The Power of Data Use Management in Action

Prasang Upadhyaya[1], Nick Anderson[1], Magdalena Balazinska[1], Bill Howe[1]
Raghav Kaushik[2], Ravi Ramamurthy[2], and Dan Suciu[1]
[1]University of Washington and [2]Microsoft
prasang@cs.uw.edu, nicka@uw.edu, {magda, billhowe}@cs.uw.edu, {skaushi, ravirama}@microsoft.com, suciu@cs.uw.edu

ABSTRACT

In this demonstration, we show-case a database management system extended with a new type of component that we call a *Data Use Manager (DUM)*. The DUM enables DBAs to attach policies to data loaded into the DBMS. It then monitors how users query the data, flags potential policy violations, recommends possible fixes, and supports offline analysis of user activities related to data policies. The demonstration uses real healthcare data.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Relational Databases

Keywords

Data Use Management, Access Control

1. INTRODUCTION

In many situations today, researchers and companies use data provided to them by a third party rather than generated internally. Such data may be collected and distributed directly [2, 3, 5] or through an intermediary such as the Azure Marketplace [1] or Infochimps [4]. The value of this data is derived not only from its information content, but in how it is used. As a result, data providers are interested in controlling data usage. For example, clinical data [17] is valuable for research, but must not be used in such a way as to expose the identity of patients; E-books for the Amazon Kindle must not be used for more than the contractual rental period [7]; or, some datasets are intended for use only with certain devices or may not be joined with other datasets [18].

Today, these restrictions on data are determined through ad-hoc negotiations and enforced contractually (*e.g.*, [11]), or perhaps through aggressive use of access control lists (ACLs). But ACLs only prevent access, not usage. Alternatively, in the context of private data, one may use differential privacy mechanisms to enforce privacy restrictions. Such mechanisms, however, only permit a limited budget of queries [16]. More importantly, ACLs and other security-oriented technologies tend to be too restrictive in contexts where the users are not malicious. In these situations, it is important to explain to the users why the operation they attempted was disallowed and make alternative suggestions, as opposed to simply throwing a "permission denied" error.

In recent work [21], we proposed a vision for a *Data Use Manager* (DUM). The DUM is a new approach to data management, which enables database administrators to restrict the operations performed on the data stored in a DBMS rather than controlling the access to the data itself. This model is quite versatile and allows one to easily express many sophisticated constraints on the use of data.

We demonstrate the first working prototype of a DUM and illustrate its functionality on the MIMIC-II dataset [17], which is an anonymized dataset of readings from advanced Intensive Care Unit patient monitoring systems and clinical data for over 33000 patients, collected over a period of seven years. Today, access to this dataset is regulated as follows: a potential user must register and must take a 3-hour online course that teaches the restrictions on data use such as the fact that users may not share the data with anyone or may not run queries to infer the identity of the patients. The user is then trusted to follow the restrictions.

In this demonstration, we show how the DUM relieves users from the burden of checking data use policies before manipulating the data, while continuing to provide a powerful method to control how the data is used. To achieve these goals, the DUM imposes no access control restrictions on the data. Instead, it enables data providers to set *declarative* data usage *policies* that spell-out the restrictions on how various datasets can be processed by data consumers. A DBA loads these policies into a DBMS together with the data. The DUM then enforces these policies automatically either online, as data consumers process the data or offline, in a post-hoc auditing session as illustrated in Figure 1.

In our technical report [20], we develop a model that supports data use management in a flexible manner within a relational DBMS and explore various design alternatives to efficiently implement DUM as a component of a DBMS. In this demonstration, we *explore the changes to the users' experience when working with usage-restricted data*:

1. We demonstrate a first DUM prototype implemented as a middleware over a relational DBMS.
2. Using this prototype and a concrete scenario based on real healthcare data [17], we compare the benefits and capabilities of protecting data (1) by relying on users to watch out for policy violations, (2) using standard

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

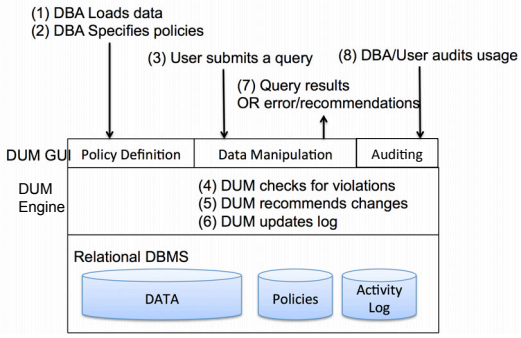

Figure 1: Overview of a DUM-enhanced DBMS.

ACLs, and (3) with a DUM. We demonstrate the use of the DUM both online and offline (*i.e.*, auditing).

3. We develop and present a novel explanation-based interface for data under DUM, where users query the database without any changes to their workflow. The system only prompts users when a violation is about to occur and provides the user with the violated policy, an *explanation* for the violation, and *recommendations* for adjusting their query to avoid the violation.

2. WHAT THE USER WILL SEE AND DO

We will take visitors to the demonstration through the following steps:

- *Policy specification:* We will briefly explain the MIMIC-II healthcare data management problem and will show how the data manipulation constraints can be stated precisely as policies on this dataset. In the DUM, policies are declaratively specified.
- *Policy enforcement:* Given these policies, the user will be asked to execute a series of six queries. Without the DUM, a user must manually inspect queries to determine if they do or do not violate any of the policies. With the DUM, the use can simply execute the queries and see if the system accepts them or not.
- *Explanations:* For rejected queries, the violated policies are reported, as well as evidence for the violation.
- *Corrective Actions:* The system also reports steps that a user might take to resolve the violation. For example, if a temporal constraint prevented the query's execution, it could be resolved by simply waiting and trying again later. Other constraints might require modifications to the query itself by adding clauses or changing parameters. The system will suggest such alternatives.
- *Offline Auditing:* Finally, we will show how a DUM can help audit data use offline, assessing both commonly-occurring and unusual data-use problems.

In addition to the DUM, we implement two DUM-aware client applications. The first enables users to execute queries and understand policy violations; the second enables the DBA to specify policies and audit any prior policy violations.

In this demonstration, instead of focusing on the efficiency of policy checking, we show how a DUM changes the experience of working with data and its benefits over using either ACLs or requiring that users manually check policies.

We now describe the system, major interfaces, and driving examples in more detail.

3. A DUM-ENABLED DBMS

We implement our DUM prototype as a middleware over a relational DBMS as opposed to a standalone client-side application. Our implementation is in JAVA and can work with any relational DBMS that provides a JDBC API. The first reason for this design choice is performance: by interacting closely with the underlying DBMS, the DUM can more efficiently manage the information necessary to enforce policies [20]. The second reason is data security. If instead of a middleware DUM module, the user were in charge of policy checking, then all queries executed with the goal of checking policy violations would themselves need to be verified.

Figure 1 shows the resulting high-level software architecture. Our prototype uses PostgreSQL as the underlying DBMS. To support the DUM, the DBMS is extended to store not only data but the policies that accompany the data. Additionally, because data use policies may need to verify the activity performed by multiple queries over time, the DBMS must also store an *Activity Log* (which we describe in more detail below).

The DUM exposes three sets of capabilities to users, each exposed as an extension to the SQL query language:

- *Policy Definition*: The DUM enables users (typically the DBA on behalf of the data owner) to specify declaratively the policies that accompany any new data loaded into the DBMS.
- *Data Manipulation*: The DUM must see all SQL queries submitted by users. For each query, the DUM verifies all policies for violations *before* executing the query. It reports problems and enables the user to ask for explanations of the problem and recommendations for avoiding the violation.
- *Auditing*: The DUM supports capabilities to audit data use, including examining queries that violated some policies, the recommendations that the system made, and the following actions by the users.

We now describe the above three functions in more detail.

3.1 Policy Definition

Users manipulate the data in a DBMS by issuing queries. Data use management thus puts constraints on the queries that can be executed over the data.

The DBAs specify the policies declaratively in SQL and these policies are stored in the database as a relation `policies` with schema `policies: (owner_id INT, policy TEXT, valid_until DATE)`. The policies are specified over a set of relations that capture various aspects of user activity and several key query features that together form the *Activity Log*. There is a single activity log for all the databases in a DBMS (instead of a log for each database separately) and the activity log's schema is known to all the data owners.

The activity log has the following schema. Each user has a unique id, `uid`; each query has a unique id, `qid`; each existing and generated tuple has a unique id, `tid`; each relation has an id `rid`; and each existing or generated column has an id `cid`. The activity log is composed of the following relations:

```
Executes(qid, uid, time)
Schema(qid, cid_o, cid_i, rid_i, agg)
Provenance(qid, tid_o, rid_i, tid_i)
```

`Executes` stores who executes which query at what time; `Schema` stores which column `cid_i` from relation `rid_i` con-

tributed to output column `cid_o` and if `cid_o` in an aggregate; and, `Provenance` stores the *where-provenance* for each output tuple `tid_o` by recording the input tuple `tid_i` from relation `rid_i` that contributed to it.

For performance reasons, the DUM does not materialize the entire Activity Log [20] but only small subsets of that log as necessary for checking existing policies.

Table 1 shows two concrete examples of declarative policies, P_1 and P_2, for the healthcare scenario. In the demonstration, we also use the following three policies.

P_3: Only select-count queries are allowed and each count result must be at least 10. This rule is a common usage agreement in exchanging clinical data [8]. It protects against triangulation attacks when a user needs to compare patient counts at different institutions.

P_4: Each query on the dataset must include a certain column for all queries executed on the dataset. Such a column might be used to ensure proper attribution of queried data.

P_5: At most 5 distinct users may query the `chartevent` relation. This policy is inspired from the e-book lending restrictions from Amazon.

Table 1 also shows three of the six demonstrated queries. The first five queries are extracted from the SQL handbook that accompanies the MIMIC-II dataset, while the last one is a query we made up.

Policy Definition Assistance. Apart from making the activity log's schema public to all data owners and the DBA, the DUM middleware performs a number of actions when a new policy is defined: It checks the policy for syntactic correctness; it evaluates if another pre-defined policy makes the new policy redundant; it checks if an existing policy is inconsistent with the current policy (for *e.g.*, if one policy restricts any query over `d_patients` to a provenance of less than 10 input tuples while another policy requires all aggregations over `d_patients` to be over at least 10 values, they contradict each other); and it provides the schema of the policy-specific diagnostic information that would be collected in case of a violation.

3.2 DUM-monitored Data Manipulation

Once policies have been defined, users can submit queries over the data. Whenever a user (or application) submits a query, the DUM intercepts the query and verifies that it does not violate policies. To check if a policy is violated, the basic idea is for the DUM to (1) start a transaction, (2) execute the query but do not return the result, (3) populate the activity log based on the query execution, (4) execute all policy queries on the updated log, and, (5) if any policy query returns *true*, rollback the transaction and flag the violation; otherwise, commit the transaction and return the result to the user. We describe more efficient policy-checking methods in our technical report [20].

The DUM thus supports more expressive policy definitions and hence permits a larger number of queries as compared to using ACLs[1] as shown in Figure 2. It is also seen that the DUM enables several powerful capabilities to assist users in fixing their rejected queries. Note that with both DUM and ACLs, it is only when queries violate a policy that the user experience changes leading them to investigate the violation.

[1]To map our policies to access control rules, we check if a policy prohibits any operation on a relation, if so, access control prevents the access to that data source for our users.

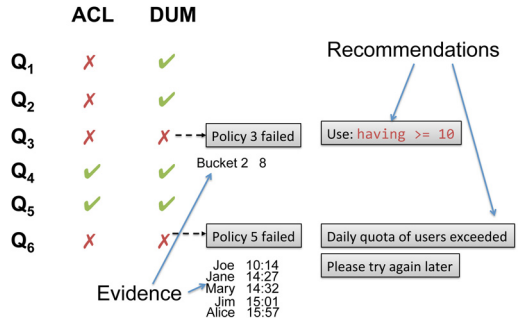

Figure 2: Cropped screenshot of DUM-aware client application after the execution of six queries.

3.2.1 Evidence and Policy Violations

For each rejected query, the DUM returns one or more policies that the query violated. Further, the DUM also generates a subset of the data or a subset of the Activity Log that provides evidence for the failure. For example, in Figure 2, the five users who had already queried the database form the evidence for a new user's query's failure due to P_5. To get this evidence, the DUM executes a modified version of the policy evaluating query that generates a set of tuples that provide evidence for the violation. Of course, the policy setter decides if users may see the evidence.

To support the above mentioned feature, the DUM populates a special relation, `violations`, with the disallowed query, `qid`, and the `policy` that was violated. Further, the `evidence` relation stores the tuples-ids and relation-ids of different relations that constitute the evidence for the policy violation. Both relations can be queried by the client to know which policy failed and why.

3.2.2 History-based Recommendations

When a query violates a policy, the DUM generates recommendations for how to resolve the problem. First, the policy setter may provide a troubleshooting message while defining the policy. For example, the troubleshooting message for policy P_5 might inform the user that the daily maximum of distinct users might have been achieved.

Second, DUM makes use of queries executed in the past to speed up the process of inferring if the existing query might lead to a violation. In case DUM rejects a query, we leverage the algorithms and data-structures used for this functionality to provide a list of previous queries that are similar to the existing query and that were deemed legal given our policies. In addition to showing the correct queries, the DUM also suggests modifications to the current query that would make the query policy-compliant. For example, if query Q_3 violates policy P_3, DUM may recommend a `having` clause to rectify the problem. Query similarity is based on query features, following our earlier work on SnipSuggest [15].

To illustrate the history-based recommender, we begin the demonstration with a large history of compliant user queries.

3.3 Offline Auditing

The DUM collects the activity data, including violations,

Table 1: Three out of the six queries and two out of the five policies that will be demonstrated.

Query	Description	SQL
Q_1	Patient's ID, sex, and date of birth	`SELECT subject_id, sex, dob FROM mimic2v26.d_patients WHERE dob < '2521-12-07' AND dob > '2304-09-13';`
Q_2	Count the number of patients in the database	`SELECT COUNT(*) FROM mimic2v26.d_patients;`
Q_3	Serum HCO3 Histogram	`SELECT bucket, COUNT(*) FROM ( SELECT width_bucket(valuenum, 0, 231, 231) AS bucket FROM mimic2v26.labevents WHERE itemid IN (50022, 50025, 50172) ) GROUP BY bucket ORDER BY bucket;`
P_1	No query over `d_patients` may return more than 200 data items. This affects queries Q_1 and Q_2. This policy is inspired from the restriction on Navteq data regarding display on low resolution devices.	`NOT EXISTS ( SELECT p.qid FROM Schema s, Provenance p WHERE s.rid_i = 'd_patients' AND s.qid = p.qid GROUP BY p.qid HAVING COUNT(DISTINCT(tid_o)) > 200 )`
P_2	No query may join the relation `poe_med` with any other relation except `poe_order`. This policy also comes from the Navteq terms of use.	`NOT EXISTS ( SELECT * FROM Schema l, Schema r WHERE l.qid = r.qid AND l.rid_i='poe_med' AND r.rid_i <> 'poe_med' AND r.rid_i <> 'poe_order')`

and stores it in the DBMS for auditing. The client application that we demonstrate aggregates that information and generates reports. We use these reports to show how offline auditing can help spot problematic policies. For example, if policy P_5 consistently prevents certain queries, perhaps the user has added a new collaborator but forgot to inform the data seller. Or if P_2 is often violated, it might suggest that a particular dataset generates value when joined with a different dataset, and P_2 should be re-negotiated.

4. RELATED WORK

The DUM is related to, and generalizes, several existing data management tools and techniques that restrict or control the access to data. These systems can be classified along two axes: type of semantics, and time of action. Access controls [12, 6] are ubiquitous mechanisms in database systems and simply restrict access to sensitive data; their semantics is precise (grant/deny), and their action is performed online (at query time). The limitation of these approaches is that they do not handle the case when users are allowed to access individual data items but do not have permission to perform certain operations, such as joins, on these data items. Privacy mechanisms such as differential privacy [9] also aim to restrict access, but are fuzzier since they restrict access to individual records while allowing access to aggregate queries. These techniques are also insufficient for a DUM because they often protect privacy by limiting data access too much, hence affecting its utility [22]. Finally, auditing systems [6, 14, 13, 10] are designed to detect data misuses after the fact, which is in contrast to the DUM which works both in an online and offline settings. Triggers [19] are inadequate since they only apply to DML statements (all our example policies are non-DML statements) and can not validate policies across multiple databases. For a more detailed discussion of related techniques, we refer the readers to our vision paper [21] and technical report [20].

5. CONCLUSION

We demonstrate a database management system extended with a Data Use Manager (DUM). Attendees interact with a database storing real, healthcare information, which is subject to constraints (a.k.a. policies) on how it can be used. The demonstration contrasts the user experience when (1) relying on users to ensure no policies are violated, (2) using access-control lists, and (3) using the DUM.

6. ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation and Microsoft through NSF CiC grant CCF-1047815 and NSF grant IIS-0915054 and additional gifts from Microsoft Research.

7. REFERENCES

[1] datamarket.azure.com/.
[2] gnip.com.
[3] www.aggdata.com/.
[4] www.infochimps.com/marketplace.
[5] www.patientslikeme.com.
[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, pages 143–154, 2002.
[7] http://www.amazon.com/gp/help/customer/display.html?nodeId=200549320.
[8] N. Anderson et al. Implementation of a deidentified federated data network for population-based cohort discovery. *Journal of the American Medical Informatics Association*, 2011.
[9] C. Dwork. A firm foundation for private data analysis. *Commun. ACM*, 54(1):86–95, 2011.
[10] D. Fabbri and K. LeFevre. Explanation-based auditing. *PVLDB*, 5(1):1–12, 2011.
[11] Factual terms of service. http://factual.com/tos.
[12] E. Ferrari. *Access Control in Data Management Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
[13] R. Hasan and M. Winslett. Efficient audit-based compliance for relational data retention. In *ASIACCS*, pages 238–248, 2011.
[14] R. Kaushik and R. Ramamurthy. Efficient auditing for complex sql queries. In *SIGMOD*, pages 697–708, 2011.
[15] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.
[16] F. D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, pages 19–30, 2009.
[17] Mimic-ii dataset. http://physionet.org/mimic2.
[18] http://www.navteq.com.
[19] http://postgresql.org/docs/9.2/static/triggers.html.
[20] P. Upadhyaya et al. An efficient implementation of a data use manager. Technical report, University of Washington, 2012.
[21] P. Upadhyaya et al. Stop that query! the need for managing data use. In *CIDR*, Jan. 2013.
[22] G. Weber et al. The shared health research information network (shrine): a prototype federated query tool for clinical data repositories. *Journal of the American Medical Informatics Association*, 16(5):624–630, 2009.