250), but not if it reveals that this is a fake update.

A3. *Return the clients who updated any employee's dept, and the time of update.*
The answer to this query on the original data is {(Kate,200), (Jack,300)}, which can easily be computed from the original audit log L_S. In the transformed history in Table III we find evidence of only one update to the department field, at time 200. This is a result of the expunction policy that removed Chris' record from time 0 to 300. Thus, the answer to this query under the retention policy is {(Kate,200)} and the record of Jack's update is lost.

Notice that the answer to query A3 is incorrect: a tuple that is in the true answer (i.e. with respect to the original data) is omitted from the new answer. From the auditor's perspective this is a worse outcome than that of A1 and A2 where the true answer is one of the possible answers. One of the goals of our framework is to provide answers to auditor queries that, although imprecise, do not lead to false conclusions. Also note that in reasoning about the answers to queries A2 and A3 we referred to the transformed transaction-time table and used it to infer actions that were performed on the database. Later in the paper we make this process explicit by computing a *sanitized audit log*, consistent with the retention policies, that can be queried directly.

In summary, the main contributions of the paper are:

- We propose declarative rules for expressing retention restrictions over a historical data model. (Section III)
- We provide a precise semantics for audit query answers under retention restrictions, and we study theoretically the impact of retention policies on the accuracy of audit queries. (Section IV)
- We implement our framework as extensions to Postgres, showing that uncertain answers can be computed efficiently over our incomplete historical data model. (Section V)
- We demonstrate (through simulation on sample data) that useful auditing can be performed in the presence retention restrictions, despite uncertain answers. (Section VI)

Our work extends and integrates techniques from temporal databases, incomplete databases, and fine-grained access control into a flexible framework for controlled auditing. We distinguish our contributions from this work in Section VII.

II. Data Model and Audit Queries

In this section we describe our data model, based on backlog and transaction-time databases [2], [3], and our language for expressing audit queries.

A. Data model

Let $(S_1, \ldots, S_k)$ be the client schema. We refer to each relation S_i as a *regular* relation to distinguish it from transaction-time relations defined below. We use *tuples*(S_i) to refer to the set of all tuples that could occur in S_i (i.e., the cross-product of the attribute domains).

Audit Log: An audit log is a complete record of the operations on a client table over time, and we maintain an audit log table L_S for each table S of the client schema. Each row in L_S represents a transaction modifying a tuple of S. Table I shows an example audit log table. In general, the schema of L_S is:

$$(\langle \textit{audit-fields} \rangle,\ \textit{ttime},\ \textit{type},\ \langle \textit{client-fields-from-S} \rangle)$$

The *audit fields* may contain an arbitrary set of attributes describing facts about the transaction. In our examples, the audit fields record the name of the issuing **client** and their **IP** address, but in general they may include many other fields describing the context of the operation. $ttime$ is a time stamp, from a totally-ordered time domain $\mathcal{T}$, reflecting the commit time of the transaction. We assume each transaction receives a unique time stamp. The *type* field describes the modification as an insert, update, or delete. The fields of the client schema describe the changes in data values. If the transaction is an *insert*, each attribute value is included; for updates, only modified values are included, with unchanged attributes set to NULL; for deletes, all attribute values are NULL. This description of an audit log is essentially a backlog database [2] with the addition of audit fields.

We assume that each audit record refers to a unique tuple, identified by the key of the client table. In practice, a transaction may affect multiple tuples. If necessary, this relationship can be recorded in a statement-id, relating the changes to tuples made by a statement. Without loss of generality we omit this.

Transaction-time relation: A *transaction-time relation* (a *t-relation* for short) represents the sequence of states of a relation in the client schema. Formally, a t-relation over S is a subset of *tuples*$(S) \times \mathcal{T}$. A tuple $(p_1, \ldots, p_n, t) \in T_S$ represents the fact that tuple $(p_1, \ldots, p_n)$ is active at time instant t. In examples (and our implementation) we use the common representation for t-relations in which $(p_1, \ldots, p_n, \textit{from}, \textit{to})$ means that $(p_1, \ldots, p_n)$ holds at each instant t, for $\textit{from} \leq t \leq \textit{to}$. Table II is an example of a t-relation.

Audit log versus T-relation: Given an audit log table L_S, a unique t-relation can be computed from it in a straightforward way by executing each statement. After a modification, the values of a tuple are active until the time instant of the next operation modifying that tuple. We use *exec* to indicate this procedure, and we define T_S to be *exec*(L_S) for each S in the client schema.

It is also possible to reverse this procedure, computing an audit log from a t-relation (although no audit fields will be included). This procedure, denoted $\textit{exec}^{-1}$, computes initial insertion transactions at the time instant a new tuple is created, subsequent update transactions at the instant of each change to a tuple, and (for tuples that are no longer active) delete transactions. Notice that computing an audit log from T_S will reproduce a table similar to L_S but with the audit fields removed: $\Pi_{ttime,type,S}(L_S) = \textit{exec}^{-1}(T_S)$.

The audit log L_S and the t-relation T_S represent similar information. As a practical matter it is not necessary to maintain both. However, in the formal development presented here, each representation serves an important purpose. We will see in the next section that retention policies are defined in terms of T_S, and can be applied directly to T_S. But T_S does not include audit fields. We will also reconstruct an audit log from the protected T_S in order to make explicit the possible inferences about changes to the database.

B. Audit queries

A variety of interesting audit queries can be expressed over T_S and L_S. L_S is a regular relation, but queries over t-relation T_S may use extended relation algebra operators to cope with transaction-time . We omit a formal description of these operators, which can be found in the literature [4], [5], and instead present examples highlighting their features.

The example audit queries from Section I-A are expressed as follows on T_S or L_S:

A1. *Return all employees who earned a salary of 10k at some point in time.* $\Pi_{name}(\sigma_{sal=10k}(T_S))$

A2. *Return the clients who updated Bob's salary, and the time of update.*

$$\Pi_{client,ttime}(\sigma_{type=upd \wedge name=Bob \wedge sal \neq NULL}(L_S))$$

A3. *Return the clients who updated any employee's dept, and the time of update.*

$$\Pi_{client,ttime}(\sigma_{type=upd \wedge dept \neq NULL}(L_S))$$

Conventional joins on t-relations are possible, as well as joins between a t-relation and regular relation. For example, our audit log L_S can be joined with T_S on the *ttime* attribute. In addition, we can use *concurrent cross-product* (denoted $\times^\diamond$) or *concurrent join* (denoted $\bowtie^\diamond$) as binary operators on t-relations that combine tuples active at common time periods. The following example query includes a concurrent self join on T_S:

A4. *Return all employees who worked in the same department as Bob at the same time.*

$$\Pi_{name}(\sigma_{name'=Bob}(T_S \bowtie^\diamond_{dept=dept'} T'_S))$$

Finally, the *time-slice* operator restricts a t-relation to a specified interval in time. For interval $[m, n]$, it can be defined as: $\tau_{m..n}(R) = R \times^\diamond \{\langle m, n\rangle\}$ where $\{\langle m, n\rangle\}$ is a singleton t-relation without user-defined attributes. The result of applying the time-slice operator is a t-relation. A regular relation representing the snapshot database at time m can be written as $\pi_{S-\{\text{from,to}\}}\ (\tau_{m..m}(T_S))$.

III. Describing and Applying Retention Policies

In this section, we define the semantics of our redaction and expunction rules, and how they are applied to the stored history.

A. Retention policy definitions

Retention policies are used to restrict access to tuples or attribute values in one or more historical states of the database. The need for retention policies arises from the sensitivity of data items in the client schema. Thus it is most natural to express retention policies in terms of the t-relation, T_S, which describes states of the client relation as it evolves through time. We define our retention policies formally below as transformations on T_S.

Our first retention operation is called **redaction**. It suppresses attribute values in tuples for a specified time period. Redaction is useful because it hides sensitive data values, but preserves the history of modification of the tuple. Our second retention operation is called **expunction**. An expunged tuple is removed from history, and the historical record is modified accordingly to hide its existence.

These two operators serve different purposes as they enact *value* removal in the case of redaction, and *existence* removal in the other. Expunction is a more extreme operation because it does not merely suppress information, but changes the historical record in ways that can substantially change answers to audit queries. We believe that a variety of privacy policies can be satisfied through the use of redaction policies alone, which will lead to more accurate auditing.

In the definitions that follow, a Boolean condition ϕ, on client relation S, is a Boolean combination of comparisons $S.A\,\theta\,c$, or $S.A\,\theta\,S.B$, for any $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

Definition 3.1 (Expunction Rule): An expunction rule, over a client table S, is denoted $E = \text{Expunge}_S(\phi, [u, v])$ where ϕ is a Boolean condition on attributes of S, and $[u, v]$ is a time interval ($u, v \in \mathcal{T}$, and $u \leq v$).

An expunction rule asserts that all tuples matching condition ϕ should be removed from a specified interval in time. When an expunction rule E is applied to a t-relation T_S, the intended result is a new t-relation. Denoted $E(T_S)$, this new t-relation consists of all facts from T_S *except* those that satisfy ϕ and have time field in $[u, v]$:

Definition 3.2 (Expunction Rule Application): For a client relation S, let T_S be a t-relation over S, and $E = \text{Expunge}_S(\phi, [u, v])$ be an expunction rule. The application of E to T_S, denoted $E(T_S)$, is a new t-relation with the same schema: $E(T_S) = T_S - \{x \in T_S \mid \phi(x) \wedge x.t \in [u, v]\}$

Unlike expunction, a redaction rule does not remove tuples from the historical record. Instead, a redaction rule asserts that the values of certain attributes should be suppressed in all tuples that match condition ϕ and are active during a specified time interval.

Definition 3.3 (Redaction Rule): A redaction rule, over client table S, is denoted $\text{Redact}_S(\phi, \mathbb{A}, [u, v])$ where ϕ is a Boolean condition on attributes of S, $\mathbb{A}$ is a subset of the columns in S, and $[u, v]$ is a time interval ($u, v \in \mathcal{T}$, and $u \leq v$).

When a redaction rule R is applied to a t-relation T_S, the intended result is a new t-relation, denoted $R(T_S)$, in which some attribute values have been suppressed. To formalize $R(T_S)$ we use a suppression function supp$(x, \mathbb{A})$ which replaces attributes of $\mathbb{A}$ in the transaction-time tuple x with variables. For example, if $x = (101, Bob, Sales, 10k, 300)$ then supp$(x, \{dept, salary\}) = (101, \text{Bob}, \mathbf{dx}, \mathbf{sx}, 300)$. We assume that suppressions of distinct values always use distinct variable names, and that all instances of a value are replaced by the same variable. The choice to use such variables instead of NULL values sacrifices some privacy because it reveals when two redacted values are identical. We believe this is a worthwhile trade off, and we show in Section V that the use of variables can substantially increase auditing accuracy for some queries. Our results can easily be adapted to a suppression function using NULL values.

Definition 3.4 (Redaction Rule Application): For a client relation S, let T_S be a t-relation over S, and R = Redact$_S(\phi, \mathbb{A}, [s, t])$ be a redaction rule. The application of R to T_S, denoted $R(T_S)$, is a new t-relation with the same schema:

$$R(T_S) = \{\text{supp}(x, \mathbb{A}) \mid x \in T_S, \phi(x), x.t \in [u, v]\} \cup \{x \mid x \in T_S, \neg\phi(x) \vee x.t \notin [u, v]\}$$

We assume for simplicity that $\mathbb{A}$ does not contain the key for table S. If the key for R is sensitive, and subject to retention policies, a surrogate non-sensitive key attribute can be introduced to the schema. This means that even if all attributes of the schema are redacted, the history of changes to a tuple is still preserved.

Having applied a redaction policy, the resulting table $R(T_S)$ is formally an *incomplete* t-relation. It is a representation of a set of possible worlds, each resulting from a different substitution of distinct values for the variables introduced by the suppression of attributes. We define incomplete relations formally in Section IV.

Retention policy composition: Retention rules can be combined to form composite retention policies. A set of redaction rules is combined by hiding any attribute value that satisfies the selection condition and time-period of *any* individual redaction rule. A set of expunction rules is combined by removing all tuples satisfying *any* individual expunction rule. Expunction rules take precedence over redaction rules: a tuple satisfying both an expunction and redaction rule will be removed rather than suppressed.

Example 3.5 In Section I-A, we described informally two retention policies. The redaction rule that *hides Bob's salary between time 0 and 250* is written formally as R = Redact$_S$(name='Bob', `sal`, $[0, 250]$). The expunction rule that *removes the record of all employees in the HR department between time 0 and 300* is written E = Expunge$_S$(dept='HR', $[0, 300]$). Table III is the t-relation that results from applying both E and R to the original table T_S shown in Table II.

TABLE IV
A SANITIZED AUDIT LOG, $P(L_S)$ TRANSFORMED UNDER THE RETENTION POLICIES OF SECTION I-A AND EXAMPLE 3.5.

client	**IP**	**ttime**	**type**	`eid`	`name`	`dept`	`sal`
Jack	1.1.1	0	ins	101	Bob	Sales	**sx**
Jack	2.1.1	100	upd	101	-	-	**sy**
Kate	3.1.1	200	upd	101	-	Mgmt	-
NULL	NULL	250	upd	101	-	-	12
Kate	4.1.1	300	upd	101	-	-	15
NULL	NULL	300	ins	201	Chris	Mgmt	10
Kate	4.1.1	500	del	201	-	-	-

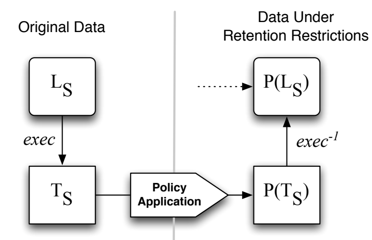


Fig. 1. Illustration of the relationships between original history (L_S and T_S) and the history under retention policy P. $P(T_S)$ is defined directly, while $P(L_S)$ is the sanitized log derived from $P(T_S)$ and including audit fields from L_S.

B. Sanitizing the audit log

Consider a policy P consisting of redaction and expunction rules. According to the definitions above, we apply the policy to T_S, to get the t-relation $P(T_S)$. As we have seen in the examples of Section I-A, the answers to audit queries are not determined completely by the table $P(T_S)$. For one, the audit fields in L_S are not present. We must use L_S in combination with $P(T_S)$ to answer queries that reference the audit fields. In addition, the operations applied to the database need to be inferred from $P(T_S)$ which represents just the history of database states. In order to combine audit field information, and to make explicit the changes to the database that are implied by $P(T_S)$, we compute a *sanitized log* consistent with $P(T_S)$. This new log is denoted $P(L_S)$ and has the property that running it results in $P(T_S)$, that is: *exec*$(P(L_S)) = P(T_S)$. The auditor, and other users, will have access to both $P(T_S)$ and the sanitized audit log. Together we refer to these as the sanitized history. The relationship between the audit log and transaction-time tables in our framework is illustrated in Figure 1.

In computing the sanitized history, we hope to satisfy the following properties.

- A sanitized history is **secret** if it respects the semantics of the policy, hiding tuples and values appropriately. This means it is not possible to infer from the protected history anything that is not present in $P(T_S)$ (the defined meaning).
- A sanitized history is **sound** if it omits information, but does not lead to false answers to audit queries. This property is ensured for all queries if the possible worlds implied by $P(T_S)$ includes the original history. In that case, the true answer to any audit query must be a

possible answer under retention restrictions.

Note that for any redaction rule R and expunction rule E, $R(T_S)$ and $E(T_S)$ are secret by definition. The challenge to secrecy comes from integrating L_S. Also note that expunction policies necessarily violate soundness. Because an expunction policy changes history by removing records, it produces false answers to audit queries.

Definition 3.6 (Sanitized Log): Let P be a retention policy consisting of redaction rules, expunction rules, or both, and let $P(T_S)$ be the (possibly incomplete) t-relation that results from applying P to T_S. The sanitized log under P is denoted $P(L_S)$ and is defined as follows:

1) Treating any variables present in $P(T_S)$ as concrete data values, compute the audit log table $exec^{-1}(P(T_S))$
2) Let $L_S^0 = \Pi_{\langle audit\text{-}fields \rangle, ttime}(L_S)$
3) $P(L_S) = L_S^0 \bowtie\!\!=_{ttime} exec^{-1}(P(T_S))$

This procedure first uses the $exec^{-1}$ to compute an audit log from $P(T_S)$. Then we extract the audit fields and time column from the original audit log. This table, L_S^0, is then joined with $exec^{-1}(P(T_S))$. We use a right outer join to preserve tuples in $exec^{-1}(P(T_S))$ which may not have a match in L_S^0. This occurs when the application of a redaction policy splits the active interval of one or more records. It suggests that an update operation occurred in the history, but the time instant of this update does not match any update in the original audit log.

Example 3.7 Table IV is the sanitized audit log computed according to the above definition, for the policy described in Example 3.5.

Note that Definition 3.6 is not itself an attractive strategy for computing the sanitized log. We describe our implementation of policy application in Section V. In addition, we will see below that policies can be "applied" logically in which case $P(L_S)$ may never be materialized.

C. Retention policy analysis

We can show the following properties of the sanitized log.

Proposition 3.8: Let L_S be an audit log, T_S the t-relation derived from it, and let P be a retention policy consisting of a set of redaction rules $R_1 \ldots R_n$ where each $R_i = \text{Redact}_S(\phi_i, \mathbb{A}_i, [u_i, v_i])$.

- The computation of $P(L_S)$ is *sound.*
- The computation of $P(L_S)$ is *secret* iff $u_i, v_i \in \Pi_{ttime}(L_S)$ for all i.

Proof: (Sketch) Soundness follows from that fact that $P(T_S)$ is sound, and the fact that $P(L_S)$ is consistent with $P(T_S)$, in the sense that $exec(P(L_S)) = P(T_S)$. It follows that the original history is one possible world of $P(L_S)$. If the condition $u_i, v_i \in \Pi_{ttime}(L_S)$ fails, then there are dangling tuples in the join described in Definition 3.6. The absence of audit fields leaks information and violates secrecy. If the condition holds then there are no dangling tuples. Secrecy follows from the fact that $R(L_S)$ is consistent with $R(T_S)$ and uses only the projection, L_S^0, of L_S. ■

The sanitized log from Example 3.7 and Table IV demonstrate the problems that result from arbitrary redaction intervals. These policies split intervals and suggest phantom updates that cannot be convincingly represented in the log. The failure of secrecy appears not to be merely an artifact of the semantics of redaction, but instead a fundamental difficulty in presenting an audit log that is consistent with a redacted data history. It is possible that secrecy could be achieved by introducing additional uncertainty about phantom modifications, but this entails a more powerful model of incompleteness, potentially sacrificing efficiency, and degrading audit query accuracy. Further investigation is a topic of future work.

As a practical matter, to avoid sacrificing secrecy for redaction rules, the desired time interval $[u, v]$ of each redaction rule can be shifted, either forward or backward, to the time of the nearest modification (to any field) in the log.

Policy/Query Independence: It is possible to decide statically, for a given policy and audit query, whether the query answer will be unaffected by the policy. This problem is closely related to the study of view independence of updates [6], [7]. Here the audit query occupies the place of the view. Our retention policies can be considered deletions (in the case of expunction) or updates (in the case of redaction). Known results provide sufficient conditions for determining policy-query independence in our framework.

D. Physical v. Logical Policy Application

The discussion above has suggested the physical application of retention policies to the audit log and derived transaction-time table, in which record removal and attribute suppression are reflected in the storage system. Physical sanitization is appropriate when privacy policies mandate removal of data, data storage is not trusted, and/or the database will be shared with others who are subject to retention restrictions.

An alternative is logical removal, in which the audit log is not physically changed. Instead, a logical view is computed which is consistent with the retention policy. Logical sanitization can support multiple distinct retention policies that can be associated with users or groups of users, in a manner very similar to an access control policy. (Under logical log sanitization, our retention policies can be seen as a combination of fine-grained and view-based access control over a transaction-time database.)

In Section V we implement our policies both physically, using an update program that transforms stored tables, and logically, by rewriting incoming audit queries to return answers in accordance with the stated policy.

IV. Audit Queries under Retention Restrictions

Under a retention policy that includes a redaction rule, audit queries must be evaluated over tables containing variables in place of some concrete values. In this section we use techniques for querying incomplete information [8], [9] to describe precisely the answers to audit queries under retention policies.

A. Incompleteness in relations and t-relations

Both regular relations and transaction-time relations can be incomplete. There are two main features that distinguish an incomplete relation from a concrete relation. The first is the presence of variables in attribute values. The second is a *status* column, included in the schema of every incomplete relation. The status column is **C** when the tuple is *certain* to exist in the relation, and **P** when the tuple may possibly exist.

Under a retention policy P, the inputs to our audit queries are the audit log table $P(L_S)$ and t-relation $P(T_S)$. Both tables may be incomplete, since they may contain variables. In addition, each of their tuples is understood to have a status of *certain*. In general, audit query answers will include both possible and certain tuples.

An incomplete relation represents a set of possible relations. Let R be a relation schema (regular or transaction-time) and let I_R be an incomplete relation over R. Also let $I_R = I_R^p \cup I_R^c$ where I_R^c are the certain tuples and I_R^p are the possible tuples. If V is the set of variables appearing in R, and f is a one-to-one function from the variables V into the domain of R, then a possible world consists of the certain tuples under f, plus any subset of possible tuples under f. Thus, the set of possible worlds represented by I_R, denoted $rep(I_R)$, is defined as:

$$rep(I_R) = \{f(I_R^c) \cup X \mid f \in F, X \subseteq f(I_R^p)\}$$

where F is the set of all one-to-one functions $f : V \rightarrow dom(R)$ and $f(I_R)$ is the relation after replacing variables according to f.

Recall that in our framework, variables only appear in attributes of the client schema – not in time stamps. Extending the definition of t-relation from Section II, an incomplete t-relation over S is a subset of *tuples*$(S) \times \mathcal{T} \times \{\mathbf{P}, \mathbf{C}\}$. A tuple $(p_1, \ldots, p_n, t, u) \in I_S$ represents the fact that tuple $(p_1, \ldots, p_n)$ is certainly active at time instant t (if $u = \mathbf{C}$) or possibly active at time instant t (if $u = \mathbf{P}$). Incomplete t-relations can also be represented as tuples $(p_1, \ldots, p_n, \mathit{from}, \mathit{to}, u)$ which means that $(p_1, \ldots, p_n)$ has status u at each instant t, for $\mathit{from} \leq t \leq \mathit{to}$.

B. Extended Relational Algebra on Incomplete Relations

Next we define the extended relational algebra operators on incomplete relations. The semantics of these operators is similar to the model of relational incompleteness presented by Biskup [10], but includes extensions for transaction-time. Naturally, these operators return incomplete relations, inheriting variables from the input relations and computing the status field appropriately for output tuples. We provide definitions of selection, cross-product, concurrent cross-product, and set difference. Join and concurrent-join are derived from these, and projection, union, and the time-slice operator are defined in a standard way.

Selection: Let I_R be an incomplete relation, and E be a selection condition that is the Boolean combination of comparisons of the form $R.x = c$ (for constant c) or $R.x = R.y$. Comparisons can evaluate to **P**, **C**, or False. If the arguments are two different constants, or two different variables, the comparison evaluates to False. The comparison of a variable with a constant evaluates to **P**. If the arguments are identical variables, or identical constants, the comparison evaluates to the *status* value for the tuple. The Boolean combination of terms is evaluated using the rules of three-valued logic where **P** is interpreted as *Unknown*, and **C** is interpreted as True.

Tuples are included in the output of the selection operator if their status evaluates to *either* **P** or **C**. When the condition E has evaluated to **P** under the comparison of a variable with a constant, this variable binding needs to be applied to the output tuple. Formally we have:

$$\sigma_E(I_R) = \{\langle f(r.*), E(r)\rangle \mid r \in R, E(r) = P \vee E(r) = C\}$$

The tuples returned have all non-status attributes (denoted $r.*$) with variables replaced under mapping f, and a new status field $E(r)$.

Example 4.1 Consider the selection condition $R.a = 100 \wedge R.b = R.c$. On the input relation $\{\langle dx, dy, 9, C\rangle\}$, the selection operation will return $\{\langle 100, 9, 9, P\rangle\}$.

Cartesian product: If I_R and I_S are two incomplete relations over schema R and S, the cartesian product $I_R \times I_S$ is defined as:

$$I_R \times I_S = \{\langle r.*, s.*, status\rangle \mid r \in I_R, s \in I_S\}$$

where *status* is set to $r.status \wedge s.status$.

Concurrent cartesian product: If I_R and I_S are two incomplete t-relations over schema R and S, the concurrent cartesian product $I_R \times I_S$ is defined as:

$$\begin{aligned} I_R \times^{\diamond} I_S \quad = \quad & \{\langle r.*, s.*, from, to, status\rangle \mid r \in I_R, s \in I_S, \\ & [r.from, r.to] \cap [s.from, s.to] \neq \emptyset\} \end{aligned}$$

where *status* is set to $r.status \wedge s.status$, $from = max(r.from, s.from)$, $to = min(r.to, s.to)$.

Duplicate Elimination: Duplicates (on the non-status columns of a table) can arise as a result of projection or union, as well as selection and join (because of the substitution for variables). If a tuple is both possible and certain, it is only necessary to preserve the certain version of the tuple. In general, duplicates on the non-status columns are eliminated by preserving a single tuple with a status value equal to the disjunction of all duplicates' status values. That is, it will be **C** if at least one duplicate had status **C**.

Set Difference: If I_R and I_S are two incomplete relations, then in computing $I_R - I_S$, the tuple $\langle r.*, status\rangle$ will be removed from I_R only when there exists a tuple $\langle s.*, C\rangle \in I_S$ where $r.*$ and $s.*$ shares the same value or variables on each attribute. Otherwise, write $\langle r.*, P\rangle$ into result when there exists a tuple $\langle s.*, status\rangle \in I_S$ where evaluation of $r.A = s.A$ (described in operator Selection section) is P or C for all attributes A in the client schema. When I_R and I_S are t-relations, we must expand the temporal intervals into instants (according to our definition of t-relation), execute the set difference, and finally coalesce them back into intervals.

Example 4.2 Recall from Section I-A that audit query A1 returns *all employees who earned a salary of 10k at some point in time*, and can be written $\Pi_{name}(\sigma_{sal=10k}(T_S))$. On the incomplete t-relation shown in Table III (for which the omitted status column is uniformly **C**) we have the intermediate result of $\sigma_{sal=10k}(T_S)$:

eid	name	dept	sal	from	to	status
101	Bob	Sales	10	0	100	P
101	Bob	Sales	10	100	200	P
101	Bob	Mgmt	10	200	250	P
201	Chris	Mgmt	10	300	500	C

and the final result of $\Pi_{name}(\sigma_{sal=10k}(T_S))$:

name	status
Bob	P
Chris	C

C. Discussion

Our representation system for incomplete relations cannot describe constraints or correlations between the possible tuples, and is therefore *incomplete* [11]. For example, we cannot represent a set of possible worlds in which tuple t_1 or tuple t_2 is present, but not both. Although the base relations in our formalism never need to describe such sets of possible worlds, the relational operators can result in correlated tuples and these correlations will be lost by our representation system. The loss of these correlations means less accurate query answers, but allows for more efficient query processing and more intuitive query answers. Through experiments, we show in Section VI that our data model allows for useful audit query answers over incomplete relations. We leave as future work the investigation of a complete representation system for transaction time relations under retention policies [12].

V. Implementation

The implementation of our framework, which is also described briefly in [13], translates our historical data model into standard relations in Postgres. Our goal is to show the practical feasibility of our framework. We optimize our implementation using commonly-available indexing strategies and query rewriting techniques. A fully optimized implementation might make use of techniques specifically designed for transaction-time data, but these are beyond the scope of our prototype.

In our implementation, the time stamp fields `from` and `to` are combined into one attribute named `trange`, which is stored as an interval type (actually a one-dimensional cube data type in Postgres). Utilizing the cube data type simplifies the expression of the concurrent join, and we also use an available R-tree implementation. In each t-relation, `status` is represented as a Boolean value.

In the remainder of the section we discuss the physical application of retention policies followed by query evaluation on physically sanitized datasets. Lastly we describe logical application of policies.

A. Physical Application of Retention Policy

Since the policies are specified over t-relations, a policy P with an arbitrary time condition $[u, v]$ may require a split of update intervals causing phantom updates in sanitized log (as demonstrated from Example 3.7 and Table IV). To avoid this, we adjust the redaction period to the nearest modification period of any field.

Application of retention policies is implemented by transforming the input rules into a set of update operations on original t-relation and possibly audit log. We assume we have all policy rules at the time of policy application. Inconsistencies may arise if in the subsequent application of new policy rules, which has been addressed in [14], [15]. Policy application for all rules is accomplished in one pass, guaranteeing that all conditions in the rules are fully evaluated on the current tuple before removing any values from that tuple.

Redaction is implemented by replacing values with variables. As described previously, variables here preserve equality even after redaction. That is to say, the relationship between value and variable is a strict one-to-one mapping. In our current implementation, we use a cryptographic hash function.

B. Audit Query Evaluation

In the following we implement in SQL the semantics of extended relational operators over incomplete relations. We describe the rewriting of SELECT-FROM-WHERE blocks to accommodate incompleteness. First, we write a WHERE clause that will select any tuple evaluating to either **P** or **C**, eliminating all others. Second, we formulate a SELECT clause that is used to compute the correct `trange` (if necessary), the status column, and return appropriate values of variable bindings. To return the correct variable bindings for selection (as described in Section IV), we must rewrite those attributes when they appear in both the SELECT list and some equality expression in the WHERE clause. If an attribute appears in two equality expressions in an OR operation, we may need to break the query into parts and union their results.

In the following description, the function `isvari(x)` tests if x is a variable. `onevari(x,y)` returns true only when one of x and y is a variable. `binds(x,y)` returns x if x is a constant, otherwise returns y. The general algorithm is as follows:

1) Reorganize the WHERE clause as a set of disjunctive conditions $\mathcal{D} = \{D_1, \ldots, D_n\}$, where each $D_i = \{c_i\}$ and c_i is a conjunction. Unite D_i and D_j: $D_i = D_i \cup D_j$ and $\mathcal{D} = \mathcal{D} - \{D_j\}$ only if for any $c_i \in D_i$ and $c_j \in D_j$, there is no equality expression related to same attribute. Finally we get $\mathcal{D}' = \{D_1, \ldots, D_m\}$ and each $D_i \in \mathcal{D}'$ will be executed in a separate query.
2) For each $D_i \in \mathcal{D}'$, create a new query, defining the query as follows:
   WHERE clause: for each conjunct $c \in D_i$, c consists of a set of conjunctive atomic expressions exp. If exp is *t.a op CON*, rewrite it as *t.a op CON or isvari(t.a)*. If exp is $t_1.a$ *op* $t_2.a$, rewrite it as $t_1.a$ *op* $t_2.a$

or onevari(t_1.a,t_2.a). If there are two expressions like t_1.*a=CON1* and t_2.*a=CON2*, add a new expression in this conjunctive term $t_1.a$ *!=* $t_2.a$. Finally add condition on trange when necessary.
FROM clause: only tables involved in D_i.
SELECT clause: Put D_i into SELECT clause, and for each $c \in D_i$, add a conjunction of related status attributes to the term computing the final status. If attribute a of select list also appears in exp like $t.a = CON$, replace a with CON as a return value. If a appears in $t_1.a = t_2.a$, use a special function $binds(t_1.a, t_2.a)$ as the result. Finally, compute the correct trange value if necessary (i.e., concurrent join).
3) Union each query generated on D_i.

Example 5.1 The following is an example query on complete table `emp`:

```
SELECT name, t1.dept, t2.sal
FROM emp AS t1, emp AS t2
WHERE t1.dept=t2.dept AND
      t1.sal=100 AND t2.sal=200
```

The algorithm above will produce the following rewritten query if `emp` is incomplete:

```
SELECT name, binds(t1.dept,t2.dept) AS dept,
       200 AS sal, (t1.dept=t2.dept AND
       t1.sal=100 AND t2.sal=200 AND
       t1.status AND t2.status) AS status
FORM emp t1, emp t2
WHERE (t1.dept=t2.dept
      OR onevari(t1.dept, t2.dept))
      AND (t1.sal=100 OR isvari(t1.sal))
      AND (t2.sal=200 OR isvari(t2.sal))
      AND t1.sal!=t2.sal
```

As discussed in section IV, duplicates may arise in the result of operations such as union, projection and join. The duplicate elimination process can be achieved by grouping on all non-status columns and then aggregating the (boolean) status column using bitwise OR.

C. Logical Solution

Our implementation above is based on the physical removal of expired information. To implement policies logically, we construct a query Q_P whose answer on T_S and L_S is equivalent to the answer of Q on $P(T_S)$ and $P(L_S)$.

For simplicity, we assume that the redaction policies satisfy the condition in Prop. 3.8. Generally the composition will begin by adopting the rewriting algorithm in the previous subsection, which results in a set of sub-queries $Q = \{Q_1, \ldots, Q_n\}$ connected by union operator. Attributes appearing in either the SELECT or WHERE clause are called critical attributes. For each sub-query Q_i, a redaction rule is *relevant* to Q_i when its redaction attribute list shares some attribute with Q_i's critical attributes. Besides the rewriting process in the previous subsection, we also should:

1) FROM clause: for each table, add a case statement modification based on its relevant redaction rules.
2) WHERE clause: for any expunction rules $(\phi, [u, v])$, add conjunction of not $(\phi \wedge$ *trange overlap* $[u, v])$.

Note that the case statement modification is inspired by similar work in [16], but we change the semantics from NULLS to variables.

VI. Evaluation

In this section we study the performance of query processing in our framework and evaluate the impact of retention policies on the accuracy of query results. Our experiments address the following key questions:

- *Overhead and Scalability*. We assess the performance overhead of evaluating audit queries using both physical and logical policy application. We test the scalability of our framework in terms of database size (the average snapshot size) and history length (the average number of versions of each tuple).
- *Accuracy of uncertain answers*. We study the impact of retention policies on the accuracy of query results. Over sample data, we measure the precision and recall of query answers as a function of the selectivity of redaction policies.
- *Suppression using variables v. NULLs*. Using NULLs is a common solution in relational database research such as fine-grained access control[16]. However, variables can hide values while preserving more information about changes. We show that the extra information kept by variables significantly increases the accuracy of audit query answers.

A. Experimental Setup

In all our experiments we use Postgres 8.3 running on an Intel Core2 workstation with 2.40GHz CPU and 2Gb memory. Our datasets are synthetically-generated histories based on our example client schema S(eid,name,dept,sal).

We generated our history with an initial set of employees that grows slowly over time through periodic insertions. We apply a random sequence of independent updates to attributes throughout the lifetime of individuals. Thus the total tuples in the t-relation and log is closely approximated by the product of two parameters: the initial number of employees (the *snapshot size*) and the average number of versions of each employee tuple (the *history length*).

We use two redaction policies[3] and three queries in our experiments. They are:

R1: Redact all department values before a specified time.
R2: Redact salary values for the Mgmt and HR department in a specified time period.
Q1: Return employees whose dept is Mgmt and whose salary is 10k.
Q2: Return all the clients who changed the salary of employees in the dept Mgmt.
Q3: Return all employees who worked in the same department as a specific employee at the same time.

[3]We do not consider expunction rules since they will simply remove tuples and reduce the size of the history.

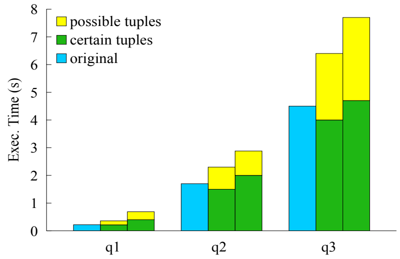


Fig. 2. Performance on three queries, for each query the bars from left to right are the original, physical and logical solution respectively.

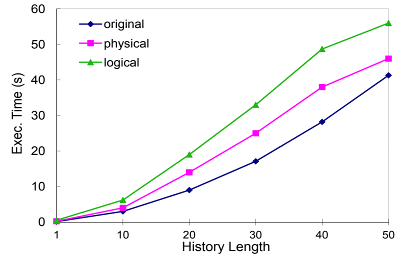


Fig. 3. Performance of Q3 on tables with variable history length and fixed snapshot size of 100,000.

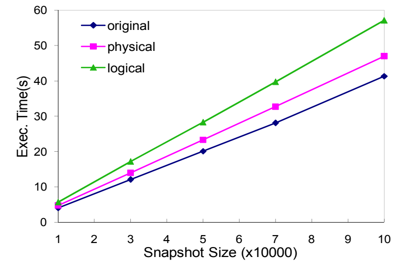


Fig. 4. Performance on Q3 of tables with variable snapshot size and fixed history length 50.

The three queries include one table scan (Q1), a traditional join of the audit log and t-relation (Q2), and a concurrent self join on the t-relation (Q3).

We measure the query execution time by reporting the average of 10 runs with the largest and smallest runs omitted.

B. Overhead and Scalability

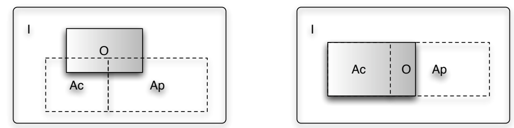
In our first experiment (shown in Figure 2) we compare the execution time of each of the three queries for both *physical* and *logical* policy application. The baseline (*original*) is the time to compute the audit query without the retention policy, that is, on the original tables. For the logical and physical techniques, we also distinguish between the time for computing certain tuples and possible tuples. The total number of tuples is one million.

We find that evaluating queries under retention restrictions has a modest overhead, to be expected from the added clauses in the queries and the fact that result sizes are increased because of uncertain tuples. In addition, the logical solution is uniformly slower that the physical because of the more complex queries required when policies are composed with queries. It is worth noting that the certain tuples alone can be computed more quickly that the original result. This is because the rewritten query computing certain tuples can ignore variables and the certain tuple set returned tends to be smaller than the true result.

These relationships hold when we scale up the size of the historical data set. Figure 3 shows the execution time of Q3 on the history with fixed snapshot size of 100k when scaling on history lengths from 1 to 50. Similarly, Figure 4 uses a fixed history length of 50, and varies the snapshot size from 10k to 100k. That is to say, the total number of tuples in the data history is up to five million in both cases. Both physical and logical execution times increase nearly linearly as the total number of tuples increases in the two graphs. Both techniques scale at close to the rate of the query on the original data, with the physical case outperforming the logical.

C. Accuracy of Uncertain Answers

Next we evaluate experimentally the accuracy of audit query answers under sample retention policies. Over the original data, an audit query can be considered to partition the set of all feasible query answers (determined by the active domain) into answer tuples and disqualified tuples. Under retention



(a) General case (b) Guarantee of precision = 1

Fig. 5. Result relationship in Venn Diagram: The answer space is I (the Largest box) and the original answer are O (shaded box), the certain tuples in our model are A_c, the possible tuples is A_p (both are boxes with doted-line).

restrictions, audit queries partition the set of feasible answers into certain answers, possible answers, and disqualified tuples. Our first measure of accuracy considers this distribution of answers as a function of the selectivity of the redaction policies. The second measurement is the *precision* and *recall* of our answers to original ones. Assume the answer space is I and the true original answer is O, the certain tuples in our model are A_c, the possible tuples is A_p. Intuitively, we want to know how large is $O \cap A_c$ (fig5(a)) in proportion to O and A_c. Formally, the precision of certain tuples is defined by $\frac{O \cap A_c}{A_c}$ and the recall of certain tuples is defined $\frac{O \cap A_c}{O}$.

We can also define precision and recall of the impossible tuples, which may be relevant to auditors since disqualifying answers has value in an investigation. The precision of impossible tuples is defined $\frac{(I-A_c-A_p)\cap(I-O)}{I-O}$ and recall of impossible tuples is defined $\frac{(I-A_c-A_p)\cap(I-O)}{I-A_c-A_p}$. Note that if we consider sound and secret retention policies, as described in Section III, then the precision of certain and impossible tuples is always equal to 1, showed in fig5(b), because the soundness (Prop. 3.8) guarantees $A_c \subseteq O$ and $O \subseteq A_c \cup A_p$.

The first experiment is performed on Q2, which is a standard join between the t-relation and the audit log. Since policy R2 is irrelevant to this query, the selectivity is measured by R1. We vary the time condition in R1 to increase its selectivity, e.g. 50% indicates that the time condition is half of the history time. Figure 6 shows the answer distribution. The first bar is the result without the policy. The true answer, which for Q2 is a set of client names, happens to return 20% of all the clients in the database. The other 80% are impossible. Under retention policies we can see the region of possible tuples grows with the selectivity of the policy. Yet, the certain tuple set remains close to 20% for reasonable selectivities of up to 10% − 25%. Figure 7 measures the recall of the certain and impossible tuples directly for the same query and policy. The recall for certain tuples decreases rapidly when selectivity

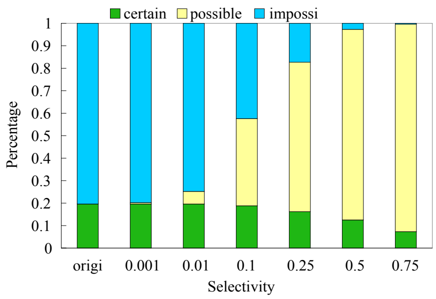


Fig. 6. Answer Distribution (Q2)

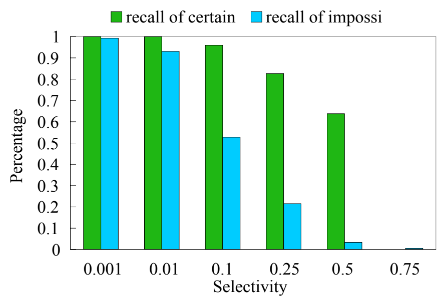


Fig. 7. Recall of Answers (Q2)

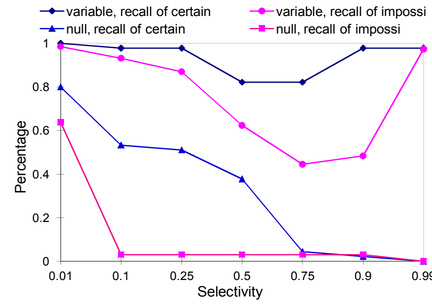


Fig. 8. Variables v.s. NULLs

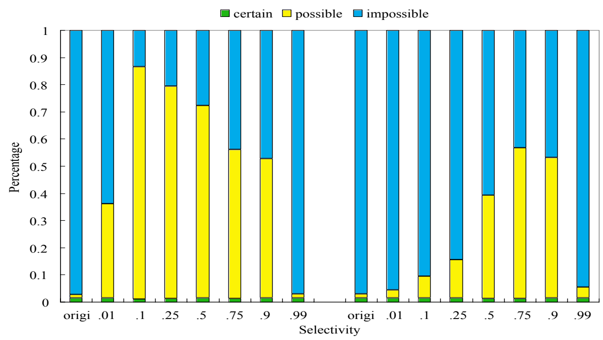


Fig. 9. Answer Distribution (Q3)

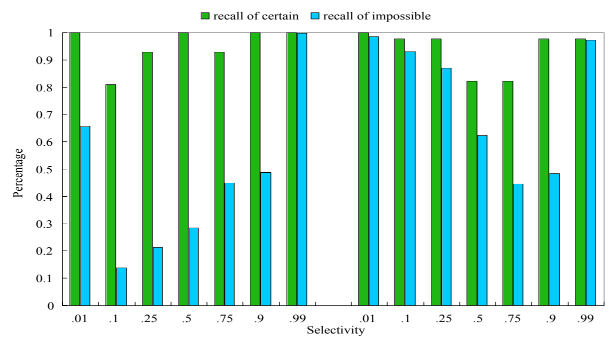


Fig. 10. Recall of Answers (Q3)

is larger than 50%. When selectivity is larger than 10%, we miss a lot of impossible answers but recall of certain answers decreases much more slowly.

The second experiment is on Q3 (the concurrent self-join). We vary the target employee E in the select condition, choosing a person who joined the company at an earlier or later time. The results are shown in Figure 9 and 10 (left side for earlier employee and right side for new employee). The trend of the answer distribution and recall is quite different from the last experiment. The percentage of possible tuples, recall of certain and impossible tuples all have an inflection point as the selectivity goes up. This is because when the selectivity is small, fewer variables are introduced to the t-relation so we can retain a high recall. When the selectivity increases, the number of variables increases and recall decreases. On the other end, when selectivity is extremely high, the t-relation is mostly variables on key attributes. We can still get high recall since the equivalence among variables can be inferred accurately. We can get very high accuracy at selectivity 100%. The difference inside the two settings of this experiment are inflection point at different selectivity level, which is decided by attribute dependency of our data.

D. Suppression using variables v. NULLs

In our final experiments we apply redaction policies using a suppression function that uses NULL values instead of variables. Figure 8 shows the recall of certain and impossible tuples on Q3 (with condition on an early employee) compared with the variable solution. Variables significantly outperform NULLs. For example, with a selectivity of 25% the recall of certain tuples is 97% using variables, but just 56% using NULLs. This is because any two tuples with NULL on the join column will produce a possible output tuple. With distinct variable assignments, only identical variables will result in an output tuple.

VII. Related Work

Retention policies and problems of expiring historical data have been studied in a variety of contexts. Garcia-Molina et al. considered expiring tuples from materialized views in a data warehouse [17]. 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 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 [18]. Skyt et al. consider vacuuming a temporal database [19]. 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. [15] considered retention restrictions on complex business records, which they describe by logical views over relations. They define protective and destructive policies, and reduce a number of retention problems to well-studied relational view problems.

Our redaction policies (especially when implemented logically) are related to fine-grained access control rules. Wang et al. [20] recently studied the correctness of query answers under cell-level access control policies, and made an important connection between that problem and models of incomplete information. To our knowledge there is little work on access control over time-varying data. Research into temporal access control models [21] refers to *access rights* that change over time, not the problem of negotiating access to data with a time dimension.

Transaction-time databases have been studied extensively by the research community including work on query languages and logical foundations [5], [4], [22], implementation techniques [23], [24], [2], techniques for accommodating time in standard databases [25], [26], as well as implemented extensions to existing systems [27]. Jensen studied querying backlog relations to monitor changes to a database [3]. Incomplete information also has a long history in databases [8], [28], [10], including in temporal databases. The model of temporal incompleteness presented by Gadia et al. [9] is more expressive than ours. It allows for uncertainty about values, as we do, but also represents certain values whose active period is uncertain. Despite work on data models and query languages to support temporal incompleteness, we are not aware of any implementations of the techniques.

Encrypting audit logs have been widely studied in the literature [29], [30], [31] with the goal of maintaining the confidentiality and integrity of log records. The problem in this paper is to allow auditing under *legal* deletion of data and logs.

VIII. Conclusions & Future Work

We have presented a framework for limiting access to historical data, while still permitting auditing. Our redaction rules hide values but preserve information about the lifetime of tuples in a database, allowing an auditor to get accurate answers from the historical record despite the information removed by retention restrictions. We demonstrated that our techniques have a modest performance overhead, even when implemented on a standard relational system, and that the uncertainty introduced by sample retention policies is acceptable.

Our approach to obscuring values with variables was shown to substantially improve answer accuracy (as compared with NULLs). However this scheme can be vulnerable to an insider attack. Suppose Bob's salary was 10k at time x but is later redacted. If Bob has the right to access both his and other employees' information, he may find Jack's salary at time y is equal to his redacted salary at time x, allowing him to infer that Jack has salary 10k at time y, in violation of the redaction policy. Solving this problem requires trading secrecy for auditing accuracy. In addition, an extension to our model could generalize or summarize values instead of redacting them. At a small cost to confidentiality, this could substantially improve auditing capabilities. Finally, a more powerful model of incompleteness might offer improved soundness and secrecy properties for sanitized histories, at the expense of increased query processing complexity. We believe each of these are promising directions for future work.

Acknowledgments: This material is based upon work supported by NSF CAREER Grant No. 0643681.

References

[1] B. Wrozek, "Electronic Data Retention Policy," 2001. [Online]. Available: http://www.sans.org/reading_room/whitepapers/backup/514.php

[2] C. S. Jensen, L. Mark, and N. Roussopoulos, "Incremental implementation model for relational databases with transaction time," *IEEE Trans. Knowl. Data Eng.*, vol. 3, no. 4, pp. 461–473, 1991.

[3] C. S. Jensen and L. Mark, "Queries on change in an extended relational model," *IEEE TKDE*, vol. 4, no. 2, pp. 192–200, 1992.

[4] S. K. Gadia, "A homogeneous relational model and query languages for temporal databases," *ACM Trans. Database Syst.*, vol. 13, no. 4, pp. 418–448, 1988.

[5] J. Chomicki, "Temporal query languages: a survey," in *Temporal Logic: ICTL'94*, vol. 827, 1994, pp. 506–534.

[6] J. A. Blakeley, N. Coburn, and P.-A. Larson, "Updating derived relations: detecting irrelevant and autonomously computable updates," *TODS*, vol. 14, no. 3, pp. 369–4000, 1989.

[7] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently updating materialized views," *SIGMOD Rec.*, vol. 15, no. 2, pp. 61–71, 1986.

[8] T. Imielinski and W. Lipski, "Incomplete information in relational databases," *J. ACM*, vol. 31, no. 4, pp. 761–791, 1984.

[9] S. K. Gadia, S. S. Nair, and Y.-C. Poon, "Incomplete information in relational temporal databases," in *18th VLDB Conference*, 1992.

[10] J. Biskup, "A foundation of codd's relational maybe-operations," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 608–636, 1983.

[11] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[12] T. Imieliński and J. W. Lipski, "Incomplete information in relational databases," *J. ACM*, vol. 31, pp. 761–791, 1984.

[13] W. Lu and G. Miklau, "Auditguard: A system for database auditing under retention restrictions," in *VLDB Demo Program*, 2008.

[14] J. Skyt, C. Jensen, and L. Mark, "A foundation for vacuuming temporal databases," *Data & Knowl. Eng.*, vol. 44, no. 1, pp. 1–29, 2003.

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

[16] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, "Limiting disclosure in hippocratic databases," *Proc. of 13th VLDB-Volume 30*, pp. 108–119, 2004.

[17] H. Garcia-Molina, W. Labio, and J. Yang, "Expiring data in a warehouse," in *Conference on VLDB*, 1998, pp. 500–511.

[18] D. Toman, "Expiration of historical databases," in *Symposium on Temporal Representation and Reasoning (TIME)*, 2001, pp. 128–135.

[19] J. Skyt, C. S. Jensen, and L. Mark, "A foundation for vacuuming temporal databases," *Data Knowl. Eng.*, vol. 44, no. 1, pp. 1–29, 2003.

[20] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun, "On the correctness criteria of fine-grained access control in relational databases," in *VLDB*, 2007, pp. 555–566.

[21] E. Bertino, C. Bettini, and P. Samarati, "A temporal authorization model," in *ACM CCS*, 1994, pp. 126–135.

[22] R. T. Snodgrass, *The TSQL2 Temporal Query Language*. Norwell, MA, USA: Kluwer Academic Publishers, 1995.

[23] R. Shaull, L. Shrira, and H. Xu, "Skippy: a new snapshot indexing method for time travel in the storage manager," in *ACM SIGMOD Conference*, 2008, pp. 637–648.

[24] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu, "Transaction time support inside a database engine," in *ICDE*, 2006.

[25] R. T. Snodgrass, *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., 2000.

[26] N. L. Sarda, "Extensions to sql for historical databases," *IEEE Trans. Knowl. Data Eng.*, vol. 2, no. 2, pp. 220–230, 1990.

[27] R. T. Snodgrass and C. S. Collberg, "The τ-MySQL transaction time support," Available at `www.cs.arizona.edu/tau/tmysql`.

[28] G. Grahne, *The Problem of Incomplete Information in Relational Databases*, ser. LNCS. Springer, 1991, vol. 554.

[29] B. Schneier and J. Kelsey, "Secure Audit Logs to Support Computer Forensics," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 2, pp. 159–176, 1999.

[30] R. Snodgrass, S. Yao, and C. Collberg, "Tamper detection in audit logs," in *13th VLDB-Volume 30*. VLDB Endowment, 2004, pp. 504–515.

[31] B. Waters, D. Balfanz, G. Durfee, and D. Smetters, "Building an encrypted and searchable audit log," in *NDSS*, vol. 6, 2004.