PolicyReplay: Misconfiguration-Response Queries for Data Breach Reporting

Daniel Fabbri*, Kristen LeFevre*, Qiang Zhu†

*Electrical Engineering & Computer Science, University of Michigan, 2260 Hayward Ave. Ann Arbor, MI 48109 USA
†Computer and Information Science, University of Michigan - Dearborn, 4901 Evergreen Rd. Dearborn, MI 48128 USA
{dfabbri, klefevre, qzhu}@umich.edu

ABSTRACT

Recent legislation has increased the requirements of organizations to report data breaches, or unauthorized access to data. While access control policies are used to restrict access to a database, these policies are complex and difficult to configure. As a result, misconfigurations sometimes allow users access to unauthorized data.

In this paper, we consider the problem of reporting data breaches after such a misconfiguration is detected. To locate past SQL queries that may have revealed unauthorized information, we introduce the novel idea of a *misconfiguration response (MR) query*. The MR-query cleanly addresses the challenges of information propagation within the database by replaying the log of operations and returning all logged queries for which the result has changed due to the misconfiguration. A strawman implementation of the MR-query would go back in time and replay all the operations that occurred in the interim, with the correct policy. However, re-executing all operations is inefficient. Instead, we develop techniques to improve reporting efficiency by reducing the number of operations that must be re-executed and reducing the cost of replaying the operations. An extensive evaluation shows that our method can reduce the total runtime by up to an order of magnitude.

1. INTRODUCTION

During the past several years, there has been growing interest in building support for regulatory compliance into database systems. One emerging class of regulations requires organizations to report data breaches (instances of unauthorized access). In healthcare, for example, the United States' 2009 Health Information Technology for Economic and Clinical Health Act (HITECH), greatly expanded the security and privacy protections afforded by the earlier HIPAA legislation. Among the new protections, HITECH requires that "covered entities" (e.g., hospitals) notify individuals and the government about any breach of protected health information.

Under normal circumstances, data in databases are protected by access control policies, which designate which users can access which data. However, access control policies are notoriously difficult to configure, and mistakes are common [18]. In this paper, we consider a common problem: A misconfiguration is detected in an access control policy after that policy has already been deployed for a period of time. To comply with data breach reporting rules, it is necessary to go back in time and determine which queries may have revealed unauthorized information.

An obvious approach to this problem is to maintain a log of all SQL queries and to retrieve those queries that explicitly accessed unauthorized data. However, if done naively, this fails to account for the fact that unauthorized information may have propagated (explicitly or implicitly) via updates. For example, when an insert operation copies a row from one table to another, this operation creates an explicit flow of information; a user who reads the copied row learns the original value. Similarly, when a row is updated based on a condition (e.g., UPDATE Patients SET Age = 'XXX' WHERE Name = 'Bob'), the operation creates an implicit flow of information since the value XXX implicitly reveals that Name = Bob. Thus, not only may a misconfigured access control policy allow a user to access unauthorized parts of the database, but the unauthorized data can be propagated to other parts of the database that can be read by future queries. Past work on database auditing considered the task of retrieving logged SQL queries that were affected by user-specified "sensitive" data, but did not address the challenge of updates [1].

1.1 Challenges

Effectively responding to database access control misconfigurations presents several challenges:

- **Updates:** The solution should be able to find all past queries that revealed unauthorized information, either by directly accessing unauthorized data or as the result of information propagation caused by update operations.
- **Lightweight/Non-disruptive:** The solution should easily integrate with existing DBMS infrastructure and introduce minimal overhead during normal database operation.
- **Efficient Response:** When a policy misconfiguration is detected, the solution should efficiently identify past queries that may have revealed unauthorized information.

1.2 Our Contributions

In response to these challenges, we propose the PolicyReplay framework, which is highlighted by the following contributions:

- We introduce the novel idea of a declarative *misconfiguration response (MR) query*, which retrieves all past database queries that may have revealed unauthorized information. Our approach is based on the following insight, which cleanly addresses the problem of information propagation: Conceptually, the MR-query returns to the point of the misconfiguration, and completely replays

*This work was supported by NSF grants CNS-0915782 and IGERT-0903629.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

the log of operations (updates and queries) using the new (correct) policy. If a query returns exactly the same result under the old (incorrect) and new (correct) policies, we know that the query has not revealed any unauthorized information. However, if the result has changed, the query is marked as "suspicious" since unauthorized information may have been revealed. The formal semantics are described in Section 3.

- Of course, the naive algorithm of returning to the point of the misconfiguration and completely replaying all past database operations is inefficient. To improve efficiency, in Section 4 we introduce a set of optimizations based on *static pruning*, *delta tables*, and *partial and simultaneous re-execution*. Our experiments (Section 5) indicate that by replaying operations in an efficient manner we are able to reduce the total runtime by an order of magnitude in several common cases.

2. PRELIMINARIES

The PolicyReplay framework supports a modern row-level access control model. During normal database operation, we maintain an *operation log*, which records the text of all SQL operations, as well as a *transaction-time backlog database* representation of historical data [10]. These structures are easily incorporated into an existing DBMS, and past work has shown that they can be maintained with little impact on normal database operation [1].

2.1 Row-Level Access Control Policies

The goal of our work is to develop methods for responding to misconfigurations in database access control policies. While the access control model itself is not our main contribution, we will use a modern row-level access control model (e.g., as found in Oracle's fine-grained access controls [17]) throughout the paper.

Formally, we will consider an old policy P_{old} and a new policy P_{new}. For each database user U, policy P_{old} (respectively, P_{new}) contains a selection of the form $\sigma_{S_{old}}(S)$ (respectively, $\sigma_{S_{new}}(S)$) for every table S in the database, where S_{old} (respectively, S_{new}) is a boolean condition involving only the attributes in table S. A user U is given access to the subset of the database specified by the selection conditions in the policy. When the user executes a database command (i.e., a SELECT, INSERT, UPDATE, or DELETE statement), each table S referenced by the command is transparently replaced with the view defined by the additional selection.

EXAMPLE 2.1. *As a simple example, consider a hospital database that contains a single table $Patients$. Consider a user Dan, and suppose that under policy P_{old} Dan is allowed to see the rows of the Patients table with $Age < 30$ (i.e., $Patients_{old} = (Age < 30)$). If Dan issues the query* `SELECT * FROM PATIENTS` *while P_{old} is in effect, then the query is automatically rewritten as* `SELECT * FROM PATIENTS WHERE Age < 30`.

Updates are handled similarly. Suppose that Dan issues the data modification command `UPDATE Patients SET Department = 'Pediatrics'`; *this is rewritten as* `UPDATE Patients SET Department = 'Pediatrics' WHERE Age < 30`.

Of course, row-level access control has its shortcomings, and alternatives have been proposed [12, 19]. Nonetheless, the row-level approach remains popular.

2.2 Operation Log

During normal database operation, a log is maintained, which records the text of all DML operations (SELECT, INSERT, UPDATE, and DELETE) that are performed. The operation log itself is an append-only table; each entry in the log contains the $timestamp$ at which the operation was executed, as well as the associated sql (a string). It may also contain additional fields, such as usernames, etc. Figure 1 shows a (pared-down) operation log.

Many existing database systems already support this kind of logging [11, 17, 21], and past work has demonstrated that the performance impact during normal database operation is minimal [1]. For the purposes of this work, we will assume that log entries are written within the same transaction as the recorded operation; this guarantees that the timestamp ordering on the log records is equivalent to the serialization order of the operations.

2.3 Transaction-Time Backlog Database

In addition to the operation log, we will make use of a simplified transaction-time backlog database [10], which supports two basic operations: insert a row and delete a row. Each time a row is inserted or deleted, the system records the following information: the time, the type of operation (insert or delete), and the value of the row being inserted or deleted.[1] Under this representation, rows are never modified or deleted in place; instead, the new or updated row is appended to the end of the table.

Formally, we will use the notation DB^b to refer to a transaction-time backlog database, and S^b to refer to a single backlog table in the database. We will use DB^τ to refer to the static snapshot database that exists at time τ and S^τ to refer to the static snapshot of table S at time τ. Any DB^τ can be constructed from DB^b using methods described in [10].

EXAMPLE 2.2. *Consider the backlog table $Patients^b$ shown in Figure 2(a), and suppose that Id is the primary key for $Patients$. $Patients^3$, the static snapshot of the table at time 3, is as follows:*

Id	*Name*	*Age*	*Disease*
1	*Alice*	*10*	*Flu*
2	*XXX*	*20*	*Ulcer*
3	*Carlos*	*35*	*Broken Arm*

3. MR-QUERIES: PROBLEM DEFINITION

When a policy misconfiguration is detected, a security administrator or compliance officer needs to take steps to respond to the misconfiguration (e.g., report the incident to government regulators). In order to do this, she must determine which queries have revealed unauthorized information. We will refer to this task as a *misconfiguration-response (MR) query*. The MR-query plays a central role in breach reporting (see Appendix B).

The goal of the MR-query is to retrieve every logged SQL query that disclosed unauthorized information. In the simplest case, a logged query is returned by the MR-query if it explicitly reads unauthorized data. However, this simple approach does not capture the propagation of information via updates. Instead, our problem formulation is based on the observation that a query is guaranteed *not* to reveal unauthorized information if its result is the same under the old (incorrect) and new (correct) policies. In contrast, when the query results are not the same, there is no such guarantee; we will refer to these SQL queries as *suspicious queries*.

DEFINITION 1 (MR-QUERY). *The MR-query takes as input the old policy P_{old}, the new policy P_{new}, timestamps t_1 and t_2 ($t_1 \leq t_2$), the operation log, and the transaction-time backlog database DB^b. The MR-query returns the set of all entries e in the operation log that contain SELECT queries, have timestamps between t_1 and t_2, and such that the results of the queries would have been different if P_{old} had been replaced with P_{new}, effective at time t_1.*

[1]The effects of SQL UPDATE commands are captured as follows: If a tuple is modified, we model this as a delete (of the old tuple), followed by an insert (of the new tuple).

lid	timestamp	sql
L1	1	SELECT * FROM Patients
L2	2	SELECT * FROM Patients WHERE Age < 12
L3	3	UPDATE Patients SET Name = 'XXX' WHERE Name = 'Bob'
L4	4	SELECT * FROM Patients WHERE Name = 'XXX'
L5	5	INSERT INTO Temp SELECT Id, Name, Age, Disease FROM Patients
L6	6	SELECT * FROM Temp

Figure 1: Sample Operation Log

Time	Op	Id	Name	Age	Disease
0	Ins	1	Alice	10	Flu
0	Ins	2	Bob	20	Ulcer
0	Ins	3	Carlos	35	Broken arm
3	Del	2	Bob	20	Ulcer
3	Ins	2	XXX	20	Ulcer

(a) $Patients^b$

Time	Op	Id	A	B	C
5	Ins	1	Alice	10	Flu
5	Ins	2	XXX	20	Ulcer

(b) $Temp^b$

Figure 2: The backlog database DB^b is the result of the actual execution (when P_{old} was in place).

The semantics of an MR-query are easily understood in terms of a naive algorithm. It begins by constructing a second backlog database $\widehat{DB}^b$ by creating a new backlog table $\widehat{S}^b$ for each table S^b in DB^b, and copying into $\widehat{S}^b$ every row in S^b with $timestamp \leq t_1$. Starting from t_1, the algorithm replays the operation log (updates and queries) using the new policy P_{new}, and applying all data modifications to $\widehat{DB}^b$. For every SELECT query in the log, it compares the result obtained using P_{new} and $\widehat{DB}^b$ to the result obtained using P_{old} and DB^b. Details are provided in Algorithm 1 in Appendix A.

EXAMPLE 3.1. *Consider a database consisting of two tables: $Patients(Id, Name, Age, Disease)$ and $Temp(Id, A, B, C)$. In both tables, Id is the primary key.*

Suppose that at time 1 the administrator deploys a policy allowing user Dan to see only those rows of $Patients$ with $Age < 30$ (i.e., $Patients_{old} = (Age < 30)$), and all rows of $Temp$ (i.e., $Temp_{old} = true$). Later, at time 7, she discovers that the policy was misconfigured, and she corrects the policy so that Dan can see only those Patients with $Age < 15$ (i.e., $Patients_{new} = (Age < 15)$ and $Temp_{new} = true$). Unfortunately, the misconfigured policy was in effect for the period between $t_1 = 1$ and $t_2 = 7$. In order to respond to the misconfiguration, the administrator needs to figure out which queries, evaluated during this time, would have produced different results if the correct policy had been in place.

Figure 1 shows an example operation log, and Figure 2 shows the backlog database DB^b. Observe that if P_{old} had been replaced with P_{new} at time 1, some of the resulting data modifications would have been different. The backlog database resulting from this case (denoted $\widehat{DB}^b$) is shown in Figure 3. For example, notice that the update in L_3 will not affect any tuples when P_{new} is in effect because user Dan does not have access to Patients with $Age > 15$ (i.e., Bob) in this case.

Finally, Figure 4 compares the results of each SELECT query when evaluated using DB^b and P_{old} and using $\widehat{DB}^b$ and P_{new}. Notice that the queries in L_1, L_4, and L_6 return different results; thus these log entries are returned as the result of the MR-query.

Time	Op	Id	Name	Age	Disease
0	Ins	1	Alice	10	Flu
0	Ins	2	Bob	20	Ulcer
0	Ins	3	Carlos	35	Broken arm

(a) $\widehat{Patients}^b$

Time	Op	Id	A	B	C
5	Ins	1	Alice	10	Flu

(b) $\widehat{Temp}^b$

Figure 3: The backlog database $\widehat{DB}^b$ is what would have resulted if P_{old} were replaced with P_{new}, effective at time 1.

Id	Name	Age	Disease
1	Alice	10	Flu
2	Bob	20	Ulcer

(a) L_1 (Old)

Id	Name	Age	Disease
1	Alice	10	Flu

(b) L_1 (New)

Id	Name	Age	Disease
1	Alice	10	Flu

(c) L_2 (Old)

Id	Name	Age	Disease
1	Alice	10	Flu

(d) L_2 (New)

Id	Name	Age	Disease
2	XXX	20	Ulcer

(e) L_4 (Old)

Id	Name	Age	Disease

(f) L_4 (New)

Id	A	B	C
1	Alice	10	Flu
2	XXX	20	Ulcer

(g) L_6 (Old)

Id	A	B	C
1	Alice	10	Flu

(h) L_6 (New)

Figure 4: Comparing the results of logged queries to illustrate the semantics of MR-queries

Less strict approaches may be insufficient to detect the disclosure of unauthorized information. For example, an alternative method might attach annotations to each row (e.g., [5]) and propagate these annotations from row to row as the log of operations is executed to track dependencies. A query would be marked suspicious if a row in the result is dependent on a row that is only accessible due to the misconfiguration. Therefore, by only executing the log under the old policy and combining the query results with annotation information, this approach can detect some suspicious activity; however, it has significant weaknesses that are managed by the MR-query.

EXAMPLE 3.2. *Consider the policies $Patients_{old} = (Age < 30)$, $Patients_{new} = (Age < 18)$ and $Temp_{new} = Temp_{old} = true$ and the log of operations and data in Figure 5. Initially, the user learns from operation O1 that Bob has flu. Then, the delete operation deletes those rows in Temp that have the same disease as a row in the Patients table. In this example, Bob is deleted because Alice also has the flu; however, Alice is only accessible due to the misconfiguration. Therefore, the user inappropriately learns from the empty set result of operation O3 that someone in the Patients table has the flu. Unfortunately, the annotation method cannot detect this unauthorized access because the result of O3 on $Temp$ is the empty set and contains no annotations to analyze. In contrast, the MR-query will mark operation O3 as suspicious since the result is different between the policies.*

As the example shows, the absence of a row in the result can

Time	Op	Id	Name	Age	Disease	Annotations
0	Ins	1	Alice	25	Flu	P1

(a) $Patients^b$

Time	Op	Id	Name	Age	Disease	Annotations
0	Ins	1	Bob	10	Flu	T1
2	Del	1	Bob	10	Flu	P1, T1

(b) $Temp^b$

lid	time	sql
O1	1	SELECT * FROM Temp
O2	2	DELETE FROM Temp t USING Patients p WHERE t.disease = p.disease
O3	3	SELECT * FROM Temp

(c) Operation Log

Id	Name	Age	Disease

(d) Result for $O3$ on Temp

Id	Name	Age	Disease
1	Bob	10	Flu

(e) Result for $O3$ on $\widehat{Temp}$

Figure 5: A backlog database and log to demonstrate the weaknesses of annotation methods.

result in the disclosure of unauthorized information when combined with the result of other queries. Unfortunately, annotation approaches are not able to detect these breaches. [2]

4. MR-QUERY EVALUATION

The naive algorithm is useful for expressing the semantics of an MR-query, but it would be inefficient to actually evaluate an MR-query in this way. In this section, we describe a set of optimizations, which greatly improve the efficiency of MR-queries:

- **Static Pruning:** When the operation log contains only queries (SELECT statements), in Section 4.2 we provide a static pruning condition by which we can determine that certain queries are unsuspicious, without re-executing them.
- **Delta Tables:** In Section 4.3, we extend the static pruning condition to an operation log that also contains data modifications (INSERT, UPDATE, and DELETE statements). The idea is to store a concise description of how the database has changed as a result of the new policy (i.e., the difference between $\widehat{DB}^b$ and DB^b) using structures called *delta tables*. Then, we extend the pruning conditions from the query-only case.
- **Partial and Simultaneous Re-Execution:** When the above pruning strategies fail, it is necessary to re-execute certain operations. To improve performance in this case, in Section 4.4, we introduce two further optimizations: The first is based on the observation that we may be able to determine mid-execution that an operation is unsuspicious, in which case we can stop executing the operation. While the naive algorithm requires that we entirely re-execute each query twice (once on $\widehat{DB}$ using P_{new} and once on DB using P_{old}), our second optimization is based on the observation that these two queries actually share much computation; thus we propose to execute the two queries simultaneously.

4.1 Class of Operations

For the remainder of this paper, we will restrict our discussion to the following classes of logged SQL operations: (i) select-project-join (SPJ) queries, (ii) insert operations where the rows to be inserted are determined by an SPJ query, (iii) update operations where attributes of a row are set to constant values if the row satisfies a selection condition, and (iv) delete operations where a row is deleted if it satisfies a selection condition. For simplicity, we will not address the larger class of aggregate-select-project-join (ASPJ) queries, but many of our techniques can be applied to this case.

[2]There are more subtle cases as well. For example, consider the case where the old policy is too strict (e.g., for some table S, $S_{old} \subseteq S_{new}$). Operations like MINUS or EXCEPT can produce unauthorized accesses.

4.2 Static Pruning (Queries Only)

We begin with the simplest case, where the operation log contains *only* queries (i.e., SELECT statements). In this case, it is sometimes possible to determine statically that a query is not suspicious (i.e., without re-executing the query). As we will see, this case is not particularly practical on its own, but it provides a building block for the general case in Section 4.3. This static pruning condition is formalized through *delta expressions*.

DEFINITION 2 (DELTA EXPRESSIONS). *Delta expressions logically describe the differences between the old and new policies. Specifically, for table S:*

- $\delta_S^- = S_{old} \wedge \neg S_{new}$ *is a* logical description *of the tuples from S that are visible to the user under the old policy, but not under the new policy.*
- $\delta_S^+ = S_{new} \wedge \neg S_{old}$ *is a* logical description *of the tuples from S that are visible to the user under the new policy, but were not visible under the old policy.*

The intuition for the static pruning condition is straightforward. Each table S can be broken down, logically, into three components: (1) tuples that were visible under the old policy, but are no longer visible under the new policy (δ_S^-), (2) tuples that were not visible under the old policy, but are visible under the new policy (δ_S^+), and (3) tuples whose visibility is unchanged. If we can determine that a query's selection condition filters out all tuples in δ_S^+ and δ_S^- (i.e., any tuples whose visibility has changed), and no rows from S have been modified, then we know that the result of the query was not affected by the policy misconfiguration. This intuition is formalized by the following theorem. (The proof can be found in Appendix C.)

THEOREM 1. *Consider a database with relations $S_1, ..., S_n$, and suppose that the operation log contains only queries (no updates). Let Q be a query in the log with associated selection condition C.*[3] *If the expression $C \wedge (\delta_{S_1}^- \vee \delta_{S_1}^+ \vee ... \vee \delta_{S_n}^- \vee \delta_{S_n}^+)$ is not satisfiable, then Q must* not *be suspicious.*[4]

EXAMPLE 4.1. *To illustrate the static pruning condition, consider the first two log records in the operation log shown in Figure 1, and suppose again that $Patients_{old} = (Age < 30)$ and $Patients_{new} = (Age < 15)$. Thus, $\delta_{Patients}^- = (15 \leq Age < 30)$ and $\delta_{Patients}^+ = false$.*

We are not able to prune $L1$ using Theorem 1. For the second query ($L2$), however, we have $C = (Age < 12)$. Notice that $C \wedge (\delta_{Patients}^- \vee \delta_{Patients}^+) = (Age < 12) \wedge ((15 \leq Age < 30) \vee false)$ is not satisfiable. This means that, regardless of the underlying database instance, the query result could not have been affected by the misconfiguration. Thus, we know that $L2$ is not suspicious, and we can prune it.

[3]C is a standard propositional formula consisting of atoms of the form $attr \Theta constant$ and $attr1 \Theta attr2$ connected by logical operations ($\wedge$, $\vee$, $\neg$), where $\Theta \in \{=, >, <, \leq, \geq, \neq\}$.

[4]Of course, the satisfiability problem is NP-complete [9]. However, the size of the input to the satisfiability problem here grows with the complexity of the conditions (δ_S^-, δ_S^+), not the data. Thus, we expect that in practice, where selection conditions are usually simple, this will perform reasonably well.

4.3 Handling Data Updates

Unfortunately, the static pruning described in the last section is only valid in the case where there are no data modifications or updates. When there are updates, we must also consider the possibility that the underlying database instances may have been modified as a result of the policy misconfiguration. This challenge is illustrated by the following example.

EXAMPLE 4.2. *Consider again the operation log in Figure 1, and suppose that* $Patients_{old} = (Age < 30)$ *and* $Patients_{new} = (Age < 15)$. *Suppose also that there is no access control condition on the table* $Temp$ *(i.e.,* $Temp_{old} = Temp_{new} = true$*). Logically, the portion of* $Temp$ *that is visible under both policies is unchanged. Indeed, we have* $\delta^-_{Temp} = \delta^+_{Temp} = false$*, so the static pruning condition from the previous section is technically satisfied, for example, for* $L6$*.*

Unfortunately, this does not take into account the occurrence of data modifications. In the example, notice that the rows inserted into $Temp$ *are different under the old and new policies (i.e.,* $Patients_{old}$ *and* $Patients_{new}$*). For this reason, the result of the query in* $L6$ *is actually different in the two cases, and* $L6$ *should be returned by the MR-query!*

To address this problem, we propose to construct *delta tables*, which store the difference between backlog tables S^b and $\widehat{S}^b$. As we replay the operation log, rather than applying updates to a full database copy (like the naive algorithm), our optimized algorithm (Algorithm 2 in Appendix A) will use the delta tables to capture the difference between the updates that occurred when operating under the old policy and the updates that would have occurred under the new policy.

DEFINITION 3 (DELTA TABLES). *Delta tables store the difference between the backlog versions of each table when using* P_{old} *and when using* P_{new}*. Specifically, for table* S *at time* t*:*

- $\Delta^-_S = \sigma_{time \leq t}(S^b - \widehat{S}^b)$ *is the set of rows that get added to the backlog table when operating under the old policy, but not under the new policy.*
- $\Delta^+_S = \sigma_{time \leq t}(\widehat{S}^b - S^b)$ *is the set of rows that get added to the backlog table when operating under the new policy, but not under the old policy.*

Thus, $\sigma_{time \leq t}(\widehat{S}^b) = \sigma_{time \leq t}(S^b) \cup \Delta^+_S - \Delta^-_S$.

Delta expressions and delta tables can be used in combination to develop a pruning condition that is valid in the presence of updates. For simplicity, consider first a logged operation that mentions a single table S. Suppose that the operation has timestamp t, and let C be the selection condition associated with the operation.[5] In this case, the following conditions are sufficient to guarantee that (i) If the operation is a query, it is *not* suspicious, and (ii) If the operation is a data modification, its effects are exactly the same under the old and new policies. Thus, we can safely ignore the operation if:

1. The expressions $C \wedge \delta^+_S$ and $C \wedge \delta^-_S$ are not satisfiable.
2. $\sigma_C(\Delta^+_S) = \sigma_C(\Delta^-_S) = \emptyset$.

[5]Both queries (SELECT statements) and data modification operations (INSERT, UPDATE, and DELETE) can include selection conditions. For example, in our sample operation log, log record $L3$ includes the selection condition WHERE Name = "Bob." In the case of an SQL statement that contains no explicit selection condition, let $C = true$.

Time	Op	Id	Name	Age	Disease
3	Del	2	Bob	20	Ulcer
3	Ins	2	XXX	20	Ulcer

(a) $\Delta^-_{Patients}$

Time	Op	Id	Name	Age	Disease

(b) $\Delta^+_{Patients}$

Time	Op	Id	A	B	C
5	Ins	2	XXX	20	Ulcer

(c) Δ^-_{Temp}

Time	Op	Id	A	B	C

(d) Δ^+_{Temp}

Figure 6: Delta-Tables for Running Example

The intuition is mostly analogous to the query-only case. Condition (1) allows us to determine statically that the logged operation relies only upon the portion of the data that is logically visible under both the old and new policies. Condition (2) additionally guarantees that none of the data selected by the operation has been altered (i.e., updated in a different way as a result of the old policy vs. the new policy). Of course, unlike the condition described in the previous section, this pruning condition is not completely static, since condition (2) depends on the specific data in the delta tables. However, this only requires re-processing the logged selection condition on the data that has been changed (i.e., the delta tables), rather than re-processing the query on the full database. For small misconfigurations, we observe that the sizes of delta tables are often small when compared to the size of the full database.

This intuition is formalized, and generalized to operations involving multiple tables, via the following theorem. (The proof can be found in Appendix C.)

THEOREM 2. *Consider an operation in the log with selection condition* C *and that references relations* $S_1, ..., S_n$*. Without loss of generality, let* C *be expressed in conjunctive normal form (CNF); that is,* C *is a conjunction of clauses, each of which is a disjunction of literals. The operation can be pruned if both of the following conditions are satisfied:*

1. *The expression* $C \wedge (\delta^-_{S_1} \vee \delta^+_{S_1} \vee ... \vee \delta^-_{S_n} \vee \delta^+_{S_n})$ *is not satisfiable.*
2. *Let* C_{S_i} *be the conjunction of clauses in* C *that mention* only *attributes in* S_i*. (If no such clauses exist, let* $C_{S_i} = true$*.) For each relation* S_i*,* $\sigma_{C_{S_i}}(\Delta^+_{S_i}) = \sigma_{C_{S_i}}(\Delta^-_{S_i}) = \emptyset$.

EXAMPLE 4.3. *To illustrate, consider the operation log shown in Figure 1. Consider also the backlog table* $Patients^b$ *shown in Figure 3. Suppose again that* $Patients_{old} = (Age < 30)$*,* $Patients_{new} = (Age < 15)$*, and* $Temp_{old} = Temp_{new} = true$*. Thus,* $\delta^-_{Patients} = (15 \leq Age < 30)$*,* $\delta^+_{Patients} = false$*,* $\delta^-_{Temp} = false$*, and* $\delta^+_{Temp} = false$*.*

The alternate backlog database $\widehat{DB}^b$*, which would be constructed if the new policy was in place, is shown in Figure 3. The delta tables* $\Delta^-_{Patients}$*,* $\Delta^+_{Patients}$*,* Δ^-_{Temp}*, and* Δ^+_{Temp} *for this example are shown in Figure 6. Using the above conditions,* $L2$ *can be pruned. However, the remaining operations (*$L1, L3, L4, L5, L6$*) have to be at least partially re-executed, as we will describe in the next section.*

4.4 Simultaneous and Partial Re-Execution

If the pruning strategies described in the previous subsections fail, it is necessary to at least partially re-execute the remaining

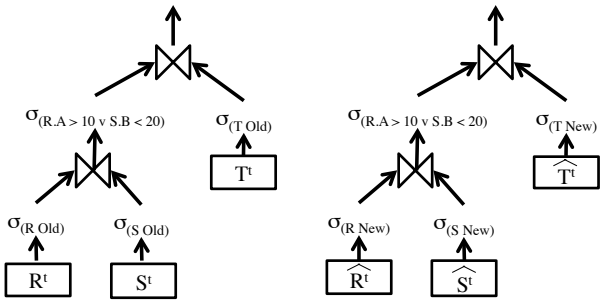

Figure 7: Query Plans For The Old and New Policies

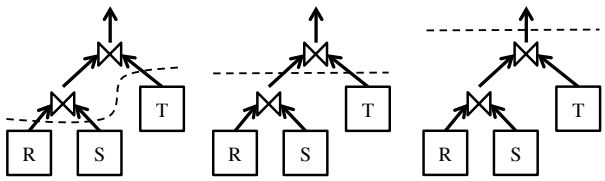

Figure 8: Possible Query Plan Cuts

logged operations. In this section, we introduce two additional optimizations. The first is based on *partial re-execution*; we can sometimes determine mid-execution that an operation can be ignored. The second is based on *simultaneous re-execution*; rather than executing each logged query twice (once with the old data and policy, and once with the new data and policy), we can often save computation by combining the two into a single query plan.

4.4.1 Partial Re-Execution

Our first optimization is based on the observation that it may not be necessary to completely re-execute every un-pruned logged operation; in some cases, we can determine mid-execution that the operation can be ignored / pruned.

Our basic approach is illustrated with a simple example. Consider the following SQL query, which was logged at time t:

```
SELECT *
FROM R, S, T
WHERE (R.A > 10 OR S.B < 20)
AND R.ID = S.ID AND S.ID = T.ID
```

Two sample plans for this query (under the old and new policies) are shown in Figure 7. For any time t, we can compute static snapshot S^t from the backlog S^b and $\widehat{S}^t$ from $\widehat{S}^b = S^b \cup \Delta_S^+ - \Delta_S^-$.

Extending the intuition from the previous section, we can safely ignore this query if we can conclude that the two plans produce the same result. In the general case, we can establish this by identifying a *cut* in the query plan such that the intermediate results of both queries at every point in the cut are equivalent. Figure 8 shows three possible cuts for the example plan.

One way to check whether a cut exists is to evaluate both queries in a "side-by-side" manner using the backlog database and delta tables. This approach re-evaluates both queries (old and new) from the bottom up; after evaluating each operator, it checks whether the results are the same in both cases. If a cut is found, there is no need to continue executing the queries. If no cut is found, the process continues until both queries are completely re-executed. A discussion of implementation tradeoffs is in Appendix E.

Of course, the weakness of this approach is that each operation must be run twice (once on the old data and policy, and once on the new data and policy). For this reason, we introduce one more optimization that allows us to evaluate both queries simultaneously.

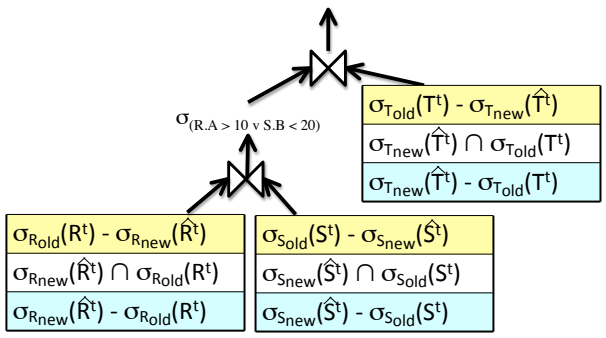

Figure 9: Combined Query Plan. Tuples flagged "New" are shown in blue (bottom third of each table), "Old" in yellow (top third), and "Unchanged" in white (middle third).

4.4.2 Simultaneous Re-Execution

One way to test if a cut exists is to evaluate both queries (old and new) in a "side-by-side" manner, using the backlog database and delta tables. However, in the worst case, this process wastes a lot of work. Notice, for example, in Figure 7, if there is significant overlap between R^t and $\widehat{R}^t$, and between S^t and $\widehat{S}^t$, then the two queries are joining many of the same tuples.

As a more efficient alternative, we instead propose merging the two (old and new) query plans to produce a single plan.[6] Figure 9 shows the merger of the plans in Figure 7. In this figure, the policy-based selection conditions (e.g., S_{old} and S_{new}) are pushed all the way down, and shown as part of the data. For each table S, we combine the rows from $\sigma_{S_{old}}(S^t)$ and $\sigma_{S_{new}}(\widehat{S}^t)$, and we add a flag to each row to indicate where it came from. "New" tuples are those that are emitted only under the new plan. "Old" tuples are emitted only under the old plan. "Unchanged" tuples are emitted under both plans. During query re-execution, these flags are created dynamically and propagated through the plan to ensure result correctness. More information on creating and propagating flags through the query plan can be found in Appendix D.

When evaluating the combined query plan, we can conclude that the query was unaffected by the misconfiguration if there exists a cut in the plan such that the intermediate results along all points in the cut contain no tuples flagged as "Old" or "New."

4.5 Putting it All Together

Using the optimizations that we have described so far, we are now ready to describe our general algorithm for processing MR-queries, which addresses many of the inefficiencies of the naive algorithm. The algorithm begins by creating, for each table S^b in DB^b, tables Δ_S^- and Δ_S^+, which are initially empty. Then, starting from t_1, it replays the operation log forward. For each logged operation, it first checks whether it is possible to prune the operation using the criterion in Theorem 2. If the pruning condition fails, then the algorithm must (at least partially) re-execute the logged operation. More specifically, we use the simultaneous re-execution plan described in Section 4.4.2. During re-execution, if a cut is found (as described in Section 4.4.1), then re-execution is aborted. Otherwise, the re-execution is carried to completion, at which point the results are compared (in the case of logged queries), or the delta-tables are updated (in the case of logged data modification operations). Pseudo-code is provided in Algorithm 2 in Appendix A.

[6] In some ways, this is related to the idea of multi-query optimization [20, 22], the goal of which is to simultaneously evaluate multiple queries, with shared sub-expressions, on a single database. In contrast, in our setting, we need to evaluate the same query on two slightly different databases.

5. EXPERIMENTAL EVALUATION

To evaluate our ideas, we implemented the PolicyReplay system (Figure 14). Given information about a policy misconfiguration, a critical component of the system is the efficient evaluation of MR-queries. We compare the performance of our optimized MR-query processing algorithm (Algorithm 2) and the naive algorithm (Algorithm 1). We utilize numerous simulated data sets and workloads that are tuned by specific parameters (Section 5.2) to determine under what conditions our optimizations improve performance.

5.1 Implementation and Environment

Our prototype system is implemented as a thin Java layer on top of PostgreSQL[7]; it currently supports the static pruning, delta tables, and simultaneous re-execution optimizations, as well as a rudimentary implementation of partial re-execution. We use the Java Constraint Programming Solver (JaCoP)[8] to implement static pruning. SQL parsing is assisted by Zql, a Java SQL Parser[9]. The experiments were executed on a dual core CPU with 2 GB of RAM, running Red Hat Linux.

5.2 Data and Workload

MR-query performance is evaluated with multiple simulated data sets and workloads of logged SQL operations. The underlying database contains tables $T_1, ...T_n$ that are each composed of ten attributes $a_1, ..., a_{10}$. We added an indexed primary key, timestamp attribute and operation-type attribute to each table in order to construct the backlog database. Table T_i consists of R_i rows. The values for attribute a_j are selected from a uniform distribution in the range [1, min($100 \times j^2, R_i$)]; the different ranges are used to vary the selectivity of selection conditions on each attribute. The simulated logged SQL workload contains INSERT, UPDATE, DELETE and SELECT operations as described in Section 4.1. The logged workload is generated by tuning the parameters in Figure 10.

Parameter	Description
Policy Misconfiguration (PM)	The selectivity of the disjunction of the delta expressions on the underlying tables
Operation Selectivity (Sel)	The selectivity of a logged operation on each table
Select to Update Ratio (Ratio, R)	The proportion of all logged operations that are SELECT statements (1.0 implies all SELECTs)
Predicate Attributes (P)	The number of attributes that may be used to create a literal in the selection condition, one of which is the attribute with the policy misconfiguration (P = 1 implies the attribute with the misconfiguration is always chosen for a literal, while P = 8 implies there is a $\frac{1}{8}$ probability that the attribute with the misconfiguration is chosen for a given literal.)
Database Size (Rows)	Number of rows initially in the database
Number of Logged Operations (Ops)	The total number of logged operations

Figure 10: Experimental SQL Workload Parameters

5.3 Results

5.3.1 Static Pruning (No Updates)

Our first set of experiments measures the effectiveness of static pruning in the simple case, where the operation log only contains queries. Figure 11 compares the runtime performance of evaluating an MR-query using the naive approach, which re-executes all logged queries, and the static pruning method, which only re-executes the queries that cannot be pruned. The figure shows performance across a range of policy misconfigurations for a workload with 1% selectivity on a single table. For small misconfigurations, static pruning is able to prune a large proportion of queries, resulting in improved performance. As the size of the misconfiguration grows, more operations must be re-executed. This trend is expected because, with a larger misconfiguration, it is more likely that a logged query's selection condition will intersect the delta expression. Additional pruning results can be found in Appendix F.

5.3.2 Pruning (With Updates)

Next, when the logged workload also contains updates, we measure the benefits of pruning with delta tables (Theorem 2). If a logged operation cannot be pruned, it is re-executed. Consider a workload on a single table where the all the parameters are fixed except for the size of the policy misconfiguration. When the misconfiguration is small (1% and 10%) as shown in Figures 12 and 13, it is more efficient to evaluate the MR-query using pruning with delta tables (Pruning + Delta Tables) than the naive method; the MR-query is evaluated more quickly because fewer operations must be re-executed. Another benefit of the delta tables is that they remove the cost of copying the database prior to evaluating the MR-query; for large databases, this cost can be large.

While pruning reduces the number of operations that must be re-executed, re-executing an operation using delta tables is more costly than re-executing an operation without delta tables because there is an extra cost to construct the table $\widehat{S}$ from S, Δ_S^-, and Δ_S^+. Thus, as the size of the delta tables grows, the re-execution cost increases. We observe that there exists a tradeoff point when it is no longer advisable to use pruning with delta tables, but is more efficient to use the naive method. This tradeoff point is determined by the parameters in the workload such as the size of the misconfiguration, the ratio of selects to updates, and the selectivity of the operations. Additional experimental results are in Appendix F.

5.3.3 Simultaneous Query Evaluation

Finally, we evaluate the effectiveness of simultaneous re-execution. Figures 12 and 13 show the performance of the naive, naive plus simultaneous re-execution (Naive + Simult.), pruning with delta tables (Pruning + Delta Tables) and pruning with delta tables plus simultaneous re-execution (Pruning + Delta + Simult.) methods for different policy misconfigurations on a single table. We find that the performance of the naive method is improved when queries are simultaneously re-executed. The performance of the pruning with simultaneous approach improves slightly for small misconfigurations because only a few queries are re-executed; for larger misconfigurations, the benefits of simultaneous re-execution decrease.

6. RELATED WORK

Most related to our work are two recent proposals for auditing SQL queries [1, 16]. At a high level, this work considers a log that records every SQL query that has been issued to a database. The task of the auditor is the following: given an audit expression that describes some *sensitive* data (e.g., a particular patient's medical record), retrieve every logged query that was *influenced* by this data. Agrawal et al. [1] formalize the idea of influence through the concept of *indispensable tuples*. While this is related to our work, there are clear differences. While one might try to capture the difference between old and new policies (i.e., our delta expressions) using the audit expressions of Agrawal et al. [1], this is problematic because their work does not consider the flow of information via updates. For example, if a record r is copied from one location to another location, say r', the auditor does not understand this. Thus,

[7] http://www.postgresql.org
[8] http://jacop.osolpro.com/
[9] http://www.gibello.com/code/zql/

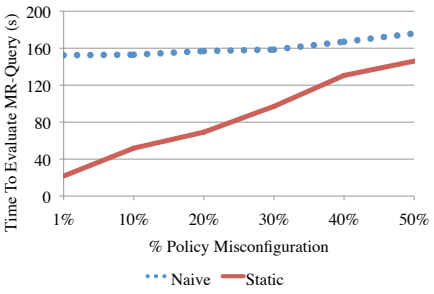

Figure 11: Static Pruning Performance (1% Sel., 250K Rows, P=1, R=1.0)

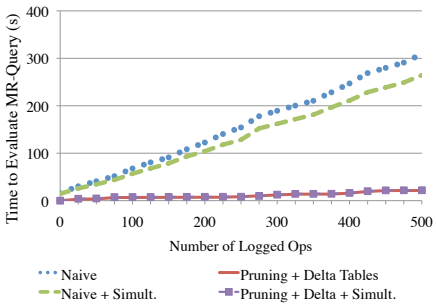

Figure 12: Performance - 1% PM (1% PM, 1% Sel., 250K Rows, P=1, R=0.9)

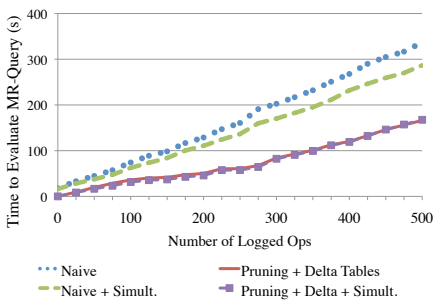

Figure 13: Performance - 10% PM (10% PM, 1% Sel., 250K Rows, P=1, R=0.9)

an audit on r would fail to retrieve SQL queries to which r' was indispensable. Our work addresses this problem cleanly through the definition of MR-queries (Section 3), which conceptually replay the entire database history, including updates, and retrieve all queries whose results have changed.

The problem of tracking how data evolves via updates and views has been studied in the areas of data provenance [6], lineage [8], and annotation [5]. While this is related to our work, as shown in Section 3, it is not clear whether these techniques can address the full scope of our problem.

Lu and Miklau consider auditing a database under data retention restrictions [15]; this work is concerned only with database updates, not auditing queries (i.e., SELECT statements).

Agrawal et al. consider auditing disclosure by relevance ranking [2]: Given a sensitive table that has been "leaked" from an organization, and a log of past SQL queries, determine which queries were most likely to have been the source of the leak.

Recent work has also focused on the problem of recovering from malicious database transactions [4, 7, 13, 14]. At a high level, if a committed transaction is discovered to have been malicious, its effects, and the effects of those transactions that depended on it, must be undone. One important difference between this and our approach is the level at which we reason about data dependencies. In transaction theory, a transaction T_2 is usually said to depend on another transaction T_1 if it reads a data object (e.g., a tuple) written by T_1. When defining the semantics of MR-queries, we are operating at a higher level of abstraction; notably, a query Q may read a tuple that was updated by some other command, but unless that tuple changes the result of Q, it is not considered to have influenced Q. Thus, while one might suggest taking a "transactional" approach to our problem (i.e., by tracking reads and writes), this approach would likely lead to larger result sets for MR-queries. Further, we consider it desirable to define the semantics on MR-queries based only on the *syntax* of the logged operations, rather than the specific plans used to execute them. Notice, for example, that the tuples read by a query vary based on the plan (e.g., scan vs. index lookup), which would affect the MR-query result if we were to take a transactional approach.

7. CONCLUSION

In this paper, we introduced the PolicyReplay framework for responding to database access control misconfigurations. One of the critical components of this framework is the misconfiguration-response (MR) query, which retrieves those queries that may have revealed unauthorized information. The naive algorithm for evaluating MR-queries can be expensive. Thus, we have developed and evaluated a suite of techniques (including pruning, delta tables, partial re-execution, and simultaneous re-execution) for improving the performance of this operation.

8. 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] R. Agrawal, A. Evfimievski, J. Kiernan, and R. Velu. Auditing disclosure by relevance ranking. In *SIGMOD*, 2007.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, 2002.

[4] K. Bai and P. Liu. Data damage tracking quarantine and recovery scheme for mission-critical database systems. In *EDBT*, 2009.

[5] D. Bhagwat, L. Chiticariu, W. chiew Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.

[6] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[7] T. Chiueh and D. Pilania. Design, implementation, and evaluation of a repairable database management system. In *ICDE*, 2005.

[8] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, 2000.

[9] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[10] C. S. Jensen, L. Mark, and N. Roussopoulos. Incremental implementation model for relational databases with transaction time. *IEEE Transactions on Knowledge and Data Engineering*, 1991.

[11] D. Kiely. SQL Server 2008 Security Overview for Database Administrators. SQL Server Technical Article, October 2007.

[12] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, 2004.

[13] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovering from malicious transactions. *Distributed and Parallel Databases*, 2000.

[14] D. Lomet, Z. Vagena, and R. Barga. Recovery from bad user transactions. In *SIGMOD*, 2006.

[15] W. Lu and G. Miklau. Auditing a database under retention restrictions. In *ICDE*, 2009.

[16] R. Motwani, S. Nabar, and D. Thomas. Auditing sql queries. In *ICDE*, 2008.

[17] A. Nanda. Fine-grained auditing for real-world problems. *Oracle Magazine*.

[18] R. Reeder, L. Bauer, L. Cranor, M. Reiter, K. Bacon, K. How, and H. Strong. Expandable grids for visualizing and authoring computer security policies. In *CHI*, 2008.

[19] S. Rizvi, A. Mendelzon, S. Sudershan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[20] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi-query optimization. In *SIGMOD*, 2000.

[21] T. J. Wasserman. DB2 UDB Security Part 5: Understanding the DB2 Audit Facility, March 2006.

[22] Y. Zhong, P. Deshpande, J. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *SIGMOD*, 1998.

APPENDIX

A. ALGORITHM DETAILS

Pseudo-code for the naive algorithm, which is used to help illustrate the semantics of MR-queries, is provided in Algorithm 1.

Algorithm 1 Naive Algorithm for Evaluating an MR-Query

Input: P_{old}, P_{new}, t_1, t_2, operation log, backlog database DB^b
Output: Set of suspicious queries

1: Construct a second backlog database $\widehat{DB}^b$ by creating a new backlog table $\widehat{S}^b$ for each table S^b in DB^b. Copy into $\widehat{S}^b$ every row in S^b with $timestamp \leq t_1$.
2: Let e be the first entry in the operation log such that $e.timestamp \geq t_1$
3: **while** $e.timestamp \leq t_2$ **do**
4: Let $t = e.timestamp$
5: **if** $e.sql$ is a data modification operation **then**
6: Evaluate $e.sql$ using policy P_{new} and $\widehat{DB}^t$. Any data modifications are applied to $\widehat{DB}^b$.
7: **else if** $e.sql$ is a SELECT statement **then**
8: Let $Q = e.sql$
9: Evaluate Q on $\widehat{DB}^t$ (using P_{new}) and also on DB^t (using P_{old}).
10: **if** $Q(P_{new}(\widehat{DB}^t)) \neq Q(P_{old}(DB^t))$ **then**
11: Add e to the MR-query result set
12: $e =$ next entry in the operation log

Pseudo-code for the optimized algorithm, which includes static pruning, delta-tables, simultaneous and partial re-execution, is shown in Algorithm 2. In the pseudo-code, $Prunable(C, delta\ tables)$ returns true if the condition in Theorem 2 is satisfied.

Algorithm 2 Optimized Algorithm for Evaluating an MR-Query

Input: P_{old}, P_{new}, t_1, t_2, operation log, backlog database DB^b
Output: Set of suspicious queries

1: For each table S^b in DB^b, create tables Δ_S^- and Δ_S^+, which are initially empty
2: Let e be the first entry in the operation log such that $e.timestamp \geq t_1$
3: **while** $e.timestamp \leq t_2$ **do**
4: Let $t = e.timestamp$
5: Let C be the selection condition associated with $e.sql$
6: **if** $Prunable(C, delta\ tables)$ **then**
7: skip the operation e
8: **else**
9: For every table S^b in DB^b, let view $\widehat{S}^b = S^b \cup \Delta_S^+ - \Delta_S^-$.
10: Simultaneously re-execute $e.sql$ on $\widehat{DB}^t$ with P_{new} and on DB^t with P_{old}
11: **if** a cut is found **then**
12: skip the operation e
13: **else if** $e.sql$ is a SELECT statement **then**
14: Add e to the MR-query result set
15: **else if** $e.sql$ is a data modification operation **then**
16: Suppose that during the original execution, $e.sql$ added tuple set T to S^b, but now $e.sql$ adds T' to S^b. Update the delta-tables accordingly: $\Delta_S^+ = \Delta_S^+ \cup (T' - T)$; $\Delta_S^- = \Delta_S^- \cup (T - T')$
17: $e =$ next entry in the operation log

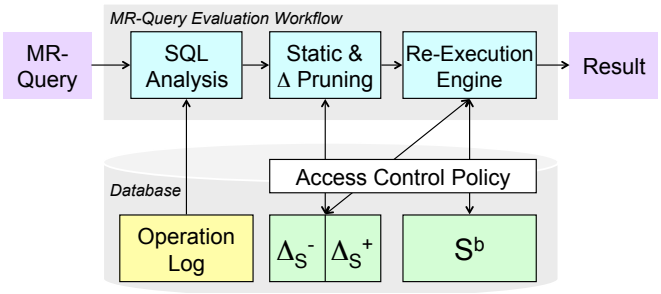

Figure 14: MR-Query Processor Components

B. CONSTRUCTING BREACH REPORTS

Misconfiguration-response queries are at the core of a broader framework for reporting data breaches. In this section, we provide a brief overview of how such reports can be constructed, and what information they can contain.

At a high level, there are two different kinds of reporting, with somewhat different goals and requirements:

- **Organizational Reporting:** In the basic setting, the organization that has experienced the breach must construct a single report summarizing the entire event. In the case of HITECH, for example, a covered entity (e.g., hospital) needs to compile a report to be sent to the regulatory government agency (in this case, the office of the Secretary of Health and Human Services). Similarly, contractors doing work on behalf of a covered entity must report breaches to the covered entity. For example, if a hospital has outsourced billing to a separate company, and that company experiences a breach, it must send a report to the hospital.
- **Individual Reporting:** In the second case, when an organization has experienced a breach, in addition to compiling a single report summarizing the event, it may also be necessary to notify individuals whose personal information was compromised as part of the breach.

We will focus primarily on organizational reporting, but we will also describe our initial ideas for extensions to individual reporting.

Throughout the main body of the paper, for ease of exposition, we used very simplistic operation logs as examples. In practice, the operation log contains the `timestamp` and `sql` text, but it also contains additional attributes, including but not limited to: the database and/or application `username` of the individual who issued the SQL command, an application identifier `appID`, and sometimes the `purpose` for which the query was issued, or a description of an external data `recipient` [3].

For each suspicious query Q, we can also explain how the query was affected by the misconfiguration. In particular, we can summarize the difference between the result that occurred under the old policy P_{old} and the result that would have occurred if P_{old} had been replaced with P_{new}. Suppose Q was logged at time t. Let Q_1 be the result of evaluating Q on DB^t using P_{old}, and let Q_2 be the result of evaluating Q on $\widehat{DB}^t$ using P_{new}. The difference between the two result sets can be summarized by two tuple sets: $Q_{new} = Q_2 - Q_1$ and $Q_{old} = Q_1 - Q_2$.

Following a misconfiguration, the most detailed organizational breach report contains information about every suspicious query Q in the operation log (i.e., each record in the result of the MR-query). This information includes the attributes from the log (e.g., `timestamp`, `sql`, `username`, `appID`, `purpose`, `recipient`), Q_{old}, and Q_{new}. Of course, we can further restrict and summarize

the information in this detailed report based on the requirements of the regulation. For example, HITECH requires reports to document to whom data was disclosed (which can be explained using `username` or `recipient`) and the type and amount of data disclosed (which can be explained using Q_{new} and Q_{old}).

Individual reports can be constructed in a similar way, but they present several additional challenges. Specifically, we only need to send a report to a user U if her information was involved in the breach, and the report should only explain how U's information was disclosed. In the future, we plan to extend the MR-query using an approach related to SQL GROUP BY. At a high level, the idea is to first divide the data into buckets, based on some attribute (e.g., a separate bucket for each patient's information). Then, the MR-query with GROUP BY (conceptually) processes the MR-query once per bucket, using only the data in that bucket.

C. PROOFS

Proof of Theorem 1: The proof is straightforward. Suppose that Q was logged at t. If there are no updates, and $\sigma_C(P_{old}(DB^t)) = \sigma_C(P_{new}(DB^t))$ (i.e., the result of applying policy P_{old} and selection condition C is the same as applying P_{new} and C), then it is safe to prune Q.

Given the following definitions, we must show that $R_1 = R_2$.

$$\begin{aligned} R_1 &= \sigma_C(\sigma_{S1_{new}}(S_1^t) \times ... \times \sigma_{Sn_{new}}(S_n^t)) \\ R_2 &= \sigma_C(\sigma_{S1_{old}}(S_1^t) \times ... \times \sigma_{Sn_{old}}(S_n^t)) \end{aligned}$$

If $C \wedge (\delta^-_{S_1} \vee \delta^+_{S_1} \vee ... \vee \delta^-_{S_n} \vee \delta^+_{S_n})$ is not satisfiable, this means that none of the following expressions is satisfiable:

$$(C \wedge S1_{old} \wedge \neg S1_{new}), (C \wedge S1_{new} \wedge \neg S1_{old}), ...,$$
$$(C \wedge Sn_{old} \wedge \neg Sn_{new}), (C \wedge Sn_{new} \wedge \neg Sn_{old})$$

Using this, it is easy to show that $R_1 - R_2 = \emptyset$ and $R_2 - R_1 = \emptyset$.

Proof of Theorem 2: Suppose that the operation occurred at time t. If $\sigma_C(P_{old}(DB^t)) = \sigma_C(P_{new}(\widehat{DB}^t))$, then it is safe to prune the operation.

Given the following definitions, we must show that $R_1 = R_2$.

$$\begin{aligned} R_1 &= \sigma_C(\sigma_{S1_{new}}(\widehat{S_1}^t) \times ... \times \sigma_{Sn_{new}}(\widehat{S_n}^t)) \\ R_2 &= \sigma_C(\sigma_{S1_{old}}(S_1^t) \times ... \times \sigma_{Sn_{old}}(S_n^t)) \end{aligned}$$

These can be rewritten as follows (by pushing down selection conjuncts that refer only to a single table):

$$\begin{aligned} R_1 &= \sigma_C(\sigma_{S1_{new}}(\sigma_{C_{S_1}}(\widehat{S_1}^t)) \times ... \times \sigma_{Sn_{new}}(\sigma_{C_{S_n}}(\widehat{S_n}^t))) \\ R_2 &= \sigma_C(\sigma_{S1_{old}}(\sigma_{C_{S_1}}(S_1^t)) \times ... \times \sigma_{Sn_{old}}(\sigma_{C_{S_n}}(S_n^t))) \end{aligned}$$

Condition (2) is sufficient to guarantee that, for $i \in 1..n$,

$$\sigma_{C_{S_i}}(\widehat{S_i}^t) = \sigma_{C_{S_i}}(S_i^t).$$

The reason for this is that no tuples satisfying the condition C_{S_i} have been modified differently as the result of the different policies, since $\sigma_{C_{S_i}}(\Delta^+_{S_i}) = \sigma_{C_{S_i}}(\Delta^-_{S_i}) = \emptyset$.

Finally, per the same argument in the proof or Theorem 1, we can show that $R_1 - R_2 = \emptyset$ and $R_2 - R_1 = \emptyset$.

D. SIMULTANEOUS RE-EXECUTION

D.1 Combined Tables and Flags

When executing a combined query on table S, we retrieve $\sigma_{S_{old}}(S^t) \cup \sigma_{S_{new}}(\widehat{S}^t)$, and we flag the input as follows:

- **"New" Flags:** $\sigma_{S_{new}}(\widehat{S}^t) - \sigma_{S_{old}}(S^t)$
- **"Old" Flags:** $\sigma_{S_{old}}(S^t) - \sigma_{S_{new}}(\widehat{S}^t)$
- **"Unchanged" Flags:** $\sigma_{S_{old}}(S^t) \cap \sigma_{S_{new}}(\widehat{S}^t)$

This can be explained as follows. When accessing a table S, under the old policy and data, we would have retrieved $Old(S) = \sigma_{S_{old}}(S^t)$. Under the new policy, we retrieve $New(S) = \sigma_{S_{new}}(\widehat{S}^t)$. This is shown, for example, in Figure 7.

When evaluating the combined policy, we will take all tuples from $New(S) \cup Old(S)$ as input. We want to flag all tuples in $New(S) - Old(S)$ as "New", tuples in $Old(S) - New(S)$ as "Old", and tuples in $New(S) \cap Old(S)$ as "Unchanged".

During query re-execution, these flags are created dynamically and propagated through the plan according to the following rules (This can be done in SQL, without modification to the DBMS engine):

- **Selection:** If a selection takes a tuple as input, and the tuple passes the selection filter, the output tuple keeps the same flag.
- **Projection:** Similarly, if a projection operator takes a tuple as input, the corresponding output tuple has the same flag.
- **Join:** The challenging operator is join. If a join takes as input two tuples with the same flag (i.e., both "New", both "Old", or both "Unchanged"), the emitted result tuple maintains that same flag. If the join takes as input two tuples such that one tuple is "Old" or "New", and the other "Unchanged", the resulting tuple inherits the "Old" or "New" flag. Finally, if a join takes as input two tuples such that one is "Old" and the other "New," even if the two tuples satisfy the join condition, no result tuple is produced. The reason for the final case is due to the fact that these input tuples are actually part of different conceptual query executions.

D.2 Implementation

The combined tables (and flags) described above can be computed using standard SQL. This implementation is for the class of queries described in Section 4.1.

The following outlines how to create a combined table, and assigns the appropriate flags to each row of the table ($New = 1$, $Old = -1$, $Unchanged = 0$). This is accomplished with SQL by taking the union of S^t with $\widehat{S^t}$; additionally, for each row from S^t, we add a flag of -1, while for each row from $\widehat{S^t}$, we add a flag of 1. We then group by all attributes in the table except the dynamically generated flag. If the sum of the flags in a group is -1, this means that the row only exists under the old policy. If the sum of the flags in a group is 0, this means that the row exists under the new and old policies (-1 + 1 = 0). If the sum of the flags in a group is 1, this means that the row only exists under the new policy.

When executing the combined query, we can propagate flags according to the rules outlined in Appendix D.1 by adding the following constraint for all pairs of tables `R` and `S` referenced in the from clause of the query:

```
((S.flag != (-1) * R.flag) OR
  (S.flag = 0 AND R.flag = 0))
```

Finally, if a query result contains only tuples with flags = 0 (i.e., unchanged rows), then the query is not suspicious. More generally, if the result of the query under the new policy is the same as the result under the old policy (even if the result contains flagged rows), then the query is not suspicious. For SPJ queries, we can check if the results under the two policies are the same by grouping by all attributes and summing the value of the flag; thus, if the old policy and new policy produce the same result for a given row, the resulting sum for the row will be zero.

E. PARTIAL RE-EXECUTION

We can safely ignore a query if we can conclude that the query plans from the old and new policy produce the same result. In the general case, we can establish this by identifying a *cut* in the query plan such that the intermediate results of both queries at every point in the cut are equivalent. Figure 8 shows three possible cuts for the example plan. The second cut, for example, would require that both of the following conditions be satisfied:

1. $\sigma_{R.A>10 \vee S.B<20}(\sigma_{R_{New}}(\widehat{R}^t) \bowtie \sigma_{S_{New}}(\widehat{S}^t))$
   $= \sigma_{R.A>10 \vee S.B<20}(\sigma_{R_{Old}}(R^t) \bowtie \sigma_{S_{Old}}(S^t))$
2. $\sigma_{T_{New}}(\widehat{T}^t) = \sigma_{T_{Old}}(T^t)$

There are several possible ways that partial re-execution can be implemented. Our current prototype supports a rudimentary version of option (3). However, in this section, we describe alternative implementation strategies, as well as the important design considerations.

1. **Internal Database Filters**: The goal of partial re-execution is to execute the query plan until we can guarantee that the result is not affected by the misconfiguration. One possible implementation strategy is to add filters inside the query processor that examine the propagation of rows through a pipelined query plan. If information from the filters can be used to determine that old and new flagged rows do not pass the filters across a cut in the query plan, then the execution of the query can be stopped. This strategy has the least impact on the query processor, and does not alter query plans. On the other hand, this method may not be practical since the internals of the database engine must be modified.
2. **Materialize and Check Intermediate Results**: For many query plans, the query optimizer chooses to materialize, rather than pipeline, some portions of the query plan. We can leverage these materialized intermediate results to check if there exits a cut in the plan that does not contain an old or new flagged row. If there exits a cut, then the execution of the query can be stopped. If old or new flagged rows do exist, the materialized results can be used as input to the next part of the query plan. The extra cost of this approach is storing the intermediate result, scanning the result for flags, and, in the cases when the result contains old or new flags, reading the intermediate result and sending it to the next part of the query plan. Overall, this approach appears practical only when the optimizer is already materializing certain intermediate query results.
3. **Left-Deep Query Plan Analyzer**: The third implementation strategy for partial re-execution relies on a left-deep query plan, which is commonly generated by modern query optimizers. In a left-deep plan, the input tables are joined one at a time, according to a total order. (For example, in Figure 7, R is joined with S, and then the result is joined with T. Thus, the join order is R, S, T.)

   Briefly, the partial re-execution algorithm for left-deep plans works as follows: We begin with the last table in the join order (In Figure 7, this is T.), and we check whether this table can be pruned according to Theorem 2. If not, we must re-execute the entire query (i.e., the only possible cut is the rightmost cut in Figure 8). If it can be pruned, then we consider the previous table in the join order (in this case, S), and check whether S can be pruned according to Theorem 2. This algorithm continues until it reaches a table that cannot be pruned, at which point we must execute the subquery rooted at that position of the plan. For example, if we find that we can prune T, but not S, then we consider the second cut in Figure 8 by re-executing $R \bowtie S$, and checking the flagged rows in the result. If the result contains no tuples flagged as "Old" or "New," we can stop. Otherwise, we must execute the entire query.

F. ADDITIONAL EXPERIMENTS

F.1 Static Pruning (No Updates)

In addition to measuring the runtime performance of evaluating the MR-query, we also measured the effectiveness of static pruning by counting the number of logged queries that were suspicious, unsuspicious, and pruned. (Recall that the number of pruned queries must be $\leq$ the total number of unsuspicious queries.) Intuitively, as more queries are pruned, performance improves, since fewer queries must be re-executed.

Figure 15 shows the pruning information across a range of policy misconfigurations with a workload that has 1% selectivity on a single table, where the policy misconfiguration is on one attribute. For small misconfigurations (PM=1%), the static pruning is able to prune a large proportion of queries (484 pruned out of 500, in this case). As the misconfiguration gets larger (PM=50%), fewer queries can be pruned; however, we are able to prune all operations that are unsuspicious. This trend is expected because, with a lager misconfiguration, it is more likely that a logged query's selection condition will intersect with the delta expressions.

In order for an operation to be statically pruned, there must exist a literal in the selection condition that contradicts the delta expressions. Thus, when the attribute with the policy misconfiguration is not in the selection condition, it is less likely that a contradiction exists and the operation can be pruned. Figure 16 shows the pruning effectiveness as the number of predicate attributes (P) increases. As the number of predicate attributes increases, fewer operations are pruned. It is important to note that in some cases we are not able to prune all unsuspicious operations using the static pruning condition. In these cases, even though the delta expressions intersect the selection condition (statically), there does not exist a row in the specific database instance that is affected by the misconfiguration.

F.2 Pruning (With Updates)

When updates are considered, in addition to measuring the runtime performance of evaluating the MR-query, we also counted the number of logged operations that were suspicious, unsuspicious, and pruned. Figure 17 shows these statistics when the log contains updates. As expected, for small misconfigurations, we are able to prune a large portion of the operations. As the misconfiguration gets larger, fewer operations are pruned.

While pruning reduces the number of operations that must be re-executed, as mentioned in Section 5.3.2, the cost of re-executing an operation using delta tables is larger than re-executing an operation without delta tables because there is an extra cost to construct the table $\widehat{S}$, which is formed by removing the rows in Δ_S^- from S and appending the rows from Δ_S^+. Thus, as the size of the delta tables grows, the cost of re-executing an operation increases. Figure 18 shows the number of rows in the database after using the naive method, which includes the rows in S and $\widehat{S}$, the number of rows contained in the delta tables, and the total number of rows when the delta tables are used, which includes S, Δ_S^-, Δ_S^+ and rows used to store the results of update operations that cannot be pruned. As expected, for small misconfigurations, the delta tables are small and the extra cost of constructing $\widehat{S}$ is small (see Figures 12 and 13). In contrast, for larger misconfigurations (50%), the delta tables are large, which slows down the performance of the pruning method

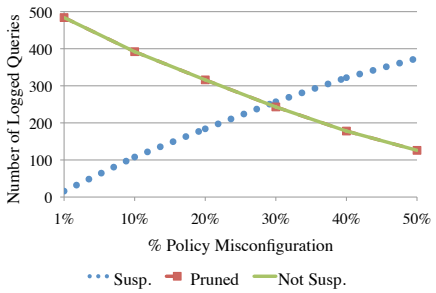

Figure 15: Static Pruning Statistics (1% Sel., 250K Rows, P=1)

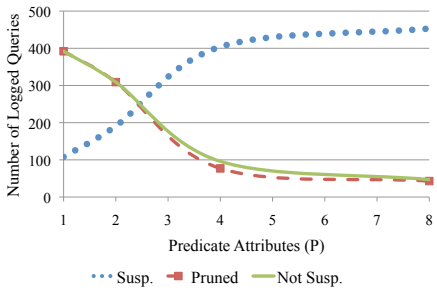

Figure 16: Static Pruning Statistics (1% Sel., 250K Rows, 10% PM)

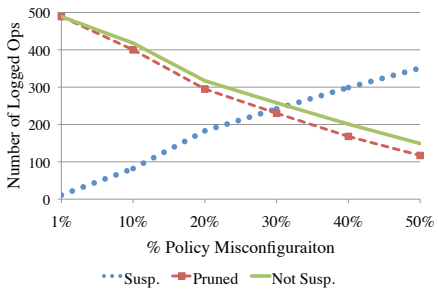

Figure 17: Pruning With Updates (1% Sel., 250K Rows, P=1, R=0.9)

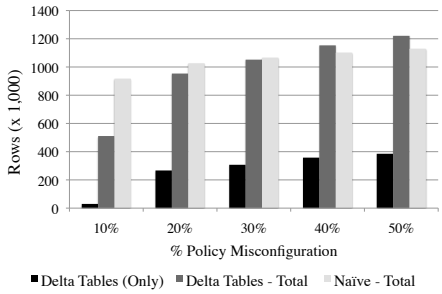

Figure 18: Database Size (1% Sel., 250K Rows, P=1, R=0.9)

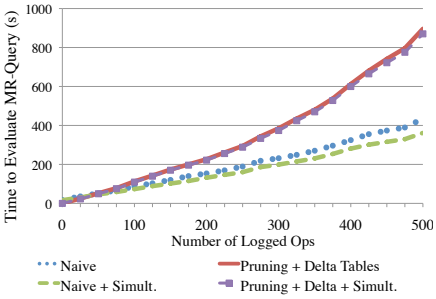

Figure 19: Performance - 50% PM (50%PM, 1%Sel., 250K Rows,P=1,0.9 Ratio)

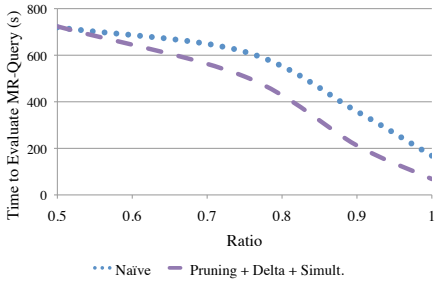

Figure 20: Ratio Impacts Performance (10% PM, 0.1% Sel., 250K Rows, P=1)

(Figure 19); in this case, it is more efficient to evaluate the MR-query using the naive method.

The select to update ratio can impact the performance of evaluating an MR-query. Figure 20 shows the time to evaluate an MR-query for the naive method, and the pruning with delta tables plus simultaneous re-execution method for a range of ratios. When the log contains many updates (i.e., has a low ratio), the performance of both methods worsens due to the higher cost of writing to the database than reading. Additionally, a larger number of updates typically will increase the size of the delta tables, resulting in fewer operations being pruned and increasing the time to re-execute a query. For a ratio of 0.5, the naive method and pruning method have similar performance for the specified set of parameters. As the ratio gets larger (fewer updates), the pruning method outperforms the naive approach.

The challenge is to determine, before the MR-query is evaluated, which method (naive plus simultaneous re-execution, or pruning with delta tables plus simultaneous re-execution) is the most efficient given the workload parameters. We believe this decision can be made by developing a cost-based optimizer. One promising approach would use the database's query optimizer to estimate the cost of re-executing each logged operation using each of the two methods; given the total estimated cost, the appropriate method can be chosen.

F.3 Partial Re-Execution

We observed that pruning with delta tables as described by Theorem 2 captures many of the cases where partial re-execution is applicable. Intuitively, pruning with delta tables is a form of partial re-execution where the query plan is cut across the leaf selection conditions. However, we did find a few cases where partial re-execution did provide additional performance benefits. Particularly, partial re-execution is beneficial when the selection condition contains a clause that is composed of a disjunction of literals that references multiple tables. In some cases, new or old rows may not be filtered by the conjunction of clauses that only refer to a single table, but are then filtered out by the clause that references multiple tables. In such a case, partial re-execution can improve re-execution performance by only re-executing a subset of the query plan.

We used the rudimentary Left-Deep Query Plan Analyzer as described in Section E to determine what cuts in the query plan should be tested. We crafted a workload of an update operation and a query that would benefit from partial re-execution. The query had three tables R, S and T, and a selectivity of 1% for each table. The query plan analyzer determined that the table T could be removed so that the sub-query of R and S only needed to be evaluated. For this example, we found that partial re-execution reduced the run time of the query by 28%.