The VLDB Journal
DOI 10.1007/s00778-012-0282-x

REGULAR PAPER

Auditing a database under retention policies

Wentian Lu · Gerome Miklau · Neil Immerman

Received: 7 May 2011 / Revised: 13 May 2012 / Accepted: 22 May 2012

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 an historical record of the past that can also pose a serious threat to privacy. Policies that 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 an historical data model that supports flexible audit queries, along with a language for retention policies that can hide individual attribute values or remove entire tuples from the history. Under retention policies, the audit history is partially incomplete. Thus, audit queries on the protected history can include imprecise results. We propose two different models (a tuple-independent model and a tuple-correlated model) for formalizing the meaning of audit queries. We implement policy application and query answering efficiently in a standard relational system and characterize the cases where accurate auditing can be achieved under retention restrictions.

Keywords Privacy · Auditing · Retention policy

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 an 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 policies limiting retention, 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. Failure to dispose of the expired data can result in serious consequences and is often viewed as an institutional risk [50]. At the same time, other forces may require the preservation of records, for example, when ongoing litigation makes removal of data unlawful. Institutions are increasingly recognizing that a consistently enforced retention policy reduces their legal risk by ensuring that electronic data are handled properly [26].

Limited retention conflicts with the goals of accurate auditing, analysis, and prediction based on past history. This conflict is evident in the guidelines for record keeping published by a records management trade group [2], which includes principles of data availability and data retention along with data disposal. Data owners thus have to

The authors gratefully acknowledge the comments of the VLDBJ editors and the anonymous reviewers. Authors Lu and Miklau were supported by NSF CAREER Grant No. 0643681.

Electronic supplementary material The online version of this article (doi:10.1007/s00778-012-0282-x) contains supplementary material, which is available to authorized users.

W. Lu (✉) · G. Miklau · N. Immerman
Department of Computer Science, University of Massachusetts,
140 Governors Drive, Amherst, MA 01003, USA
e-mail: wen@cs.umass.edu

G. Miklau
e-mail: miklau@cs.umass.edu

N. Immerman
e-mail: immerman@cs.umass.edu

Published online: 06 July 2012

carefully balance the need for accurate auditing with the privacy goals of retention policies. An emerging industry has begun to address the needs of these institutions, building systems that offer varying combinations of records and document management, archiving, eDiscovery, retention, and compliance services [9,15,31,33,35,51,52]. Unfortunately, existing mechanisms for auditing and managing historical records have few capabilities for managing the balance between these 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 an 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.

1.1 Applications

The tension between audit analyses and retention restrictions is present in a broad range of industries where sensitive records are managed, including financial services, healthcare, insurance, technology, education, telecommunications, and others. For example, financial legislation mandates limited retention periods for personal credit reports, including special treatment of negative credit events that are purged from records separately from other events. Search engines are not governed by legislation in the United States, but many elect to sanitize their search logs after 9 months. Logs are generally not disposed of completely, but certain fields are removed to reduce identifying information and to resist subpoenas and court orders.

Healthcare databases store sensitive information about patients, physicians, test results, diagnoses, billing details, and hospital procedures. State and federal laws specify record retention time frames that may depend on whether a patient is enrolled in medicare or medicaid, whether the patient is a minor, whether the medical procedure involves immunization, or on the statute of limitations for medical malpractice claims. At the same time, after mandated retention periods have passed, physicians may have discretion about how or when to dispose of records, or whether to partially sanitize records to remove personally identifiable data or sensitive diagnoses, while still permitting historical analysis. For example, we will define the operation of redaction on fields in a record. This could be applied to the diagnosis field of medical records while still permitting an analysis of a physician's consistency of diagnosis based on test results.

As another example, the office of information technology in a university is responsible for maintaining and monitoring network services for faculty, students, and staff. Network logs may contain information about machines, network connections, web browsing history, search engine requests, and/or file transfers, where users are identified by IP address or login name. Internal auditing may include analyzing logs for evidence of security vulnerabilities with the university. Researchers within the university may wish to perform traffic analysis on network logs. Lastly, external authorities such as the RIAA may request log data pertaining to specified users or specified content. In this setting, retention restrictions arise from privacy protections of individuals using the network. Some logs that are retained for network security purposes may be subject to removal of identifiers or sensitive content like web addresses. In addition, information technology staff reportedly prefer the timely removal of some network logs so that they do not have to bear the cost of inquiries by external authorities.

Next we provide an overview of the motivation and contributions of this work through the following detailed example over a simple employment database. The schema and queries serve as a running example in later sections of the paper.

1.2 Example scenario

We begin with a database consisting of 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 1 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 2 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

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

Table 1 The audit log L_S 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 2 The transaction-time table T_S describing the data history of the client table. It is derived from the audit log in Table 1

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

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 10 at some point in time.*
A2. *Return the clients who updated Bob's salary and the time of update.*
A3. *Return the clients who updated any employee's dept and the time of update.*
A4. *Return the time periods when Bob earns a salary of 10.*

Some audit queries are conventional queries over a transaction-time data model (such as A1, A4). Others ask specifically about changes and reference the special audit fields contained in the audit log (such as A2, A3).

The *compliance officer* is a trusted entity, 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. The compliance officer is also responsible for enforcing preservation rules, which reflect requirements to keep certain data items in the database. Notably, these 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.

Table 3 The transaction-time table, transformed under the following retention policies: $\text{Redact}_S(name = Bob, \{salary\}, [0, 250])$ and $\text{Expunge}_S(dept = HR, [0, 300])$. 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

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*. We believe these rules are sufficiently expressive for practical applications, allowing users to selectively choose related data items, which could be tuples, selected tuples, or individual attribute values [30]. We also support a basic rule for **preservation**, which takes priority over the removal rules above, ensuring that specified records are not altered or removed.

Applying a set of retention rules transforms the stored history of the database.[2] Table 3 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*). Instead of suppression with NULLs, we use variables to support more accurate auditing by retaining more information, as different values are suppressed to different variables. Also note that there is an extra row in Table 3 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 12. 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 pur-

[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. Section 8 provides further detail.

poses, the expunged row is included in Table 3, but displayed with a gray background.

A main goal of this paper is to 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 10 at some point in time.*
This query is a straightforward selection on the transaction-time table. On the original data in Table 2, the answer to this query is {Bob, Chris}. On Table 3, 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 3 shows that Bob's salary definitely changed at time 100 (from *sx* to *sy*) and at time 300 (from 12 to 15). In addition, it *may* have changed at time 250 (from *sy* to 12), 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 12 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 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 3, 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 audit queries that, while possibly 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.

The answer to an audit query under retention rules usually consists of two parts: certain tuples and possible tuples. Such results are *uncertain* answers, because they are computed on a history with incompleteness introduced by applying retention rules. On the contrary, querying the original history without retention rules returns a *real* answer. Intuitively, each uncertain answer represents a set of real answers, each of which is returned by the query over some original history that is consistent with the transformed history under the retention policies. In the case of A1, the uncertain result could represent two real answers. One is {Chris} when A1 is executed over the history where neither sx nor sy is 10, and the other is {Chris,Bob} when either sx or sy is 10. Similarly, if the uncertain result of A1 contains two possible tuples, say, Bob and Ann, we have four real answers represented, {Chris}, {Chris, Bob},{Chris, Ann}, and {Chris, Bob, Ann}. Here, we simply assume the existence of each possible tuple in a real answer is *independent* of others, and thus, we have four different ways of choosing two possible tuples. In this paper, we propose the **Tuple-Independent model (TI)** for answering audit queries under retention policies and define the semantics of uncertain answers returned by TI under this independence assumption. Later we will show, due to this assumption and the extended relational algebra, that there is no extra cost to decide certain and possible tuples of the query results since they are efficiently computed during query evaluation. Thus, TI guarantees efficiency, and this fact serves as a major advantage of the TI model, along with it simplicity. However, the independence assumption is not always correct, and thus, the information delivered by the uncertain answer is not precise, as demonstrated by the following query A4. To solve this problem, we introduce a more sophisticated model, the **Tuple-Correlated model (TC)**, which does not rely on an independence assumption and gives precise interpretations of uncertain answers.

A4. *Return the time periods when Bob earns a salary of 10.*

The answer to this query on the original data is {(0,100)}, which can easily be computed from the original audit log L_S. In the transformed history in Table 3, our result is {(0,100),(100,200),(200,250)}, and all are possible answers due to the unknown value of two variables sx and sy. When using TI, we assume the three periods are independent of each other, and therefore, we could interpret them as eight different real answers. However, a closer look will tell us such an assumption is invalid. (100,200) and (200,250) are correlated because they are bound to the same variable sy. (0,100) is also not independent of either (100,200) or (200,250) because sx and sy are correlated: they are not equal (recall that they represent distinct values). In fact, this uncertain result only represents three different real answers. If sx equals 10, then (0,100) is the output; if sy equals 10, then (100,200) and (200,250) is the output; finally if neither sx nor sy is 10, then the output is empty. For a more accurate representation, we use the TC model that can maintain the correlations among three time periods. Instead of indicating "certain" or "possible" directly for each tuple, TC records extra information in the form of conditions associated with each tuple. The example above may be represented as:

$$\begin{aligned}(0, 100) &: sx = 10,\\ (100, 200) &: sy = 10,\\ (200, 250) &: sy = 10\end{aligned}$$

Because all three equations are satisfiable, we have three possible tuples. Tuples (100,200) and (200,250) occur together, or not at all, depending on the assignment to sy.

Query A4 demonstrates that there are cases where the independence assumption fails, and thus, the TI model is incapable of representing the result accurately. Our TC model abandons the independence assumption and is able to provide accurate answers by recording equalities and inequalities of variables. We use this extra information to decide certain and possible tuples. From the auditors' perspective, the ability to calculate the correlation is important and delivers more valuable information. For instance, given a possible suspect Alice, if the auditor has some external information in Alice's favor, the TC model can help to answer questions like "Who remains a suspect if I assume Alice is not a suspect?" Our system (using either the TI or TC models) returns uncertain answers that reflect the unavoidable imprecision of carrying out an audit task in the presence of a partially removed history. In the absence of our techniques, a conventional system would be unlikely to produce valid query answers at all.

We use the term *expressiveness* to measure the ability to precisely represent the set of correct answers. TC is strictly more expressive than TI because it can interpret the uncertain answer of A4, but TI cannot. After analyzing their expressiveness and investigating other alternatives later in this paper, we conclude that the combination of TI and TC meets the needs of our application. We will see that the cost of TC's expressiveness is the decreased efficiency of deciding which tuples are certain and which are possible.

In summary, the main contributions of this paper are as follows:

- We propose declarative rules for expressing retention restrictions over an historical data model (Sect. 4).
- We define the tuple-independent model (TI) for answering audit queries in the presence of retention restrictions, and we analyze the impact of retention policies on the accuracy of audit queries (Sect. 5).
- We present the tuple-correlated model (TC) for answering audit queries. Tuple-level correlations are captured by additional conditions appended to each tuple. We define the extended relational algebra for TC. We compare the expressiveness of TI and TC and prove that TC is a complete data model meaning that it can represent any possible set of answers (Sect. 6). We show the advantages of TI and TC in comparison to other models ("Appendix").[3]
- We discuss the complexity of deciding whether tuples are possible or certain in Sect. 7. In TI, this is given explicitly by an extra column. In TC, deciding that a tuple is possible is NP-complete and deciding that it is certain is coNP-complete. However, for a large subclass of instances, we show that efficient scheduling algorithms can determine possiblility in P.
- We implement our framework via extensions to Postgres, showing that uncertain answers can be computed efficiently over both models (Sect. 8).
- We demonstrate (through simulation on sample data) that useful auditing can be performed in the presence of retention restrictions, despite uncertain answers. The study of the impact of retention policies on the accuracy of query results under TI and TC shows cases where TC can significantly improve accuracy over TI (Sect. 9).

We describe our threat model, in Sect. 2, and our data model and queries, in Sect. 3. We distinguish our contributions from related work in Sect. 10.

[3] See supplementary material associated with the online version of this article on the journal's web site

2 Threat model and security objectives

2.1 Adversaries

Our threat model focuses on two major categories of adversary: auditors and external authorities.

An *auditor* is an authenticated user of the system who is permitted to ask queries about past events in the database. We use the single term *auditor* to refer to either an entity external to the enterprise, who is authorized to perform audit tasks, or a user internal to the enterprise, who wishes to compute analytics or monitor changes in the database. We assume auditors are not capable of subverting standard authentication procedures or access controls imposed by the compliance officer. In our framework, this means that the auditor is restricted to the sanitized data history only.

An *external authority* is an entity, such as a legislative body, a governmental institution, or a legal authority, capable of issuing audit queries that the enterprise is compelled to answer using all information available in the database. An external authority is not restricted by access controls imposed by the compliance officer. However, information that is physically removed from the data history will no longer be available to anyone, even the external authority. In addition, the external authority can issue a data hold to the compliance officer, preventing the compliance officer from removing specified data from the history.

2.2 Threats and security objectives

2.2.1 Data disclosure

The primary threat we address is unintended disclosure of the data history. When the compliance officer intends to protect portions of the data history through one or more retention policies, but that data history is nevertheless exposed to an auditor or external authority, then data disclosure has occurred. For example, in our motivating scenario, if Bob's salary is not appropriately sanitized by Policy 1, the answer to Query A1 may have Bob as a certain answer, resulting in a compromise of Bob's privacy.

2.2.2 Maintenance of data holds

We assume that if an external authority issues a data hold for a portion of the data history, the enterprise is required to retain that history for later audit queries by the external authority. Failure to comply with this requirement may result in significant liability for the enterprise, so we consider maintenance of data holds an important security property of our framework.

We consider avoiding data disclosure and respecting data holds as non-negotiable requirements of our framework, treating these as hard constraints that must be met. Subject to these constraints, we desire to provide the best utility and availability possible for auditing. In the best case, audit answers are precise. If they are not precise, the auditor may be faced with uncertainty about the actual audit query answer, but we nevertheless insist that answers be *sound*, so that they do not lead to false conclusions. These assumptions favor compliance over auditing and imply that some retention policies established by the compliance officer may not allow accurate auditing for some auditing queries. It is possible for the compliance officer to detect the interaction between a retention policy and audit query (see discussion in Sect. 4.3).

There are other enterprise security threats that are not the primary focus of this work. We assume that conventional methods are used to prohibit database clients from altering the data history or log in any manner other than through inserts, updates, and deletes on the current snapshot. We also assume that the compliance officer is trusted to implement policies correctly: we do not defend against the threat posed by an untrusted compliance officer intentionally altering the log. Such threats have received considerable attention elsewhere [10,17,18,31,35,39,42]. Lastly, while external authorities are capable of accessing any data stored by the enterprise, we assume they cannot carry out a full forensic examination of the enterprise system to reveal further data remnants that may be retained. Such threats have been considered by prior work [46] and we assume suitable countermeasures are employed.

2.3 Achieving security objectives

The retention policies described in this paper have a single well-defined semantics (described in Sect. 4). But applying the retention policies to the data history can be done in one of two ways: physically or logically. Logical implementation addresses the threat of data disclosure only with respect to auditors, but not external authorities. An advantage of logical implementation is that it is easy to modify or reapply the retention policy, and because data is not physically removed, logical application never conflicts with data hold requirements. Physical implementation, in which redaction and expunction result in physical removal of data, is more secure, addressing the threat of data disclosure for both auditors and external authorities.

3 Data model and audit queries

In this section, we describe our data model, based on backlog and transaction-time databases [21,22], and our language for expressing audit queries.

3.1 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. $tuples(S_i)$ is the set of all tuples that could occur in S_i (i.e., the cross-product of the attribute domains).

3.1.1 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 1 shows an example audit log table. In general, the schema of L_S is:

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

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 [22] 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.

3.1.2 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 $(p_1, \ldots, p_n, from, to)$ means that $(p_1, \ldots, p_n)$ holds at each instant t, for $from \leq t \leq to$. Table 2 is an example of a t-relation.

3.1.3 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 $exec^{-1}$, 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) deletion transactions. Notice that computing an audit log from T_S will reproduce a table similar to L_S, but with the audit fields removed: $\Pi_{ttime,type,S}(L_S) = exec^{-1}(T_S)$.

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.

3.2 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 [8,11], and instead present examples highlighting their features.

The example audit queries from Sect. 1.2 are expressed as follows on T_S or L_S:

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

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.*

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

A4. *Return all the time periods when Bob earns a salary of 10.* $\Pi_{from,to}(\sigma_{sal=10}(T_S))$

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 additional example query includes a concurrent self-join on T_S:

A5. *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 the 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 $\pi_{S-\{\text{from,to}\}}(\tau_{m..m}(T_S))$.

4 Describing and applying retention policies

In this section, we define the semantics of our redaction, expunction, and preservation rules, and discuss how they are applied to the stored history. When the implementation respects the semantics of these rules, the threats and security properties in Sect. 2 will be satisfied.

4.1 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 1 (*Expunction Rule*) An expunction rule, over a client table S, is denoted $\mathcal{E} = \text{Expunge}_S(\phi, [u, v])$ 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 $\mathcal{E}(T_S)$, this new t-relation consists of all facts from T_S *except* those that satisfy ϕ and have time field in $[u, v]$:

Definition 2 (*Expunction Rule Application*) For a client relation S, let T_S be a t-relation over S, and

$$\mathcal{E} = \text{Expunge}_S(\phi, [u, v])$$

be an expunction rule. The application of $\mathcal{E}$ to T_S, denoted $\mathcal{E}(T_S)$, is a new t-relation with the same schema: $\mathcal{E}(T_S) = T_S - \{x \in T_S \mid \phi(x) \wedge x.t \in [u, v]\}$

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 (*Redaction Rule*) A redaction rule, over client table S, is denoted $\mathcal{R} = \text{Redact}_S(\phi, \mathbb{A}, [u, v])$ 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 $\mathcal{R}$ is applied to a t-relation T_S, the intended result is a new t-relation, denoted $\mathcal{R}(T_S)$, in which some attribute values have been suppressed. To formalize $\mathcal{R}(T_S)$, we use a suppression function $\text{supp}(x, \mathbb{A})$, which replaces attributes of $\mathbb{A}$ in the transaction-time tuple x with variables. For example, if $x = (101, \text{Bob}, \text{Sales}, 10, 300)$ then $\text{supp}(x, \{dept, salary\}) = (101, \text{Bob}, dx, sx, 300)$. 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.

Definition 4 (*Redaction Rule Application*) For a client relation S, let T_S be a t-relation over S, and $\mathcal{R} = \text{Redact}_S(\phi, \mathbb{A}, [s, t])$ be a redaction rule. The application of $\mathcal{R}$ to T_S,

denoted $\mathcal{R}(T_S)$, is a new t-relation with the same schema:

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

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 $\mathcal{R}(T_S)$ 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 Sect. 5.

4.1.1 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 1 In Sect. 1.2, we described informally two retention policies. The redaction rule that *hides Bob's salary between time 0 and 250* is written formally as $R = \text{Redact}_S(\text{name='Bob'}, \texttt{sal}, [0, 250])$. The expunction rule that *removes the record of all employees in the HR department between time 0 and 300* is written $E = \text{Expunge}_S(\text{dept='HR'}, [0, 300])$. Table 3 is the t-relation that results from applying both E and R to the original table T_S shown in Table 2.

4.1.2 Suppression by variables versus NULLs

The choice to use variables instead of NULL values for cell suppression allows for improved audit accuracy, but can sacrifice confidentiality because it reveals when two redacted values are identical. For example, suppose Bob's salary was 10 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 10 at time y, in violation of the redaction policy.

Nevertheless, we believe this is a worthwhile trade-off and we show in Sect. 9 that the use of variables can substantially increase auditing accuracy for some queries. Our framework can easily be adapted to a suppression function using NULL values.

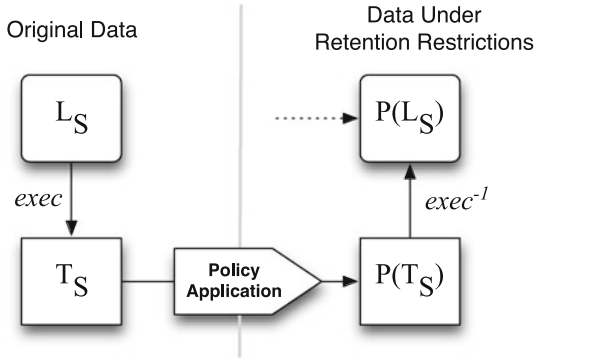

Fig. 1 Illustration of the relationships between original history (L_S and T_S) and the history under retention policy P. $P(T_S)$ is defined directly, while $P(L_S)$ is the sanitized log derived from $P(T_S)$ and including audit fields from L_S

4.2 Sanitizing the audit log

Consider a policy $\mathcal{P}$ consisting of redaction and expunction rules. According to the definitions above, we apply the policy to T_S to get the t-relation $\mathcal{P}(T_S)$. As we have seen in the examples of Sect. 1.2, the answers to audit queries are not determined completely by the table $\mathcal{P}(T_S)$. For one, the audit fields in L_S are not present. We must use L_S in combination with $\mathcal{P}(T_S)$ to answer queries that reference the audit fields. In addition, the operations applied to the database need to be inferred from $\mathcal{P}(T_S)$, 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 $\mathcal{P}(T_S)$, we compute a *sanitized log* consistent with $\mathcal{P}(T_S)$. This new log is denoted $\mathcal{P}(L_S)$ and has the property that running it results in $\mathcal{P}(T_S)$, that is: $exec(\mathcal{P}(L_S)) = \mathcal{P}(T_S)$. The auditor, and other users, will have access to both $\mathcal{P}(T_S)$ 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 Fig. 1.

When 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 $\mathcal{P}(T_S)$ (the defined meaning). This property defines the fundamentals of preventing data disclosure (in Sect. 2).
- 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 $\mathcal{P}(T_S)$ include the original history. In that

Table 4 A sanitized audit log, $P(L_S)$, transformed under the retention policies of Sect. 1.2 and Example 1

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	–	–	–

case, the true answer to any audit query must be a possible answer under retention restrictions. This property is essential for data utility (in Sect. 2), and it provides the basis for answering queries precisely.

Note that for any redaction rule $\mathcal{R}$ and expunction rule $\mathcal{E}$, $\mathcal{R}(T_S)$ and $\mathcal{E}(T_S)$ 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 5 (*Sanitized Log*) Let $\mathcal{P}$ be a retention policy consisting of redaction rules, expunction rules, or both, and let $\mathcal{P}(T_S)$ be the (possibly incomplete) t-relation that results from applying $\mathcal{P}$ to T_S. The sanitized log under $\mathcal{P}$ is denoted $\mathcal{P}(L_S)$ and is defined as follows:

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

This procedure first uses the $exec^{-1}$ to compute an audit log from $\mathcal{P}(T_S)$. Then we extract the audit fields and time column from the original audit log. This table, L_S^0, is then joined with $exec^{-1}(\mathcal{P}(T_S))$. We use a right outer join to preserve tuples in $exec^{-1}(\mathcal{P}(T_S))$, 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 2 Table 4 is the sanitized audit log computed according to the above definition, for the policy described in Example 1.

Note that Definition 5 is not itself an attractive strategy for computing the sanitized log. We describe our implementation of policy application in Sect. 8. In addition, we will see below that policies can be "applied" logically, in which case $\mathcal{P}(L_S)$ need not be materialized.

4.3 Retention policy analysis

We can show the following properties of the sanitized log.

Proposition 1 *Let L_S be an audit log, T_S the t-relation derived from it, and let $\mathcal{P}$ be a retention policy consisting of a set of redaction rules $R_1 \ldots R_n$ where each $R_i = Redact_S(\phi_i, \mathbb{A}_i, [u_i, v_i])$.*

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

Proof (Sketch) Soundness follows from that fact that $\mathcal{P}(T_S)$ is sound, and the fact that $\mathcal{P}(L_S)$ is consistent with $\mathcal{P}(T_S)$, in the sense that $exec(\mathcal{P}(L_S)) = \mathcal{P}(T_S)$. It follows that the original history is one possible world of $\mathcal{P}(L_S)$. If the condition $u_i, v_i \in \Pi_{ttime}(L_S)$ fails, then there are dangling tuples in the join described in Definition 5. 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 $\mathcal{R}(L_S)$ is consistent with $\mathcal{R}(T_S)$ and uses only the projection, L_S^0, of L_S. □

The sanitized log from Example 2 and Table 4 demonstrates 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.

4.3.1 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.

4.4 Supporting preservation rules

Redaction and expunction are removal rules. They implicitly indicate the semantics of data holds: the system only removes information that satisfies removal rules and retains the rest. Thus, when there is a litigation hold, we change the existing removal rules accordingly to prevent unwanted deletion. However, a specific preservation rule may provide more flexibility for compliance officers. Our framework is able to support tuple-level preservation rules. A preservation rule tells the system to retain all tuples matching the conditions in ψ for a specified interval of time.

Definition 6 (*Preservation Rule*) A preservation rule, over a client table S, is denoted $\mathcal{H} = \text{Presrv}_S(\psi, [u, v])$ where ψ is a Boolean condition on attributes of S, and $[u, v]$ is a time interval ($u, v \in \mathcal{T}$, and $u \leq v$).

When a preservation rule alone is applied to a t-relation, the t-relation is unchanged. When a preservation rule and a removal rule are applied together, the process of generating the new t-relation should ensure that those tuples matching the preservation rule are always retained, taking priority over the removal rule. To maintain data holds (described in Section 2), preservation rules must be applied correctly.

Definition 7 (*Preservation Rule Application*) For a client relation S, let T_S be a t-relation over S, and $\mathcal{P}$ be the current set of removal rules (expunction and redaction). If a preservation rule $\mathcal{H} = \text{Presrv}_S(\psi, [u, v])$ is added to $\mathcal{P}$ to get $\mathcal{P}_+ = \mathcal{P} \cup \{\mathcal{H}\}$, then we generate a new set of removal rules, $\mathcal{P}'$, from $\mathcal{P}$ by transforming each condition $p.\phi$ for $p \in \mathcal{P}$ into $p.\phi \wedge (\neg\psi \vee (p.t \notin [u, v]))$ where $p.t$ is the specified time period in original rule p. If $\mathcal{P}_+(T_S)$ denotes the application of all the rules, we then have: $\mathcal{P}_+(T_S) = \mathcal{P}'(T_S)$

The definition above defines the semantics of integrating preservation rules by logically transforming the original removal rules. It follows from the definition that preservation rules take precedence over removal rules. Further, the properties of removal rules and sanitization processes defined earlier in this section hold also for policies that include preservation rules.

4.5 Physical versus logical policy application

The discussion above has implicitly 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 it is necessary to defend against external authorities as well as auditors, or when privacy policies mandate direct removal of data.

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, which physical deletion is unable to support. 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.

By the semantics of preservation rules, we always physically enforce them. This fact applies to all preservation rules with no exceptions. But whether to logically enforce them is a choice made by compliance officers. For example, if preservation rules are the requirements of external authorities, the compliance officer first prevents physical deletion of the related tuples to maintain the data hold. At the same time, if he thinks that these preservation rules are beyond the auditor's accessibility, he will probably choose to ignore them in the logical application, which means allowing logical deletion of the tuples matching these preservation rules. By doing this, the system provides better privacy guarantees while still satisfying data holds. Alternatively, the compliance officer can add a data hold purely for the sake of preserving information of interest to auditors. In this case, he will enforce the preservation rule both physically and logically. But one should be warned that in this circumstance, data could also be exposed to external authorities, and thus, data disclosure is not prevented with respect to them.

Based on the discussion above, we believe a hybrid method of history sanitization, instead of purely physical or logical application, would better accommodate common scenarios. Assume $\mathcal{P}$ is the set of rules defined on the database, containing preservation rules $\mathcal{H}$ and removal rules $\mathcal{R}$. The compliance officer can adopt a hybrid approach as follows. Physical application is always executed on all preservation rules $\mathcal{H}$ since it does not prevent logical application of these rules later. Physical application on removal rules should be carefully chosen as logical sanitization is not available for them subsequently. So the officer selects a subset consisting of important removal rules $\mathcal{R}_1 \subseteq \mathcal{R}$ for physical sanitization. Next, in the logical application step, for each group of users/auditors, he decides what information to retain for them and enforces a corresponding subset of preservation rules $\mathcal{H}_1 \in \mathcal{H}$. For removal rules, he decides what to be deleted logically for each group of users/auditors and enforces a proper subset of non-physically applied removal rules $\mathcal{R}_2 \subseteq \mathcal{R} - \mathcal{R}_1$.

In Sect. 8, we describe the implementation of 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).

5 Audit queries under retention restrictions: a tuple-independent model

Under a retention policy that includes a redaction rule, audit queries must be evaluated over tables containing variables in place of some concrete values, that is, this table contains *incomplete information* or *uncertainty*. In this section, we discuss the tuple-independent model (TI), using techniques for querying incomplete information [20] to describe precisely the answers to audit queries under retention policies. The major benefit of TI is that there is no additional cost of deciding certain and possible tuples because extended relational operators can compute and label each tuple explicitly on the fly.

5.1 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 $\mathcal{P}$, the inputs to our audit queries are the audit log table $\mathcal{P}(L_S)$ and t-relation $\mathcal{P}(T_S)$. 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:

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

where F is the set of all one-to-one functions that assign values in the relevant domains to variables in V, 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 Sect. 3, an incomplete t-relation over S is a set 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 $(p_1, \ldots, p_n, from, to, u)$ which means that $(p_1, \ldots, p_n)$ has status u at each instant t, for $from \leq t \leq to$.

5.2 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 [5], 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.

5.2.1 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

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

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

Example 3 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 $\{\langle 100, 9, 9, P\rangle\}$.

5.2.2 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 $r.status \wedge s.status$.

5.2.3 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:

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

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

5.2.4 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**.

5.2.5 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 $\langle s.*, \mathbf{C}\rangle \in I_S$ where $r.*$ and $s.*$ shares the same value or variables on each attribute. Otherwise, write $\langle r.*, \mathbf{P}\rangle$ 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. The rest of the tuples in I_R that do not match the two cases above will remain unchanged in the result. 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 Recall from Sect. 1.2 that audit query A1 returns *all employees who earned a salary of 10 at some point in time* and can be written $\Pi_{name}(\sigma_{sal=10}(T_S))$. On the incomplete t-relation shown in Table 3 (for which the omitted status column is uniformly **C**), we have the intermediate result of $\sigma_{sal=10}(T_S)$:

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 $\Pi_{name}(\sigma_{sal=10}(T_S))$:

name	status
Bob	P
Chris	C

6 Audit queries under retention restrictions: a tuple-correlated model

As we have seen, in the query A4 of the motivating scenario, TI is incapable of representing answers accurately due to the failure of the tuple-independence assumption. In such cases, the P and C status is no longer enough to preserve a precise result. In this section, we introduce a tuple-correlated model (TC) for the purpose of more accurate auditing. TC achieves greater accuracy by maintaining correlations among tuples explicitly. It appends additional conditions to each tuple during query processing, instead of simply using "possible" and "certain" indicators. We will define the TC model and its relational algebra operations. Also we will show the benefit in terms of the expressiveness. However, the extra conditions make checking certain and possible tuples more complicated (remember in TI there is no additional cost for that), and we will discuss that in Sect. 7. Comparison with other models is investigated in "Appendix".[4]

6.1 Representing incompleteness

In the TC model, we associate the schema with an extra column *cond*. *cond* represents a conjunction of clauses, where each clause is a variable–variable or variable–constant comparison, for example, $X < Y$ and $Z > 5$. Consider a database D consisting of relations over schemas $R_1, R_2, \ldots$. Each schema $R_i = \{A_{i1}, A_{i2}, \ldots, A_{ij}, cond\}$. Let $\mathbb{A} = \bigcup A_{ij}$. We define a function $h : \mathbb{A} \to T$ to classify each attribute into some data type, where T is the set of all *data type*s. In TC, all the values (variables) in the same column have the same data type, and values (variables) are only allowed to compare with those of the same type. For example, attributes *salary* and *bonus* are of the same type and are comparable.

Side conditions $\eta(D)$ (or $\eta(I_R)$) are defined for database D (or relation I_R, the incomplete relation over schema R). $\eta(D)$ (or $\eta(I_R)$) is a conjunction of inequalities, which captures the distinctness among variables of the same type. The definition of the side condition conforms to the semantics of retention policies and captures constraints that apply to all the tuples, in contrast to tuple-level conditions in the *cond* column. $v(D)$ (or $v(I_R)$) represents all the variables involved in D (or I_R). For each variable x, $dom(x)$ represents the domain of the variable. Usually when a variable is corresponding to some attribute, $dom(x)$ is the domain of that attribute. Example 5 illustrates a relation r in TC.

[4] See footnote 3.

Example 5 $r=$

name	sal	cond
Bob	x	$x < 50$
Chris	y	true

$\eta(r) = \{x \neq y\}$, $v(r) = \{x, y\}$, $0 \leq dom(x) = dom(y) \leq 100$

An *assignment* for database D is a mapping from all variables in $v(D)$ to their domains, that is, $\forall x \in v(D)$, $f(x) \in dom(x)$. An assignment f for database D is *qualified* when $f \models \eta(D)$. A set of tuples S is a *possible world* represented by D if and only if there is a qualified assignment f for D and S is equal to the set of tuples when we replace all variables with values in D, that is, $f(D) = S$. Thus, the set of possible worlds represented by database D, denoted $rep(D)$, is defined as:

$$rep(D) = \{f(D) \mid f \mid = \eta(D)\}$$

Example 6 For the database in Example 5,

$$f = \{\langle x, 10\rangle, \langle y, 20\rangle\}$$

is a qualified assignment, therefore the possible world represented by f is $f(r) =$

name	sal
Bob	10
Chris	20

6.1.1 TC versus TI

TI allows variables but no tuple-level local conditions. The implicit constraint on distinctness of variables in TI is written explicitly by the side condition in TC. Another difference is that in TC you can specify domains of variables, while in TI, variables are always assumed to have infinite domains in order to simplify the constraints.

6.1.2 TC versus c-table

In a general c-table (conditional table) [20], each tuple is associated with a condition, which is a Boolean combination of equalities. A TC table can be viewed as an extended c-table with general local inequalities and special global conditions, as the side condition plus explicitly claimed variable domains can be written as global condition in a c-table. In our application, the side conditions are usually from retention policies, and tuple-level conditions are generated by queries, thus TC separates the two distinct types of constraints.

6.2 Extended relational algebra

We now describe a slightly different extended relational algebra compared to TI (Sect. 5.2), since we have to incorporate conditions in the query evaluation. The semantics of relational operators are defined as follows. Let I_R, J_R, and I_S be tables in database D. Note that the side condition of D remains unchanged after query evaluation.

$$\begin{aligned}
\Pi_A(I_R) &= \{\langle r.A, r.cond\rangle \mid r \in I_R\} \\
\sigma_E(I_R) &= \{\langle r.*, r.cond \wedge E(r)\rangle \mid r \in I_R\} \\
I_R \times I_S &= \{\langle r.*, s.*, r.cond \wedge s.cond\rangle \mid r \in I_R, s \in I_S\} \\
I_R \times^{\diamond} I_S &= \{\langle r.*, s.*, from, to, r.cond \wedge s.cond\rangle \mid \\
&\qquad r \in I_R, s \in I_S, [r.from, r.to] \cap [s.from, s.to] \\
&\qquad \neq \emptyset\} \text{ where } from = max(r.from, s.from), \\
&\qquad to = min(r.to, s.to) \\
I_R \cup J_R &= \{\langle t.*, t.cond\rangle \mid t \in I_R \vee t \in J_R\}
\end{aligned}$$

In projection (Π), we always preserve the *cond* column in the result. For those duplicates with the same non-*cond* attribute values but no *cond* formula, we could combine them into a single tuple by taking the disjunction of all *cond* formulas. In TC, we choose not to do this in order to keep each formula succinct. This does not change the semantics of queries and relations. In selection (σ), we return all non-cond attribute values (denoted as $r.*$) and extend the *cond* column by a conjunction with the selecting condition E, which itself is a conjunction. If the selection condition is $E = E_1 \cup E_2$, we will execute two selection operations followed by a union operation. Cross-product ($\times$) is defined by combining tuples in two inputs and taking the conjunction of their *cond* columns. Concurrent cross-product ($\times^{\diamond}$) is computed in a similar way as in TI model, plus the process on *cond* columns as in a normal cross-product defined above.

6.3 Expressiveness

In the context of incomplete databases, expressiveness measures the ability to represent sets of possible worlds. A data model is said to be *complete* [37] when it can represent any set of possible worlds. TI is not complete because it is impossible to represent a set of possible worlds in which two possible tuples are mutually exclusive, as shown in query A4 in the motivating scenario. However, TC is complete.

Theorem 1 *TC is a complete data model, that is, any set of possible worlds can be represented by a TC table.*

Proof Assume we have any set of possible worlds $W = \{r_1, r_2, \ldots, r_n\}$. Now we construct a table R in TC: we generate a new relation by adding each tuple in every possible world, as well as distinguishing them by appending *cond* condition $z = i$ for the ith possible world. Variable z has $dom(z) \in [1, n]$. Therefore, any qualified assignment $f(z) = i$ will only associate with the ith possible world. It is

obvious that the relation R represents the exactly same set of possible worlds of W. □

We say that model A is at most as expressive as model B ($A \preceq B$) if for any relation a in A there exists some relation b in B such that $rep(a) = rep(b)$ where $rep()$ denotes the set of possible worlds represented by the relation. A is as expressive as B if and only if $A \preceq B \wedge B \preceq A$. From the theorem above and the fact that TI cannot capture tuple correlations, we have the following corollary.

Corollary 1 *TC is more expressive than TI, that is, TI $\preceq$ TC and TC $\not\preceq$ TI.*

7 Complexity

For the TI model, as we have seen, certain and possible tuples are decided by the *status* column. However, we will show the problem of checking if a given tuple in TC is possible is NP-complete in Theorem 2. And Theorem 5 states that deciding a certain tuple is coNP-complete. However, we show in Theorems 3 and 4 that for a large subclass of instances, the possibility problem is in polynomial time. The certainty problem remains hard even within subclasses, therefore we use an exhaustive search with heuristics to compute certain tuples.

7.1 Deciding possible tuples

Given a TC table, there are usually tuples whose *cond* formulas are unsatisfiable, which means their existence is impossible. Computing possible tuples is the process of eliminating those unsatisfiable tuples. We begin with the definition of database satisfiability and possible tuples.

Definition 8 A database D in TC is *satisfiable* when it has a qualified assignment.

It is clear satisfiability of D is decided by checking satisfiability of $\eta(D) \wedge \bigwedge_{x \in v(D)} dom(x)$, where $\eta(D)$ is the side condition of the database, which indicates the distinctness among variables of the same type and $\bigwedge_{x \in v(D)} dom(x)$ defines all of the variable domains in D.

Definition 9 A tuple t is a *possible tuple* in database D when there exists a qualified assignment f such that $f \models t.cond$.

Recall that a qualified assignment f of database D satisfies $\forall x \in v(D)$, $f(x) \in dom(x)$, and $f \models \eta(D)$. Thus, it is easy to see that deciding possibility of the tuple t is equivalent to the satisfiability problem of the following formula:

$$\psi(t) = t.cond \wedge \eta(D) \wedge \bigwedge_{x \in v(D)} dom(x)$$

Of course, the satisifiability of database D is a necessary condition for the satisfiability of any of its tuples. When we know that D is satisfiable, we can simplify the above condition $\psi(t)$ by replacing D with the current tuple t, that is, $t.cond \wedge \eta(t) \wedge \bigwedge_{x \in v(t)} dom(x)$. Here $\eta(t) \wedge \bigwedge_{x \in v(t)} dom(x)$ are side conditions and domains only related to variables involved in $t.cond$. The simplified $\psi(t)$ is equivalent to the original one when database D is satisfiable: assignments to variables not in $t.cond$ will not change the satisfiability of tuple t. For simplicity, we assume that all of the variables in $t.cond$ have the same type.

Theorem 2 *Given a database D in TC, and tuple $t \in D$, deciding whether t is a possible tuple is NP-complete.*

The proof is a reduction from the clique problem, which is known to be NP-hard. The detailed proof is in "Appendix".[5]

We next show that two natural restrictions of the satisfiability problem can be solved in polynomial time. We avoid the richness of constraints that lead to the above NP-completeness by restricting the kind of constraints that can occur at the same time, namely we do not allow constraints to simultaneously express ordering, for example, $X < Y$, and distinctness from a given constant, for example, $X \neq C$. Recall that distinct variables are required to take distinct values. The two subclasses of TC corresponding to these restrictions are named $\text{TC}_<$ and $\text{TC}_{\neq}$.

Theorem 3 *Given a database D in $TC_<$, and tuple $t \in D$, deciding whether t is a possible tuple is in P.*

Proof We first rewrite the $\psi(t)$ as an *H-representation* of tuple t, H_t, consisting of two different sets of inequalities $H_{t,1}$ and $H_{t,2}$. 1) $H_{t,1}$: inequalities like $X < Y$. Since inequality $X < Y$ is equivalent to $X \leq Y - 1$, we only need $<$ to represent the relationship between variables ($X \neq Y$ is implicit since we have X and Y the same type). These inequalities define a topological ordering of variables. 2) $H_{t,2}$: inequalities like $X \in [X^L, X^R]$. The lower bound of X is noted as X^L, while X^R is the upper bound. To compute the lower and upper bound of each variable, we take advantage of the transitive property of $<$-relationship. For example, if $X > 5 \wedge Y > X$, we have $Y > X > 5$, and because $X \neq Y$, we further have $Y^L = 7$. That is, if $X < Y$, Y^L will be updated to $max(Y^L, X^L + 1)$ and X^R is updated to $min(X^R, Y^R - 1)$. This can be done by selecting variables in a topological ordering and inverse topological ordering.

It is clear that the H-representation H_t is equivalent to $\psi(t)$. Now we are ready to create a scheduling problem such that there are n unit-time jobs (n equals the number of variables in H_t) with release times (X^Ls in $H_{t,2}$), deadline times (X^Rs in $H_{t,2}$) and arbitrary precedence constraints

[5] See footnote 3.

(defined by $H_{t,1}$). It is easy to see finding a feasible schedule for this problem is equivalent to our tuple satisfiability in $\text{TC}_<$. Since computing H_t is in P and finding a feasible schedule for this problem is in P [24] , the tuple satisfiability can be solved in polynomial time. □

Theorem 4 *Given a database D in $\text{TC}_{\neq}$, and tuple $t \in D$, deciding whether t is a possible tuple is in P.*

Proof Consider the H-representation of $\psi(t)$, different from H_t in $\text{TC}_<$, we will have an empty $H_{t,1}$ since there is no variable comparison and each variable has a union of sets of intervals in $H_{t,2}$, instead of a single interval as in $\text{TC}_<$. For example, $X > 1 \wedge X < 10 \wedge X \neq 5$ will result in $X \in [2, 4] \cup [6, 9]$. Now we are ready to create a scheduling problem such that there are n unit-time jobs (n equals the number of variables in H_t) with multiple release and deadline times (defined by the intervals in H_2). It is easy to see that finding a feasible schedule for this problem is equivalent to our tuple satisfiability in $\text{TC}_{\neq}$. Since computing H_t is in P and finding a feasible schedule for this problem is in P [40], the tuple satisfiability can be solved in polynomial time. □

Remark 1 Recall that we simplify tuple satisfiability by assuming the database is satisfiable. Actually, database satisfiability is a special case of Theorem 4 because the satisfiability of formula $\bigwedge_{x \in v(D)} dom(x) \wedge \eta(D)$ does not contain inequalities like $X < Y$. Thus, deciding database satisfiability can be done in polynomial time. In addition, when uncertainty is generated by applying retention policies defined in this paper, the database is always satisfiable.

Remark 2 Consider the subclass $\text{TC}_{=\neq}$ consisting of TC restricted to conditions of the form $X \neq C$ and $X = C$ in $t.cond \wedge \bigwedge_{x \in v(t)} dom(x)$. Such conditions are common for variables in enumerative domains in which there is no ordering among values, for example, department type. Recall that $\text{TC}_{\neq}$ allows all kinds of variable–constant comparisons, but not variable–variable comparisons. Since $\text{TC}_{=\neq}$ is a special case of $\text{TC}_{\neq}$, we could use the algorithm for $\text{TC}_{\neq}$. Nevertheless, we can do it faster using an alternative method. Since there is no ordering among variables and constants, all variables in $t.cond$ are treated equally. We can randomly pick one variable X and assign it a qualified constant C such that C has not been assigned to other variables and $X \neq C$ does not exist. If all variables are assignable, then it is satisfiable, otherwise it is not.

Remark 3 When there are multiple variable types for tuple t, we classify inequalities by data types. As long as the constraints concerning each distinct data type fall entirely in $\text{TC}_<$ or $\text{TC}_{\neq}$, we can always compute possible tuples in polynomial time. For example, $t.cond \equiv X_{sal} < Y_{sal}, Z_{dept} \neq$ "HR" where $X_{sal} < Y_{sal}$ is in $\text{TC}_<$ (the salary type) and Z_{dept} is in $\text{TC}_{\neq}$ (the department type).

Remark 4 A combination of $\text{TC}_<$ and $\text{TC}_{\neq}$ provides adequate expressiveness for our purposes. $\text{TC}_<$ is well designed for ordered domains (e.g., integer domains like *salary*) and $\text{TC}_{\neq}$ is suitable for unordered domains (e.g., enumerative domains like *department*). If we consider the WHERE clauses of the TPC-H queries, each can be represented in $\text{TC}_<$ and $\text{TC}_{\neq}$ under any of our retention policies. This suggests that in many cases, NP-hardness is not a practical problem. Nevertheless, when the general TC model cannot be avoided, we have to search the space for satisfiable assignments, and the complexity bound is exponential in the number of variables, n, which is bounded by a property of the schema. Variables are generated by redaction policies, and n cannot exceed the number of columns belonging to the same data type, thus we expect to see n very small. In the TPC-H workload, n cannot be larger than ten. With this small number of variables, complexity exponential in n will be feasible and add very limited additional burden when compared to $\text{TC}_<$ and $\text{TC}_{\neq}$.

7.2 Deciding certain tuples

A tuple is certain when it occurs in every possible world represented by the database.

Definition 10 Suppose we are given a database D, and the set of possible worlds represented by D, $W = \{f_1(D), f_2(D), \ldots\}$ where $F = \{f_1, f_2, \ldots\}$ is the set of all qualified assignments for D. A tuple t is *certain* iff it exists in every possible world $f_i(D)$, for any $f_i \in F$.

As each possible world contains only constants, we can infer that a certain tuple contains no variables. In addition, if a tuple exists in every possible world, its *cond* formula should be always satisfiable for all qualified assignments. To compute the certain tuples in a TC table, we have a two-step process. First, we compute the *certain v-tuples*. A certain v-tuple is a relaxed version of a certain tuple, meaning its *cond* formula is always satisfiable but it could have variables in some columns. We merge tuples with the same non-*cond* column into a new tuple t and generate the new $t.cond$ formula by making a disjunction of all the *cond* formulas. Then if $f(t.cond) = true$ for every qualified assignment f, t is a certain v-tuple. Second, we transform certain v-tuples to certain tuples. Obviously, a certain v-tuple is a certain tuple when it does not contain variables. When t has a variable on attribute A, it can be transformed to a certain tuple if and only if there are another $|dom(A)| - 1$ certain v-tuples with different variables of A. In other words, there are at least $|dom(A)|$ certain v-tuples with distinct variables of A, therefore, by distinctness of variables of the same data type, each corresponds to a certain tuple. When the size of database is

unbounded, the difficulty of the first step dominates, since step two can be computed efficiently.

Theorem 5 *For each variant of the TC model discussed above, namely TC, $TC_{<}$, $TC_{\neq}$ and $TC_{=\neq}$, the tuple certainty problem is coNP-complete.*

The proof involves reductions from the 3DNF tautology problem to prove the coNP-completeness for the TC classes. A detailed proof is in the "Appendix".[6]

We can do an exhaustive search to detect certain tuples by a backtracking method. We choose a variable, assign a value, and then simplify the formula, recursively checking if it is still a certain tuple. Assume a variable's *valid intervals* consists of the union of all intervals in which the variable could find a qualified assignment that makes at least one of the conditions in *cond* true. One necessary condition for a certain tuple is that each variable should have its valid intervals equal to its domain. Thus, in each recursive step, if any variable has a smaller valid interval than the domain, the tuple is not a certain tuple. This heuristic will help to eliminate non-certain tuples quickly.

The worst case complexity of calculating certain tuples is primarily determined by a term exponential in the number of variables n. Similar to our discussion in Remark 4 about possible tuples, n tends to be a small number. Thus, in practical cases, for example, for schemas like TPC-H, the overhead added here is limited.

8 Implementation

The implementation of our framework 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. Note the earlier implementation described in [28,29] only includes the TI model.

As a performance optimization, both the audit log and the transaction-time tables are stored in our implementation. As noted earlier, the transaction-time tables are redundant since they can be computed from the audit log. However, materializing these tables and maintaining them upon changes to the log eases query expression and evaluation for some audit queries. The efficiency gains seem well worth the space overhead, which is roughly double that of storing the audit log alone. 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 TI, $status$ is represented as a Boolean value. In TC, we split the conjunction in the *cond* formula and put each inequality (or equality) into a text value column.

Recall that the application of policies can be executed either physically or logically (see the discussion in Sect. 4.5). In the remainder of the section, we discuss the physical application of retention policies followed by query evaluation on physically sanitized data sets. Then we describe the logical application of policies. Lastly we discuss the computation of possible and certain tuples.

8.1 The physical application of retention policies

The application of retention policies is implemented by transforming the input rules into a set of update operations on the original t-relation and possibly the audit log. Inconsistencies may arise if the subsequent application of new policy rules touches the previously sanitized attributes [3,41]. For example, one policy p_1 removes the department information and the other policy p_2 hides employees' salary in the HR department. Applying p_1 first will result in a different sanitized history than if p_1 is applied second. In this example, p_2 will remove nothing if p_1 is already applied; however, the sanitized history will be different if p_1 is applied first. To avoid this, we assume we have all the policy rules at the time of policy application. Policy application for all rules is accomplished in one-pass scanning of the table, sanitizing each tuple against all rules, which guarantees that all the conditions in the rules are fully evaluated on the current tuple before removing any values from that tuple. For example, if an employee is in the HR department, both his salary and department information will be deleted when we have p_1 and p_2.

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. Specifically, each data type has a distinct hash function, which allows consistent variable assignments on the same values even across multiple tables. Remember that we define the data type when introducing TC in Sect. 6, and this concept can also be applied on TI. As a data type only relates to the comparability and does nothing with the domain, another benefit of utilizing a hash function for each data type is to enable comparison of variables that belong to different attributes, for example, comparing *salary* and *bonus*.

Since the policies are specified over t-relations, a policy $\mathcal{P}$ with an arbitrary time condition $[u, v]$ may require a split of update intervals causing phantom updates in the sanitized

[6] See footnote 3.

log (as demonstrated from Example 2 and Table 4), which results in residual disclosure and false conclusions in query evaluation (meaning audit answers will no longer be sound). To avoid this, we adjust the redaction period to the nearest modification period of any field. However, this method might be too restrictive and hinders periodic policy application, especially when the nearest modification period is much longer than that required by the retention policies. To favor practicality and periodic application but still achieve no residual disclosure and soundness, one possible approach is to impose a system-wide "soft" limitation of the active period of time for each tuple. As a tuple's active period is defined as $to - from$, the requirement ensures that no tuple is going to stay in the current snapshot of the database longer than the limit *roughly*. For example, if Bob's salary remains unchanged for about one year, which reaches the limit of a tuple's active lifetime, the system will input a new tuple with the same salary starting from some randomly selected date and archive the old tuple in the history. In the log table, all audit fields of the new tuple are copied from the last update. Thus, we can align the time with finer granularity and apply policies periodically. This randomly cyclic strategy preserves the soundness of query answering as long as we treat the system's behaviors as true updates of the history. Moreover, by assigning consistent variables on the same values, auditors can still correctly monitor the changes to values.

8.2 Audit query evaluation

Next we implement in SQL the semantics of extended relational operators over incomplete relations for both TI and TC. The basic strategy is to rewrite SELECT-FROM-WHERE blocks to accommodate incompleteness.

8.2.1 The TI model

To get uncertain answers for any given user query, the query evaluator runs over the rewritten version of that query. During query processing, it retains tuples whose *status* column evaluates to either **P** or **C** on the original WHERE clause and eliminates all others. Then it computes the correct *trange* (if necessary), the status column, and appropriate values of variable bindings for the query results.

In the following algorithm, 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. $binds(x, y)$ represents the value bindings, described in Sect. 5. It outputs x if x is a constant, otherwise it outputs y. In addition, to simplify the representation, we assume that the WHERE clause of the user query is always a conjunction of multiple condition expressions. If there are attributes appearing in two conditions connected by the OR operator, for example, `sal=10 OR sal=20`, we can break the query into parts and later combine their results. The algorithm for rewriting user queries is as follows:

1. WHERE clause: rewrite each condition by the following rules. $T.A$ stands for attribute A in table T. $\theta \in \{=, \neq, <, \leq, >, \geq\}$. c, c_1 and c_2 are constants.

$$A\ \theta\ c \Rightarrow (A\ \theta\ c\ \mathsf{OR}\ isvari(A)) \quad (1)$$

$$\text{Let } Z \equiv (A\ \theta\ B\ \mathsf{OR}\ onevari(A, B))$$

$$A\ \theta\ B$$

$$\Rightarrow Z \qquad (\text{if } \theta \in \{=, \leq, \geq\}) \quad (2)$$

$$\Rightarrow Z\ \mathsf{AND}\ A\ !=\ B \quad (\text{if } \theta \in \{<, >\}) \quad (3)$$

$$\text{if exists } T_1.A\ =\ c_1 \text{ and } T_2.A\ =\ c_2\ (c_1 \neq c_2)$$

$$\Rightarrow \text{append } T_1.A\ !=\ T_2.A \quad (4)$$

   The general idea of rewriting a condition is to allow the query processing to keep not only those tuples satisfying the condition but also those that could *possibly* satisfy the condition when variables are involved. Rule (1) tells the query evaluator that when A is a variable it will also retain the tuple. When comparing two attributes A and B, by rule (2), the answer is yes when $A\theta B$ is true, or one of them is a variable. If both of them are variables and the comparison is $<$ or $>$, we additionally make sure they are two different variables, by rule (3). Similarly, in rule (4), the same attribute in different tables is compared with different concrete values. Finally, we also add conditions on *trange* when necessary.
2. SELECT clause: for each column A in the original SELECT clause, we rewrite it by the following rules. Assume W is the original WHERE clause.

$$\textbf{If } A \text{ is } status: \quad (5)$$

$$\Rightarrow (W\ \mathsf{AND}\ T.status)\ \mathsf{AS}\ status$$

$$\textbf{Elseif } A \in W \text{ and exists } T.A = c: \quad (6)$$

$$\Rightarrow c\ \mathsf{AS}\ A$$

$$\textbf{Elseif } A \in W \text{ and exists } T_1.A = T_2.A: \quad (7)$$

$$\Rightarrow binds(T_1.A, T_2.A)\ \mathsf{AS}\ A$$

$$\textbf{Else} \Rightarrow A \quad (8)$$

To calculate the *status* column, as shown by rule (5) above, put the original WHERE clause into SELECT clause and add a conjunction of related status columns to the term. Rule (6) ensures the concrete value is returned if there is an equality condition on that column. We must rewrite those columns when they appear in both the SELECT list and some equality expression in the WHERE clause, in order to make sure query evaluation returns the concrete value as shown in rule (6), or the correct variable bindings for the selection as shown

in rule (7). Finally, compute the correct *trange* value if necessary (i.e., for concurrent join).

Example 7 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 t1.dept,
       200 AS t2.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
```

We first apply rule (1) and (4) to generate the AND-terms in the new query since there are `t1.sal = 100` and `t2.sal = 200`. Rule (2) is also applied on `t1.dept = t2.dept`. Rule (8) keeps `name` in the selection list. We have `200 AS t2.sal` and `binds... AS t1.dept` by rule (6) and (7). Finally, we use rule (5) to calculate *status* column in the selection list.

As discussed in Sect. 5, 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.

8.2.2 The TC model

We apply a similar rewriting process as we used in TI. The difference is how to generate *cond* formulas in the result and eliminate unsatisfiable tuples.

Example 8 Given a query asking for employees whose bonus is more than his salary, we rewrite it as follows:

```
SELECT name,
     CASE WHEN isvari(sal) or isvari(bonus)
          THEN '[money]'|| sal ||'<'|| bonus
          ELSE NULL
     END AS cond_1
FROM emp
WHERE (sal<bonus OR onevari(sal, bonus))
     AND sal!= bonus AND
     check_sat(cond, array[
          '[money]'|| sal ||'<'|| bonus])
```

In the SELECT clause, we insert a case statement to output *cond* formulas with each condition recorded in a single column. In the example, as shown in the SELECT clause above, we refer to it as `cond_1`. Because the query is comparing *salary* with *bonus*, the condition only exists in the result when at least one of them is a variable. For example, if salary is z and bonus is 10, by the CASE statement, the produced inequality will be `[money] z<10`, because salary and bonus both belong to the data type "money". The number of inequalities generated, which is the number of CASE statements needed, is determined by the length of the original WHERE clause. (Actually we could reduce the size of the original WHERE clause by the process of computing the H-representation described in the proof of Theorems 3 and 4 when we treat each attribute name in the WHERE clause as a variable).

In the WHERE clause, as the last step before results are passed to the SELECT clause, a customized function *check_sat* is called to check tuple satisfiability by inputing two parameters: the current *cond* formula and inequalities formed by the condition in the original WHERE clause.

8.3 Logical policy implementation

The implementation above is based on the physical removal of expired information. We can also implement policies logically, or virtually, without altering the stored contents of the database. A query Q is not evaluated on the underlying database directly, but is first composed with the policy $\mathcal{P}$ to generate a rewritten query $Q_{\mathcal{P}}$. The rewritten query can be evaluated safely on the base relations and produces a result equivalent to evaluating $\mathcal{P}$ on a physically altered database.

For simplicity, we assume that the redaction policies satisfy the condition in Proposition 1 of Sect. 4.3, so that their application is sound and secret. Generally the composition will begin by adopting the rewriting algorithm in the previous subsection. Attributes appearing in either the SELECT or WHERE clause are called *critical attributes*. A redaction rule is *relevant* to Q when its redaction attribute list shares some attribute with Q's critical attributes. In addition to the rewriting process in the previous subsection, we also make the following changes:

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 [25], but we change the semantics from NULLs to variables.

8.4 Improving query evaluation in TC

In Sect. 7, we discussed how to decide possible and certain tuples given a TC table. Consider an incomplete history D

generated by the application of retention policies, and suppose we wish to compute possible and certain tuples of query q over D. Since D has no *cond* column, the conditions in the results will only be introduced by q's WHERE clause. In fact, the size of the WHERE clause might be reduced by computing its H-representation, described in the proof of Theorems 3 and 4. Each H-representation H_t contains two sets of inequalities: $H_{t,1}$, inequalities like $X < Y$ and $H_{t,2}$, inequalities like $X \in [X^L, X^R]$. The size of $H_{t,1}$ is bounded by $O(n^2)$ and the size of $H_{t,2}$ is bounded by $O(n+c)$ where n is the number of column names in q and c is the number of $\neq$-inequalities. So if there are few or no $\neq$-inequalities in q, the size of WHERE clause (the number of inequalities introduced by q) is usually very limited when the number of columns in the database D is small.

In TC, possible tuples can be checked during query processing and certain tuples are computed as a separate step after query evaluation is finished. To improve the performance, we may be able to take advantage of static analysis on the original query before rewriting and executing it on the database system. A simple example is that if a query q has only $=$-comparisons, its possible tuples are the same under TI and TC. Moreover, if this q also contains only columns not touched by retention rules, which means returned results are always complete, TI and TC result in the same set of certain tuples.

It is also possible to predetermine the satisfiability of results for some queries. Consider the WHERE clause as a formula. We first replace each attribute name with a different variable of that data type. If there exists equality between two variables $x = y$, we replace all occurrences of y with x (this is important because of the distinctness among variables of the same type). Now it is obvious the result is an empty set (no possible tuples) if this formula is not satisfiable. When this formula is a tautology (for any qualified assignment), and all the columns in the WHERE clause are removed together by redaction policies, then all possible tuples in the result are certain v-tuples. (Recall that certain v-tuples are tuples with variables and an empty *cond* column, defined in Sect. 7.2.)

9 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:

Performance. We assess the performance overhead of evaluating audit queries using both physical and logical policy application on TI and TC.

Accuracy of uncertain answers. We study the impact of retention policies on the accuracy of query results under TI and TC. Over sample data, we measure the precision and recall of query answers as a function of the selectivity of redaction policies. We characterize the cases where accurate auditing can be achieved under retention restrictions. And we show that TC can improve the accuracy significantly over TI in some cases. We also compare the accuracy with suppression only using NULLs. Using NULLs is a common solution in relational database research such as fine-grained access control [25]. 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.

9.1 Experimental setup

In all our experiments, we use Postgres 8.3 running on an Intel Core2 machine with 2.26 GHz CPU and 4Gb memory. Our data sets are synthetically generated histories based on our example client schema S(eid,name,dept,sal,bonus). Here *sal* and *bonus* have the same data type that allows comparing between them.

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 *original snapshot size*) and the average number of versions of each employee tuple (the *history length*). We measure the query execution time by reporting the average of 10 runs with the largest and smallest runs omitted.

9.2 Performance

We use three redaction policies[7] and five queries in our experiments. They are:

- R1: (HideSal) Redact salary values for a set of departments d_{s1} before a specified time t_1.
- R2: (HideBonus) Redact bonus values for a set of departments d_{s2} before a specified time t_2.
- R3: (HideDept) Redact department values in a specified time period p_1.
- Q1: (GetAll) Return the whole emp table.
- Q2: (GetEmp1) Return employees who are in department d_1 and have salary m_1.
- Q3: (GetEmp2) Return employees' information where salary is less than m_2 and bonus is larger than m_3.
- Q4: (GetCowker1) Return all employees who worked in the same department as a specific employee e at the same time.

[7] We do not consider expunction and preservation rules since they will simply remove or preserve tuples and change the size of the history.

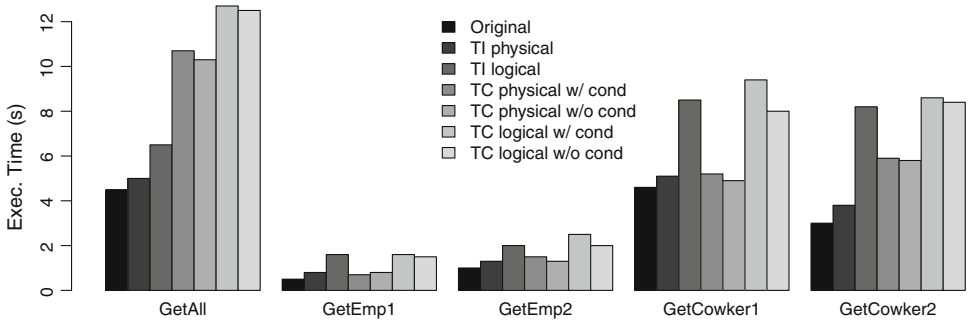

Fig. 2 Performance of the five queries. For each query, the *bars* from *left* to *right* represent the execution time on different models. "physical"/"logical" means policy application is implemented physically or logically. "w/" or "w/o cond" decides whether the result will contain condition formulas

Q5: (GetCowker2) Return all employees who earned more bonus than their salary and worked in the same department with a given employee e, at same period of time, as long as the returned employees have a smaller salary than e, but a larger bonus than e.

We measure the execution time of each query under *physical* and *logical* implementation of TI and TC models. For the TC model, we also consider the case of returning results with and without *cond* formulas. The baseline (*original*) is the time to compute the audit query without the retention policy, that is, on the original tables. In GetEmp1, d_1 is in the set d_{s1}, and thus, there is uncertainty in the answers due to HideSal and HideDept. Note that GetAll has no WHERE clause and GetEmp1 and GetCowker1 only contain $=$-comparison; therefore, they can be answered accurately by TI model, and thus a satisfiability check by function call in the database system is not necessary in TC. In GetEmp2, we set $m_2 < m_3$, for example, $m_2 = 10$ and $m_3 = 40$. Consider an employee has the same salary and bonus, both redacted to variable x. Then he will not be a qualified result for GetEmp2, because $x < 10 \wedge x > 40$ is unsatisfiable. GetCowker2 has a more sophisticated situation. So rewriting GetEmp2 and GetCowker2 in the TI model can not produce results accurately. Only the TC model is able to answer these two queries properly. The execution time on a history (roughly one million tuples) with 10,000 initial employees (snapshot size) and 100 versions for each one (history length) is illustrated by Fig. 2. Generally, 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 the TC model, the online satisfiability check is implemented as a function in the plpython language in PostgreSQL. To estimate the overhead of the function call in PostgreSQL, we create a fake satisfiability check function in plpython that does nothing but returns a True value and insert it into GetAll's WHERE clause in TC. As GetAll returns all one million tuples, the system will execute the fake function on each of them, which results in a lot of the extra cost for TC model. Considering this cost is introduced by the system and the size of result, we consider this overhead acceptable. It would be possible to reduce this overhead by using more efficient native language of the database system.

Since GetEmp1 and GetCowker1 only contain equality comparisons, checking tuple satisfiability is not needed, and the only difference between TI and TC is the way they generate status and *cond* columns. As expected, the performance is very close between these two models for both physical and logical implementations. GetEmp2 and GetCowker2 add an extra cost of checking tuple satisfiability. Each tuple of GetEmp2's result has a condition consisting of at most two comparisons. In GetCowker2, there are at most four variables, and the size of the WHERE clause is about eight. For these two queries, Fig. 2 shows that computing possible (satisfiable) tuples in TC adds a modest extra cost to TI when we take into consideration the cost of the system call discussed above. When TC does not include the *cond* column in the result, the performance is closer to TI.

In addition, the logical solution is uniformly slower than the physical because of the more complex queries required when policies are composed with queries. Another reason is the lack of indexes. When a query is logically rewritten, the only usable index is the one built on *ttime* column. A possible optimization is only integrating relevant policies into query rewriting, for example, in GetCowker1, redaction rules for removing two salary columns can be omitted from logical queries.

It is worth noting that the certain tuples alone can be computed more quickly than the original result in TI [29]. This is because, given the rewritten query, computing certain tuples can ignore variables, and the certain tuple set returned tends to be smaller than the true result. In TC, computing certain tuples is a separate step after query evaluation is done, and the execution time could be slow, depending on the complexity of formulas and the number of duplicates.

9.3 Accuracy of uncertain answers

Next we evaluate experimentally the accuracy of audit query answers under retention policies. We demonstrate the cases that TC and TI are at the same level of accuracy and the cases

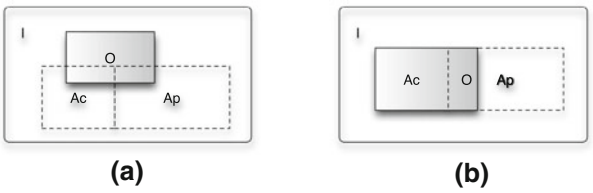

Fig. 3 Result relationship in Venn Diagram: The answer space is I (*the largest box*) and the original answer are O (shaded box), the certain tuples in our model are A_c, the possible tuples is A_p (both are boxes with *dotted-line*). **a** General case. **b** Guarantee of precision = 1

when TC improves upon TI. 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 qualified tuples and disqualified tuples. Under retention restrictions, an audit query partitions the set of feasible answers into certain tuples, possible tuples, and disqualified tuples.

Our first measurement of accuracy considers the distribution of answers as a function of the selectivity of the redaction policies. The second measurement is the *precision* and *recall* of our answers with respect to the original answers. Assume the answer space is I and the original answer is O. The certain tuples in our model are A_c, the possible tuples are A_p. For simplicity, we assume no variables in the possible tuples. Intuitively, we want to know how large $O \cap A_c$ (Fig. 3a) is 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 by $\frac{O \cap A_c}{O}$.

We can also define precision and recall of the disqualified tuples, which may be relevant to auditors since they might have value in an investigation. Then $I - O$ contains the disqualified tuples in the original answers and $I - A_c - A_p$ is the set of disqualified tuples computed in the incomplete history. The precision of disqualified tuples is defined $\frac{(I-A_c-A_p) \cap (I-O)}{I-O}$ and recall of disqualified tuples is defined $\frac{(I-A_c-A_p) \cap (I-O)}{I-A_c-A_p}$. If we consider sound and secret retention policies, as described in Sect. 4, then the precision of certain and disqualified tuples is always equal to 1, shown in Fig. 3b, because the soundness (Proposition 1) guarantees $A_c \subseteq O$ and $O \subseteq A_c \cup A_p$.

The first experiment is performed on GetCowker1 in the previous section (the concurrent self-join). The query answers in TI and TC will have exactly the same set of possible and certain tuples, although they will differ in the way they represent the condition. The answer distribution and recall are shown in Fig. 4a, b. At the beginning, there are no possible answers against the original history, and thus, the recall of the certain and disqualified tuples is 1. When there are values removed by retention rules, possible answers are introduced. The percentage of possible tuples and the recall of the certain and disqualified tuples all have an inflection point as the selectivity goes up. This is because, when the removal rate is low, fewer variables are introduced so we can retain a high recall. When the rate increases, the number of variables increases and the recall decreases. On the other hand, when the rate is extremely high, the incomplete history is mostly replaced with variables on the join attribute: department. We will get high recall since the equivalence among variables can be inferred accurately, for example, two employees both working in HR department result in the same variable x in their department attribute. Therefore, there are fewer possible answers and we get very high accuracy when all the department information is removed, similar to the answers under the original history.

There are also many queries where TC can obtain much greater accuracy. Figure 4c, d show the difference in recall of disqualified tuples for TI and TC given two queries. (We omit the answer distribution figures here.) Since possible tuples in the uncertain results are always coming from the disqualified tuples in original results, these two figures actually illustrate the difference in size of the possible (satisfiable) tuples. That is, TI will output more tuples, which should be unsatisfiable and eliminated. In Fig. 4c, we remove history by deleting salary values. When salary is replaced with a variable X, the *cond* condition in the result will be $bonus < X < 10$, that is, the tuple is possible only when bonus is less than 10. In TI, this fact cannot be captured. Therefore, when the number of removed $salary$ attributes increases, the size of possible tuples grows and finally all the tuples are possible when the removal rate reaches 100 %. However, in the case of TC, the size increases linearly because salary and bonus are generated randomly, and the probability of bonus less than 10 is independent of the removal of salary. In Fig. 4d, the WHERE clause in the query is an unsatisfiable formula. Thus, no matter how we redact salary or bonus individually or jointly, TC will always return an empty result (as does the query over the original history) while TI increases quickly when we remove more. Note that when all of the salary and bonus attributes are redacted, TI does not return all the tuples because it eliminates the tuple where salary and bonus are replaced with the same variable. In fact, we can actually use the static analysis discussed in Sect. 8.4 for TI to avoid this but we can do nothing for the case of Fig. 4c.

Figure 4e, f show the recall of certain tuples for two different queries. In Fig. 4e, with condition $salary < bonus$, a certain tuple in TC has conditions such as $X < Y \vee Y < X$, or $X < 10 \vee 8 < X$ after merging tuples with identical non-*cond* columns. Thus, we can expect a smooth reduction when the removal rate increases. The result shows ten thousand certain tuples in the original history and half of the tuples in the case when all of the salary and bonus are redacted. For TI, the size of the certain tuples is decreasing in proportion to increasing redaction and ending at 0. In Fig. 4f, according to discussion in Sect. 8.4, the WHERE clause becomes a tautology when $salary$ and $bonus$ are redacted jointly. In this case,

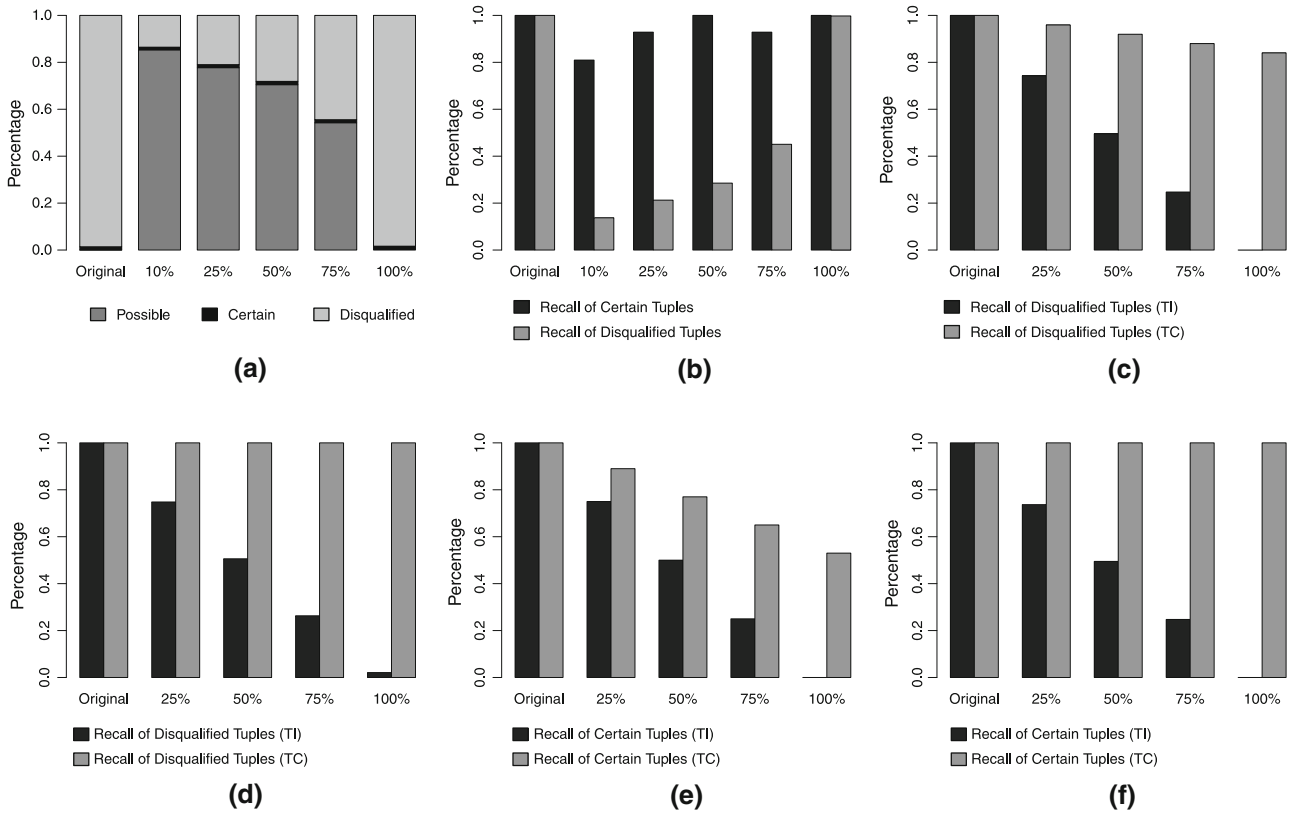

Fig. 4 Accuracy of uncertain answers. We measure the accuracy (Y-axis) in terms of the removal rate of values in the history (X-axis) defined by redaction rules. In **a** and **b**, we use the redaction rules defined in the previous section. **c** is performed under a rule that removes only $salary$. **d**–**f** have the same redaction rule that deletes $salary$ and $bonus$ together. **a** Answer Distribution (GetCowker1). **b** Recall of Answers (GetCowker1). **c** Recall of disqualified tuples, given q = select * from emp where salary > bonus and salary < 10. **d** Recall of disqualified tuples, given q = select * from emp where salary > bonus and salary<10 and bonus > 40. **e** Recall of certain tuples, given q = select eid, dept from emp where salary < bonus. **f** Recall of certain tuples, given q =select eid, dept from emp where salary = bonus and (salary < 10 or bonus ≥ 10)

TC can capture the full semantics of the query no matter how much salary and bonus is removed. As expected, TI's result becomes worse when more salary and bonus is removed.

9.3.1 Suppression using variables versus NULLs

In our final experiments, we apply redaction policies with a suppression function that uses NULL values instead of variables. In the "Appendix"[8] we show that using nulls in TI, called TI^{null}, has the same expressive power as Gadia's model [12] for incomplete temporal databases. Figure 5 shows the recall of certain and disqualified tuples on GetEmp2 (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.

10 Related work

Retention policies and problems of expiring historical data have been studied in a variety of contexts. Garcia-Molina et al. [13] considered expiring tuples from materialized views in a data warehouse. 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 [47]. Skyt et al. [41] consider vacuuming a temporal database. 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

[8] See footnote 3.

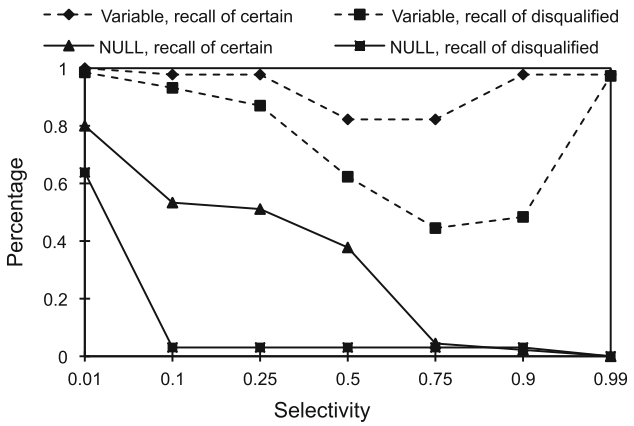

Fig. 5 Compare the accuracy of query results between suppression with variables and suppression with NULLs, measured by the recall of certain and disqualified tuples

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. Ataullah et al. [3] 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.

A number of authors have considered maintaining data integrity and preventing deletion in the context of auditing. Hasan and Winslett [17] considered the case when requested information is subject to a litigation hold, and they addressed the threat of an untrustworthy process vacuuming expired records. Their solution uses write-once read-many (WORM) storage and extra auditing actions for enforcement of a litigation hold, instead of relying on a DBMS. In [18], they proposed a transaction log architecture to ensure that database contents are long-term immutable. Both of these solutions are complementary to our framework when considering the institution itself as an adversary. In a different setting, Perez and Moreau consider the problem of securing provenance-based audits [32] by protecting the integrity of provenance information. Fabbri et al. [10] detect unauthorized access by reexecuting a log of past operations. Encrypting audit logs has also been widely studied in the literature [38,42,49] with the goal of maintaining the confidentiality and integrity of log records.

Our redaction policies (especially when implemented logically) are related to fine-grained access control rules. Wang et al. [48] 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 [4] 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 [8,11,43], implementation techniques [22,27,39], techniques for accommodating time in standard databases [36,44], as well as implemented extensions to existing systems [45]. Jensen [21] studied querying backlog relations to monitor changes to a database. Incomplete information also has a long history in databases [5,14,20], including in temporal databases.

The data model in this work combines technologies in uncertain database and temporal database communities. Temporal and transaction-time databases have been studied for over 20 years. However, incomplete temporal databases have attracted less attention. The data models of Gadia [12] and Koubarakis [23] establish the foundations of this area. A recent model U-relation [1] for uncertain databases can also be extended with temporal information. We compare with these works in detail in the "Appendix".[9]

When computing possible tuples in TC, efficiently solving the satisfiability problem of conjunctive inequalities is essential. Generally, the complexity depends on the domain of variables (dense or sparse), which is determined by the corresponding column in the schema, supported operators ($=, <, >, \leq, \geq, \neq$), form of conditions ($X$ op Y, X op C or more general linear inequalities), and type of formulas (conjunctive or disjunctive). In [34], the authors proved the general satisfiability problem for conjunctive inequalities is NP-hard. Restricted versions, such as eliminating $\neq$ and only considering the real domain, can be solved efficiently in linear time [16]. For disjunctive inequalities, Hochbaum [19] proved that even 2I-SAT is NP-complete, when considering linear inequalities. 2I-SAT only allows at most two inequalities per clause. The distinctness among variables in TC distinguishes our problem from all of the above.

The scheduling problem has a close relationship to our TC model. The "jobs" in the scheduling problem correspond to variables in TC. The distinctness among variables guarantees that each job will start at a different time on a single machine. $TC_<$ and $TC_\neq$ can be solved by efficient scheduling algorithms [24,40]. It is possible that other scheduling solutions are applicable to TC and its variants.

11 Conclusions and 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

[9] See supplementary material associated with the online version of this article on the journal's web site.

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 using a standard relational system, and that the uncertainty introduced by sample retention policies is acceptable. By proposing two different models, we allow users to tune the system between accuracy and performance: TI gives you better performance, but less accuracy, while TC offers improved accuracy for audit queries under sanitized histories, at the expense of increased query processing complexity.

We assume that retention policies are non-negotiable, despite the auditors' interest in analysis tasks. This assumption could be reconsidered in the future work by prioritizing auditing accuracy, at the potential cost of retention policy secrecy. In addition, a compelling extension to our sanitization model could use generalization or summarization of values instead of redaction. This would impose some cost to confidentiality, but may significantly improve auditing capabilities. Currently, our preservation rules consist of tuple-level specifications. In the future we would like to integrate more complex view-based preservation rules, such as those considered by [3,17], or rules targeting specific attribute values. We would like to investigate alternatives for supporting the periodic application of retention policies as a database evolves. And we would like to evaluate our system using data histories based on well-known benchmark databases such as TPC-H, or using real data sets and workloads, as well as explore other physical organizations that could lead to improved performance.

References

1. Antova, L., Jansen, T., Koch, C., Olteanu, D.: Fast and simple relational processing of uncertain data. In: ICDE, pp. 983–992 (2008)
2. ARMA Internaltional: Generally Accepted Recordkeeping Principles. http://www.arma.org/GARP/
3. Ataullah, A., Aboulnaga, A., Tompa, F.: Records retention in relational database systems. In: Proceeding of the ACM Conference on Information and Knowledge Management (CIKM), pp. 873–882 (2008)
4. Bertino, E., Bettini, C., Samarati, P.: A temporal authorization model. In: ACM Conference on Computer and Communications Security (CCS), pp. 126–135. ACM Press, New York (1994)
5. Biskup, J.: A foundation of codd's relational maybe-operations. ACM Trans. Database Syst. **8**, 608–636 (1983)
6. Blakeley, J., Coburn, N., Larson, P.: Updating derived relations: detecting irrelevant and autonomously computable updates. TODS **14**(3), 369–400 (1989)
7. Blakeley, J.A., Larson, P.A., Tompa, F.W.: Efficiently updating materialized views. SIGMOD Rec. **15**(2), 61–71 (1986)
8. Chomicki, J.: Temporal query languages: a survey. In: Temporal Logic (ICTL'94), vol. 827, pp. 506–534 (1994)
9. EMC Corporation: http://www.emc.com
10. Fabbri, D., LeFevre, K., Zhu, Q.: PolicyReplay: misconfiguration-response queries for data breach reporting. In: Proceedings of the VLDB Endowment, vol. 3, no. (1–2), pp. 36–47 (2010)
11. Gadia, S.K.: A homogeneous relational model and query languages for temporal databases. ACM Trans. Database Syst. **13**, 418–448 (1988)
12. Gadia, S.K., Nair, S.S., Poon, Y.C.: Incomplete information in relational temporal databases. In: 18th VLDB Conference (1992)
13. Garcia-Molina, H., Labio, W., Yang, J.: Expiring data in a warehouse. In: VLDB Conference, pp. 500–511 (1998)
14. Grahne, G.: The Problem of Incomplete Information in Relational Databases. Springer, Berlin (1991)
15. GRM LLC: http://www.grmdocumentmanagement.com
16. Guo, S., Sun, W., Weiss, M.: Solving satisfiability and implication problems in database systems. ACM Trans. Database Syst. **21**(2), 270–293 (1996)
17. Hasan, R., Winslett, M.: Trustworthy vacuuming and litigation holds in long-term high-integrity records retention. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 621–632. ACM (2010)
18. Hasan, R., Winslett, M., Mitra, S.: Efficient Audit-based Compliance for Relational Data Retention. UIUC Dept. of CS Tech Report UIUCDCS-R-2009-3044 (2009)
19. Hochbaum, D., Moreno-Centeno, E.: The inequality-satisfiability problem. Oper. Res. Lett. **36**(2), 229–233 (2008)
20. Imielinski, T., Lipski, W.: Incomplete information in relational databases. J. ACM **31**(4), 761–791 (1984)
21. Jensen, C.S., Mark, L.: Queries on change in an extended relational model. IEEE TKDE **4**, 192–200 (1992)
22. Jensen, C.S., Mark, L., Roussopoulos, N.: Incremental implementation model for relational databases with transaction time. IEEE Trans. Knowl. Data Eng. **3**, 461–473 (1991)
23. Koubarakis, M.: Database models for infinite and indefinite temporal information. Inf. Syst. **19**, 141 (1994)
24. Lageweg, B., Lenstra, J., Kan, A.: Minimizing maximum lateness on one machine: computational experience and some applications. Stat. Neerl. **30**(1), 25–41 (1976)
25. LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., DeWitt, D.: Limiting disclosure in hippocratic databases. In: VLDB Conference, pp. 108–119 (2004)
26. LexisNexis: Document Retention & Destruction Policies for Digital Data. http://www.lexisnexis.com/applieddiscovery/lawlibrary/whitePapers/ADI_WP_DocRetentionDestruction.pdf
27. Lomet, D.B., Barga, R.S., Mokbel, M.F., Shegalov, G., Wang, R. Zhu, Y.: Transaction time support inside a database engine. In: ICDE, p. 35 (2006)
28. Lu, W., Miklau, G.: AuditGuard: a system for database auditing under retention restrictions. IN: Proceedings of the VLDB Endowment vol. 1, no. 2, pp. 1484–1487 (2008)
29. Lu, W., Miklau, G.: Auditing a database under retention restrictions. In: IEEE International Conference on Data Engineering (ICDE), pp. 42–53 (2009)
30. Mullins, C.S.: Database Archiving for Long-term Data Retention. http://www.tdan.com/view-articles/4591 (2006)
31. OpenText Corporation: http://www.opentext.com
32. Perez, R.A., Moreau, L.: Securing provenance-based audits. In: International Provenance and Annotation Workshop 2010. Springer, Berlin (2010)
33. RainStor Inc.: http://rainstor.com
34. Rosenkrantz, D.J., Hunt, H.B.: Processing conjunctive predicates and queries. In: VLDB Conference, p. 72 (1980)
35. SAND Technology: http://www.sand.com
36. Sarda, N.L.: Extensions to sql for historical databases. IEEE Trans. Knowl. Data Eng. **2**, 220–230 (1990)
37. Sarma, A., Benjelloun, O., Halevy, A., Widom, J.: Working models for uncertain data. In: ICDE (2006)

38. Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Trans. Inf. Syst. Secur. **2**(2), 159–176 (1999)
39. Shaull, R., Shrira, L., Xu, H.: Skippy: a new snapshot indexing method for time travel in the storage manager. In: ACM SIGMOD Conference, pp. 637–648 (2008)
40. Simons, B., Sipser, M.: On scheduling unit-length jobs with multiple release time/deadline intervals. Oper. Res. 80–88 (1984)
41. Skyt, J., Jensen, C., Mark, L.: A foundation for vacuuming temporal databases. Data Knowl. Eng. **44**(1), 1–29 (2003)
42. Snodgrass, R., Yao, S., Collberg, C.: Tamper detection in audit logs. In: 13th VLDB Conference, pp. 504–515 (2004)
43. Snodgrass, R.T.: The TSQL2 Temporal Query Language. Kluwer Academic Publishers, Norwell (1995)
44. Snodgrass, R.T.: Developing time-oriented database applications in SQL. Morgan Kaufmann Publishers Inc., San Francisco (1999)
45. Snodgrass, R.T., Collberg, C.S.: The τ-BerkeleyDB Temporal Subsystem. Published: Available at http://www.cs.arizona.edu/tau/tbdb/
46. Stahlberg, P., Miklau, G., Levine, B.N.: Threats to privacy in the forensic analysis of database systems. In: SIGMOD Conference, pp. 91–102 (2007)
47. Toman, D.: Expiration of historical databases. In: Symposium on Temporal Representation and Reasoning (TIME), pp. 128–135 (2001)
48. Wang, Q., Yu, T., Li, N., Lobo, J., Bertino, E., Irwin, K., Byun, J.W.: On the correctness criteria of fine-grained access control in relational databases. In: VLDB Conference, pp. 555–566 (2007)
49. Waters, B., Balfanz, D., Durfee, G., Smetters, D.: Building an encrypted and searchable audit log. In: NDSS, vol. 6 (2004)
50. Wrozek, B.: Electronic Data Retention Policy (2001). http://www.sans.org/reading_room/whitepapers/backup/electronic-data-retention-policy_514
51. ZL Technologies, Inc.: http://www.zlti.com
52. ZyLAB: http://www.zylab.com