On Scaling Up Sensitive Data Auditing

Yupeng Fu
UC San Diego
yupeng.fu@gmail.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

Ravishankar Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

This paper studies the following problem: given (1) a query and (2) a set of sensitive records, find the subset of records "accessed" by the query. The notion of a query accessing a single record is adopted from prior work. There are several scenarios where the number of sensitive records is large (in the millions.) The novel challenge addressed in this work is to develop a general-purpose solution for complex SQL that scales in the number of sensitive records. We propose efficient techniques that improves upon straightforward alternatives by orders of magnitude. Our empirical evaluation over the TPC-H benchmark data illustrates the benefits of our techniques.

1. INTRODUCTION

Databases are used to store sensitive information such as health records, employee records and customer data motivating the need for developing a security infrastructure as part of a DBMS. The security infrastructure of database systems consists of mechanisms such as access control and encryption whose goal is to *prevent* security breaches. However, there is an increasing recognition in the security community that since it is impossible to prevent security breaches perfectly, we must also develop mechanisms that *cope* with breaches retroactively [15, 27]. Despite the availability of preventive mechanisms in modern DBMSs, data breaches are common [4]. One well-known recent example is the Swiss bank breach [25] of 2010 where an insider in a Swiss bank sold personal information about German customers to the German government. One of the most common reasons for data breaches is insider attacks where the insider gets information about sensitive data by running SQL queries and examining their results.

Therefore, an important part of the security infrastructure of a database system is an auditing system that monitors various operations during production and can be used to aposteriori investigate security breaches. All major database vendors provide the ability to monitor various operations such as the query and update statements issued in production and generate an audit log. Tools to analyze the audit log offline are limited to answer questions about schema objects, such as finding queries that accessed sensitive columns. However, in order to investigate data breaches, we need sophisticated tools that join the audit log with the *data* in the database in order to identify parts of the data that were breached and the users responsible for the breach. We refer to the above as *data auditing*. Traditional mechanisms such as triggers can be used to trace updates to sensitive data. However, data breaches happen primarily through data *reads* for which triggers are inapplicable.

Our data auditing framework is based on the fundamental notion of a query "accessing" (or "referencing") sensitive data which we define formally later in this section. We use the above notion to perform the following operation: given as input (1) an audit log containing a sequence of statements, (2) a set of sensitive records, find for each statement, the subset of sensitive records accessed by it. For example, Figure 1 shows a database with bank customers and an audit log. The edges in the figure show the records accessed by each statement. By modeling the audit log as a table, we can think of the above operation as performing a join between two tables — one containing statements and one containing records in the database. We therefore model the above operation as an *operator* that we call the *References* operator. By composing the `References` operator with standard relational operations such as filters, grouping, aggregation and sorting, we obtain a rich framework for analyzing the audit log. We could for example, subset the audit log and the database using various predicates before invoking the operator and run queries on its result.

In the event of a data breach as in the Swiss bank breach, two of the main goals of auditing are to identify (1) the user that initiated the breach, and (2) the subset of customers whose data got breached. In this setting, we could use the `References` operator as follows. In order to identify the user that initiated the data breach, we could run the following queries shown pictorially in Figure 1: (1) find users who accessed more than a minimum number of sensitive records, (2) order users by the number of sensitive records accessed, (3) if we know a time range during which the breach happened and a superset of customers whose data likely got breached, say those with a minimum account balance, then we add predicates to focus the analysis. Now, suppose that user `JoeAdmin` is suspected of being the insider that initiated the data breach. We can find the customers whose data likely got breached by finding records accessed by the query issued by `JoeAdmin` (also shown in Figure 1.) In our example, `JoeAdmin` has accessed the records of `Alice` but not `Chen` or `Bob`.

The goal of this paper is to develop the above data auditing framework. Offline audit log analysis involves going back in time to older states of the database and is therefore expensive; like security in general, retroactive security comes with a cost. However, it is possible to mitigate the cost of going back in time in a variety of ways. One way is to initiate periodic audits pro-actively

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 5
Copyright 2013 VLDB Endowment 2150-8097/13/03... $ 10.00.

Customer

Name	*Balance*
Alice	150k
Bob	80k
Chen	130k

Audit Log

User	*Query*	*Time*
JoeAdmin	Select * From Customer Where Balance > 140k	4/15/2012
JaneAdmin	Select count(*) From Customer Where exists (Select * From Customer Where Name = 'Chen' and Balance > 100k)	4/15/2012

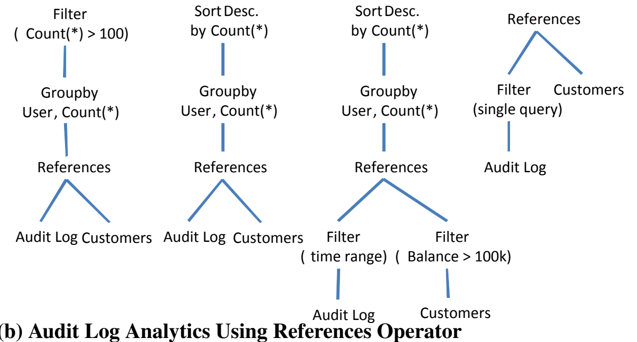


Figure 1: (a) Sample Customer Database And Audit Log. (b) Audit Log Analytics Using References Operator

to check for suspicious activity. In this way, we can limit the extent to which we need to go back in time. Alternately, we could maintain periodic backups, or use standard mechanisms such as change tracking [18] and database flashback [21] to reduce the cost of time-travel. On the other hand, in order for our data auditing framework to be feasible, we need an efficient implementation for the `References` operator; in particular given a *single* query and a set of sensitive records, find the subset of sensitive records accessed by it.

The above problem is the focus of this paper. There are two parts to the problem. One is to design a semantics for data access with meaningful privacy guarantees and the second is to design an efficient implementation for full SQL.

1.1 Auditing Semantics

Simplistic definitions such as checking the query output for the record are adequate only for restricted classes of queries. Accordingly, prior work has studied various definitions. Two fundamentally different approaches have been adopted. One is data instance-independent including the notion of *perfect privacy* [19, 16] and slightly weaker alternatives such as *weak syntactic suspiciousness* [20]. The other is instance-dependent [1, 14], where a query is said to access a sensitive record if deleting the record changes its result.

Among the various semantics proposed, there is a tradeoff between security and feasibility of implementation for full SQL. It is known that instance-independent approaches yield stronger security guarantees than instance-dependent ones. However, while efficient auditing techniques have been developed for select-project-join (SPJ) queries [16], previous work [14] has shown that testing the notion of a *critical* tuple [19] that underlies all previously proposed instance independent definitions is undecidable for complex SQL that includes subqueries.

On the other hand, it is possible to implement the above instance-dependent definition for a complex query using two query executions — we run the original query and a modified version that is rewritten to exclude the sensitive record, and check if their results are equal.

In this paper, we present an auditing system for full SQL for the above instance-dependent semantics; ours is among the first pieces of work focusing on full SQL. While our semantics is not perfect from a security point of view (for example, it leads to *negative disclosures* as we will discuss in Section 3), previous work [14] has formally characterized the weaker guarantee that it does yield, that we summarize in Section 3. Developing auditing semantics with stronger guarantees (whether instance-dependent or instance-independent) than ours and that can also be efficiently audited for complex SQL is an interesting open problem.

1.2 Efficient Evaluation

Prior work has also developed efficient techniques for auditing but only for the special case of a *single* sensitive record. The main motivation for the above special case is the health-care domain where the United States Health Insurance Portability and Accountability Act (HIPAA) requires every health care provider to find every entity to whom a given individual's information has been revealed. In contrast, in the general auditing framework introduced above we have as input a large number of sensitive records.

1.3 Challenge And Contributions

A general-purpose baseline algorithm for the problem we are studying involves two executions *per sensitive record*. The number of sensitive records could be linear in the *data* size — in the bank scenario for example, every customer's record is sensitive and the number of customers could be in the millions. Hence, the baseline algorithm is prohibitively expensive. The main contribution of this paper is to develop efficient techniques to scale data auditing with the number of sensitive records. While the data auditing problem is reminiscent of data provenance, the state of the art techniques in data provenance are not applicable for our definition [10] (see Section 2 for a detailed discussion).

We now briefly discuss the main insight in our solution. The baseline algorithm described above requires the original query to be executed over different subsets of the database including the original database and one per sensitive record that excludes the record. We note that a significant part of the database is common across these executions. We therefore formulate a multi-instance query processing problem (Section 4) where we are required to compute the result of a given query over a large number of input database instances. We develop techniques to compute the results of the query over different instances in a single pass by modifying every relational operator to produce results over multiple instances (Section 5). Our key innovations are to represent multiple instances succinctly using *compressed* annotations and develop techniques to perform relational operations directly over compressed annotations without decompressing them — the idea of compression is the key distinction between our work and prior work on annotation propagation [3, 6, 12]. We show formally and empirically the importance of compression in making our techniques efficient. We also report the result of a detailed empirical evaluation (Section 6) using the queries in the TPC-H benchmark. We find that over the 10GB version of the TPC-H database, if we perform auditing treating each of the 1.5 million customers as sensitive, using our techniques has roughly the same cost as the straightforward algorithm run over 5 customers in the *worst* case.

2. RELATED WORK

Section 1 discusses the prior work in data auditing. In this section, we discuss other related work. A body of work that is related to ours is the work on data provenance and causality [5, 17, 11, 13]. The notion of an indispensable record that we use is similar to the notion of a counterfactual record studied in data provenance [17]. Prior work on data provenance has studied various definitions of provenance which have been categorized in recent work by Glavic et al. [9, 10]. One of the main points of classification is whether the data provenance definition is sensitive to the execution plan used for the query. Since data auditing focuses on the information that is revealed about a database by running a query and examining its output, a plan-insensitive definition is more desirable. Execution plans are influenced by factors extraneous to the logical properties of the query such as the physical design of the database and the available statistics. Hence, all data auditing definitions proposed in prior work and analyzed from a privacy perspective are plan-insensitive, including ours. The state of the art practical technique for computing provenance at scale is recent work by Glavic et al. [8] that proposes an implementation that works for TPC-H queries. However, the above work is for a plan-sensitive notion of provenance. As acknowledged by Glavic et al. [10], for a plan-insensitive notion such as ours, there is no prior work that has been demonstrated to work at scale for complex queries. Further, the techniques for computing plan-sensitive provenance are based on rewriting the input SQL query whereas ours is based on modifying query execution.

Our approach of tuple annotations is similar to previous proposals for tuple annotations [3, 6, 12]; in particular our logic for propagating annotations is similar to prior work [12, 2]. Our main innovation is to compress our annotations and perform all operations directly on the compressed annotations. As we will show, compression is crucial in letting our techniques scale. Our multi-instance query evaluation problem is similar to the problem of query evaluation over probabilistic databases [24]. As with data provenance, no techniques for efficient evaluation are known for probabilistic databases for complex queries. We note that our technique does not solve the generic problem of query evaluation over probabilistic databases — we are primarily concerned with the set of worlds that are induced by an audit expression where the number of worlds is linear in the data size and not exponential in the data size as is the case with probabilistic databases.

Finally, we note that the problem of multi-instance query evaluation is a dual of multi-query optimization [7]. In multi-query optimization, the database instance is fixed and the goal is to answer multiple queries. In our problem, the query is fixed and the goal is to find its result over multiple instances. Furthermore, the scale involved is different — the state of the art in multi-query optimization [7] scales up to thousands of queries, whereas we have to consider millions of database instances. On the other hand, our database instances have a large overlap which is what we exploit to improve the query evaluation efficiency.

3. AUDITING SEMANTICS

In this section, we introduce our auditing semantics. We begin with preliminaries including the overall auditing infrastructure and the notion of an *audit expression* used to specify sensitive records. We then discuss the auditing semantics, its privacy guarantee and introduce the `References` operator.

3.1 Preliminaries

3.1.1 Auditing Infrastructure

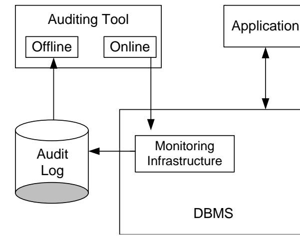


Figure 2: Auditing Infrastructure

Our auditing system consists of two components — an *online* component and an *offline* component, shown in Figure 2. The online component is used in production to log the query and update statements issued to the database system. Along with each query/update statement, we also log the corresponding user id. The audit trail is logged in a separate, secure and tamper proof log which is used to perform auditing in the offline component. When we are analyzing a statement in the audit trail offline, we need to reconstruct the state of the database to when the statement was originally run. We accomplish the above using standard ways such as change tracking [18] and database flashback [21]. Finally, for ease of exposition, in the rest of the paper we focus only on auditing queries (on a given database instance) while noting that it is possible to extend our techniques to handle updates.

3.1.2 Audit Expression

We specify sensitive records in the form of a view that we refer to as an *audit expression*. In this paper, we consider single-table views of the form:

```
select <sensitive columns>
from T where <predicate>
```

We think of each record in the result of the audit expression as being sensitive.

EXAMPLE 3.1. *Consider the database in Figure 1. The following expression specifies that the record of every customer with a balance greater than 100k is sensitive.*

```
select * from Customer where Balance > 100k
```

The above expression specifies that the records corresponding to Alice and Chen are sensitive.

We note that our focus on single-table views is for ease of exposition. It is possible to extend our techniques to cover a larger class of audit expressions where the sensitive data (1) is obtained by joining more than one table, and (2) spans more than one record.

3.2 Data Access

The basis for all data auditing semantics is to define what it means for a query to have *accessed* a particular record. As discussed in Section 1, we adopt a previously proposed definition.

DEFINITION 3.2. *[1] Given a database instance D and a query Q, a record r in the database is said to be* indispensable *if deleting r from D changes the result of Q.* □

EXAMPLE 3.3. *In Figure 1, we can see that the records corresponding to* `Alice` *and* `Chen` *are indispensable to the queries issued by* `JoeAdmin` *and* `JaneAdmin` *respectively.* □

We say that a query Q accesses a set of columns $\bar{A}$ in a relation if there is no equivalent query that excludes $\bar{A}$ — the intuition is that since every column in $\bar{A}$ is sensitive, Q is "safe" only if it uses none of them. We combine the above definition with Definition 3.2 to obtain the following:

DEFINITION 3.4. *Given a database instance D, a query Q, a audit expression V, a record r in the output of V is said to be* accessed *by Q if: (1) Q accesses the sensitive columns in the definition of V and (2) the base record underlying r is indispensable to Q.* □

As discussed in Section 1, it is possible to check indispensability using two query executions (it is possible to do better for simple queries such as select-project-join queries.)

3.3 Privacy Guarantees

Prior work [14] has analyzed the guarantees yielded by our definition above which we briefly describe below. We introduce the privacy guarantee by beginning with an example that illustrates the limitation of our definition.

EXAMPLE 3.5. *[14] Suppose the customer table in the TPC-H database has a credit rating attribute. Suppose that in the current instance of the database, customer* John Doe *has a credit rating of 700. Consider the following queries, Q1:*

```
select sum(CreditRating - 700) from customer
```

and Q2:

```
select sum(CreditRating - 700)
from customer where c_custname <> John Doe
```

By checking if the results of the two queries are equal, an adversary can learn that John Does credit rating is 700. However, the tuple corresponding to John Doe is not accessed by either Q1 or Q2. Thus our auditing semantics would fail to detect the above attack. □

The attack in Example 3.5 essentially requires knowing the credit rating value apriori — if we change the credit rating of John Doe to say 600, query Q1 accesses the corresponding tuple and is therefore flagged as unsafe. Thus, if the adversary does not know the value of John Does credit rating upfront, then by issuing queries Q1 and Q2, he is taking a risk of being detected by the audit system.

Similarly, it is known that the definition above leads to negative disclosures — for example, if we find all customers whose credit rating is equal to 600 and John Doe is not a part of the result, then the adversary learns that John Doe's credit rating is *not* 600, whereas by the above definition the query is safe. However, in the above attack, the adversary presumably does not know apriori whether John Doe's credit rating is 600 and is therefore taking a risk, for if John Doe's credit rating is 600, then the query would no longer be safe. The above intuition is used to develop the notion of *risk-free* attacks [14] where it is shown that under our auditing semantics, no attack is risk-free.

3.4 References Operator

We now introduce the main problem studied in this paper. We call the operation that performs data auditing the `References` operator which we now define.

DEFINITION 3.6. *Given a query Q and a audit expression V, the* `References` *operator returns all records r such that (1) r is in the output of V, and (2) r is accessed by Q.* □

EXAMPLE 3.7. *The edges in Figure 1 show the result of applying the* `References` *operator over each query in the audit log.* □

We note that Section 1 introduces a more general version of the above operator where the input is a set of statements. As a first step, in this paper, we focus on the case of a single query. Henceforth in this paper, we refer to the single query case as the `References` operator.

It is possible to implement the `References` operator efficiently for a restricted class of queries. For example, for select-project-join queries (without duplicate elimination), we can modify the query to return the primary keys of the underlying relations and use the output of the modified query to determine all indispensable records. The records accessed are those that are indispensable and present in the audit expression. Our goal however is to develop efficient techniques for complex queries such as the ones in the TPC-H benchmark. The straightforward algorithm that follows from Definition 3.6 is to iterate over all records in the view V and check for each record whether it is accessed using the query rewriting method discussed in Section 3.2. Since the number of records in V can be very large, the above algorithm is prohibitively expensive.

One straightforward way of optimizing the above algorithm is to avoid iterating over all records in the audit expression by using the predicates in the query as the following example illustrates.

EXAMPLE 3.8. *Consider the audit expression discussed in Example 3.1. Suppose that we have a modified schema where the* `Customer` *table has a* `Nationality` *column. Consider any complex query issued over German customers. Then, it is sufficient to iterate over all records in the following modified view.*

```
select * from Customer where
Balance > 100k and Nationality = 'German'
```

We call the modified expression the *filtered* expression. The details of filtering are straightforward and we omit them from the paper. Algorithm 3.1 illustrates the *baseline* algorithm for the `References` operator. It filters the audit expression using the query predicates and then iterates over all records in the filtered expression. Note that the algorithm performs only one execution of the original query since that part is common across all sensitive records. While filtering can significantly reduce the number of records over which we need to iterate, Algorithm 3.1 can still be prohibitively expensive when the query predicates are not selective as we will show in our empirical evaluation in Section 6. Therefore, we seek a more efficient implementation of the `References` operator. This is the focus of the rest of the paper.

Algorithm 3.1 BASELINE ALGORITHM

Input: Query Q, Database D, audit expression V
Output: Set of records in V that are accessed by Q
1: Compute the result of $Q(D)$
2: $Accessed$ = $null$
3: Compute the filtered expression
4: For each record r in the filtered expression
5: If the results of $Q(D)$ and $Q(D - r)$ differ
6: $Accessed$ = $Accessed \cup \{r\}$
7: End For
8: Return $Accessed$

4. QUERY EVALUATION OVER MULTIPLE INSTANCES

Our goal is to develop techniques that significantly improve upon the baseline approach for complex queries. The baseline approach requires the same query to be run over different database instances obtained by deleting sensitive records from the original database. Our goal is to compute the result of the query over all instances in a single pass. This section formalizes the multi-instance query evaluation problem.

C_Custkey	C_FirstName	C_LastName	C_Acctbal	
1	Joe	Frank	150k	→ D1 = D -r1
2	Steve	Hanks	110k	→ D2 = D -r2
3	Joe	Baker	110k	→ D3 = D -r3
4	Tom	Hill	70k	D0 = D

Figure 3: Illustrating Worlds

4.1 Overview Of References Operator Algorithm

Let the records in the audit expression be $r_1, \ldots, r_{n-1}$. Given a query Q, we can break down the overall implementation of the `References` operator into two parts:

1. *Query Evaluation on n Databases*: We compute the results of Q on each of the instances $D_0 = D$, $D_1 = D - r_1, \ldots, D_{n-1} = D - r_{n-1}$. We observe that the n databases have most records in common. Our goal is to perform this part efficiently by sharing computation among the n instances.

2. *Result Computation*: Use the above results to find the records that were accessed by Q.

Most of the paper focuses on the first part above. In the rest of this section, we first develop terminology to formulate the first part above. We then briefly discuss the algorithm for the second part. Section 5 presents an evaluation algorithm for the first part.

4.2 Multiple Worlds

We use R to denote a relation and $Cols(R)$ to denote its columns. A relational schema $\bar{R}$ is a set of relation names $\{R_1, \ldots, R_m\}$. We let D denote a database instance. A *world set* for schema $\bar{R}$ is a set of database instances $\mathcal{D} = \{D_0, \ldots, D_{n-1}\}$ each with schema $\bar{R}$. Each database D_i is called a *world* identified by subscript i. The instance of relation R_j in world i is denoted R_{ij}. We drop the reference to the schema when it is clear from the context. Given a database D, the world-set *induced* by an audit expression V over D, denoted $\mathcal{D}_V$, is the world-set $\{D - t : t \in V\} \cup \{D\}$. We refer to such a world-set as an *induced* world-set.

EXAMPLE 4.1. *Consider a customer database and an audit expression defined as follows.*

```
Select * From Customer Where C_Acctbal > 100k
```

Figure 3 shows an instance of the `Customer` *table. We note that rows r_1, r_2, r_3 belong to the view. Thus, the world-set induced by the above view has four worlds $\{D_0 = D, D_1 = D - r_1, D_2 = D - r_2, D_3 = D - r_3\}$. We note that instances corresponding to relations other than* `Customer` *are the same in all worlds.*

The output of a query Q on a world-set $\mathcal{D} = \{D_0, \ldots, D_{n-1}\}$ is the world-set $Q(\mathcal{D}) = \{Q(D_0), \ldots, Q(D_{n-1})\}$. Our overall goal is to compute the result of a query over a world-set induced by an audit expression V over a database instance D.

4.3 Representing Multiple Worlds

Suppose that we have an induced world-set with n worlds. If we represent the world-set by explicitly enumerating the worlds, the space consumed is $\Omega(n^2)$. Therefore, we seek to represent a world-set more succinctly. We note that the body of work on probabilistic databases [24] has studied various models of representing a world-set. While previous work on probabilistic databases is able to model a larger class of world-sets where the number of worlds can be exponential in the size of any single world, query evaluation techniques have only been developed for restricted classes of queries.

C_Custkey	C_FirstName	C_LastName	C_Acctbal	Worlds
1	Joe	Frank	150k	<0,2,3>
2	Steve	Hanks	110k	<0,1,3>
3	Joe	Baker	110k	<0,1,2>
4	Tom	Hill	70k	<0,1,2,3>

Figure 4: Record Annotations

In contrast, our world-set is more restricted — the number of worlds is the number of records in the audit expression. On the other hand, our goal is to support evaluation of arbitrary queries. We exploit the fact that we only have to represent a smaller number of worlds by representing the world-set by annotating every record *explicitly* with the set of worlds in which the record is present. Since the set of worlds containing a record can be large, we represent the set succinctly by *compressing* it as we will discuss in Section 4.4.

Formally, we represent a world-set $\mathcal{D}$ as follows: to each relation R_j in $\bar{R}$, we add a distinguished set-valued column that we call an *annotation* column, denoted $Worlds$ to yield relation R_j^w called the *annotated* relation corresponding to R_j. We refer to the above schema as the *annotated* schema. We populate an instance of the above annotated schema as follows. For each relation R_j^w, the instance has all records in $\cup_i R_{ij}$. The annotation of record $r \in R_j^w$, denoted $r.Worlds$, has the set of (identifiers of) worlds containing the record, that is the set $\{i : r \in R_{ij}\}$. Formally, the instance has the set of records $\{< r, r.Worlds >: r \in \cup_i R_{ij}, r.Worlds = \{i : r \in R_{ij}\}\}$. The above database instance is called an *annotated* instance denoted D^w. We illustrate through an example.

EXAMPLE 4.2. *Figure 4 shows the representation of the world-set discussed in Example 4.1. The annotations are shown in the* `Worlds` *column. We only show annotations for the* `Customer` *relation since all other relations are the same in all worlds.*

We note that the annotation representation presented above is complete [22] in that *any* world-set can be represented using annotations. Therefore, the above representation is closed under query evaluation for any arbitrary query. The world-set represented by an annotated database instance D^w is called $WorldSet(D^w)$.

4.4 Annotation Compression

Consider the world-set induced by an audit expression V with $n-1$ records. The number of worlds is n and each record in V is present in $n-1$ of them. If we represent the annotation of a record by explicitly listing its elements, the total space consumed by all annotations together is $O(n^2)$. We therefore seek a more succinct set representation. For each record in V, we could consider storing the complement of the set of worlds containing it. The complement is succinct since it has only one world. However, we will see in Section 5 that for queries involving aggregation, the complement based representation can get large for the intermediate records.

We represent small and large sets in a uniformly succinct manner by using *run-length encoding*. For any annotation, we consider its equivalent n-dimensional boolean vector. For a boolean vector v, we refer to its ith dimension as v_i; dimensions range from 0 to $n-1$. We use standard run-length encoding to compress boolean

vectors by processing them in dimension order. The run-length encoding of vector v, denoted $RLE(v)$ is a sequence of *runs* of the form $< b, start, end >$ where b is a boolean value, $start$ is the starting dimension of the run and end, the ending dimension.

The vector equivalent to the annotation of a record in the audit expression has the form $v = <1 \ldots 1 (n_1 \text{ times})\ 0 1 \ldots 1 (n_2 \text{ times})>$ where $n_1 + n_2 = n - 1$. $RLE(v)$ is the following sequence of runs: $\{< 1, 0, n_1 - 1 >, < 0, n_1, n_1 >, < 1, n_1 + 1, n - 1 >\}$ — the total number of runs is 3 even though the original set has $n - 1$ elements. Therefore, the total number of runs over all annotations together is $O(n)$. Similarly, if an intermediate record generated via query evaluation is present only in a small number of worlds, the run-length encoding would again have a correspondingly small number of runs. Furthermore, we will show in Section 5 that by using run-length encoding, we can operate on the compressed annotations directly without decompressing them.

In the rest of the paper, we use the number of runs as a measure of the space consumed by an annotation. When clear from the context, we use v to also denote $RLE(v)$. For a run run, the corresponding boolean run value is denoted $run.b$ (we use similar notation for $start$ and end.) Henceforth in the paper, when we discuss an annotation, we refer to its compressed encoding. In the examples and figures, for ease of exposition, we show an annotation by explicitly listing its elements while noting that the underlying representation uses compression. We are now ready to define the query evaluation problem.

4.5 Problem Statement

As stated above, our overall goal is to compute the result of a query over a world-set induced by an audit expression V over a database instance D. We address the above goal by formulating a more general problem.

Problem Statement: We are given as input a world-set $\mathcal{D} = \{D_0, \ldots, D_{n-1}\}$ in the form of an annotated database instance D^w where the annotations are compressed using run-length encoding. Given query Q, our goal is to compute the annotated relation that represents the world-set $Q(\mathcal{D}) = \{Q(D_0), \ldots, Q(D_{n-1})\}$.

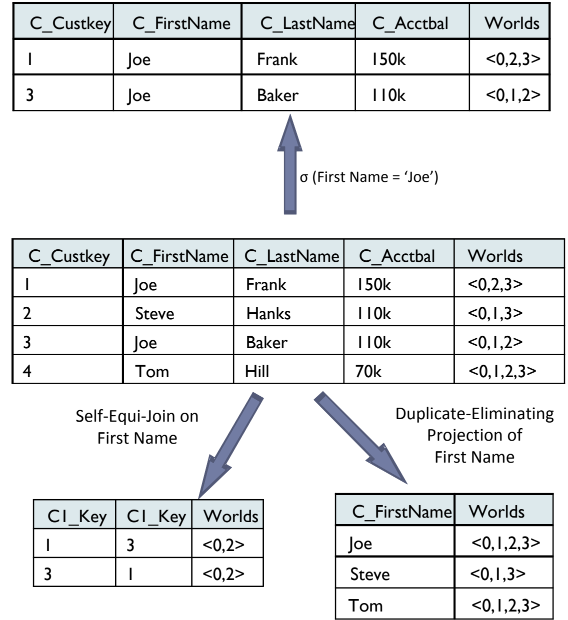
We recall that the straightforward algorithm for query evaluation namely running Q over each world is prohibitively expensive when V has a large number of rows. Our goal is to compute the result efficiently by sharing computation among the worlds. Section 5 addresses the above problem.

4.6 Computing The Final Output

Let the audit expression be V and the database instance be D. We recall that the formal problem we have described above computes the result of $Q(\mathcal{D}_V)$ as an annotated relation. We now briefly discuss the second step of the `References` operator (Section 4.1) that consumes the above result and finds all records in the audit expression referenced. Consider a record $r \in V$. Suppose that the world associated with r, $D - r$, is i and the full database is 0. Record r is referenced if the annotation of some output record contains i but not 0 or vice versa. The above step can be implemented in a straightforward manner using algorithms for operating over compressed annotations described in the next section and we omit the details.

5. QUERY EVALUATION ALGORITHM

We now describe the query evaluation algorithm. We can break down an SQL query into a tree of operators — select, project (duplicate preserving and duplicate eliminating), cross product (cross product and filter together cover joins), groupby-aggregation, anti-joins (covering `NOT EXISTS` subqueries) and top-k. The above operators suffice to represent a rich class of queries including constructs like subqueries; in particular the above operators can represent all queries in the TPC-H benchmark [26]. We modify each operator to consume annotated inputs representing the input world-sets and produce an annotated output representing the result world-set. This section describes the modifications to each operator. There are two parts to our description. We first describe the logic for determining the output annotations and then discuss how we operate on compressed annotations. Since our evaluation algorithm operates in a single pass over the database, it is significantly more efficient than the baseline algorithm discussed earlier.



C_Custkey	C_FirstName	C_LastName	C_Acctbal	Worlds
1	Joe	Frank	150k	<0,2,3>
3	Joe	Baker	110k	<0,1,2>

C_Custkey	C_FirstName	C_LastName	C_Acctbal	Worlds
1	Joe	Frank	150k	<0,2,3>
2	Steve	Hanks	110k	<0,1,3>
3	Joe	Baker	110k	<0,1,2>
4	Tom	Hill	70k	<0,1,2,3>

C1_Key	C1_Key	Worlds
1	3	<0,2>
3	1	<0,2>

C_FirstName	Worlds
Joe	<0,1,2,3>
Steve	<0,1,3>
Tom	<0,1,2,3>

Figure 5: Annotation Propagation for Select-Project-Join

5.1 Select-Project-Join

We first discuss the relational algebra operators select (σ), duplicate-eliminating project (π) and cross-product ($\times$) since join can be reduced to a selection over a cross-product. We denote the modified operators σ', π' and $\times'$ respectively. We modify each operator by letting it operate exactly as the original operator and adding the following logic to populate the annotation field of each intermediate record.

1. For select and duplicate preserving project, the annotation of an output record is the same as its input annotation. The intuition is that the selection or the projection is only a function of the attributes and not the annotations.

2. For cross-product, the annotation of an output record $< r_1, r_2 >$ is the intersection of the annotations of the input records, that is $r_1.Worlds \cap r_2.Worlds$. The intuition is that a record-pair (r_1, r_2) is present in exactly the worlds that contain both r_1 and r_2.

3. For duplicate-eliminating project, the annotation of an output record is the union of the annotations of all input duplicates. More precisely, suppose that the projection is on columns $\bar{C}$, then the annotation of output record r

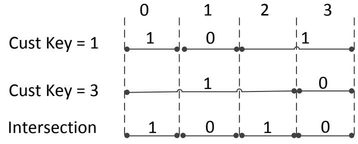


Figure 6: **Merging Runs For Computing Intersection**

is $\cup_{r':\pi_{\bar{C}}(r')=r} r'.Worlds$. The intuition is that an output record is present in a world if it contains *any* of the underlying duplicates.

A join is equivalent to a cross-product followed by a filter. Suppose we have a θ-join with the filter predicate θ. It follows from the cross-product and filter evaluations described above that we can evaluate the join by running the original join over R and S and compute the annotation for every pair of joining records as the intersection of the base records' annotations. Figure 5 illustrates examples of the above technique.

We now discuss how the above operator modifications can be extended to handle annotation compression. The straightforward method is to decompress the annotation and invoke the above technique. However, the size of a decompressed annotation can be large. We therefore design methods that operate on the compressed annotations directly. Since a selection and duplicate-preserving projection do not modify the input annotations, they can propagated as is without decompression. A cross-product (and join) requires computing the pairwise intersection of annotations. We note that set intersection corresponds to computing the bitwise `and` of the corresponding boolean vectors. The bitwise `and` of two vectors compressed using run-length encoding can be computed by merging their runs *without* decompressing either of them. The output of the merge returns the output in the desired compressed form. Figure 6 illustrates the merging of runs to compute their bitwise `and`.

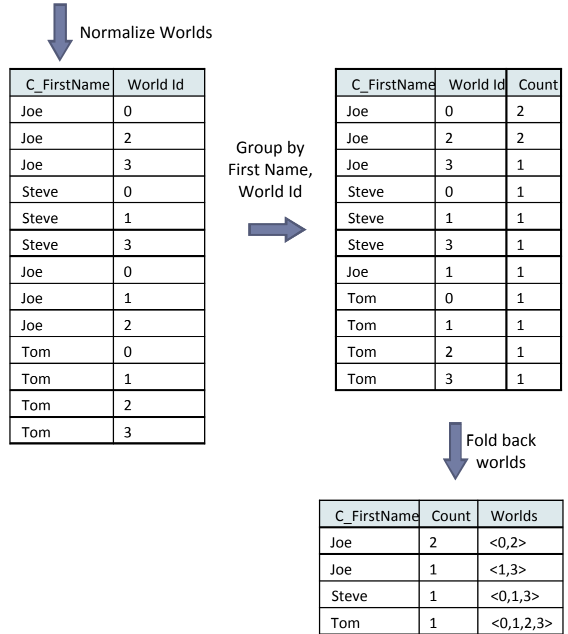
A duplicate-eliminating projection requires us to take the union of a collection of annotations. Since union corresponds to the bitwise `or` of the equivalent boolean vectors, we can reduce the overall union to a series of pairwise operations on the compressed annotations. (Section 5.2 introduces multi-way merge algorithms for computing aggregations including multi-way union.)

One of the biggest challenges in prior work on record annotations is the size of the annotations which can be linear in the size of the database even for select-project-join queries [24]. We now formally show that by using run-length compression, the number of runs in output annotations is linear in the number of cross-product operators in the query which is typically much smaller than the database size.

THEOREM 5.1. *Let $\mathcal{D}$ be a world-set induced by an audit expression V over database D. Let $\mathcal{D}$ be represented by annotated database D^w. Consider a query Q represented as a tree of select-project-cross-product operators. As before, let Q' denote the tree obtained by modifying each operator. The number of runs in each record in the output of $Q'(D^w)$ is linear in the number of cross-products in Q.*

5.2 Groupby-Aggregation

The input to the groupby operator is an annotated relation R^w. The goal is to compute the result of the groupby for each world. In order to illustrate the groupby output, we state an adaptation of the straightforward evaluation that iterates over all worlds:



C_Custkey	C_FirstName	C_LastName	C_Acctbal	Worlds
1	Joe	Frank	150k	<0,2,3>
2	Steve	Hanks	110k	<0,1,3>
3	Joe	Baker	110k	<0,1,2>
4	Tom	Hill	70k	<0,1,2,3>

C_FirstName	World Id
Joe	0
Joe	2
Joe	3
Steve	0
Steve	1
Steve	3
Joe	0
Joe	1
Joe	2
Tom	0
Tom	1
Tom	2
Tom	3

C_FirstName	World Id	Count
Joe	0	2
Joe	2	2
Joe	3	1
Steve	0	1
Steve	1	1
Steve	3	1
Joe	1	1
Tom	0	1
Tom	1	1
Tom	2	1
Tom	3	1

C_FirstName	Count	Worlds
Joe	2	<0,2>
Joe	1	<1,3>
Steve	1	<0,1,3>
Tom	1	<0,1,2,3>

Figure 7: **Basic Evaluation of Groupby**

1. Convert the annotated relation to first normal form to obtain a relation of (record,world-id) pairs.

2. Modify the original groupby operation to add the world-id column to the grouping columns and run the groupby.

3. Fold the results back into the form of an annotated table representing the result.

We illustrate through an example.

EXAMPLE 5.2. *Consider the annotated relation in Figure 4. Suppose we wish to group by the customer first name and compute the number of rows per group:*

```
Select C_FirstName, Count(*)
From Customer
Group by C_FirstName
```

Figure 7 illustrates the above method for computing the result of groupby.

The main issue with the above method is that the size of the first normal form table can be really large as we illustrate below.

EXAMPLE 5.3. *Consider a world-set induced by an audit expression over a database where the number of worlds is n. The size of the first normal form relation computed over the relation underlying the view is $O(n^2)$.*

We now develop our evaluation algorithm for groupby. We first note that when there is no aggregation to be performed, groupby

is a duplicate-eliminating projection. Therefore, the technique proposed in Section 5.1 is applicable. Based on our discussion in Section 5.1, we also observe that the set of groups in R^w is the set of groups we need to consider in the output. The main challenge we face is to compute the aggregates. As illustrated in Figure 7, the aggregates can be different for different worlds. Our goal is to compute all the different aggregates efficiently by working on the compressed annotations.

We describe our approach for a single `Sum` aggregation (the techniques extend for more complex aggregates). Suppose that the column being summed is A. The basic evaluation of groupby outlined above keeps track of the aggregation in different worlds by explicitly materializing the values that contribute to each world. Our idea is to extend compression to represent the above information. For a record $r \in R^w$, we modify the boolean vector corresponding to its annotation to replace the bit 0 with the value 0 and the bit 1 with the value $r.A$. We note that the above step can be performed on the compressed vector directly to yield a compressed numeric vector. We denote the value in a run run as $run.value$.

EXAMPLE 5.4. *Suppose we wish to compute the sum of all account balances for the annotated relation in Figure 4. Consider the first record, corresponding to the customer with* `c_custkey` *1. The boolean vector corresponding to its annotation is* $< 1, 0, 1, 1 >$. *This is converted to the numeric vector* $< 150k, 0, 150k, 150k >$. *The above operation can be performed on the compressed vector directly.*

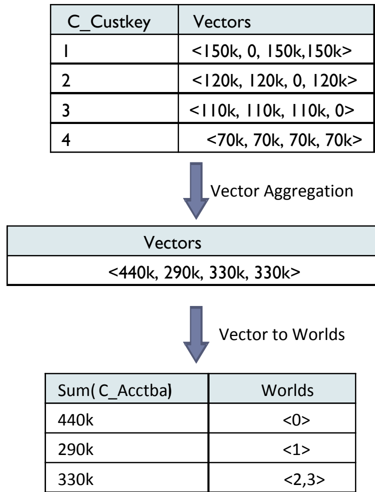


Figure 8: Illustrating Sum

It is straightforward to run a binary numeric operation over compressed numeric vectors. Therefore, we have a possible approach to compute the sum of a set of compressed vectors.

EXAMPLE 5.5. *Continuing with Example 5.4, the four vectors to be added and the sum vector are shown in Figure 8. Once the sum vector is computed, we produce the output in the expected form of an annotated relation, also shown in the Figure. We note here that the annotation of the summation 440k has only one world. This illustrates that even though the base table records are present in most worlds, the intermediate records produced in query evaluation could be present only in a small number of worlds (as we had noted in Section 4.4). The above example illustrates why we use run-length compression instead of storing the complement.*

In the above example, we note that there are three distinct aggregates for the 4 dimensions. In general, the overall aggregate can have different values in each dimension leading to an increase in space consumed by the aggregate vector. However, the final result size is still linear in n. For example, consider two extreme cases: (1) there is a small number of groups with a large number of distinct aggregates - even though each aggregate vector could consume significant memory, there would only be a small number of groups with such aggregates. (2) there are a large number of groups but with a small number of distinct aggregates - once again the total memory consumed will not sharply increase. This intuition is formalized in the following result.

LEMMA 5.6. *If there are n worlds, the total number of runs over all annotations in the final annotated relation returned by the aggregation is at most $2 \times n$.*

We now discuss the running time of computing the aggregates. We first show that performing a multi-way sum of compressed vectors as a series of binary summations can be expensive.

EXAMPLE 5.7. *Let us consider a generalized version of Examples 5.4 and 5.5. Consider an audit expression V with $n-1$ records in its output, leading to n worlds. So every record in the input annotated relation has an annotation with 3 runs. Suppose that we wish to perform a summation on one of the attributes. Consider the aggregate vector when $n/2$ records have been processed. As noted above, in general the aggregate is different in dimensions. When aggregating the remaining $n/2$ vectors, every binary operation over the vectors has to examine the aggregate vector. Therefore, the overall time taken is $O(n^2)$ in the worst case.*

We address the above problem by performing a multi-way merge. We store all vectors to be aggregated in memory. We make a sweep concurrently over all the vectors being aggregated from the smallest to the largest dimension. Whenever we encounter a run boundary, the result of the `Sum` computation potentially changes and so is recomputed. Our main observation is that the new value of the summation can be computed from the current sum as follows: suppose that the run boundary corresponds to a transition from a run with value x to a run with value y. Then, the new summation can be obtained from the current sum by subtracting x and adding y. Further, instead of considering all dimensions explicitly, it suffices to iterate over all run boundaries in order. Algorithm 5.1 shows the pseudo-code for the multi-way merge — given a collection $\bar{V}$ of vectors, we let $\texttt{Sum}(\bar{V})$ denote the dimension-wise sum of all vectors; we denote the lth vector in the collection as $\bar{V}[l]$. We note that even though in Step 6 of Algorithm 5.1, more than one run is potentially considered, every run boundary is processed precisely once. This observation leads to the following result which formalizes the efficiency of our algorithm. We note that the above algorithm can be generalized to other SQL aggregate functions (namely `count`, `min`, `max`) and multiple aggregations in a straightforward manner.

Figure 9 illustrates the summation discussed in Example 5.5.

THEOREM 5.8. *Suppose that we wish to compute any of the following aggregates —* `sum, count, min, max` *over an input annotated relation where the total number of runs over all annotations is N. The above groupby evaluation algorithm runs in time $O(N \lg(N))$.*

It is not hard to see that our query evaluation algorithm above yields significant benefits over the basic implementation of the groupby operator.

EXAMPLE 5.9. *In Example 5.7, the result of the groupby query is computed using the above technique in time $O(n \lg(n))$.*

Algorithm 5.1 MULTI-WAY MERGE OF COMPRESSED VECTORS

Input: Collection of vectors $\bar{V}$
Output: RLE encoding of vector $\texttt{Sum}(\bar{V})$
1: Let run_l be the first run of $\bar{V}[l]$.
2: Set $currentSum$ to 0
3: While (there are unprocessed runs in every vector)
4: Let $maxStart = max_l(run_l.start$
5: Let $minEnd = min_l(run_l.end$
6: For each (run_l with $run_l.start = maxStart$)
7: Let $prevValue$ be the value in the run preceding run_l
8: $currentSum = currentSum - prevValue + run_l.value$
9: Let run denote the following run:
10: $run.value = currentSum$
11: $run.start = maxStart$
12: $run.end = minEnd$.
13: Append run to the output run sequence by either growing the last sequence or creating a new run.
14: For each (run_l with $run_l.end = minEnd$)
15: Move run_l to the next run of $\bar{V}[l]$.
16: End While
17: Return the output run sequence

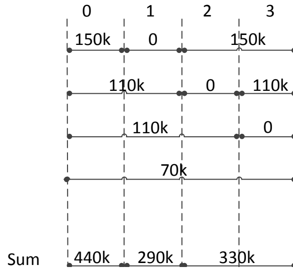


Figure 9: Merging Runs For Computing Sum

5.2.1 Space Consumption

The multi-way merge algorithm requires the vectors being aggregated to be cached in memory. We note that the space required is not a function of the number of worlds but the number of records participating in the aggregation. We illustrate with an example.

EXAMPLE 5.10. *Consider the following query.*

```
Select Sum(O_TotalPrice)
From Customer, Orders
Where C_Custkey = O_Custkey
    and C_MktSegment = 'AUTOMOBILE'
```

The number of records being aggregated is proportional to the number of orders in the automobile market segment. This could be much larger than the number of customers.

We reduce the space consumption in the above example by observing that orders made by the same customer share the same annotation. Thus, we can first compute a standard summation of all orders per customer before invoking our groupby evaluator. Then, the number of records input to our groupby evaluator is at most the number of customers in the automobile market segment. In general, we first compute the aggregation over all records sharing the same annotation using the standard aggregation supported by the database system. Our groupby evaluator consumes the resulting relation. The above technique is applicable to all standard SQL aggregates namely `sum`, `count`, `average`, `min` and `max`. Our experiments (that include the 100GB version of TPC-H) demonstrate our ability to work with large data. In order to scale our evaluation to an even larger number of individuals, we can leverage standard techniques such as partitioning (e.g., as used in a hybrid hash join). We defer the details of a partitioning based strategy to future work.

5.3 Top-k

We recall that the algorithm for computing the `min` aggregate is similar to the `sum` aggregate discussed in Section 5.2. We note that `min` is a special case of Top-k where $k = 1$. Not surprisingly, our algorithm for computing the Top-k on an annotated relation is similar to the algorithm for computing `min`. We store the records in memory, make a sweep concurrently over all annotations maintaining the Top-k for the worlds seen so far. We omit the details of the algorithm.

THEOREM 5.11. *Suppose that the number of worlds is n and that we wish to compute the Top-k over an annotated relation where the total number of runs over all annotations is N. The above evaluation algorithm runs in time $O(k \times N \lg(N))$. The total number of runs in the output is at most $k \times n$.*

We note that we can address the memory consumption of our modified Top-k operator using techniques similar to the groupby operator as discussed in Section 5.2.1.

5.4 Anti-Join

We consider a generalized version of set difference namely anti-join - such an operator is used to evaluate NOT EXISTS subqueries. Given two relations R and S and a set of columns $\bar{A}$, the anti-join operator $R \not\bowtie_{\bar{A}} S$ returns every record in R that (equi-)joins with no record in S on the columns $\bar{A}$. Our goal is to design a modified anti-join operator that has as input annotated relations R^w and S^w and a set of columns $\bar{A}$ ($Worlds \notin \bar{A}$), denoted $R^w \not\bowtie'_{\bar{A}} S^w$. It is not hard to see that we can break the result of $R^w \not\bowtie'_{\bar{A}} S^w$ into two parts:

1. $R^w \not\bowtie_{\bar{A}} S^w$. The above expression accounts for records in R^w that do not join with any record in S^w.

2. The second part handles record in R^w that do join with records in S^w. In order to account for duplicates in S^w, we first project $\bar{A}$ from S^w, that is compute $\pi'_{\bar{A}}(S^w)$. For each record $< r, s >$ in the output of $R^w \bowtie \pi'_{\bar{A}}(S^w)$, we return the record $< r, r.Worlds - s.Worlds >$ if $r.Worlds - s.Worlds \neq \phi$. The annotation $r.Worlds - s.Worlds$ finds worlds where r occurs but s does not by computing their set difference. We can compute the set difference between annotations directly in their compressed form using a pairwise merge.

We now remark on the space and time consumed by our modified set difference operator. We show the following result.

LEMMA 5.12. *If every record in R^w and S^w has an annotation with at most B 0s, then every output record has at most $4 \times B$ runs. Suppose that the total number of records in R^w and S^w is N. Then, the running time is $O(B \times N)$.*

5.5 Additional Optimization for Subqueries

We first note that the above operators suffice to address a large class of complex SQL queries including the queries in the TPC-H benchmark. We use standard decorrelation techniques [23] to produce a plan only involving the above operators. However, there is an additional optimization that we can perform for subqueries which we now illustrate through an example. Consider the following subquery.

```
Select *
From Customer
Where C_AcctBal > (Select Avg(C_AcctBal)
                   From Customer)
```

The above query is executed as follows: the inner query is executed once to produce the single aggregate which is used as part of the outer as a selection predicate.

Now let us consider an "annotated" evaluation of the above plan. Executing the inner query yields multiple values of the aggregate corresponding to different worlds. Therefore when we use the result of the inner as part of the outer query, the selection becomes an inequality join. While our techniques described for joins are applicable, the performance of the join is poor since it is an inequality join.

We address the above issue as follows. The selection connecting the inner and outer subqueries gets converted to a join since we return the result of the inner query as an annotated relation. Instead, consider returning the intermediate aggregate vector we compute as illustrated in Figure 8 — we model the above join as a selection over the aggregate vector of the form *column op vector*.

For the example data in Figure 8, the aggregate vector pertaining to the above query is $< 110k, 290k/3, 110k, 110k >$. The selection predicate is `C_AcctBal` $> \ < 110k, 290k/3, 110k, 110k >$. For a given value of *column*, the predicate is true on a subset of dimensions that we refer to as the *satisfying* dimensions. For each record, we compute the satisfying dimensions and (1) return the record only if there is at least one satisfying dimension, (2) modify the annotation to be the set of satisfying annotations. The above operation can be evaluated on the compressed annotations.

5.6 Analysis

We now discuss the correctness and efficiency of the overall query evaluation. For a query Q represented as a tree of operators, let Q' denote the modified operator tree obtained by replacing each operator with its modification. We show the following result that proves the correctness of the above operator modifications.

THEOREM 5.13. *Let $\mathcal{D}$ be a world-set represented by annotated database D^w. Consider a query Q represented as a tree of operators. Then, $WorldSet(Q'(D^w)) = Q(\mathcal{D})$.*

While we have described analyses of the efficiency of various operator modifications above, we defer a formal analysis of the efficiency and space consumption of an arbitrary operator tree composed of the above operators to future work. We conduct an empirical evaluation over benchmark queries and data to evaluate the performance of our query evaluation in practice. The next section describes our evaluation.

6. EXPERIMENTS

We now describe our empirical evaluation of the techniques presented in the paper. The goals of our study are: (1) to study the benefits yielded by our annotation based approach compared to the baseline approach, (2) to study if the proposed algorithms can work well "at scale" - both in the data size as well as the number of sensitive records (3) to study the utility of compressing annotations.

6.1 Implementation

We implement our `References` operator as a client-side tool. We receive a SQL query as input. The query is first parsed using the query optimizer to obtain a physical plan. Although we described our query evaluation in terms of operators, our implementation is based on *blocks* that span multiple operators. For example, we collapse a sequence of select, project and join operations into a single select-project-join block. In order to detect the blocks from the physical plan, we convert the plan into a logical form that is then converted into a tree of blocks. The query optimizer decorrelates subqueries in a cost-based manner. However, if the final physical plan returned does contain a subquery, we explicitly decorrelate the subquery. We use the audit expression to induce a world-set which is used along with the block tree to run the multi-world query evaluation algorithm. The final result of the `References` operation is then computed using the result of the query over all worlds. We encapsulate set and vector operations as user-defined functions. Thus, most of the references operator logic executes in the database server. The client logic mostly issues queries to the server. This enables us to scale our implementation with data size.

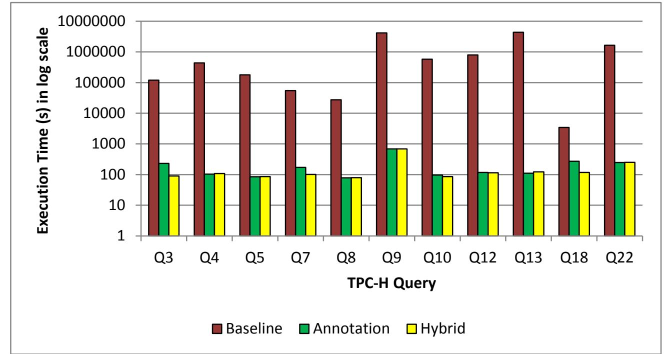


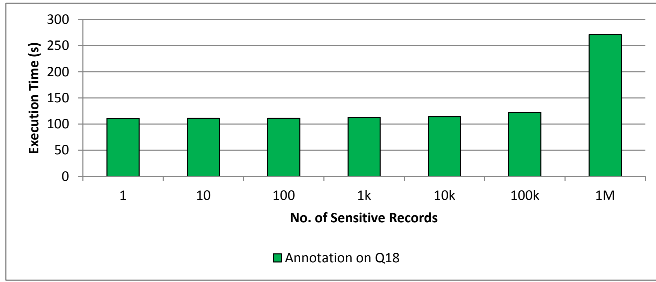
Figure 10: Comparison With Baseline

6.2 Experimental Setup

We begin by describing our empirical set up. In order to study complex queries, our experiments are conducted over the TPC-H benchmark data and queries [26]. Our audit expression contains all customers. Since only queries over the `customer` table can access customer rows, we focus on queries from the benchmark over the `customer` table. The queries involve complex constructs of SQL such as grouping, nested queries and top-k. Most results are reported on the 10GB version of the database. We use Microsoft SQL Server 2008 for all experiments. Experiments are conducted on a dual-processor machine with 12GB of RAM. The physical design used represents a basic tuned database that has indexes on primary keys and foreign keys. All execution times reported are cold buffer times. The running time of the baseline algorithm is an estimate. Recall that the baseline algorithm evaluates $Q(D - t)$ for a (potentially) large number of t. Our estimate multiplies the total number of iterations with the expected running time for a randomly chosen t. We approximate the expected value in the standard way by computing the average running time over a small number of randomly chosen t to obtain a high-confidence estimate.

We name the algorithms as follows in the plots below: (1) we name the baseline algorithm that filters the audit expression using the query predicates as `Baseline`, (2) we refer to the query evaluation algorithm that leverages annotation propagation as `Annotation`, (3) we note that the algorithm that uses the query predicates as a filtering on the rows in the audit expression is complementary to the the annotated query evaluation approach and thus can be combined with it. We denote the algorithm that uses both approaches as `Hybrid`.

6.3 Comparison With Baseline Algorithm



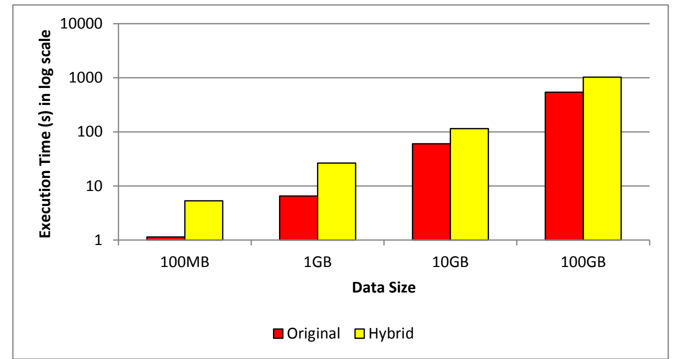


Figure 11: Varying (a)the number of sensitive records (b)data size

Fig. 10 shows the results of the comparison between our algorithm for the `References` operator and the baseline algorithm. The X-axis shows the benchmark queries and the Y-axis reports the execution time in seconds. The Y-axis is plotted in log scale. As noted above, the data size is 10GB. We observe from Fig. 10 that on the whole, our annotated query evaluation algorithm yields orders of magnitude improvement in performance over the `Baseline` algorithm. The number of customers in the 10GB data set is 1.5 million. The `Baseline` algorithm does succeed in pruning the customers significantly by using the filtering step; for example, for query 18, the number of customers is pruned using the query predicate to 597. While the filtering drops the number of query executions from 1.5M to 597, the time taken for 597 executions is still significant. On the other hand, our algorithm computes the accessed rows in one annotated query execution. Further, we also observe that combining the filtering offered by `Baseline` with the `Annotation` algorithm improves the overall performance of the `References` operator substantially in several cases. For example, for query 3 and query 18, `Hybrid` is more than twice as fast as `Annotation` (the comparison between `Annotation` and `Hybrid` can also be seen in Fig. 12). For some queries such as 5, 8 and 13, `Hybrid` is slightly worse than `Annotation`. The reason is that while filtering the audit expression using the query predicates is beneficial, we also incur the cost of running the filter. Sometimes, the combined execution time exceeds the time for `Annotation`.

6.4 Varying The Number Of Sensitive Records

We next study the performance of our algorithm as the size of the audit expression varies. We vary the size of the audit expression by restricting the range of customer keys. Fig. 11(a) shows the results of our experiment. The X-axis shows the number of customers in the audit expression and the Y-axis, the execution time in seconds. We pick query 18 from the benchmark since it is among the most expensive in Fig. 10 and since it is one of the most complex queries in our study and involves grouping, subqueries and top-k. We report the time taken by the `Annotation` technique since we wish to study the effect of the audit expression size on the annotated query execution in the absence of any filtering. As expected, the query execution time increases as the number of sensitive records increases. However, the increase in execution time is less than the corresponding increase in the number of sensitive records — this is because our representation of worlds is compressed and the size of the compressed representation and the cost of processing compressed information increases sub-linearly with the number of worlds.

6.5 Effect of Data Size

We next study the scalability of our annotated query evaluation algorithm as the data size increases. We again pick query 18 since it is the most complex in our query set. We vary the size of the TPC-H data set from 100MB to 100GB. Fig. 11(b) shows the result of our experiment. The X-axis plots the data size and the Y-axis, the execution time in log scale (we also plot the execution time of the original query). The main trend we observe from the figure is that the performance of our annotated query evaluation algorithm scales similar to the original query. The above result is noteworthy because the number of customers increases linearly with data size therefore the number of query executions in the `Baseline` algorithm also increases linearly; therefore the `Baseline` algorithm is expected to scale quadratically with data size. However, our the `Annotation` algorithm scales linearly as the original query does.

6.6 Utility of Operating on Compressed Annotations

Finally, we study the utility of leveraging annotation compression. We use the 100MB database for this experiment. We pick query 13 and show the results of two executions: (1) the `Annotation` algorithm, (2) a modified version of the `Annotation` algorithm in which all operators operate by first decompressing the annotation and then compressing the result. We find that while the `Annotation` algorithm runs in 5.32 seconds, the modified version runs in 49.27 seconds. Therefore we obtain an order of magnitude improvement in performance by operating directly on compressed annotations. The benefits of operating directly on compressed annotations is even more significant for larger data.

6.7 Summary

We now summarize the results of our empirical evaluation. The results indicate that the annotations based approach is essential for enabling auditing at large scale. The baseline approach (even with filtering) could still involve hundreds of query executions and is not practical. Interestingly, the filtering approach is complementary and can be run in conjunction with the annotation based execution scheme. In order to place the experimental results in context, it is interesting to compare the running time of the `References` operator with that of the original query. Fig. 12 shows the results of the comparison. The X-axis shows the queries and the Y-axis, the execution time in seconds (we note that Fig. 12 is not in log scale). For queries 3, 5, 7 and 8, our technique is at most 1.5 times as expensive as the original query. We also note that although `Hybrid` is substantially faster than `Annotation`, the `Annotation` technique is on average 4 times slower than the original query and in the worst case (query 13), 11 times slower; this holds out the promise

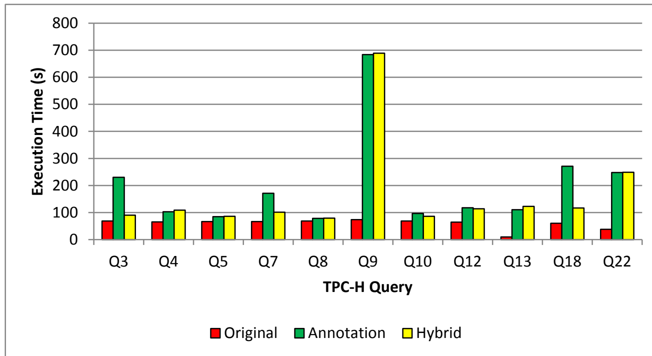


Figure 12: **Comparison With Original Query**

that even for cases when the filtering implied by `Baseline` does not yield any benefit, the `Annotation` technique still has an acceptable running time. While a factor of 11 when compared to the original query execution may seem like a high overhead, recall that checking if a query accessed an individual's record requires two executions for complex queries (Section 3.4). The above result shows that we can effectively check if a query accessed 1.5M users at the overhead of running the query 10 times — which is roughly the cost of checking 5 users using the straightforward algorithm. This points to the effectiveness of our framework.

We note that the key to the scalability of our framework is the ability of the operators to work directly on the compressed annotations. In our experiments thus far, this optimization has worked out well. We defer an evaluation of the scalablility of our algorithms over other query workloads to future work.

7. CONCLUSIONS

In this paper, we studied the problem of efficiently finding all sensitive records referenced by a given query. The main challenge we addressed was to develop a technique that scales with the number of sensitive records. We formulated a multi-instance query evaluation problem where we want to find the result of a query on multiple subsets of the database. We proposed techniques that efficiently perform such multi-instance query evaluation in a single pass by sharing computation and demonstrated the scalability of our techniques over the queries in the TPC-H benchmark. A few interesting directions for future work are: (1) developing more tools that build upon our `References` operator to mine the audit log, (2) choosing the plan and physical design that minimizes the cost of multi-instance execution.

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*, pages 516–527, 2004.

[2] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *PODS*, 2011.

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

[4] Privacy Rights Clearinghouse. Chronology of data breaches. `http://www.privacyrights.org/data-breach`.

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

[6] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and querying databases through colors and blocks. In *ICDE*, 2006.

[7] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6), 2012.

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

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

[10] B. Glavic and R. J. Miller. Reexamining some holy grails of data provenance. In *TaPP '11: 3rd USENIX Workshop on the Theory and Practice of Provenance*, 2011.

[11] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Provenance in ORCHESTRA. *IEEE Data Eng. Bull.*, 33(3), 2010.

[12] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[13] R. Ikeda and J. Widom. Panda: A system for provenance and data. *IEEE Data Eng. Bull.*, 33(3), 2010.

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

[15] Butler Lampson. Privacy and security: Usable security: how to get it. *Commun. ACM*, 52(11):25–27, November 2009.

[16] A. Machanavajjhala and J. Gehrke. On the efficiency of checking perfect privacy. In *PODS*, 2006.

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

[18] Microsoft Corporation. SQL Server 2008 Change Data Capture. http://msdn.microsoft.com/en-us/library/bb522489.aspx.

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

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

[21] Oracle Corporation. Oracle Flashback Query. http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28424/adfns_flashback.htm.

[22] A. Das Sarma, O. Benjelloun, A. Y. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.

[23] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *ICDE*, 1996.

[24] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[25] Germany Tackles Tax Evasion. Wall Street Journal, Feb 7 2010.

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

[27] D. J. Weitzner, H. Abelson, T. Berners-Lee, et al. Information accountability. *Commun. ACM*, 51(6):82–87, June 2008.