Whodunit: An Auditing Tool for Detecting Data Breaches

Raghav Kaushik
Microsoft Research
One Microsoft Way
Redmond WA
skaushi@microsoft.com

Ravi Ramamurthy
Microsoft Research
One Microsoft Way
Redmond WA
ravirama@microsoft.com

ABSTRACT
Commercial database systems provide support to maintain an audit trail that can be analyzed offline to identify potential threats to data security. We present a tool that performs *data auditing* that asks for an audit trail of all users and queries that referenced sensitive data, for example "find all queries and corresponding users that referenced John Doe's salary in the last six months". Our tool: (1) handles complex SQL queries including constructs such as grouping, aggregation and subqueries, (2) has privacy guarantees, and (3) incorporates novel optimization techniques for efficiently auditing a large workload of complex SQL queries.

1. INTRODUCTION
Database systems are used today as the primary repository of the most valuable information in any organization. As the volume of sensitive data (e.g., health care information, credit card information) stored in these repositories has increased, protecting the security of the data has gained increasing importance. Further, data compliance laws such as the Sarbanes-Oaxley act and the Health Insurance Portability and Accountability Act (HIPAA) mandates the responsible management of sensitive data.

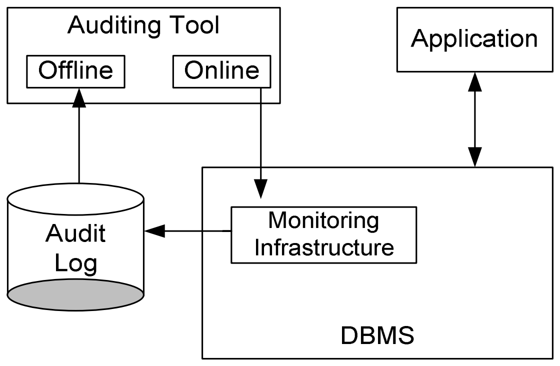


Figure 1. Architecture.

One of the important components of the security infrastructure is an auditing system (see Figure 1) that can be used to aposteriori investigate potential security breaches. Accordingly, there has been an increase in database auditing products on the market including from the major database vendors (e.g., [9] [10]). As the database system is in production, these products monitor various operations such as user logins, queries, data updates and DDL statements to obtain an audit trail. In addition, the database system provides an "audit-analysis" tool which can help in an offline analysis of the audit log to answer questions about access to schema objects. Examples of such *schema auditing* are as follows.

(1) Find queries and updates issued by a given user.

(2) Find queries accessing sensitive columns such as PII columns.

(3) Find failed login attempts as an important user.

In this demonstration, we present a *data auditing* tool that can be used to analyze the audit trail in correlation with the data present in the database. A prototypical example of data auditing is single tuple auditing where the goal is to find all queries and update statements that "referenced" a particular tuple (e.g., find all queries that referenced John Doe's salary). Such analysis is important to discover potential breaches of sensitive information; it was recently reported [2] that Kaiser Permanente recently fired fifteen employees for inappropriately viewing the medical records of Nadya Suleman, the highly publicized "octomom".

In general, data auditing can be more complex including examples such as the following.

(1) Find "important" customers (defined using appropriate filters on the data) that were referenced by queries issued by a particular analyst.

(2) Find queries that reference the account balance of at least three "important" customers.

The main challenges we address are (1) to define a semantics of data auditing that has privacy guarantees and simultaneously leads to a feasible implementation for arbitrarily complex SQL queries including constructs such as grouping, aggregation and subqueries, and (2) to perform auditing efficiently over a potentially large workload of SQL queries. We believe that ours is the first general purpose tool for data auditing that can support arbitrary SQL queries with privacy guarantees. Our tool is based on our recent work [5]. In Section 2, we provide a brief technical overview of the tool (termed Whodunit). Section 3 discusses example demo scenarios.

2. TECHNICAL OVERVIEW
The basis for all data auditing semantics is to define what it means for a query to have referenced a particular tuple (that is,

single-tuple auditing). Prior work has proposed two fundamentally different semantics for data auditing which we can classify broadly as (data) instance dependent and (data) instance independent.

In the instance-dependent approach [1], a query is said to access a tuple if deleting the tuple changes the query result on the database instance where the query was originally run. Unfortunately, even for the special case of single-tuple auditing, subsequent work has shown that the instance dependent approach can lead to breaches of privacy [7][8]. In contrast, an instance independent approach can be used to get strong privacy guarantees [7][8]. Under the instance independent approach, a query is said to have accessed a tuple if there is *some* database instance where deleting it changes the query result.

Previous work [6][7][8]developing the instance independent approach has focused on increasing the class of queries that can be audited efficiently while retaining strong privacy guarantees. However, our recent paper [5] shows that the previously proposed instance independent semantics are computationally incompatible with complex SQL—in the presence of subqueries, enforcing the semantics becomes un-decidable.

Thus, in order to build a general purpose data-auditing tool that can handle arbitrary SQL queries, we fall back to the instance-dependent approach. Our recent work [5] formalizes the instance dependent semantics using the notion of *query differentials*. The notation *Q(D)* denotes the result of executing query *Q* in a database instance *D*.

DEFINITION. *Given a database instance D, a query Q and a tuple t specified by the value v of its primary key, the differential of query Q (denoted Q') is defined as Q rewritten to exclude tuple t from T (by adding the predicate T.id ≠ v). A query is defined to reference tuple t if Q(D) ≠ Q'(D). If Q(D) = Q'(D), we say that Q is safe with respect to t.*

Our instance dependent semantics yields a feasible implementation for an arbitrary query --- it is possible to perform single tuple auditing by running the query and a rewritten version that excludes the tuple and checking if the results are equal.

While we inherit the known privacy limitations of the instance dependent approach [7][8], we show [5] that a weaker privacy guarantee can still be obtained. We introduce the notion of a *risk-free* attack [5] and show that under our instance dependent semantics, no attack is risk-free. Intuitively, this guarantee means that an attacker may get access to sensitive information but not without taking a risk of getting detected. While the above guarantee falls short of the stronger privacy guarantees yielded by the instance independent approach, we believe it offers us an interesting way forward in addressing the full complexity of SQL.

In general, we would like to audit not only a single tuple but more complex audit expressions. Similar to previous work, we formulate our audit expression in the form of a *forbidden view* that captures the sensitive information. For the purposes of this proposal, we consider forbidden views of the form:

```
Create Forbidden View <ViewName> as
Select * From T Where <predicate>
Partition By <T.Key>
```

Our paper [5] discusses a larger class of forbidden views that include joins and arbitrary partitioning columns (and our tool supports the larger class).

For example, in a health care database, we can create a forbidden view to include all patients suffering from AIDS as follows.

```
Create Forbidden View SensitivePatients As
Select * From Patients
Where Disease = “AIDS”
Partition By PatientID
```

Each partition encapsulates the sensitive information corresponding to an individual. In the above example, each partition corresponds to an individual patient record that is sensitive. (We note again that our tool supports a larger class of forbidden views presented in our paper [5] where the sensitive information corresponding to each individual can span multiple records.)

Our tool performs data auditing based on a novel REFERENCES operator. The operator (see Figure 2) takes as input a set of queries and a forbidden view with partition id PID and outputs (Q, PID) pairs such that query Q references partition PID.

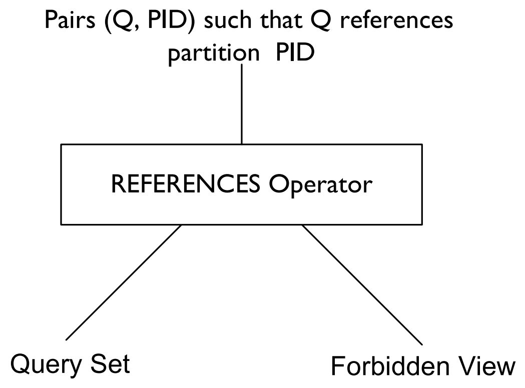


Figure 2. References Operator.

A straightforward implementation of the above operator performs a cross product of the partitions and the queries and for each pair, computes the differential of the query and checks if it is equal to the original query by running them. The straightforward implementation can be prohibitively expensive. Accordingly, our tool includes optimizations that help improve the efficiency of data auditing. An important optimization that our tool uses is the notion of an audit optimizer that can check if a query and its differential are equal *without any execution*. The idea is to start with an algebraic plan for a query and find if we can “reach” a plan for the corresponding differential query by transforming the initial plan. The plan is transformed using equivalence rules in the usual way deployed by any rule-based query optimizer. The main difference is that in addition to the standard rules that hold for all database instances (e.g., pushing a selection below join, join commutativity etc.), we also handle rules that are instance specific. Instance specific rules are naturally derived from audit checks—a query that passes the audit is equivalent to the rewritten query. Thus, we can leverage the results of previously audited queries to optimize the overheads of auditing. Our paper [5] carries a more detailed description of the audit optimizer.

In general, we note that the query set can also include update statements. We currently handle updates by finding the query underlying the update and extending our semantics for queries. Our paper [5] carries a more detailed discussion on updates.

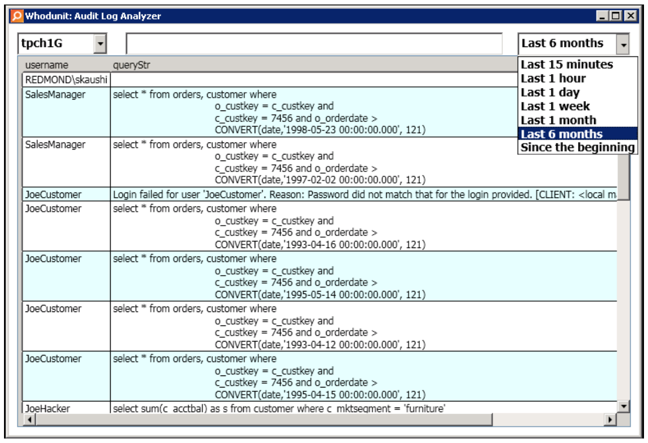
The REFERENCES operator in conjunction with other relational operators provides a natural programming interface for expressing data auditing tasks such as the examples in the introduction. In the following section, we describe our prototype and outline a sample demo scenario.

3. DEMO SCENARIO

The current prototype is built as a client application and uses Microsoft SQL Server as the underlying database system (see [5] for details). Our prototype builds on the existing audit analysis/audit log viewer tool [10] that is provided by Microsoft SQL Server. We now provide a brief overview of the different features supported by our tool.

3.1 Schema Auditing

As mentioned in the introduction, audit log analysis tools provide interfaces for simple search queries over the audit log. Our tool can support such schema auditing functionality.

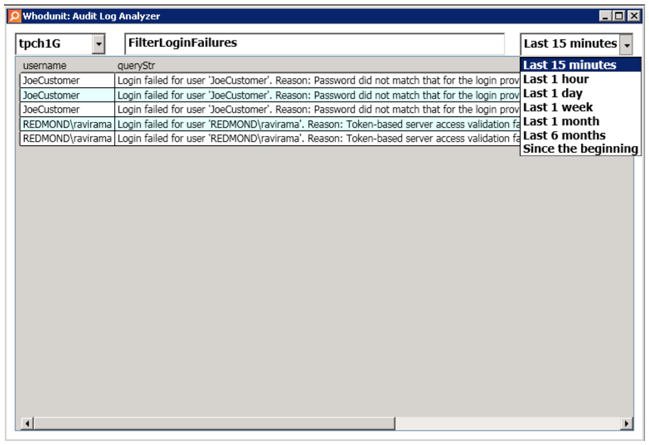


username	queryStr
REDMOND\skaushi	
SalesManager	select * from orders, customer where o_custkey = c_custkey and c_custkey = 7456 and o_orderdate > CONVERT(date,'1998-05-23 00:00:00.000', 121)
SalesManager	select * from orders, customer where o_custkey = c_custkey and c_custkey = 7456 and o_orderdate > CONVERT(date,'1997-02-02 00:00:00.000', 121)
JoeCustomer	Login failed for user 'JoeCustomer'. Reason: Password did not match that for the login provided. [CLIENT: <local m
JoeCustomer	select * from orders, customer where o_custkey = c_custkey and c_custkey = 7456 and o_orderdate > CONVERT(date,'1993-04-16 00:00:00.000', 121)
JoeCustomer	select * from orders, customer where o_custkey = c_custkey and c_custkey = 7456 and o_orderdate > CONVERT(date,'1995-05-14 00:00:00.000', 121)
JoeCustomer	select * from orders, customer where o_custkey = c_custkey and c_custkey = 7456 and o_orderdate > CONVERT(date,'1993-04-12 00:00:00.000', 121)
JoeCustomer	select * from orders, customer where o_custkey = c_custkey and c_custkey = 7456 and o_orderdate > CONVERT(date,'1995-04-15 00:00:00.000', 121)
JoeHacker	select sum(c_acctbal) as s from customer where c_mktsegment = 'furniture'

Figure 3. Filtering using Timestamp.

Figure 3 illustrates the user interface for the tool. In addition to a basic search box, there are drop-down menus for common operations such as selecting a database as well as filtering the audit log based on the timestamp of the event. For example, Figure 3 illustrates how we can filter the audit log events based on a predicate on the timestamp of the event. The set of events to be displayed in the user interface is configurable; we only show the *username* and *queryStr* fields in the following examples for brevity.

The user can use the search box for a keyword search over the contents of the audit log. In addition, our system supports relational operators over the audit log. For example, Figure 4 illustrates how we can check for login failures in the last fifteen minutes by using the FilterLoginFailures primitive.



username	queryStr
JoeCustomer	Login failed for user 'JoeCustomer'. Reason: Password did not match that for the login prov
JoeCustomer	Login failed for user 'JoeCustomer'. Reason: Password did not match that for the login prov
JoeCustomer	Login failed for user 'JoeCustomer'. Reason: Password did not match that for the login prov
REDMOND\ravirama	Login failed for user 'REDMOND\ravirama'. Reason: Token-based server access validation fa
REDMOND\ravirama	Login failed for user 'REDMOND\ravirama'. Reason: Token-based server access validation fa

Figure 4. Primitive Operators for Filtering.

3.2 References Operator

In addition to providing support for basic schema auditing (as discussed in the previous examples), our tool supports the REFERENCES operator which significantly expands the scope of audit analysis to include data auditing.

In order to demonstrate the utility of our tool for data auditing, we now illustrate a sample "session". We will use the TPC-H [11] database for illustration. The TPC-H database includes a CUSTOMER table which we assume contains sensitive information such as credit card information or customer account balance.

Consider the following scenario. Assume that the sensitive data of interest is the customer information of "premium" customers – customers that have a high account balance (say greater than $100,000). In order to detect data breaches, assume the security administrator would like to carry out the following task:

"*Find all premium customers whose information has been accessed by user JoeAnalyst within the last week*"

Clearly, a simple search interface over the audit log is insufficient for executing the above task; we now illustrate how this task is enabled in our prototype.

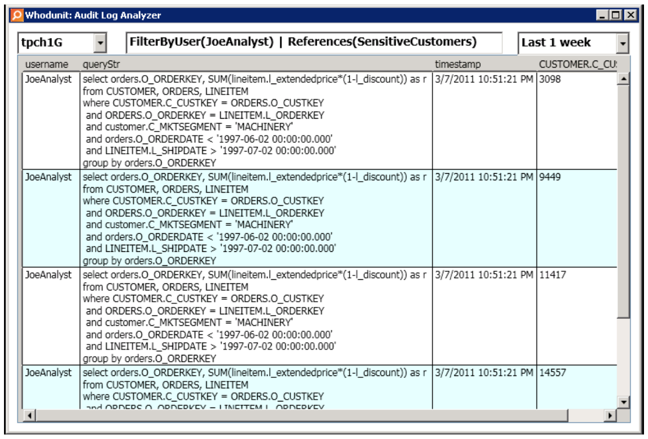
We first create a forbidden view to capture the sensitive customers as follows:

```
Create Forbidden View SensitiveCustomers As
Select * From Customers
Where C_Acctbal > 100K
Partition By CustomerID
```

Figure 5 shows how we can then execute the above task via operator composition. We pose the following command to execute the above task.

```
FilterByUser(JoeAnalyst) | References(SensitiveCustomers)
```

The first operator obtains the subset of queries issued by JoeAnalyst in the last week and the second operator is used to check if any of the premium customers were referenced by these queries. The output window indicates for each query issued by JoeAnalyst, the corresponding customer IDs of the premium customers referenced by it.



```
FilterByUser(JoeAnalyst) | References(SensitiveCustomers)

username    queryStr                                                                      timestamp               CUSTOMER.C_CU:
JoeAnalyst  select orders.O_ORDERKEY, SUM(lineitem.l_extendedprice*(1-l_discount)) as r  3/7/2011 10:51:21 PM    3098
            from CUSTOMER, ORDERS, LINEITEM
            where CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
             and ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
             and customer.C_MKTSEGMENT = 'MACHINERY'
             and orders.O_ORDERDATE < '1997-06-02 00:00:00.000'
             and LINEITEM.L_SHIPDATE > '1997-07-02 00:00:00.000'
            group by orders.O_ORDERKEY
JoeAnalyst  select orders.O_ORDERKEY, SUM(lineitem.l_extendedprice*(1-l_discount)) as r  3/7/2011 10:51:21 PM    9449
            from CUSTOMER, ORDERS, LINEITEM
            where CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
             and ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
             and customer.C_MKTSEGMENT = 'MACHINERY'
             and orders.O_ORDERDATE < '1997-06-02 00:00:00.000'
             and LINEITEM.L_SHIPDATE > '1997-07-02 00:00:00.000'
            group by orders.O_ORDERKEY
JoeAnalyst  select orders.O_ORDERKEY, SUM(lineitem.l_extendedprice*(1-l_discount)) as r  3/7/2011 10:51:21 PM    11417
            from CUSTOMER, ORDERS, LINEITEM
            where CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
             and ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
             and customer.C_MKTSEGMENT = 'MACHINERY'
             and orders.O_ORDERDATE < '1997-06-02 00:00:00.000'
             and LINEITEM.L_SHIPDATE > '1997-07-02 00:00:00.000'
            group by orders.O_ORDERKEY
JoeAnalyst  select orders.O_ORDERKEY, SUM(lineitem.l_extendedprice*(1-l_discount)) as r  3/7/2011 10:51:21 PM    14557
            from CUSTOMER, ORDERS, LINEITEM
            where CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
```

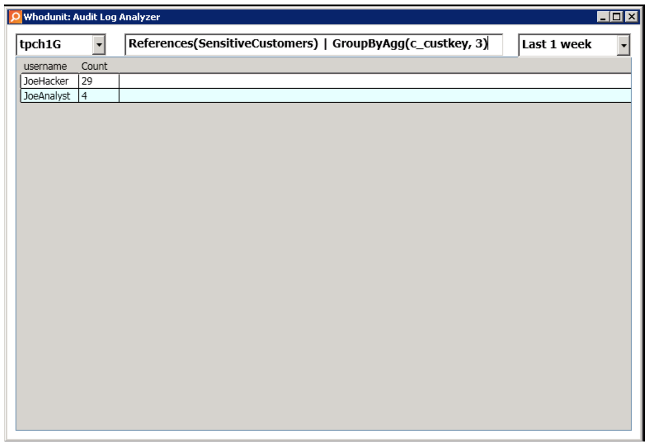
Figure 5. References Operator.

3.3 Support for Ad-hoc Analytics

Our tool also supports a variety of primitive aggregate/grouping functions which can be used for more sophisticated ad-hoc analytics over the audit log. Consider the following task that is a variant of the previous example.

"*Find all users that have accessed the information of at-least three premium customers within the last week*"

This task can be again supported in our tool via operator composition.



```
References(SensitiveCustomers) | GroupByAgg(c_custkey, 3)

username    Count
JoeHacker   29
JoeAnalyst  4
```

Figure 6. Example of Analytics.

Figure 6 shows that we pose the command

References(SensitiveCustomers) | GroupbyAgg(c_custkey, 3)

to compose the References operator (as in the previous analysis) with a group-by operator to figure out the users who have accessed at-least three distinct premium customers.

To summarize, in this demonstration we present a tool for audit log analysis with several novel features including: 1) Data auditing using the notion of forbidden views 2) Flexible set of composable operators that enable rich analytics over the audit log 3) Support for arbitrary SQL queries with privacy guarantees.

4. ACKNOWLEDGMENTS

We would like to thank Yuri Siradeghyan for assisting us in developing the user interface of the tool.

5. REFERENCES

[1] R.Agrawal et al., "*Auditing Compliance with a Hippocratic Database"* in VLDB 2004.

[2] J.Cart, "*Kaiser fires staffers who snooped into suleman's files*" in The Los Angeles Times. March 31 2009.

[3] D.Fabbri, K.Lefevre, D.Zhu, "*PolicyReplay: Misconfiguration Response Queries for Data Breach Reporting"* in PVLDB 3(1): 36-47(2010).

[4] C.Farkis, S. Jajodia, "*The Inference Problem. A Survey*", ACM SIGKDD Explorations 4(2):6-11(2002).

[5] R.Kaushik, R.Ramamurthy, "*Efficient Auditing for Complex SQL Queries*" in SIGMOD 2011.

[6] A. Machanavajjhala , J.Gehrke, "*On the efficiency of checking perfect privacy*" in PODS 2006.

[7] G. Miklau, D. Suciu, "*A Formal Analysis of Information Disclosure in Data Exchange*" in SIGMOD 2004.

[8] R.Motwani, S.U.Nabar, D.Thomas, "*Auditing SQL Queries*" in ICDE 2008.

[9] Oracle Audit Vault. http://www.oracle.com/technetwork/database/audit-vault/

[10] "Understanding SQL Server Audit". http://msdn.microsoft.com/en-us/library/cc280386.aspx

[11] TPC-H Benchmark. http://www.tpc.org/tpch/