Automatic Enforcement of Data Use Policies with DataLawyer

Prasang Upadhyaya
Department of Computer Science and Engineering
University of Washington
prasang@cs.uw.edu

Magdalena Balazinska
Department of Computer Science and Engineering
University of Washington
magda@cs.uw.edu

Dan Suciu
Department of Computer Science and Engineering
University of Washington
suciu@cs.uw.edu

ABSTRACT

Data has value and is increasingly being exchanged for commercial and research purposes. Data, however, is typically accompanied by terms of use, which limit how it can be used. To date, there are only a few, ad-hoc methods to enforce these terms. We propose DataLawyer, a new system to formally specify usage policies and check them automatically at query runtime in a relational database management system (DBMS). We develop a new model to specify policies compactly and precisely. We introduce novel algorithms to efficiently evaluate policies that can cut policy-checking overheads to only a few percent of the total query runtime. We implement DataLawyer and evaluate it on a real database from the health-care domain.

1. INTRODUCTION

Data has value; it is increasingly bought and sold on the Web and exchanged between organizations and individuals. Schomm et al. [44] list 46 commercial data suppliers; 34 of whom provide data in a relational or semi-structured form[1]. Even companies whose business model has traditionally been based on selling information, such as Dun & Bradstreet or Reuters[2], are starting to offer some of their valuable data on the Web, albeit under major restrictions. Some of the most successful companies today are those that hold, can produce, or acquire unique and valuable data, such as geographical data (Google maps), social data (Facebook), corporate data (Dun&Bradstreed), maps (Navteq), or miscellaneous data extracted, integrated, and cleaned (Factual).

Whether sold online or offline, data typically comes with terms of use, which limit how the buyer can use the data. These are usually written by lawyers, span multiple pages, are difficult to read and are sometimes ambiguous. We performed an informal survey of 13 data providers[3] and found that terms of use accompanied all of the datasets or the APIs to acquire those datasets. Table 1 lists some examples of the terms we found. These terms restrict, in various ways, how the data can be used: for example P_1 says that Navteq prohibits users from joining their map data with other datasets: if the user wants to join, she needs to purchase the data at a higher price; P_2 limits which users can access a given data item depending on the context; P_3 limits the amount of data that can be retrieved in a time window, etc. These are much simplified descriptions. In our survey, the average length of the terms of use document was 4489 words or about 8.3 pages; all were written in natural language, most often using legal terms, were often difficult to read, understand, and remember, and were sometimes ambiguous.

While individual end-users happily ignore terms of use (commonly clicking the *I-agree* box without even opening the accompanying document), when a major firm acquires a valuable database, it cannot ignore its terms of use; they risk significant losses if their employees fail to adhere to the terms of use. This means that the company is responsible for, and interested in monitoring how its employees access, view, use, and manipulate data from its valuable databases. This often applies also to data produced internally by a company. For example, all large Internet companies put significant value on their user data (and their user privacy) and restrict both the employees who can access the data and what they can do with that data: *e.g.*, end-users may have agreed that their data be used to improve a certain product but for no other purpose. The company must respect this restriction.

To date, firms have few ways to enforce terms of use and typically rely on access restrictions and extensive employee training.

In this paper, we propose to specify the usage policies formally, and check them automatically at query time. We start by introducing a formal language, based on SQL, for specifying usage policies, which is rich enough to express all policies encountered in our survey (such as those in Table 1). Then, we present the DataLawyer system for enforcing data use policies automatically, inside a relational database management system (DBMS). DataLawyer is as a middleware layer on top of a relational DBMS that allows users to run normal SQL queries, but before letting a query execute, it checks all policies. If any policy is violated, the query is rejected and the user is informed about the violation; otherwise, the query is evaluated in normal fashion.

The major challenge in any policy enforcement system is performance. Without the system, users would issue regular

[1] The other data type are free style reports.

[2] Reuters, world's oldest information company, was founded in 1851.

[3] Foursquare [8], Yelp [20], Azure Marketplace [17], Twitter [14], Infochimps [9], Socrata [15], Xignite [19], Digital Folio [6], DataSift [5], World Bank [18], Navteq [12], data.gov.uk [13], and DataMarket [4].

	Examples of terms of use	Restriction type
P_1	Overlaying Navteq data with any other data is prohibited (Navteq [12])	Prohibit joins
P_2	Each book may be lent once for 2 weeks while being inaccessible by the lender (Amazon Kindle [1])	Group licenses
P_3	All queries, totaled over a month, may return up to 2M chars at the free tier (MS Translator [10])	Generating free samples
P_4	OAuth calls are permitted 350 requests per hour (from Twitter [14] and a similar policy at Foursquare [8])	Rate limiting
P_5	Queries that try to identify an individual referenced in the database are prohibited (MIMIC II [11])	Limit information disclosure
P_6	You are required to display all attribution information and any proprietary notices associated with the Foursquare Data (Foursquare [8] and similar policies in Yelp [20] and World Bank [18])	Attribution and provenance
P_7	Don't aggregate or blend our star ratings and review counts with other providers. You may show content from multiple providers, but Yelp data should stand on its own (Yelp [20])	Disallow aggregations but allow joins and unions

Table 1: Example policies from commercial data sellers and their informal classification.

SQL queries. In the case of long-running analytical queries, the overhead of policy checking is easily amortized. In all other cases, however, policy checking can significantly slow down performance (Figure 1 in §5, for example, shows how a baseline non-optimized policy checking approach can impose a high and *growing* overhead).

Automatic policy enforcement is expensive because the system needs to record a significant amount of information about each query in a *usage log*: *e.g.*, the user identifier and query time, the query text, the query result, the provenance expression. In addition, it needs to execute every policy against the data and the usage log in order to check compliance. Done naïvely, this can become multiple orders of magnitude slower than running the SQL query alone. High overhead presents a major barrier for adoption: a company would not adopt a policy enforcement system, if it significantly slows down its daily operations.

To address the performance challenge, we propose novel algorithms to efficiently evaluate policies. These algorithms are based on query-rewriting techniques that leverage the separation of policies into data (the usage log component) and query (the declaratively specified policies). We develop two major optimizations.

The first optimization *log compaction*, addresses the problem of *high and growing* overhead of policy evaluation due to a growing usage log. This optimization removes from the log tuples that are no longer necessary for future policy checking. This optimization is based on the observation that, while in principle one could write policies that check the log arbitrarily far in the past, in practice policies tend to look for specific event patterns that are restricted to a limited subset of the log. For example, P_1 in Table 1 checks that Navteq data is not joined with other data sets. Since we know this policy was enforced for all past queries, we only need to keep the tail of the log generated by the current query. P_3 only requires the maintenance of the past month of data related to the free tier of service. Notice that log compaction is much harder than the tail-compaction used in recovery logs, which simply deletes the tail of the log after the last checkpoint: in our case we need to reason about the semantics of the policies. We show that, for each policy specified in the system, one can compute an *absolute witness*, which is a subset of the log that is guaranteed to be sufficient to evaluate that policy, both now and in the future. The log compaction is obtained by replacing the log with the union of all absolute witnesses for all policies.

The second optimization, *interleaved evaluation*, represents an advanced method to efficiently evaluate expensive policies. We start from the observation that, by far, the most common case is when all policies are satisfied: this is when users issue queries that comply with the policies, and this is when they expect to see no significant slowdown over using the system without policy enforcement. Our second observation is that, when a policy is satisfied, it is usually because there is a fragment of the policy that already proves it is satisfied. In interleaved evaluation, we evaluate simplified versions of the original policies, and stop evaluation early, when we have determined that there are no violations.

Contributions In summary, this paper makes the following contributions:

1. In §3, we propose a novel model to specify policies with the desiderata that any such model be sufficiently flexible to express a variety of policies, while being compact and precise. Our key idea is to first capture a relevant subset of user and database actions taken during query execution; and then to specify the policies *declaratively* over those logs as states of the log that are inconsistent with the intent of the policies. We describe the semantics of our data model and show how the common policies found in real world terms of use can be concisely and precisely expressed in our data model.
2. In §4, we present our optimized policy evaluation methods including log compaction and interleaved evaluation.
3. Lastly, in §5, we show experimental results from evaluating variants of policies defined in Table 1 on the MIMIC II [11] database. Our results demonstrate the practicality of our approach and the importance of our optimizations. While it is possible with DataLawyer to write policies that perform expensive checks, DataLawyer's optimizations enable the system to keep the overhead constant and, in many cases, cut that overhead to a few percent of the total query runtime, which is from 10× up to 330× less than an unoptimized implementation.

The source code [21] for DataLawyer's implementation for PostgreSQL is available publicly.

2. MOTIVATION

DataLawyer's goal is to enable data sellers to specify precise data use policies and to help data buyers to use the data without violating any of the policies.

DataLawyer should not be confused with an access control system [36]; such systems restrict users to a *fixed* set of authorization privileges (or access modes), which are strictly limited to reading or writing columns or rows. In contrast, terms of use refer to complex scenarios, *e.g.*, in Table 1, P_1 specifies that the data may not be joined with any external datasets, P_4 limits the rate of queries, P_7 disallows aggregation; none of these are captured by access control systems.

DataLawyer is also not intended to protect against a malicious user; with some patience and effort a malicious user can extract the entire data without breaching any policy, then store it on her own device and use at will. The system is designed to help honest users comply with terms of use that are difficult to read, understand, and remember. It is also meant to help large corporations monitor how their valuable data is used internally by their employees.

Finally, we note another potential application of DataLawyer: usage-based data pricing. Wang et al. [46] postulate that the value of data is both intrinsic (*e.g.*, based on its completeness and accuracy) and extrinsic (*i.e.*, it depends on the context in which it is used). Data owners frequently control the extrinsic value of data by limiting the kind of operations allowed on it. For example, Factual [7], an online data vendor, prices its data based both on volume and on what the buyer uses the data for: data used in ads is priced differently from data used in applications and prices can also vary between applications. DataLawyer can be used to compute the price of the data dynamically, *e.g.*, based on how the data was used during the last billing period.

3. POLICIES

In this section, we present the policy specification formalism, the usage log, and their semantics.

3.1 Policy Specification

For each term of use, the data owner defines a policy π, which is a SQL query of the following form:

```
SELECT DISTINCT [error-message] FROM ...
WHERE ... GROUP BY ... HAVING ...
```

The `FROM` clause contains base tables, or `select-from-where-groupby-having` subqueries. The `WHERE` and `HAVING` clauses are conjunctions of atomic predicates without subqueries.

The SQL query may refer both to the database and to a *usage log* that we describe in §3.2. If the policy π returns the empty set, denoted $\pi = \emptyset$, then the policy is satisfied. Otherwise, we say that π returns *true*, denoted $\pi \neq \emptyset$; in that case a violation has been detected and the error-message specified in the policy is returned to the user.

We illustrate our policy language with two examples.

Example 3.1. *P_{5b} is a concrete variant of P_5 in Table 1:*

> *P_{5b}: Stop queries where fewer than 10* `patient`*s contribute to any output tuple.*

This policy is from the MIMIC II (Multiparameter Intelligent Monitoring in Intensive Care) [11] database that contains readings from patient monitoring systems and clinical data collected at an ICU over seven years. In our system, the policy is specified as follows:

```
SELECT DISTINCT 'P5b violated: Fewer than 10
  patients contribute to an answer' AS errorMessage
FROM Provenance p
WHERE p.irid = 'patients'
GROUP BY p.ts, p.otid
HAVING COUNT(distinct p.itid) < 10
```

The table `Provenance(ts, otid, irid, itid)` in the `FROM` clause is part of the usage log (described in detail below). A record in `Provenance` represents the fact that input tuple `itid` from input relation `irid` belongs to the provenance of the output tuple `otid` of the query executed at time `ts`. To simplify the presentation, we assume that timestamps are taken from an integer clock with sufficient granularity that each query has a unique `ts` attribute.[4] In the examples, we further assume that timestamps are expressed in seconds. Thus, the policy simply checks if there exists some query that has an output tuple whose provenance has fewer than 10 distinct input tuples from the `patients` table, which is one of the tables in the MIMIC II database[5]. If the answer is nonempty, then the policy has been violated. The policy appears to check all *queries, but our system only evaluates it on the* last *query, namely the query currently requested by the user (because all previous queries have already been checked and are known to satisfy all policies); we explain this and other optimizations in §4.*

Example 3.2. *We now illustrate a more complex policy, involving temporal restrictions:*

> *P_{2b}: At most 10 distinct users from the group `'Students'` are allowed to query `patients` in any window of 14 days.*

This policy generalizes P_2 in Table 1, and is adapted to the same MIMIC database, for illustration purposes. The policy is expressed as follows in our language:

```
SELECT DISTINCT 'P2b violated: More than 10 users
   executed queries in 14 days.' AS errorMessage
FROM Users u, Schema s, Groups g, Clock c
WHERE u.ts = s.ts and s.irid = 'patients'
  and u.uid = g.uid AND g.gid = 'Students'
  and u.ts > c.ts - 1209600
HAVING COUNT(distinct u.uid) > 10
```

Notice that the query has no `GROUP BY` clause: it checks if the total number of distinct user id's that referred to `patients` over the last 14 days is greater than 10. This policy refers to two other tables in the log: `Users`, which records the user id of the user issuing the query, and `Schema`, which records the schema information for each query; both are described below. The `Clock` relation has a single row and a single column updated by the system[6]

3.2 Usage Log

We now describe the *usage log*, denoted $\mathbf{L}$, that captures all features of queries executed on the database that are necessary to enforce a set of data use policies. The log consists of m relations, $\mathbf{L} = (R_1, \ldots, R_m)$, where each relation R_i captures features of a particular type. Any feature can be stored in the log; the only requirement is that each relation has a timestamp attribute, $\mathtt{R}_i\mathtt{(ts,\ldots)}$. In addition, the system defines m *log-generating functions* (referred to as functions for brevity), $\mathbf{f} = (f_1, \ldots, f_m)$. When a query q is executed on the database instance D, if it satisfies all policies then, for each relation R_i, the system uses the function f_i to compute the set of features $S_i = f_i(q, D)$ to be appended to R_i: it then updates $R_i = R_i \cup (\{t\} \times S_i)$, where t is the current timestamp.

In addition to the usage log, the system also exposes the `Clock` relation that has a single row with a single attribute corresponding to the current time (see Example 3.2).

Example 3.3. *The usage log in our DataLawyer system prototype consists of the following three relations:*

[4] We could relax this assumption by adding a separate, unique `query id` attribute but that complicates examples.

[5] Its schema is `patients(pid, dob, sex)`.

[6] Alternatively, we could have used a function like `CURRENT_TIMESTAMP()`, with the same semantics as `c.ts`.

```
Schema(ts, ocid, irid, icid, agg)
Users(ts, uid)
Provenance(ts, otid, irid, itid)
```

`Schema` records the schema information of each query: a record `(ts, ocid, irid, icid, agg)` represents the fact that the answer to the query executed at time `ts` contains a column `ocid`, which stores a value derived from the input column `icid` from the input relation `irid`; `agg` indicates whether an aggregate was used. For example, the query, `SELECT T.A AS K, (T.B + T.C) AS L FROM T`, generates three rows in `Schema`:

```
(ts, K, T, A, false)
(ts, L, T, B, false)
(ts, L, T, C, false)
```

The log-generating function $f_{\texttt{Schema}}(q, D)$ takes as input a SQL query q and computes all tuples that need to be inserted in `Schema` on behalf of q, by performing static analysis on q; this function does not need the database instance D.

The `Provenance` relation contains complete provenance information. We use the set of contributing tuples provenance, *also called* lineage *[45], where, for each output tuple `otid` we record all contributing input tuples `irid, itid`. The function $f_{\texttt{Provenance}}(q, D)$ computes the provenance of q on D by running a `SELECT * ...` query derived from q, similarly to Perm [38]. Finally, `Users` stores, for each query, the id of the user who executed that query.*

In the rest of the paper, we assume that the schema of the usage log is that given in Example 3.3. We emphasize, however, that all our optimization techniques apply to an arbitrary schema: to add a new relation R_i to the log, the systems administrator only has to write the corresponding log-generating function $f_i(q, D)$.

3.3 Semantics

We can now define the semantics of our policy manager. We denote by $\mathbf{L}_t$ the log up to timestamp t, and denote $\mathbf{\Pi} = \{\pi_1, \ldots, \pi_p\}$ the set of policies defined in the system. Informally, when a user asks a query q over the database D at time t, the system first appends $\{t\} \times f_i(q, D)$ to each log relation R_i, to produce a tentative log $\mathbf{L}'_t$. Then, it checks all policies on the new log; if all return $\emptyset$, in notation $\mathbf{\Pi}(t, \mathbf{L}'_t, D) = \emptyset$, then the query is executed and the answer is returned to the user; otherwise the query is rejected and the log is reverted to $\mathbf{L}_{t-1}$. Formally:

$$\mathbf{L}_0 = \emptyset$$
$$\mathbf{L}'_t = \mathbf{L}_{t-1} \cup (\{t\} \times \mathbf{f}(q, D))$$
$$\mathbf{L}_t = \begin{cases} \mathbf{L}'_t & \text{if } \mathbf{\Pi}(t, \mathbf{L}'_t, D)) = \emptyset \\ \mathbf{L}_{t-1} & \text{otherwise} \end{cases} \quad (1)$$

Here $\mathbf{f}(q, D)$ denotes all functions $f_1, \ldots, f_m$. Each policy $\pi(t, \mathbf{L}_t, D)$ may refer to the log, the database, and also the timestamp t, obtained from `Clock` (see Example 3.2).

4. THE DATALAWYER SYSTEM

When the user issues a new query, a naïve way to check the policies is to apply directly the semantics of Eq.(1): generate the increment of the log $\mathbf{L}$, write it to disk, then iterate over the policies and evaluate them. If any violation is found, revert the log. We do not consider this naïve strategy. Instead, our NoOpt Algorithm 1 incorporates the following straightforward optimizations:

1. Generate only those logs R_i mentioned in the policy definitions. For example, if no policy uses `Provenance`, then do not generate that log at all.
2. Store the generated increments to the logs $\mathbf{f}(q, D)$ in temporary tables in memory and keep them there while checking the policies. Write them to disk only when all policies are satisfied. Without this optimization, in the case of failure, the log increments have to be deleted from disk.

Even with these two optimizations, checking policies with the NoOpt algorithm is impractical because (as we show) it can be 1-2 orders of magnitude slower than the query that the user wants to run. Additionally, with NoOpt, policy checking times increase as more queries execute against the database. In this section, we first present optimizations that (1) reduce the size of the log to keep policy-checking times constant (and low), and (2) use an optimized evaluation strategy for the set of policies to reduce policy-checking times compared to NoOpt. We then put these optimizations together and show how to apply them for a *set* of policies defined on a dataset.

Algorithm 1: The NoOpt Algorithm

```
Input  : Timestamp $t$, query $q$, database $D$.
Output: Update the log or abort.
begin
    BEGIN TRANSACTION
    $\mathbf{L} \leftarrow \mathbf{L} \cup (\{t\} \times \mathbf{f}(q, D))$
1   $\pi_{union} \leftarrow \pi_1 \cup \cdots \cup \pi_k$
2   if $\pi_{union}(t, \mathbf{L}, D) = \emptyset$ then COMMIT;
3   else ABORT;
```

4.1 Data Minimization

In theory, the log can grow forever, and the policies may inspect the entire log, from the beginning of time. In practice this never happens. We describe optimization techniques that exploit the fact that policies look only for restricted events back in time.

4.1.1 Time-Independent Policies

We start with a very simple optimization, which identifies when a policy depends only on the current query, and not on the log history. For example, policy P_1 in Table 1 prohibits joins of a dataset with other datasets: it only depends on the current query q and not the log history. In other words, we do not need to examine the entire log and check what previous queries have done.

We call a policy *time-independent* if it can be checked by examining only the increment of the log instead of the entire log: formally, denoting $\mathbf{L}_{past} = \mathbf{L}_{t-1}$ and $\mathbf{L}_{present} = \mathbf{L}_t - \mathbf{L}_{t-1}$, then $\pi(\mathbf{L}_t, D) = \pi(\mathbf{L}_{past}, D) \cup \pi(\mathbf{L}_{present}, D)$. We give here a syntactic criterion for time-independence. One subtlety is that a time-independent policy is usually not written in a way in which it refers only to the current time, but checks a property for all timestamps: we need to use the fact that the policy was true in the past to infer that it suffices to check only the current time. Formally, a SQL policy π is time-independent if it, and all its subqueries in the `FROM` clause, satisfy the following conditions: (a) all timestamp attributes from all relations are joined, and (b) if π contains any aggregates then the group-by attributes include the timestamp. If π is time-independent, we rewrite it to a policy denoted by π_{ind} by adding a selection requiring that the timestamps, `ts`, refer to the current clock, `c.ts`.

Example 4.1. *Policy P_1 and its optimization are:*

```
P1: SELECT DISTINCT 'No external joins allowed'
    FROM Schema p1, Schema p2
    WHERE p1.ts=p2.ts and p1.irid='Navteq' and p2.irid != 'Navteq'

P1_IND: SELECT DISTINCT 'No external joins allowed'
        FROM Schema p1, Schema p2, Clock c
        WHERE p1.ts = c.ts and p2.ts = c.ts and p1.ts = p2.ts
         and p1.irid = 'Navteq' and p2.irid != 'Navteq'
```

Thus, we restrict the policy to check only the current timestamp. This is correct only *because we know that the policy was satisfied in the past. In addition to restricting policy evaluation to the log increment, we can further optimize the system by not creating the log at all: this is handled by the log-compaction optimization, discussed next.*

4.1.2 Log Compaction

The log compaction optimization removes from the log entries that are guaranteed to be unnecessary now, and at any time in the future[7]. Notice that this is far more complex that the tail compression in recovery logs, which deletes the entire log preceding the last successful checkpoint. In our case, the compression must take into account the semantics of the policies and reason about which tuples will never be used in the future.

Example 4.2. *Consider policy P_{2b} from Example 3.2: no more than 10 distinct users from group* `'Student'` *may execute a query on* `patients` *in any window of 14 days. Assume for a moment that this is the only policy in our system. Then, it is obvious that we can (a) store in the log only entries belonging to users in the group* `'Student'` *and referring to table* `patients`*, and (b) remove all entries older than 14 days. By repeating this kind of reasoning to all policies we can compute a subset of the log that is sufficient to check all policies in the future.*

We give now the formal definition. Fan [35] defines a witness for a query Q to be a subset of the database s.t. Q returns the same answer on the witness as on the entire database. Adapting to our setting, let π be any query (Boolean or not), and $\mathbf{L}_t, D$ be the current log and the current database. A *witness* for π is a subset $\mathbf{L}_t^w \subseteq \mathbf{L}_t$ such that $\pi(t, \mathbf{L}_t, D) = \pi(t, \mathbf{L}_t^w, D)$. We call the tuples $T = \mathbf{L}_t - \mathbf{L}_t^w$ *dispensable tuples* (for the given witness). We could remove the dispensable tuples from the log without affecting the policy at the current time. They may, however, become necessary in the future. If $t \leq t'$ then we write $\mathbf{L}_t \leq \mathbf{L}_{t'}$ to denote the fact that $\mathbf{L}_{t'}$ is an extension of the log from time t to time t'.

Definition 4.1. *Let π be a query (Boolean or not). A set of tuples $T \subseteq \mathbf{L}_t$ is called* absolutely dispensable *for π, if for any future evolution of the log $L_{t'} \geq L_t$, $\pi(t', \mathbf{L}_{t'}, D) = \pi(t', \mathbf{L}_{t'} - T, D)$. We call $\mathbf{L}_t^w = \mathbf{L}_t - T$ an* absolute witness.

The Log Compaction Algorithm 2 examines each policy π in turn, and computes a witness $\mathbf{R}_{i,\pi}^w$ required by that policy, for each log relation $\mathtt{R}_i$. Then, it takes the union of all witnesses required by all policies, as well as by their subqueries occurring in the `FROM` clause. The heart of the algorithm consist of computing $\mathbf{R}_{i,\pi}^w$, the witness for

[7]If a new policy is added at time t, DataLawyer restricts its history to start at time t by adding extra predicates on the timestamp attributes.

Algorithm 2: Log Compaction Algorithm: `compact`

```
Input  : A set of policies $\mathbf{\Pi}$
Output: Witness tuples for each log relations $\mathtt{R}_i \in \mathbf{L}_t$
begin
1   foreach $\mathtt{R}_i$ do $\mathbf{R}_i^w \leftarrow \emptyset$
2   foreach $\pi \in \mathbf{\Pi}$ do
        foreach subquery $Q_i$ in $\pi$'s FROM-clause do
3           $(\mathbf{R}_1^w, \ldots, \mathbf{R}_m^w) \leftarrow (\mathbf{R}_1^w, \ldots, \mathbf{R}_m^w) \cup \mathtt{compact}(\{Q_i\})$
        foreach log relation $\mathtt{R}_i$ in $\pi$'s FROM-clause do
4           $\mathbf{R}_i^w \leftarrow \mathbf{R}_i^w \cup \mathbf{R}_{i,\pi}^w$ // See text.
    return $(\mathbf{R}_1^w, \ldots, \mathbf{R}_m^w)$
```

$\mathtt{R}_i$ required by the policy (or subquery) π. Computing a minimal witness is NP-hard in general [2], so we settle for heuristics, based on the structure of the policy π. If the policy has subqueries in the `FROM` clause, when we handle them separately. For example, to compute the witness for `SELECT ...FROM Subquery, R1, R2 WHERE ...HAVING ...` we compute separately the witness for `Subquery` and for the modified query $\pi =$ `SELECT * FROM R1, R2 WERE ...`, then union the witnesses. In the remainder of this section we assume that π is a policy without subqueries, and show how to compute a witness $\mathtt{R}_{i,\pi}^w$.

Note that setting $\mathtt{R}_{i,\pi}^w = \mathtt{R}_i$ always gives us a correct witness, which is equivalent to not doing any log compaction. In the remainder of this section we describe how to compute a smaller witness for certain policies π, starting with simple ones and successively generalizing to more complex ones.

No Clock, Full Query. Consider a query π that does not refer to the clock (current time), and also has no projections:

$$\pi = \mathtt{SELECT * FROM}\ \mathtt{R}_1, \ldots, \mathtt{R}_m, \mathtt{D}_1, \ldots, \mathtt{D}_q\ \mathtt{WHERE} \ldots$$

Here, $\mathtt{R}_i$, $i = 1, m$ are part of the usage log, and $\mathtt{D}_j$, $j = 1, q$ are part of the database, e.g. `Groups` in Example 3.2. Note that, while all policies in our system are Boolean queries, Def. 4.1 also applies to full queries and we start here by describing how to derive the witness $\mathbf{R}_{i,\pi}^w$ for a full query π. Let $\mathtt{R}_i$'s *neighborhood* $N(\mathtt{R}_i) = \{\mathtt{R}_{i_1}, \ldots, \mathtt{R}_{i_v}\}$ be the set of all other log relations that equijoin on the timestamp with $\mathtt{R}_i$, directly or indirectly. Note that this set may be empty.

Lemma 4.1. *The following queries define a witness for the policy π and relations $\mathtt{R}_i$:*

$$\mathtt{R}_{i,\pi}^w = \mathit{SELECT\ DISTINCT}\ \mathtt{R}_i.* \quad \mathit{FROM}\ \mathtt{R}_i, \mathtt{R}_{i_1}, \ldots, \mathtt{R}_{i_v}, \mathtt{D}_1, \ldots, \mathtt{D}_q\ \mathit{WHERE}\ \ldots \tag{2}$$

The FROM *clause contains* $\mathtt{R}_i$*, its neighborhood, and the database relations, and the* WHERE *clause contains all conditions in π that refer only to the relations in the* FROM *clause. If the same relation name $\mathtt{R}_i$ occurs multiple times in π (self-joins), then the witness $\mathtt{R}_{i,\pi}^w$ is obtained as the union of the queries (2), one for each occurrence of $\mathtt{R}_i$ in the* FROM *clause (see Example 4.4 below).*

(Proof sketch) We outline the proof that $\mathtt{R}_{i,\pi}^w$, in Eq. 2, correctly computes an absolute witness of relation $\mathtt{R}_i$ for policy π. We consider the possible cases. First, the case when $\mathtt{R}_i$'s neighborhood is empty: then the query simply selects those tuples in $\mathtt{R}_i$ that satisfy all predicates on $\mathtt{R}_i$ in the policy: obviously, all tuples that do not satisfy these predicates are dispensable for evaluating the policy, both now and in the future. Second, if the neighborhood is non-empty, then $\mathtt{R}_i$

is semi-joined with the other R_{i_j}'s (this is a semi-join reduction [25]): all other tuples are dispensable now, and are also dispensable in the future because all the R_{i_j}'s are joined on the timestamp, and no new tuples are being added at a current, or past timestamp.

No Clock, Boolean queries. We now generalize the algorithm to Boolean queries π, still without reference to the Clock. If π has a HAVING clause, then we drop the GROUP-BY and HAVING clauses, replace SELECT with SELECT * to treat the policy as a full query, and compute an absolute witness for the full query using Eq.(2); in other words, we do not take advantage of the fact that π is Boolean. Otherwise:
$\pi =$ SELECT DISTINCT 'Error' FROM $R_1, \ldots, R_m, D_1, \ldots, D_q$ WHERE ...
In this case we can compute a smaller witness than that for the full query. Let $N(R_i) = \{R_{i_1}, \ldots\}$, be the neighborhood of R_i, and denote X the set of all attributes of R_i occurring in a join predicate.

Lemma 4.2. *The following query (obtained by modifying Eq.(2)) computes an absolute witness to π:*

$$R^w_{i,\pi} = \textit{SELECT DISTINCT ON } (R_i.X), R_i.* \qquad (3)$$
$$\textit{FROM } R_i, R_{i_1}, \ldots, R_{i_v}, D_1, \ldots, D_q \;\; \textit{WHERE } \ldots$$

(Proof sketch) Recall that the DISTINCT ON statement in SQL nondeterministically chooses a single witness from an entire group of tuples. For example, SELECT DISTINCT ON (R.A), R.B FROM R chooses nondeterministically a value R.B for each distinct value R.A. In other words, the witness is computed by nondeterministically choosing any tuple that contributes to the output. Notice that the absolute witness is not unique: for each distinct value of X, the algorithm can choose any tuple with those values of the attributes X.

Adding the Clock. Finally, we consider queries (Boolean or not) referring to Clock; recall that this is done through an expression Clock c in the FROM clause. We assume all predicates on the clock are of the form c.ts op expression, where op is one of $<, \leq, >, \geq, =$, in other words op cannot be $\neq$; we apply a set of simple transformation rules to rewrite expressions like $u.ts > c.ts - 5$ into $c.ts < u.ts + 5$. We don't perform log compaction on policies that have an inequality operator, $\neq$, on the clock. Furthermore, we replace every equality predicate $c.ts = expr$ with $c.ts \leq expr$ and $c.ts \geq expr$.

Lemma 4.3. *Let π be a policy where all predicates on the clock are of the form c.ts op expr where op $\in \{<, \leq, >, \geq\}$. Then, an absolute witness can be computed by the query (Eq. 2) or (Eq. 3) (depending on whether π is Boolean or not), with the following modifications: (a) Drop predicates of the form c.ts > expression, (b) Replace every predicates of the form c.ts < expression (or $\leq$), with currenttime$+1 <$ expression (or $\leq$), where currenttime is a constant that represents the current value of the clock.*

Notice that a time-independent policy π_{ind} (Section 4.1.1) will return an empty witness, in other words it does not contribute anything to the log. Indeed, such a policy contains the predicate $c.ts = R_i.ts$, which is rewritten to currenttime $+ 1 \leq R_i.ts$, which evaluates to false because all new tuples in R_i have the current time-stamp.

(Proof sketch) Without dropping or modifying the predicates referring to the clock, the expressions (Eq. 2) or (Eq. 3) will compute a witness, but not necessarily an absolute witness; we note that, for (Eq. 3), all attributes occurring in expression of c.ts < expression are included in the DISTINCT ON attributes $R_i.X$ (they are considered as occurring in a join). By dropping the predicate c.ts > expression we increase the set of witnesses, and ensure that we also include all witnesses in the future, when c.ts will be larger. Similarly, by modifying c.ts < expression to currenttime $+ 1 <$ expression we drop all tuples that will be dispensable starting with the next time stamp.

We illustrate log compaction with two examples.

Example 4.3. *Continuing Example 4.2, we show how DataLawyer computes the absolute witness for the query P2b in Example 3.2. First, transform the policy into a full query. Since the join is on the timestamp, the neighborhood of Users and of Schema includes the other: $N(Users) = \{Schema\}$ and $N(Schema) = \{Users\}$. Moreover, we update the predicate on time. Therefore, our system computes the absolute witness for Users as:*

```
SELECT DISTINCT u.*
FROM Users u, Schema s, Groups g
WHERE u.ts = s.ts and s.irid = 'patients' and u.uid = g.uid
  and g.gid = 'Student' and u.ts > currentTime + 1 - 1209600
```

In other words, we only record users from 'Student' and only if they have issued a query on Patients in the last 14 days, less one time unit. Note that other policies compute their own absolute witnesses for Users: the system takes their union. Similarly for Schema:

```
SELECT DISTINCT s.*
FROM Users u, Schema s, Groups g
WHERE u.ts = s.ts and s.irid = 'patients' and u.uid = g.uid
  and g.gid = 'Student' and u.ts > currentTime + 1 - 1209600
```

Example 4.4. *We now illustrate how we do log compaction for P1_OPT in Example 4.1. Notice that this is a DISTINCT query, and has a self-join, so the absolute witness is obtained by taking the union of two queries:*

```
Schema_w:
  (SELECT DISTINCT ON (p1.ts), p1.*
   FROM Schema p1, Schema p2
   WHERE p1.ts = currentTime+1 and p2.ts = currentTime+1
     and p1.ts = p2.ts and p1.irid = 'Navteq' and p2 != 'Navteq')
  UNION
  (SELECT DISTINCT ON (p2.ts), p2.*
   FROM Schema p1, Schema p2
   WHERE p1.ts = currentTime+1 and p2.ts = currentTime+1
     and p1.ts = p2.ts and p1.irid = 'Navteq' and p2 != 'Navteq')
```

This query, however, returns the empty set, because all occurrences in Schema have the timestamp strictly less than currentTime. As a consequence, if this were the only policy, then the system will not generate any log at all.

4.2 Policy Minimization

Next, we focus on the policies themselves, $\mathbf{\Pi} = \{\pi_1, \ldots, \pi_k\}$ and describe optimized ways to compute them.

4.2.1 Interleaved Policy Evaluation

Recall that a policy i is satisfied when π_i returns false. Hence, a query can proceed when all π_i return false. We make two observations. First, by far the most common case is when the policies evaluate to false. This is the normal use of the database, when users ask queries that comply with the policies: our main goal is to speed up this case. Second, if a policy π_i returns false (the common case), it is often for a simple reason, for example because some part of π_i is false, e.g. one predicate or a join of only two relations; it suffices to find a partial expression of π_i that evaluates to false, then we do not need to compute the entire policy. Based on this intuition we develop the following optimization.

We review two standard definitions. (1) A policy query π is *monotone* if, for any two instances $\mathbf{L} \subseteq \mathbf{L}'$ and $D \subseteq D'$, we have $\pi(t, \mathbf{L}, D) \subseteq \pi(t, \mathbf{L}', D')$. All SPJU queries, and Boolean queries with aggregate conditions of the form `having count([distinct] x) > k` are monotone. In contrast, conditions of the form `having count(...) < k` are non-monotone. (2) Given two policy queries π, π' we say that π is *contained* in π' if, for all $\mathbf{L}, D$, $\pi(t, \mathbf{L}, D) \subseteq \pi'(t, \mathbf{L}, D)$. Since policies are Boolean queries, we denote containment by $\pi \Rightarrow \pi'$, which means that if π is true then π' is necessarily true (but not the other way around).

Let $\mathbf{S} \subseteq \mathbf{L}$ be a subset of the log relations. The *partial policy* for π and $\mathbf{S}$, in notation $\pi_{\mathbf{S}}$, is the policy obtained from π by simply removing all references to relations in $\mathbf{L} - \mathbf{S}$ and also removing the `having` condition if it refers to any relations in $\mathbf{L} - \mathbf{S}$. That is, the partial policy performs only the joins on the relations in $\mathbf{S}$ and in the database D. Note that the query is always syntactically correct. We prove the following:

LEMMA 4.4. *Suppose π is a monotone policy without aggregates. Then, for any partial policy, we have $\pi \Rightarrow \pi_{\mathbf{S}}$. The same holds for a monotone policy with aggregates, if all relations in $\mathbf{L} - \mathbf{S}$ are joined on their keys.*

PROOF. (Sketch) First, if π is a Conjunctive Query without aggregates, then, there exists a query homomorphism from $\pi_{\mathbf{S}} \rightarrow \pi$, which maps every atom of $\pi_{\mathbf{S}}$ to the same atom in π; by the classic result on conjunctive query containment [32] we conclude that $\pi \Rightarrow \pi_{\mathbf{S}}$. For queries that have an aggregate condition `having count(...) > k` we note that the relations in $\mathbf{L} - \mathbf{S}$ cannot raise the count, because they are joined on their keys. $\square$

Algorithm 3: Interleaved Policy Evaluation

```
input  : A set of monotone policies $\mathbf{\Pi}$ and a query $q$
output: true if a violation occurs, false otherwise
begin
    $\mathbf{S} \leftarrow \emptyset$
    for $f_i \in \mathbf{f}$ do
        // Update the log $R_i$ with its increment
        // (including timestamp t)
        $R_i \leftarrow R_i \cup (\{t\} \times f_i(q, D))$
        $\mathbf{S} \leftarrow \mathbf{S} \cup R_i$
        for $\pi_k \in \mathbf{\Pi}$ do
            $\pi' \leftarrow \pi_{k,\mathbf{S}}$
            if $\pi'(t, \mathbf{S}, D) = \emptyset$ then $\mathbf{\Pi} \leftarrow \mathbf{\Pi} - \{\pi_k\}$
        // See §4.3, Improved Partial Policies
        if $\mathbf{\Pi} = \emptyset$ then break;
    return $\mathbf{\Pi} \neq \emptyset$
```

Based on the lemma, Algorithm 3 describes an optimized strategy for evaluating a set of monotone policies $\mathbf{\Pi}$. We add one by one the log relations R_i to the set $\mathbf{S}$. At each step, we compute the log function f_i to obtain all new tuples added by the current query q to $\mathtt{R}_i$, then check for all policies π_k the conditions that refer just to the current log relations in $\mathbf{S}$: if any such policy returns false, we remove it from $\mathbf{\Pi}$. Next, we add a new log relation R_i to $\mathbf{S}$ and iterate. We stop when all log relations have been added to $\mathbf{S}$. In §4.3, we present an extension to interleaved that also allows us to terminate when $\mathbf{\Pi}$ becomes empty. If $\mathbf{\Pi} = \emptyset$, then it means that all policies have been found to be false (the common case); otherwise, there was at least one violation.

An important decision is in which order to add the log relations R_i to $\mathbf{S}$. Our current system uses a fixed order, which is chosen experimentally, offline, by optimizing over an existing log. In our prototype implementation, the order was experimentally found to be: `Users` followed by `Schema` followed by `Provenance`.

EXAMPLE 4.5. *Consider policy* `P2b` *from Example 3.2, and suppose we add the log relations in this order to* $\mathbf{S}$*:* `Users`, `Schema`*. Then we obtain two partial policies in addition to the full policy* `P2b`*:*

```
P2d: SELECT DISTINCT 1
     FROM Groups g, Clock c
     WHERE g.gid = 'Student'
P2c: SELECT DISTINCT 1
     FROM Users u, Groups g, Clock c
     WHERE u.uid = g.uid AND g.gid = 'Student'
       and u.ts > c.ts - 1209600
     HAVING COUNT(distinct u.uid) > 10
P2b: SELECT DISTINCT 1
     FROM Users u, Schema s, Groups g, Clock c
     WHERE u.ts = s.ts and s.irid = 'patients'
       and u.uid = g.uid AND g.gid = 'Student'
       and u.ts > c.ts - 1209600
     HAVING COUNT(distinct u.uid) > 10
```

The system starts optimistically by checking the first partial policy, `P2d`*; if there are no users in the* `Student` *group, then the policy is guaranteed to be satisfied. Otherwise the system proceeds with the second partial policy* `P2c`*, which checks if at least 10 users from that group have asked any queries in the last 14 days; if there are no such users then the policy is satisfied. Only if there are such users does the system proceeds with the full policy* `P2b`*.*

4.2.2 Policy Unification

Finally, we describe a simple, but very effective optimization, which consolidates multiple policies with the same structure but different constants into a single policy that uses a separate table for the constants. Variants of this technique have been employed in different settings in prior research [37] and is an example of transforming *queries into data*. We explain this technique through an example.

EXAMPLE 4.6. *Consider parameterized policies:*

```
Px = SELECT DISTINCT 'Error' FROM Users u, Groups g
     WHERE u.uid = g.uid AND g.gid = X
     HAVING COUNT(distinct u.uid) > 10
```

Here `X = {'Student', 'Postdoc', ...}`*. We unify them in a single policy:*

```
P1 = SELECT DISTINCT 'Error'
     FROM Users u, Groups g, Constants c
     WHERE u.uid = g.uid AND g.gid = c.const
     GROUP BY c.const
     HAVING COUNT(distinct u.uid) > 10
```

The new table `Constants` *contains all constants* `'Student'`, `'Postdoc'`, `...` *used in the policies.*

4.3 Advanced Optimizations

We briefly outline two advanced optimizations that extend the previous optimizations. Both apply to policies where all log-generating functions join on the timestamp.

Preemptive Log Compaction. The optimization is to compute the partial query `LCQ'` for the log compaction query

LCQ using only the logs that have been generated. If LCQ' is empty, LCQ would also be empty, so we might as well not generate the remaining logs.

Improved Partial Policies. For interleaved execution (§4.2.1), we only stop early, if a partial policy is satisfied (*i.e.*, produces the empty output). But we can do better. If the partial policy produces a non-empty output, and that output does not depend on the latest increment to the logs (at the current time), then no tuple from the current timestamp would contribute to the output of the policy. But since the policy was tested to be valid in the past, it will continue to be valid in the current timestamp. We formally define and evaluate this technique in our technical report [2].

4.4 Putting It All Together

DataLawyer puts all optimizations together as follows: **Offline Phase.** Perform the following static analysis on the policies:

1. Apply policy unification (§4.2.2). Denote $\mathbf{\Pi}$ the resulting set of policies.
2. For each time-independent policy $\pi \in \mathbf{\Pi}$, replace it by its optimized rewriting π_{ind} (§4.1.1). Let $\mathbf{\Pi}_{mon} \subseteq \mathbf{\Pi}$ denote the set of monotone policies (§4.2.1).

Online Phase. For each query q, perform the following actions, in order.

1. Run the Interleaved Policy Evaluation Algorithm 3 on the monotone policies $\mathbf{\Pi}_{mon}$. If it returns `true`, abort and make no changes to the usage logs. Else, let $\mathbf{L}_{gen}$ denote the log relations computed by the Algorithm.
2. For each non-monotone policy $\pi \in \mathbf{\Pi} \setminus \mathbf{\Pi}_{mon}$, compute the log relations R not yet in $\mathbf{L}_{gen}$ and add them to $\mathbf{L}_{gen}$, by applying the corresponding log function f. Then, evaluate π. If any policy returns `true`, abort.
3. Run log compaction (Algorithm 2) over $\mathbf{\Pi}$. Recall (§4.1.2) that only time-dependent policies contribute anything to the log. As a further optimization, do Preemptive Log Compaction (§4.3) to prune out policies that do not require log compaction.
4. Flush log to disk. Execute q.

5. EVALUATION

We evaluate the overhead of policy checking with DataLawyer compared to only executing queries in PostgreSQL (§5.1). We also study the performance of DataLawyer's optimizations compared to the NoOpt strategy: Log compaction (§5.2), time-independent policies (§5.3), interleaved policy evaluation (§5.4), and policy unification (§5.5).

We run all experiments on the MIMIC-II dataset, which is an anonymized dataset of readings from advanced Intensive Care Unit patient monitoring systems for over 33000 patients, collected over a period of seven years. The subset of data we experiment on is over 21GB in size. All experiments are conducted on a single server running PostgreSQL 9.2 over OS X 10.9.4, equipped with a 2.7 GHz Intel Core i7 processor and 16 GB DDR3 RAM.

The experimental setup consists of enforcing the policies in Table 2, which are adapted to our dataset from the policies we introduced in Table 1. We experiment with two users, with `uid = 0` or `uid = 1`. The users repeatedly submit one of four distinct queries as shown in Table 3. The query times range from $0.25ms$ to approximately $2s$.

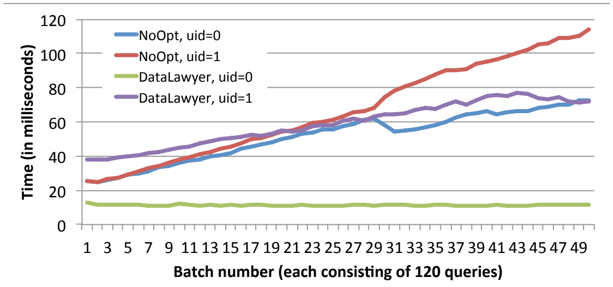


Figure 1: Policy and query evaluation time for DataLawyer and NoOpt on policy P_6 and query W_1 (fastest). DataLawyer's overhead stabilizes while NoOpt's grows continuously and quickly exceeds DataLawyer's overhead. Queries are submitted in batches of 120. The x-axis shows the batch number. The y-axis shows the average query and policy evaluation time for each batch.

5.1 Overhead of DataLawyer with All Optimizations Enabled

We first address the fundamental question of DataLawyer's overall practicality: What is the overhead of policy evaluation with DataLawyer compared with plain query evaluation with PostgreSQL and compared with NoOpt?

To answer these questions, we execute (multiple times) each query from Table 3 while enforcing one policy at the time. This lets us measure the performance of DataLawyer (with all optimizations turned on) and NoOpt, both with increasingly expensive policies and with increasingly expensive queries. We measure the query execution time, the overhead of tracking usage, the overhead of evaluating policies, and additionally, for DataLawyer, the overhead of compacting the logs. Further, for each policy-query combination, we run the experiment as user 0 (where DataLawyer can quickly infer through interleaved policy evaluation that no policy is applicable) and as user 1 (where the policies must eventually be evaluated in full to determine compliance).

Figure 1 shows how policy checking overhead grows continuously for NoOpt, while it quickly stabilizes to an approximately constant overhead for DataLawyer. The figure shows what happens for query W_1 and policy P_6 but the same trends occur for all policies and queries. In fact, for user 0, query W_4, and policy P_4, the overhead is $14ms$ for DataLawyer, while exceeding $2.7s$ for NoOpt after just 10 queries, leading to an almost 330× reduction in overhead. The cause for the growing overhead with NoOpt is the increasing usage history. DataLawyer's log compaction optimization prunes the parts of the log that are no longer needed, keeping the overhead constant after an initial ramp-up period. Of course, this pruning initially adds overhead compared with NoOpt.

We consider the overhead of policy evaluation in more detail, focusing on W_4 (long query) and W_2 (short query). For NoOpt, because the overhead grows, we measure the overhead after the first and tenth query for W_4 and the first and 400^{th} query for W_2. For DataLawyer, we measure the overhead once it stabilizes. Figure 2 shows the results.

As seen, for long queries (W_4) and cheap policies (P_1 and P_2), the overhead of policy checking is negligible for both DataLawyer and NoOpt[8]. For short queries (W_2), even for

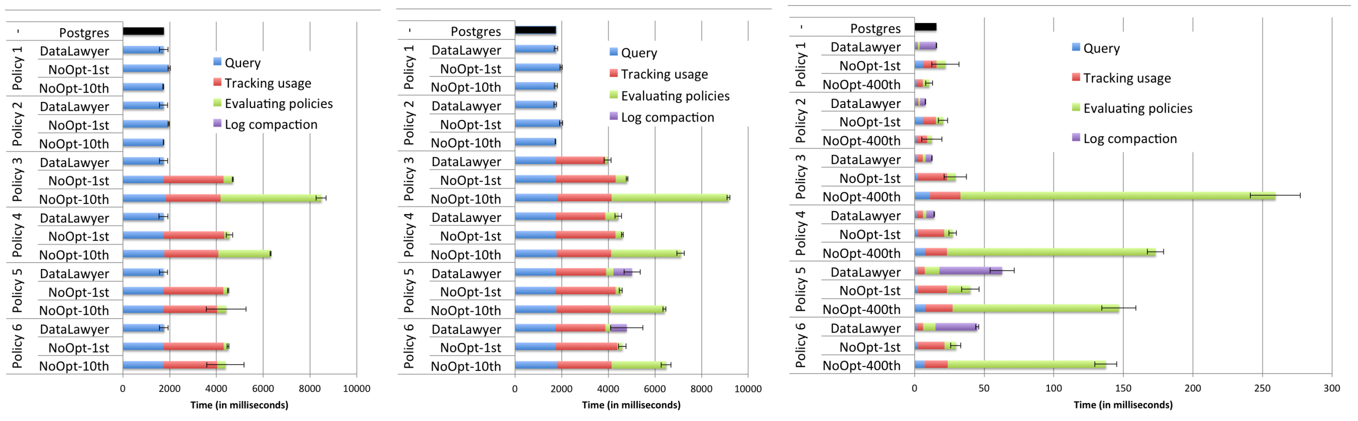
[8]For policies 1 and 2, it appears as if the later evaluations are faster

	Policy
P_1	A maximum of 10 `distinct` users can query the database from the group of users from university `'X'` in any window of `200ms`
P_2	User with `uid = 1` can not `join poe_order` with any other relation except `poe_med`
P_3	User with `uid = 1` can not execute any query on relation `d_patients` that returns more than 100 tuples
P_4	No output tuple on a query over `chartevents` for `uid = 1` should have less than or equal to 3 input tuples contributing to it
P_5	In no span of `3s`, aggregated over all queries, can user with `uid = 1` produce output that uses more than half the total tuples in `d_patients`.
P_6	In any span of `300ms`, user with `uid = 1` should not use the same input tuple from `d_patients` more than `1000` times.

Table 2: The policies used in the experiments. P_1 uses the cheapest log-generating function (`Users`). Note, group `'X'` contains user 1 but not user 0. P_2 uses both `Users` and `Schema` log-generation functions. The remaining policies are the most expensive policies and use the `Provenance` log-generating function.

	Time	Query
W_1	$0.25ms$	`SELECT * FROM d_patients WHERE subject_id = 186`
W_2	$15.69ms$	`SELECT c.subject_id, p.sex, COUNT(c.subject_id) FROM chartevents c, d_patients p WHERE c.subject_id = 489 AND p.subject_id = c.subject_id AND itemid = 211 GROUP BY c.subject_id, p.sex HAVING COUNT(c.subject_id) > 1`
W_3	$170.43ms$	`SELECT c.subject_id, p.sex, COUNT(c.subject_id) FROM chartevents c, d_patients p WHERE c.subject_id < 1000 AND c.subject_id > 930 AND p.subject_id = c.subject_id AND itemid = 211 GROUP BY c.subject_id, p.sex HAVING COUNT(c.subject_id) > 23`
W_4	$1756.6ms$	`SELECT c.subject_id, p.sex, COUNT(c.subject_id) FROM chartevents c, d_patients p WHERE c.subject_id < 1450 AND c.subject_id > 800 AND p.subject_id = c.subject_id AND itemid = 211 GROUP BY c.subject_id, p.sex HAVING COUNT(c.subject_id) > 1000`

Table 3: Queries used in experiments. Queries selected to cover a wide range of runtimes.



(a) (b) (c)

Figure 2: Policy and query evaluation time (in ms) for DataLawyer and NoOpt for all policies. Figures 2a and 2b show times for query W_4 for users with `uid=0` and `uid=1`, respectively; Figure 2c shows times for query W_2 for `uid=1`. The topmost bar is the query's evaluation time on an unmodified PostgreSQL (warm cache). DataLawyer's numbers were measured over a warm cache, once the overhead stabilizes. For NoOpt, we show the time for the first query (cold cache) and for the 10^{th} query (warm) for the first two figures, and the 400^{th} query (warm) for the last one. Error bars show the standard deviation over 12 runs for NoOpt and 50 runs (or more) for DataLawyer.

cheap policies, the overhead becomes visible. But, the policy checking overhead remains low (below 50 ms), maintaining the interactive speeds of these short queries.

Overheads become significant for the more expensive policies P_3 through P_6, which all use the `Provenance` log-generating function. As seen in Figure 1, the overhead with NoOpt grows quickly. For example, the total time taken to check and execute queries for policy P_3 increases by 1.8× and 1.9× for Figures 2a and 2b, respectively, between the first and the tenth query while increasing by 8.8× for Figure 2c between the 1^{st} and the 400^{th}. In contrast, DataLawyer maintains a significantly lower and constant overhead.

for NoOpt, but that is because the first query was over a cold cache. In the long run, this advantage vanishes.

These overheads have two components: the overhead of tracking usage and the overhead of evaluating the policies. For NoOpt, the overhead of the former is solely dependent on the usage logs mentioned in the policy definition. For a given query, the overhead is thus approximately constant across policies that use the same logs (*e.g.*, P_3 through P_6). For DataLawyer, the overhead of tracking usage is split into tracking the usage and compacting the log. Here, interleaved query evaluation and preemptive log compaction (§4.3) enable DataLawyer to avoid generating any usage logs in some cases such as for user 0 on query W_4 in Figure 2a. In other cases, DataLawyer generates the logs and stores them in temporary tables in memory. Unlike NoOpt, DataLawyer compacts these tables before writing anything to disk, which is why the overhead of tracking usage is smaller for Data-

Lawyer than for NoOpt even when both use the same logs, especially apparent for query W_2. Log compaction can add a significant overhead, which nevertheless pays off within the first 10 to 400 queries in this experiment. Log compaction's overhead is a function of existing log size, log increment size, and policy complexity. Hence, for policies P_1 and P_2 that rely on small usage logs, the overhead is tiny, while for policies P_5 and P_6, that rely on all three logs and require multiple joins and aggregations, the overheads are noticeable. Interestingly, in multi-threaded systems, one can return the result of the query to the user before log compaction finishes, thus the effective latency seen by the user may, in some cases such as for W_2 (policies P_5 and P_6), be as little as 23% of the time reported by a single-threaded system.

The overhead for evaluating policies is what dominates the overhead for NoOpt in later stages, while it stays constant and small for DataLawyer. Log compaction helps to keep the policy evaluation time small and constant for DataLawyer as illustrated in Figure 2b. Additionally, when applicable, as it does in case of user 0, interleaved evaluation allows DataLawyer to evaluate policies with practically no overhead; in our experiments, the maximum overhead over all policies and queries for user 0 was $3ms$ whereas the corresponding overhead for user 1 was $540ms$.

In spite of all its optimizations, for expensive queries, such as W_4, and for the most expensive class of policies, DataLawyer imposes a relative overhead up to 2× to 3× for `uid=1`. In general, for the latter user, a 100% overhead is unavoidable for policies P_3 through P_6, since they need the provenance, which is usually more expensive to generate than evaluating the query.

5.2 Log Compaction Optimization

As the end-to-end results show, log compaction (§4.1.2) is crucial for DataLawyer to maintain a constant overhead as more queries are executed against the database.

Log compaction removes dispensable tuples from the usage log and this reduces the time spent in policy evaluation for policies that are not time-independent. Note that both DataLawyer and NoOpt keep the newly generated usage log increments in memory, only pushing them to disk after verifying all policies. Thus, log compaction may also reduce the tuples from the latest log increment that are appended to the usage log, after checking each valid query. Here, we measure three phases of log compaction: (a) *marking*: the log compaction queries are executed over the disk-resident log and its in-memory increment, to determine which tuples to retain, and they are marked, (b) *delete*: the unmarked tuples are deleted, and (c) *insert*: the remaining tuples in the increment are appended to the log on disk.

To determine which tuples need to be removed DataLawyer must execute possibly multiple log compaction queries. Therefore, unless enough tuples are pruned this can be a significant overhead. Figure 3 shows the overhead of this optimization for three of the six policies: policies 1, 5, and 6 for the four queries by user with `uid=1`. No log pruning is needed for time-independent policies 2, 3, and 4 and hence they are not shown in the graph.

We explain what DataLawyer prunes for each policy: for P_1, the algorithm only retains the latest timestamp of the latest query, and only by the users in the group 'X'; for P_5, it only retains log entries for user 1's queries over `d_patients` in the window specified and only retains the latest instance

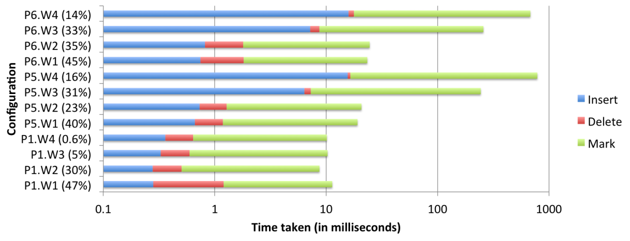


Figure 3: Overheads of Log Compaction: Time taken in the three phases of log compaction. Mark identifies the tuples to discard; Delete removes them; Insert appends the new tuples that were not marked for deletion. A configuration such as P6.W3 is interpreted as query W3 tested for policy 6. The percentages in parenthesis is the fraction of time spent in log compaction compared to the total policy checking and query evaluation time.

Count	P_2	P_2 - No ti	P_3	P_3 - No ti	P_4	P_4 - No ti
1	205	222	491	924	653	1355
5	198	208	473	881	643	2732
10	202	211	471	900	685	5110
15	212	229	480	949	648	8046
20	199	210	475	894	655	11809

Table 4: Policy and query evaluation time (in ms) for DataLawyer after executing multiple counts of query W_3 with time-independent policies 2, 3, and 4. Runtimes are reported with and without (represented as "No ti") the time-independent optimization. In both cases, all other optimizations are enabled.

of the tuples accessed by the user; and for P_6, it prunes out the tuples outside the sliding window.

For policies 1, 5, and 6, and for all queries, the bulk of the overhead of compacting the usage log is the "marking" stage where the tuples to be retained are selected. The high overhead is because 3 passes are made over the usage logs: the first to unmark all tuples, the next to compute the log compaction query which generates a set of tuple ids to retain, and the third pass to mark these tuples for retention. In contrast, NoOpt's overhead, apart from computing the usage logs, is about the same as the "insert" phase (since NoOpt does not prune any tuples, it may take slightly longer). Interestingly, as Figure 2 shows, in spite of such high overhead for pruning the log, the optimization pays off rapidly.

In our experiments, DataLawyer prunes the log after each new query. Such eager pruning, however, is not necessary. Instead, DataLawyer could compact the log less frequently or whenever the system has idle resources to further reduce the policy checking overhead.

5.3 Time-Independent Policies Optimization

We now test the impact of the time-independent optimization (§4.1.1) in the presence of the other optimizations. Recall that for this optimization, DataLawyer automatically adds extra constraints on the `ts` attribute, to enable log compaction to later prune the entire log. In fact, in our implementation, DataLawyer flags time-independent policies and never stores the log on disk in the first place thus completely avoiding any log-compaction-related checks and deletes.

Policies 2, 3, and 4 are time independent. Table 4 shows the time taken by DataLawyer to evaluate these policies on query W_3, once with this optimization on, and once without.

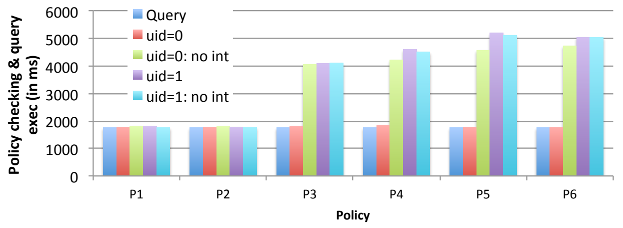


Figure 4: Policy and query evaluation time (in msecs) for each policy and query W_4. We consider two users, `uid=0` and `uid=1`, and for two versions of DataLawyer, one with all the optimizations and one with all optimizations but interleaved execution (indicated by "no int").

The primary benefit of the time-independent optimization over basic log compaction, is that this optimization allows DataLawyer to prune the log even for policies that involve aggregates but have no sliding time windows. For example, for policies 3 and 4, log compaction on the original policies without the added predicates on time does not prune any tuples. As a result, the overhead for both policies grows over time. The log compaction algorithm can not reason over aggregates and instead, we compact the corresponding full query, which ends up selecting all tuples for retention. In contrast, identifying the policies as time-independent permits the system to discard these tuples.

A secondary benefit is that, although the optimization can not avoid generating the logs, it avoids running the log compaction tests or appending any tuples to the disk.

Overall, in our experiments, this optimization halved the policy evaluation and query time for policies 3 and 4. Not much difference is seen in the case of policy 2 since it is a much cheaper policy to check and it produces a tiny log.

Thus, to ensure high performance, both optimizations, time-independence and log compaction prove beneficial.

5.4 Interleaved Policy Evaluation

We now evaluate the benefit of interleaved policy evaluation (§4.2.1). We quantify both the benefit when the optimization leads to early pruning as well the extra overhead when it does not. The overhead is due to executing multiple queries, each an approximation of the policy, instead of directly executing the original policy.

We evaluate each policy in isolation, for each query, once for user 0 and then for user 1. By design, interleaved execution prunes the policy after generating the cheapest log, `Users`, for user 0; and this provides an upper bound on its benefits. For the other user, interleaved execution only leads to an overhead with no pruning. For comparison, we provide the runtime with this optimization turned off.

Figure 4 shows that for user 0, interleaved policy evaluation can cut the runtime by more than half compared with not using the optimization. The resulting policy checking overhead drops to within 2.5% of the query evaluation time and remains nearly constant across all policies.

In user 1 though, interleaved execution does not lead to early pruning. DataLawyer without interleaved performs better than with interleaved. However, the differences are small. For the query shown, the maximum difference was of 1.7% of the runtime without interleaved. Even for the other queries (not shown), the differences are small, both in absolute as well as relative terms.

Another benefit of interleaved optimization is that in the presence of multiple policies, as opposed to a single policy as in this experiment, the benefits are additive, while the overheads are sub-additive. The benefits add up since the time saved by early pruning for each policy is independent of whether another policy is pruned or not, whereas the overheads are sub-additive since the log generation, which can be expensive as for policies 3, 4, 5, and 6 for `Provenance`, is done once and the output is shared across the policies.

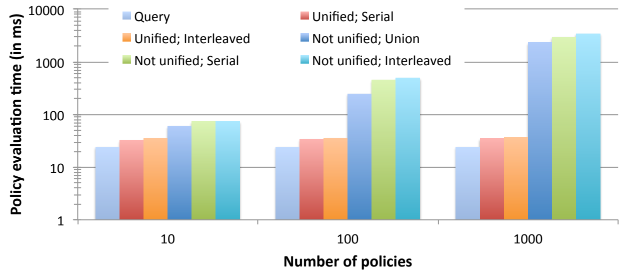


Figure 5: Policy Unification: A comparison of the average time to verify the policies (using either union, serial, or interleaved policy evaluation) and execute the query as we scale up the number of policies that can be unified to a single policy.

5.5 Policy Unification Optimization

We now answer how the time to evaluate policies change as we increase the number of policies, where policies are identical except for their parameters.

In this experiment, we check policy P_1 for query W_1. We vary the number of policies by two orders of magnitude by running three experiments: (a) 10 users each running 1000 queries (and a policy like P_1 for each user), (b) 100 users and 100 queries each (and a policy like P_1 for each user), and (c) 1000 users running one query each (and a policy like P_1 for each user). All queries execute in a round robin fashion. In this setting, the total number of queries executed remains constant but the number of users and thus the number of policies grows from 10 to 1000.

For the case when the policies are not unified, we compare three policy evaluation strategies : 1) **union**: Union all the (boolean) policies and execute one large policy, 2) **serial**: Execute policies one at a time, and 3) **interleaved**: Use interleaved execution (§4.2.1). For the case of the single unified policy, we compare serial (serial and union are identical with one policy) to interleaved.

Figure 5 shows that irrespective of the strategy, without unification, policy checking time grows linearly with the number of policies. The union is the most efficient because it avoids the overhead of multiple JDBC calls (serial takes from 23% to 87% more time to check the same policies); while interleaved is costlier (up to 16% for 1000 users) than serial since interleaved makes 2× more JDBC calls that serial for the specific policy used in this experiment.

On the other hand, unification of policies to a single one leads to a constant time to evaluate all the policies even after scaling up by two orders of magnitude. And this is *independent* of the policy evaluation algorithm used. This is because the unified policy introduces a relatively small dataset (with a maximum of 1000 rows in our experiments) to join with and that easily fits in memory.

Figure 5 validates the intuition that policy evaluation should be $\mathcal{O}(n)$, where n is the number of policies; while unification should lead to policies that are slightly more expensive to compute than an individual policy before unification, but that run in constant time irrespective of the number of merged policies.

6. DISCUSSION

DataLawyer's approach enables the expression of a wide variety of policies, but there are limitations to what can be expressed. We identify two limitations: First, boolean policies only allow accept/reject semantics. As a result, DataLawyer cannot support policies that require other semantics, such as for example creating a log entry when a violation occurs. Second, DataLawyer does not support policies defined over the actions of DataLawyer itself. Were that allowed, we could define a policy preventing DataLawyer from rejecting two successive queries from the same user leading to unenforceable sets of policies.

Another limitation of DataLawyer is that it cannot enforce *all* expressible policies *efficiently*. Currently, we only have full support for policies with monotone aggregate conditions (*e.g.*`having count([distinct] x) > k`, see §4.2.1), and only limited support for non-monotonic aggregates. Additionally, some policies are *"hard"* because they require storing a significant amount of history. An interesting area of future work is to use approximate policies to improve performance: The system first runs a simpler test that quickly validates most queries, but occasionally flags a valid query as suspicious and spends extra time to do the precise check.

A benefit of DataLawyer's approach is its potential extensibility to new domains by defining one or more new log-generating functions. These functions can be arbitrary pieces of code. We give two examples. First, consider a policy that restricts queries from 'mobile' devices to output sizes of 10 tuples. To enable such a policy in DataLawyer one has to write a new log-generating function that parses the database connection string or the user-agent headers and populates a new table in the usage log with device information; the policy itself is a simple SQL query over the new usage log. As another example, consider a tweak on policy P_4 from Table 1 to make it sensitive to the server load: "no user should be able to issue more than 50 requests per hour when the system load exceeds 80%." To implement such a policy one must, (a) define a log-generating function to populate the usage log with the current system load, and (b) write the corresponding SQL query.

Two important future research problems are both related to usability. The first is to help users debug queries that are deemed non-compliant. The other is to reduce the effort of translating text policies to our framework. Our survey indicated that there is a lot more structure to these policies and it may be possible to come up with templates (domain specific, if required) that can be later tweaked to get the set of policies for an organization. Policy generation by example might be another useful direction for future work.

Finally, we note that, while DataLawyer is an important step toward automatic enforcement of data use policies, it does not obviate the need for signed agreements and lawyers, because it does not control what happens to the data once it leaves the system.

7. RELATED WORK

Data auditing. Most auditing systems [41, 26, 40, 42, 34] detect data misuses, but only after the fact. In the online case, some prior techniques such as those that rely on reordering of queries [41] are not applicable; other techniques are data instance independent [42] and only make use of the structure of the queries themselves, unlike our semantics, which are data dependent.
Privacy Mechanisms. DataLawyer's goal is not to protect privacy, its goal is to verify that queries follow a pre-defined set of usage policies.
Access control. Access control approaches [26, 36] do not handle the case when users are allowed to see individual data items but do not have permission to perform certain operations, such as joins, on these data items.
Usage Models. The $UCON_{ABC}$ model [43], is a generic framework that models 16 usage control scenarios (such as UNIX access control lists and Digital Rights Management). DataLawyer subsumes six of those, the additional complexity is due to the expressiveness of the relational model.

DBAs may also automatically enforce performance related policies using Teradata'a Active Management System [16], but they do not have support for general data usage policies.
Complex Event Processing (CEP). Theoretically, some policies may be encoded as patterns for a CEP engine [33, 29, 47, 22] that can then search them in a stream of log increments as new queries arrive. Unlike CEP engines though, DataLawyer controls if and when to generate the log stream, which our experiments show to be a critical optimization. CEPs usually use non-deterministic finite automata to represent patterns. DataLawyer's policies are more general since arbitrary code is permitted for log-generating functions.
Multi-Query Optimization (MQO). Many techniques [27, 39, 28] for MQO identify common subexpressions in the query and then store the intermediate results to be reused by multiple queries. SharedDB [37] also provides a complementary set of techniques to ours. Although, DataLawyer can use these techniques, its primary performance improvements come from exploiting the fact that the policies are boolean. Further, in our setting, we also need to worry about regular and frequent updates to the underlying data (usage logs) that also provide opportunities for significant performance improvements.
Triggers. Triggers [23, 24] are only executed for DML statements and not for non-DML statements unlike the policies discussed in this paper.
Provenance Management. Provenance and annotation management [30, 31] store how data moves through databases over its life cycle. Their algorithms focus on reducing the provenance storage overhead and its querying. These techniques are orthogonal to our system, for which, provenance is just one possible log generating function.
User Interface. The interface that displays the message to the user and recommends alternative actions was demonstrated in an early prototype of our system [3].

8. CONCLUSION

We developed DataLawyer, a middleware system to specify and enforce data use policies on relational databases. Our approach includes a SQL-based formalism to precisely define policies and novel algorithms to automatically and efficiently evaluate them. Experiments on a real dataset from the health-care domain demonstrate overhead reductions of up to 330× compared to a direct implementation of such a system on existing databases.

9. REFERENCES

[1] Amazon Kindle. https://kdp.amazon.com/help?topicId=A2JGI9S4FDM39Q.

[2] Anonymized.

[3] Anonymized.

[4] DataMarket. https://datamarket.com.

[5] DataSift. http://datasift.com/terms/.

[6] Digital Folio. http://www.digitalfolio.com/Home/TermsOfService.

[7] Factual. www.factual.com.

[8] Foursquare terms of use. https://foursquare.com/legal/api/platformpolicy.

[9] Infochimps Data Marketplace. www.infochimps.com/Marketplace.

[10] Microsoft Translator. http://datamarket.azure.com/dataset/bing/microsofttranslator.

[11] MIMIC II. http://physionet.org/mimic2.

[12] Navteq. www.navigation.com.

[13] Rate data.gov.uk. http://www.nationalarchives.gov.uk/doc/open-government-licence/.

[14] Rate Limiting. https://dev.twitter.com/docs/rate-limiting/1.

[15] Socrata. http://www.socrata.com/.

[16] Teradata Active System Management. http://www.teradata.com/article.aspx?id=1602.

[17] Windows Azure Marketplace. http://datamarket.azure.com/.

[18] World Bank. https://openknowledge.worldbank.org/terms-of-use.

[19] Xignite. www.xignite.com.

[20] Yelp Display Requirements. http://www.yelp.com/developers/getting_started/display_requirements.

[21] DataLawyer Source Code. https://www.dropbox.com/sh/t0k3ajef57o8cjr/AACdfiZnPIqd_1nzsFdyOV_6a, 2013.

[22] Oracle Event Processing Language Reference. http://docs.oracle.com/cd/E14571_01/apirefs.1111/e14304/overview.htm#i1024819, 2013.

[23] Oracle: Fine Grained Auditing. http://www.oracle.com/technetwork/database/security/index-083815.html, 2013.

[24] PostgreSQL: Audit Triggers. http://wiki.postgresql.org/wiki/Audit_trigger, 2013.

[25] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[26] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, pages 143–154, 2002.

[27] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Materialized view and index selection tool for microsoft sql server 2000. In *SIGMOD Conference*, page 608, 2001.

[28] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[29] A. P. Buchmann and B. Koldehofe. Complex event processing. *it - Information Technology*, 51(5):241–242, 2009.

[30] P. Buneman, A. P. Chapman, and J. Cheney. Provenance management in curated databases. In *In SIGMOD âĂŹ06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2006.

[31] P. Buneman, J. Cheney, W. C. Tan, and S. Vansummeren. Curated databases. In *PODS*, pages 1–12, 2008.

[32] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[33] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422, 2007.

[34] D. Fabbri, K. LeFevre, and Q. Zhu. Policyreplay: Misconfiguration-response queries for data breach reporting. *PVLDB*, 3(1):36–47, 2010.

[35] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, pages 51–62, 2014.

[36] E. Ferrari. *Access Control in Data Management Systems.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[37] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, Feb. 2012.

[38] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[39] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.

[40] R. Hasan and M. Winslett. Efficient audit-based compliance for relational data retention. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 238–248, New York, NY, USA, 2011. ACM.

[41] R. Kaushik and R. Ramamurthy. Efficient auditing for complex sql queries. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 697–708, New York, NY, USA, 2011. ACM.

[42] R. Motwani, S. U. Nabar, and D. Thomas. Auditing sql queries. In *ICDE*, pages 287–296, 2008.

[43] J. Park and R. Sandhu. The uconabc usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, Feb. 2004.

[44] F. Schomm, F. Stahl, and G. Vossen. Marketplaces for data: An initial survey. *SIGMOD Rec.*, 42(1):15–26, May 2013.

[45] V. Tannen. Provenance for database transformations. In *EDBT*, page 1, 2010.

[46] R. Y. Wang and D. M. Strong. Beyond accuracy: What data quality means to data consumers. *J. Manage. Inf. Syst.*, 12(4):5–33, Mar. 1996.

[47] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418, 2006.