Authorization-Aware Optimization for Multi-Provider Queries

Ekaterina B. Dimitrova
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
ekaterina@cs.pitt.edu

Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
panos@cs.pitt.edu

Adam J. Lee
Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania, USA
adamlee@cs.pitt.edu

ABSTRACT

The sharing of sensitive personal information via cloud platforms motivates the need for measures that aim to minimize information leakage to unauthorized users. In this work, we propose a novel SQL optimizer that strikes a balance between query runtime performance and private information exposure. Our approach to ensuring that the access control policies regulating data disclosure are enforced during distributed query execution is based upon a state-of-the-art authorization model from the literature and a preference-aware query optimizer. Our preliminary studies show that our approach outperforms the way the authorization model was originally implemented in terms of query runtime performance which is crucial for the operation on Big Data. To improve it, we adjust the algorithms utilized by a preference-aware query optimizer.

CCS CONCEPTS

• **Information systems** → **Query optimization**; • **Security and privacy** → *Management and querying of encrypted data*;

KEYWORDS

Big data, Distributed databases, Query optimization, Access model, Privacy

ACM Reference Format:
Ekaterina B. Dimitrova, Panos K. Chrysanthis, and Adam J. Lee. 2019. Authorization-Aware Optimization for Multi-Provider Queries . In *Proceedings of The 34th ACM/SIGAPP Symposium on Applied Computing (SAC'19).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3299731

1 INTRODUCTION

Cloud technologies facilitate the storage and processing of significant amounts of data in a decentralized manner. While they bring many benefits, they also introduce security concerns about data and query privacy and confidentiality [8], [14], [23]. On the client side, query processing in the cloud may leak data to third parties about the users' intentions or their private information. On the server side, distributed query processing may expose confidential data to unexpected parties as data flows traverse the system.

There is a need to protect private information during distributed query evaluation and processing. Traditional optimization methods for executing a query across multiple providers tend to favor performance over privacy, in that optimal plans from a performance perspective may make disclosures of sensitive data to intermediate nodes in the execution plan that could otherwise be prevented. Similarly, data releases previously considered to present anonymized data could be easily deanonymized. For example, there is an emerging trend of sharing data between partnering organizations in healthcare, following specific established data governance procedures (e.g., hospitals and insurance companies store and exchange information about their patients and customers). By being oblivious to such privacy policies, traditional query optimization and execution processes cannot guarantee them.

An attempt to balance privacy and performance was made in [4], where the authors carry out a two-phased process: the query was first optimized using an off-the-shelf optimizer, and then post-processed to ensure that the access control policies of data providers were respected during the distributed execution of the query. However, this two-phased approach separates query performance optimization and authorization policy enforcement. As a result, the optimality of the modified query evaluation plan may not be preserved, as the modifications needed for policy enforcement may lead to inefficient query execution.

In this work, we present a solution that integrates query optimization and access control authorization in a single phase, eliminating the shortcoming of the above two-phase approach. Towards this goal, we have integrated the access control authorization model into the distributed query optimizer PAQO [9],[12], [13]. PAQO is a version of the PostgreSQL query optimizer that uses user-specified preference constraints as an additional optimization metric [10], [11]. By including data providers' authorization constraints at optimization time, the resulting query execution plan strikes an optimal balance between policy enforcement and execution efficiency.

Contributions:

- We identify inefficiencies that arise when the access controls and visibility constraints at each data provider are considered and enforced by post-processing physical execution plans, rather than considering these as inputs to the optimization process.
- We propose a solution that mitigates the identified limitations of the two-phased approach to enforcing access controls and visibility constraints by considering these constraints as optimization metrics along with the traditional performance optimization criteria.
- We show how our proposed solution can be realized in PAQO, an existing distributed query optimizer. Specifically, the new *authorization-aware* version of PAQO provides users with

SAC'19, April 8-12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3299731

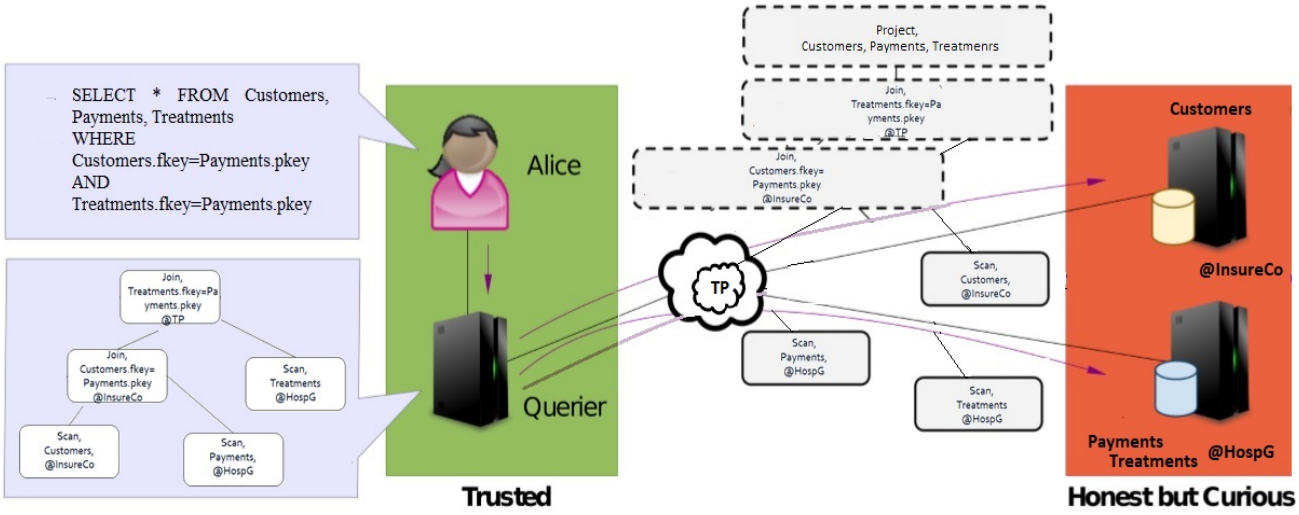

Figure 1: Use Case Scenario - classic strawman approach for query evaluation

the opportunity to take advantage of the performance of the cloud providers, while at the same time allowing data authorities to maintain the control over their data.

- We evaluate our solution via a time- and space-complexity analysis of the proposed algorithms, as well as a cost/benefit analysis.

Outline: The remainder of this paper is structured as follows: Section 2 describes our use case scenario and Section 3 the query execution model. Section 4 presents the implementation of a state-of-the-art authorization model to mitigate the presented privacy issues. In Section 5, we identify performance limitations of this implementation and propose an integration of the model into the distributed query optimizer PAQO to overcome these limitations. We discuss the time and space complexity of our solution in Section 6 and present a cost benefit analysis in Section 7. We discuss related work in Section 8 and provide a concluding summary in Section 9.

2 USE CASE SCENARIO

The following is an example that we will use throughout the paper to highlight the problem and our solution.

Example: Alice has her health insurance provided by the *InsureCo* insurance company, which maintains a database named *Sales*. Customers' SSNs, names, addresses, and insurance policies are stored in the table **Customers** on Sales. Furthermore, Alice is registered as a patient at the hospital *HospG*. HospG hosts a database server that contains relations for their financial department (**Payments** covering customers' SSNs, names, payments, coverage/lack of insurance) and the medical departments (**Treatments** relation). Alice has records in both databases. In addition, she is using an application that allows her to execute the following query to receive a full report on her medical and health insurance coverage situation:

*SELECT * FROM Customers, Payments, Treatments*
WHERE Customers.fkey=Payments.pkey AND
Treatments.fkey=Payments.pkey;

3 QUERY EXECUTION MODEL

In this paper we consider the model illustrated in Figure 1. We assume a multi-provider environment in which some servers are database servers, and others are computational servers participating in the execution of distributed queries. Users submit queries to a trusted provider. We assume that the trusted provider knows a priori all of the servers participating in the system via an expanded catalog that includes cached metadata about these remote servers.

When a user submits a query, the query optimizer at the invoking provider determines the optimal query plan and distributes parts of it to each server, database or computational, determined as being part of the query evaluation. The involved servers will evaluate their assigned portions of the query execution plan, combine their intermediate results, and return the final query results to the user. In this paper, we consider only selection, projection, and join queries (hereafter abbreviated as SPJ queries). We assume that third-party providers are able to carry out encrypted joins.

By introducing the opportunity to export rows or attributes, users may take advantage of third party providers' processing power while minimizing the leakage of confidential information. Encrypted visibility is considered as joins between relations and evaluating conditions can be performed over encrypted attribute values. Deterministic symmetric encryption can be accommodated during evaluation of equality conditions in selections and joins.

4 AUTHORIZATION MODEL

Every database server which owns the control over certain data is defined as its *data authority* (DA). We assume DAs protect all tables by means of access controls (e.g., using the industry standard RBAC [21]) and hence, users only obtain the results that they are authorized to see.

We also assume that providers, database, and computational servers in the system are *honest-but-curious passive* adversaries. These providers will correctly evaluate the query sub-plans assigned to them, and will return the correct results to the users, but in doing

so, they will see the data if it is plain text, and may try to infer information regarding the users. For this reason, we assume that each data authority establishes its own authorization requirements (i.e., visibility rules) for data release. For simplicity in this paper, and without the lack of generality, we assume that policies are specified at the table level. All attributes in a table are set with the same authorization policies.

Data is stored and processed in either *plain text* or *encrypted* form, and the controlling DA specifies the subject's level of visibility, as in [4]:

Definition 4.1. (Visibility Constraint) A visibility constraint defined by a data authority DA_i is a triple:

$$< subject, table, visibility >$$

where a *subject* can be another data authority DA_j, end-user u, or a provider S and the *visibility* can be

- ***plaintext*** – the subject has a complete visibility on the attribute's values;
- ***encrypted*** – the subject has a visibility only on an encrypted version of the attribute values;
- ***no visibility*** – the subject cannot view either the encrypted or the plaintext values of attributes.

Whenever no visibility restriction is defined against a subject, the *no visibility* rule is applied.

It is expected that users will have authorization only over plaintext attributes values, since users need to be able to access the responses to their own queries. Also, the data authorities storing a relation are expected to have plaintext authorization over the attributes of the very same relation. As stated above, providers and any other authorities have access rights as dictated by DA policies in order to prevent confidential information leakage.

As an example, Table 1 shows the visibility constraints in our use case scenario. In this, *HospG* considers that it can provide the following accesses to operate on its data:

(1) InsureCo - access only to the encrypted version of the data located in **Payments** and **Treatments**;
(2) TP (a third-party provider) - access only to the encrypted version of the data located in both **Payments** and **Treatments**;

At the same time, *InsureCo* on its own gives authorization only to TP to operate on the encrypted version of its data in **Customers**.

Table 1: Authorizations for the Subjects of our Example

Subjects	Plaintext	Encrypted
HospG	Payments Treatments	Payments Treatments
InsureCo	Customers	Customers Payments
TP	-	Payments, Treatments, Customers

The explicit visibility constraints on tables implicitly determine the visibility of views defined on these tables as well as of intermediate tables, resulting from the execution of the query, e.g., partial results. This means that a provider can process a query if the provider has either explicit or implicit visibility on all tables that are part of the query.

For instance, the following query can be executed at the database server S_1 where table R1 is stored and involves $R2.B$, which was derived from a table $R3$ stored at a database server S_2, only if S_1 has a permission to see $R3$.

SELECT R1.A FROM R1, R2 WHERE R1.A=R2.B;

In this example, attribute $R2.B$ is referred to as an implicit attribute in [4].

Definition 4.2. (Implicit Attribute) An attribute that does not necessarily appear in a resulting relation schema but that has to be taken into account in the computation of the relation is implicit.

Definition 4.3. (Implicit Visibility) The (implicit) visibility of an intermediate relation during the execution of a query is determined by the visibility of the implicit attributes.

In the next section, we will explain how our pruning algorithm takes into account potential implicit visibility and mitigates the risks of private information leakage caused by it.

5 AUTHORIZATION-AWARE QUERY OPTIMIZATION

In this section, we first present the shortcomings of the strawman approach, and then present our solution.

5.1 Strawman Approach

In [4] the authors present an authorization model that accommodates the strawman approach of creating query evaluation plans using an off-the-shelf query optimizer and then post-processing the resulting plan to enforce any required access controls. Effectively, the post-processing is a second optimization phase integrated into physical plan generation that assigns execution locations to each operation according to the DA's authorization requirements. However, this disconnect between optimization and the required authorization model leads to the creation of unnecessarily inefficient query plans.

Consider again our running example:

*SELECT * FROM Customers, Payments, Treatments*
WHERE Customers.fkey=Payments.pkey AND
Treatments.fkey=Payments.pkey;

Let us assume that **Customers** has a cardinality of 100 tuples, **Payments** 5,000,000 tuples, and **Treatments** 200 tuples. We further assume that the joins are performed on attributes in **Customers** and **Treatments** that are foreign keys to **Payments**.

In optimizing this query, a traditional SQL query optimizer (the first phase of a two-phase approach) would first join **Customers** and **Payments**, as **Customers** is half the size of **Treatments**, resulting in the query plan shown in Figure 2.

Post-processing this plan to support the authorization restrictions would disallow joining the tables **Customers** and **Payments**

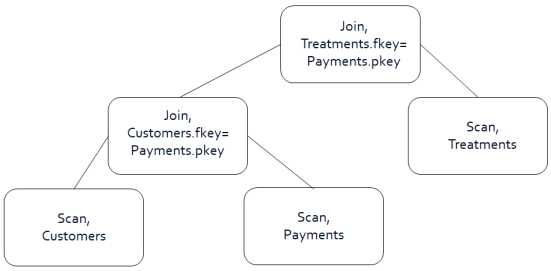

Figure 2: Tree generated by the traditional optimizer - PostgreSQL

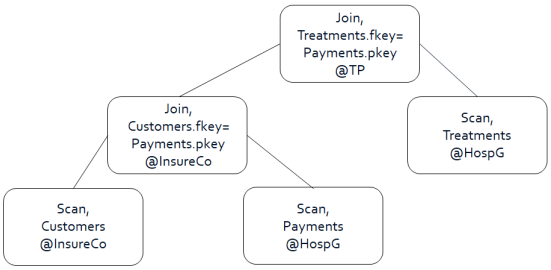

Figure 3: Inefficient site-assigned plan based on the authorization model and the tree generated by the traditional optimizer - PostgreSQL

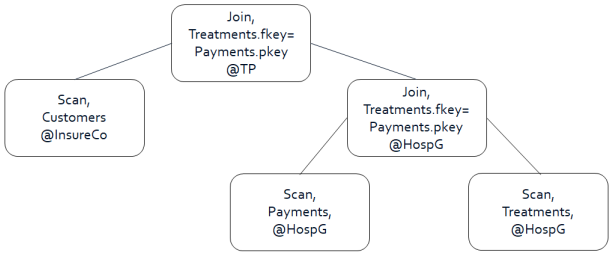

Figure 4: Efficient site-assigned plan based on our algorithm and the authorization model

at *HospG*. Hence, the second phase would logically choose *InsureCo* as a location to evaluate this join (as in Figure 3), though this comes at a great cost to query performance. Under this plan, all of **Payments** (5,000,000 tuples) must be shipped over the network to *InsureCo*, incurring a great cost to not only total query evaluation time, but also network bandwidth utilization. Further, according to the presented authorization scheme, the cost of encryption of all the 5,000,000 tuples will also be incurred.

5.2 Authorization-Aware Approach

To avoid the shortcomings of two-phase optimization just discussed, we take a synergistic approach to query optimization, accounting for the triple, encryption/decryption, location, and query performance during the query optimization process. We observe that an *authorization-aware* (AA) approach reduces the execution time of the query by joining first **Payments** and **Treatments** together at *HospG* and by utilizing indexes on **Payments** to speed up the join and avoid scanning the entire **Payments** table. Also, it will avoid the need of data shipping and encryption of all the 5,000,000 tuples. Clearly, such a plan (Figure 4) can only be discovered by considering the authorizations and the possible locations when determining the join order in a query plan.

In order to avoid implementing an authorization-aware optimizer from scratch, we built upon the PAQO query optimizer[12]. PAQO is a version of the PostgreSQL query optimizer that uses user-specified requirements and preferences as additional optimization metrics. We use PAQO to encode authorization constraints as inputs to the query optimization process. Our modifications allow us to track implicit visibility, as described in the previous section, by bookkeeping additional information on the derived tables during plan enumeration. This data is then used in our pruning algorithm.

5.2.1 Node Descriptors. In order to achieve our goal, we define slightly modified and extended versions of the node descriptors used by PAQO to identify the portions of the query that should be specially handled according to the authorization scheme in place.

Query plans are trees of nodes that represent relational algebra operations. Each node descriptor will be a quadruplet

$$< op, params, p, auth >$$

where *op* is the operation represented by the node, *params* represents the parameters to that operation, p is the principal (e.g., a database server or a third party provider) assigned to evaluate the operation, and *auth* identifies whether the principal has encrypted or plaintext visibility over the data in place.

These node descriptors are used to *match* query tree nodes that the data authorities would like to be evaluated in a specific way according to the required authorization model. $*$ will be used as a general wildcard. Setting *op, params*, or p to be in a node descriptor will cause that portion of the node descriptor to match any value in the corresponding portion of a query plan node. For our running example, the authorizations will be defined as follows:

$< *, \{Payments\}, HospG, PlainText >,$
$< *, \{Payments\}, HospG, Encrypted >$
$< *, \{Treatments\}, HospG, Encrypted >$
$< *, \{Treatments\}, HospG, PlainText >$
$< *, \{Customers\}, InsureCo, PlainText >$
$< *, \{Customers\}, InsureCo, Encrypted >$
$< *, \{Payments\}, InsureCo, Encrypted >$
$< *, \{Payments\}, TP, Encrypted >$
$< *, \{Treatments\}, TP, Encrypted >$
$< *, \{Customers\}, TP, Encrypted >$

The list of node descriptors will define our Authorization list used later in our modified PAQO algorithm.

5.2.2 Query Optimization Considering Predefined Authorizations. By adopting a greedy, heuristic approach to optimizing queries, PAQO is able to efficiently produce highly preferred plans. This approach allows PAQO to protect the intension of reasonably-sized user queries. As we will see in the complexity analysis, the implemented modifications incur only a linear overhead of checking authorization requirements for each plan generated over the PAQO algorithms.

In Algorithm 1, we present a modified version of the classic dynamic programming-based query optimization algorithm that includes data owners' authorization requirements in the optimization process and also accounts for the use of multiple, distributed evaluation sites in constructing query plans. The sets of usable evaluation sites within the distributed system and the required authorizations are taken as input to the optimization. In addition to being needed to prune out unusable access paths (Line 4 of Algorithm 1), required authorizations are needed by the JOINPLANS function (Line 15). Additional bookkeeping is implemented for every new derived table (node) needed by the PRUNEPLANS function (Line 2).

First, ACCESSPLANS (Algorithm 2) must iterate through all possible evaluation locations in order to ensure that efficient plans are found (Line3). Two plans are enumerated per location one considering the data will be encrypted and another one considering the case the data is available in plain text. This is the only thing that differs this function by the original ACCESSPLANS function used by the traditional/PAQO optimization algorithm.

An iteration through all evaluation sites is similarly performed in JOINPLANS (Algorithm 3). In this case, however, there is no easy way to reduce the list of potential sites. All sites must be explored, not just the sites that evaluate the children of a new join node. Consider two sub-plans p1 evaluated at S1 and p2 evaluated at S2. Let us assume that p1 and p2 are significantly faster than any other plans realize their respective relations. Let us further assume that the condition to be used in joining p1 and p2 is required by our authorization scheme to be evaluated at S3. If we make the reasonable assumption that the cost to ship the results of both p1 and p2 to S3 and perform a join is less than the cost to ship only one of the results and perform a crossproduct, then the fastest plan upholding required authorizations cannot be found by iterating only through sites evaluating the children of a prospective join.

ACCESSPLANS and JOINPLANS must check to make sure each newly produced plan does not violate any authorization requirement (Lines 7 and 9 (encrypted data), or 11 and 12 (plaintext data)). Any plans that violate an authorization requirement are promptly pruned from the search space. Finally, the classical PRUNEPLANS function (Algorithm 4) must be slightly modified. For every node of a potential query plan we will keep a parent/parents of base tables from which the new table was derived. Based on this bookkeeping we are going to check for implicit data leakage in the PRUNEPLANS function (Line 2).

Considering the presented algorithm for our running example, the final query plan produced by our optimizer will look as the one presented in Figure 4. By joining first **Payments** and **Treatments** together at *HospG*, indexes on **Payments** can again be utilized to speed up the join and avoid scanning all of table **Payments**. Also, we will avoid the data shipping and encryption of all the 5,000,000 tuples.

5.2.3 Computing and Distributing Assignments. Query operation assignments should also cover establishing and distributing keys for attributes which need to be encrypted/decrypted during the query plan execution. The only requirement about the key establishment is that attributes involved in some predicate in encrypted form need to be encrypted with the same key. The key associated with an attribute should be distributed only to the subject responsible

Algorithm 1 Dynamic programming algorithm that accounts for authorization requirements and covers implicit visibility restrictions

Require: SPJ query q on relations $R_1...R_n$, with selection/projection/join conditions $C_1...C_m$, required authorizations $AUTH_1...AUTH_r$
Require: Possible evaluation sites $S_1...S_s$
Require: A list of nodes (derived tables) D
1: $subplans \leftarrow EMPTY\_LIST$
2: $subplans[1].add(\{EMPTY\_LIST\})$
3: **for** $i = 1$ to n do **do**
4: $optPlan[\{\{R_i\}] \leftarrow$
5: $accessPlans(R_i, \{AUTH_1...AUTH_r\})$
6: **for** $i = 2$ to n do **do**
7: **for all** P $\{R_1...R_n\}$ such that $|P| = i$ do **do**
8: $optPlan[S] \leftarrow 0$
9: **for all** O $\subset$ P do **do**
10: $l \leftarrow optPlan[O]$
11: $r \leftarrow optPlan[P/O]$
12: $rs \leftarrow \{AUTH_1...AUTH_r\}$
13: $cs \leftarrow \{C_1...C_m\}$
14: $ss \leftarrow \{S_1...S_s\}$
15: $optPlan[P] \leftarrow optPlan[P] \cup jps$
16: $prunePlans(optPlan[P])$
return $optPlan[R_1...R_n]$

Algorithm 2 ACCESSPLANS: Access plan enumeration with authorization checking

Require: A relation R_i to enumerate access plans for
Require: A list of required authorizations $AUTH_1...AUTH_r$
Require: The conditions $C_1...C_M$ specified as part of the query
Require: Possible evaluation sites $S_1...S_s$
1: $aPlans \leftarrow 0$
2: $c \leftarrow POSSIBLE\_CONDS(R_i, \{C_1, ..., C_m\})$
3: **for** $k = 1$ to s **do**
4: **for all** physical scan operators, po **do**
5: $New_{encr} \leftarrow NEWSCAN(po, R_i, c)$
6: $aPlans \leftarrow sPlans \cup \{New_{encr}\}$
7: $New_{pText} \leftarrow NEWSCAN(po, R_i, c)$
8: $aPlans \leftarrow aPlans \cup \{New_{encr}\}$
9: **if** not VIOLATES AUTH($New_{encr}\{AUTH_1...AUTH_r\}$ **then**)
10: $aPlans \leftarrow aPlans \cup \{New_{encr}\}$
11: **if** not VIOLATES AUTH($New_{pText}\{AUTH_1...AUTH_r\}$ **then**)
12: $aPlans \leftarrow aPlans \cup \{New_{pText}\}$
return aPlans

for the encryption and decryption of the attribute. Also, encryption/decryption operation assignment should be considered.

Based on our approach the following query operation assignment plan is considered:

(1) Operation assignment based on the presented distributed query authorization algorithm

(2) Post-order visit of the query plan to extend it with the encryption/decryption operations
(3) Establishment of the required keys
(4) Distribution of the sub-queries and the keys to the involved subjects

In our work we consider a negligible cost for the encryption and decryption operations (e.g., if AES is used). Finally, it is to be noted that the end user requesting a query execution is required to be authorized to access all the data which is input to the query.

Algorithm 3 JOINPLANS: Join enumeration pseudocode

Require: A set of plans that will make up the left side of a new root join $Left_1 ... Left_u$
Require: A set of plans that will make up the right side of a new root join $Right_1 ... Right_v$
Require: A list of authorizations $AUTH_1 ... AUTH_r$
Require: The conditions $C_1 ... C_m$ specified as part of the query
Require: Possible evaluation sites $S_1 ... S_s$
1: $jPlans \leftarrow 0$
2: **for** $i = 1$ to u **do**
3: **for** $j = 1$ to v **do**
4: $c \leftarrow POSSIBLE\_CONDS(Left_i, Right_j, \{C_1 ... C_m\})$
5: **for** $k = 1$ to s **do**
6: **for all** physical join operators, po **do**
7: $New_{encr} \leftarrow Left_i \bowtie po, cRight_j$ at S_k
8: **if** not VIOLATES AUTH $(New_{encr}\{AUTH_1 ... AUTH_r\})$ **then**
9: $jPlans \leftarrow jPlans \cup \{New_{encr}$
10: $New_{pText} \leftarrow Left_i \bowtie po, cRight_j at S_k$
11: **if** not VIOLATES AUTH $(New_{pText}\{AUTH_1 ... AUTH_r\})$ **then**
12: $jPlans \leftarrow jPlans \cup \{New_{pText}$
return jPlans

Algorithm 4 PRUNEPLANS : A function to prune dominated plans from a given join level, considering the implicit visibility

Require: A list Q of query plans joining the same number of base relations.
Require: A list of nodes (derived tables) D
1: **for all** p ϵ Q **do**
2: $CHECK\_IMPLICIT(p, D)$
3: $reject \leftarrow False$
4: **for all** $other \epsilon Q | p != otherdo$ **do**
5: **if** $ROOT\_SITE(p) == ROOT\_SITE(other)$ **then**
6: **if** p has more interesting sort order **then**
7: **if** cost(p) < cost(other) **then**
8: Q.remove(other)
9: **else if** cost(plan) > cost(other) **then**
10: $reject \leftarrow True$
11: break
12: **if** reject then **then**
13: Q.remove(p)

6 ALGORITHM COMPLEXITY

In our complexity analysis, let us assume only one physical scan operator and only one physical join operator are available to the optimizer. Also, no interesting sort orders can be taken advantage of during optimization, and no authorization requirements can be violated by any query plans. As described previously, the changes to the traditional PostgresSQL algorithm needed to support the authorization requirements are the addition of distributed evaluation sites and the use of authorization requirements to prune violating plans.

*Lemma 1:The time complexity of our AA (Authorization-Aware) query optimization algorithm for queries over n base relations with r required authorizations can be evaluated on s potential servers is O(s^3 *3^n *4*2r).*

Proof. Let us assume the worst case scenario when there is no possible pruning. All possible locations have plain text visibility on all the data. Each entry in optPlan corresponds to at most s plans (note there cannot be multiple plans in the same entry that differ only by sort order or physical operator). There are $2s$ different scan plans for each relation, and $2s$ different join plans for each set of base relations–We need to consider both cases, encrypted and plain text visibility. Hence, for each call to JOINPLANS, all $2s$ plans in optPlan[O] (Line 10, Algorithnm 1) must be combined with all $2s$ plans from optPlan[P/O] (Line 11, Algorithnm 1) with all s sites considered for evaluation, increasing the runtime complexity by a factor of $4s^3$. As per our assumptions, no plans can be pruned due to requirement violation, and hence such pruning has no effect on the time complexity. Authorization requirements must be checked for each plan that is realized, however, and that does have a slight effect on the runtime complexity. For each plan realized, it must be determined if it violates any of the r authorization requirements. This leads to the addition of the $2r$ term in our run time complexity. □

Lemma 2: The space complexity of our query optimization algorithm for queries over n base relations with r required authorization constraints that can be evaluated on s potential servers is

$$O((2s*8*2^n + 8*s^3)*2r).$$

Proof. Our presented algorithm must store $2s$ times as many plans as the PostgreSQL dynamic programming algorithm (the $2s$ plans in each entry of optPlan), and must further have $8*s^3$ memory available for the JOINPLANS function. As per our algorithm we must further save, for each plan, a record of which authorization requirement it upholds/violates. □

Our algorithm imposes only a linear overhead of checking authorization requirements for each plan generated over the O(s^3*3^n) bound established in the literature for dynamic programming based optimization of distributed queries.

7 COST BENEFIT ANALYSIS

In our analysis, we have used the latest statistics from a large university medical center (UPMC) [22]. In Figures 5 to 8, we examine the performance of the approach described in [4] as well as our

proposed approach for a range of table sizes and authorization setups.

According to the latest UPMC statistics [22], the number of outpatient visits during the last year is 4.7 million. Based on these statistics, in our analysis we are going to fix the number of selected records in table **Payments** to 4.7 millions and vary only the number of records for tables **Customers** and **Treatments**. We decided to concentrate ourselves only on the data shipping cost based on the observation that it produces the biggest challenges for the respective query [2], utilizing only three tables. Referring to another statistic provided by UPMC (members covered by UPMC insurance services), we are going to set the upper limit of selected records in both tables, **Customers** and **Treatments** to 3.4 million. For simplicity, the cost unit we accommodate in our graphics is "number of tuples shipped to another site." Even if the encryption cost is negligible, for completeness we considered additional cost of 0.1 per tuple whenever encryption/decryption is expected to happen.

In Figures 5, 6, and 7, we follow the authorizations from our use case scenario. In Figure 8, we slightly change the authorizations giving the option to *HospG* to operate on the encrypted version of **Customers** relation. By doing this we show the variation between the two explored costs when we have different level of access restrictions applied.

Our calculations represented in Figures 5, 6, 7, and 8 show that our revised PAQO algorithm outperforms the approach described in [4]. By taking into consideration the authorization requirements and the data storage locations during the query optimization phase, the optimizer is able to consider more potential plans (looking at operations being run on both encrypted attributes or plaintext) and prune early those prohibited by the established DA policies. This early pruning will help especially in the cases where the authorization setup is more restrictive.

As Figure 5 shows, for our running example, our proposed authorization-aware distributed query optimizer will manage to reduce the cost by 4 millions of tuples. We would like to point out that the part of the graphic where both costs have the same value is the part where the classical optimizer receives less tuples from the **Treatments** relation than the **Customers** one. Because of this reason, the plans produced by both algorithms are the same, respectively the cost would be the same.

8 RELATED WORK

In this section we briefly review prior work related to distributed query processing, authorization enforcement in database systems, and data encryption in distributed environments.

Distributed Query Processing The optimization and processing of queries over distributed database systems has been an area of active research for several decades. A huge amount of work on distributed database query optimization has focused primarily on decreasing optimization time and improving the plans generated. Query optimization was first introduced by Yao and Hevner [16]. In the late 1970s, authors used heuristic with exhaustive enumeration approach to optimize the queries. In 1980s researchers as Ceri and Palagatti [3], Zhou, Chen, Li and Yu [24], Peter Apers [1], Lam and Martin [17] proposed the different query optimization strategies. The query optimization model was further extended by Rho and

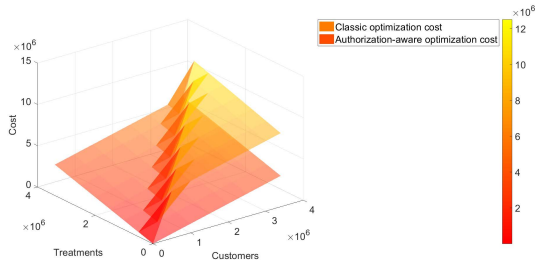

Figure 5: Cost variation considering the authorizations provided in our use case scenario

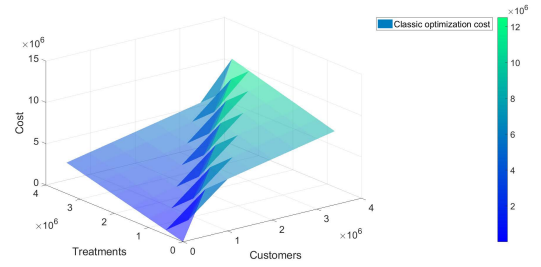

Figure 6: Classic optimization cost variation

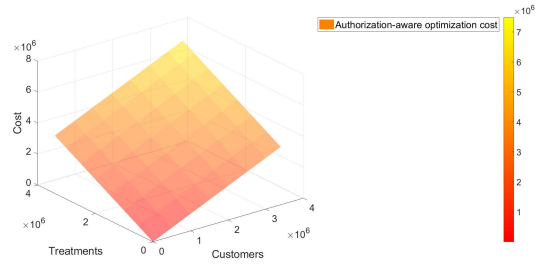

Figure 7: Cost based on our algorithm, considering data shipment and encryption

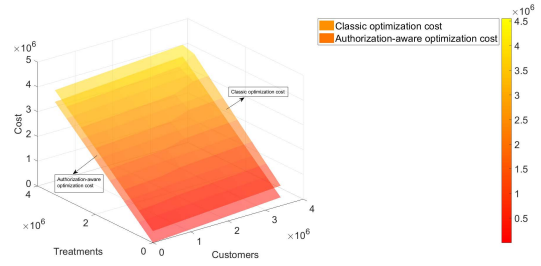

Figure 8: Cost variation considering the authorizations provided in our use case plus giving HospG the opportunity to operate on the encrypted version of the Customers relation

March in 1995 [19]. During 21st century, Ahmet Cosar [20], Zehai Zhou [25] used Genetic Algorithm to optimize the distributed queries. Traditionally Exhaustive Enumeration with some heuristics algorithms (Dynamic Programming, Branch and Bound, Greedy Algorithm, etc.) was dominantly used to optimize queries [15]. In

PAQO [12] the authors present the first distributed query optimizer to include user specified constraints as additional optimization metrics. While we take advantage of its pruning approach, we use it to prune plans which violate authorization requirements explicitly or implicitly (during computation).

Authorization Enforcement in Database Systems In [4], [5] the authors develop a novel approach for the specification and enforcement of authorizations that enables controlled data sharing for collaborative queries in the cloud. Data authorities can establish authorizations regulating access to their data distinguishing three visibility levels (no visibility, encrypted visibility, and plain text visibility). Authorizations are enforced in the query execution by possibly restricting operation assignments to other parties and by adjusting visibility of data on-the-fly. Thus, users and data authorities are enabled to fully enjoy the benefits and economic savings of the competitive open cloud market, while maintaining control over data. While the proposed model is novel, it has a significant limitation in its implementation in a distributed environment. In this work we mitigated its shortcomings by incorporating the required authorizations into the query optimizer heuristics.

Data Encryption in Distributed Environment. In [6] the authors create design techniques to verify the integrity of query results computed by potentially untrusted servers. Other privacy-related works [7, 18] explore the use and support of encryption to protect data during query execution. These solutions are only complimentary to our adopted approach which integrates authorization requirements as a metric into the distributed optimization process.

9 CONCLUSION

In this paper, we identify the limitations of a recently developed state-of-the-art data authorization model targeted for distributed execution of SQL queries in a multi-provider environment such as on the cloud. In this, a strawman approach was utilized to post-process an optimized query evaluation plan, produced by an off-the-shelf optimizer, in order to enforce the access control policies of data providers during the distributed execution of the query.

While that model provides a flexible approach enabling controlled collaborative query execution in a distributed environment, we showed that it achieves this at a potentially unreasonable high cost, making it unpractical for general adoption. Its major shortcoming is that it fails to fully exploit third-party providers for the execution of the distributed queries.

In response, we propose integration of the authorization model into the distributed query optimization process. In this way, we can maximize performance while maintaining data privacy. We explore the benefits of our proposed *authorization-aware* approach through the use of examples and a time and space complexity analysis. Our future work will include a full-featured implementation of the authorization-aware optimizer and extensive experimental results covering broader use case scenarios.

ACKNOWLEDGMENTS

We would like to thank Nicholas Farnan for providing the PAQO code and valuable guidance on its use and modification. This work was supported, in part, by the NSF under awards CNS–1253204, CNS–1704139 and CPS–1739413, and by NIH under award U01HL137159. This paper does not represent the views of NSF and NIH.

REFERENCES

[1] Peter M.G. Apers, Alan R. Hevner, and S. Bing Yao. 1983. Optimization Algorithms for Distributed Queries. (1983).
[2] Berkley. 2018. Latency Numbers Every Programmer Should Know. (2018). https://people.eecs.berkley.edu/~rsc/research/interactive_latency.html
[3] Stefano Ceri and Giuseppe Pelagatti. 1982. Allocation of Operations in Distributed Database Access. *IEEE Trans. Comput.* C-31 (1982), 119–129.
[4] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Giovanni Livraga, Stefano Paraboschi, and Pierangela Samarati. 2017. An Authorization Model for Multi Provider Queries. *Proc. VLDB Endow.* 11, 3 (Nov. 2017).
[5] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. 2011. Authorization Enforcement in Distributed Query Evaluation. *J. Comput. Secur.* 19, 4 (Dec. 2011), 751–794.
[6] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. 2016. Efficient Integrity Checks for Join Queries in the Cloud. *Journal of Computer Security (JCS)* 24, 3 (2016), 347–378.
[7] Sabrina De Capitani di Vimercati, Sara Foresti, Giovanni Livraga, and Pierangela Samarati. 2016. Practical Techniques Building on Encryption for Protecting and Managing Data in the Cloud. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*. 205–239.
[8] Sabrina De Capitani di Vimercati, Sara Foresti, and Pierangela Samarati. 2015. Data Protection in Cloud Scenarios. In *Data Privacy Management, and Security Assurance - 10th International Workshop, DPM 2015, and 4th International Workshop, QASA 2015, Vienna, Austria, September 21-22, 2015. Revised Selected Papers*. 3–10.
[9] Nicholas L Farnan. 2015. Efficient, Locally-Enforceable Querier Privacy for Distributed Database Systems. (January 2015).
[10] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. [n. d.]. Enabling Intensional Access Control via Preference-aware Query Optimization.
[11] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2011. Don'T Reveal My Intension: Protecting User Privacy Using Declarative Preferences During Distributed Query Processing. In *Proceedings of the 16th European Conference on Research in Computer Security (Proc. ESORICS'11)*. 628–647.
[12] Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis, and Ting Yu. 2013. PAQO: A Preference-aware Query Optimizer for PostgreSQL. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1334–1337.
[13] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. 2014. PAQO: Preference-aware query optimization for decentralized database systems. In *2014 IEEE 30th International Conference on Data Engineering*. 424–435.
[14] Nicholas L. Farnan, Adam J. Lee, and Ting Yu. 2010. Investigating Privacy-aware Distributed Query Evaluation. In *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society (Proc. WPES '10)*. 43–52.
[15] Reza Ghaemi, Amin Milani Fard, Hamid Tabatabaee, and Mahdi Sadeghizadeh. 2008. Evolutionary query optimization for heterogeneous distributed database systems. *World Academy of science* 43 (2008), 43–49.
[16] Alan R. Hevner and S. Bing Yao. 1979. Query Processing in Distributed Database System. *IEEE Transactions on Software Engineering* SE-5 (1979), 177–187.
[17] Patrick Martin, K. H. Lam, and Judy I. Russell. 1990. An Evaluation of Site Selection Algorithms for Distributed Query Processing. *Comput. J.* 33, 1 (1990), 61–70.
[18] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 85–100.
[19] Sangkyu Rho. 2015. Information Systems (ICIS) 1231-1995 Designing Distributed Database Systems for Efficient Operation.
[20] Ender Sevinç and Ahmet Coşar. 2011. An Evolutionary Genetic Algorithm for Optimization of Distributed Database Queries. *Comput. J.* 54, 5 (May 2011), 717–725.
[21] S. Tran and M. Mohan. 2006. Security information management challenges and solutions. (July 2006). http://www.ibm.com/developerworks/data/library/techarticle/dm-0607tran/index.html
[22] UPMC. 2018. By the Numbers: UPMC Facts and Figures. (2018). https://www.upmc.com/about/facts/numbers
[23] Sabrina De Vimercati, Sara Foresti, and Pierangela Samarati. 2015. Data Security Issues in Cloud Scenarios. In *Proceedings of the 11th International Conference on Information Systems Security - Volume 9478 (ICISS 2015)*. 3–10.
[24] Lin Zhou, Yan Chen, Taoying Li, and Yingying Yu. 2012. The Semi-join Query Optimization In Distributed Database System. (2012).
[25] Zehai Zhou. 2007. Using Heuristics and Genetic Algorithms for Large-scale Database Query Optimization.