Open Access

Open Journal of Internet of Things (OJIOT)
Volume 3, Issue 1, 2017
http://www.ronpub.com/ojiot
ISSN 2364-7108

Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy

Hannes Grunert, Andreas Heuer

Database Research Group, University of Rostock, Albert-Einstein-Straße 22, 18051 Rostock, Germany,
{hg, ah}@informatik.uni-rostock.de

Abstract

In this paper we show how existing query rewriting and query containment techniques can be used to achieve an efficient and privacy-aware processing of queries. To achieve this, the whole network structure, from data producing sensors up to cloud computers, is utilized to create a database machine consisting of billions of devices from the Internet of Things. Based on previous research in the field of database theory, especially query rewriting, we present a concept to split a query into fragment and remainder queries. Fragment queries can operate on resource limited devices to filter and preaggregate data. Remainder queries take these data and execute the last, complex part of the original queries on more powerful devices. As a result, less data is processed and forwarded in the network and the privacy principle of data minimization is accomplished.

Type of Paper and Keywords

Regular research paper: *query rewriting, query containment, privacy, databases, fog, cloud*

1 Introduction

In the Internet of Things, a variety of heterogeneous devices [10, 27] with different capabilities are involved in a complex computation chain (see Figure 1). Especially in capability restricted environments, such as sensor networks, it is not ensured that the processing unit can handle every type of query. Thus, it might be possible that data cannot be filtered by complex constraints on a sensor node. Through this, only a subset of these constraints can be applied directly on that node and the rest of the filtering has to be done on a more powerful node. By sending more data than intended to, e.g., a cloud provider, the provider can execute additional analysis tasks on the data and retrieve more information than intended or allowed. To prevent this, it has to be ensured that the amount of additional data is limited to a minimum to ensure the users' privacy concerns.

In order to minimize data, scientific calculations can partially be pushed from cloud servers down to local computers or even sensor nodes. To determine which parts of a query can be pushed down, Query Containment algorithms can be applied. The problem of query rewriting and query containment (and equivalence) has been studied by many research groups to solve problems in query optimization and information integration. While query rewriting is focussing on finding a rewriting r for a given query Q, query containment checks for a given r and Q if they are contained in each other:

This paper is accepted at the *International Workshop on Very Large Internet of Things (VLIoT 2017)* in conjunction with the VLDB 2017 Conference in Munich, Germany. The proceedings of VLIoT@VLDB 2017 are published in the Open Journal of Internet of Things (OJIOT) as special issue.

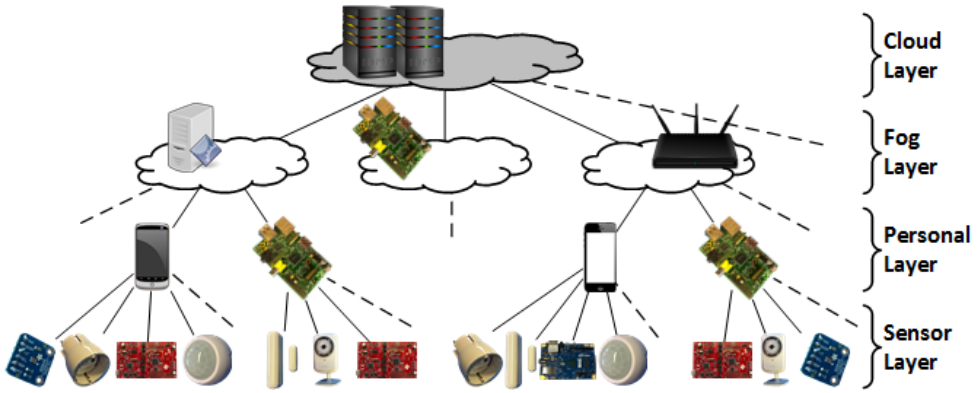


Figure 1: Layered System Approach

Let D be a database and $Q_i, i \in \mathbb{N}$ be some database queries. Q_1 is a subset query of Q_2 ($Q_1 \sqsubseteq Q_2$), if for every database D $Q_1(D) \subseteq Q_2(D)$ holds, where $Q_i(D)$ is the result of Q_i.

A main application of the Query Containment Problem is Answering Queries using Views (AQuV). The problem is defined as follows: given a query Q_1 on a database D and a set of views V over the same database, can Q_1 be answered by using only the views? Previous research (see Section 2) has focused on finding maximally-contained sets of rewritings Q_2 of Q_1 using only V instead of the database D, which is a partial answer to Q_1 and contains the maximal amount of answers.

Contribution: In this paper, we focus on finding a *Rewriting Supremum* Q_2 of Q_1, such that $Q_2 \sqsupseteq Q_1$ and Q_2 contains the minimum amount of additional tuples in respect to Q_1. In the best case, this minimal superset is equivalent to the original query Q_1. If such a rewriting exists, it is possible to use existing algorithms for query rewriting. Otherwise, these algorithms have to be modified.

Running example: As a running example in this paper, we will use a query Q, which consists of various predicates[1]:

$$\begin{aligned} Q(sum(x), y; y) := \; & x < 5 \\ & \wedge\, y\, BETWEEN\, 2\, AND\, 5 \\ & \wedge\, AVG(z) < AVG(x) \\ & \wedge\, regr\_slope(x, y) < 1. \end{aligned} \tag{1}$$

Q is a query in the canonical conjunctive normal form (CCNF) and consists of multiple predicates, which apply either to a single tuple or to an aggregated group. Later, we will call a predicate in a CCNF-query Q a *subgoal* of Q. This query is also an aggregate query, which calculates the sum of the x-values for each distinct value of y.

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, including aggregates and capability constraints. In Section 4 and 5 we introduce our concept to test containment of queries with complex aggregates. Section 6 applies our approach to more complex example queries. Our conclusions are outlined in Section 7.

2 PArADISE

Our query rewriting concept is part of the PArADISE[2] framework for privacy aware query processing. The main idea of the framework is to vertically distribute the execution of a query in a given system environment (see Figure 1). Thus, the privacy of the users, whose data are collected by various sensors and are stored in databases of different characteristics, is preserved. We refer to this process of the query execution as a *Layered System Approach*, which can be compared to *Edge Computing* approaches [28].

The layered architecture consists of four logically distinguishable layers. The *Sensor Layer* includes the sensors, which are very resource-constrained in terms of CPU, memory, and power. The *Personal Layer* consists of mobile devices or embedded systems, like mobile phones or edge nodes of a WSN. Router, home media centers, private servers, etc. build up the *Fog Layer*. The *Cloud Layer* is built by powerful servers, like data centers for Web Services.

From the top to the bottom layer resource constraints

[1] For a sample relation on this query see `http://ls-dbis.de/vliot-example`.

[2] Privacy AwaRe Assistive Distributed Information System Environment

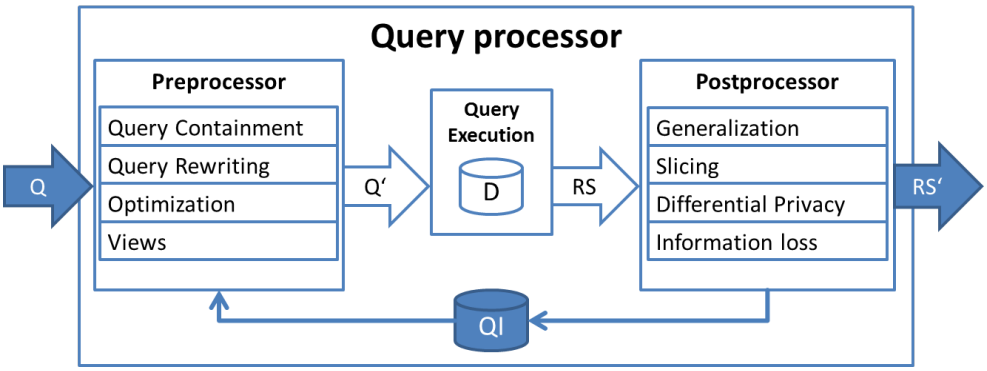


Figure 2: Query Processor

are increasing and the amount of possible database related functionalities and operations are decreasing. In terms of privacy, each layer defines a strict transition to define which data and to which granularity it is passed upwards. This allows the fine-grained protection of critical personal data as the information can be stored and processed within the local parts of the system. Generally, the lower the layer, the higher is the ability of the user to control its own data. As lower layers are more resource constrained than the upper ones, the middle layers provide functionalities for data processing. This enables optimized query execution according to the given resource constraints.

On every node, a customized JDBC driver (see Figure 2) is running as a middleware between the different layers. As input, the processor accepts a relational query formulated in SQL (and derivatives) and returns a resultset, which is an array of arrays of objects (a *relation* in terms of the relational model). The query processor consists of a preprocessor, which analyzes the query, while the postprocessor modifies the result of the query.

In the postprocessor, different metrics and algorithms for testing and ensuring privacy are implemented. This includes generalization based techniques to ensure k-anonymity [26], l-diversity [21] and t-closeness [19], permutation based techniques like Data Slicing [20] as well as Differential Privacy [6]. To parameterize these algorithms, we use for each base relation a set of quasi identifiers (QI) [3], which are calculated by an efficient algorithm [9] directly in the database. To prevent deanonymization attacks, like homogenity attacks and attacks via strong background knowledge, the anonymized results are reviewed again [8].

The detected QIs are also used in the preprocessor to modify the query to prevent access on sensitive data. This includes the prevention of a projection which includes all attributes of a QI at once and the prevention of an apparent range query which may only return one tuple. The preprocessor is also responsible for the query rewriting of the input query into (1) a partial query that is executed on the current layer and (2) a remainder query that is executed on the parent layer. This concept has been briefly introduced in [11]. In this paper we show how previous research on query containment and query rewriting can be utilized to perform the decomposition of the query.

3 State of the Art

The problems of Query Rewriting and Query Containment have been investigated for several years [1, 15]. In this section, we give a brief overview on a variety of concepts to test the containment and equivalence of relational queries. In the next section, we show how these concepts can be adapted to create a privacy aware query processing in the Internet of Things.

3.1 Classical Query Rewriting

For reasons of space, we give here just a short overview on established concepts. For details, please refer to [13, 31] or the original publications themselves.

Bucket: The Bucket algorithm [17] reformulates a conjunctive query on a given set of views into a rewritten conjunctive query on the database relations. Considering each subgoal in the query as a standalone, it determines which views may be useful for each subgoal. By this, the number of rewritings to be taken into account can be reduced.

The Bucket algorithm rewrites a query Q in two steps: First, a bucket is created for each subgoal G in Q containing all views that are necessary to answer G. Afterwards, the algorithm finds a set of conjunctive query rewritings that contains one conjunct c from every bucket. Each rewriting shows a way to retain a partial answer to Q using only the views. By building the

union of the rewritings, the maximally contained query rewriting is created.

Inverse Rules: The Inverse-Rules algorithm [5] constructs a set of rules that invert the views. An inverse rule is constructed for every subgoal in the body of a given view. For every variable that appears in the view definitions, a function symbol in the heads of the inverse rules is created. These function symbols show, which information can be extracted from the view definitions. The union of the inverse rules builds a maximally contained set of rewritings to answer a query Q.

MiniCon: The key idea of the MiniCon algorithm [24] is to consider how each of the variables in the query can be used in the available views instead of combining rewritings for each subgoal of the query. By doing so, the algorithm considers fewer combinations of views to find a suitable rewriting. In the first step, the MiniCon algorithm determines, which views contain subgoals that correspond to subgoals in the given query. Afterwards, the algorithm has to find the minimal amount of additional subgoals that have to be mapped to the subgoals in the set of views. In the second step, these mappings are combined to get the query rewritings.

3.2 Query Rewriting with Aggregates, Dependencies and Complex Comparisons

Semantic Integrity Constraints: In [30], Can Türker shows how to compute for two given integrity constraints I_1 and I_2 the relationship between each other. For two constraints c_1 and c_2, there exist five possible relationships. c_1 and c_2 can either be disjoint (i. e. they have no tuple in common), equivalent (they return the same result), c_1 contains c_2, c_2 contains c_1, or they overlap (i. e. it depends on the data).

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. Allowed comparisons operators include $<, \leq, =, \neq, \geq$ and $>$.

To determine the relationship between two sets of constraints, a weighted graph based approach is introduced. This graph algorithm tests the constraints for strongly connected components, where each component is a variable, which is represented as a node in the graph. Türker further extends his approach by adding aggregate constraints for simple aggregate functions as well as inclusion dependencies and functional dependencies.

Rewriting Aggregate Queries: Cohen et al. examine in [2] the QCP for aggregate queries under bag semantics. For a subset of aggregates, the so-called *expandable aggregates*, like min/max, count, sum and standard deviation[3], it is possible to test containment of queries containing these aggregates.

An aggregate query (α-query) is a disjunctive query defined as follows:

$$\begin{aligned} Q(\alpha(Y); X) \leftarrow\ & r_1(Z_1, \ldots, X, Y) \\ & \vee \ldots \\ & \vee r_n(Z_n, X, Y), \end{aligned} \tag{2}$$

where α is an aggregate function, X are the grouping and Y the aggregation attributes. The evaluation of the query works in two phases: (1) grouping and then (2) aggregation for each group. If a tuple fulfills multiple conditions, it will be counted multiple times in the aggregate function.

Their approach also allows the integration of integrity constraints and functional dependencies. The approach handles both bag and set semantics. As other QCP algorithms, it returns a finite, maximally-contained set of rewritings by building mappings from the original relations to a set of views.

3.3 Rewriting with Constraints

Chase and Backchase: The Chase/Backchase algorithm [23] can be used to find equivalent queries under a set of constraints C that are defined over a set of views and relations. C can include tuple-generating dependencies (TGDs) as well as equality-generating dependencies (EGDs), if the constraints are weakly acyclic. During the chase, a universal query plan, which includes all alternatives to answer a given query under the constraints, is generated. Then, the backchase searches for a minimal subset in the query plan that is equivalent to the original query.

In [4], this approach is extended and optimized by using a provenance-directed backchase. In the chase phase, provenance information is stored that can be used to generate the minimal subquery more efficiently in the backchase phase.

Capability-Sensitive Query Processing: In [7], Garcia-Molina et al. propose a scheme called *GenCompact* for generating capability-sensitive plans for relational queries. It is guaranteed that the sources can support, in respect to their capabilities, the generated query plans. Queries with the Boolean operators $\wedge$ and $\vee$ are transformed into either a CNF or a DNF. Based on the capabilites of the sources, a compact plan generator rewrites a given query. The rewrite module reorders the

[3] Complex aggregates like regression analysis and autocorrelation consists of such aggregates.

predicates to execute supported operators first. A cost model calculates for every generated plan the cost of the plan by estimating the size of the expected result. Afterwards, rules for pruning impure, sub-optimal and dominating rules are applied. At last, the plan generator produces a single plan for each condition and processes them separately for $\vee$- and $\wedge$-nodes.

Papakonstantinou et al. present a similar approach for Capability based rewriting (CBR) in [22]. Given a set of possible operations and a query that shall be executed on a given layer L, CBR determines partial SPJ queries that can be executed on L.

In [18], the theory of Answering Queries using Views is extended to the problem of Answering Queries using Restricted Capabilities. They use an infinite set of views to represent a special capability of resource restricted processors. To make this infinite set usable in practice, the infinite set of views is partitioned into equivalence classes. It is proven that a query can be answered by this infinite set of views if and only if it can be answered by a single query selected in one of the equivalence classes.

3.4 State of the Art: Summary

The approaches for Query Rewriting, Query Containment, and Answering Queries using Views (AQuV) introduced above are too restricted in two aspects. First, we have to consider more complex queries than SPJ queries such as statistical functions in database queries and are forced to handle them in rewritings. Second, the AQuV techniques map queries to an allowed set of views, while we need query rewritings to an allowed set of operators or capabilities. This is a more complex problem than mapping to views, because operators or capabilities are (seen formally) an infinite set of views. Additionally, AQuV techniques aim at queries that calculate a maximally contained *subset* of the original resultset. We need a *superset* of the original resultset, to be able to perform what we call *remainder queries* (see the next section).

4 Vertical Fragmentation of Complex Queries

Activity and intention recognition algorithms in smart appliances are often complex techniques like Hidden Markov Models [16], Fast Fourier transformations [14] and autocorrelation and regression analysis tasks. Currently, most systems collect data from various sensors and store them in the cloud. Then, the actual calculation is done on a cluster of multiple high performance servers. Privacy is often compromised, because sensible information is handed towards the cloud, even if this information can be preprocessed and prefiltered on a local node.

Our approach splits a complex query vertically into query fragments and remainder queries. Each of these fragments and remainders can be calculated on a node that has enough capacities and allows specific operations to be executed.

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 is reached. This leads to a query chain on a database D:

$$Q(D) := Q_n(Q_{n-1}(\dots Q_1(D))) \tag{3}$$

The results of the partial queries always contain a superset of the results of what is needed to get the same result as the original query. A simple, but quite negative example for a partial query Q_n is a query that returns every remaining tuple and every attribute:

$$Q_n := \pi_*(\sigma_{True}(Q_{n-1}(D_{n-1}))), \tag{4}$$

where D_{n-1} is the data processed on the layer L_{n-1}.

4.1 Answering Queries using Operators (AQuO)

To find a query that contains the minimal amount of additional information, but contains only a restricted set of operations, we have to revisit the Query Containment Problem as the theory in the background. The classical Query Containment Problem is best known from the "Answering Queries using Views" problem, which is specified as follows: Given a database D, a query Q and a set of views V over D, we search for a query Q_1, which is a rewriting r over D and uses only the views in V, so that

$$Q_1(D) \sqsubseteq Q(D) \Leftrightarrow \forall d \in D : Q_1(d) \subseteq Q(d). \tag{5}$$

We say that Q_1 is a *Maximally contained set of Rewritings* of $Q(D)$, if

$$\nexists Q' : Q_1(D) \sqsubset Q'(D) \sqsubseteq Q(D). \tag{6}$$

In the best case, $Q_1(D) \equiv Q(D)$ holds.

We will now slightly modify the AQuV problem to motivate our *Answering Queries using Operators* (AQuO) problem: 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 $r(Q) = Q_1$, such that

$$Q_1(D) \sqsupseteq Q(D) \Leftrightarrow \forall d \in D : Q_1(d) \supseteq Q(d) \quad (7)$$

and Q_1 uses only operations from O_1.

We call Q_1 a *Rewriting Supremum*[4], if

$$\nexists Q' : Q_1(D) \sqsupset Q'(D) \sqsupseteq Q(D). \quad (8)$$

In the best case, $Q_1(D) \equiv Q(D)$ holds. The best case is equal to the AQuV point of view from above.

The tricky point: As we mentioned in the Introduction, we want to *minimize* the amount of data processed by the information systems. With AQuO, it seems that Q_1 returns *more* data as a result of the query than the original query Q.

In reality, information systems gather all information from the data sources and do the aggregation and selection part of the query at a central node (cloud server with data warehouse, ...). As a consequence, we have as Q_1 a "SELECT * FROM table" query, which is executed on, for instance, a sensor node and collects all data. Nothing is preselected or preaggregated here, and the remainder query Q_δ does nearly all the work on the server side. This happens quite often when new information systems are designed. The developers frequently do not know which minimal amount of data is needed to perform the given task. Thus, they decide to collect all the data and they decide only later, which data will actually be included in the calculation when the system goes live: "Give me all you got. I will decide later on what happens with the data".

4.2 Algorithm

Like in other Query Containment Problem (QCP) approaches, we deal with conjunctive normal form queries Q_{CNF}, which have the form

$$Q(\alpha(X); Y, P) := \bigwedge_i \bigvee_j (\neg) p_{ij}, \quad (9)$$

where α is an aggregate function over a set of attributes X grouped by a set of attributes Y and p_{ij} is a (negated) predicate from the set of predicates P. In our approach, a predicate can either be a simple comparison (attribute-attribute or attribute-constant) from a where- or having-clause or even a subquery. We call each disjunction term a *subgoal* G_i of Q_{CNF}:

$$G_i = \bigvee_j (\neg) p_{ij}. \quad (10)$$

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

For example, subgoal G_3 of the example query Q is defined as follows:

$$G_3 := AVG(z) < AVG(x), \quad (11)$$

where the term is part of a having-clause defined in Q.

Given a query Q in a CNF, we want to find a mapping r to a query Q_1 with a limited set of operations. In order to find r, we have to map each subgoal G_i of Q to one or more equivalent or superset-generating subgoals $G_{i'}$ in Q_1.

Assume that Q has the form

$$Q := G_1 \wedge G_2 \wedge \cdots \wedge G_x \ldots G_n \quad (12)$$

and Q_1 has the form

$$Q_1 := G_1 \wedge G_{2'} \wedge \cdots \wedge \top \ldots G_m, \quad (13)$$

where m and n are the number of subgoals in Q and Q_1, and $\top$ is a subgoal that returns every tuple. We call $m(G_1) \equiv G_1$ an *equivalent, operator retaining mapping* of the subgoal G_1, if

$$\forall d \in D : m(G_1)(d) \equiv G_1(d) \wedge ops(G_1) = ops(m(G_1)) \quad (14)$$

holds. $m(G_2) \equiv G_{2'} \wedge G_{2''} \wedge \ldots$ is an *equivalent, fragmented mapping* of the subgoal G_2, if it is an equivalent mapping that is split into multiple subgoals that may contain different operators. A *partial mapping* $m(G_n)$ maps a subgoal G_n to a subgoal G_m, so that

$$\forall d \in D : G_m(d) := m(G_n)(d) \supseteq G_n(d) \quad (15)$$

holds. We call $m(G_x) = \top$ a *not applicable mapping*, if G_x contains at least one operator that cannot be executed on the current layer and there exists no suitable rewriting of G_x. The number of subgoals can differ from Q to Q_1 when fragmented mappings occur or there exists a subgoal G_w in Q_1 which has multiple corresponding subgoals in Q.

Example: Let Q be the example query from the Introduction, given in conjunctive normal form and $L := \{L_1, L_2\}$. Assume that L_2 has the capability to perform all operations. L_1 has limited capabilities, so that only a subset of operations O_1 is allowed: $O_1 := \{<, <=, =>, >, =, MIN, MAX\}$. Q consists of four subgoals:

- $G_1 := x < 5$
- $G_2 := y\, BETWEEN\, 2\, AND\, 5$
- $G_3 := AVG(z) < AVG(x)$
- $G_4 := regr\_slope(x, y) < 1$

With regards to O_1, G_4 cannot be executed on L_1, while G_2 can easily be rewritten by replacing the between predicate by <=- and =>-predicates. G_1 is a simple subgoal that can directly be executed on L_1. By applying the query rewriting approach by Can Türker, it is possible to replace the predicates in G_3 by MIN- and MAX-predicates.

One possible rewriting of Q is the partial query Q_1 on L_1:

$$\begin{aligned} Q_1(x,y,z;y) := \; & x < 5 \\ & \wedge \; y >= 2 \wedge y <= 5 \\ & \wedge \; MIN(z) < MAX(x) \\ & \wedge \; \top. \end{aligned} \tag{16}$$

with the following subgoals:

- $G_a := x < 5$
- $G_b := y >= 2$
- $G_c := y <= 5$
- $G_d := MIN(z) < MAX(x)$
- $G_e := \top$

The rewriting of Q to Q_1 contains an equivalent mapping from G_1 to G_a and an equivalent, fragmented mapping from G_2 to G_b and G_c. $m(G_3) = G_d$ is a partial mapping based on the condition that $MIN(X) \leq AVG(X)$ and $AVG(X) \leq MAX(X)$ holds [30]. By this, we can assume that G_d returns at least the same tuples than G_3. G_e returns the whole data, because there exists no mapping (as far as we know) of G_4 that returns more tuples than G_4 but less than all tuples.

Given two queries Q_1, Q and a database D, we can solve the AQuO problem by testing the subgoals:

$$\begin{aligned} & Q_1(D) \sqsupseteq Q(D) \Leftrightarrow \\ \forall d \in D : \forall G_i \in Q : \exists m : m(G_i)(d) \supseteq G_i(d) \end{aligned} \tag{17}$$

For every database instance d of the database D and every subgoal G_i from the original query Q, there exists a mapping m, such that the evaluation of $m(G_i)$ on d returns more tuples than G_i. In the worst case, all tuples are returned for each subgoal.

Based on this, we can express the *Rewriting Supremum* (RS) in a similar way. Q_1 is a RS, if

$$\begin{aligned} & \nexists Q' : Q_1(D) \sqsupset Q'(D) \sqsupseteq Q(D) \Leftrightarrow \\ & \forall d \in D : \forall G_i \in Q : \nexists m' : \\ & m(G_i)(D) \supset m'(G_i)(D) \supseteq G_i(D) \end{aligned} \tag{18}$$

4.3 Splitting the Query

Up to now, we have built a partial query Q_1 from the given query Q. Q_1 is executed on the layer L_1. For the rest of the execution, a remainder query Q_δ is needed, which removes the additional tuples and does the final aggregation on top of $Q_1(D)$: $Q \equiv Q_\delta(Q_1(D))$.

Q can be expressed as a conjunction of three subsets of its subgoals: $Q := \bigwedge G_X \wedge \bigwedge G_Y \wedge \bigwedge G_Z$, where

- G_X := set of (mapped) equivalent subgoals
- G_Y := set of superset generating subgoals
- G_Z := set of unmapped subgoals

Example: In the previous step, we transformed the query Q into the partial query Q_1. Given that partial rewriting, every subgoal from Q can be put into one of the three sets:

- $G_X := \{x < 5, y \, BETWEEN \, 2 \, AND \, 5\}$
- $G_Y := \{AVG(z) < AVG(x)\}$
- $G_Z := \{regr\_slope(x,y) < 1\}$

By constructing G_X, G_Y and G_Z, Q_δ can be defined as follows:

$$Q_\delta := \bigwedge G_Y \wedge \bigwedge G_Z \tag{19}$$

Thus, Q_δ contains all partial and all not applicable mapped subgoals. On the other hand, all subgoals from G_X, that have been fully executed by Q_1 on L_1, do not have to be executed again in Q_δ.

Example: By combining G_Y and G_Z, Q_δ is defined as follows:

$$\begin{aligned} Q_\delta := \; & AVG(z) < AVG(x) \\ & \wedge regr\_slope(x,y) < 1 \end{aligned} \tag{20}$$

The idea of splitting predicates in multiple parts is not completely new. It is a well-known concept that is used by algebraic optimization [29] in many database systems. For example, one of these rules allows the partial execution of selection predicates F on the base relations r_1 before a join with r_2: $\sigma_F(r_1 \bowtie r_2) \Leftrightarrow \sigma_F(r_1) \bowtie r_2$, if the attributes in F are a subset of the relation schema of r_1.

While these rules were intended to be used for a more efficient query processing by reducing the amount of comparisons between both relations, they can also be used for increasing privacy. If some parts of the selection are done on the local nodes (the base relations), less data is sent to the next layer, which executes the join operation. Our approach extends these algebraic rules by adding new query containment checks. In the next section, we will show how this approach can easily be assigned to complex aggregate queries.

4.4 Proof of Equivalence

Before we can handle complex queries, we have to show the correctness of our query rewriting. After rewriting the original query Q, we have a query chain QC (see equation 3). We will now show that Q is equivalent to QC. Without losing generality, we will prove the equivalence for a single rewriting step. Thus, our query chain consists of Q_1 as the partial query and Q_δ as the remainder query:

$$Q \equiv Q_\delta(Q_1(D)) \tag{21}$$

Example:

$$\begin{aligned} Q \equiv\, & AVG(z) < AVG(x) \\ & \wedge\, regr\_slope(x,y) < 1(\\ & \quad x < 5 \\ & \quad \wedge\, y >= 2 \wedge y <= 5 \\ & \quad \wedge\, MIN(z) < MAX(x) \\ & \quad \wedge\, \top) \end{aligned} \tag{22}$$

Proof. "⇒":
The proof follows directly from the construction of Q_1 and Q_δ from Q (see Subsection 4.2).
"⇐":
From logical optimization of database queries, we know that the logical AND is commutative for two sets of selection predicates F_1 and F_2:

$$\begin{aligned} & \sigma_{F_1}(\sigma_{F_2}(D)) \Leftrightarrow \\ & \sigma_{F_1 \wedge F_2}(D) \Leftrightarrow \\ & \sigma_{F_2}(\sigma_{F_1}(D)) \end{aligned} \tag{23}$$

Let F_2 contain the predicates from Q_δ and F_1 contain the predicates from Q_1. Because

$$\forall G_x \in Q_\delta : \exists G_{x'} \in Q_1, \tag{24}$$

with $G_x(D) \subseteq G_{x'}(D)$, based on the construction of Q_1 and Q_δ, we know that $G_{x'}$ returns a superset of tuples of D in respect to G_x. Due to this, we can remove each $G_{x'}$ from F_1 in the $F_1 \wedge F_2$ selection predicate, so that only the predicates from F_2 and the unreplicated predicates in Q_1 remain. In the case of rewritten subgoals, the equivalent subgoal from the construction step is inserted instead of the corresponding G_x. The query now contains only the predicates that are also used in Q. By this, $Q(D) \equiv Q_\delta(Q_1(D))$ holds. □

Example: For our running example, F_1 contains the predicates $x < 5$, $y >= 2$, $y <= 5$, $MIN(z) < MAX(x)$ and $\top$. F_2 consists of the predicates $AVG(z) < AVG(x)$ and $regr\_slope(x,y) < 1$. Thus,

$$\begin{aligned} F_2(F_1(D)) := \, & AVG(z) < AVG(x) \\ & \wedge\, regr\_slope(x,y) < 1(\\ & \quad x < 5 \\ & \quad \wedge\, y >= 2 \wedge y <= 5 \\ & \quad \wedge\, MIN(z) < MAX(x) \\ & \quad \wedge\, \top(D)). \end{aligned} \tag{25}$$

By applying equation 23, we get

$$\begin{aligned} F_2 \wedge F_1(D) := \, & AVG(z) < AVG(x) \\ & \wedge\, regr\_slope(x,y) < 1 \\ & \wedge\, x < 5 \\ & \wedge\, y >= 2 \wedge y <= 5 \\ & \wedge\, MIN(z) < MAX(x) \\ & \wedge\, \top(D). \end{aligned} \tag{26}$$

Now, some predicates can be removed, because they are overlapped by others:

1. $y\, BETWEEN\, 2\, AND\, 5 \equiv y >= 2 \wedge y <= 5$
2. $AVG(z) < AVG(x) \subseteq MIN(z) < MAX(x)$
3. $regr\_slope(x,y) < 1 \subseteq \top$

Thus, all right sides can be removed from $F_2 \wedge F_1$. The following query Q' remains:

$$\begin{aligned} Q' := \, & x < 5 \\ & \wedge\, y\, BETWEEN\, 2\, AND\, 5 \\ & \wedge\, AVG(z) < AVG(x) \\ & \wedge\, regr\_slope(x,y) < 1(D), \end{aligned} \tag{27}$$

which is equivalent to Q.

4.5 Unsupported Logical AND

Regarding sensor networks, the lowest layer L_1 contains nodes with a very restricted set of capabilities and operations, or without enough energy to process more than one subgoal at once. Therefore, it might happen that the logic AND operation $\wedge$ cannot be applied on L_1. As a result, complex predicates, with multiple conditions, cannot be executed on this layer. In order to preprocess the data on that node, one of the subgoals have to be chosen to be executed in the query Q_1. To decide which of the subgoals will be executed, the subgoals are ordered by their (descending) selectivity: $Q_{KNF^o} := ORDER(Q_{KNF})$

The function $ORDER$ orders each subgoal in Q_{KNF} by their descending selectivity. The ordering is due to

the fact, that a layer L_i may not support the operator $\wedge$ or do not have enough energy to process more than one subgoal at once.

As an example, the query Q from our running example has the following selectivities:

- $SEL(x < 5) = 0,5$
- $SEL(AVG(z) < AVG(x)) = 0,42$
- $SEL(y\,BETWEEN\,2\,AND\,5) = 0,05$
- $SEL(regr\_slope(x,y) < 1) = 0,01$,

where SEL is the size of the expected result divided by the cardinality of the relation. For an aggregate function agg over a set of attributes X, this is the number of groups specified by the grouping set Y divided by the cardinality c of the relation R:

$$SEL(agg(X)) := \frac{\#groups(Y)}{c(R)}. \tag{28}$$

After sorting the selectivities, the subgoal with the highest selectivity, which contains only supported operations, is chosen to be executed on L_1. In the example, this is the predicate $x < 5$. The regression analysis, which has the highest selectivity, cannot be executed, because it is an unsupported operation on L_1. Regarding the BETWEEN-predicate, it also cannot be executed, because BETWEEN is unsupported and the equivalent rewriting $y >= 2 \wedge y <= 5$ contains the unsupported logical AND. The predicate, which compares the average values of x and z, is unsupported and we assume for this example, that the rewriting $MIN(z) < MAX(x)$ has a lower selectivity than $x < 5$. Thus, only the predicate $x < 5$ remains for the execution on L_1.

4.6 Heads

Based on our rewriting concept, we can also define which attributes have to be passed through the query chain. As a prerequisite, we define two functions *var* and *head*: *head(X)* returns the head of a query X, i. e. the query signature. *var(X)* returns all variables given in a subgoal predicate X or a query head X.

The required attributes that must be returned by the remainder query Q_δ are the same as in the original query Q:

$$var(head(Q_\delta)) := var(head(Q)) \tag{29}$$

Similar to Q_δ, Q_1 must contain all variables from the head of Q_δ. Additionally, all attributes that appear in a subgoal of Q_δ, must be passed by Q_1:

$$var(head(Q_1)) := var(head(Q_\delta)) \bigcup var(G_i), G_i \in Q_1 \tag{30}$$

In our running example, Q_1 returns the attributes x, y and z. x and y are needed for the final output of Q_δ and z appears in the average comparison.

Algorithm 1: Query Distribution to Layers

```
Data: Query tree QT, set of layers L
Result: a set of query fragments F
i := 0;
F_0j := leaf(QT);
while i <= n do
    if F_ij ⊑ L_i then
        add parent to F_ij;
        combine all F_ij with same parent;
    else
        assign F_ij to L_i;
        remove F_ij from QT;
        add D_ij at same position;
        i++;
    end
end
if L_n contains root(QT) then
    return F;
else
    return ⊥;
end
```

4.7 Distribution of the Aggregate Fragments

If a query Q is translated into a query tree (see the next section), the assignment of the query fragments to the layers can directly be taken over by parsing the tree from its leaves to its root. Algorithm 1 shows how each fragment is assigned to a suitable layer. The algorithm takes the query tree QT as an input. Additionally, a set of layers L is given. Each layer $L_i \in L, 0 < i < n, i \in \mathbb{N}$, has a set of capabilities, defined by the allowed operations O_i. With L_0, we define the bottom layer where the raw data is processed, while L_n is the final output layer which outputs the result of Q. The algorithm outputs a set of fragments F_i which can be executed on the layer L_i.

After initialization, the algorithm parses the tree, beginning at the leaf nodes. While each node supports only allowed operations from O_i, the whole subtree is assigned to L_i. Otherwise, the unsupported nodes are placed on the next layer L_{i+1} and the subtrees are replaced by leaf nodes, which contain the intermediate

result. The procedure is repeated until the root of the query tree is reached. Our algorithm is based on the basic Query Folding algorithm [25], which can be computed in time exponential in the size of the query. Implementation details and test results can be found in [32].

5 Rewriting Complex Queries

Previous approaches for rewriting queries have several drawbacks in terms of complex aggregates and functions. Regarding linear regression, correlation or even Hidden Markov Models and Support Vector Machines, these approaches will fail to find a rewriting *r* for a given query *Q* which contains complex aggregates. This is either due to the usage of two or more different simple aggregates or because two counts are used on different attributes. With respect to the approach of Türker [30], the result will always lead to an overlapping relationship for *Q* and *r*.

Currently, only simple algorithms can be split up into their basic functions. The transformation of complex queries into simple fragments can be done automatically with our approach. By rewriting the complex query Q into Q_j and Q_δ, where Q_δ is executed on a more powerful layer, we can transfer only those data to the cloud, which do not compromise privacy. We will now show how an extension of the theory of query containment and query optimization can consider more complex queries, including complex statistical functions using aggregation and grouping.

The handling of data in IoT environments will be rethought fundamentally. Currently, data is just pushed to the cloud while the layered approach enables new methods to store, process and query data on the lower layers.

As we mentioned before, it is possible to use existing algorithm to calculate a Rewriting Supremum under capability constraints, if and only if the rewriting is equivalent to the query, because the rewriting returns exactly the same set of answers as the query. Thus, it cannot contain any additional information.

5.1 Meta Algebra

We introduced our Layer fragmentation in [12] and showed how a query can be split into fragments [11]. We will now introduce our meta algebra for rewriting complex queries for a layered evaluation.

Our algebra is a tuple $H := (O, D)$ with O being a set of allowed operations[5] and D being a multiset of typed data[6]. At this point, we will start with a simple example: As D, we have the domain of integer values and as O the operations "+" and "*". Correspondingly, H is defined as follows: $H := (\{+, *\}, \mathbb{Z})$

Data and operations can be seen as nodes in a directed forest[7]. A directed forest is a directed acyclic graph, which is not necessary connected (multiple trees). Data nodes can have a directed outgoing edge to an operator node. These nodes are the leaves of the trees. They are raw data, which are still not processed by an operation.

An operator node has one or more incoming edges from data or operator nodes and can have a directed outgoing edge (with a typed output) to another operator node. Consequently, an operation cannot be a leaf. Operations can have a certain number of incoming edges (the number of arguments) and each edge has to be of the appropriate type (e. g. multiplication is only allowed on numeric values, not on databases).

For simplification, data nodes and subtrees can occur multiple times (bag semantics). This will allow an easier evaluation when rewriting the queries. As a special case, the forest will become a tree when we have the final result of a query or a function after executing the last operation. Table 1 shows the translation of the calculation $(1 + 2) * (2 + 3)$ into the forest structure. The translation is done step by step from the raw data H^0, over the interim forest H^1 to the final tree H^2.

In database research, such trees are well-known when it comes to query optimization. Our approach extends these trees by allowing any type of operation (in this paper we focus on relational algebra with aggregates and calculations) and by restricting the allowed set of operations by capability constraints. Additionally, the operations and the data will be split on multiple computing units with different capability constraints.

5.2 Reducing the Complexity

When it comes to complex aggregates it becomes more difficult to determine the relationship between two queries. To reduce the complexity we introduce the concept of operator equivalence and containment. A set of operators O_1 contains ($\sqsupseteq_O$) another set of operators O_2 if it can do at least the same operations. For example, the set consisting only of the sum operator is a subset of the set consisting of the add operator:

$$\{add\} \sqsupseteq_O \{sum\}. \tag{31}$$

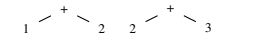
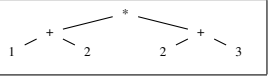
While the sum operator can only calculate the total sum for a given data set D, the add operator can calculate the sum of any two elements over D. Another well-known example is the average over a data set, which is a

[5] For the relational case: operations from the relational algebra

[6] For the relational case: relations and views

[7] If we use set semantics: directed, hierarchical graph

Table 1: Translation of a calculation into a forest structure

$H^0 := (\{+, *\}, \{1, 2, 3\})$	1 2 3
$H^1 := (\{+, *\}, \{(1 + 2), (2 + 3)\})$	
$H^2 := (\{+, *\}, \{(1 + 2) * (2 + 3)\})$	

subset of the set consisting of the operations sum, count and division:

$$\{avg\} \sqsubseteq_O \{sum, count, div\}. \tag{32}$$

O_1 and O_2 are said to be equivalent ($\equiv_O$), if they contain each other:

$$O_1 \equiv_O O_2 \Leftrightarrow O_1 \sqsubseteq_O O_2 \wedge O_2 \sqsubseteq_O O_1 \tag{33}$$

The same applies to the data: One set of data D_1 is contained in D_2, if every tree t $\in D_1$ is also in D_2 or can be rewritten by only using the operators from the related set O_1. The rewriting can be achieved by using either the associative, distributive, ... laws, the optimization rules of the relational algebra as well as the algorithm presented in Section 2. Two data sets are equivalent, if D_1 contains D_2 and D_2 contains D_1:

$$D_1 \equiv_D D_2 \Leftrightarrow D_1 \sqsubseteq_D D_2 \wedge D_2 \sqsubseteq_D D_1 \tag{34}$$

By breaking down complex functions and aggregates into sets of primitives we gain two advantages: First, we do not have to test the containment relation for every function against every other function. Second, we also get to know which complex functions require the same operators and the same data nodes for the calculation. By this, we can detect privacy violations. For instance, a smart (assistive) system wants to calculate the regression slope as part of an intention recognition analysis. By breaking down the regression into its primitives addition, multiplication and count, we can easily see that these operations can also be used to perform a correlation analysis over the data, which may be an unintended function or even malicious attack to comprimise the privacy of the user.

5.3 Query Execution

Algorithm 3 shows how for a database D and a set of layers L a given query Q can be fragmented into subqueries Q_i, so that Q_i can be executed on a capability restricted layer L_i. In the first step, the conjunctive normal form Q_{KNF} is built for Q, such that $Q_{KNF} := G_1 \wedge \cdots \wedge G_o$, with $G_j := p_{j1} \vee \cdots \vee p_{jn}$. Afterwards, the conjuncts are ordered by their selectivity in descending order. The order is stored in Q_{KNF^o}.

Algorithm 2: ExecuteSubgoals

```
Data: An ordered KNF-Query Q_KNF^o, Database
      D, set of n layers L
Result: A partially executed Query Q'_KNF^o
while L_x do not support ∧ x < n do
    G_high := findHighSelectivitySubgoal(Q_KNF^o);
    if ∃G_high then
        D_{x+1} := G_high(D_x);
        Remove G_high from Q_KNF^o
    end
    x++;
end
```

From the lowest layer L_0 onwards, it is checked whether L_x supports the operator $\wedge$. If not, the query cannot be executed on L_x, but it is possible to execute a single subgoal G_j (see Algorithm 2), if L_x has the capability to execute it. If such subgoals exist, the subgoal with the highest selectivity is executed on the data D_i from L_i and the preaggregated, prefiltered result is stored in D_{x+1} and handed up to layer L_{x+1}. Additionally, G_j is removed from Q_{KNF^o}, because a second execution will have no effect on the data. The procedure is repeated until L_x supports $\wedge$.

As soon as a layer L_x supports $\wedge$, the combination of subgoals is considered for execution. If it is possible to execute one or more subgoals G_i with the specified operations on L_x, they will be executed and removed from the remainder query Q_{x+1}, which will be executed on L_{x+1}. In case that a Rewriting Supremum r for G_i exists, $r(G_i)$ will be executed on L_x, but G_i will remain in Q_{x+1} to eliminate that "bit more" from $r(G_i)$. If there exists no suitable rewriting for G_i, it will be executed on L_{x+1}.

As soon as all subgoals are processed, the query Q'_x, which contains all remaining subgoals or their rewritings, will be executed on L_x resulting in the resultset D_{x+1}. If there are no subgoals to be executed on L_{x+1}, D_{x+1} is equivalent to $Q(D)$ and can be sent to the top layer. Otherwise, the capability check and the rewriting is repeated until the top Layer L_n is reached. In case that the top layer is reached, the query is executed

Algorithm 3: Query Rewriting and Partial Execution

Data: Query **Q**, Database **D**, set of n layers **L**
Result: A rewritten Query **Q'**
$Q_{KNF} := buildKNF(Q)$;
$Q_{KNF^o} := order(Q_{KNF})$;
$x := 0$;
ExecuteSubgoals;
$Q_x := Q_{KNF^o}$;
$D_x := getData(L_x)$;
while $x < n$ **do**
  for *each* G_i *in* Q_x **do**
    if L_x *supports* $Op(G_i)$ **then**
      Keep G_i on L_x;
    else
      if $\exists$ *Rewriting* r: $r(G_i) \equiv G_i$ *and* $r(G_i)$ *is supported on* L_x **then**
        replace G_i with $r(G_i)$;
      else
        if $\exists$ *Rewriting* r: $G_i \sqsubset r(G_i)$ *and* $r(G_i)$ *is Rewriting Supremum and* $r(G_i)$ *is supported on* L_x **then**
          replace G_i with $r(G_i)$;
          mark G_i for L_{x+1};
        else
          remove G_i;
          mark G_i for L_{x+1};
        end
      end
    end
  end
  $Q_x := \bigwedge_i G_i$ on L_x;
  $D_{x+1} :=$ Execute $Q_x(D_x)$ on L_x;
end
return D_{x+1};

and there are still subgoals remaining, there will be no result. But that would also mean that $Q(D)$ could not have been executed on L_n at all, because it needs the same base operations as its rewriting.

6 Example Queries

Figure 3 shows a rewriting tree for the calculation of a linear regression slope $reg\_slope(X, Y)$ over two numerical attributes X and Y. The regression slope is calculated as follows:

$$reg\_slope(X,Y) := \frac{\sum_{i=1}^{n}\left(X_i-\overline{X}\right)*\left(Y_i-\overline{Y}\right)}{\sum_{i=1}^{n}\left(X_i-\overline{X}\right)^2}$$

As a first distribution approach, the query is split into two fragments: The blue subqueries calculate the difference of an X value to the average of X, $\overline{X}$. The same applies for the Y values in the green subqueries. Because $\left(X_i - \overline{X}\right)$ is needed twice in the calculation, it appears twice as a data node for the rest of the calculation, which is done on the Cloud Layer (orange). The rest of the query (pair-wise multiplication, summation and division) is performed on that layer.

Figure 4 shows how a correlation $corr(X, Y)$ over two numerical attributes X and Y can be represented in a rewriting tree. The distribution of the query stays the same as in the regression analysis. The correlation is calculated as follows:

$$corr(X,Y) := \frac{\sum_{i=1}^{n}\left(X_i-\overline{X}\right)*\left(Y_i-\overline{Y}\right)}{\sqrt{\sum_{i=1}^{n}\left(X_i-\overline{X}\right)^2*\sum_{i=1}^{n}\left(Y_i-\overline{Y}\right)^2}}$$

For correlation analysis, it is insufficient to build query fragments that calculate the difference between the X-values and the mean of the X-values (in the figures with blue background) on the sensor node sending the rest of the query to the cloud provider. The same applies to the Y-values (green background). This is due to the fact that these intermediate results can also be used to perform a regression analysis. To prevent such an unintended analysis, the data has to be in a more aggregated form. This means, that the data nodes, which arrive at the Cloud Layer, must contain only data which cannot be used by any other algorithm.

Assume for our scenario, that the Sensor Layer does not have enough power to aggregate the data any further. To prevent the data to be sent to the Cloud Layer in its preaggregated form, we add an additional layer between the Sensors and the Cloud: the Home Media Center Layer. We assume that this layer has enough power to multiply each x-y-pair in the counter and to square and to sum up the x-y-pairs in denominator (gray layer in Figure 5). The rest of the analysis, the summation of the counter, the extraction of the root in the denominator and the final division, is done on the Cloud Layer. Note that the distribution of the query fragments in the regression slope tree has also to be modified in order to prevent an unintended correlation analysis.

7 Conclusions and Future Work

In this paper we presented a new approach for rewriting queries with aggregates under capability constraints. We introduced a concept for decomposition of complex aggregates into simple atoms and showed how a query can be rewritten into another query with a restricted set of operations, which returns a Rewriting Supremum to answer a given query.

As a use case, we utilized our approach to generate a

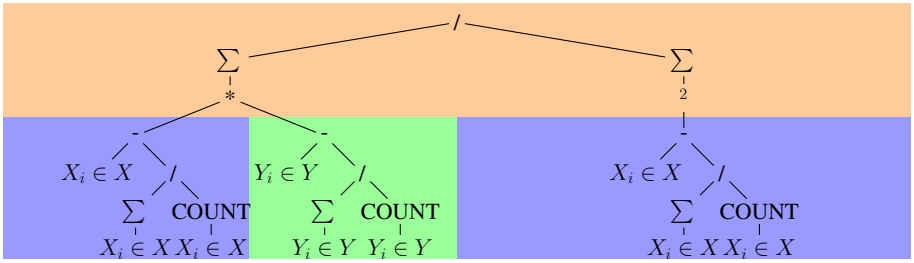


Figure 3: Rewriting tree for the aggregate function *regression slope* (green/blue: Fog Layer, orange: Cloud Layer)

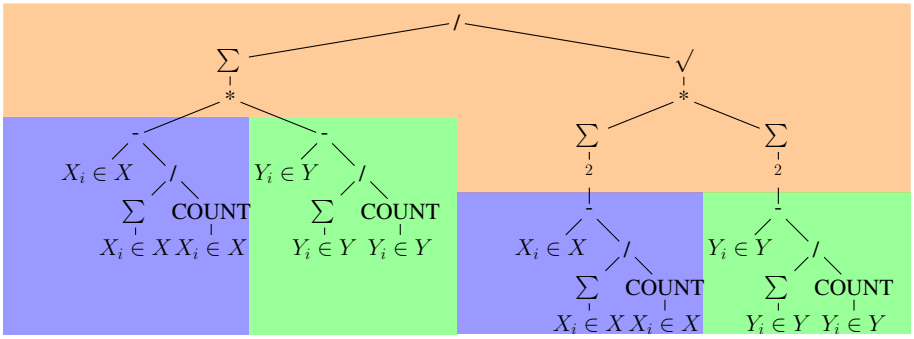


Figure 4: Rewriting tree for the aggregate function *correlation* (green/blue: Fog Layer, orange: Cloud Layer)

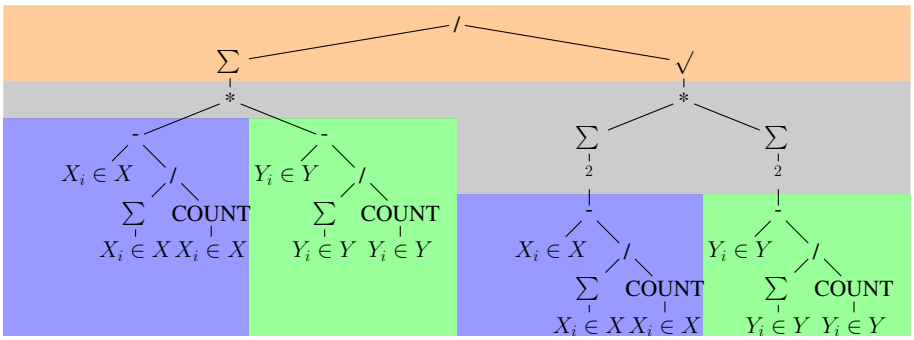


Figure 5: Rewriting tree for the aggregate function *correlation* with an improved fragmentation (green/blue: Fog Layer, orange: Cloud Layer, gray: Home Media Center Layer)

privacy aware decomposition of regression an correlation analysis into several query fragments, which are distributed in an Internet of Things scenario. In this scenario, simple filters and projections are done on a sensor node, while parts of complex aggregates are done on nodes between the sensors and the cloud server. The final result is computed on the cloud server, which does not get the raw input any more. Thus, no unintended analysis can be applied by the cloud provider and the user's privacy is protected.

It remains an open research question, if it is possible to gain additional knowledge out of the aggregated data via a malicious query Q' on the intermediate results. For example, Q' could be an inverse function, which restores the raw data from the aggregated results. Furthermore, intermediate results could be used for other calculations than intended. In our example with the correlation and the regression analysis, the blue and green intermediate results are the same. Thus, both queries are possible, even if only one was intended.

Future work will concentrate on finding rewritings, which allow no further usage of intermediate results rather than the intended queries. Also, we did not provide a detailed complexity analysis of our rewriting approach. Another interesting point is to integrate further rewriting algorithms, especially for aggregate

queries, in our distribution algorithm to find more suitable rewritings for a given set of operations. We also want to investigate how encryption mechanisms can be integrated into the computation chain.

ACKNOWLEDGMENTS

We thank the following students for their support by implementing parts of our framework: Felix Wächter, Martin Haufschild, Jan Tepke, Hannes Steffenhagen, Christoph Damerius (SQL query splitting), Richard Dabels, Johann Kluth, Jörg Stüwe, Roman Titok, Alex Lymar (anonymization) and Johannes Goltz (deanonymization). We also thank the anonymous referees for their constructive comments.

REFERENCES

[1] R. Chirkova, "Query Containment," in *Encyclopedia of Database Systems*. Springer US, 2009, pp. 2249–2253.

[2] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting Aggregate Queries Using Views," in *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 1999, pp. 155–166.

[3] T. Dalenius, "Finding a Needle In a Haystack or Identifying Anonymous Census Records," *Journal of Official Statistics*, vol. 2, no. 3, pp. 329–336, 1986.

[4] A. Deutsch and R. Hull, "Provenance-Directed Chase&Backchase," in *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, ser. Lecture Notes in Computer Science, V. Tannen, L. Wong, L. Libkin, W. Fan, W. Tan, and M. P. Fourman, Eds., vol. 8000. Springer, 2013, pp. 227–236.

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

[6] C. Dwork, "Differential privacy: A Survey of Results," in *International Conference on Theory and Applications of Models of Computation*, 2008, pp. 1–19.

[7] H. Garcia-Molina, W. Labio, and R. Yerneni, "Capability-sensitive Query Processing on Internet Sources," in *Proceedings of the 15th International Conference on Data Engineering 1999*, 1999, pp. 50–59.

[8] J. Goltz, "De-Anonymisierungsverfahren: Kategorisierung und deren Anwendung für Datenbankanfragen," Bachelor's thesis, Universität Rostock, 2017, in German.

[9] H. Grunert and A. Heuer, "Big Data und der Fluch der Dimensionalität: Die effiziente Suche nach Quasi-Identifikatoren in hochdimensionalen Daten," in *Proceedings of the 26th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken)*, 2014, in German.

[10] H. Grunert and A. Heuer, "Privacy Protection through Query Rewriting in Smart Environments," University of Rostock, Tech. Rep. CS-01-16, 2016.

[11] H. Grunert and A. Heuer, "Privacy Protection through Query Rewriting in Smart Environments," in *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, Bordeaux, France, March 15-16*, E. Pitoura, S. Maabout, G. Koutrika, A. Marian, L. Tanca, I. Manolescu, and K. Stefanidis, Eds., 2016, pp. 708–709.

[12] H. Grunert, M. Kasparick, B. Butzin, A. Heuer, and D. Timmermann, "From Cloud to Fog and Sunny Sensors," in *Proceedings of the Conference "Lernen, Wissen, Daten, Analysen", Potsdam, Germany, September 12-14,*, R. Krestel, D. Mottin, and E. Müller, Eds., vol. 1670, 2016, pp. 83–88.

[13] A. Y. Halevy, "Answering Queries Using Views: A Survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[14] J. Joshi, R. Goecke, S. Alghowinem, A. Dhall, M. Wagner, J. Epps, G. Parker, and M. Breakspear, "Multimodal Assistive Technologies for Depression Diagnosis and Monitoring," *Journal on Multimodal User Interfaces*, vol. 7, no. 3, pp. 217–228, 2013.

[15] P. G. Kolaitis and M. Y. Vardi, "Conjunctive-Query Containment and Constraint Satisfaction," *17. Symposium on Principles of Database Systems, Seattle*, pp. 205–213, 1998.

[16] F. Krüger, K. Yordanova, A. Hein, and T. Kirste, "Plan Synthesis for Probabilistic Activity Recognition," in *ICAART (2)*, 2013, pp. 283–288.

[17] A. Levy, A. Rajaraman, and J. Ordille, "Querying Heterogeneous Information Sources Using Source Descriptions," Stanford InfoLab, Tech. Rep., 1996.

[18] 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.

[19] N. Li, T. Li, and S. Venkatasubramanian, "t-closeness: Privacy Beyond k-Anonymity and l-Diversity." in *ICDE*, vol. 7, 2007, pp. 106–115.

[20] T. L. Li, N. Li, J. Zhang, and I. Molloy, "Slicing: A New Approach for Privacy Preserving Data Publishing," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, vol. 24, no. 3, pp. 561–574, March 2012.

[21] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-Anonymity," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 3, 2007.

[22] 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.

[23] L. Popa, "Object/Relational Query Optimization with Chase and Backchase," *IRCS Technical Reports Series*, p. 19, 2001.

[24] 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.

[25] X. Qian, "Query Folding," in *ICDE*. IEEE Computer Society, 1996, pp. 48–55.

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

[27] M. Schmude, "Systematische Untersuchung der Anfragekapazität verschiedener DBMS," Bachelor's thesis, Universität Rostock, 2016, in German.

[28] W. Shi and S. Dustdar, "The Promise of Edge Computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[29] J. M. Smith and P. Y.-T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Communications of the ACM*, vol. 18, no. 10, pp. 568–579, 1975.

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

[31] V. Vassalos, "Answering Queries Using Views," in *Encyclopedia of Database Systems*. Springer US, 2009.

[32] F. Wächter and M. Haufschild, "PArADISE Sensor- and Appliance-Level," https://eprints.dbis.informatik.uni-rostock.de/823/, 2017, in German.

AUTHOR BIOGRAPHIES

Hannes Grunert was born in Ribnitz-Damgarten (Germany). He received his B.Sc. degree 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.