Query-Driven Enforcement of Rule-Based Policies for Data-Privacy Compliance

Peter K. Schwab, Maximilian S. Langohr, Jonas Röckl, Demian E. Vöhringer,
Andreas M. Wahl, and Klaus Meyer-Wegener

FAU Erlangen-Nürnberg, Lehrstuhl für Informatik 6 (Datenmanagement)
{vorname.nachname}@fau.de, https://www.cs6.tf.fau.de/

Abstract. Data privacy is currently a topic in vogue for many organizations. Many of them run enterprise data lakes as data source for an ungoverned ecosystem, wherein they have no overview concerning data processing. They aim for mechanisms that require unsophisticated implementation, are easy to use, assume as little technical knowledge as possible, and enable their privacy officers to determine who processes which data when and for which purpose. To overcome these challenges, we present a framework for query-driven enforcement of rule-based policies to achieve data-privacy compliance. Our framework can be integrated minimally intrusive in existing IT landscapes. In contrast to existing approaches, privacy officers do not require profound technical knowledge because our framework also enables non-experts in evaluating data processing in SQL queries by intuitionally comprehensive, tree-shaped visualizations. Queries can be classified as legal or illegal regarding data-privacy compliance. We provide a domain-specific language for defining policy rules that can be enforced automatically and in real-time.

Keywords: Classification Rules · Data Privacy · Query-Driven

1 Introduction

Data privacy is a trending topic, especially due to the EU General Data Protection Regulation (GDPR) taken effect in May 2018 [3]. This law sets standards for companies regarding data processing as high as never before [4]. However, many companies are badly prepared and have invested only insufficient financial and human resources to fulfill the GDPR needs [12]. Especially ungoverned ecosystems around enterprise data lakes pose tremendous challenges concerning data-privacy compliance. While these common data storage and processing environments democratize access to data sources within an organization, there often has not yet been established a suitable infrastructure for monitoring data usage.

Problem Statement: To overcome these challenges, companies need guidance and support to transform their businesses to comply with the GDPR regulations in an efficient way. Data processing in today's world still happens in many cases via SQL. However, privacy officers frequently do not have even basic SQL knowledge because database (DB) users are becoming more and more non-experts [8]. Besides, SQL queries are often embedded in dedicated software applications. This means, they cannot be reviewed separately. Instead, privacy officers need even more comprehensive knowledge in this case because they also have to review the source code generating the queries. So the companies aim for mechanisms that require unsophisticated implementation, are easy to use, assume as little technical knowledge as possible, and enable their privacy officers to determine who processes which data when and for which purpose.

Contribution: We present a framework for query-driven enforcement of rule-based policies to achieve data-privacy compliance. Our approach enables real-time data-privacy compliance in data processing for already existing system landscapes. In order to trustworthily assess data processing, our approach focuses on the queries running on the companies' target DBs. Privacy officers need not be SQL experts for this assessment, because we visualize data processing for them. Visualization is based on a tree-shaped structure that is intuitionally comprehensive, and illustrates a query's whole data flow, not only its result.

The implementation of our prototype is work in progress. Our framework is minimally intrusive, since it does not significantly affect productive operation. It is inserted between end-users' SQL tooling and the target DBs, it intercepts submitted queries and logs them in our internal query repository. The privacy officers can classify logged queries as *legal* or *illegal* regarding internal data-privacy regulations by means of rules based on the nodes of the tree-shaped query representation. Newly submitted queries are checked in real-time against a pre-filtered, compatible set of all defined rules. As soon as a rule matches, the respective query is prohibited to run on the target DBs. This approach extends the framework for query-driven data minimization we presented in [10] and [11].

This paper presents a user story illustrating the basic idea and benefits of our approach (cf. sec. 2), sketches the framework's architecture and reference implementation (cf. sec. 3), and proposes an evaluation concept (cf. sec. 4).

2 Query-driven Rule-based Policy Enforcement

To explain the basic idea behind our approach and to illustrate its benefits, we take up the user story based on a clinical research scenario described in [10], and expand it by the process of query-driven rule-based policy enforcement. In this scenario, the hospital's employees access the target DBs of their clinical information system (CIS) directly via SQL-based tools. There are administration secretaries and doctors generating different reports for their purposes. And there is Alice, the hospital's privacy officer. As already described in [10], our system is inserted between the employees' tooling and the productive CIS target DBs. Our system intercepts and parses submitted queries, automatically derives related characteristics, like the query's runtime and processed schema elements, i. e. relations and attributes, and logs everything to our internal query repository. This repository is the base for our approach. This paper describes functionality that notably exceeds the repository's function range.

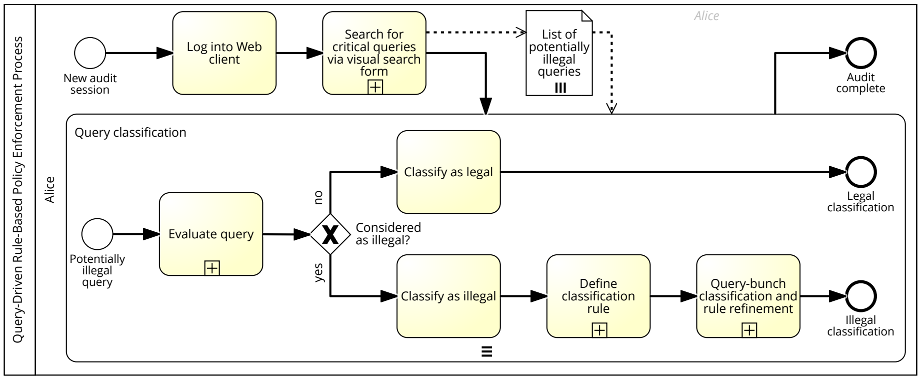
Query-Driven Rule-Based Policy Enforcement Process
Alice
Alice
New audit session
Log into Web client
Search for critical queries via visual search form
List of potentially illegal queries
Audit complete
Query classification
Classify as legal
no
Legal classification
Potentially illegal query
Evaluate query
Considered as illegal?
yes
Classify as illegal
Define classification rule
Query-bunch classification and rule refinement
Illegal classification

Fig. 1. Alice's Process for Query-Driven Rule-Based Policy Enforcement

Query Auditing: Now, we enhance this scenario because Alice also wants to ensure data-privacy compliance for the hospital's data processing. The BPMN diagram in fig. 1 illustrates her part in this user story. As she only has little technical knowledge regarding SQL, our framework supports her with some visualization tools. First, Alice performs an audit based on the queries logged in our internal query repository. She browses this repository for critical queries by use of the visual query search form in our Web client. It is introduced in [11] and allows for filtering queries by users and usage of schema elements. As she is well acquainted with the data stored in the CIS DBs, she quickly finds some critical queries that are potentially *illegal* regarding data-privacy compliance. Alice is no expert in understanding SQL. Our system supports her in comprehending a query's data processing by visualizing the data flow through the query based on an intuitionally comprehensible, tree-shaped structure. Every tree node represents an operation performed in this query and includes a verbal description of this operation. SQL functionality is reduced to the manageable set of relational operators and nested sub-queries are already resolved.

Rule Definition: Alice evaluates the critical queries in a row: If she considers a query to be in violation of any data-privacy regulation, she classifies it as *illegal*. In addition, Alice defines a rule that generalizes the query's parts containing the *illegal* data processing. She is supposed to define this rule because only she has adequate knowledge and skills to properly assess which parts of a query violate current data-protection regulations. Generalization of query parts means for example resolving query-specific renaming of schema attributes. To achieve this, rule definition is based on the visualized tree and its logical operator nodes.

To illustrate some examples for basic rules, we assume two tables A and B holding personal data of a larger group of people. Table A holds characteristics that easily allow to identify a certain person, whereas the characteristics in table B are equal for many of these people:

```
A( pseudonym, name, street, city, date_of_birth )
B( pseudonym[A], gender, race, professional_group )
```

Alice defines the following basic rules. They can be combined and nested arbitrarily to prevent any kind of illegal data processing. In a separate field of the rule-definition dialog, there can also be stated an explanation and a reason.

Rule #1: *Joining tables A and B via the pseudonym attribute is illegal because the result contains personal-related information.*

Rule #2: *The filter «B.gender = 'F' AND B.professional_ group = 'Civil engineering'» is illegal because result set is so small that it could allow drawing conclusions on personal-related information.*

Rule #3: *The filter «COUNT(*) < 5» after any grouping in table B is illegal because it can reveal possible filter criteria to generate queries with a small result set that could allow drawing conclusions on personal-related information.*

Rule Refinement: When Alice adds a new classification rule to our system, the data operations of the queries logged in the internal query repository are analyzed and the Web client displays a list of all queries matching with the rule definition. These queries are similar to the originally classified query in the sense that they contain the same logical operators working on the same schema elements as defined in the rule. Alice can filter these queries by several criteria, e. g. by use of certain schema elements, and hereby classify a bunch of queries in a single click. When needed, she makes use of the tree-shaped visualization to understand a query's data processing. If she comes across a query that matches a rule, but in spite of that, Alice considers it *legal*, she classifies this query respectively and refines the related rule definition, for example by adding further matching criteria. This increases the rules' quality and classification reliability.

Real-Time Policy Enforcement: As soon as Alice has defined the first rule, automatic real-time policy enforcement is enabled. The BPMN diagram in fig. 2 shows the interaction of the system's backend server with the end-users and all DBs involved in this enforcement:

When our system intercepts a submitted query, it not only logs it in the internal query repository, but also checks it against a pre-filtered, compatible set of all defined rules. If a single rule matches for this query, it is prohibited to run on the target DBs. Finally, instead of forwarding the query's result to the end-user's tool, a respective policy message including further information of the matching rule is sent. This guarantees an automatic and real-time policy enforcement for data-privacy compliance.

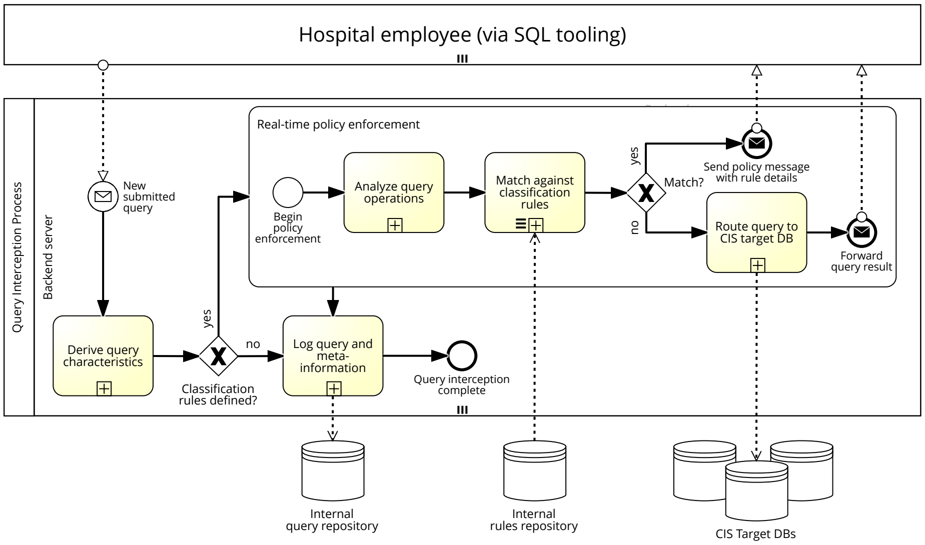
Hospital employee (via SQL tooling)
Query Interception Process
Backend server
New submitted query
Derive query characteristics
yes
no
Classification rules defined?
Real-time policy enforcement
Begin policy enforcement
Analyze query operations
Match against classification rules
yes
no
Match?
Send policy message with rule details
Route query to CIS target DB
Forward query result
Log query and meta-information
Query interception complete
Internal query repository
Internal rules repository
CIS Target DBs

Fig. 2. Interaction of the System's Backend Server with End-Users and Involved DBs

3 Architecture and Reference Implementation

This paper provides query-driven rule-based policy enforcement for data-privacy compliance. For this purpose, we enhance a framework for query-driven data minimization presented in [11]. Fig. 3 shows the underlying client-server architecture. The server is implemented in Java; the multi-user client uses contemporary Web technologies. Client-server communication follows a RESTful JSON API over HTTP using the Spring Framework[2]. Components added within this paper's scope are highlighted in the figure by a green background, components from external sources with dashed and thick boundary lines. The new components' functionality and their interaction with the whole framework are covered below.

3.1 Query Operator Analysis

The *Query operator analysis module* uses Apache Calcite[3] for parsing a query into a logical query plan. Calcite handles standard SQL and several of its dialects. Dialects not supported by Calcite, e. g. Presto, Athena, or Teiid[4], are also not supported by our system. Calcite obtains the required schema information directly from the target DBs. Our external DB interface supports all relational DBMS compliant to the JDBC API.

To ease further analysis, Calcite's query planner also performs a query normalization. This includes resolving multi-joins so that each node in the query

[2] `http://spring.io/projects/spring-framework`

[3] `https://calcite.apache.org/`

[4] `https://prestosql.io`, `https://aws.amazon.com/athena` and `http://teiid.io`

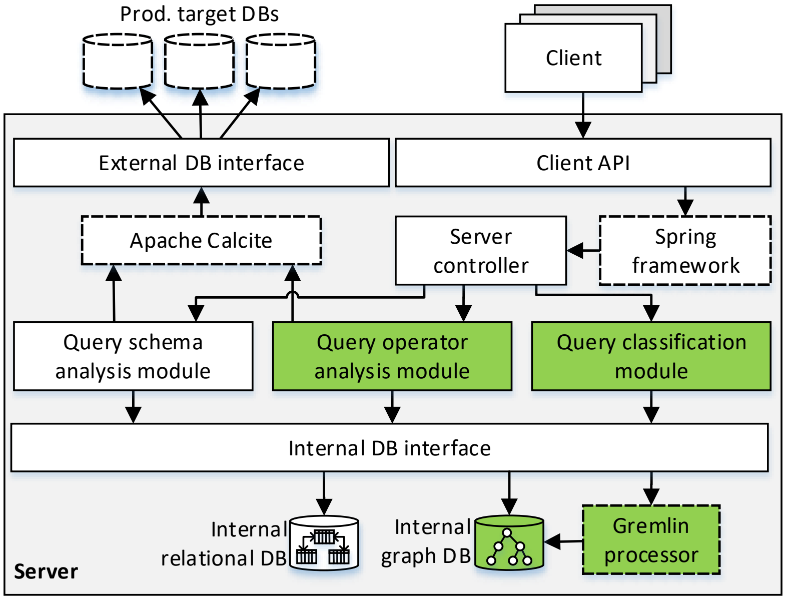

Fig. 3. The System Architecture of Our Framework

plan has at most two successors. This enables a uniform visualization of the tree-shaped query representation, which facilitates the evaluation of the query's data processing for the privacy officers.

By applying the visitor pattern, the resulting plan is traversed to gather helpful meta-information and to analyze its logical operator nodes. We support most of Calcite's provided node types and its variants. As a limitation, we do not parse for example calculations in restrictions and projections, but store them as simple text string. In our opinion, contents of these calculations are not relevant for a classification regarding data-privacy compliance. The same applies for *CASE* statements. Our approach also fails in uncovering any non-compliant data processing inside of stored procedures or user-defined functions because such calls are not resolved. This is out-of-scope of this paper.

Finally, the enriched plan is persisted in the internal graph DB via the internal DB interface. We are using Ferma[5] for the mapping of Java objects to elements in a graph. References between individual nodes of the plan are represented by edges in the graph DB. Edge labels are not needed because all relevant information is contained in the nodes. The edges are always directed from the root to the leaf nodes, according to the order direction of the node processing for the comparison of the query with the policy rules.

[5] http://syncleus.com/Ferma/

Query Plan Operator Meta-Information: Some of the gathered meta-information is persisted supplementary in the internal relational DB to speed up access to it. This includes the depth of each plan, the used relations, and the operator nodes. We use this information for pre-filtering of compatible query subsets to avoid checking the whole set of queries against the defined rules.

Additionally, for each operator node in the plan the following meta-information is stored: A textual description of the node and an equivalent SQL query retrieving the result data its respective subtree will produce. Privacy officers can display this information for a better understanding of the operator's functionality. Furthermore, the relations used by a node and their schema attributes are stored separately. Columns resulting from aggregation or renaming operations are resolved during the analysis, and the original schema attributes are stored in the node's meta-information to detect provenance for such columns easily.

3.2 Query Classification and Rule Definition

The enriched query plan is the basis for a query's classification as *legal* or *illegal* regarding data-privacy regulations. The *Query classification module* provides this functionality.

A query's classification by itself is just another DB string holding the respective information. In addition to this mere classification, privacy officers have to define a related rule that generalizes the related query's way of data processing. For rule definition, we propose a domain-specific language (DSL) especially designed for this purpose. We use the graph-computing framework Apache TinkerPop and its included graph traversal language Gremlin[6]. TinkerPop supports a variety of different graph DB implementations and Gremlin allows for user-defined DSLs. Our DSL supports all operators that can occur in the enriched logical query plan. Using a DSL increases flexibility and can easily be enhanced with further functionality. The same applies for graph DBs as new relationships can be added without having to restructure the DB schema. That is why we decided to integrate a collateral graph DB to the existing relational DB.

Rule Definition via DSL: Each rule definition starts with the `Query` statement, followed by one or several keywords separated by a dot. Each keyword describes a logical operator and can be specified by several optional parameters. The order of the keywords is relevant and will be considered for the graph traversals. We will explain our DSL with the following two example rules based on the rules #1 and #3 introduced in section 2:

```
Rule #1: Query.hasJoin( "A.pseudonym", "B.pseudonym" )
Rule #3: Query.hasGroupBy( ).hasFilter( "B.*", "<", "5", "COUNT" )
```

Rule #1 defines a graph traversal searching for queries joining tables A and B via the pseudonym attribute. The following queries are examples for a match to this rule:

[6] `http://tinkerpop.apache.org/gremlin.html`

```
> SELECT name, gender, race, professional_group
  FROM A JOIN B USING (pseudonym);
> SELECT *
  FROM A, B
  WHERE A.pseudonym = B.pseudonym;
> SELECT name
  FROM A JOIN B ON (A.pseudonym = B.pseudonym)
  WHERE gender = 'M';
```

Rule #3 defines a graph traversal searching for queries containing an aggregation at any point, followed by a filter. The aggregation keyword represents all `GROUP BY` clauses in SQL. Equivalents for the filter are all `HAVING` clauses. `WHERE` clauses do not apply in this case because they filter rows before the aggregation. The related filter condition says that the attribute `B.*` is used in the aggregation function `COUNT`. The resulting value has to be smaller than five. The following queries are examples for a match to this rule:

```
> SELECT COUNT(pseudonym), gender
  FROM B GROUP BY gender
  HAVING COUNT(pseudonym) < 5;
> SELECT COUNT(*), race, professional_group
  FROM B GROUP BY race, professional_group
  HAVING COUNT(*) < 5;
```

Rule Processing: The rules formulated in our DSL are stored as text in the relational DB. Gremlin converts this text string into a domain-specific traversal that can run on a graph DB. In this step, domain-specific keywords and their parameters are compiled into their associated Java objects and method calls. Finally, the enriched query-plan graph will be queried by the traversals defined in the rules. Up to now, we do not have implemented any graph-similarity or comparable algorithms. A query matches to a rule just if the domain-specific rule traversal provides a result for this query.

3.3 Real-time Policy Enforcement

To enable real-time policy enforcement, we integrate the JDBC proxy driver presented in [15] in our framework. This requires least configuration effort and is minimally intrusive as the proxy just wraps the native drivers of the target DB systems and enables supplementary functionality at runtime by using reflection techniques. The driver supports tools that use JDBC to connect to target DBs and is inserted between the end-users' SQL tooling and the target DBs. Fig. 4 illustrates its interaction with end-users, our framework, and the target DBs.

When an end-user's SQL tool sends a query to a target DB (step 1), our proxy intercepts this query and routes it to our framework's server application (step 2). We use Apache Kafka[7] as message-oriented middleware for communication.

[7] `https://kafka.apache.org/`

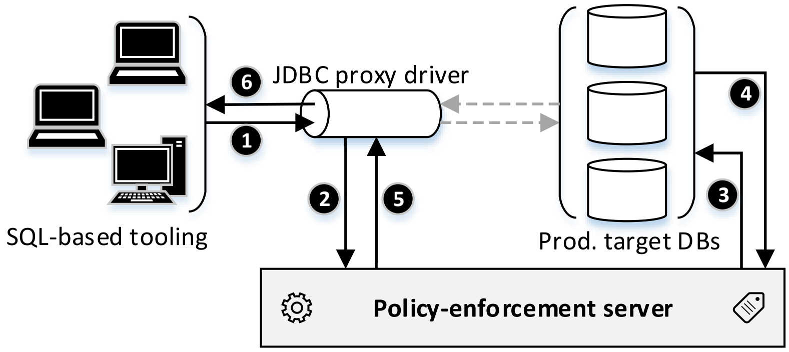

Fig. 4. Proxy driver interaction for real-time policy enforcement

The server parses the query and generates the enriched query plan. This plan is checked against all rules available in the system. If no rule matches, the query is assumed compliant to the data-privacy regulations and is forwarded to the target DB (step 3), which sends an appropriate response back to the server (step 4). Before this response is forwarded to the proxy (step 5), some of its meta-information (including runtime, number of tuples, and attributes in the result) is extracted and stored in the server's query repository. Finally, the proxy routes the response back to the SQL tool that has sent the initial query (step 6).

If one or several rules match the query, it is not sent to the designated target DB, because it is considered non-compliant to data-privacy regulations. Therefore, the steps 3 and 4 are skipped, and instead an appropriate message of type java.sql.SQLException is routed via the proxy (step 5) to the end-user's SQL tool (step 6).

Optimization of Query-Rule Comparison: In order to optimize the effort of comparing an incoming query with every existing rule for a match, we narrow down the number of qualifying rules by pre-filtering. We use the meta-information defined in section 3.1 to exclude rules that cannot apply for a certain query, e. g. rules that do not use the same relation and operator set like the query, or rules using more operators than the query. However, neither the operator order nor which operator uses which relation are considered in this step.

4 Evaluation Proposal

As mentioned in the previous section, our implementation is based on a proxy driver that intercepts the submitted queries and processes additional analysis steps. Since we build an interactive system that resides between the end-user and the target DB, it is inevitable that the execution of a query in general takes longer than a direct execution on the DB. The increase in the runtime of a query plays a decisive role for the acceptance of the whole system. In a followup evaluation, one could also consider the memory consumption, the network communication or other relevant quantities.

To achieve expedient runtime measurements, we adopt a two-sided approach:

First, we analyze the logical processing steps that are executed for each query. This includes parsing the query, analyzing its logical query plan, enriching this plan and persisting it in the internal query repository. We isolate these steps in a unit test. Each of these unit tests is executed repeatedly via the *Java Monitor Harness (JMH)* [8]. This tool has explicitly been built for nano/ micro/ milli benchmarks in Java and takes into account that modern computer systems behave indeterministically. The tool even tries to reduce these effects, e. g. via cache-warm-up runs. This procedure allows obtaining a good estimate of the cost ratios of individual logical processing steps. A comparison with the actual runtime of the query also gives an initial insight into the runtime overhead of our proposed system.

Since we develop an interactive application for query-driven rule-based policy enforcement, we do not rely on just measuring the runtime of idealized and isolated unit tests. Instead, we also measure the actual runtime of code parts used by an interactive call to our system under practical conditions. We plan to evaluate our framework based on the TPC-DS benchmark[9] queries. Hence, we utilize techniques of aspect-oriented programming [7] to insert small-footprint runtime measurement aspects into our application, while keeping the code clean. In contrast to the first evaluation part, this one explicitly ignores any indeterministic effects and therefore may produce less accurate results. However, we assume that this is a suitable method for demonstrating the eligibility of our interactive approach even under typical and practical (not idealistic) circumstances.

Because of our two-sided evaluation approach, we expect to be able to demonstrate that our system's runtime overhead is acceptable assuming ideal conditions (which means the algorithmic complexity of our analysis process is feasible), but it is usable without any restrictions even under practical circumstances.

5 State of the Art

In our literature search, we found several similar approaches. They can be grouped into approaches that identify illegal queries after their execution, i. e. auditing approaches, those that prevent the execution of illegal queries in advance and general-purpose query-log analysis tools.

Both Qapla [9] and DataLawyer [13] are systems designed for ad-hoc policy enforcement, allowing to specify policies in plain SQL. However, an important feature of our approach is that privacy officers need not have SQL knowledge, but can define policy rules for stored data in an intuitive and controllable way.

Regarding the auditing of SQL queries, Raghav Kaushik [6] and his colleagues have postulated essential compromises that are required to audit arbitrary SQL queries in respect to data-privacy compliance. In fact, this means to adopt a

[8] `https://openjdk.java.net/projects/code-tools/jmh/`

[9] `http://www.tpc.org/tpcds/`

weaker privacy model than the one we assume, and that does not comply with today's strict data-privacy regulations. To the contrary, we decided to not implement every single SQL feature, but then being able to prevent the execution of illegal queries at runtime. However, our subset of SQL instructions nevertheless covers most of the usual and common SQL keywords that are relevant for enforcing data-privacy compliance (cf. section 3.1).

Agrawal et al. describe the advantages of a user-friendly and intuitive specification of policies in [1]. However, their system is exclusively specialized in auditing, whereas we aim to enforce data-protection guidelines at runtime as well. The same holds for the approach presented in [5].

In addition to scientific approaches, we also found a commercial product, which offers tooling for the classification of data-lake accesses with regard to their data-protection compliance. *BigDataRevealed*[10] is a GDPR application solution and allows finding GDPR-regulated data in data lakes by searching for suspicious column names. A column "*Social Security Number*" may, for example, be classified as data that is protected by GDPR. However, we are not aware of any tool that is able to examine incoming requests and to deny a request if classified as illegal according to a given policy.

There are also several approaches for interactive SQL query log analysis (e.g. [14], [2]). While these projects provide broad general-purpose analysis capabilities, we focus on enabling non-experts to analyze SQL workloads.

6 Conclusion and Future Work

Our framework allows for automatic query classification concerning data-privacy regulations in real-time. In an initial audit phase, privacy officers classify appropriately selected queries from our internal query repository that intercepts queries submitted to the target systems. During this audit phase, the privacy officers also define policy rules for a data processing conforming to data privacy. Unclassified queries in the query logs (the "test set") are then classified semi-automatically in bunches with minimal user interaction. After that, our system is inserted minimally intrusively between the end-users' SQL tooling and the target DBs by a JDBC proxy DB driver. From now on, our system automatically enforces the defined privacy-policy rules on the intercepted queries. This guarantees a strict prohibition of data processing not conforming to data privacy.

The implementation of the functionality for parsing the queries and the rules engine for classification is practically completed. Our implementation handles also sophisticated queries and can enforce the defined policy rules. We are currently implementing the functionality to perform the proposed evaluation measurements for our approach. We are also integrating the proxy driver for real-time policy enforcement at this time.

There is also some future work to accomplish. We need to enhance our framework by the tree-shaped visualization mentioned above. It will support the privacy officers in evaluating the queries' data processing. Furthermore, we aim

[10] `https://gdprapplication.blog`

to provide a user interface for rule definition that supports a graphical building of rules by dragging and dropping desired logical operator nodes from the tree-shaped visualization.

Acknowledgement: The authors would like to thank the anonymous reviewers for their valuable remarks.

References

1. Agrawal, R., et al.: Auditing compliance with a hippocratic database. In: (e)Proc. 30th Int. Conf. on VLDB, Toronto, Canada, Aug 31 - Sep 3. pp. 516–527 (2004)
2. den Bussche, J.V., et al.: Towards practical meta-querying. Inf. Syst. **30**(4), 317–332 (2005)
3. Council of EU: Council regulation (EU) no 2016/679 of the European Parliament and of the Council of 27 apr 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/ec (General Data Protection Regulation). OJ **L 119**, 1–88 (2016)
4. Goddard, M.: The EU General Data Protection Regulation (GDPR): European regulation that has a global impact. Int. Journal of Market Research **59**(6), 703–705 (2017)
5. Goyal, V., et al.: A unified audit expression model for auditing SQL queries. In: Data and Applications Security XXII, 22nd Annual IFIP WG 11.3 Working Conf. on Data and Applications Security, London, UK, Jul 13-16. pp. 33–47 (2008)
6. Kaushik, R., et al.: Efficient auditing for complex SQL queries. In: Proc. ACM SIGMOD, Athens, Greece, Jun 12-16. pp. 697–708 (2011)
7. Kiczales, G., et al.: Aspect-oriented programming. In: ECOOP'97 - Object-Oriented Programming, 11th European Conf., Jyväskylä, Finland, Jun 9-13. pp. 220–242 (1997)
8. Li, F., et al.: Constructing an interactive natural language interface for relational databases. PVLDB **8**(1), 73–84 (2014)
9. Mehta, A., et al.: Qapla: Policy compliance for database-backed systems. In: 26th USENIX Security Symposium, Vancouver, BC, Canada, Aug 16-18. pp. 1463–1479 (2017)
10. Schwab, P.K., et al.: Towards query-driven data minimization. In: Proc. Conf. LWDA, Mannheim, Germany, Aug 22-24. pp. 335–338 (2018)
11. Schwab, P.K., et al.: Query-driven data minimization with the dataeconomist. In: Advances in Database Technology - 22nd Int. Conf. on Extending Database Technology, EDBT, Lisbon, Portugal, Mar 26-29. pp. 614–617 (2019)
12. Tikkinen-Piri, C., et al.: EU general data protection regulation: Changes and implications for personal data collecting companies. Computer Law & Security Review **34**(1), 134–153 (2018)
13. Upadhyaya, P., et al.: Automatic enforcement of data use policies with datalawyer. In: Proc. ACM SIGMOD, Melbourne, Victoria, Australia, May 31 - Jun 4. pp. 213–225 (2015)
14. Wahl, A.M., et al.: Crossing an OCEAN of queries: analyzing SQL query logs with oceanlog. In: Proc. of the 30th Int. Conf. on Scientific and Statistical Database Management, SSDBM, Bozen-Bolzano, Italy, Jul 09-11. pp. 30:1–30:4 (2018)
15. Wahl, A.M., et al.: Minimally-intrusive augmentation of data science workflows. In: Proc. Conf. LWDA, Mannheim, Germany, Aug 22-24. pp. 339–342 (2018)