SELECT Triggers For Data Auditing

Daniel Fabbri #1, Ravi Ramamurthy *2, Raghav Kaushik *3

#Electrical Engineering & Computer Science, University of Michigan
2260 Hayward Street, Ann Arbor MI 48109 USA
[1]dfabbri@umich.edu
*Microsoft Research
One Microsoft Way, Redmond WA 98052 USA
[2]ravirama@microsoft.com
[3]skaushi@microsoft.com

***Abstract*—Auditing is a key part of the security infrastructure in a database system. While commercial database systems provide mechanisms such as triggers that can be used to track and log any changes made to "sensitive" data using UPDATE queries, they are not useful for tracking accesses to sensitive data using complex SQL queries, which is important for many applications given recent laws such as HIPAA. In this paper, we propose the notion of SELECT triggers that extends triggers to work for SELECT queries in order to facilitate data auditing. We discuss the challenges in integrating SELECT triggers in a database system including specification, semantics as well as efficient implementation techniques. We have prototyped our framework in a commercial database system and present an experimental evaluation of our framework using the TPC-H benchmark.**

I. Introduction

A key component of a database security infrastructure is an auditing system. An important class of auditing is *data auditing* that requires the careful monitoring of accesses to "sensitive" data. The objective of the data auditing system is to correlate the operations executed on a database with specific data items in the database. One of the basic mechanisms provided by most commercial database systems for data auditing is the notion of triggers, which enables row-level auditing of DDL/DML statements. Using triggers, an administrator can handle important data auditing tasks such as: 1) finding updates that change a salary value by more than 50%, and 2) maintaining a history of changes to a sensitive column (e.g., salary).

However, there are many scenarios that require row-level auditing support for SELECT queries (that go beyond the functionality offered by triggers today). The key functionality required is to check if a SQL query "accessed" some sensitive data. In fact, such functionality is required by laws such as HIPAA [15]. A common source for data breaches are insider attacks, where the insider gets information about sensitive data by running SQL queries and examining their results.

Data auditing for SQL queries has been studied in previous research [1], [9], [14] — the goal is to answer the following question: given a query (or update) statement and a set of individuals, find the subset of individuals whose data was "accessed" by the query (in general, sensitive data can be any information stored in the database rather than individuals). One of the challenges addressed by prior work on data auditing is to define what it means for a query to access the sensitive data corresponding to an individual. Briefly, the approach generally adopted is to use ideas related to the body of work on data provenance [4] to identify when an individual's data contributes substantively to the query result.

Prior work on data auditing for SQL queries assumes an *offline* architecture where the audit log records all SQL queries that were executed and the analysis of whether a particular query accessed some sensitive data is carried out at a later point in time. Offline analysis (e.g., [5], [9]) however can be expensive because it potentially requires efficient access to previous states of the database, either requiring an "as-of" querying capability [8] or a rollback of the database state using the database log. As a result, such functionality can be cumbersome to use in certain scenarios such as those pertaining to HIPAA as shown in the following example.

Example 1.1: The United States Health Insurance Portability and Accountability Act (HIPAA) enables every patient to demand from their health care provider the name of every entity to whom her information has been revealed. For example [1], if a patient Alice receives advertisements for diabetes tests, she can check whether her health care provider has released the information that she is at risk of developing diabetes. In order to comply with HIPAA, the health care provider is required to maintain a tamper-proof audit log of all SQL queries issued to the system; the log can be used to identify all entities that accessed Alice's health record. In order to provide such functionality when Alice makes such a request, a security admin has to analyze the audit log to check for queries that "accessed" Alice's record. This check may involve a rollback of the database to the state that existed when the query was originally executed, as well as several query executions in order to compute this result (in general, for complex queries, computing if a query accessed sensitive information requires re-executing the query — see Section II for details).

Given that triggers are a well-known interface for application development and are already used for auditing scenarios pertaining to INSERT/DELETE queries, it is natural to consider extending triggers to SELECT queries for data auditing. For the scenario discussed in Example 1.1, a security admin can install a SELECT trigger that keeps track of which queries access Alice's record as the queries execute. By piggybacking on the database instance in which the query executes, such functionality avoids the need to rollback the database. However, for Example 1.1, since we do not know

in advance which patient will request her record, we need the ability to audit for all patients - thus SELECT triggers must provide the ability to scale to a large number of tuples.

SELECT triggers also open up the possibility of real-time feedback on access to sensitive information which is intrinsically interesting and can enable addressing scenarios such as: 1) finding users that have accessed more than a given number of patient records with a particular disease, and 2) finding all patient records accessed by each doctor last week ordered by the number of patients accessed.

A. Challenges

The focus of this paper is to design an efficient framework that integrates SELECT triggers in a database system. Extending the notion of triggers to work for SELECT queries raises a variety of interesting challenges.

Specification/Semantics: Triggers are declaratively specified in a *query independent* manner to perform an action when specific data items are accessed. Unlike INSERT/DELETE queries where it is immediately apparent when to execute a trigger (e.g., upon inserting a row), it can be more subtle to determine when a SELECT trigger should be executed as the following example demonstrates.

Example 1.2: Consider the following two queries with which a user can infer if a particular patient Alice has cancer.

```
SELECT * FROM Patients P, Disease D
WHERE P.PatientID = D.PatientID
  AND Name = 'Alice'
  AND Disease = 'cancer'

SELECT 1 FROM Patients
WHERE exists
(SELECT * FROM Patients P, Disease D
  WHERE P.PatientID = D.PatientID
    AND Name = 'Alice'
    AND Disease = 'cancer')
```

The queries illustrate it is possible for a row to influence the output of a query even if the row is only a part of some subexpression in a SQL query. Triggering events based on the output of a query would not work for the second query in the example.

One can consider semantics for triggering events based on READ locks acquired by a transaction, however this approach can result in a large number of false positives, where a row is incorrectly marked as having been accessed by the query (for example, consider a join query where most tuples are filtered by the join predicate, if we base our semantics on READ locks we would deem all tuples from both tables being joined as accessed). Thus, more robust definitions are needed to define what it means for a query to access data.

Prior work (e.g., [1], [9], [14]) has defined the semantics for data auditing using ideas related to data provenance [4] to identify when an individual's data contributes substantively to the query result. However, definitions for data auditing based on provenance (see Section II for an overview) are "heavy-weight" in that they either require the propagation of non-trivial state in the form of annotations or require non-trivial additional processing including materialization of intermediate results. Thus, directly integrating such techniques as part of a triggering system is challenging and adds significant overhead to query execution.

Efficient Implementation: Since the existing techniques for data auditing are "heavy-weight", we need to relax the semantics such that it enables a more light-weight framework for SELECT triggers. We propose a design where the system provides one-sided guarantees - there are no false negatives, where a row is incorrectly marked as having not been accessed by the query (which is important for the case of data auditing). For the class of select-join (SJ) queries, we guarantee the same result as the offline system, but yield false positives for more complex queries (see Section III). To ensure correctness, the offline system must verify all queries that are thought to access sensitive data. Even though the offline system is still required in the auditing infrastructure, SELECT triggers can serve as an important filter to reduce the number of queries that the offline system must process and therefore improves the overall auditing performance.

Interestingly, we can enable the above semantics with a light-weight mechanism termed *audit operators* that are similar to "data viewers" that do not modify the logic of a query plan but "sniff" the records flowing between the different relational operators. Just as standard database triggers check conditions and perform actions during data updates, audit operators check intermediate records generated by a query execution plan for access to sensitive information, and perform actions for the data that are accessed. Similar to triggers, the audit operator does carry a cost and needs to be used with care. But as a mechanism, it affords a reasonable balance between query execution efficiency and the ability to monitor access to sensitive data online.

While the mechanism of "data viewers" has been proposed in prior work for query debugging such as SQL Server Integration Services [11] and Inspector Gadget [16], the above query debugging frameworks operate directly on dataflow graphs that do not permit a query-independent specification, which is necessary for SQL. As with the case of existing triggers, we adopt a *declarative* approach where the programmer merely specifies what the sensitive data are through an *audit expression*. We let our system determine the placement of the audit operator, which leads to several challenges: 1) where should the operator be placed? and 2) are all the edges where operators can be placed meaningful from the point of view of provenance which has been proposed in the past for offline auditing? To address these challenges, we formulate an operator placement problem that inserts audit operators into query plans.

B. Contributions

To summarize, we study the novel problem of extending the notion of triggers to work for SELECT queries, which has important applications for data auditing. We discuss the specification and semantics of SELECT triggers (Section II) and describe a low overhead mechanism termed an audit operator for enabling this functionality. In order to retain a declarative interface, the system needs to place an audit operator appropriately in a query plan - we study the audit

operator placement problem in Section III. We have prototyped our framework by modifying a commercial database system (Microsoft SQL Server). We discuss the key extensions required to both the query optimizer and the query execution engine that are required to support the audit operator in Section IV. Our experimental results (Section V) on queries from the TPC-H benchmark [18] indicate that SELECT triggers attain a low false positive rate and can scale to a large number of individuals/tuples for a small additional overhead - for instance, we can audit a large number of individuals (around a million customers in the TPC-H benchmark) at an additional overhead of 2%.

II. SELECT Triggers Specification

Traditionally triggers are configured to automatically execute code in response to INSERT, UPDATE and DELETE commands on a database. For example, triggers are commonly used to ensure data integrity, track changes and replicate data when INSERT commands are executed. This framework works well because it is immediately apparent when rows are inserted, updated or deleted. Unfortunately, SELECT commands (i.e., queries) are currently not part of the SQL trigger standard.

In this paper, we extend triggers to SELECT commands. Just like INSERT, UPDATE and DELETE triggers, it is necessary to understand when a trigger should execute and what action they should perform. For this work, we consider the following query-independent specification:

```
on ACCESS to <SENSITIVE DATA> do <ACTION>
```

During query execution, the system records accesses to the sensitive data and stores this information in the query's ACCESSED internal state. The ACCESSED internal state is a per-query, in-memory relation that maintains access information and can be used by the trigger's action. This internal state is similar to SQL's use of the OLD and NEW variables for UPDATE triggers to reference previously existing or newly created data. After the query completes, the action is executed. The action takes the form of an SQL/T-SQL fragment, can reference the query's ACCESSED internal state and is executed as its own system transaction. The action executes even if the query is aborted to account for queries that read a subset of the result. Like traditional triggers, SELECT triggers are cascading. As a result, a SELECT trigger's action can trigger an UPDATE trigger, which in turn can trigger other SELECT triggers.

We note that the described semantics for SELECT triggers are specific for the data-auditing problem. However, it is easy to imagine other uses of SELECT triggers that may require alternative semantics. For example, an administrator may wish to configure SELECT triggers to execute before the query result is returned to warn users that they are accessing sensitive data. We leave these variations to future work.

The challenge is then to define how to specify sensitive data and what it means to access data. The rest of the section describes *audit expressions* as a means to specify the sensitive data and the provenance semantics used to determine when data are accessed.

A. Audit Expression

Sensitive information is specified in the form of an *audit expression.* Just like SQL, audit expressions provide a declarative format to specify data and the database system determines if that data are accessed. In this paper, we restrict audit expressions to queries with simple predicates that do not involve subqueries, and joins are limited to key-foreign key relationships (we currently impose these restrictions in order to maintain the privacy guarantees of the auditing system as discussed in [9]). Audit expressions are structured as follows.

```
CREATE AUDIT EXPRESSION <NAME> AS
 SELECT <SENSITIVE COLUMNS>
 FROM <TABLES T, ..., Tn>
 WHERE <PREDICATE>
 FOR SENSITIVE TABLE <T>,
     PARTITION BY <KEY>
```

An audit expression's *sensitive table* specifies the table to monitor for accesses, and the associated *partition-by* key specifies what information should be stored in the ACCESSED internal state (such as the tuple's primary key). We refer to values from the partition-by key as IDs. For ease of exposition, we restrict audit expressions to a single sensitive table (the sensitive columns must also be from this sensitive table). However, it is possible to audit the broader class of audit expressions described above.

Example 2.1: Consider a health care database with tables *Patients(PatientID, Name, Age, Zip)* and *Disease(PatientID, Disease)*. Suppose that we wish to specify that Alice's records are sensitive. This can be done using the following audit expression:

```
CREATE AUDIT EXPRESSION Audit_Alice AS
 SELECT *
 FROM Patients
 WHERE Name = 'Alice'
 FOR SENSITIVE TABLE Patients,
     PARTITION BY PatientID
```

Example 2.2: Similarly, suppose that we wish to specify that the personal information pertaining to all patients suffering from cancer is sensitive. We can do so by specifying the following expression.

```
CREATE AUDIT EXPRESSION Audit_Cancer AS
 SELECT P.*
 FROM Patients P, Disease D
 WHERE P.PatientID = D.PatientID
     AND Disease = 'cancer'
 FOR SENSITIVE TABLE Patients,
     PARTITION BY PatientID
```

B. Data Access

The basis for data access semantics is to define what it means for a query to *access* a particular record. The rich body of work on data privacy [3] has conclusively shown that privacy can be compromised in subtle ways using innocuous looking queries such as aggregates. Therefore, simplistic definitions such as examining the query output for sensitive

data are inadequate from a privacy point of view. Slightly more complicated approaches that compare query selection conditions to audit expression selection conditions are not effective either because they frequently classify queries as having accessed sensitive data even though the data did not influence the query's computation (see Example 6.1 for further discussion).

Instead, prior work [1], [5], [9], [13], [14] has relied on the notion of data provenance to define data access. The idea is to define a record as accessed if it *influences* the query output.

Definition 2.3: [1] Given a database instance D and a query Q, a tuple t in the sensitive table T is said to *influence* Q if deleting t from T changes the result of Q. □

Example 2.4: Consider the queries from Example 1.2. Suppose that there is a patient named Alice in the database who is suffering from cancer. Then Alice's record is accessed by the query because deleting the tuple would change the query result, even though Alice's record is not always contained in the query result. □

We note that the notion of a tuple influencing a query is based on a definition of data provenance, namely the notion of a counter-factual record [10]. There, the goal is to find the set of tuples τ such that after removing τ from the database, the database is in a state where inserting/removing tuple t removes tuple r from the query result. However, the notions of a counter-factual record and determining if a tuple is accessed are not identical since we are not interested in the provenance of any one output record; rather our goal is to find all input records that influenced the output overall.

Before we can define what it means to access data, we must consider which columns are accessed by the query. We say that a query Q accesses a set of columns if it cannot be equivalently rewritten to exclude the columns. Combining this statement with Definition 2.3, we obtain the following definition for sensitive data access.

Definition 2.5: Given a database instance D, a query Q and an audit expression E, a tuple t in the output of E is said to be *accessed* by Q if: (1) Q accesses the sensitive columns in the definition of E and (2) tuple t influences Q.

Checking if Q accesses a set of columns is straightforward. Therefore, for ease of exposition, in the rest of the paper we assume that all columns in the relation underlying the audit expression are sensitive while noting that all techniques extend in a straightforward manner to allow a subset of columns to be sensitive. In the case of UPDATE and DELETE commands (which read information before modifying it), we revert to the traditional trigger semantics to determine when data are accessed, which is consistent with Definition 2.5.

Previous work [9], [13], [14] has formally analyzed the privacy guarantees yielded by the above definition. One limitation of a system that applies the definition is that it is susceptible to negative disclosures because an attacker can learn about data not in the database without executing a trigger [9]. Another limitation is that accesses may be missed when duplicates are eliminated in the query (i.e., set semantics). For example, consider the database with two patients named Alice that have cancer; if the queries in Example 1.2 were modified to find the distinct names of patients with cancer, then removing one of their records would not change the query result. We acknowledge these limitations, but note that they are inherent when supporting the generality of SQL.

It is important to note that a simple mechanism to check if a tuple influences a query is to execute the query twice, once on the database instance with the tuple and once without it, and then compare the results. While this is a general mechanism that is applicable for all SQL queries, it is not feasible for triggers because it adds significant overhead.

C. Trigger Actions and Applications

There are multiple practical applications of SELECT triggers for data auditing. The simplest example is the action of writing an audit log entry for each sensitive piece of data that is accessed. Recall that the ACCESSED internal state stores information about the tuples that were accessed by the query.

```
CREATE TRIGGER Log_Alice_Accesses
 ON ACCESS TO Audit_Alice AS
 INSERT INTO Log
 SELECT now(), userID(), sql(), PatientID
 FROM ACCESSED
```

Each log entry records the time when the query was executed, the user who executed the query, the query's SQL text and the PatientIDs that were accessed, which is Alice's ID for the given audit expression (now(), userID() and sql() are database methods). The ON ACCESS TO clause specifies the audit expression (i.e., the sensitive data) and the associated attributes that are available from the ACCESSED internal state for the trigger's action (i.e., the partition-by key).

The trigger's action executes as a system transaction and retains the locks acquired by the query for the partition-by key to ensure that the recorded access information is consistent with the database state when the query was executed. However, other database state can change in the interim between the access and action executing. Note that this is similar to the case for traditional triggers (e.g., with AFTER INSERT semantics).

In some cases, writing every PatientID may be excessive. Instead, an administrator may want to know more general information about what data are accessed. For example, suppose the administrator wants to monitor the set of departments associated with the cancer patients whose data are accessed. This action can be expressed as follows using the existing table *Departments(PatientID, DeptID)*.

```
CREATE TRIGGER Log_Cancer_Dept_Accesses
 ON ACCESS TO Audit_Cancer AS
 INSERT INTO Log
 SELECT DISTINCT now(), userID(), sql(),
   D.DeptID
 FROM ACCESSED A, Departments D
 WHERE A.PatientID = D.PatientID
```

SELECT triggers can be combined with other triggers to produce more sophisticated systems. For example, SELECT triggers that write to the log can be combined with an INSERT

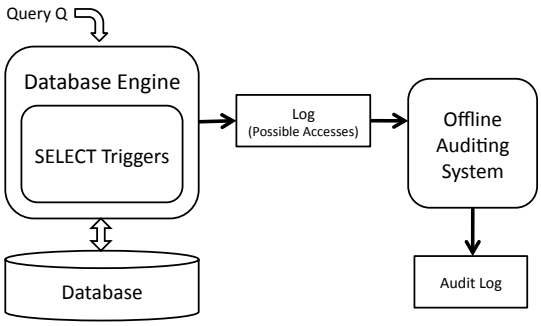


Fig. 1. Overview of the auditing system. The administrator initially creates a SELECT trigger that specifies the sensitive data and the action to perform when the data are accessed. As a query executes, accesses to the sensitive data are recorded. After the query completes, the action executes. In the auditing case, access information is written to the log for further analysis by the offline auditing system. The remaining queries and associated accesses are not audited further.

trigger to notify the administrator if a user accesses more than ten sensitive patients in a single day as follows.

```
CREATE TRIGGER Notify
 ON Log AFTER INSERT AS
 IF (SELECT count(DISTINCT PatientID) > 10
  FROM Log
  WHERE Date = NEW.Date
   AND UserID = NEW.UserID)
  SEND EMAIL
```

III. Mechanism for SELECT triggers

As described in Section II-B, the simplest way to determine if sensitive data are accessed is to execute the query twice, once with the sensitive tuple and once without it, and then compare the results. Such a procedure is obviously not a feasible implementation for SELECT triggers because of its overhead. More sophisticated alternatives involve some form of provenance computation. Prior work has approached the problem either (1) by modifying query execution to add annotations to each intermediate record, or (2) by defining inverses for each operator so that at the end of execution, the inversion of the operator can be invoked over the query output to compute the provenance and determine if a sensitive tuple influenced the result. However, both of the above mechanisms impose a significant overhead on query execution time. For instance, the state-of-the-art technique for computing provenance for complex queries can incur an overhead of up to 5x [6]. Accordingly, relying on prior work in data provenance, the logical place to determine if data are accessed is in an *offline* auditing system.

However, as mentioned in Section I, there are cases where offline auditing is undesirable. For example, an administrator may want to know immediately when sensitive data are accessed rather than waiting days, if not weeks for the offline system to run. This section outlines an alternative mechanism to check if sensitive data are accessed in an *online* manner that piggybacks on query execution.

A. Desiderata

Designing SELECT triggers to use the offline auditing mechanisms is not feasible due to its overhead. Instead, SELECT triggers must implement a "light-weight" notion of data auditing. This light-weight approach is characterized by its efficiency and generality to audit any input query. To attain this efficiency, we allow for the possibility of false positives (i.e., where a sensitive tuple is incorrectly marked as having been accessed). Because of the possibility of false positives, the offline auditing system is required to analyze potential accesses to ensure correctness. Furthermore, this possibility requires that SELECT triggers do not produce false negatives (i.e., where a sensitive tuple is incorrectly marked as having not been accessed and the SELECT trigger does not execute), otherwise accesses to sensitive data could be missed.

Figure 1 illustrates the overall auditing system, and the relationship between SELECT triggers and the offline auditing system. SELECT triggers serve as a filter for the offline system so that there are fewer queries and associated accesses to audit. This design can significantly reduce the offline auditing effort. While offline auditing performance gains are dependent on the query workload and the trigger's specification, it is easy to imagine scenarios where SELECT triggers significantly reduce the fraction of the tuples that need to be audited offline.

We summarize the desired properties of SELECT triggers as follows.

- **Efficient:** SELECT triggers should implement a "light-weight" notion of data auditing that is able to efficiently determine if sensitive data are accessed. We allow for the possibility of false positives to attain this efficiency.
- **General:** SELECT triggers should function correctly for any SQL query no matter its complexity.
- **No False Negatives:** SELECT triggers should execute for every sensitive tuple that is accessed.

Interestingly, these desired properties can be be met by a fairly light-weight mechanism called an *audit operator*, which is similar to a data viewer.

B. Audit Operator

An audit operator is a logical operator similar to a data viewer that efficiently analyzes data passing through it during query execution to determine if sensitive data are accessed. Specifically, an audit operator takes as input an audit expression E and determines which tuples in the output of E are accessed by the query. The audit operator acts similarly to a relational filter operator in that it evaluates an IN predicate with the audit expression. The major difference from a filter operator is that instead of filtering tuples that do not satisfy the predicate, audit operators act as a no-op and instead write the partition-by information of tuples that satisfy the predicate to the ACCESSED internal state, which can then be used by the SELECT trigger's action to write to the log (i.e., Section II-A). We discuss one possible implementation in Section IV-A.

Audit operators can be placed between any nodes in a query plan. The challenge is to place audit operators such that they satisfy the desired properties described in Section III-A. We refer to a query execution plan that includes an audit operator as an *instrumented* query plan. Audit operator placement can be subtle because it can lead to false positives and false negatives, which we demonstrate with the following two examples.

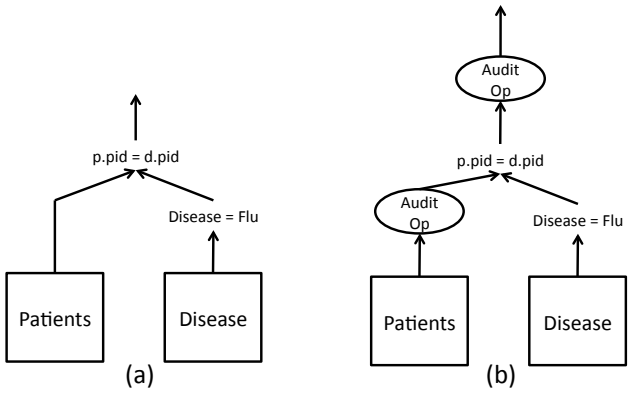
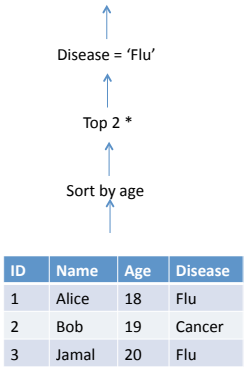


Fig. 2. (a) Original query plan and (b) instrumented query plan with audit operators.



ID	Name	Age	Disease
1	Alice	18	Flu
2	Bob	19	Cancer
3	Jamal	20	Flu

Fig. 3. Some audit operator placements produce false negatives.

Example 3.1: Consider the audit expression that tests if Alice's medical record is accessed (i.e., Example 2.1). Consider the following query that is represented by the query plan in Figure 2(a).

```
SELECT P.PatientID, Name,  Age, Zip
FROM Patients P, Disease D
WHERE P.PatientID = D.PatientID
AND D.Disease = 'flu'
```

Audit operators can be added to the query plan to test for sensitive data at either of the edges shown in part (b). If a tuple passes through an audit operator with data satisfying the audit expression, then the partition-by key is recorded in the ACCESSED internal state. For instance, consider the audit operator that is placed at the output of the scan of the Patients table in Figure 2(b). Assume there are two patients that satisfy the predicate $(Name = Alice)$ but only one of them has the flu. The audit operator would add the PatientIDs of both patients to the ACCESSED internal state, thus resulting in a false positive. Note that an audit operator placed at the output of the join would not generate this false positive. □

It is important to note that different audit operator placements can result in different false positive rates. However, the number of false positives is independent of the physical operators used in the query plan. From Example 3.1, a nested-loop join or an index-intersection join would result in the same number of false positives for the query.

A simple heuristic to construct an instrumented query plan with minimal false positives is to place an audit operator at the highest point in the query plan where the sensitive data are accessible - certain database systems use a similar algorithm for placing user defined functions (UDFs) in a query plan. If we make the simplifying assumption that operators typically only filter rows (i.e., no cross-products, non-foreign key joins, etc.), then the **highest-node** heuristic ensures that the number of false positives will be minimized since its input will have the smallest cardinality among all candidate edges where the audit operator can be placed. However, as the following example demonstrates this heuristic can result in an instrumented plan that produces false negatives.

Example 3.2: Consider a health care database and the query plan shown in Figure 3 that finds which among the two youngest patients has flu. Consider the top most edge in the plan where PatientIDs are visible (which happens to be the top of the query plan). Since Bob is among the two youngest patients and does *not* suffer from flu, the record corresponding to Bob does not flow past the top-most edge. Suppose that the audit expression covers all patients. If we place the audit operator at the top-most edge, the record corresponding to Bob does not appear as part of the audit log. This leads to a false negative — the record corresponding to Bob is accessed by the above query, since deleting it changes the query result (specifically, the output of the top-2 operator). □

C. Audit Operator Placement

Next, we define the audit operator placement problem and its associated properties for a single audit expression E. We refer to the set of partition-by IDs generated by the audit expression as $sensitiveIDs$. We refer to the set of partition-by IDs generated by audit operators as $auditIDs$ (in the case when multiple audit operators are added to a query plan, the ACCESSED internal state contains the union of all $auditIDs$). We denote the set of partition-by IDs corresponding to E that are accessed by a query as $accessedIDs$ (as determined by the offline auditing system). We define the following properties of an instrumented query execution plan.

Definition 3.3 (False Positive): An instrumented query plan for a query Q has a false positive if there exists an ID such that $ID \in auditIDs$ and $ID \notin accessedIDs$ (i.e., the audit operators generate an ID that the query does not access).

Definition 3.4 (No False Negatives): An instrumented query plan for a query Q has no false negatives if $accessedIDs \subseteq auditIDs$ (i.e., every accessed ID is audited).

Audit Operator Placement Problem: Given a query execution plan and an audit expression E, the goal is to obtain an instrumented execution plan P such that: (1) P produces no false negatives. (2) Among all instrumented plans that produce no false negatives, P has the fewest number of false positives.

A natural heuristic to the problem is to insert an audit operator above the leaf-level node of the sensitive table in the execution plan (i.e., the nodes that read data from tables or indexes). If the sensitive table is instantiated multiple times (e.g., self-joins), then one audit operator is placed above each instance of the table. It is important to note that database optimizers push single table filters into the leaf node. Therefore, this heuristic has the effect of placing audit operators above both the read and the single table predicate; otherwise, this approach would be equivalent to the READ locks discussed in Section I. We can show that the **leaf-node** heuristic (unlike the **highest-node** heuristic) guarantees no false negatives.

Claim 3.5: The Leaf-Node Heuristic generates an instrumented query plan that produces no false negatives.

Proof: Consider an $ID \in accessedIDs$. Irrespective of the query execution plan, the corresponding tuple would have been accessed at some leaf-level operator in the execution plan and thus passed as an input to the audit operator immediately above it in the plan and thus, $ID \in auditIDs$. ■

While the leaf-node heuristic guarantees no false negatives, this heuristic can incur a large number of false positives. For instance, in the query plan in Figure 2(b), if we assume that the selection predicate on the Patients table and the join predicate are independent and the join selectivity is 1%, then an audit operator placed at the output of the Patients table can result in a false positive rate of 99%.

1) Placement Algorithm: In order to improve the false positive rate of the **leaf-node** heuristic, we use a placement algorithm that is a refinement of it. The key idea is to initially place the audit operator at the leaf-level nodes like the **leaf-node** heuristic and then pull-up the audit operator along the edges of commutative operators (e.g., selections, joins, etc.) In contrast, the audit operator cannot be pulled above non-commutative operators such as top-k operators. We refer to this variant of the placement algorithm as the **highest-commutative-node** heuristic.

Because audit operators are a variation of the filter operator (but act as a no-op), we can use filter commutativity to pull-up the audit operator. However, we note that the placement algorithm is independent of the implementation of the operator. Leveraging commutativity is essential in obtaining an instrumented query plan that produces no false negatives. For instance, consider Example 3.2 that showed that the **highest-node** heuristic can produce false negatives — the query included a $top - k$ operator that is not commutative.

Claim 3.6: The Highest-Commutative-Node Heuristic generates an instrumented query plan that produces no false negatives.

Proof: We present a proof sketch for the special case where we are required to only place one audit operator. The extension to the case when the query involves self-joins is straightforward. For ease of exposition, we use the notation Q to denote both a query and its execution plan.

Algorithm 1 Audit Operator Placement Algorithm

Input: Audit expression E and the query plan for query Q.
Output: Instrumented query plan for query Q.

```
1:  for Each sensitive table T in the query plan do
2:    Q.InsertAuditOperatorAboveTable(T)
3:  end for
4:  pulledUp = True
5:  while pulledUp = True do
6:    pulledUp = False
7:    for Each audit operator A do
8:      parentOperator = Q.parentOperatorOf(A)
9:      if Commute(A, parentOperator) then
10:       Q.pullOperatorAbove(A, parentOperator)
11:       pulledUp = True
12:     end if
13:   end for
14: end while
15: Return the instrumented query plan Q.
```

Fix a query Q given by an arbitrary execution plan for Q. Consider a $sensitiveID$ in the output of E that is *not* entered in $auditID$ by the highest-commutative-node heuristic. In order to prove there are no false negatives, we need to show that the tuple t corresponding to $sensitiveID$ is *not* accessed by the query. Since the audit operator is a no-op, the output of the instrumented plan is the result of Q. Consider a modified plan Q' where the audit operator is replaced with a *real* filter with the predicate `sensitiveID` $\neq$ ID. Since the audit operator does not see any record with $sensitiveID$, the above modified plan produces the same result as Q. Since the audit operator can be moved to the leaf by commutativity and since filter commutativity laws hold independent of the specific filter predicate (so long as the columns named in the predicates are the same), the *real* filter above can also be placed at the leaf to obtain a plan Q'' equivalent to Q'. Therefore, the latter plan Q'' also produces the same result as Q. Running a plan where tuple t is excluded at the leaf is equivalent to running Q in a database where tuple t does not exist (denoted as $Q(D-t)$). Thus, it follows that the results $Q(D)$ and $Q(D-t)$ are the same and tuple t is not accessed by the query. ■

The **highest-commutative-node** heuristic algorithm is described in Algorithm 1. The algorithm takes as input an audit expression and a query execution plan and outputs an instrumented query plan. The audit operators are initially placed above leaf nodes corresponding to the sensitive table of the audit expression. They are then are pulled above other commutative operators (e.g., selections, joins, etc.). While the **highest-commutative-node** heuristic guarantees a query plan that produces no false negatives, it can also incur false positives. However, we can show for an important but restricted class of queries that the heuristic will produce identical results to an offline auditing system.

Theorem 3.7: For the class of SJ queries, the instrumented query plan obtained using the **highest-commutative-node** heuristic does not produce any false positives. [1]

[1]In Section IV-A we discuss how the attributes needed for auditing can be propagated to the audit operator so that there are no false positives for the larger class of select-project-join (SPJ) queries.

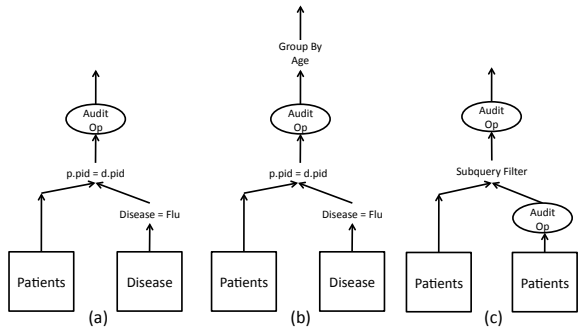


Fig. 4. Instrumented query plans with audit operators.

Proof: Given the class of SJ queries, note that the **highest-commutative-node** heuristic would place the audit operator at the root of the query plan (since both selections and joins are commutative with the audit operator). Consider an $ID \in accessedIDs$, it would be part of the output of the original query for SJ queries and since the audit operator is placed at the root of the plan, $ID \in auditIDs$. ■

Example 3.8: Consider the audit expression for Alice's medical record and the query plans shown in Figure 4 that are created after executing Algorithm 1.

(a) For the query from Example 3.1, a single audit operator is placed at the top of the query plan.

(b) The next query counts the instances of flu by age. The algorithm adds a single audit operator below the group-by operator.

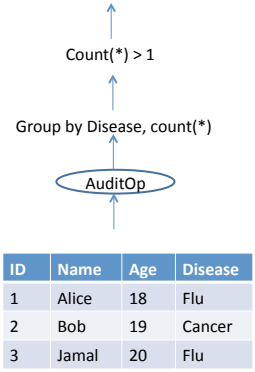
```
SELECT Age, COUNT(D.Disease)
FROM Patients P, Disease D
WHERE P.PatientID = D.PatientID
AND Disease = 'flu'
GROUP BY Age
```

(c) For the following query which involves a nested subquery, two audit operators are added to the query plan, one at the top of the query plan and another at the top of the subquery. The audit operator cannot be pulled out of the subquery because the data would be out of the subquery's scope. The ACCESSED internal state records the union of the accessed tuples.

```
SELECT * FROM Patients P1
WHERE Name
 IN (SELECT Name FROM Patients P2
   WHERE P1.Zip <> P2.Zip )
```

The **highest-commutative-node** heuristic places audit operators at the "highest-possible" edge such that it still produces a query plan with no false negatives. Higher placements typically produce fewer false positives. We note that this simplifying assumption (that operators typically filter rows) may not apply to all workloads and the **highest-commutative-node** heuristic can also yield false positives as the following example shows.

Example 3.9: Consider a health care database and the query plan shown in Figure 5 that finds all diseases with at least two patients. Suppose that all patient records are sensitive. The **highest-commutative-node** heuristic places the audit operator below the group-by. Therefore, the record corresponding to



ID	Name	Age	Disease
1	Alice	18	Flu
2	Bob	19	Cancer
3	Jamal	20	Flu

Fig. 5. False Positives From Highest-Commutative-Node Heuristic.

Bob is added to the ACCESSED internal state. However, Bob's record is not accessed by the query because it is filtered by the HAVING clause. □

2) Placement Summary: This section addressed the question of developing an efficient mechanism to determine if sensitive data are accessed. To this end, we presented the audit operator, which acts as a data viewer and applies a simple predicate to test if sensitive data pass through it. If the audit operator detects sensitive data, it writes the partition-by key to the ACCESSED internal state, which can be used by the SELECT trigger's action to write to the log. To place audit operators in query plans, we presented the **highest-commutative-node** heuristic that attempts to reduce false positives and guarantees no false negatives. We used heuristics instead of attempting to solve the placement problem explicitly because false positives vary with each plan, making it unclear how to always find an optimal placement. For the restricted class of SJ queries, we showed that the placement algorithm is equivalent to the offline auditing system. We note that this work was presented with respect to a single audit expression, but is generalizable to multiple audit expressions that are tested simultaneously.

IV. Implementation

We have prototyped SELECT triggers in a commercial database system (Microsoft SQL Server). Our implementation involved implementing the audit operator and extending the query optimizer and the query execution engine to support the audit operator.

The audit operator is derived from the standard filter operator. As a result we could reuse most of the required modules such as transformation rules and cost estimation to integrate the audit operator into the query optimizer. However, we did modify the audit operator's functionality so that it would act as a no-op (its selectivity was also set to 1.0), and accumulate IDs in the ACCESSED internal state.

A. Physical Audit Operator

The straightforward implementation of an audit operator would be equivalent to a filter operator with an IN clause that evaluates the predicate corresponding to the audit expression E and writes the partition-by IDs to the ACCESSED internal state. However, there are two issues with this approach.

- It requires additional I/Os to access attributes that are referenced in the audit expression but are not required for query evaluation. For instance, consider an audit expression that audits for patients in a particular age group. For some queries, this attribute may not be required for evaluating the query plan.
- It requires additional CPU to propagate attributes that are referenced in the audit expression but again are not required for query evaluation.

Instead, the audit expression is stored as a materialized view of IDs (i.e., the partition-by key) and the audit operator checks if the corresponding IDs are present in its input stream — the set of IDs that are present are written to the ACCESSED internal state.

1) Compiling Audit Expressions to IDs: When an audit expression is declared, it is stored as a materialized view of sensitiveIDs, which are maintained during updates with standard materialized view maintenance algorithms. The advantages of this approach are as follows.

- Microsoft SQL Server uses a clustered index for rowIDs. Because the partition-by key and the clustered index often coincide, compiling an audit expression to the set of corresponding keys has an important advantage that in most cases it does not require any additional I/Os to read the IDs (since they are read anyway).
- Less CPU is needed to propagate only the ID column (note that this is independent of complexity as well as the number of attributes referenced by the audit expression's selection condition).
- Audit operator placement works for audit expressions with joins because the IDs are materialized from a single sensitive table.

Beyond the leaf level nodes, the IDs will be projected only if the operators above need them for evaluating the original query. We implemented an optimization that forces the propagation of IDs in the query plan at the cost of some additional CPU (of course, IDs cannot be propagated through operators such as group-by). Our experiments (Section V) show that propagating IDs incur low overheads for complex queries in the TPC-H benchmark (less than 1%).

2) Audit Operator Implementation: The audit operator essentially needs to perform an intersection between the sensitiveIDs of an audit expression and the input tuples. The audit operator accomplishes this by implementing a "hash-join" where the hash table contains the sensitiveIDs and the hash probes are the input rows. The IDs that are joined are marked as auditIDs. We assume that the sensitiveIDs can fit in memory. If they cannot, standard optimizations such as bloom filters can be used instead. Because audit operators support the getNext interface, they can be placed at the output of any edge in the query execution plan. As far as the rest of query processing is concerned, an audit operator is a no-op. It outputs all input tuples, which is necessary to guarantee the correctness of the query results.

At the end of query execution, the ACCESSED internal state stores the set of auditIDs. These data are then made available to the SELECT trigger's action. For the current implementation, we support actions where auditIDs are written to the log. We plan to extend this functionality in the future.

B. Optimization

The database query optimizer was modified to incorporate the **highest-commutative-node** algorithm. Specifically, because of the physical audit operator implementation (Section IV-A), the algorithm pulls-up audit operators above other operators that commute with an IN clause on the partition-by key.

Logically, audit operators do not influence the choice of the optimal query plan and therefore can be inserted into the query plan before or after optimization. However, modifying optimized query plans proved to be difficult because of the relative complexities of physical operators compared to logical operators. Instead, we inserted audit operators after logical optimization, but before physical optimization. This approach has the benefit that the relative position of the operator is unlikely to change much between logical and physical optimization.

Ideally, the optimizer would generate an instrumented query plan that produces the same query result as a non-instrumented optimized query plan, and ensures the correct placement of audit operators. However, because the audit operator is derived from the filter operator, optimizations can have unexpected side effects. Specifically, Microsoft SQL Server uses a rule based optimizer (e.g., [2]), which contains transformation rules that are unaware of audit operators, and these rules can result in incorrect audit operator placements or incorrect query results. We present two examples to demonstrate these incorrect optimizations.

Example 4.1: Consider the following query and the audit expression for Alice's medical record (Alice has PatientID = 1234). We include the audit expression within the query as bold text, but note that the condition should act as a no-op.

```
SELECT * FROM Patients
WHERE PatientID = 7777
 AND PatientID IN (1234)
```

Initially, because the audit operator is derived from a filter operator, the optimizer believed the selection condition contained a contradiction (i.e., a tuple cannot have different IDs) and therefore incorrectly forced an empty-set result. □

Example 4.2: Similarly, consider the following query and the audit expression for Alice's medical record.

```
SELECT * FROM Patients P1
WHERE PatientID IN (1234) AND PatientID
 IN (SELECT PatientID FROM Patients P2
  WHERE P1.Zip <> P2.Zip
   AND P2.PatientID IN (1234))
```

The optimizer simplified the subquery to a top-1 query because it believed only Alice's ID would be returned. □

We extended optimizer rules to maintain the correct placement of audit operators in query plans, to treat audit operators as no-ops and to prevent audit operators from being optimized with non-audit operators.

V. Experimental Evaluation

In this section, we present an experimental evaluation of our prototype system by modifying a commercial database system (Microsoft SQL Server). We implemented the physical audit operator as described in Section IV-A to test if sensitive data are accessed. For each tuple the audit operator believes has been accessed, the associated partition-by ID is written to an audit log. We implemented both the **leaf-node** heuristic and the **highest-commutative-node** heuristic (denoted as **hcn** heuristic). The **leaf-node** heuristic serves as a reference point to compare against. Our implementation currently supports single table audit expressions and adds a single audit operator to the query execution plan for an audit expression - in particular, our implementation currently does not support queries with self-joins. The framework can be extended to support more complex audit expressions as well as using multiple audit expressions in the same query execution plan. In order to compare the false positive rates, we also implemented a client-side offline auditing tool to compute the set of tuples accessed by a query by suitably rewriting the query as described in [9].

The goals of the experimental evaluation are as follows.

- To measure the overhead of SELECT triggers on query evaluation.
- To measure the overhead of SELECT triggers with respect to parameters such as predicate selectivity, audit expression cardinality (i.e., number of "individuals" being audited for) and query complexity.
- To measure the false positive rate of SELECT triggers with respect to the offline auditing system and their sensitivity to parameters such as predicate selectivity, audit expression cardinality and query complexity.

We use the TPC-H benchmark [18]. We report the results on the 10GB version of the database. The experiments were carried out in a Intel Xeon 2.4GHz machine with 24GB main memory.

A. Results on a Join Query

We first present results on a micro-benchmark involving a join query. The template of the query is shown below.

```
SELECT * FROM orders, customer
WHERE c_custkey = o_custkey
AND   c_acctbal > $1
AND   o_orderdate > $2
```

We used the following audit expression, which audits all customers in a particular market segment. For the 10GB version of the database, this segment corresponds to nearly 20% of the Customer table (around 300K customers).

```
CREATE AUDIT EXPRESSION Audit_Customer AS
 SELECT * from customer
 WHERE c_mktsegment = $3
 FOR SENSITIVE TABLE customer,
 PARTITION BY c_custkey
```

Figure 6 plots the offline audit cardinality (i.e., $accessedIDs$) and the **leaf-node** heuristic audit cardinality (i.e., $auditIDs$) as a function of the selectivity of the predicate in the *Orders* table (varying the parameter $2). As the graph shows, the **leaf-node** heuristic can result in a large number of false positives. For instance, for the data point corresponding to a predicate selectivity of 10%, the **leaf-node** heuristic estimates the number of customer IDs accessed by the query to be around 250K customers when the actual number is around 100K customers. As discussed in Section III, for the class of SJ queries, the **hcn** heuristic is equivalent to an offline auditor and thus does not produce any false positives unlike the **leaf-node** heuristic.

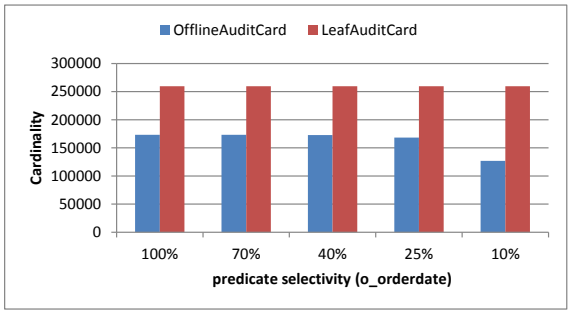


Fig. 6. Micro-Benchmark: False Positives

Figure 7 plots the additional overheads of using the audit operator. We plot the additional overheads relative to the original execution time for both heuristics. The **leaf-node** heuristic always places the audit operator above the customer scan in the join query. As the predicate selectivity of the $orderdate$ predicate increases, the fraction of the customer tuples that satisfy the predicate on the $acctbal$ column but are filtered by the join predicate increases. Thus, the overheads (both the CPU overheads of checking the audit condition as well as the I/O overheads of persisting the IDs) spent on auditing the customer tuples that are subsequently filtered by the join correspondingly increase. As a result, the **leaf-node** heuristic can incur significant overheads (around 10%) - the **hcn** heuristic however checks for the audit expression at the output of the join for the above query and is more robust to the selectivity of the predicate.

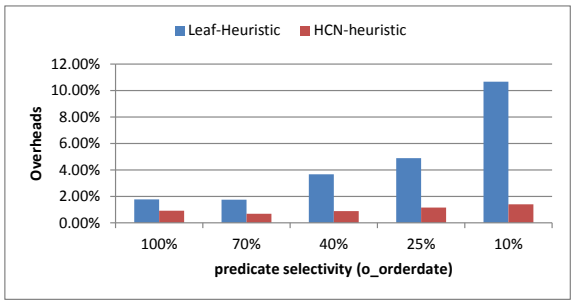


Fig. 7. Micro-Benchmark: Overheads For Predicate Selectivity

B. Varying audit expression cardinality

Another important parameter that influences the overheads of SELECT triggers is the cardinality of the audit expression (i.e., the number of $accessedIDs$). We fix the join query to the instance corresponding to the 40% data point in Figure 7. We vary the audit expression cardinality from 1 (the special

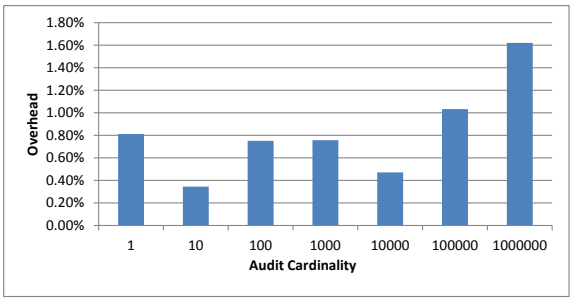


Fig. 8. **HCN** Micro-Benchmark: Overheads For Audit Cardinality

case of single-tuple auditing) up to a million customers. Figure 8 plots the relative overheads for the **hcn** heuristic. As the results indicate, the framework can audit even a large number of customers at a reasonably small overhead. For instance, the additional overhead incurred in auditing a million customers is only around 2%. Of course, this result can vary depending on the query used as well the selectivity of the predicates involved. In the following section, we evaluate our framework using complex queries from the TPC-H benchmark [18].

C. Results on Complex Queries

We present results on a workload of TPC-H queries. We used a subset of seven queries from the query workload - all queries that reference the Customer table and do not include self-joins. The queries included complex aggregates, top-k operators as well as joins of up to 8 tables. We fix the audit expression to all customers in a particular market segment (as in the previous case).

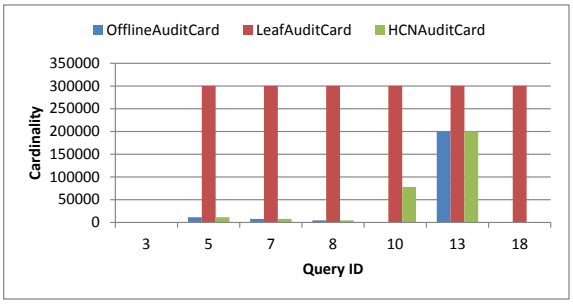


Fig. 9. Evaluating False Positives for Complex Queries

We first compare the false positive rate of the **hcn** and **leaf-node** heuristics against the offline auditing system. Recall that both heuristics can lead to false positives for complex queries that go beyond the class of SJ queries (see Section III). Figure 9 plots the offline audit cardinality (obtained using Definition 2.5) as well as the corresponding audit cardinalities obtained using the **hcn** and **leaf-node** heuristics. The **leaf-node** heuristic has high false positive rates for most queries. This is mainly due to the fact that in the TPC-H benchmark most queries do not have any predicates on the Customer table - as a result, the **leaf-node** heuristic assumes all the customers in the particular market segment are "accessed" while most of these tuples are filtered by subsequent joins with other tables. In contrast, the **hcn** heuristic places the audit operator at the highest point in the query plan such that no false negatives occur while incuring a much lower false positive rate than the **leaf-node** heuristic. We note that the **hcn** heuristic incurs a large number of false positives for Query 10 due to a top-k clause, and requires an offline auditing system to verify the results.

Figure 10 plots the overheads of the audit operator (using the **hcn** heuristic) for the workload. The results show that the **hcn** heuristic incurs low overheads (around 1%) for the queries in the TPC-H benchmark - note that this overhead also includes the additional cost of augmenting the plan with IDs (see Section IV-A2). Given that computing provenance (using Definition 2.5) is not a feasible solution for SELECT triggers, we think that the **hcn** heuristic is a mechanism that provides an interesting trade-off between the false positive rate and the overhead imposed on query execution.

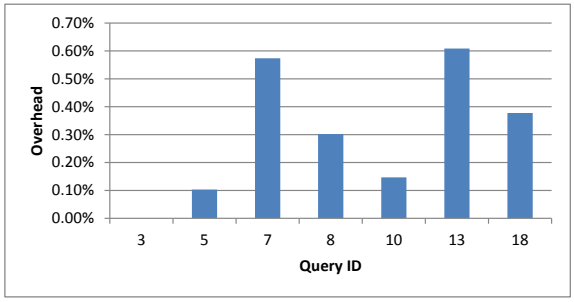


Fig. 10. **HCN** Overheads for Complex Queries

D. Summary

Our experimental evaluation shows that the **hcn** heuristic can work well for the queries in the TPC-H workload: low overheads when auditing a large number of individuals (around a million customers), and low false positive rates for most of the queries in our evaluation. In some cases, a large number of false positives can occur (as shown for Query 10 in Figure 9). Thus, an offline auditing tool is necessary to verify the results. We note that there are other parts of the auditing framework that require more careful study, which we defer to future work. In particular, we need to evaluate the effectiveness of the framework for more complex audit expressions (beyond single table audit expressions).

Another benefit of SELECT triggers is that they reduce the overall auditing run time by filtering queries and their associated accesses that must be analyzed by the offline system. These performance gains are dependent on the database's workload and the trigger's specification (i.e., the amount of sensitive data and the action that is performed). Exploring the scalability of the overall auditing system is an interesting direction for future work.

VI. Related Work

In designing an auditing system, we need to first determine what it means for a query to "access" sensitive data - a form of data provenance. Past work (see [7] for an overview) has developed different methods to address this problem. The key trade-off is the balance between the need to audit arbitrary queries and the need to provide strong privacy guarantees. An *instance independent* auditing approach determines if a query accessed a tuple if there exists *some* database instance in which

the query's result changes when the sensitive tuple is removed [14]. While this type of auditing semantics provides for strong privacy guarantees, it is not applicable for arbitrary SQL (e.g., groupings, aggregates and correlated subqueries).

Alternatively, *instance dependent* auditing semantics determine if a query accessed a tuple if the query's result changes when the sensitive tuple is removed from the *current* database instance [1], [5], [9]. The current database instance is the database state that existed at the time when the query was originally executed. As a result, instance dependent systems require a database that can reconstruct past states either using a temporal database [8] or a "point-in-time" recover API (e.g., [12]). However, unlike the instance independent semantics, this approach is applicable for arbitrary queries. It is important to note that these semantics do not explicitly deal with the problem of inference via updates and other implicit channels.

While data provenance can be computed efficiently for simple classes of queries, the only published technique for provenance computation for complex queries [6] can add a significant overhead (up to of a factor of 5x) for queries in the TPC-H benchmark. Therefore, data auditing is typically deferred to an offline analysis phase.

There has been prior work on a declarative approach for data auditing. Oracle Fine Grained Auditing [17] also provides a declarative interface in which any user can specify an audit expression with single table predicates (e.g. Name = 'Alice'). However, they do not adopt an execution-based approach, but use a static analysis approach to generate an audit log. The query optimizer determines if the query's selection condition logically intersects with the audit expression's selection condition (i.e., using instance independent auditing semantics). If such an intersection exists, the query is deemed to have accessed the sensitive data. The following example illustrates this auditing approach.

Example 6.1: Consider the audit expression DeptName = 'Dermatology' on the table DepartmentNames(DeptID, DeptName). If we assume that the department ID of the 'Oncology' department is 10, then both of the following queries are identical.

```
SELECT * FROM DepartmentNames
WHERE DeptName = 'Oncology'

SELECT * FROM DepartmentNames
WHERE DeptID = 10
```

The static analysis approach would determine that the first query does not reference the audit expression since the intersection is empty. However, it cannot infer the same for the second query, and would incorrectly assume that the query could have accessed the audit expression, leading to a false positive. Our framework using audit operators does not incur a false positive for the above query. □

While the static analysis approach is efficient, it can produce false positives as demonstrated by the example. In fact, this approach would produce false positives for almost all of the queries (with the exception of Query 3) used in our experimental evaluation. Thus, SELECT triggers offer a more robust solution for online auditing.

There is prior work on weaker semantics for provenance computation [19]. However, the work in [19] requires the notion of operator inverses (where a user registers user-defined functions to compute the weak-inversion) and is not applicable for general SQL. In contrast, our focus in this paper is on a light-weight mechanism for computing weak-inverses that is applicable for complex SQL queries (including sub-queries, aggregates, etc.).

VII. Conclusions

There are many scenarios that require row-level auditing support for SELECT queries that cannot be supported by existing database triggers. In this paper we introduce the notion of SELECT triggers as an enabler for data auditing. The key challenge in integrating SELECT triggers in a database system involves engineering a low overhead mechanism while ensuring the semantics are rich enough to capture the subtlety of data access using SQL queries. We present a design which allows for the possibility of false positives for complex queries using a light-weight mechanism termed an audit operator and have prototyped our framework in a commercial database system. Our experiments with the TPC-H benchmark show that our framework yields low false positive rates for most queries while incurring low overhead. We defer a more thorough performance evaluation and possible extensions to optimize offline auditing performance with SELECT triggers to future work.

References

[1] R. Agrawal, R. J. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *VLDB*, 2004.

[2] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.

[3] B. Chen, D. Kifer, K. LeFevre, and A. Machanavajjhala. Privacy-preserving data publishing. *Foundations and Trends in Databases*, 2(1-2), 2009.

[4] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[5] D. Fabbri, K. LeFevre, and Q. Zhu. PolicyReplay: Misconfiguration-response queries for data breach reporting. In *VLDB*, 2010.

[6] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010.

[7] B. Glavic and K. R. Dittrich. Data provenance: A categorization of existing approaches. In *BTW*, 2007.

[8] C. S. Jensen and R. T. Snodgrass. Temporal database. In *Encyclopedia of Database Systems*. 2009.

[9] R. Kaushik and R. Ramamurthy. Efficient auditing for complex SQL queries. In *SIGMOD*, 2011.

[10] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 33(3), 2010.

[11] Microsoft Corporation. SQL Server Integration Services. http://msdn.microsoft.com/en-us/library/ms141026.aspx.

[12] Microsoft Corporation. SQL Server Point-in-time Restore. http://msdn.microsoft.com/en-us/library/ms190982.aspx.

[13] G. Miklau and D. Suciu. A formal analysis of information disclosure in data exchange. In *SIGMOD*, 2004.

[14] R. Motwani, S. U. Nabar, and D. Thomas. Auditing SQL queries. In *ICDE*, 2008.

[15] U.S. Department of Health & Human Services. Health insurance portability and accountability act (HIPAA). http://www.hhs.gov/ocr/privacy/.

[16] C. Olston and B. Reed. Inspector Gadget: A framework for custom monitoring and debugging of distributed dataflows. In *SIGMOD*, 2011.

[17] Oracle Corporation. Oracle audit vault. http://www.oracle.com/us/products/database/audit-vault-066522.html.

[18] The TPC-H Benchmark. http://www.tpc.org.

[19] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.