Auditing Compliance with a Hippocratic Database

Rakesh Agrawal Roberto Bayardo Christos Faloutsos Jerry Kiernan Ralf Rantzau Ramakrishnan Srikant

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We introduce an auditing framework for determining whether a database system is adhering to its data disclosure policies. Users formulate audit expressions to specify the (sensitive) data subject to disclosure review. An audit component accepts audit expressions and returns all queries (deemed "suspicious") that accessed the specified data during their execution.

The overhead of our approach on query processing is small, involving primarily the logging of each query string along with other minor annotations. Database triggers are used to capture updates in a backlog database. At the time of audit, a static analysis phase selects a subset of logged queries for further analysis. These queries are combined and transformed into an SQL audit query, which when run against the backlog database, identifies the suspicious queries efficiently and precisely.

We describe the algorithms and data structures used in a DB2-based implementation of this framework. Experimental results reinforce our design choices and show the practicality of the approach.

1 Introduction

The requirement for responsibly managing privacy sensitive data is being mandated internationally through legislations and guidelines such as the United States Fair Information Practices Act, the European Union Privacy Directive, the Canadian Standard Association's Model Code for the Protection of Personal Information, the Australian Privacy Amendment Act, the Japanese Personal Information Protection Law, and others. A vision for a Hippocratic database [2] proposes ten privacy principles for managing private data responsibly. A vital principle among these is compliance, which requires the database to verify that it adheres to its declared data disclosure policy.

Consider Alice who gets a blood test done at Healthco, a company whose privacy policy stipulates that it does not release patient data to external parties without the patient's consent. After some time, Alice starts receiving advertisements for an over-the-counter diabetes test. She suspects that Healthco might have released the information that she is at risk of developing diabetes. The United States Health Insurance Portability and Accountability Act (HIPAA) empowers Alice to demand from Healthco the name of every entity to whom Healthco has disclosed her information. As another example, consider Bob who consented that Healthco can provide his medical data to its affiliates for the purposes of research, provided his personally identifiable information was excluded. Later on, Bob could ask Healthco to show that they indeed did exclude his name, social security number, and address when they provided his medical record to the Cardio Institute. The demand for demonstrating compliance need not only arise from an externally initiated complaint – a company may institute periodic internal audits to proactively guard against potential exposures.

One approach to verifying that a database adheres to its disclosure policies might be to support data disclosure auditing by physically logging the results of each query. Problems with this approach include the following:

- it imposes a substantial overhead on normal query processing, particularly for queries that produce many results, and
- the actual disclosure auditing it supports is limited, since data disclosed by a query is not necessarily reflected by its output.

As an example of the limitations on disclosure auditing, consider P3P [5], which allows individuals to specify whether an enterprise can use their data in an aggregation. Verifying that database accesses have been compliant with such user preferences is not possible given only a log of aggregated results. To address this issue, one might instead consider logging the tuples "read" by a query during its execution instead of its output. However, determining which tuples accessed during query processing were actually disclosed is non-trivial. In addition, such a change dramatically increases logging overhead.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

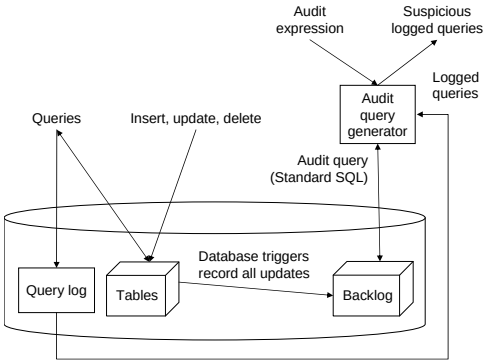


Figure 1: The system architecture

1.1 Our Contribution

We propose a system that can be used to audit whether the database system executed a query in the past that accessed the specified data. The ideal system should have the following properties:

- **Non-disruptive:** The system should put minimal burden on normal query processing.
- **Fast and precise:** The system should be able to quickly and precisely identify all the queries that accessed the specified data.
- **Fine-grained:** It should be possible to audit even a single field of a specific record.
- **Convenient:** The language for specifying data of interest should be intuitive and user friendly.

The proposed audit system satisfies the above desiderata. Figure 1 shows the overall architecture of our system. During normal operation, the text of every query processed by the database system is logged along with annotations such as the time when the query was executed, the user submitting the query, and the query's purpose. The system uses database triggers to capture and record all updates to base tables in backlog tables for recovering the state of the database at any past point in time. Read queries, which are usually predominant, do not write any tuple to the backlog database.

To perform an audit, the auditor formulates an audit expression that declaratively specifies the data of interest. Audit expressions are designed to be identical to the SQL queries, allowing audits to be performed at the level of an individual cell of a table. The audit expression is processed by the audit query generator, which first performs a static analysis of the expression to select a subset of logged queries that could potentially disclose the specified information. It then combines and transforms the selected queries into a single audit query by augmenting them with additional predicates derived from the audit expression. This audit query, expressed in standard SQL, when run against the backlog database yields the precise set of logged queries that accessed the designated data. Indices on the backlog tables make the execution of the audit query fast.

1.2 Assumptions

- There are subtle ways in which the combination of the results of a series of queries may reveal certain information. For example, the statistical database literature [1] discusses how individual information can be deduced by running several aggregate queries and the database security literature [3] shows how covert channels can be used to leak information. We limit ourselves to the problem of determining if the specified data was disclosed by a single query when that query is considered in isolation. We also assume that the queries do not use outside knowledge to deduce information without detection.
- The SQL queries we consider comprise a single **select** clause. A large class of queries (including those containing existential subqueries) can be converted into this form [12]. Specifically, we consider queries containing selection, projection (including **distinct**), relational join, and aggregation (including **having**) operations.

1.3 Paper Layout

The rest of the paper is organized as follows. Section 2 provides the syntax of an audit expression. We then propose the concept of an indispensable tuple, which in turn is used to identify suspicious queries with respect to an audit expression. Section 3 describes the system structures needed to support the proposed auditing capability. Specifically, we discuss the use of triggers to implement recovery of past database states. We also provide temporal extensions used to support the execution of an audit query, and give details of the query log. Section 4 states the algorithm for generating the audit query from an audit expression. Section 5 presents performance results, Section 6 discusses related work, and Section 7 concludes with a summary and directions for future work.

2 Definitions

We have a database D, which is a collection of base tables. We denote the scheme of table T as $T(C_0, C_1, \ldots, C_m)$ and use $t.C$ to refer to the value of the field C in tuple t. We will use the following schema in our examples:

Customer (cid, name, address, phone, zip, contact)
Treatment (pcid, date, rcid, did, disease, duration)
Doctor (did, name)

The primary keys have been underlined. A customer can be a patient, someone accepting financial responsibility for a patient's treatment, or an emergency *contact*. The Treatment table uses *pcid* to identify the patient receiving the treatment and uses *rcid* to identify the customer assuming financial responsibility for the treatment (who could be the same person as the patient). The *date* is the start date of the treatment and *duration* reflects the length of the treatment. Other column names are self-explanatory. To simplify exposition, we will assume that the database has referential integrity and that no field value is null.

2.1 Audit Expressions

We propose to use expressions that are very close to SQL queries to enable an auditor to conveniently specify the queries of interest, termed suspicious queries.

Specifically, the proposed syntax of an audit expression is identical to that of a select-project-join (SPJ) query without any **distinct** in the select list, except that **audit** replaces the key word **select** and the elements of the audit list are restricted to be column names:

audit *audit list*
from *table list*
where *condition list*

Let $\mathcal{U}$ be the cross product of all the base tables in the database. The audit expression marks a set of cells in the table $\mathcal{U}$. The marked cells belong to the columns in the audit list for the tuples that satisfy the predicate in the **where** clause. We are interested in finding those queries that access all the marked cells in any of the tuples in $\mathcal{U}$. These are the suspicious queries with respect to the audit expression.

Example 1 We want to audit if the disease information of anyone living in the ZIP code 95120 was disclosed. Here is the audit expression:

audit disease
from Customer c, Treatment t
where c.cid = t.pcid **and** c.zip = '95120'

This audit expression marks the cells corresponding to the disease column of those tuples in the Customer × Treatment table that have c.cid = t.pcid and c.zip = 95120. Any query that accesses the disease column of any of these tuples will be considered suspicious.

2.2 Informal Definitions

We introduce the notion of the indispensability of a tuple and then use it to define suspicious queries.

Informal Definition 1 (Indispensable Tuple - $ind(t, Q)$) *A tuple $t \in \mathcal{U}$ is indispensable in the computation of a query Q, if its omission makes a difference.*

Informal Definition 2 (Candidate Query - $cand(Q, A)$) *A query Q is a candidate query with respect to an audit expression A, if Q accesses all the columns that A specifies in its audit list.*

Informal Definition 3 (Suspicious Query - $susp(Q, A)$) *A candidate query Q is suspicious with respect to an audit expression A, if Q and A share an indispensable tuple.*

Example 2 Consider the audit expression A given in Example 1 and the following query Q:

select address
from Customer c, Treatment t
where c.cid = t.pcid **and** t.disease = 'diabetes'

We see that Q is a candidate query with respect to A as it accesses the disease column that A is auditing. Consider the Customer × Treatment table. Clearly, tuples that match the join condition and have diabetes in the disease column are indispensable for Q. Similarly, tuples that match the join condition and have 95120 as the zip code are indispensable for A. Therefore Q will be deemed suspicious with respect to A if there was some customer who lived in the ZIP code 95120 and was also treated for diabetes.

Example 3 Consider the query Q from Example 2 and the following audit expression A:

audit address
from Customer c, Treatment t
where c.cid = t.pcid **and** t.disease = 'cancer'

Q will not be deemed suspicious with respect to A because no tuple in Customer × Treatment can simultaneously satisfy the predicates of Q and A. But how about Alice who has both cancer and diabetes? Although Q discloses Alice's address, the fact that Alice has cancer is not relevant to the query: Q only asks for people who have diabetes. In other words, anyone looking at the output of the query will not learn that Alice has cancer. Hence it is reasonable to not consider the query to be suspicious. Note that all the tuples of Customer × Treatment marked by A have cancer in the disease column and Q does not access any one of them.

2.3 Formal Definitions

Let the query Q and audit expression A be of the form:

$$Q = \ldots(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})) \quad (1)$$

$$A = \overline{\pi}_{C_{OA}}(\sigma_{P_A}(\mathcal{T} \times \mathcal{S})) \quad (2)$$

where $\mathcal{T}$, $\mathcal{R}$, $\mathcal{S}$ are *virtual* tables of the database D, that is, cross products of base tables:

$$\mathcal{T} = T_1 \times T_2 \times \ldots \times T_n$$
$$\mathcal{R} = R_1 \times R_2 \times \ldots \times R_m$$
$$\mathcal{S} = S_1 \times S_2 \times \ldots \times S_k.$$

The operator $\overline{\pi}$ is the multi-set projection operator that preserves duplicates in the output (as opposed to the relational projection operator π which eliminates duplicates). Note that $\mathcal{T}$ is common to Q and A.

We denote by C_Q the column names that appear anywhere in a query Q, and by C_{OQ} the column names appearing in the select list of Q. Similarly, C_{OA} denotes the column names present in the audit list of an audit expression A. P_Q denotes the predicate of the query and P_A is the predicate of the audit expression. We refer to the tuples of any virtual table as *virtual* tuples.

We now formalize the definition of indispensability, for all classes of queries of interest. Specifically, we discuss (a) SPJ queries, (b) queries with aggregation without **having**, and (c) queries with aggregation and **having**.

2.3.1 Indispensability - SPJ queries

Consider first a SPJ query that does not contain a **distinct** in its select list. This case is the most important case on which the rest of the cases will be based. For such queries, the form of the query of Eq. (1) is specialized to:

$$Q = \overline{\pi}_{C_{OQ}}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})). \quad (3)$$

We can now formalize the definition of an indispensable tuple for an SPJ query:

Definition 1 (Indispensability - SPJ) *A (virtual) tuple $v \in \mathcal{T}$ is indispensable for an SPJ query Q if the result of Q changes when we delete v:*

$$ind(v, Q) \Leftrightarrow \overline{\pi}_{C_Q}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})) \neq \overline{\pi}_{C_Q}(\sigma_{P_Q}((\mathcal{T} - \{v\}) \times \mathcal{R})).$$

Theorem 1 *A (virtual) tuple $v \in \mathcal{T}$ of the SPJ query Q is indispensable if and only if*

$$\sigma_{P_Q}(\{v\} \times \mathcal{R}) \neq \emptyset.$$

Proof From Definition 1, we have

$$ind(v, Q) \Leftrightarrow \overline{\pi}_{C_Q}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})) \neq \overline{\pi}_{C_Q}(\sigma_{P_Q}((\mathcal{T} - \{v\}) \times \mathcal{R})).$$

Since the projections $\overline{\pi}$ maintain the duplicates, we have

$$\begin{aligned} ind(v, Q) &\Leftrightarrow \sigma_{P_Q}(\mathcal{T} \times \mathcal{R}) \neq \sigma_{P_Q}((\mathcal{T} - \{v\}) \times \mathcal{R}) \\ &\Leftrightarrow \sigma_{P_Q}(\mathcal{T} \times \mathcal{R}) \neq \sigma_{P_Q}(\mathcal{T} \times \mathcal{R}) - \sigma_{P_Q}(\{v\} \times \mathcal{R}) \\ &\Leftrightarrow \sigma_{P_Q}(\{v\} \times \mathcal{R}) \neq \emptyset. \end{aligned}$$

■

Queries with **distinct** in the **select** clause produce a duplicate-free table. Such queries have the form $Q = \pi_{C_{OQ}}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$. Let Q' be the SPJ query obtained from the original query Q after removing **distinct** from the query text. Then, we have the following definition:

Definition 2 (Indispensability - Distinct) *A (virtual) tuple v is indispensable for $Q = \pi_{C_{OQ}}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$ if and only if it is indispensable for $Q' = \overline{\pi}_{C_{OQ}}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$.*

The motivation for this definition will become apparent after the upcoming discussion of aggregation queries. Queries with **distinct** can be viewed as a special case of aggregation, the aggregation function being the first tuple in a group.

We can state succinctly:

Observation 1 *Duplicate elimination does not change the set of indispensable tuples for an SPJ query.*

2.3.2 Indispensability - Aggregation without having

The definition of indispensability of a tuple for an aggregation query requires extra care. Consider a query that computes average salary per department. If Alice happens to have exactly the average salary of her department and her tuple is omitted, the query result will not be affected. However, it will be wrong to treat Alice's tuple as dispensable because the privacy systems such as P3P allow individuals to opt out of the use of their values in the computation of an aggregation.

The form of the query of Eq. (1) for an aggregation query without a **having** clause is specialized to:

$$Q = {}_{gby}\gamma_{agg}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})). \quad (4)$$

where *gby* are the grouping columns and *agg* represent aggregations like **avg**(duration), **count**(disease).

Consider the query Q' that is a version of Q, but without aggregations. That is, Q' has exactly the same **from** and **where** clauses, and a **select** clause with the same columns as Q, but without the aggregation functions. Note that the columns used in *agg* (e.g. duration, disease) are included in the select list of Q'.

Definition 3 (Indispensability - Aggregation) *A (virtual) tuple v is indispensable for Q if and only if it is indispensable for the aggregate-free version Q'.*

Example 4 Consider the following query that outputs average duration of diabetes treatment by doctor:

```
select   name, avg(duration)
from     Doctor d, Treatment t
where    d.did = t.did and t.disease = 'diabetes'
group by name
```

Indispensability of a tuple t in the the above query is determined by considering the indispensability of t in the following SPJ query:

```
select   name, duration
from     Doctor d, Treatment t
where    d.did = t.did and t.disease = 'diabetes'
```

We find that every Treatment tuple having diabetes in the disease field is indispensable. Thus the fact that the duration values of these tuples were used in computing the output is not lost.

The following is immediate:

Observation 2 *A tuple v is indispensable for $Q = {}_{gby}\gamma_{agg}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$ if and only if it is indispensable for $Q' = \overline{\pi}_{C_Q}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$.*

2.3.3 Indispensability - Aggregation with having

We will use the query in following example to help with the explanations.

Example 5 Our query is a modified version of the query given in Example 4. It outputs average duration of diabetes treatment, but only for those doctors for whom this average is greater than 100 days:

```
select   name, avg(duration)
from     Doctor d, Treatment t
where    d.did = t.did and t.disease = 'diabetes'
group by name
having   avg(duration) > 100
```

The general form of an aggregation query Q that includes a **having** clause can be written as:

$$Q = \sigma_{P_H}({}_{gby}\gamma_{agg}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))). \quad (5)$$

Compared to Eq. (4), we now have an extra **having** predicate P_H (**avg**(duration) > 100 in Example 5). Any group that does not satisfy this predicate is not included in the result of Q, which implies that any tuple belonging to a group that gets filtered out by P_H is dispensable.

Let Q' be the **having**-free version of Q, obtained by simply removing the **having** clause from Q.

Definition 4 (Indispensability - Aggregation with having) *A (virtual) tuple v is indispensable for Q if and only if it is indispensable for Q' and it belongs to a group that satisfies the* having *predicate P_H.*

We will again recast indispensability in terms of an SPJ query. Define a group table G as:

$$G = {}_{gby}\gamma_{agg}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})). \quad (6)$$

For our example query, G will have two columns: name and **avg**(duration). It will have as many tuples as there are doctors who treat diabetes. Every tuple will have the average duration of diabetes treatment for the corresponding doctor.

Next form the following table:

$$QG = \sigma_{P_G}((\sigma_{P_Q}(\mathcal{T} \times \mathcal{R})) \times G) \quad (7)$$

where P_G is the natural join condition on the group-by columns, *gby*. We have augmented the result tuples of the **having**-free version of Q with the corresponding group values. The query Q can now be computed from $\sigma_{P_H}(QG)$.

It follows then

Observation 3 *A (virtual) tuple $v \in \mathcal{T}$ is indispensable for query Q with aggregation and* having *if and only if v is indispensable for the SPJ query*

$$\overline{\pi}_{C_Q}(\sigma_{P_H}(\sigma_{P_G}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R} \times G)))). \quad (8)$$

2.3.4 Suspicious Queries

We first define a maximal virtual tuple for queries $Q1$ and $Q2$.

Definition 5 (Maximal Virtual Tuple) *A tuple v is a maximal virtual tuple for queries $Q1$ and $Q2$, if it belongs to the cross product of common tables in their* from *clauses.*

We can now formalize the definitions of *candidate* and *suspicious* queries.

Definition 6 (Candidate Query) *A query Q is a candidate query with respect to the audit expression A if and only if*

$$C_Q \supseteq C_{OA}.$$

Definition 7 (Suspicious Query) *A candidate query Q is suspicious with respect to audit expression A if they share an indispensable* maximal virtual tuple *v, that is:*

$$susp(Q,A) \Leftrightarrow \exists v \in \mathcal{T}\ s.t.\ ind(v,Q) \wedge ind(v,A)$$

where $\mathcal{T} = T_1 \times T_2 \times \ldots \times T_n$ is the cross product of the common tables in Q and A.

3 System Structures

We now discuss the system structures needed to handle audits in the presence of updates to the database.

3.1 Full Audit Expression

The audit expression is prepended with an additional **during** clause that specifies the time period of interest:

during *start-time* **to** *end-time*
audit *audit-list*
...

Only if a query has accessed the data of concern during the specified time period is the query deemed suspicious.

Privacy policies specify who is allowed to receive what information and for what purpose [2, 5]. An audit expression can use the **otherthan** clause to specify the purpose-recipient pairs to whom the data disclosure does not constitute non-compliance:

otherthan *purpose-recipient pairs*
during *start-time* **to** *end-time*
audit *audit-list*
...

3.2 Query Log

As shown in Figure 1, the audit system maintains a log of past queries executed over the database. The query log is used during the static analysis to limit the set of logged queries that are transformed into an audit query.

Our prototype implementation has a thin middleware that lies between the application and the database engine. This middleware has been implemented as an extension to the JDBC driver. The middleware intercepts queries and writes the query string and associated annotations to the log. We assume the isolation level of serializable [8] and log only queries of committed transactions.

The annotations include the timestamp of when the query finished, the ID of the user issuing the query, and the purpose and the recipient information extracted from the context of the application [10, 11] in which the query was embedded. The query log is maintained as a table.

Note that some database systems (e.g., DB2) provide the facility for logging incoming queries. In such cases, this capability can be extended to log additional information required for auditing.

3.3 Temporal Extensions

We determine if a candidate query Q accessed the data specified in an audit expression by selectively playing back history. We thus need to recreate the state of the database as it existed at the time Q was executed. A backlog database [9] is eminently suited for this purpose.

We describe two organizations for the backlog database: time stamped and interval stamped. In both the organizations, a backlog table T^b is created for every table T in the database. T^b records all updates to T. We will assume that every table T has a primary key column P; the system can create an internally generated key column otherwise.

3.3.1 Time stamped Organization

This organization is based on the ideas presented in [9]. Aside from all columns in T, a tuple in T^b has two additional columns: *TS* that stores the time when a tuple is inserted into T^b, and *OP* that takes one of the values from {'insert', 'delete', 'update'}. For every table, three triggers are created to capture updates. An insert trigger responds to inserts in table T by inserting a tuple with identical values into T^b and setting its *OP* column to 'insert'. An update trigger responds to updates to T by inserting a tuple into T^b having the after values of the tuple in T and setting its *OP* column to 'update'. A delete trigger responds to deletes in T by inserting into T^b the value of the tuple before the delete operation and setting its *OP* column to 'delete'. In all the three cases, the value of the *TS* column for the new tuple is set to the time of the operation.

To recover the state of T at time τ, we need to generate the "snapshot" of T at time τ. This is achieved by defining a view T^τ over the backlog table T^b:

$$\begin{aligned} T^\tau \quad = \quad & \pi_{P,C_1,\ldots,C_m}(\{t \mid \\ & t \in T^b \wedge t.TS \leq \tau \wedge t.OP \neq \text{'delete'} \wedge \\ & \nexists r \in T^b \text{ s.t. } t.P = r.P \wedge r.TS \leq \tau \wedge r.TS > t.TS\}). \end{aligned}$$

The scheme for T^τ is identical to T. T^τ contains at most one tuple from T^b for every distinct primary key value P. Among a group of tuples in T^b having an identical primary key value, the selected tuple t is the one that was created at or before time τ, is not a deleted tuple, and there is no other tuple r having the same primary key value that was created at or before time τ but whose creation time is later than that of t.

3.3.2 Interval stamped Organization

In this organization, the end time (*TE*) of a tuple is explicitly stored in addition to the start time (*TS*). Thus, the combination of *TS* and *TE* for a tuple gives the time period during which the tuple was alive. A null value of *TE* is treated as current time. The operation field (*OP*) is no longer necessary.

When a new tuple t is inserted into T, the insert trigger also adds t to T^b, setting its *TE* column to null. When a tuple $t \in T$ is updated, the update trigger searches for the tuple $b \in T^b$ such that $b.P = t.P \wedge b.TE =$ null and sets $b.TE$ to the current time. Additionally, the trigger inserts a copy of t into T^b with updated values and its *TE* column set to null. When a tuple t is deleted from T, the delete trigger searches for $b \in T^b$ such that $b.P = t.P \wedge b.TE =$ null and sets $b.TE$ to the current time.

3.3.3 Indexing

We propose two strategies for indexing a backlog table T^b:

1. *Eager*: Index is kept fresh and updated every time T^b is updated.
2. *Lazy*: Index is created afresh at the time of audit. Otherwise, T^b is kept unindexed.

The advantage of the eager strategy is that there is no latency at the time of audit due to the time needed to build the index. However, an update during normal query processing is burdened with the additional overhead of updating the index. The trade-off is reversed in the lazy strategy.

We can also choose which columns are indexed. We can index the primary key. We can also create a composite index consisting of the primary key concatenated with the timestamp. We explore the performance trade-offs in managing backlog tables in Section 5.

4 Algorithms

The audit query is generated in two steps:

1. *Static Analysis*: Select *candidate* queries (i.e., potentially suspicious queries) from the query log. (Our use of the term "candidate query" in this section refers to a query that passes the static analysis. All such queries are also candidate queries according to the formal definition.)
2. *Audit Query Generation*: Augment every *candidate* query with information from the audit expression and combine them into an audit query that identifies the suspicious queries.

4.1 Static Analysis

For a given audit expression A, some queries will be judged as non-candidates, and excluded immediately. We use four static tests, explained next. The query log is indexed to make these tests fast.

The first is by comparing the attribute names: with audit columns C_{OA}, we simply check whether $C_Q \supseteq C_{OA}$. The second test checks whether the timestamp of query Q is out of range with respect to the audit interval in the **during** clause of A. The third test checks whether the purpose-recipient pair of Q matches any of the purpose-recipient pairs specified in the **otherthan** clause of A. Finally, we can eliminate some queries by checking for contradictions between the predicates P_Q and P_A, such as $P_Q = (age > 40)$ and $P_A = (age < 20)$. This class of tests is an instance of the constraint satisfaction problem, for which many solution techniques are available [6].

4.2 Audit Query Generation

At the end of static analysis, we have a set of candidate queries $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ that are potentially suspicious with respect to the audit expression A. We augment every Q_i with information in A, producing another query AQ_i defined against the view of the backlog database at time τ_i, where τ_i is the timestamp of Q_i as recorded in the query log. If we were to execute these AQ_i queries, those with non-empty results will comprise the exact set of suspicious queries. However, to increase opportunities for optimization, all AQ_i are combined into one audit query AQ whose output is a set of query identifiers corresponding to those AQ_i that yield non-empty results. This audit query is the one that is executed against the backlog database.

// Q is a simple selection query over a single table T, executed at time τ.
// A is an audit expression over the same table T.

1) **create** an empty QGM for the audit query AQ
2) **add** Q to AQ
3) **add** A to AQ
4) **rewrite** A to range over the result of Q instead of T
5) **replace** A's audit list with id(Q)
6) **replace** T with the view T^{τ}

Figure 2: Audit query generation for simple selections

To simplify exposition, we will assume henceforth that $\mathcal{Q}$ has only one query Q and discuss how it is transformed into an audit query AQ. Our implementation makes use of the Query Graph Model (QGM) to manipulate Q and A to generate AQ.[1] To avoid QGM diagrams from becoming unwieldy, we will abbreviate column names. For our example schema reproduced below, the abbreviated column names used in the figures are indicated in bold letters:

Customer (**c**id, **n**ame, **a**ddress, p**h**one, **z**ip, c**o**ntact)
Treatment (**p**cid, **r**cid, d**i**d, di**s**ease, d**u**ration, da**t**e)
Doctor (**d**id, na**m**e)

4.3 Simple Selections

Consider first the simple case of a candidate query Q involving a selection over a single base table T and the audit expression A over the same table. This case is a special case of the upcoming SPJ queries. However, we present it for pedagogical reasons.

Lemma 1 *Let T be a base table of our database D. Let $A = \overline{\pi}_{C_{OA}}(\sigma_{P_A}(T))$ be an audit expression and let $Q = \overline{\pi}_{C_Q}(\sigma_{P_Q}(T))$ be a candidate query. Q is suspicious with respect to A if and only if $\sigma_{P_A}(\sigma_{P_Q}(T)) \neq \emptyset$.*

Proof From the upcoming Theorem 2, by substituting T for $\mathcal{T}$, and ignoring the non-existing $\mathcal{R}$ and $\mathcal{S}$. ■

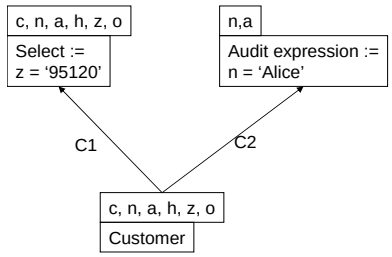
Thus, given that query Q has passed the static analysis, we need to check whether the combined selection $\sigma_{P_A}(\sigma_{P_Q}(T))$ is empty or not, which is what Figure 2 implements using the QGM representation. We illustrate the audit query generation algorithm using the following example.

Example 6 Candidate query Q: Retrieve all customers in ZIP code 95120.

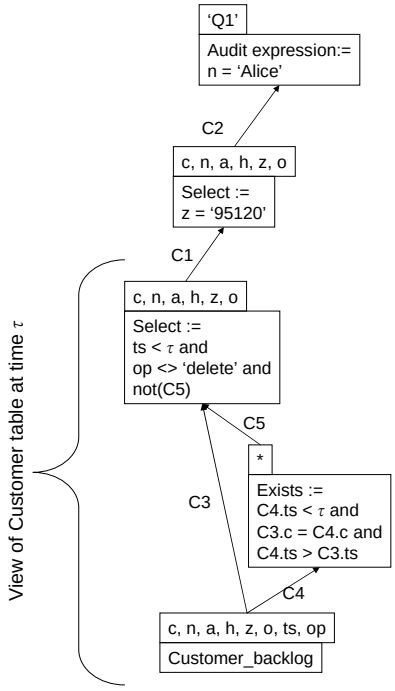
select *
from Customer
where zip = '95120'

Audit expression A: Find queries that have accessed Alice's name and address.

[1] QGM [12] is a graphical representation that captures the semantics of queries and provides convenient data structures for transforming a query into equivalent forms. QGM is composed of entities portrayed as boxes and relationships among entities portrayed as lines between boxes. Entities can be operators such as table, select, group, union, etc. Lines between operators represent quantifiers that feed an operator by ranging over the output of the other operator.



(a) After Line 3 in Figure 2



(b) After Line 6 in Figure 2

Figure 3: QGM for Example 6 (simple selection)

audit name, address
from Customer
where name = 'Alice'

Figure 3(a) shows the state of the QGM graph after Line 3 (Figure 2). A new QGM structure for the audit query AQ has been created and both the candidate query Q and the audit expression A have been added to AQ. Figure 3(b) shows the state of QGM after Line 6. Line 4 has changed the audit expression's quantifier (range variable) C_2 from ranging over the Customer table to ranging over the result of the query Q. As part of this transformation, each column referenced by C_2 is changed to reference a column of Q's output. If a column referenced by C_2 is not in the output of Q, it is propagated up from the Customer table to be included in the Q's select list. Line 5 replaces the audit list with Q's id: Q1. Finally, Line 6 replaces the Customer table with a view of the Customer table at time τ when Q

1) **create** an empty QGM for the audit query *AQ*
2) **add** Q to *AQ*
3) **add** A to *AQ*

4) **rewrite** A to additionally range over the result of Q with quantifier x
5) **for each** quantifier r in A which is over a table T also in Q
6) **substitute** x in place of r in A
7) **replace** A's audit list with $\mathrm{id}(Q)$

8) **replace** every table T_i referenced in *AQ* with T_i^{τ}.

Figure 4: Audit query generation when both the candidate query and the audit expression contain joins

completed.

4.4 SPJ Queries

Consider now the case when the candidate query as well as the audit expression contain joins in the WHERE clauses. The audit list may contain columns from multiple tables and the join condition in the candidate query may be different from the one in the audit expression.

Theorem 2 *A candidate SPJ query* $Q = \overline{\pi}_{C_Q}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$ *is suspicious with respect to an audit expression* $A = \overline{\pi}_{C_{OA}}(\sigma_{P_A}(\mathcal{T} \times \mathcal{S}))$ *if and only if*

$$\sigma_{P_A}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R} \times \mathcal{S})) \neq \emptyset.$$

Proof According to our Definition 7, we have

$$\begin{aligned} susp(Q,A) &\Leftrightarrow \exists m \in \mathcal{T}\ s.t.\ ind(m,Q) \wedge ind(m,A) \\ &\Leftrightarrow \exists m \in \mathcal{T}, r \in \mathcal{R}, s \in \mathcal{S}\ s.t. \\ &\quad P_Q(\{m\,r\}) \wedge P_A(\{m\,s\}) \\ &\Leftrightarrow \exists m \in \mathcal{T}, r \in \mathcal{R}, s \in \mathcal{S}\ s.t. \\ &\quad \{m\,r\,s\} \in \sigma_{P_A}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R} \times \mathcal{S})) \\ &\Leftrightarrow \sigma_{P_A}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R} \times \mathcal{S})) \neq \emptyset. \end{aligned}$$

■

Figure 4 gives the algorithm. Note that an audit expression A may have multiple quantifiers, only some subset of which may range over a table that also appears in query Q. These are the only ones for which A is made to range over the result of the query (Lines 5-6). For others, A continues to range over the original tables. We illustrate the algorithm using the following example.

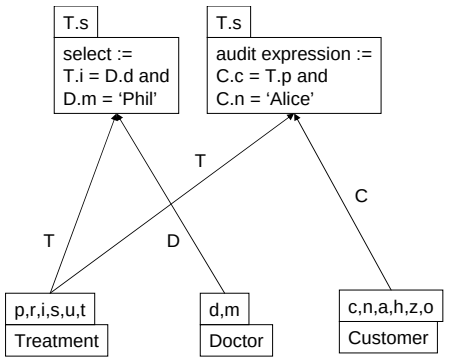
Example 7 Candidate query Q: Find all diseases treated by doctor Phil.

select T.disease
from Treatment T, Doctor D
where T.did = D.did **and** D.name = 'Phil'

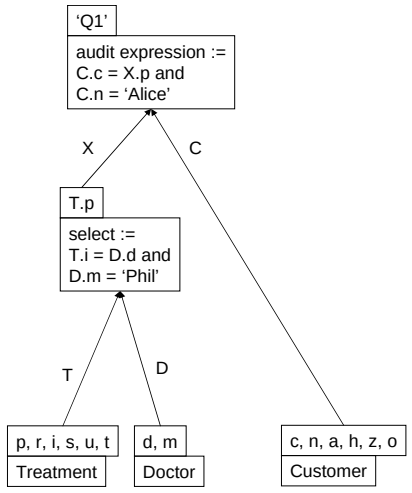
Audit expression A: Find queries that have disclosed the diseases of Alice.

audit T.disease
from Customer C, Treatment T
where C.cid = T.pcid **and** C.name = 'Alice'

Figure 5(a) shows the initial QGM (after Line 3) and Figure 5(b) shows the final QGM (after Line 7). In the final QGM, the audit expression ranges over the result of the query and then joins the results with the Customer table since Customer only appears in the audit expression.



(a) After Line 3 in Figure 4



(b) After Line 7 in Figure 4

Figure 5: QGM for Example 7 (join)

4.5 Aggregation

To determine if an aggregate query without a **having** clause is suspicious, aggregate functions are simply removed along with the **group by** clause. Columns previously referenced by aggregate functions are added to the select list of the query. The resulting SPJ query is then handled using the algorithm given in Figure 4.

If the aggregate query, however, additionally contains a **having** clause, the predicate therein might have eliminated the data specified by the audit expression from the query result. Simply removing the **having** clause can thus lead to false positives. This limitation is overcome by the algorithm given in Figure 6, which is based on the upcoming theorem.

Recall that the general form of such a query is given by Eq. (5): $Q = \sigma_{P_H}(\ {}_{gby}\gamma_{agg}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$. By Eq. (6), the group table $G = {}_{gby}\gamma_{agg}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R}))$. As always, the audit expression is $A = \overline{\pi}_{C_{OA}}(\sigma_{P_A}(\mathcal{T} \times \mathcal{S}))$.

Theorem 3 *A candidate query* Q *with aggregation and* **having** *is suspicious with respect to an audit expression* A

// The QGM of an aggregate query Q that
includes **having** is a triplet (Q_s, Q_g, Q_h):

// Q_s is the SPJ part of Q,
// Q_g contains aggregations ranging over Q_s, and
// Q_h is a selection over Q_g representing **having**.

1) **create** an empty QGM for the audit query AQ
2) **add** Q to AQ
3) **add** A to AQ
4) **rewrite** A to additionally range over the result of Q_s with quantifier x

5) **for each** quantifier r in A which is over a table T also in Q
6) **substitute** x in place of r in A
7) **replace** the audit list of A with the grouping columns of Q_g
8) **create** a new empty select box B and add it to AQ
9) **add** Q_h, A as inputs to B
10) **join** inputs of B on grouping columns from Q_h and A
11) **replace** $\Pi(B)$ with id(Q)

12) **replace** every table T_i referenced in AQ
with its backlog counterpart T_i^{τ} at time τ.

Figure 6: Audit query generation for an aggregate query containing **having**

if and only if

$$\sigma_{P_A}(\sigma_{P_H}(\sigma_{P_G}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R} \times G \times \mathcal{S})))) \neq \emptyset. \qquad (9)$$

Proof From Observation 3, the query Q has the same indispensable tuples as the SPJ query Q' below:

$$Q' = \overline{\pi}_{C_Q}(\sigma_{P_H}(\sigma_{P_G}(\sigma_{P_Q}(\mathcal{T} \times \mathcal{R} \times G)))).$$

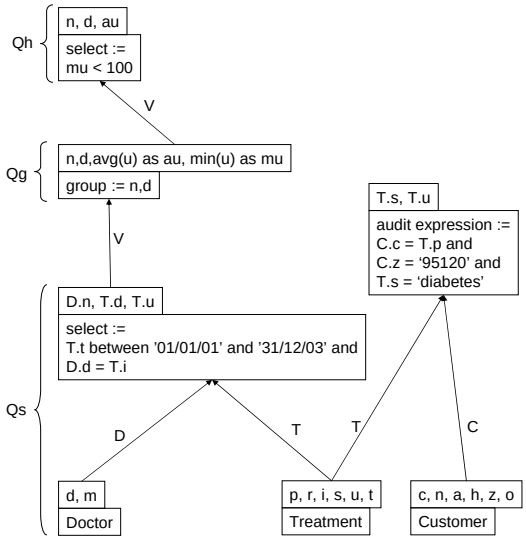
Then, from Theorem 2 we have that Q' is suspicious if and only if Eq. (9) holds. ■

An aggregate query with a **having** clause can be viewed as consisting of three parts: Q_s, Q_g, and Q_h. The first part, Q_s, ignores grouping and aggregation and finds the tuples qualifying the WHERE clause. Grouping and aggregations are then applied to this result in Q_g. Finally, any predicates on groups are applied using a selection operator over the result of grouping and aggregation in Q_h. A new select box is created on Line 8 in Figure 6. This operator joins the tuples emanating from Q_h with those from A to ensure that these A tuples were not all filtered out by the **having** predicates in Q_h. We illustrate the algorithm with Example 8.

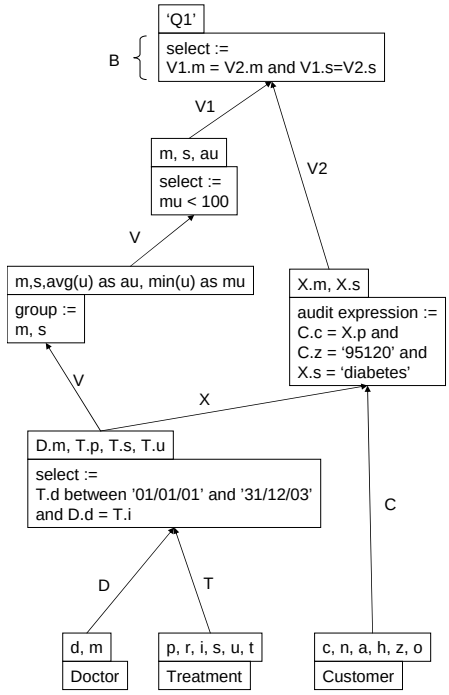
Example 8 Candidate query Q: Compute the average treatment duration grouped by disease and the doctor performing the treatment for treatments which were between 01/01/2001 and 31/12/2003 having a minimum duration $<$ 100.

select	D.name, T.disease, avg(T.duration)
from	Doctor D, Treatment T
where	T.date **between** '01/01/2001' **and** '31/12/2003'
	and D.did = T.did
group by	D.name, T.disease
having	min (T.duration) < 100

Audit expression A: Find queries that have accessed the disease and treatment duration of patients who have diabetes and live in ZIP code 95120.



(a) After Line 3 in Figure 6



(b) After Line 11 in Figure 6

Figure 7: QGM for Example 8 (aggregation)

audit	T.disease, T.duration
from	Customer C, Treatment T
where	C.cid = T.pcid **and** C.zip = '95120'
	and T.disease = 'diabetes'

The QGM for the candidate query Q in Example 8 integrated with the audit expression is shown in Figure 7(a). Figure 7(b) shows the audit query. The select box B ensures that the groups formed by the grouping operator that

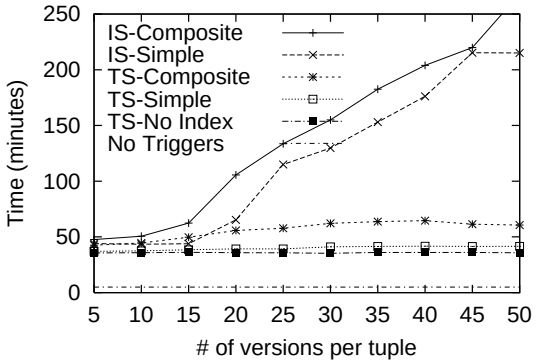


Figure 8: Cost of maintaining backlog tables.

survived the **having** predicate match the audit expression's data.

5 Performance

This section presents the results of performance experiments from a DB2 implementation of our auditing solution. Specifically, we study the overhead imposed on normal query processing, and the cost of conducting audits.

Experiments were performed on an IBM Intellistation M Pro 6868 having an 800 MHz Pentium III processor, 512 MB of memory, and a 16.9 GB disk drive, and running Windows 2000 Version 5.00.2195 service pack 4. The same machine was used to host data as well as backlog tables and to run audits. The DBMS used was DB2 UDB Version 7 with default settings. We would have liked to use real-life query logs and data tables, but no such dataset was available. We therefore performed our experiments on the Supplier table of the TPC-H database [15], using synthetic workload. The results below give the average warm performance numbers.

We report results for time stamped as well as interval stamped organizations of the backlog tables. We consider three cases: no index, simple index on the supplier key *SKEY*, and composite index on *SKEY* and start time *TS*. We explore both eager and lazy strategies for updating indices.

We will write Supplier^b to refer to the Supplier backlog table.

5.1 Burden on Normal Query Processing

We performed the following experiment to study the overhead of maintaining backlog tables. The Supplier table contained 100,000 tuples and Supplier^b started with a copy of every Supplier tuple. An SQL update statement updated every Supplier tuple, resulting in the creation of a new version of every supplier in Supplier^b. Forty nine such update statements were executed, each adding 100,000 tuples to Supplier^b that finally ended up having fifty versions of every supplier and a total of 5 million tuples. Indices of Supplier^b were updated eagerly.

The update operation on the Supplier table took 5.2 minutes to complete when performed unburdened with the maintenance of Supplier^b. This time essentially consists of sequentially reading the Supplier tuples and writing them back after updating one of the values. Figure 8 shows the total time taken by the successive update operations when the additional time spent by the database system on firing the update triggers and the resultant operations on Supplier^b was also included. In the performance graphs, TS (IS) denotes the time stamped (interval stamped) organization.

With the time stamped organization, the update trigger simply adds a new tuple to Supplier^b corresponding to every updated data tuple. Therefore, when there is no index on Supplier^b, the overhead experienced by successive updates remains fairly constant. When there is an index on *SKEY*, the overhead is a bit larger due to additional index updates, and this overhead increases a little for the later updates because the size of the index grows. The composite index on *SKEY* and *TS* obviously has a somewhat larger overhead than the simple index.

For equivalent operation with the interval stamped organization, the update trigger first locates the most recent version of the tuple, updates its end time, and then adds a new current tuple. Unfortunately, the cost of locating the most recent version becomes prohibitively large when there is no index; hence it is not shown in the figure. Even when there is an index on *SKEY*, all the versions of a tuple need to be brought into memory to select the most recent of them. If different versions of a supplier do not remain clustered on the same page (which we found to be the case even when we had a clustered index on *SKEY*), the number of page faults increases with the number of versions, resulting in a rapid degradation of performance. Having the additional index on *TS* does not help in cutting down the number of versions that are examined before the most recent one is found. On the other hand, the overhead increases somewhat due to additional index updates and a larger index.

It is substantially faster (per tuple) to sequentially update all the tuples of a table in DB2 than to update an individual tuple. Thus, updating all the tuples of the Supplier table with backlog maintenance using the time stamped organization is about 10 times slower than without maintenance. We next performed another experiment in which only one Supplier tuple was updated. Supplier^b had 25 versions of every tuple in this experiment (the average number of versions in the first experiment). The cost of an update to a single tuple with backlog maintenance was now on average 3 times the cost of the same update without maintenance when using the simple index, and 3.7 times when using the composite index.

We note that in most of the installations, there are far more read queries than update operations. Updates are often batched and performed while the system is under a light load. The read queries in our audit system do not incur any overhead beyond logging the query string (which is anyway done in many installations).

5.2 Eager vs. Lazy Indexing

It is clear from the discussion in the previous section that the lazy indexing is not a viable option for the interval stamped

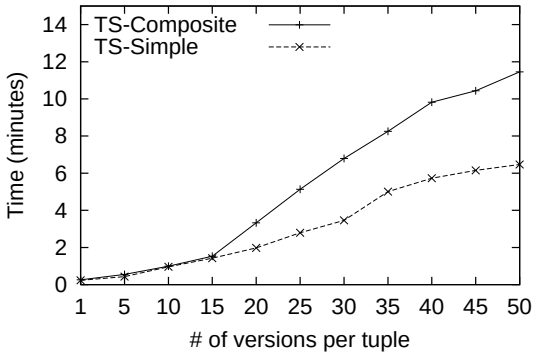


Figure 9: Cost of building indices over the backlog table.

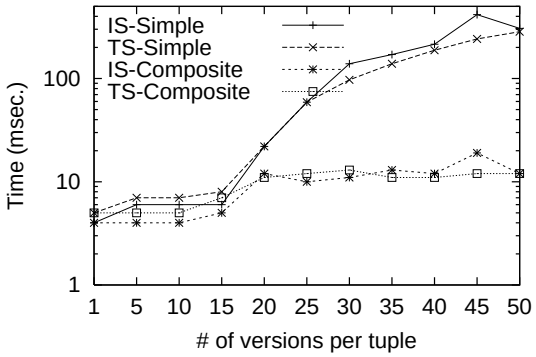


Figure 10: Execution time of an audit query

organization. However, the overhead of eagerly updating the indices can be avoided in the time stamped organization by building them as needed at the time of audit. Figure 9 shows the time needed to build indices from scratch with an initial set of 100,000 suppliers in Supplier^b, while increasing the number of versions of each supplier. The results are shown both for a simple index on *SKEY* and for a composite index on *SKEY* and *TS*.

We see that the index construction times are such that it would be acceptable to adopt the lazy strategy and strictly create indices at audit time.

5.3 Performance of Audit

We study the audit performance by measuring the execution time of simple audit queries. The audit expression is of the form:

during t_1 **to** t_2
audit name **from** Supplier **where** skey = k

We set both t_1 and t_2 to the time when the initial versions of the Supplier tuples were created. The value of k is randomly set to one of the values of *SKEY* present in the Supplier table. We assume that static analysis has yielded one candidate query: **select * from** Supplier.

We consider both time stamped and interval stamped organizations, with simple as well as composite indices. Figures 10 shows the results.

Both the time stamped and interval stamped organizations benefit a great deal from the composite index on *SKEY* and *TS* as the number of versions becomes large. When there is an index only on *SKEY*, the query plan first selects all versions of a tuple with the matching supplier key and then selects the correct version amongst the matching tuples. These versions might reside on different disk pages and cause page faults as Supplier^b becomes large. The composite index avoids this problem.

We also see that with few versions of a tuple, the interval stamped organization has a slight performance advantage, but loses this advantage as the number of versions increases. The interval stamped organization requires an extra timestamp attribute to record the end time of validity of a tuple. The larger tuple size results in more page faults as the number of versions increases, and thus the impact of the larger size outweighs the benefit of the simpler interval stamped computation over the time stamped organization that requires a join.

5.4 Takeaways

The composite index on the primary key and start time pays large dividend over an index only on the primary key at the time of audit, although it puts a slightly larger burden on updates if the indices are updated eagerly.

The interval stamped organization has a slight advantage over the time stamped organization at the time of audit if the number of versions is small. However, the lazy strategy for updating indices cannot be used with the interval stamped organization and eager updating becomes quite expensive as the number of versions increases. Overall, the use of time stamped organization along with the lazy strategy for updating indices is recommended. However, the eager strategy is also not too burdensome for the time stamped organization.

The system supports efficient auditing without substantially burdening normal query processing tasks.

6 Related Work

Closely related to compliance is the privacy principle of *limited disclosure*, which means that the database should not communicate private information outside the database for reasons other than those for which there is consent from the data subject [2, 10]. Clearly, the two are complimentary. The principle of limited disclosure comes into play at the time a query is executed against the database, whereas demonstrating compliance is post facto and is concerned with showing that usage of the database indeed observed limited disclosure in every query execution.

Oracle [11] offers a "fine-grained auditing" function where the administrator can specify that read queries are to be logged if they access specified tables. This function logs various user context data along with the query issued, the time it was issued, and other system parameters including the "system change number". Oracle also supports "flashback queries" whereby the state of the database can be reverted to the state implied by a given system change number. A logged query can then be rerun as if the database

was in that state to determine what data was revealed when the query was originally run. There does not appear to be any auditing facility whereby an audit predicate can be processed to discover which queries disclosed data specified by the audit expression. Instead, Oracle seems to offer the temporal database (flashback queries) and query logging (fine-grained auditing) components largely independent of each other.

The problem of matching a query against an audit expression bears resemblance to the problem of predicate locking [7] that tests if the predicates associated with two lock requests are mutually satisfiable. Besides being expensive, this test can lead to false positives when applied to the auditing problem. Related work also includes the literature on query processing over views that contains the notion of augmenting a user query with predicates derived from the view definition [13]. Also related is the work on optimizing a group of queries (e.g. [4, 14]) that can be profitably used by our system to accelerate the execution of audit queries.

7 Summary

We identified the problem of verifying whether a database system is complying with its data disclosure policies through auditing. Given the accelerated pace at which legislations are being introduced to govern data management practices, this problem represents a significant opportunity for database research. We formalized the problem through the fundamental concepts of indispensability and suspiciousness. Additional contributions include a carefully designed and implemented system that meets the design goals enunciated in the introduction:

- **Convenient:** The audit expression language used by our system reuses the familiar SQL syntax, providing a familiar, declarative and expressive means for specifying the data whose disclosure is subject to review.
- **Fine-grained:** The audit expression language allows the auditor to specify even a single field of a record as subject for review.
- **Fast and precise audits:** Our system combines the audit expression with logged queries into an SQL audit query that examines only the specific data necessary to determine suspiciousness. Guided by our implementation and experimentation with various backlogging and indexing strategies, we proposed system structures to support efficient audit query execution.
- **Non-disruptive:** Our system imposes only a small burden on the execution of most queries. Rather than logging query results or the tuples accessed by a query, it logs the query strings. While update operations require some additional backlog database maintenance, the predominant read queries are processed without any further encumbrance.

We have considered a data disclosure model in which the querier does not possess any outside knowledge and the information gained is limited to what could be learnt from the current query. It would be interesting to see how our framework could be extended to support more adversarial disclosure scenarios. Other remaining work includes how schema evolution can be gracefully accommodated in the audit system. Finally, we feel it would be beneficial to the community to develop a set of comprehensive benchmarks for measuring and testing the effectiveness and performance of any database auditing proposal.

Acknowledgements Christos Faloutsos was on leave from Carnegie Mellon; he was partially supported by the National Science Foundation under Grants No. IIS-0083148, IIS-0209107, IIS-0205224, SENSOR-0329549, and IIS-0326322.

References

[1] N. Adam and J. Wortman. Security-control methods for statistical databases. *ACM Computing Surveys*, 21(4):515–556, Dec. 1989.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th Int'l Conference on Very Large Databases*, Hong Kong, China, August 2002.

[3] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison Wesley, 1995.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Conference on Management of Data*, Dallas, Texas, 2000.

[5] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. The platform for privacy preferences 1.0 (P3P1.0) specifi cation. W3C Recommendation, April 2002.

[6] R. Dechter. *Constraint Processing*. Morgan Kaufman, 2003.

[7] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1992.

[9] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental implementation model for relational databases with transaction time. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473, December 1991.

[10] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in Hippocratic databases. In *30th Int'l Conf. on Very Large Data Bases*, Toronto, Canada, August 2004.

[11] A. Nanda and D. K. Burleson. *Oracle Privacy Security Auditing*. Rampant, 2003.

[12] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *ACM SIGMOD Conference on Management of Data*, San Diego, California, 1992.

[13] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[14] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[15] TPC-H decision support benchmark. http://www.tpc.org.