Performance Evaluation of Policy-Based SQL Query Classification for Data-Privacy Compliance

Peter K. Schwab¹ · Jonas Röckl² · Maximilian S. Langohr¹ · Klaus Meyer-Wegener¹

Received: 21 May 2021 / Accepted: 4 August 2021 / Published online: 1 September 2021
© The Author(s) 2021

Abstract

Data science must respect privacy in many situations. We have built a query repository with automatic SQL query classification according to data-privacy directives. It can intercept queries that violate the directives, since a JDBC proxy driver inserted between the end-users' SQL tooling and the target data consults the repository for the compliance of each query. Still, this slows down query processing. This paper presents two optimizations implemented to increase classification performance and describes a measurement environment that allows quantifying the induced performance overhead. We present measurement results and show that our optimized implementation significantly reduces classification latency. The query metadata (QM) is stored in both relational and graph-based databases. Whereas query classification can be done in a few ms on average using relational QM, a graph-based classification is orders of magnitude more expensive at 137 ms on average. However, the graphs contain more precise information, and thus in some cases the final decision requires to check them, too. Our optimizations considerably reduce the number of graph-based classifications and, thus, decrease the latency to 0.35 ms in 87% of the classification cases.

Keywords Query classification · Policy rules · Data privacy · Performance

1 Introduction

In the Big-Data era, the amount of data generated and processed is still growing every day. While storing heterogeneous, large-scale data is not a problem anymore, data science is anxious to create new knowledge from that data. Even in situations where many different storage formats are used (in so-called “data lakes”), the desired evaluations are often coded in SQL as a query language, leaving the actual binding of table and attribute names to later stages

of the process [4, 30]. And with Apache Drill [9], an “open-source SQL query engine for Big-Data exploration”¹ is now available. So it is still quite common that users submit SQL queries to extract the desired information from the data. Each query has a specific purpose and therefore contains knowledge about how data should be processed to gain new insights. Not all these queries, however, may be compliant to the privacy regulations of an organization. This is one reason why a given set of queries should be assessed. Improving the writing of new queries could be another. Additional query metadata (QM), like related user names, result statistics, or the query context can be included to enhance the assessment for better results. For example, the same query can return different numbers of result tuples in different target systems, which allows to draw conclusions about individuals in one case, but not in the other. A query's purpose is also important for an appropriate assessment of data-privacy compliance. Processing personal data can be granted in the context of a scientific study, but not in the context of advertisement.

✉ Peter K. Schwab
peter.schwab@fau.de

Jonas Röckl
jonas.roeckl@fau.de

Maximilian S. Langohr
maximilian.langohr@fau.de

Klaus Meyer-Wegener
klaus.meyer-wegener@fau.de

¹ Computer Science 6 (Data Management), FAU, Martensstr. 3, 91058 Erlangen, Germany

² Computer Science 1 (IT Security Infrastructures), FAU, Martensstr. 3, 91058 Erlangen, Germany

¹ <https://drill.apache.org/docs/drill-introduction/>.

1.1 Problem Statement

Assessing SQL queries is not an easy undertaking, as there are various syntactic structures for equivalent queries, e.g. common table expressions instead of subqueries. Due to the enormous amount of contemplable queries, a manual assessment is far too time-consuming. QM can be extracted from the data-storage systems only with substantial effort. Assessment results remain mostly tacit knowledge in the heads of the users and are not stored in any way.

We need novel approaches to the assessment of SQL queries by automatic derivation of QM and classification of queries. Users should be enabled to browse these QM, classify queries based on their QM, and extend QM with their assessment results, all without needing profound technical knowledge.

1.2 Contribution

Previous work has already presented our extensible query repository (QREP) for policy-based SQL query classification using relational and graph-based data models to store the queries together with their QM [21–23]. In this paper, we present two optimizations that reduce classification latency. We evaluate the overall classification performance of QREP, show that long-running graph traversals (GTs) are the bottleneck in classification, and demonstrate how our optimizations reduce the classification latency by orders of magnitude, avoiding GTs in 87% of all cases. We outline related work in Sect. 2, summarize the functionality of QREP in Sect. 3, illustrate its reference implementation and the newly added optimizations in Sect. 4, provide the benchmark setup and results in Sect. 5, and conclude the paper in Sect. 6.

2 State of the Art

Recent research efforts related to our approach can be categorized as follows:

2.1 Query-Log Analysis

There are many approaches dealing with the derivation of QM based on query-log analysis. Some of them provide mechanisms to query the QM, but none supports policy definitions or query classification.

In [28], the authors map derived QM to a purely graph-based model. As their focus is on knowledge sharing, they additionally derive the queries' temporal and social context. A domain-specific query-filtering mechanism enables a comprehensive analysis. However, their long-running GTs slow down the system's performance. The commercial

tool Collibra² and [3] represent schema lineage graphically, but do not consider other QM. Another graph-based approach provides policies for query rewriting [20] but aims at a faster query execution.

Dedicated systems for query management are provided in [13]. Like the approaches with keyword-based searches over SQL query logs [14, 26], they extract only a very limited set of QM. For the semantics and underlying source schemas of SQL queries, different representations are provided e.g. in [11, 15, 19]. They do not support complete structural QM like schema lineage and completely lack contextual QM.

2.2 Privacy Languages

Privacy languages aim to be automatically processable [17]. In contrast to our approach, they have a broader focus than just enforcing a privacy-compliant data processing. We only deal here with the policies of systems that relate to *how* data is processed.

The *Data Capsule* connects personal data with privacy policies that allow only a certain data processing [29]. In contrast to our approach, these policies are not limited to SQL systems but must be defined by the users to whom the personal data relates. QREP provides more QM that can be included in the policies, e.g. environmental QM. In [24], a policy-specification language based on simple **ALLOW** and **DENY** clauses is presented. It organizes attribute values into concept lattices. Policies can generically describe privacy regulations. Again, our approach provides more QM for policy definition relating to how data is processed. The authors of [7] present a layered privacy language that can restrict the processing of personal data only for specific purposes. Although it can be easily extended, the policies cannot specify how the data may be processed.

2.3 Query Auditing

There are many approaches of auditing queries in order to prevent disclosure of personal data [18]. In [12], significant compromises are postulated, which are necessary to check arbitrary SQL queries according to data-privacy directives. However, the proposed privacy model does not comply with current regulations like GDPR. There are many approaches providing policies that require profound technical knowledge for definition and consider only a limited set of QM [2, 5, 6]. Also, Statistical DBMSs deal with the question of how one can effectively prevent that conclusions on individuals are drawn by queries [1, 25]. Related policies often barely consider QM and are hard-coded in the DBMS. Many approaches for online auditing of SQL

² <https://www.collibra.com/data-lineage>.

queries achieve a better performance than our approach [8, 10, 16, 27], but consider only a very limited set of QM in their policies. *BigDataRevealed*³ is a commercial GDPR application solution, which enables policy definition to classify data-lake accesses according to data-privacy directives by searching for suspicious column names. This approach also barely considers QM.

3 Policy-Based Query Classification

QREP analyzes queries and automatically derives QM that can be enriched with contextual information, e.g. the query purpose. Based on the QM, generic domain-specific policy rules can be defined to externalize tacit knowledge concerning data processing. QREP automatically classifies queries according to the policy rules and stores the result as contextual QM. All QREP parts have been illustrated in prior publications. We briefly summarize them for a better understanding of the optimizations and the benchmark presented in this paper.

3.1 QM Management

The first part is the management of three QM groups:

1. *Structural* QM can be derived automatically from the query itself by analysis of SQL query logs [22]. Examples are attributes accessed or operations performed, like aggregations and restrictions. To extract structural QM, we normalize a query's tree of relational-algebra operators using transformation rules consistent with relational algebra, so that a single tree covers many syntactical variants of semantically equivalent queries, for example, by harmonizing subquery and join expressions. We enrich this tree with schema information to reveal schema lineage [22]. We represent this group of QM as a multi-relational property graph and also in a relational schema. On the one hand, this enables the definition of policy rules with a simple descriptive syntax. On the other hand, these rules can still analyze the QM with short response times.
2. *Environmental* QM can be derived automatically from the query's target DB management system (DBMS), e.g. the related DB user, or result statistics like query runtime and the number of result tuples. We represent this QM group in a relational schema.
3. *Contextual* QM cannot be derived automatically because it is mainly based on tacit expert knowledge, for example, a query's purpose, its importance, its compliance regarding data-privacy directives, or its aptitude for hardware

acceleration. We represent contextual QM as a multi-relational property graph.

3.2 Domain-Specific Policy Rules

Privacy officers can externalize their tacit knowledge concerning data-privacy compliance in the form of policies based on Boolean conditional rules [23]. If a query matches a certain QM-based query pattern in the rule's condition part, this query is classified according to the rule's consequent part. Basic patterns can be combined by logical conjunction. A query matches the related rule only if the Boolean expression resulting from the pattern evaluation is true.

Each basic pattern either refers to exactly one QM entry or combines related ones. To write down the patterns, we provide a domain-specific language (DSL) that is not limited to our exemplary use case of data privacy and can be applied flexibly in a multitude of scenarios in which analysis of QM is beneficial. The DSL simplifies the access to the QM by encapsulating the choice of SQL queries for the relational schema and traversals of the multi-relational graph. All query patterns are mapped to a relational schema for persistence.

For example, the query in Fig. 1 displays a company's top earners. As they reveal personal data, the policy rule in Fig. 2 classifies such queries as *non-compliant* to data-privacy directives for users with role *developer*. Lines 2 and 3 check whether a query connects the relations *employees* and *salaries* by any means, and line 4 checks for the role of the user executing the query. Only if all checks are positive, the query matches the rule.

3.3 Automatic Query Classification

The classification process incrementally aligns the basic patterns in the policy rules with the QM. Once such a query/

```

1 SELECT  e.name, SUM(s.salary) AS salary
2 FROM    employees e
3 JOIN    salaries s USING (emp_no)
4 GROUP BY e.emp_no
5 HAVING  SUM(s.salary) > 120000

```

Fig. 1 Query to retrieve the top earners of a company

```

1 IF q.match(
2   connectsTable('employees')
3   AND connectsTable('salaries')
4   AND hasUserGroup('developer')
5 ) THEN q.classify('non-compliant')

```

Fig. 2 An exemplary policy rule

³ <https://www.gdpr.associates/partners/bigdatarevealed-bdr/>.

policy alignment is evaluated to false, the query does not match the policy rule and the alignments with the policy's remaining basic patterns can be skipped.

For data-privacy classification, QREP follows a blacklist approach. That means, each query is initially assumed to be *compliant*. When it matches a policy rule, then the query is classified as *non-compliant* [23]. Thus, QREP can classify a query by different policy rules without contradicting results. The classification result is included in the query's contextual QM.

QREP can also run a whitelist approach, where queries are initially assumed to be *non-compliant* and have to be "released" by policy rules. This would be more consistent with the principle of "privacy by design". In the following, we focus on a minimal-intrusive deployment of our system in existing IT landscapes. Therefore, a blacklist approach is more user-friendly because it allows policy rules to be added step-by-step without initially blocking all query results and paralyzing workflows.

Two basic use cases can trigger the classification:

1. *New-query alignment* checks a new query q that is inserted in QREP against all existing policy rules $r \in R_{QREP}$. If this query does not match any policy rule, it is assumed to be compliant. Otherwise, its contextual QM entry "data-privacy compliance" changes its value from initially "compliant" to the classification result "non-compliant".
2. *New-policy alignment* checks a newly defined policy rule r against all queries $q \in R_{QREP}$ that are stored in QREP. All matching queries are classified as "non-compliant". If a query has already been classified by another rule, the query's value of "data-privacy compliance" may already be "non-compliant". It is then just overwritten with the same value.

New-policy alignment is often more extensive and therefore induces more latency than new-query alignment. It re-

quires a higher number of alignments because $|R_{QREP}| \ll |Q_{QREP}|$.

4 Reference Implementation

QREP features a conventional three-tier client-server architecture, which is explained in detail in [21]. Fig. 3 shows its overall architecture. Data and application tier are server-sided; platform-independent multi-user clients using contemporary Web technologies form the presentation tier. The server is implemented in Java and employs the Spring Framework. Client-server communication uses a RESTful JSON API over HTTP.

4.1 qrep Modules

The business logic of QREP is encapsulated in three functionally independent server modules:

1. *QM Management* automatically derives structural and environmental QM. This module integrates Apache Calcite, which provides the functionality for parsing and analyzing SQL queries. Contextual QM can be maintained in the Web clients or indicated during policy-rule definition. ORM and OGM frameworks bridge the gap between in-memory Java objects and their respective physical representation. For now, we run PostgreSQL as the internal RDB and OrientDB as the internal graph DB (GDB).
2. *Policy-Rule Definition* has been built with the graph-computing framework Apache TinkerPop and provides our DSL. It is written in Groovy, a Java superset. Groovy allows easy adjustment and extension of DSL keywords. We use PostgreSQL also for the persistence of policy rules and all their features.

Fig. 3 The overall three-tier client-server architecture of QREP

3. *Automatic Query Classification* matches the QM against the policy patterns. If QM is stored in the internal GDB, the corresponding policy patterns are transformed into GTs via Gremlin, a DB-agnostic query language included in Apache TinkerPop. Classification results are passed to the *QM Management*, which persists them as contextual QM.

4.2 Optimization 1: Fingerprint for Quick Elimination

As first enhancement of our previous work, we have extended the query/policy alignment by an additional pre-filtering step based on a fingerprint to reduce classification latency. A fingerprint is a simplified, dense numerical representation of basic rule patterns and of QM stored in the internal RDB. It can be kept in main memory thanks to its small memory footprint. At the beginning of the classification process, QREP fetches all query and rule fingerprints in only one DB access each. This speeds up the query/policy alignment. The fingerprint replaces the names of schema elements and relational operations with their system-wide unique IDs. It does not contain every detail of the QM. For example, the theta operators involved in a filter predicate and its constant values for comparison are not included. If a pattern contains details like these, the alignment will always require a GT. Fig. 4 shows an excerpt of a fingerprint. The numbers in lines 1–2 represent quantities. All the other numbers represent sets of unique IDs of relational operations (line 6), relations (line 7), and attributes (lines 8–10).

If there is no respective basic pattern, the related value for the rule-fingerprint entry is *null* for quantities (line 3) and empty for sets (line 5). These values are not aligned with the respective query-fingerprint entries.

Eq. 1 illustrates the evaluation of a fingerprint check fc . An entry e of a rule fingerprint f_r or a query fingerprint f_q can be of type *quantity* or of type *set*. The i quantity entries of the rule fingerprint are compared for equality with

those of the query fingerprint. For the subsequent j set entries, it is checked whether the respective entry list of the rule fingerprint is a subset of the query fingerprint's entry list. Quantity entries are placed first in the fingerprints because they can be evaluated faster than set alignments. The main goal is to find a sub-condition that is false as fast as possible. In this case, the query does not match the rule and the whole alignment can stop immediately without any additional DB lookup. Otherwise, if the whole term evaluates to true, the classification process continues regularly and aligns the remaining basic patterns step by step with the query. For the QM that is not contained in the respective fingerprints, a GT is built and executed on the internal GDB.

$$fc(r, q) = \left(\bigwedge_{n=0}^{i-1} f_r[e_n] == f_q[e_n] \right) \wedge \left(\bigwedge_{n=i}^{i+j-1} f_r[e_n] \subseteq f_q[e_n] \right) \quad | \quad i, j \in \mathbb{N}_0 \quad (1)$$

Fingerprint evaluation speeds up the classification process by significantly reducing the number of long-running GTs, which increase latency enormously.

4.3 Optimization 2: Faster Classification Mode

Previous work only provided a “*full*” classification mode, i.e. a query was matched against every policy rule (in new-query alignment), or a rule was matched against every query (in new-policy alignment) to get a complete list of a query's data-privacy breaches. As the second optimization, we have added another classification mode named “*first*”. It shortens *new-query alignment* by exiting the classification process when a query first matches a policy rule. Further matches with other rules will not change the classification result anyway, due to our blacklist approach. Thus, the respective alignments can be skipped.

5 Evaluation

This section illustrates the benchmark setup and presents the evaluation results.

5.1 Benchmark Setup

Queries in QREP pass through different phases. To evaluate the latency induced by classification, we isolate the corresponding calls of the two phases *execution* and *classification* and measure their runtime. We capture latencies for both the fingerprint evaluation and the GTs. For this, we run the benchmark for each query/policy alignment, emulating a system where only this particular policy is active to

```

1  { "resultSize": 45,
2    "restrictions": 6,
3    "aggregations": null,
4    ...
5    "usesSetOperators": [],
6    "usesClauses": [119, 352],
7    "accessesTables": [7, 5, 13],
8    "projectsOn": [29, 32, 54],
9    "aggregatesOn": [54],
10   "groupsBy": [29, 32],
11   ... }
```

Fig. 4 QM representation in a fingerprint

prevent unwanted matches that tamper with our measurements. Based on this, we can calculate latencies for both new-policy alignment and new-query alignment, as well as for both “*first*” and “*full*” classification mode.

5.1.1 Micro-Benchmark

We use the Java Microbenchmark Harness (JMH)⁴, which allows to build, run, and analyze benchmarks of a set of methods in JVM-targeting languages. To reduce irregularities originating from indeterminism in both hardware (e.g. branch prediction and caching) and software (e.g. kernel-level and user-level scheduling), JMH repeatedly measures a method’s runtime relying on different layers of variation.

First, the runtime of a single *invocation* of a method is measured. JMH chains multiple invocations. They are executed serially one after the other and form a so-called *iteration* of ten seconds. In this time frame, the benchmarked method is invoked as often as possible. We perform eight of such iterations. Multiple concurrent *threads* can run the iterations. However, JMH expects to run an independent piece of code without side effects (e.g. an algorithmic computation). This does not hold for QREP, as we interact with several DBMSs. Thus, our benchmarks are single-threaded because multiple threads would concurrently generate workload on the same DBMS, which tampers with our measurements. JMH spawns eight *forks* to execute the benchmark threads. Unlike the threads, the forks are synchronized, so that each fork waits for its predecessor process to terminate before running the benchmarks. After all the forks have finished their workload, the results are aggregated. Assuming a normal distribution, JMH calculates the average runtime and the confidence interval.

5.1.2 Multi-Machine Setup

As we only use one thread per benchmark, we lose one degree of variation. To compensate for this, we add another degree of variation by executing the benchmark on eight physical machines in parallel before aggregating the results. These eight machines are identical in construction, based on an Intel Core i7-4790 running at 3.6 GHz, 32 GB of RAM, and 8 GB of swap memory. The OS is Fedora 29 with a GNU/Linux 5.2.7-100 kernel. The JVM implementation is OpenJDK 1.8.0_212.

5.1.3 Isolating Internal Data Stocks

We use Docker, a tool for OS-level virtualization, to containerize the internal and target DBMSs.

This allows us to flexibly spin up and tear down an isolated DBMS. In addition to the DBMS *images*, we maintain a set of Docker *volumes*. They encapsulate and isolate different data stocks that effectively originate from the QM derived during query parsing and analysis. To benchmark the classification process of a query, we inject a data stock including the corresponding QM. We use separate volumes encapsulating different data stocks for benchmarking. This allows us to repeatedly run the same benchmark on a well-defined input data stock. We spin up the DBMS layer, run the benchmark invocations repeatedly, and tear down the DBMS layer again.

In contrast to our internal infrastructure, we explicitly assume that the end-user queries run in a permanent IT landscape and have already been processed before. The hardware, the OS, the JVM, and the target DBMSs may have cached intermediate data. By this, query results can be delivered comparatively fast.

5.1.4 Benchmark Queries and Policies

This case study focuses on **SELECT** queries because they are typically used in data-science projects and contain the most knowledge about data processing. We enumerate the analytical queries from the established *TPC-DS*⁵ benchmark according to the TPC-DS query template IDs and partition them into three groups: short-running (< 150 ms), medium-running (< 1 s), and long-running queries (≥ 1 s).

We use a set of 38 carefully chosen policy rules constructed in a way that (a) any of the policy rules matches to at least one query and vice versa, (b) there are policy rules exclusively based on fingerprint evaluation, graph-based evaluation, and mixed evaluation, (c) there are policy rules that match short-running, medium-running and long-running queries, and (d) any query matches at most 15% of all policy rules.

Fig. 5 shows the basic patterns we use in the policy rules for our evaluation. As many operation names are similar to those of the relational algebra, most of them should be self-explanatory. We focus on basic patterns related to structural QM as their evaluation requires many and complex GTs. Eight basic patterns rely on QM stored in our RDB and can be evaluated completely fingerprint-based (lines 1–8). Three patterns rely on QM stored as a graph and can only be evaluated partially based on fingerprints (lines 9–13), i.e. a graph-based evaluation might be necessary.

5.2 Evaluation Results

We select 27 distinct short-running queries, 23 distinct medium-running queries, and 10 distinct long-running

⁴ <https://openjdk.java.net/projects/code-tools/jmh/>.

⁵ <http://www.tpc.org/tpcds/>.

```

1  accessesTable ( <table> )
2  accessesAttribute( <table>, <attr> )
3  projectsOn( <table>, <attr> )
4  aggregatesOn( <table>, <attr> )
5  groupsBy( <table>, <attr> )
6  usesSetOperator( <set operator> )
7  usesClause( <SQL clause> )
8  hasResultSize( <theta operator>, <size> )
9  connectsTable( <table> )
10 connectsAttribute( <table>, <attr> )
11 restrictsOn( <table>, <attr>
12             [, <aggregation>][, <operator>]
13             [, <value>] )

```

Fig. 5 Basic query patterns used in the policy rules

queries for evaluation. The shortest query execution time is < 4 ms, the longest > 4 min. To cover a broad range, we test policies that match exactly one query and queries that are matched by exactly one policy. In the same manner, we benchmark policies that match multiple queries and queries that are matched by multiple policies. Out of the 38 policies in the test set, 18 policies are based on fingerprint evaluation exclusively, 6 policies are based on graph-based evaluation, and 14 policies require both types of evaluation. The policy rules start with a single basic pattern to test individual patterns and continuously increase the number to 33 combined patterns. We test 2280 unique query/policy alignments, of which 236 (10%) result in a match. As we obviously cannot plot each of them, we aggregate the alignments by the different pattern combinations. Table 1 shows the number of pattern combinations for the respective rules and the matching queries.

5.2.1 Fingerprint Impact

Fig. 6 shows the average latency caused by the fingerprint evaluation, grouped by the pattern combinations. The average latency is < 0.35 ms and increases even the short-running queries by only 1%, which is negligible. Since only

10% of the queries are matching a rule, the non-matching queries substantially affect the total average latency.

If we consider only the query/policy alignments that result in matches, i.e. the evaluations where all fingerprint entries have to be aligned, we find that the fingerprint latency is a constant factor independent of the number of entries to be matched. For example, the runtime for a fingerprint filled from only one basic pattern (pattern combination one) is almost the same as that for a fingerprint filled from 35 basic patterns (combination fifteen). If we consider all query/policy alignments, i.e. those resulting in matches and those not resulting in matches, the latency decreases slightly, since for non-matching queries not all the fingerprint entries need to be aligned.

Fig. 7 shows the average proportion of how often the fingerprint evaluation shortens the whole alignment process. Its consequences for alignments resulting in matches are always zero, so this bar is not plotted.

Considering only the alignments not resulting in a match, the fingerprint shortens the alignment process in 95% for most groups and in 87% of all cases. This is not surprising, since fingerprint entries cover most of the widely used QM. However, the pattern combinations seven to nine deviate from the others. Each of them includes two policy rules. In five out of these six, we explicitly specify basic patterns containing QM details that are not contained in the related fingerprint, so that a GT is required. Only for one policy of combination seven, the fingerprint evaluation shortens the whole alignment in 98% of all cases, so that the average value considering both rules of this combination is a little bit less than 50%.

5.2.2 Total Classification Latency

Fig. 8 shows the average classification latency for a single query/policy alignment. A single GT causes a latency of 137 ms on average. The latencies induced by GTs predominantly influence the overall runtime, while the latencies caused by fingerprint evaluations carry no weight. Since

Table 1 The number of rules, their patterns, and the matching queries

# Combination No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Σ
# Relational Patterns	1	2	3	4	5	6	0	0	0	1	1	2	3	6	28	62
# Graph-Based Patterns	0	0	0	0	0	0	1	2	3	1	2	2	5	3	5	24
# Respective Rules	7	1	1	3	3	3	2	2	2	1	2	1	3	6	1	38
# Distinct Query Matches	60	7	6	3	3	3	15	6	15	11	10	3	4	7	1	154
# Total Query Matches	138	7	6	3	3	3	15	6	17	11	12	3	4	7	1	236

Fig. 6 The fingerprint-evaluation latency is a constant factor < 0.5 ms

Fig. 7 The fingerprint shortens the alignment process in 87% of all cases

the alignments not resulting in a match are mostly shortened by the fingerprint evaluation, they require significantly less GTs than the alignments resulting in a match, which results in a considerable latency reduction.

Especially the plotted bars of pattern combinations eight and nine are interesting because no fingerprint-based latency reduction is possible here. For combination eight, the alignments not resulting in a match have a longer runtime than those resulting in a match. This is because the GTs in the first case have to search the whole graph without finding a match, whereas the GTs in the second case can stop as soon as they have found a match. This also applies to combination nine. However, the related rules contain each three basic patterns requiring a GT, in contrast to combination eight with only two such patterns per rule. Since alignments not resulting in a match can stop as soon as the first GT does not find a match, these alignments have to

perform less GTs than the alignments resulting in a match, because the latter have to perform every GT. Thus, the number of the required GTs significantly increases the latency. In particular, this is the case for pattern combination thirteen, where up to five GTs are required. All of them must be done for alignments resulting in a match. However, we regard this increase of processing time for such queries as justifiable, as QREP classifies them as *non-compliant* and does not pass the query results to the end-users anyway.

Fig. 9 shows the average classification latency for a complete new-policy alignment. As expected, the pattern combinations of policy rules for which no fingerprint-based latency reduction is possible significantly increase the latency. In total, the latency of a complete new-policy alignment arithmetically averaged for all queries in “full” classification mode is 1430 ms. And the latency of a complete new-

Fig. 8 GTs determine the latency of a single query/policy alignment

Fig. 9 GTs significantly increase classification latency of new-policy alignment

query alignment arithmetically averaged for all policy rules in this mode is 906 ms.

Table 2 shows the percentage overhead to the query-execution runtime caused by the classification. The table shows the difference of the average overhead for the three runtime groups before and after our optimizations. For short-running queries, this latency is still in the orders of magnitude beyond the runtime. For long-running queries typical of analytics-query processing and targeted by QREP (e.g. business-report generation), latency is now negligible compared to runtime.

5.2.3 Classification-Mode Speedup

During new-query alignment in “full” classification mode, each query is aligned with each of the 38 policy rules. In our underlying use case of data-privacy compliance and

the corresponding blacklist approach, the alignment process can stop after the first match with a rule. Thus, the impact of “first” classification mode significantly depends on the number of alignment matches. The more often and the earlier a query matches a rule, the more alignments can be skipped.

To achieve a more balanced scenario, we drop the 14 policies with the highest match rate. This rate does not implicitly correlate with the number of utilized basic patterns. For example, our benchmark includes two rules utilizing only one pattern. One matches 56 queries and the other only one. The rule dropping leads to 24 rules remaining and reduces the total number of matches by almost 75% to 60 and the overall match rate from 10% to 4%. There are 31 compliant queries matching no policy rule and 29 non-compliant queries matching at least one rule. The latter must be aligned with 1 to 23 policies.

Table 2 Percentage overhead for the three runtime groups

Runtime Group	Before	After	Difference
Short	6997%	1694%	5303%
Medium	496%	135%	361%
Long	38%	7%	31%

In total, only 1073 out of 1440 possible alignments must be done. That makes 18 alignments per query on average and means a reduction of 25%. The number of GTs is reduced by 26%.

6 Conclusion and Future Work

We have implemented two optimizations that have significantly reduced classification latency in QREP. Our benchmark evaluation has revealed that each GT increases the latency by 137ms on average. The latency caused by the fingerprint optimization does not carry weight, however, avoids GTs in 87% of all cases. By this, the latency of a single new-query alignment is reduced to 0.35 ms. In total, the latency of new-policy alignment in “full” classification mode shrinks on average from 34095 ms to 1430 ms, and the latency of new-query alignment on average from 21601 ms to 906 ms.

The “first” classification mode is only applicable to new-query alignment with a blacklist approach. Its impact heavily depends on the total number of policy rules, their order, and the number of queries matching the policies. We have configured a balanced scenario where 31 requests matched policies and 29 did not. In this scenario, the total number of alignments has been reduced by 25% and the number of GTs decreased by 26%.

In future, we will extend the fingerprint by additional graph-based QM to further reduce the number of GTs. We also aim to revamp our data model for contextual QM to enable an evolutionary query classification at runtime and to extend our DSL to fit it for arbitrary use cases beyond data privacy.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view

a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Adam et al (1989) Security-control methods for statistical databases: a comparative study. *ACM Comput Surv* 21(4):515–556
- Biswas et al (2011) Transforming privacy policies to auditing specifications. In: 13th Int. Symposium HASE. IEEE, Boca Raton, pp 368–375
- Danaparamita et al (2011) Queryviz: helping users understand SQL queries and their patterns. In: 14th Int. Conf. EDBT Uppsala, pp 558–561
- Dong et al (2005) Malleable schemas: a preliminary report. In: 8th Int. Workshop WebDB Baltimore, Maryland, USA, pp 139–144
- Fabbri et al (2013) SELECT triggers for data auditing. In: ICDE Brisbane, pp 1141–1152
- Garg et al (2011) Policy auditing over incomplete logs: theory, implementation and applications. In: ACM Conf. CCS Chicago, IL, pp 151–162
- Gerl et al (2018) Lpl, towards a GDPR-compliant privacy language: formal definition and usage. *Trans Large Scale Data Knowl Cent Syst* 37:41–80
- Goyal et al (2008) A unified audit expression model for auditing SQL queries. In: Annual IFIP WG 11.3 Working Conf. Data and Applications Security London. LNCS, vol 5094. Springer, Berlin, pp 33–47
- Hausenblas et al (2013) Apache drill: interactive ad-hoc analysis at scale. *Big Data* 1(2):100–104
- Johnson et al (2007) Compliance with data protection laws using hippocratic database active enforcement and auditing. *IBM Syst J* 46(2):255–264
- Kahng et al (2016) Interactive browsing and navigation in relational databases. *PVLDB* 9(12):1017–1028
- Kaushik et al (2011) Efficient auditing for complex SQL queries. In: ACM SIGMOD Athens, pp 697–708
- Khoussainova et al (2009) A case for a collaborative query management system. In: 4th Biennial CIDR Asilomar, CA
- Khoussainova et al (2011) Session-based browsing for more effective query reuse. In: 23rd Int. Conf. SSDBM Portland, OR, pp 583–585
- Koutrika et al (2010) Explaining structured queries in natural language. In: 26th ICDE Long Beach, CA, pp 333–344
- Mehta et al (2017) Qapla: policy compliance for database-backed systems. In: USENIX Security Symposium Vancouver, BC, pp 1463–1479
- Morel et al (2019) Three dimensions of privacy policies. *CoRR abs/1908.06814*
- Nabar et al (2008) A survey of query auditing techniques for data privacy. In: Privacy-preserving data mining – models and algorithms. Springer, Boston, pp 415–431
- Papastefanatos et al (2005) Hecataeus: a framework for representing sql constructs as graphs. In: 10th Int. Workshop EMMSAD, vol 5
- Pirahesh et al (1992) Extensible/rule based query rewrite optimization in starburst. In: ACM SIGMOD San Diego, CA, pp 39–48
- Schwab et al (2019) Query-driven enforcement of rule-based policies for data-privacy compliance. In: LWDA Berlin, pp 42–53
- Schwab et al (2020) A framework for dsl-based query classification using relational and graph-based data models. In: 3rd ACM SIGMOD Joint Int. Workshop GRADES-NDA Portland, OR, pp 10:1–10:5
- Schwab et al (2020) We know what you did last session – policy-based query classification for data-privacy compliance with the DataEconomist. In: 32nd Int. Conf. SSDBM Vienna

24. Sen et al (2014) Bootstrapping privacy compliance in big data systems. In: IEEE Computer Society (ed) IEEE Symposium on SP Berkeley, CA, pp 327–342
25. Shoshani (1982) Statistical databases: characteristics, problems, and some solutions. In: VLDB Mexico City. Morgan Kaufmann, San Francisco, pp 208–222
26. Srivastava et al (2007) Intensional associations between data and metadata. In: ACM SIGMOD Beijing, pp 401–412
27. Upadhyaya et al (2015) Automatic enforcement of data use policies with datalawyer. In: ACM SIGMOD Melbourne, Victoria, pp 213–225
28. Wahl et al (2018) A graph-based framework for analyzing SQL query logs. In: 1st ACM SIGMOD Joint Int. Workshop GRADES-NDA Houston, TX, pp 11:1–11:5
29. Wang et al (2019) Data capsule: a new paradigm for automatic compliance with data privacy regulations. In: Poly and DMAH, VLDB Workshops Los Angeles, CA. LNCS, vol 11721. Springer, Cham, pp 3–23
30. Zhou et al (2007) Query relaxation using malleable schemas. In: ACM SIGMOD Beijing, China, pp 545–556