Open Access

Open Journal of Internet of Things (OJIOT)
Volume 4, Issue 1, 2018
http://www.ronpub.com/ojiot
ISSN 2364-7108

Query Rewriting by Contract under Privacy Constraints

Hannes Grunert, Andreas Heuer

Database Research Group, University of Rostock, 18051 Rostock, Germany,
{hg, ah}@informatik.uni-rostock.de

Abstract

In this paper we show how Query Rewriting rules and Containment checks of aggregate queries can be combined with Contract-based programming techniques. Based on the combination of both worlds, we are able to find new Query Rewriting rules for queries containing aggregate constraints. These rules can either be used to improve the overall system performance or, in our use case, to implement a privacy-aware way to process queries. By integrating them in our PArADISE framework, we can now process and rewrite all types of OLAP queries, including complex aggregate functions and group-by extensions. In our framework, we use the whole network structure, from data producing sensors up to cloud computers, to automatically deploy an edge computing subnetwork. On each edge node, so-called fragment queries of a genuine query are executed to filter and to aggregate data on resource restricted sensor nodes. As a result of integrating Contract-based programming approaches, we are now able to not only process less data but also to produce less data in the result. Thus, the privacy principle of data minimization is accomplished.

Type of Paper and Keywords

Regular research paper: *query rewriting, privacy-by-design, aggregate constraints, edge computing*

1 Introduction

The Internet of Things consists of a variety of heterogeneous devices [13] with different capabilities. Several devices build a computation chain (see Figure 1) to process complex queries that are sent from a web service and process data collected by sensor nodes. Especially in wireless sensor networks with reduced capabilities, it is not ensured that the sensor nodes can handle every type of query. To avoid privacy issues by sending all raw data to the cloud provider, it has to be ensured that parts of the query can be applied directly on sensor nodes while the rest of the filtering has to be done on a more powerful device.

By limiting the amount of data to a minimum, privacy leaks can be prevented. To minimize data, calculations can partially be pushed down from web servers to local computers or even to sensor nodes. By adapting Query Containment algorithms, these parts can be determined.

Contribution: In our previous paper [14], we introduced the concept of a *Rewriting Supremum* Q_2 of a query Q_1, such that $Q_2 \sqsupseteq Q_1$[1], where Q_2 only consists of operators that are executable on the current layer and contains the

This paper is accepted at the *International Workshop on Very Large Internet of Things (VLIoT 2018)* in conjunction with the VLDB 2018 Conference in Rio de Janeiro, Brazil. The proceedings of VLIoT@VLDB 2018 are published in the Open Journal of Internet of Things (OJIOT) as special issue.

[1] $Q_2 \sqsupseteq Q_1$ holds for two queries Q_2 and Q_1 iff $Q_2(d) \supseteq Q_1(d)$ for all databases d satisfying the database schema and the integrity constraints.

```
tracks({id, title, length, album}, {{id}})
    tracks(album) → albums(id)
albums({id, name}, {{id}})
users({id, firstname, lastname, birthday}, {{id}})
buys({user, track}, {{user, track}})
    buys(user) → users(id)
    buys(track) → tracks(id)
```

Listing 1: Relational scheme for the running example

minimal amount of additional tuples in respect to Q_1. This paper examines existing algorithms and rules for query rewriting and adapts them to find the Rewriting Supremum. To achieve this, we combine the results from various Query Containment techniques with the *Design by Contract* concept known from the programming language Eiffel into our own approach, *Query Rewriting by Contract*. Our approach covers the problem of query rewriting in resource restricted environments where not every operator is available by giving a substitute query. Finally, we give an evaluation of our rewriting rules regarding runtime and data minimization.

Running Example: As a running example in this paper, we will use an excerpt of a music database, which consists of the relations *users*, *tracks* and *albums* (see Listing 1). To explain our notation in the listing, we use *tracks* as an example: This relation scheme consists of attributes *id*, *title*, *length*, and *album*. The attribute *id* is the only key of this scheme. The attribute *album* in this relation is a foreign key referring to the attribute *id* in relation *albums*.

For reasons of space, we can not present our full use case with all evaluated rules and queries in this paper. For the full study, including detailed execution plans, please refer to our website[2], which also includes the evaluations on the TCP-H benchmark and on an IoT scenario.

At a first glance, it seems as if this example has nothing to do with the Internet of Things, but when it comes to Smart Metering and Smart Environments, the data in such relations can be used for various (and unintended!) analysis, like media piracy detection [10].

Outline: The rest of the paper is structured as follows: The next section gives a brief overview of our framework for privacy aware query processing. Section 3 describes the state of the art in Query Rewriting approaches and the Design by Contract concept. In Section 4 we introduce our concept to rewrite complex queries with aggregate constraints. Section 5 evaluates our approach on some example queries. Our conclusions and future research directions are outlined in Section 6.

2 PArADISE

Our Contract-based Query Rewriting concept is part of the PArADISE[3] framework for privacy aware query processing. The execution of a given query is vertically distributed in a given system environment (see Figure 1) by our framework. This *Layered System Approach*, which can be compared to *Edge Computing* approaches [29], enables information systems to guarantee privacy aspects on the fly.

This architecture consists of sensors and mobile devices in the lower layers. The sensors are very resource-constrained in terms of CPU, memory, and power. Mobile devices, like mobile phones or edge nodes of a WSN, are less resource-constrained. The higher layers consists of routers, home media centers, local servers as well as Cloud-based web services, which are executed on powerful servers. Going from the top layer to the bottom layer, resource constraints are increasing, while the amount of executable database related operations is decreasing.

From the privacy point of view, each layer is a strict borderline to define the granularity of the data that is passed upwards. While the lower layers allow the user to control its own data, lower layers are more resource constrained. To solve this problem, an optimized query execution according to the given constraints is necessary.

Our framework is deployed as a middleware query processor (see Figure 2) on every node. The query processor consists of a preprocessor, which analyzes and rewrites the query, while the postprocessor modifies the result of the query in terms of anonymization metrics like k-anonymity [27]. The preprocessor is also responsible for the query rewriting of the input query into (1) a remainder query that is executed on the current layer and (2) a query fragment that is executed on the lower layers.

In [14], we showed how complex OLAP queries, e. g. regression and correlation analysis, can be split up into several query fragments that consist of basic aggregates.

[2] Permalink: `https://dbis.informatik.uni-rostock.de/links/vldb2018`

[3] Privacy AwaRe Assistive Distributed Information System Environment

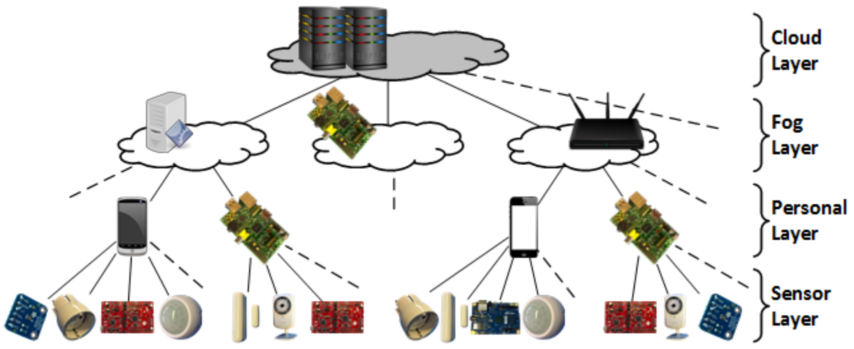

Figure 1: Layered System Approach

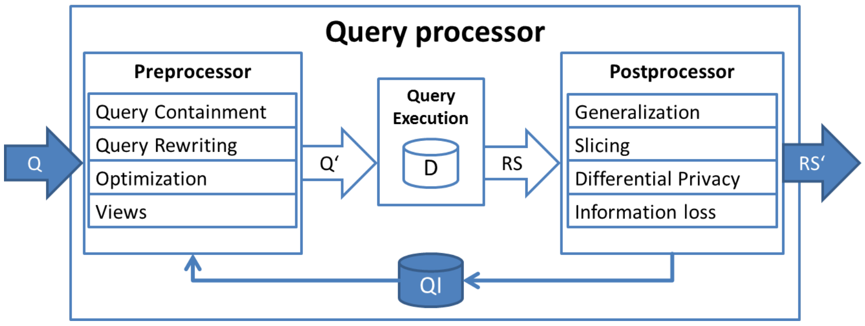

Figure 2: The Query Processor of PArADISE. (If query rewriting (left side) does not supply enough privacy, the result is anonymized by different techniques (right side). Quasi-Identifiers (QI) are used to parameterize the rewriting and anonymization.)

In this paper, we show how previous research on Query Containment and Contract-based programming can be utilized to optimize the fragments even more. We will concentrate on three classes of queries: (1) simple conjunctive queries, (2) linear arithmetic constraints and (3) aggregate queries with grouping.

3 State of the Art

In the following, the current state of the art in research and technology in the affected fields is briefly presented. This concerns the areas of cloud, edge and fog computing, theoretical and practical approaches to Query Containment and Query Rewriting.

3.1 Privacy

Data privacy, in particular the aspects of data avoidance and data minimization, has been seen as a legal rather than a technical problem in the past [20]. While many approaches have dealt with the anonymization of the result of a query [27, 15, 23], the aspect of data minimization by rewriting the query has been neglected as a key concept for achieving data protection on the technical side [19].

3.2 Cloud, Edge and Fog Computing

In the age of Big Data, the storage and processing of information are increasingly taking place in the cloud. Providers of such systems offer a variety of online services and ways to store our (personal) information. This also includes services used in assistive systems (e. g. in our own home). Unfortunately, the providers of the services usually do ignore or, at least, do not guarantee privacy with regard to unintended analysis by the provider.

Instead of using the server farms in data centers, the Internet of Things, consisting of sensors and devices, can process those data to take over parts of the query execution. This leads to an approach similar to that of fog or edge computing [28, 8], with the difference that in our approach the decision about how to distribute the query is done automatically (Privacy-by-Design). The distribution is based on the restricted query capabilities of the devices; e. g. TinyDB[4] does not support projections.

[4] `https://tinydb.readthedocs.io/en/latest/`

3.3 Query Containment

In this subsection, we give a brief overview on a variety of concepts to test the containment and equivalence of relational queries. The formal foundations of Query Containment and Query Equivalence, as well as the NP-completeness proof for the complexity of conjunctive queries, have been formally introduced by Chandra and Merlin in [3]. The Query Containment Problem (QCP) is defined as follows:

A query Q_2 is contained in the query Q_1 ($Q_2 \sqsubseteq Q_1$), if and only if for each instance d of the database D the query result $Q_2(d)$ is a subset of the query result $Q_1(d)$:

$$\forall d \in D : Q_2(d) \subseteq Q_1(d). \qquad (1)$$

Two queries are equivalent ($Q_1 \equiv Q_2$), if they contain each other.

For simple conjunctive queries (SPJ for short; consisting of simple selections with equality, projections and joins), Chandra and Merlin [3] show that the problem is computable. Based on their work, many approaches to further classes of queries have been developed: In [31] and [1], negated query predicates are added to the queries. Klug [17] extends the possible comparison predicates by further arithmetic operators ($<$, $\leq$, $\geq$, $>$). Chaudhuri and Vardi [4] translate the lessons learned from considering the QCP from set semantics to multiset semantics. The complexity of using multisets for general queries remains still open [5]. For a recent review of different bag and set approaches, please refer to Kolaitis in [18].

In addition to multiset semantics, the consideration of aggregation and grouping predicates in complex queries is also important. Klug [16] allows the integration of simple aggregates (sum, max, min, count) by extending the relational algebra. In [6, 7], these approaches are transferred from set to multiset semantics, whereby the sum, the maximum, minimum and the counting of (distinct) tuples also allows the comparison of queries on multisets. Grumbach [12] introduces additional aggregates with the average and the percentage calculation. In [2] the QCP is extended to queries with recursion.

More complex query containment considerations, especially the combination of aggregates with arithmetic comparisons (including negation) and functional dependencies as well as inclusion dependencies are considered by Can Türker in [30]. Türker divides the so-called linear arithmetic constraints into four classes: Attribute-value predicates (for range queries, $LAC1$), attribute-attribute comparisons ($LAC2$), with addition ($LAC3$) and multiplication ($LAC4$) over the integer domain. He further extends his approach by adding aggregate constraints for simple aggregate functions.

3.4 Answering Queries Using Views or Using Capabilities

To describe the operators or capabilities at the lower layers of our system, we have to adapt techniques used for the *Answering Queries using Views* problem (AQuV) so far. Here, Query Containment is checked for a query Q_1 against the base relations on our database and Q_2 against (materialized) views available in our database schema.

Algorithms for testing Query Containment for Answering Queries using Views include the *Bucket* algorithm [21], the *Inverse Rules* algorithm [9] and the *MiniCon* algorithm [26].

Instead of concrete views, we need more abstract capabilities to describe the operators at the lower layers of the system. Approaches for integrating capabilities are *GenCompact* [11], Capability Based Rewritings (CBR) [25] and capabilities or operators defined by equivalence classes of views in [22].

3.5 State of the Art: Summary

What is missing in the previous approaches are extensions of relational queries to arbitrary combinations of aggregate functions, groupings as well as window functions and fuzzy joins, for example on timestamps. The operations are needed in smart and assistive IoT environments and applications. Such complex queries cannot be analyzed adequately with the existing approaches and algorithms.

In addition, the query capabilities of the individual devices are included as parameters in the query transformation approach. The previous approaches to capability-based query transformation have been used for simple queries in mediator-based information integration systems. Unfortunately, these techniques are only suitable for simple SPJ queries and assume that the data sources have unrestricted main and hard disk memory for the calculation. To reduce the complexity, we adapted and combined the Design by Contract approach [24] from the programming language Eiffel with Query Containment checks to our own concept *Query Rewriting by Contract.*

4 CONCEPT

Our general approach to achieve data minimization is to split a complex query vertically into query fragments and remainder queries. The results of the fragment queries always contain a superset of the results of what is needed to get the same result as the original query. Each of these fragments and remainders can be calculated on a node that has enough capacities and allows specific operations to be executed (see [14]).

First, we give a short overview of our concept for rewriting queries under constraints. Then, we explain our new extension *Query Rewriting by Contract* to find the most suitable rewriting. Finally, we show how basic contract rules can be combined to find more complex rewritings.

4.1 Query Rewriting under Constraints in General

Given a query Q and a set of Node Layers L, Q is rewritten and split into a partial query Q_1 and a remainder query Q_δ. Q_1 can be executed on L_1 locally, while the remainder Q_δ is sent to the next Layer L_2. If L_2 supports all operations in Q_δ, Q_δ is executed on L_2 and the result is returned. Otherwise, Q_δ is split into a partial query Q_2 and a new remainder query $Q_{\delta'}$ and the procedure is repeated with $Q_{\delta'}$ until the cloud layer L_n is reached. This leads to a query chain on a database D:

$$Q(D) := Q_n(Q_{n-1}(\ldots Q_1(D))) \tag{2}$$

Based on the Query Containment Problem, we define our *Answering Queries using Operators* (AQuO) problem as follows: Given a database D, a query Q and a set of Layers L with each $L_i \in L$ having a set of operators O_i. The AQuO-problem asks for a rewriting r with $r(Q) = Q_i$, such that

$$Q_i(D) \sqsupseteq Q(D) \Leftrightarrow \forall d \in D : Q_i(d) \supseteq Q(d) \tag{3}$$

and Q_i uses only operations from O_i.

We call Q_i a *Rewriting Supremum*[5], if

$$\nexists Q'_i : Q_i(D) \sqsupset Q'_i(D) \sqsupseteq Q(D). \tag{4}$$

In the best case, $Q_i(D) \equiv Q(D)$ holds.

4.2 Query Rewriting by Contract

Our *Query Rewriting by Contract* concept to find the mentioned Rewriting Supremum is based on a huge set of rewriting rules. Each rule consists of a *left side*, the original query fragment, and a *right side*, the rewritten query fragment. Additionally, we assign *invariants*, *preconditions* and *postconditions* for each rule, which have a similar meaning as the corresponding terms in Eiffel. Invariants are conditions that are always valid (for example, a constant c has to be greater than zero all the time). Pre- and postconditions are tracked for the lower and the upper layer separately.

[5] A Rewriting Supremum is a rewritten query, that returns minimally *more* than or the same amount of tuples as the original query.

To trigger a rule, the preconditions and the invariants have to be satisfied. Regarding restricted capabilities, this will be the existence of an unsupported operator on the lower layer in most cases, e. g. an unsupported aggregate function, like the average over a numeric attribute. If we find a query fragment that matches the left side of the rule and satisfies the preconditions and invariants, the rule is triggered and the query fragment is replaced by the right side of the rule. Afterwards, the postconditions are checked. If all postconditions are satisfied, the rewriting of the given query fragment is finished. Otherwise, the postconditions are transformed into new preconditions that have to be satisfied by another rule. This procedure is repeated, until all fragments are rewritten and satisfy every postcondition and every invariant.

To prevent cycles, we keep track of equivalent rewritings, so that we do not execute the same rule twice for the same query fragment. Because of the definition of Query Containment, the prevented cycles and the higher capabilities on each layer, the rewriting process will terminate.

To introduce our rule set, we first present our simplest rule, the $\theta - \square$ rule:

Rule: $\theta(r) \sqsubseteq_K \square(r)$
Invariants: —
Preconditions:

- lower layer: θ is not supported
- upper layer: —

Postconditions:

- lower layer: —
- upper layer: θ is supported

Contract-based rules are marked with an K ($\sqsubseteq_K$) to delimit them from normal containment rules. The main idea behind the $\theta - \square$ rule is to push any unsupported operator θ from the lower layer to the upper layer. This circumstance can also be seen in the pre- and postconditions. The rule takes any relational operator θ from the lower layer and adds a function for the identical mapping $\square$, where $\square(r) \equiv r$ holds, to each child node r of θ. On the upper layer, $\theta(r)$ is executed instead of $\square(r)$.

We assume, that the operator $\square$ can be executed on every layer. Without executing any other operation on the lower layer, $\square(r)$ is equal to transmitting every tuple and every attribute from r to the upper layer.

The visualization of the $\theta - \square$ rule is given in Figure 3 as a relational algebra tree. The small example query fragment shows a simple projection π on the

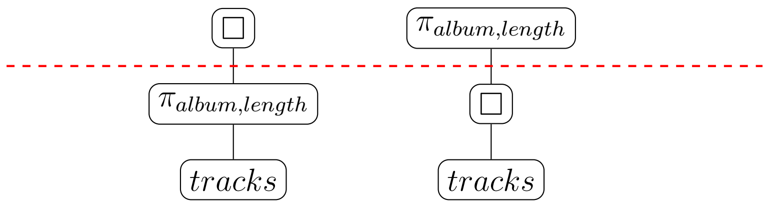

Figure 3: A simple rewrite rule for placing an operator on a higher layer

```
WITH X AS(
    SELECT album, length
    FROM tracks)
SELECT *
FROM X
```

Listing 2: Unmodified SQL sample query for rewriting rule $\theta - \square$

```
WITH X AS(
    SELECT *
    FROM tracks)
SELECT album, length
FROM X
```

Listing 3: Rewritten SQL sample query for rewriting rule $\theta - \square$

relation $tracks$. On the left side, we see the unmodified query, which executes the projection on the lower layer (below the red line), while the upper layer just reads the data transmitted from the lower layer. The right side shows the modified query, whereby the projection is not executed on the lower layer, but on the upper layer. To realize this, $\square$ and π (as an instance of θ) are swapped between the layers.

For the evaluation of our rule set, we transform the expression from the relational algebra back to SQL. To simulate a multi node environment, we used Common Table Expressions (CTE, the WITH clause in SQL) to prevent unintended optimization between the two layers. Listing 2 represents the unmodified query fragment, while Listing 3 shows the corresponding rewritten SQL statement.

The operator $\square$ will be not present in the query fragments in most cases. If possible, we use the $\theta - \square$ rule in combination with other rules without mentioning it explicitly.

To prevent unnecessary many rewritings, we apply the basic optimization rules from database theory to push down selections and projections towards the leaf nodes of the rewriting tree of the whole query and not only in the query fragments. Simple selections and projections are supported by most database system and embedded devices, so we only have to deal with more complex operators like aggregates and grouping. Additionally, we are breaking down complex functions, aggregates and group-by extensions into sets of primitives. By this, we get the advantage that we do not have to test the containment relation for every function against every other function. For instance, a regression analysis can be split into its primitives, addition, multiplication and count, so that we can apply our rule set on it.

Based on this initial query tree, we calculate an initial distribution of the query fragments to the specific layer. Beginning at the leaf nodes, we go up the tree until we find an unsupported operator. The child node, or nodes in case of binary operators like joins, resp., of the unsupported operator are then marked as query fragments, which can be executed on the lowest layer. The results of the fragments are inserted as new leaf nodes in the remainder query. The procedure is repeated on the next layer until the root node is reached. Details on this algorithm can be found in [14]. In the following, we concentrate on applying further optimizations after the initial distribution of the query.

After the initial distribution, the algorithm for the execution of our Contract-based rewriting is executed. While there is no possibility to swap operators by rules of the “normal” optimization, our rule set contains additional rules to push down further operators towards the leaf nodes. By applying rules that return no equal result, but a superset of the desired result, we can add more filter operations on lower nodes.

At a first glance, this might look like a deoptimization of the query, because additional and unnecessary operators are added to the execution of the query. This is true for “traditional” database systems, where everything is executed on a single node or a single cluster of homogeneous computing units. In the Internet of Things, we have a heterogeneous network of different devices, like sensor, router and server nodes. All nodes in the processing chain of a query have to touch the data at least once until the result is computed on the root node (e. g., a cloud server). If the nodes have to forward the data anyway, they can pre-filter or pre-aggregate the data in addition.

This procedure has three advantages: First, the

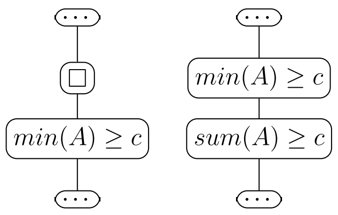

Figure 4: Rewriting rule A14 for the aggregate function min

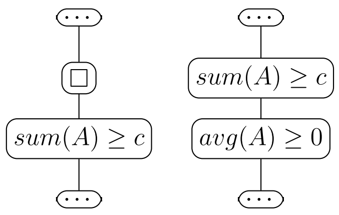

Figure 5: Rewriting rule A23 for the aggregate function sum

parallelization on thousands or millions of nodes can be exploited. Second, with transferring less data to the upper layers, these layers have to compute less data from all nodes. Thus, they execute the query fragments much faster. And finally, the lower nodes have a lower power consumption due to the reduced data transmission. Especially in mobile nodes, the batteries of such devices have a longer lifetime.

4.3 Finding New Rules

By converting unsatisfied postconditions into new preconditions, multiple contract rules can be applied at once on the same fragment. However, this can result in unnecessary operators between the supported filter on the lower layer and the final filter on the upper layer. To overcome this problem, we can apply our rule set also to detect these unnecessary operators. This algorithm finds these operators as follows: Consider two rewriting rules $X \sqsubseteq_K Y$ and $Y \sqsubseteq_K Z$. When both rules are applied on the same query fragment, we get the rewriting chain $X \sqsubseteq_K Y \sqsubseteq_K Z$. If X and Y are executed on the upper layer and Z is executed on the lower layer, we can skip Y, because it is not a Rewriting Supremum w. r. t. to the upper layer. Otherwise, if X is executed on the upper layer and Y and Z are executed on the lower layer, Z can be skipped.

The second case is illustrated in the following example. Let X be the minimum over a numeric attribute A, which has to be greater than a constant c. Let Y be the sum and Z the average over the same attribute, whereby the average is compared to 0 and not to c. The first rule, internally numbered as $A14$, is visualized in Figure 4; the second rule, $A23$, is shown in Figure 5.

Rule A14 is defined as follows:

Rule A14: $min(A) \geq c \sqsubseteq_K sum(A) \geq c$
Invariants: $c \geq 0$
Preconditions:

- Lower layer: min is not supported
- Upper layer: —

Postconditions:

- Lower layer: sum is supported
- Upper layer: min and $\geq$ are supported

Rule A23 is defined as follows:

Rule A23: $sum(A) \geq c \sqsubseteq_K avg(A) \geq 0$
Invariants: $c \geq 0$
Preconditions:

- Lower layer: sum is not supported
- Upper layer: —

Postconditions:

- Lower layer: avg is supported
- Upper layer: sum and $\geq$ are supported

Suppose that $min(A) \geq c$ and $sum(A) \geq c$ are not supported on the lower layer, we can apply first rule $A14$ and rule $A23$ afterwards. This will first result in the second tree from the left in Figure 6 after applying the first rule, and in the third tree after the second rule. As we can see, the sum-filter is unnecessary on the upper layer and can be removed, because $sum(A) \geq c \sqsubseteq avg(A)$ holds (fourth tree).

This results in the following new rule:

New rule: $min(A) \geq c \sqsubseteq_K avg(A) \geq 0$
Invariants: $c \geq 0$
Preconditions:

- Lower layer: min is not supported
- Upper layer: —

Postconditions:

- Lower layer: avg is supported
- Upper layer: min and $\geq$ are supported

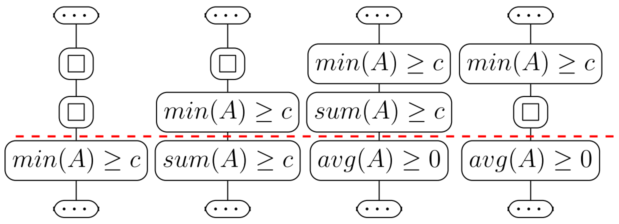

Figure 6: A combination of rewriting rules A14 and A23 via inference

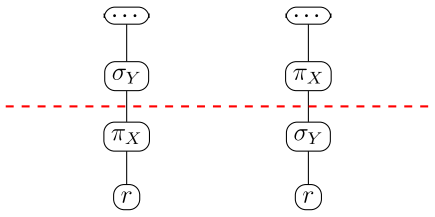

Figure 7: Rewriting tree for the rewriting rule K09

```
WITH X AS(
  SELECT title , length
  FROM tracks
)
SELECT *
FROM X
WHERE length > 250000
```

Listing 4: Unmodified SQL query for rewriting rule K09

Please note, that the new rule is only used for the specific query and is not added to our basic rule set, where we only store the minimal amount of rules needed. In the next section, we evaluate the runtime of such rewriting rules in a simulated environment. While data minimization – and maintaining the correct result – can be achieved by applying these rules, it is unknown how this affects the runtime of IoT information systems.

5 Evaluation

For reasons of space, we cannot evaluate every rewriting rule of our rule set[6] in this paper. To show the variety of our rule set, we present detailed measurements for one rule of the classical query rewriting (class K), one rule for linear arithmetic constraints (class L) and two different aggregate constraints (class A).

[6] Currently 80 rules, see `https://dbis.informatik.uni-rostock.de/links/vldb2018` for the current rule set.

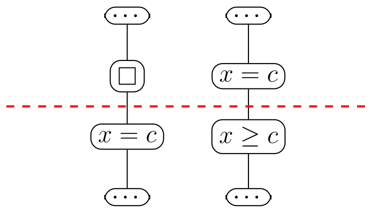

Figure 8: Rewriting tree for the rewriting rule L03

```
WITH X AS(
  SELECT *
  FROM tracks
  WHERE length > 250000
)
SELECT title , length
FROM X
```

Listing 5: Rewritten SQL query for rewriting rule K09

5.1 Rules in Detail

For each rule, we list the left and right side of the rewriting, the invariants, the preconditions and the postconditions. To achieve a better understanding, we also show the visualization of the rewriting tree. For the evaluation, we give two queries, one without and one with the applied rewriting rule. The results regarding runtime are given in the next subsection.

5.1.1 K09: Commutativity of Selection and Projection – V1

The following rule changes the order of a simple selection with a projection (see Figure 7). This rule is based on standard database optimizer engines, so we needed a CTE in the SQL examples (see Listings 4 and 5) to prevent unintended optimizations. Note that the rewriting itself is equivalent, but because of different pre- and postconditions it results in two different Contract-based rules.

Rule: $\sigma_Y(\pi_X(r)) \sqsubseteq_K \pi_X(\sigma_Y(r))$

```
SELECT album, length
FROM tracks
WHERE length = 250000
```

Listing 6: Unmodified SQL query for rewriting rule L03

```
WITH X AS(
  SELECT album, length
  FROM tracks
  WHERE length >= 250000
)
SELECT album, length
FROM X
WHERE length = 250000
```

Listing 7: Rewritten SQL query for rewriting rule L03

Invariants: —
Preconditions:

- lower layer: π_X is not supported
- upper layer: —

Postconditions:

- lower layer: σ_Y is supported
- upper layer: π_X is supported

5.1.2 L3: equal-to → greater-than-or-equal-to (attribute-constant)

The following rule adds an additional filter to a query (see Figure 8). This rule is based on [30] and uses CTEs (see Listings 6 and 7).

Rule: $x = c \sqsubseteq_K x \geq c$
Invariants: —
Preconditions:

- lower layer: $=$ is not supported
- upper layer: —

Postconditions:

- lower layer: $\geq$ is supported
- upper layer: $=$ is supported

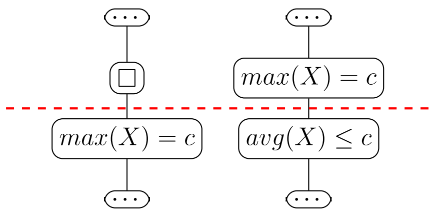

Figure 9: Rewriting tree for the rewriting rule A10

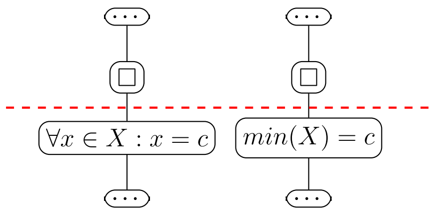

Figure 10: Rewriting tree for the rewriting rule A29

5.1.3 A10: maximum-equal-to → average-less-than-or-equal-to (attribute-constant)

The following rule adds an additional aggregate constraint to an other aggregate constraint (see Figure 9). This rule is also based on [30]. For such aggregate queries we do not need a CTE, because most database engines are unable to optimize such queries (see Listings 8 and 9).

Rule: $max(X) = c \sqsubseteq_K avg(X) \leq c$
Invariants: —
Preconditions:

- lower layer: max or $=$ are not supported
- upper layer: —

Postconditions:

- lower layer: avg and $\leq$ are supported
- upper layer: max and $=$ are supported

5.1.4 A29: equal-to → minimum-equal-to (attribute-constant)

The following rule replaces a universal quantifier constraint with an aggregate constraint (see Figure 10). This rule is based on the work from Can Türker [30]. At a first glance, the rewriting may sound confusing, so we took a quite simple rule: If the value of an attribute A is always equal to the constant c, the minimum of A has to be equal to c. Please note, that the rule is only valid in the given direction. Example queries are shown in Listings 10 and 11.

```
SELECT album
FROM tracks
GROUP BY album
HAVING max(length) = 250000
```

Listing 8: Unmodified SQL query for rewriting rule A10

```
SELECT album
FROM (
  SELECT album, length
  FROM tracks
  WHERE album IN (
    SELECT album
    FROM tracks
    GROUP BY album
    HAVING avg(length) <= 250000
  )
) AS X
GROUP BY album
HAVING max(length) = 250000
```

Listing 9: Rewritten SQL query for rewriting rule A10

Rule: $\forall x \in X : x = c \sqsubseteq_K min(X) = c$
Invariants: —
Preconditions:

- lower layer: $\forall$ is not supported
- upper layer: —

Postconditions:

- lower layer: min and $=$ are supported
- upper layer: —

5.2 Evaluation Results in Detail

The following tables present our evaluation results for every rewriting rule. The tests were performed on a single database server running PostgreSQL 9.6 on Centos 7.4. Regarding hardware aspects, the server was equipped with four single core CPUs (Intel Xeon E312xx (Sandy Bridge) @ 2000 MHz and 4 MB Cache) and 8 GB main memory. In contrast to sensor hardware, the server is of course over equipped. The goal of our performance measurement is to only show the runtime increase by rewriting the query.

Each table shows up to eight rewriting rules in both variants: $-$ for the unmodified query, $+$ for the rewritten query. The corresponding example queries for every rule and variant were executed ten times (measured in milliseconds). By building the ratio of the average

```
SELECT DISTINCT album
FROM tracks
WHERE album NOT IN(
  SELECT album
  FROM tracks
  WHERE NOT(length = 250000)
)
```

Listing 10: Unmodified SQL query for rewriting rule A29

```
SELECT DISTINCT album
FROM tracks
GROUP BY album
HAVING min(length) = 250000
```

Listing 11: Rewritten SQL query for rewriting rule A29

runtime of both variants, we calculated the overhead for the rewritten query.

In case of the classic optimization rules (class K, see Tables 1, 2 and 3), we used common table expressions for the subqueries to prevent internal, unintended optimizations. If there is no significant increase through the rewriting, the overhead should be around -20% to 20% due to measuring fluctuations. As we can see, some of the rules result into a huge overhead, while some increase the overall performance of the system. That confirms years of research in database optimization.

In case of the aggregate constraints (class A, see Tables 4, 5, 6, 7, 8, 9 and 10), we used CTEs. For linear arithmetic constraints (class L, see Table 11), we used CTEs only on the right side of the rule to express the additional filter operation. Through the additional filter, the runtime decreased slightly or increased up to 100% (except to rule $L07$ that had a terrible performance). Most of the queries had an increase around 50% to 80%. This has confirmed our expectation, because most of the queries filtered out around 20% to 50% of the tuples, so that at least half of the database was scanned twice. Regarding a possible IoT scenario, this will not affect the runtime, because the data from sensor etc. will be queried, no matter if a filter is added or not.

Most interesting and surprising is the decrease in the runtime by replacing universal quantifier with group-by aggregate constraints ($A27$ to $A34$). If aggregate functions are supported on the lower layers, like sensors, the rewriting will help to increase the performance of the system.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented our evaluation results for our approach to rewriting queries with aggregate constraints

Rule	K01		K02		K03		K04		K05		K06		K07		K08	
Variant	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+
#1	40	24	8	22	68	93	84	89	34	31	7	10	6	10	11	8
#2	38	23	7	21	59	85	83	82	32	24	6	10	7	10	11	8
#3	39	22	7	21	60	96	81	81	32	24	6	11	6	10	11	8
#4	39	20	7	21	59	97	84	81	32	25	6	11	6	10	11	8
#5	39	20	7	22	59	98	82	81	34	26	6	10	6	10	11	7
#6	38	21	7	21	60	95	81	83	33	27	6	11	6	10	11	8
#7	38	21	6	21	61	94	83	82	32	27	6	11	6	11	11	7
#8	40	21	6	21	59	97	84	78	33	24	6	10	7	11	11	7
#9	39	21	6	21	62	99	84	80	32	24	7	10	6	14	11	7
#10	39	20	7	22	59	94	83	78	32	24	6	10	6	13	11	7
AVG	38,9	21,3	6,8	21,3	60,6	94,8	82,9	81,5	32,6	25,6	6,2	10,4	6,2	10,9	11,0	7,5
Overhead in %	-45,24		213,24		56,44		-1,69		-21,47		67,74		75,81		-31,82	

Table 1: Measurements for the rules K01 to K08 on the Amarok dataset

Rule	K09		K10		K11		K12		K13		K14		K15		K16	
Variant	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+
#1	24	21	8	16	22	18	7	16	5	3	3	3	2	1	1	1
#2	24	21	7	16	21	17	6	16	3	1	2	3	2	2	2	2
#3	22	22	7	16	21	17	6	16	2	2	2	3	1	1	1	2
#4	21	22	7	16	23	17	6	16	3	2	3	3	2	2	1	2
#5	21	22	6	16	21	17	6	16	3	1	2	4	2	1	2	2
#6	21	22	6	16	21	17	6	16	2	2	3	3	1	2	1	2
#7	21	22	6	16	21	17	6	16	3	1	2	3	2	1	1	2
#8	21	34	6	16	21	17	6	16	3	2	2	4	1	1	2	2
#9	25	27	6	16	21	17	6	16	3	2	3	3	2	2	1	1
#10	34	23	6	16	21	16	6	16	3	1	2	4	2	1	1	2
AVG	23,4	23,6	6,5	16,0	21,3	17,0	6,1	16,0	3,0	1,7	2,4	3,3	1,7	1,4	1,3	1,8
Overhead in %	0,85		146,15		-20,19		162,30		-43,33		37,50		-17,65		38,46	

Table 2: Measurements for the rules K09 to K16 on the Amarok dataset

Rule	K17		K18		K19		K20		K21	
Variant	-	+	-	+	-	+	-	+	-	+
#1	1	2	4	4	3	7	25	34	28	32
#2	2	2	5	3	4	5	24	34	27	31
#3	2	1	4	4	3	6	25	33	28	30
#4	3	2	5	3	4	5	24	33	28	30
#5	1	2	4	4	3	6	24	33	28	38
#6	2	1	5	3	3	5	24	33	28	33
#7	2	2	4	4	4	5	24	33	28	28
#8	2	2	5	5	3	4	24	33	28	28
#9	2	1	5	5	6	5	24	33	27	27
#10	1	2	4	5	3	5	24	33	28	27
AVG	1,8	1,7	4,5	4,0	3,6	5,3	24,2	33,2	27,8	30,4
Overhead in %	-5,56		-11,11		47,22		37,19		9,35	

Table 3: Measurements for the rules K17 to K21 on the Amarok dataset

under privacy and capability constraints. We introduced a concept for rewriting complex aggregate queries with Contract-based rewriting rules to show how a query can be rewritten into another query with a restricted set of given operators.

As a use case, we used an intuitive example based on a music database which can be part of an Internet of Things scenario, where music is analyzed to determine user preferences. In this scenario, multiple queries, containing simple filters and projections as well as complex aggregates were rewritten and split into multiple queries in order to execute them between the sensors and the cloud server. Thus, no unintended additional analysis can be applied by the cloud provider, because the data is transmitted only in the form of highly aggregated information for the specified purpose.

We showed on some example queries how the runtime as well as the amount of transmitted information changes in contrast to normal queries. Taking into account that data have to be touched on every node in the

Rule	**A01**		**A02**		**A03**		**A04**		**A05**		**A06**		**A07**		**A08**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	41	23	12	37	11	23	11	30	12	27	12	27	12	26	11	25
#2	13	23	12	30	11	23	11	35	11	27	11	27	12	29	11	33
#3	12	22	12	26	11	30	11	29	11	27	11	30	11	27	11	26
#4	13	22	17	26	11	29	11	28	11	27	11	28	11	26	11	28
#5	12	37	14	23	11	28	11	28	14	27	11	27	11	25	11	26
#6	12	27	13	23	11	28	11	28	17	27	11	27	11	26	11	26
#7	12	23	12	23	11	38	11	34	14	27	11	27	11	30	11	24
#8	11	23	13	23	11	30	11	28	13	27	11	27	11	26	11	25
#9	11	22	12	22	11	40	11	37	12	27	11	31	11	29	11	24
#10	11	29	11	22	11	33	11	28	11	27	11	29	11	26	11	25
AVG	14,8	25,1	12,8	25,5	11,0	30,2	11,0	30,5	12,6	27,0	11,1	28,0	11,2	27,0	11,0	26,2
Overhead in %	69,59		99,22		174,55		177,27		114,29		152,25		141,07		138,18	

Table 4: Measurements for the rules A01 to A08 on the Amarok dataset

Rule	**A09**		**A10**		**A11**		**A12**		**A13**		**A14**		**A15**		**A16**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	11	25	12	22	12	22	12	31	12	28	12	28	12	31	14	34
#2	11	25	11	22	12	25	12	28	12	28	12	28	15	30	13	32
#3	11	25	11	22	12	23	12	23	12	28	12	28	15	31	13	33
#4	11	25	12	22	11	25	12	22	12	28	12	28	13	31	13	32
#5	11	25	12	22	12	24	12	22	11	29	12	27	12	30	12	33
#6	11	25	12	22	12	23	12	22	12	29	11	24	12	30	12	32
#7	11	25	15	22	12	23	12	22	12	29	12	24	11	36	12	33
#8	12	25	13	22	11	29	12	22	12	30	11	24	11	34	13	34
#9	12	25	12	22	12	24	12	22	11	29	12	29	12	25	13	32
#10	12	24	12	22	12	22	12	22	12	29	12	29	11	24	13	31
AVG	11,3	24,9	12,2	22,0	11,8	24,0	12,0	23,6	11,8	28,7	11,8	26,9	12,4	30,2	12,8	32,6
Overhead in %	120,35		80,33		103,39		96,67		143,22		127,97		143,55		154,69	

Table 5: Measurements for the rules A09 to A16 on the Amarok dataset

Rule	**A17**		**A18**		**A19**		**A20**		**A21**		**A22**		**A23**		**A24**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	13	32	13	27	19	25	14	25	15	29	16	29	12	25	13	25
#2	13	30	13	27	15	25	14	25	15	29	15	29	12	25	13	25
#3	13	27	13	27	14	25	14	25	15	29	15	29	12	25	13	25
#4	12	27	13	27	14	25	14	25	15	29	15	29	13	25	12	26
#5	12	27	13	27	14	25	14	25	15	29	22	29	14	25	12	25
#6	13	27	13	27	14	24	14	26	15	28	18	29	14	25	12	26
#7	13	43	13	27	14	25	14	26	15	28	16	29	13	25	12	26
#8	13	27	13	27	14	25	14	25	15	28	15	28	13	25	12	26
#9	14	27	13	27	14	25	14	25	15	29	15	29	13	25	12	25
#10	13	27	21	27	14	25	14	25	15	29	15	29	12	25	12	25
AVG	12,9	29,4	13,8	27,0	14,6	24,9	14,0	25,2	15,0	28,7	16,2	28,9	12,8	25,0	12,3	25,4
Overhead in %	127,91		95,65		70,55		80,00		91,33		78,40		95,31		106,50	

Table 6: Measurements for the rules A17 to A24 on the Amarok dataset

processing chain to transmit it to the next layer, the runtime increases in most cases only slightly. In some cases, it even decreases, because significantly less data is processed on the next layer.

It remains an open research question, if this result also holds on real, heterogeneous hardware, where the lower layers have less computing power than the upper layers but allow the parallel execution of the queries. The longer runtime on the lower layers may be repealed by the fact that the higher layers, which integrate all information from the sensors, have to compute less data.

Our future work will concentrate on integrating more Query Containment checks and thus on finding even more possible rewritings in combination with our existing Contract-based rule set. If you find one, do not hesitate to contact us! Also, we want to investigate the effect of the selectivity of the query predicates on the runtime in the distributed IoT scenario.

Rule	**A25**		**A26**		**A27**		**A28**		**A29**		**A30**		**A31**		**A32**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	14	28	14	28	17	9	17	9	19	26	19	26	19	16	17	15
#2	13	28	13	29	25	9	17	9	19	26	20	26	18	15	17	19
#3	13	28	13	29	19	9	17	9	19	25	20	26	18	15	17	17
#4	14	29	13	29	17	9	17	9	20	26	19	26	22	16	17	17
#5	13	29	13	30	17	9	17	9	19	26	19	26	20	16	17	16
#6	13	29	13	30	17	9	17	9	19	26	19	26	18	16	17	16
#7	13	29	16	29	17	9	17	9	19	26	19	26	17	15	17	16
#8	13	29	17	29	17	9	17	9	20	26	19	26	17	15	17	16
#9	16	28	15	30	17	9	17	9	20	26	21	26	17	15	17	16
#10	14	28	13	29	17	9	17	9	19	26	21	26	17	15	17	16
AVG	13,6	28,5	14,0	29,2	18,0	9,0	17,0	9,0	19,3	25,9	19,6	26,0	18,3	15,4	17,0	16,4
Overhead in %	109,56		108,57		-50,00		-47,06		34,20		32,65		-15,85		-3,53	

Table 7: Measurements for the rules A25 to A32 on the Amarok dataset

Rule	**A33**		**A34**		**A35**		**A36**		**A37**		**A38**		**A39**		**A40**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	17	9	16	10	17	27	16	27	19	28	17	27	18	27	19	28
#2	16	9	16	10	16	26	18	27	19	27	17	27	17	27	18	28
#3	16	9	16	10	16	26	17	27	19	27	17	26	17	30	19	27
#4	16	9	16	10	16	26	17	27	23	27	17	26	17	37	18	28
#5	16	10	16	10	16	26	17	27	23	27	17	26	18	29	18	28
#6	17	10	16	10	17	26	17	27	20	28	17	26	18	26	18	28
#7	16	10	17	10	16	26	19	26	20	45	17	27	17	26	18	28
#8	16	10	17	10	16	27	18	26	20	30	17	26	17	25	18	28
#9	16	9	16	10	16	27	16	26	20	28	17	26	17	26	18	28
#10	16	9	16	10	16	26	16	26	20	28	17	27	17	26	18	28
AVG	16,2	9,4	16,2	10,0	16,2	26,3	17,1	26,6	20,3	29,5	17,0	26,4	17,3	27,9	18,2	27,9
Overhead in %	-41,98		-38,27		62,35		55,56		45,32		55,29		61,27		53,30	

Table 8: Measurements for the rules A33 to A40 on the Amarok dataset

Rule	**A41**		**A42**		**A43**		**A44**		**A45**		**A46**		**A47**		**A48**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	18	29	18	31	17	33	14	33	16	34	9	16	11	16	16	28
#2	17	28	18	31	16	33	14	31	16	34	9	16	11	16	14	30
#3	16	28	18	40	16	31	14	30	17	42	9	16	11	16	12	35
#4	15	28	18	36	16	32	14	30	16	33	9	16	11	16	11	28
#5	15	28	18	38	16	32	14	35	17	32	9	16	11	16	11	27
#6	15	28	18	39	17	28	14	39	16	32	9	16	11	16	10	27
#7	15	28	18	38	17	26	14	31	16	44	9	16	11	19	10	27
#8	15	28	18	38	16	26	14	30	16	36	9	16	11	17	9	27
#9	15	28	18	37	16	26	14	32	16	35	12	16	11	17	9	27
#10	15	28	18	37	16	26	14	31	16	32	11	16	11	17	9	27
AVG	15,6	28,1	18,0	36,5	16,3	29,3	14,0	32,2	16,2	35,4	9,5	16,0	11,0	16,6	11,1	28,3
Overhead in %	80,13		102,78		79,75		130,00		118,52		68,42		50,91		154,95	

Table 9: Measurements for the rules A43 to A48 on the Amarok dataset

REFERENCES

[1] F. N. Afrati, C. Li, and P. Mitra, "On containment of conjunctive queries with arithmetic comparisons," in *the 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18*, 2004, pp. 459–476.

[2] M. Benedikt, G. Puppis, and H. Vu, "The complexity of higher-order queries," *Inf. Comput.*, vol. 244, pp. 172–202, 2015.

[3] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational data bases," in *the 9th ACM Symposium on Theory of Computing*, 1977, pp. 77–90.

[4] S. Chaudhuri and M. Y. Vardi, "Optimization of real conjunctive queries," in *the 20th Symposium on Principles of Database Systems*, C. Beeri, Ed., 1993, pp. 59–70.

Rule	**A49**		**A50**		**A51**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**
#1	9	17	9	17	9	27
#2	9	16	9	16	9	27
#3	9	16	9	16	9	28
#4	9	16	9	17	9	30
#5	9	16	9	16	9	35
#6	9	17	9	16	9	29
#7	9	16	9	16	9	37
#8	9	17	9	17	9	29
#9	9	16	9	16	9	35
#10	9	16	9	17	9	30
AVG	9,0	16,3	9,0	16,4	9,0	30,7
Overhead in %	81,11		82,22		241,11	

Table 10: Measurements for the rules A49 to A51 on the Amarok dataset

Rule	**L01**		**L02**		**L03**		**L04**		**L05**		**L06**		**L07**		**L08**	
Variant	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**	**-**	**+**
#1	15	20	7	11	6	18	15	19	10	12	6	9	8	15	23	29
#2	15	19	6	11	7	13	15	23	10	11	7	8	7	14	23	30
#3	14	18	7	11	6	12	15	26	11	13	7	9	6	14	22	30
#4	15	19	7	12	7	12	15	21	10	12	6	8	8	15	23	29
#5	15	19	6	10	6	11	14	20	10	12	7	8	7	14	22	30
#6	15	19	7	11	7	11	15	19	10	13	6	8	6	14	23	29
#7	16	20	6	12	6	11	14	19	10	12	7	9	7	14	23	31
#8	14	19	7	11	6	11	15	18	11	12	7	8	7	15	23	30
#9	15	19	6	10	7	11	14	19	12	13	6	8	6	14	23	30
#10	15	20	6	11	6	10	15	18	10	12	7	9	7	14	23	30
AVG	14,9	19,2	6,5	11,0	6,4	12,0	14,7	20,2	10,4	12,2	6,6	8,4	6,9	14,3	22,8	29,8
Overhead in %	28,86		69,23		87,50		37,41		17,31		27,27		107,25		30,70	

Table 11: Measurements for the rules L01 to L08 on the Amarok dataset

[5] S. Cohen, "Equivalence of queries combining set and bag-set semantics," in *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, Chicago, Illinois, USA*, S. Vansummeren, Ed., 2006, pp. 70–79.

[6] S. Cohen, W. Nutt, and Y. Sagiv, "Deciding equivalences among conjunctive aggregate queries," *J. ACM*, vol. 54, no. 2, p. 5, 2007.

[7] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting Aggregate Queries Using Views," in *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, Philadelphia, Pennsylvania, USA*, 1999, pp. 155–166.

[8] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya, "Fog computing: principles, architectures, and applications," *CoRR*, vol. abs/1601.02752, 2016.

[9] O. M. Duschka and M. R. Genesereth, "Query planning in infomaster," in *Proceedings of the 1997 ACM symposium on Applied computing*. ACM, 1997, pp. 109–111.

[10] M. Enev, S. Gupta, T. Kohno, and S. N. Patel, "Televisions, video privacy, and powerline electromagnetic interference," in *ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 537–550.

[11] H. Garcia-Molina, W. Labio, and R. Yerneni, "Capability-sensitive query processing on internet sources," in *the 15th International Conference on Data Engineering*, 1999, pp. 50–59.

[12] S. Grumbach, M. Rafanelli, and L. Tininini, "On the equivalence and rewriting of aggregate queries," *Acta Inf.*, vol. 40, no. 8, pp. 529–584, 2004.

[13] H. Grunert and A. Heuer, "Privacy Protection through Query Rewriting in Smart Environments," in *Proceedings of the 19th International Conference on Extending Database Technology, Bordeaux, France, Bordeaux, France, March 15-16*, 2016, pp. 708–709.

[14] H. Grunert and A. Heuer, "Rewriting complex queries from cloud to fog under capability constraints to protect the users' privacy," *Open Journal of Internet Of Things (OJIOT)*,

vol. 3, no. 1, pp. 31–45, 2017, special Issue: Proceedings of the International Workshop on Very Large Internet of Things (VLIoT 2017) in conjunction with the VLDB 2017 Conference in Munich, Germany. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-2017080613421

[15] I. Hegedus and M. Jelasity, "Differentially private linear models for gossip learning through data perturbation," *Open Journal of Internet Of Things (OJIOT)*, vol. 3, no. 1, pp. 62–74, 2017, special Issue: Proceedings of the International Workshop on Very Large Internet of Things (VLIoT 2017) in conjunction with the VLDB 2017 Conference in Munich, Germany. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-2017080613445

[16] A. C. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *J. ACM*, vol. 29, no. 3, pp. 699–717, 1982.

[17] A. C. Klug, "On conjunctive queries containing inequalities," *J. ACM*, vol. 35, no. 1, pp. 146–160, 1988.

[18] P. G. Kolaitis, "The query containment problem: Set semantics vs. bag semantics," in *Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management, Puebla/Cholula, Mexico, May 21-23*, L. Bravo and M. Lenzerini, Eds., vol. 1087, 2013.

[19] B.-J. Koops, J.-H. Hoepman, and R. Leenes, "Open-source intelligence and privacy by design," *Computer Law & Security Review*, vol. 29, no. 6, pp. 676–688, 2013.

[20] M. Langheinrich, "Privacy by design—principles of privacy-aware ubiquitous systems," in *International conference on Ubiquitous Computing*. Springer, 2001, pp. 273–291.

[21] A. Levy, A. Rajaraman, and J. Ordille, "Querying heterogeneous information sources using source descriptions," Stanford InfoLab, Tech. Rep., 1996.

[22] A. Y. Levy, A. Rajaraman, and J. D. Ullman, "Answering queries using limited external query processors," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 69–82, 1999.

[23] T. Li, N. Li, J. Zhang, and I. Molloy, "Slicing: A new approach for privacy preserving data publishing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 3, pp. 561–574, 2012.

[24] B. Meyer, "Applying design by contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[25] Y. Papakonstantinou, A. Gupta, and L. Haas, "Capabilities-based query rewriting in mediator systems," *Distributed and Parallel Databases*, vol. 6, no. 1, pp. 73–110, 1998.

[26] R. Pottinger and A. Halevy, "MiniCon: A scalable algorithm for answering queries using views," *The VLDB Journal - The International Journal on Very Large Data Bases*, vol. 10, no. 2-3, pp. 182–198, 2001.

[27] P. Samarati, "Protecting respondents identities in microdata release," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 6, pp. 1010–1027, 2001.

[28] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[29] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[30] C. Türker, *Semantic Integrity Constraints in Federated Database Schemata*, ser. DISDBIS. Infix Verlag, St. Augustin, Germany, 1999.

[31] F. Wei and G. Lausen, "Containment of conjunctive queries with safe negation," in *9th International Conference on Database Theory, Siena, Italy, January 8-10*, vol. 2572. Springer, 2003, pp. 343–357.

AUTHOR BIOGRAPHIES

Hannes Grunert was born in Ribnitz-Damgarten (Germany). He received his B.Sc. and his M.Sc. degree in Computer Science from the University of Rostock, Germany, in 2011 and 2013, respectively. He is currently a PhD student at the University of Rostock. His work is focused on privacy aware query processing.

Andreas Heuer studied Mathematics and Computer Science at the Technical University of Clausthal from 1978 to 1984. He got his PhD and Habilitation at the TU Clausthal in 1988 and 1993, resp. Since 1994, he is full professor for Database and Information Systems at the University of Rostock. He is interested in fundamentals of database models and languages, and in big data analytics, here especially performance, privacy and provenance.