QueryGuard: Privacy-preserving Latency-aware Query Optimization for Edge Computing

Runhua Xu *, Balaji Palanisamy † and James Joshi ‡

School of Computing and Information
University of Pittsburgh, Pittsburgh, PA USA
⋆runhua.xu@pitt.edu, †bpalan@pitt.edu, ‡jjoshi@pitt.edu

Abstract—The emerging edge computing paradigm has enabled applications having low response time requirements to meet the quality of service needs of applications by moving the computations to the edge of the network that is geographically closer to the end-users and end-devices. Despite the low latency advantages provided by the edge computing model, there are significant privacy risks associated with the adoption of edge computing services for applications dealing with sensitive data. In contrast to cloud data centers where system infrastructures are managed through strict and regularized policies, edge computing nodes are scattered geographically and may not have the same degree of regulatory and monitoring oversight. This can lead to higher privacy risks for the data processed and stored at the edge nodes, thus making them less trusted. In this paper, we show that a direct application of traditional performance-based query optimization techniques in edge computing can lead to unexpected data disclosure risks at the edge nodes. We propose a new privacy-preserving latency-aware query optimization framework, QueryGuard, that simultaneously tackles the privacy-aware distributed query processing problem while optimizing the queries for latency. Our experimental evaluation demonstrates that QueryGuard achieves better performance in terms of execution time and memory usage than conventional distributed query optimization techniques while also enforcing the required constraints related to data privacy.

Index Terms—query optimization; data privacy; latency awareness; edge computing; database management

I. INTRODUCTION

The emerging edge computing paradigm has enabled applications having low response time requirements to meet quality of service needs of applications by moving computations to the edge of the network that is geographically closer to the end-users and end-devices [1]–[3]. A key distinguishing feature of edge computing is its ability to store and process large amounts of data on servers and computing units located closer to the data sources such as sensors and mobile devices. This enables it to provide low latency services for highly interactive applications. For instance, augmented reality applications with real-time computation requirements can be deployed as an edge computing application to meet the response time requirements. In a highly distributed data processing environment such as those in the Internet-of-things (IoT), edge computing provides a natural solution to deal with the decentralized and low-latency computation of data generated in the IoT devices.

Distributed database management approaches offer an efficient ways to manage and process large amounts of decentralized data for data-driven applications [4]–[11]. However, adopting distributed database techniques in edge computing brings new challenges and concerns. First, in contrast to cloud data centers where cloud servers are managed through strict and regularized policies, edge nodes may not have the same degree of regulatory and monitoring oversight. This may lead to higher privacy risks compared to that in cloud servers. In particular, when dealing with a join query, traditional distributed query processing techniques sometimes ship selected or projected data to different nodes, some of which may be untrusted or semi-trusted. Thus, such techniques may lead to greater disclosure of private information within the edge nodes. Traditional query processing techniques in the distributed environments aim at optimizing the queries in terms of the most efficient query plan and do not focus on addressing the privacy concerns in distributed settings [5], [6], [8]–[11]. Although cryptography based solutions have been adopted in database systems in [12], [13], such schemes either incur a significant computation cost for implementing the crypto operations or require the use of trusted third party to support the operation [4], [7]. Secondly, as edge computing nodes are scattered geographically with varying degrees of network connectivity in terms of network bandwidth and latency, optimizing distributed query processing in edge computing requires a special emphasis on network latency compared to that in traditional query processing where the emphasis is primarily on minimizing the query computation time.

In this paper, we propose QueryGuard, a privacy-preserving, latency-aware query optimization framework, to tackle the challenges of join query optimization in an edge computing environment. Our proposed work deals with both the privacy concerns as well as latency optimization for distributed join query processing in edge computing environments. The proposed query optimization mechanism generates optimal query execution plans that ensure users’ privacy preferences on their sensitive data stored in edge nodes during the query execution phase. In particular, the mechanism automatically controls the movement of sensitive data in a cross-site join operation so as to avoid the sensitive data being stored or disclosed to an untrustworthy node in the decentralized computing infrastructure. The proposed query optimization framework also optimizes for the latency of the join queries by dynamically considering the network characteristics of the edge computing environment. We experimentally evaluate the
performance of the proposed query optimization techniques and the results demonstrate that the proposed methods achieve better performance in terms of execution time and memory usage compared to conventional distributed query optimization techniques while also enforcing the privacy constraints.

II. BACKGROUND AND MOTIVATION

Edge computing refers to computing infrastructures that enable computations and data processing tasks to be performed at the edges of the network, closer to the data sources, allowing low latency applications to meet their short response time requirements [2]. Fig.1 illustrates an example where several edge nodes are distributed at the edge of the Internet. This helps provide storage and computing services for IoT sensors closer to the edge of the network.

A join query \( t_1, s_1, s_2 \) may be requested from a mobile application at the edge node \( E_2 \). As there exists multiple query plans for this join query, the query optimizer chooses the optimal query plan that minimizes the query execution time. For instance, there are three possible join query statements for the query: \( t_1 \bowtie s_1 \bowtie s_2, t_1 \bowtie s_2 \bowtie s_1, \) and \( s_1 \bowtie s_2 \bowtie t_1 \), based on the order of the join operation. For each query, several query execution plan candidates can be generated from the query statement based on different permutations of the join order, projection, and selection operations. In an edge computing environment, each query plan incurs a different query processing and latency cost as stream/relations are stored and placed in different edge nodes.

A. Latency-aware Query Optimization

Even though traditional distributed query processing techniques could be used to manage stream/relational data, they are not directly suitable for applications that require low response time such as real-time interactive applications and as a result, such approaches yield a sub-optimal performance in edge computing environments. Suppose that a mobile-based application supporting audio-based walking guide is used by people who are blind for navigation; when they are walking around, even a small latency delay may result in a serious deviation from the correct path. Such latency sensitive applications usually have a higher requirement on latency optimization, which can be supported by using an edge computing based approach.

As another example to illustrate the benefits of adopting edge computing for applications requiring low latency query processing, suppose that edge nodes \( E_1, E_2, E_3 \) are located in city \( A \) as shown in Fig.1. Let the distance between \( E_1 \) and \( E_2, E_3 \), and \( E_1 \) and \( E_3 \) be \( d_{e_1,e_2}, d_{e_2,e_3} \) and \( d_{e_1,e_3} \) miles, respectively. Let the closest cloud data center \( B \) be \( d_{a,b} \) miles away from city \( A \). We consider the estimated network traffic latency of each location as in Table I based on [14].

Table I: Simulated Locations and their Network Latency

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance</th>
<th>Latency (ms)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 - E_2 ) ( d_{e_1,e_2} )</td>
<td>0.022 * ( d_{e_1,e_2} )+ 4.862</td>
<td>( t_{e_1,e_2} ) = 5.082</td>
<td></td>
</tr>
<tr>
<td>( E_2 - E_3 ) ( d_{e_2,e_3} )</td>
<td>0.022 * ( d_{e_2,e_3} )+ 4.862</td>
<td>( t_{e_2,e_3} ) = 5.202</td>
<td></td>
</tr>
<tr>
<td>( E_1 - E_3 ) ( d_{e_1,e_3} )</td>
<td>0.022 * ( d_{e_1,e_3} )+ 4.862</td>
<td>( t_{e_1,e_3} ) = 5.102</td>
<td></td>
</tr>
<tr>
<td>( A - B ) ( d_{a,b} )</td>
<td>0.022 * ( d_{a,b} )+ 4.862</td>
<td>( t_{a,b} ) = 28.862</td>
<td></td>
</tr>
</tbody>
</table>

Note that the latency of network traffic is estimated based on the distance using a linear model: \( y = 0.022x+4.862 \) with coefficient of determination \( (R^2 = 0.907) \) proposed in [14].

† The distance between the data center and the city is assumed to be 1000 miles, while the distance between edge nodes is 10, 20, and 15 miles, respectively.

Table II: Simulated Parameter settings and values

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>30 min</td>
<td>Time interval of the query</td>
</tr>
<tr>
<td>( v_{c_1} )</td>
<td>1 KB/min</td>
<td>Speed of stream data generating at edge ( E_1 )</td>
</tr>
<tr>
<td>( v_{c_2} )</td>
<td>2 KB/min</td>
<td>Speed of stream data generating at edge ( E_2 )</td>
</tr>
<tr>
<td>( v_{c_3} )</td>
<td>3 KB/min</td>
<td>Speed of stream data generating at edge ( E_3 )</td>
</tr>
<tr>
<td>( v_{net} )</td>
<td>100 Mbit/s</td>
<td>Ethernet speed</td>
</tr>
<tr>
<td>( T )</td>
<td>10 ms</td>
<td>Query time in a single machine</td>
</tr>
</tbody>
</table>

(i) Edge-based approach. The total estimated time includes the maximum time of shipping data from \( E_2, E_3 \) to \( E_1 \) and the time of query processing in \( E_1 \), as follows:

\[
t_{edge} = \max_{i \in \{e_2,e_3\}, j \in \{e_1,e_2,e_3\}} (v_i t/v_{net} + t_j) + T,
\]

where \( v_i \) represent the data generation speed and \( v_j \) is the latency time.

(ii) Cloud-based approach. The total estimated time includes the maximum time of shipping data from \( E_1, E_2 \) and \( E_3 \) to \( B \), the time of query processing in \( B \), and the time of returning query results:

\[
t_{cloud} = \max_{i \in \{e_1,e_2,e_3\}} (v_i t/v_{net}+t_{a,b})+T+\sum v_i t/v_{net}+t_{a,b},
\]

where \( v_i \) represent the data generation speed.

As a result, the estimated query time of cloud-based approach \( t_{cloud} \) is about 84.818 ms, while the edge-based approach \( t_{edge} \) is about 22.223 ms using the values in Table II. This indicates that the cloud-based approach incurs nearly 4 times the cost of the edge-based approach.

B. Privacy-preserving Query Processing

Although edge computing provides significant advantages in tackling latency issues for short response time applications, it also brings new privacy challenges when deployed in distributed, semi-trusted computing environments. In contrast to cloud data centers where cloud servers are managed through strict and regularized policies, edge nodes may not have the same degree of regulatory and monitoring oversight; hence, edge nodes may lead to higher privacy risks compared to cloud servers. In particular, when dealing with a join query, traditional distributed query processing techniques may ship selected or projected data to various trusted or semi-trusted nodes, thus, increasing data privacy risks at the edge nodes. We illustrate the problem more clearly using the example shown in Fig.1 where \( E_1, E_2, E_3 \) are marked as private edge nodes.
the adversary at the public edge node.

As a result, the query plan candidates (c) and (d) have privacy leakage risks as a result of shipping sensitive information \( \text{attr}_2 \) and \( \text{attr}_3 \) to a public edge node \( E_4 \), which is indicated with red font and rectangle in Fig. 2.

**Privacy Disclosure Specification.** Suppose if some of public edge nodes, e.g., \( E_4 \), may be controlled by the adversary who tries to collect users’ private information, then even though the adversary cannot control the private edge nodes where sensitive data is stored, the adversary can still acquire these sensitive data as part of the intermediate data transferred during the join query execution. In this example, the query plan candidate (c) and (d) may be chosen as the optimal query plan if they have the lowest estimated cost. As a result, the adversary at the public edge node \( E_4 \) can acquire the intermediate sensitive data even if he does not have access to edge nodes where the sensitive data is stored. Here, the key challenge is determining how to satisfy data privacy as well as low latency processing requirements when generating query execution plans such that both the objectives are met simultaneously.

**Privacy Model.** The privacy model in our framework includes two parts: (i) using privacy preference approach to control the data shipment scope and (ii) using privacy privilege levels to control the data shipment direction among the edge nodes.

**Privacy Level.** In the proposed model, each edge node is assigned a privacy level. The privacy level of data can be directly inferred from the privacy levels of edge nodes where the data is stored. We assume that the default privacy level of the data stored in the edge node is the node’s privacy level. In addition, we employ a function to deal with the issue of privacy level calculation when the sensitive data is allowed to be stored in multiple edge nodes. After the initialization, each edge node and the data stored in it will be assigned the same privacy level. In our work, we ensure the no ship down in join operation principle during query processing. In other words, it is not allowed to ship either the input data or intermediate data to an edge node where its privacy level is lower than the privacy level of the data itself.

**Privacy Preference.** The data managed by edge nodes should be assigned a privacy preference parameter to control the data shipment scope in the edge computing environment. Users may have preferred edge nodes to deal with their data, hence, they have higher trustworthiness on such edge nodes. For instance, suppose that community \( A \) and \( B \) have cooperative relationship, while community \( A \) and \( C \) do not have such relationship. Thus users from \( A \) have lower privacy leakage concerns on edge nodes from \( B \) compared to edge nodes from \( C \). As a result, users from \( A \) may prefer their data processed at edge nodes that belong to \( B \) instead of \( C \) to deal with their privacy concerns. As the privacy preference is specified by data owners, setting such preference is a subjective approach to protect users’ private information.

**Privacy Guarantee.** Our privacy model can ensure that no privacy-sensitive information is disclosed in the distributed query processing phase in the edge computing environment. That is, if an adversary controls a public edge node, it will
not infer any privacy-sensitive information from monitoring the distributed query operations. Furthermore, even if the adversary controls a private edge node with privacy level \( p \), it cannot infer any sensitive information with privacy level higher than \( p \).

**Adversary.** In our work, we assume two types of adversaries, namely the public adversary and the private adversary. The public adversary has complete control of public edge nodes and can access any data stored in public edge nodes. Similarly, the private adversary can access the private edge nodes belonging to a specific privacy level. The public adversary has neither access to private edge nodes nor communication channels among the private edge nodes, while the private adversary can access both public edge nodes and the communication channels between its controlled edge nodes and other edge nodes. We also assume that a private adversary at a given privacy level cannot obtain information in the edge nodes at a higher privacy level. Here, the adversary can access any intermediate data shipped to its controlled edge nodes during the query plan execution phase, which is referred as the intermediate-data inference attack. For instance, as shown in Fig.1, if an adversary at site \( E_4 \) wants to query public information \( attr_1 \) and \( attr_2 \) by performing join query on \( s_2 \) and \( t_1 \), even though the query result does not disclose any sensitive information, the adversary can analyze the intermediate data, namely the joined table, to find sensitive attribute \( attr_2 \).

### III. QueryGuard Framework

#### A. QueryGuard Formalization

**Assumptions.** As we focus on query optimization and stream query processing [15] employing CQL as the query language which supports direct stream-to-relation operation, we will not differentiate stream data and relation data in our framework unless necessary. Thus, we use \( RS \) to denote any relation/stream in the rest of the paper. Also, in our work, we only consider select-project-join (SPJ) queries, i.e., the query involving selection, projection and join operations. We do not consider the data slice and partition problems and hence the relations or streams used in our work are not fragmented.

Let \( S_{RS} \) be a set of \( RS \) in the edge computing environment. Let \( RS_i \) denote an arbitrary relation or stream, where \( RS_i \subseteq S_{RS} \). We assume that we have a set of edge nodes, denoted as \( S_{edge} \) and \( e_j \in S_{edge} \) is an arbitrary edge server and for each edge node, e.g., \( e_j \), it manages a set of \( RS \), denoted as \( RS_{e_j} \), which indicates \( S_{RS,e_j} \subseteq S_{RS} \).

We next define the notations related to the privacy notion. We assume that \( L_p \) represents a privacy level list with size \( n \), and \( p_{x} \in L_p \) denotes privacy level \( x \). Each edge node is assigned a privacy level, denoted as \( f_{privacy}(e_j) : e_j \mapsto p_x \). The privacy level list has the two following properties: (i) \( L_p \) is an ordered list where sequence element with higher subscript in the list has higher privacy level, and (ii) for each edge node it can only belong to one privacy level. However, \( RS \) may be stored in several edge nodes, denoted as \( D_{RS} \), at the same time. In this case, the privacy level of \( RS \) is calculated as \( f_{privacy}(RS) := \min_{v_{e_j} \in D_{RS}} f_{privacy}(e_j) \).

The privacy preservation requirement restricts that \( RS \) can only be shipped among the edge nodes with the same privacy level or the edge nodes with higher privacy level during the query execution phase. We formalize two constraints regarding users’ privacy namely privacy level constraint and privacy preference constraint. The privacy level constraint is an objective constraint that limits the shipment direction of sensitive data according to privacy levels. The privacy preference constraint is a subjective constraint that controls the data shipment range based on data owner’s subjective preference. The constraints are defined as follows:

**Privacy Level Constraint.** Let \( C_{pl} \) be a privacy level constraint that defines limitations of the shipment direction in the query execution plan generation phase.

\[
C_{pl}(RS_i, RS_j) := e_i \xrightarrow{ship} e_j \text{ s.t. } f_{privacy}(RS_i) \leq f_{privacy}(RS_j)
\]

where \( RS_i \in S_{RS,e_i}, RS_j \in S_{RS,e_j}. \)

**Privacy Preference Constraint.** Let \( C_{pp} \) be a privacy preference constraint that defines shipment scope requirement in the query plan generation phase.

\[
C_{pp}(RS_i, RS_j) := e_i \xrightarrow{ship} e_j \text{ s.t. } \arg \min_{p_{e_i} \in S_{RS}} f_{privacy}(e_j) \leq \lambda
\]

where \( RS_i \in S_{RS,e_i}, RS_j \in S_{RS,e_j} \), and \( \lambda \) is a threshold assigned with \( RS_i, p(e_i, e_j) \) is the preference shipment scope.

Our QueryGuard framework is formally described as follows:

**Definition 1. QueryGuard Specification.** Let \( S_{L} \) be the set of all query execution plans for a query \( Q \) and \( L_{opt} \) represents the optimized query execution plan based on the latency-aware cost measure method \( f_{cost}(\cdot) \).

\[
L_{opt} := \arg \min_{S_{L} \in Q} f_{cost}(L) \text{ s.t. } C_{pl} \text{ and } C_{pp}
\]

Note that \( C_{pl} \) and \( C_{pp} \) is related to the privacy-preserving approaches to tackle the privacy constraints, while the latency-aware approach is based on designing new cost model \( f_{cost}(\cdot) \). The techniques to obtain these two components in the model are the key contributions of this paper.

#### B. Preliminaries

The architecture of the edge query processor consists of a parser, a re-writer, a query optimizer, a query executor and a catalog [10]. We describe them next.

**Catalog.** The catalog stores all information that is needed to parse, rewrite, and optimize a query [10]. For instance, the catalog may include the schema regarding relations, indices, and views. Other statistics and the current system state could also be stored in the catalog. In the distributed setting, the catalog also includes additional information such as the location, replicas of relations, and the sites.

**Cost Model.** In order to find an optimal query plan, the query optimizer employs a cost model that accurately estimates the system resources used for each operator in the query. For instance, selection, projection and join operations need to be
estimated by the cost model in a centralized database system. When it comes to distributed edge query processing, the cost estimation on join operators becomes more complex due to varying network conditions and the connectivity between the edge nodes. Even though some of plan candidates result in the same results, the cost of these plans may vary by several orders of magnitude.

**Query Optimization.** The query optimizer employs an enumeration algorithm to enumerate the entire search space. Specifically, the main goal of the query optimization process is to take an input query and produce a specific query execution plan that guides the query executor how the query should be executed. Here, the problem of finding the best plan is NP-complete. Typically, there are three categories of enumeration algorithm to deal with query optimization: exhaustive search, heuristic-based search and randomized search [16]. The traditional dynamic programming enumeration algorithm is a popular exhaustive search algorithm, which has been widely used in a large number of commercial database management systems. In our work, we combine the exhaustive search approach and the heuristic-based approach to deal with the query optimization. To be specific, we use traditional dynamic programming enumeration algorithm as the skeleton with the help of heuristic rules to prune unsatisfied branches.

**C. QueryGuard Framework**

We propose our QueryGuard framework in Algorithm 1 that is constructed using the skeleton of traditional dynamic programming enumeration algorithm where the optimal plan is generated by joining optimal sub-plans in a bottom-up manner. In order to produce the best possible plans, we employ the iterative dynamic programming approach that promises the best plans compared to other algorithms even if dynamic programming turns out to be not viable [11]. We incorporate heuristic-based approaches in our algorithm which can guide the search into several specified sub-plans in the entire search space. Essentially, our proposed work can be considered as a combination of dynamic programming exhaustive search and heuristic-based search to achieve the optimal query plan.

When generating query plans by joining different sub-plans in a bottom-up manner, we adopt the privacy-preserving constraints as part of the heuristics to incorporate the privacy-aware data processing constraints. The heuristics include both privacy level constraint and privacy preference constraint proposed in Section III-A. Any plan candidates that do not pass the check under heuristic rules are pruned from the search space immediately (see line 11). Finally, the network latency measures are considered to prune the rest of plan candidates to generate the final optimal query plan (see lines 12 and 19). The specific implementation of the privacy join mechanism and the latency aware mechanism is described in the next section.

The critical phases in the QueryGuard framework is illustrated in Fig.3. Suppose that there are four RS assigned with four privacy levels, which are deployed in eight edge nodes. In traditional approaches, there are many possible join shipment candidates and the query optimization will try to choose the join shipment with the lowest resource cost. In our proposed QueryGuard framework, we limit the data shipment scope and control the shipment direction due to privacy constraints. For instance, $E_1$, $E_5$, $E_6$ are in the same data preference scope according to the specified threshold. As a result, the possible join shipment candidates between $E_1$ and $E_4$ is pruned. In addition, the shipment from $E_5$ to $E_6$ is also pruned due to the violation of privacy level constraint. After this step, we adopt our proposed dynamic latency-aware cost model to generate the final optimal plan.

### IV. QueryGuard Query Processing Techniques

#### A. Privacy-preserving Query Optimization in QueryGuard

To tackle the privacy-aware query processing challenges outlined in Section II, we propose a novel privacy-join function in the QueryGuard framework (Algorithm 2). The privacy-preserving join function avoids potential privacy information leakage by generating privacy-preserving query plans. The algorithm takes two sets of query plans that need to be joined and returns possible privacy-preserving joined query plans. To begin, the algorithm initializes a list that will be used to store the join plan candidates (line 1). For each possible edge site, it tries to create a new join-node based on a plan-pair where two plans are selected from two sets of query plans in an iterative manner, respectively (lines 3-13). Before creating a new join-node, the plan candidates will be checked to see if it violates the privacy constraints. If it violates, such potential query plans will be pruned from the search space tree (lines 6-7 and lines 10-11); otherwise, the new join-node will be created with assigned edge site (lines 12-13).

---

**Algorithm 1: Pseudocode for QueryGuard framework**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input: A set of relations or streams $R = {R_i}$ with size $n$ generated from a query $Q$</td>
</tr>
<tr>
<td>2</td>
<td>Output: The optimized query plan</td>
</tr>
<tr>
<td>3</td>
<td>for $i = 1$ to $n$ do</td>
</tr>
</tbody>
</table>
| 4    | \[
| 5    | b := balanced-parameter(|$\text{toDo}$|, $k$) |
| 6    | forall $S \subseteq R$ and $|S| = i$ do |
| 7    | plans($S$) := $\emptyset$ |
| 8    | forall $O \subseteq S$ and $O \neq \emptyset$ do |
| 9    | \[
| 10   | plans($S$) = plans($S$) $\cup$ PRIVACY-JOIN(plans($O$), plans($S$ $\setminus$ $O$)) |
| 11   | CLAIM-AWARE-PRUNE(plans($S$)) |
| 12   | endfor |
| 13   | find $P, V$ with $P \in$ plans($V$), $V \subseteq$ $\text{toDo}$, $|V| = k$ such that |
| 14   | eval($P$) = min{eval($P'$) $|$ $P' \in$ plans($W$), $W \subseteq$ $\text{toDo}$, $|W| = k$} |
| 15   | generate new symbol: $T$, plans($T$) = $\{P\}$ |
| 16   | $\text{toDo} = \text{toDo} - V \cup \{T\}$ |
| 17   | forall $O \subseteq V$ do |
| 18   | delete(plans($O$)) |
| 19   | endfor |
| 20   | endfor |
| 21   | endfor |
| 22   | final-plan(plans($R$)) |
| 23   | CLAIM-AWARE-PRUNE(plans($R$)) |
| 24   | return plans($R$) |
Fig. 3. An illustration of the critical phases in QueryGuard framework. Note that the symbol $E_i, S_j$ represents an edge node $E_i$ where relation/stream data $S_j$ stored. For simplicity, data $S_j$ also implies privacy level $PL_j$, where $\forall x < y, PL_x < PL_y$. Edge node $E_1, E_5, E_6$ are supposed in one privacy preference scope, while the rest of edge nodes are in another privacy preference scope.

**Algorithm 2: Privacy-preserving join algorithm**

```
function PRIVACY-JOIN(lplans, rplans)
   join-plans := {} ;
   foreach possible edge e do
      for plan l in lplans do
         if PREERENCE-CONSTRAINT(l, e) then continue;
      lpp := LEVEL-CONSTRAINT(l, e) ;
      if lpp.flag then continue;
      for plan r in rplans do
         if PREERENCE-CONSTRAINT(r, e) then continue;
      rpp := LEVEL-CONSTRAINT(r, e) ;
      if rpp.flag then continue;
      join := new node(lpp.root, rpp.root, e) ;
      join-plans.add(join) ;
   return join-plans

function LEVEL-CONSTRAINT(p, e)
   flag := false
   if p.root.site ≠ e then
      if p.root.$P > e. P$ then return (true, null) ;
   else
      rcv_node := new node(p.root, e)
      set $P$ of rcv_node same to $P$ of e.
      p.root := rcv_node
   return (flag, p)

function PREERENCE-CONSTRAINT(p, e)
   foreach leaf node in p do
      $\lambda$ := transmission threshold of leaf node
      if $\lambda$ is Set type then
         if $e \notin \lambda$ then return true;
   return false
```

Note that only the root of a plan is assigned a possible edge site at any time due to our framework’s bottom-up construction when building the query plans. The newly created join-node is the root of two previously generated query plans’ roots that already have edge site assignment.

Here, we present two privacy-constraint functions used in the privacy-join function. One is the level-constraint function that automatically controls the data shipment directions according to privacy level settings. The other is preference-constraint function that is a subjective privacy setting approach to limit the scope of data dissemination.

1) Privacy Level Constraint: According to our proposed privacy model (see Section II-B), $RS$ will be tagged with a specific privacy level, denoted as $P$, where $P \in \mathbb{N}^0$. Here larger numbers represent higher privacy level, hence, we use $P := 0$ to denote the public data. We present the details of level-constraint function in Algorithm 2. The function first checks the root site of the plan candidate by comparing the current iterative site to the root site of the plan candidate. If the two sites are the same edge site, it will skip without any actions due to no cross-site shipment, hence, there is no privacy leakage risk; otherwise, it will generate a receive-node to denote the shipment procedure (line 17). It then compares the privacy level between the current iterative site and the root site of the plan candidate. If the privacy level of the root of the plan candidate is higher than the site’s privacy level, the function returns a true signal to prune the join operation in the search space tree (line 18); otherwise, the function creates a new receive-node with the same privacy level of the current site (line 20). Finally, the new receive-node will replace the root of the plan (line 22).

Note that considering our privacy guarantee, the root of the plan candidate will be assigned to corresponding privacy level if the root node is transferred to a higher privacy level edge site, due to avoiding the intermediate-data-inference exposure (line 21).

2) Privacy Preference Constraint: As proposed in Section III-A, the privacy preference constraint limits the data shipment scope. Here, we use a list to represent the threshold $\lambda$. To be specific, each edge node will be assigned a list to indicate its preferred shipment scope in the edge computing environment. In the query execution phase, the data in an edge node $E_i$ is not allowed to ship to other edge nodes that are not in the threshold list of $E_i$.

We use an example in Fig.4 to illustrate the privacy preference constraint on the shipment scope. Here, if edge nodes $E_2, E_3$ are the desired edge nodes in the shipment scope by a user who stores data in $E_1$, the threshold will be denoted as $\lambda_{E_1} = \{E_1, E_2, E_3\}$. The preference-constraint function is presented in Algorithm 2. $RS$ is assigned with a threshold $\lambda$ to
control the scope for sensitive data dissemination. The query plan including the cross-site join operations that try to ship the data out of the scope will be removed from the query plan candidate list. Specifically, the function first traverses the entire leaf nodes, i.e., RS in the query plan, to find the threshold (lines 25-26). Then, if the threshold has a valid format, the function checks the presence of the target site $e$ in the threshold list (lines 27-28); After checking each leaf node, if there is still no constraint violation, the function returns a false signal (line 29).

B. Cost Model and Latency-aware Optimization

Even though the centralized cost measurement model is reasonably straightforward to apply, the cost model to estimate the cost of performing cross-site joins is still a challenge due to unpredictable and time-varying network characteristics [10], [17], [18]. In the centralized setting, the cost of performing $SPJ$ operations in the absence of indices is the cost to scan relations and writing out the results, which is related to I/O operations. Similarly, the $seq-window$ operator (see CQL in STREAM [15]) that uses a temporary queue and synopsis structures is also an I/O related operation. Here we use $C_{cent}$ to represent all operation costs in the centralized setting.

Dynamic Cost Model. In the distributed setting, the cost of performing cross-site join is an important component of the overall query execution cost [10]. The cost model in our framework directly adopts $C_{cent}$ based on previous work and focuses on additional cost incurred in the distributed setting. The cost model is described as follows:

$$f_{cost}(L) = C_{cent} + \sum_{(e_i \rightarrow e_j) \in L} (n_{bytes, e_i \rightarrow e_j} \cdot t_{estimate}^{e_i \rightarrow e_j})$$

where $n_{bytes}$ and $t_{estimate}$ are the number of bytes shipped from $e_i$ to $e_j$, and the estimated time for shipping one byte, respectively.

We next present the details of how to measure the estimated time of transferring one byte of data. As a straightforward approach, this could be achieved by sending $n_{send}$ bytes and recording the average time $t_{avg}$ for every $t_{interval}$. However, it is unwise to monitor the real-time network traffic for a database system due to excessive consumption of bandwidth resources. Therefore, we use the network traffic performance observed in the last $t_{interval}$ to estimate the current performance. Specifically, the estimated $t_{estimate}$ is defined as follows:

$$t_{estimate}^{e_i \rightarrow e_j} = \alpha \cdot t_{avg} / n_{send}$$

where $\alpha$ is the coefficient that indicates the potential risk. Here we define $\alpha$ as $\arctan(d_{geo}(e_i, e_j)) \cdot 2/\pi$, where $d_{geo}(e_i, e_j)$ is the geographical distance of edge server $e_i$ and $e_j$. The geographical distance is a constant and does not change over time. A longer distance indicates higher potential risk, hence, we use distance as the coefficient in the above equation and we use the $arctan$ function as the normalization method for distance instead of the min-max normalization method due to consideration of non-linear property in the $arctan$ function, which resembles more real-world scenarios.

Algorithm 3: Latency-aware function

```plaintext
function LATENCY-AWARE-PRUNE(plan(S))
result := \{\}\;
foreach site e do t[e] := null;
foreach plan p in plans(S) do
  c := extract the catalog information;
  if \(f_c(p) < f_c(t[e])\) such t[e]#null then \(t[e] := p;\)
foreach site e do result.add(t[e]) such t[e]#null;
return result
```

Latency-aware Optimization. The latency-aware optimization is based on the dynamic cost model, which tries to choose the query plan that has the minimum cost value. The generated query plan has lower latency due to two reasons: (i) it benefits from the edge computing settings, i.e., the edge server is close to the query executor, and (ii) it adopts our proposed cost model that dynamically adjusts the evaluation weight according to the performance of the network traffic. The function is presented in Algorithm 3. The function first initializes an empty set that will be used to store the pruned plans and a temporary array to store plan candidates for each possible site (lines 2-3). Based on the dynamic cost model, the function traverses the entire plan candidates to find the plan that has the minimum cost and stores it in the temporary array (lines 4-6). Finally, it clears up the final results and returns them (lines 7-8).

V. EXPERIMENTAL EVALUATION

A. General Setup

We performed the experiments on a Unix-like operation system. The main hardware includes four 2.5 GHz Intel Core i7 processors, 16GB memory, and SSD hard disk. All algorithms of our framework were implemented using Java.

We simulate a set of edge nodes with artificially injected network latency between them. The query optimization phase only focuses on the generation of a logic query execution plan, hence, the simulation approach with proper catalog settings is used in our study, similar to the experimental setup in earlier work [9], [11]. Specifically, we simulate 15 edge nodes with specific geography information, as shown in Table III. The latency (ms) of the network traffic is estimated based on the distance (miles) using a linear model proposed in [14]. The model is specified as $y = 0.022x + 4.862$. All the experiments were executed using randomly generated queries over randomly generated relations/streams that are distributed

<table>
<thead>
<tr>
<th>TABLE III</th>
<th>EDGE NODE SIMULATION.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge Node Address</td>
<td>Privacy Level</td>
</tr>
<tr>
<td>10.0.1.1</td>
<td>1-8</td>
</tr>
<tr>
<td>10.0.1.9</td>
<td>0</td>
</tr>
<tr>
<td>10.0.1.10</td>
<td>1</td>
</tr>
<tr>
<td>10.0.1.11</td>
<td>2</td>
</tr>
<tr>
<td>10.0.1.12</td>
<td>3</td>
</tr>
<tr>
<td>10.0.1.13</td>
<td>0</td>
</tr>
<tr>
<td>10.0.1.14</td>
<td>2</td>
</tr>
<tr>
<td>10.0.1.15</td>
<td>3</td>
</tr>
</tbody>
</table>
on the 15 edge nodes. This experiment setup is similar to model used in [9], [11].

B. Results

We first compare the performance of the proposed QueryGuard approach with existing IDP1 approaches in terms of execution time and memory usage. The privacy-preserving feature is illustrated using a case study on optimization of a randomly generated query. Finally, we present the performance analysis of latency-aware feature of QueryGuard.

1) Comparison to IDP1: For our experiments, we first generate 10 random relations with random cardinality. The cardinality distribution of each random relations/streams is shown in Table IV. The test queries are also generated randomly with the relation size ranging from 3 to 10 for each query topology. In our experiment, we test 5 types of query namely chain topology, cycle topology, star topology, clique topology, and mixed type. Then, we execute IDP1 [11] and our proposed QueryGuard algorithm for each query 5 times for each topology and collect the experiment results to calculate the mean value and standard deviation value.

The comparison results for execution time and memory usage are shown in Fig.5 and Fig.6, respectively. Compared to the IDP1 algorithm, our proposed technique has non-negligible performance advantage both in execution time and memory usage aspects. Recalling the details in Algorithm 1, it is reasonable according to the theoretical analysis. Our privacy setting operations in the algorithm is a heuristic that leads to early pruning. In other words, if one node of the branch in the search space tree violates the privacy constraints, the search on such branch will be stopped immediately. Thus, it will save both time and memory when executing our proposed approach. Note that the purpose of the comparison is illustrating the performance efficiency of our privacy settings. Even though \( k \) is an important parameter in iterative dynamic programming query optimization algorithm, where \( 1 \leq k \leq n \), \( n \) is the total size of relations, here we just test three case of \( k \), i.e., \( k = 3, 5, 7 \).

2) Case study of privacy-preserving processing: We run a series of experiments to evaluate the effect of the privacy-preserving query processing feature in QueryGuard. First, we randomly generated relations and stored them in the 15 edge nodes with specified transmission threshold. Then, we test several random queries by generating optimal query plans and validate if the privacy requirements are achieved.

### Table IV

<table>
<thead>
<tr>
<th>Relation Type</th>
<th>Cardinality of Relation</th>
<th>Simulation Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>10-100</td>
<td>5%</td>
</tr>
<tr>
<td>II</td>
<td>100-1000</td>
<td>15%</td>
</tr>
<tr>
<td>III</td>
<td>1,000-10,000</td>
<td>30%</td>
</tr>
<tr>
<td>IV</td>
<td>10,000-100,000</td>
<td>30%</td>
</tr>
<tr>
<td>V</td>
<td>100,000-100,000</td>
<td>15%</td>
</tr>
<tr>
<td>VI</td>
<td>1,000,000-10,000,000</td>
<td>5%</td>
</tr>
</tbody>
</table>

\( ^{\dagger} \) The cardinality of a stream indicates the size of synopsis in DSMS.

### Table V

<table>
<thead>
<tr>
<th>Relation/Stream</th>
<th>Edge Node</th>
<th>Transmission Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>10.0.1{3,4,5,7,10,14,15}</td>
<td>10.0.1{1-12}</td>
</tr>
<tr>
<td>B2</td>
<td>10.0.1{6,8,11,12}</td>
<td>10.0.1{1-12}</td>
</tr>
<tr>
<td>C3</td>
<td>10.0.1{2,6,11}</td>
<td>10.0.1{1-12}</td>
</tr>
<tr>
<td>D4</td>
<td>10.0.1{2,4,5,6,11,12,13}</td>
<td>10.0.1{1-12}</td>
</tr>
<tr>
<td>E5</td>
<td>10.0.1{4,12,13}</td>
<td>10.0.1{1-12}</td>
</tr>
</tbody>
</table>

Here, we use an instance to illustrate the feature. Specifically, we present the 5 random relations, i.e., A1-E5, which are depicted in Table V. According to privacy function (see Section III-A), we can infer that the privacy level of B2 is 2, while others’ privacy levels are 0. The specified transmission threshold, i.e., 10.0.1\{1-12\}, indicates that the relations and immediate data during the query execution phase will not be transferred outside of Pennsylvania State (referring to location information in Table III). We can find that B2 and C3 are located at Pennsylvania entirely. Here we suppose the query site is at 10.0.1.6. The generated privacy-preserving optimal query plan is shown in Fig.7 based on a cycle join query, i.e., A1<>B2<>C3<>D4<>E5<>A1. According to the optimal query execution plan depicted in Fig.7, both privacy level constraint and privacy preference constraint are not violated. Our proposed technique satisfies the privacy goals.

3) Effect of latency awareness: We also run a series of experiments to evaluate whether the latency-aware cost model influences the performance of our proposed framework. We perform another group of experiments with constant network traffic instead of the proposed latency-aware approach in order to compare with the normal QueryGuard algorithm. As shown in Fig.8, we present the comparison result of the performance of our QueryGuard algorithm to the QueryGuard algorithm without latency-aware setting in aspects of execution time and memory usage. The latency-aware setting has a negligible effect on the memory usage of the algorithm, while the execution time cost has slight growth when the relation number increase. Note that here we only present the results of chain and clique queries due to page limitation. The remaining three query topologies have the similar trend, hence, we do not present them.

VI. RELATED WORK

Recent advances in edge/fog computing brings both advantages and challenges [1]–[3]. The low latency feature of edge computing can enable the application with lower responsive time requirement possible due to its close-to-data computing model. Thus, edge computing is becoming a critical infrastructure in emerging application scenarios such as smart city, Internet of Vehicles, and Internet of Things [3]. However, compared to the management of cloud data centers, edge computing’s loose, non-strict management may lead to higher privacy risk [1], [2]. Privacy issues is a significant concern when adopting traditional applications to be deployed in an edge computing environment.

Query optimization in databases is a classical topic. The query optimizer plays an important role in modern
DBMS/DSMS architectures. Several enumeration approaches have been proposed to perform query optimization, which includes randomized search, exhaustive search, heuristic-based search [16]. The dynamic programming enumeration algorithm pioneered in IBM’s System R project is widely adopted in commercial DBMSs [8], [19]. Dynamic programming based query optimizers can be extended to the distributed environments such as the edge computing model. To tackle the space complexity problem of dynamic programming approach, the iterative dynamic programming (IDP) was proposed in [11]. Distributed query processing is a central component in distributed database management systems [20]. Several works were proposed in [5], [9] to address issues in the aspects of special join, parallelism, communication, and cache.

The network performance has a significant influence in the distributed query processing and the query processing efficiency, hence, several network-aware approaches were proposed in [17], [18], [21], [22] to tackle the query processing issues based on network performance. Ahmad and Cetintemel proposed network-aware query processing in widely distributed Internet environment by leveraging knowledge of network characteristics [17]. Li et al. proposed a federated information system with query cost calibrator that calibrates the cost function based on system load and network latency [21]. Srivastava et al. focused on the problem on how to place operators along the nodes of the hierarchy in stream scenarios so that the overall cost of computation and data transmission is minimized [22]. Pietzuch et al. proposed a SBON layer between a stream-processing system and the physical network that manages operator placement for stream-processing systems [18]. Even though network-aware approaches have been used in query processing in [17], [18], [21], they depend on the existing knowledge of network characteristics such as topology and link bandwidth which would be sometimes difficult to obtain in decentralized edge computing scenarios.

We note that the traditional query optimization research, especially distributed query processing techniques, have not considered the privacy issues associated with the data transmission during query processing [6], [10], [11], [16]. A few database systems [12], [13] employ cryptography to protect the underlying data, however, such cryptographic techniques are not very efficient in edge computing scenarios due to their heavy computation cost. PAQO proposed in [4] deals with the privacy concerns in the query generation phase. Its privacy protection is based on inquirers’ subjective setting rather than based on data owner’s perspective. SMCQL proposed in [7] translates SQL statements into secure multiparty computation primitives to tackle privacy issues in the private data network, but it does not address the intermediate-data-inference disclosure. Both PAQO and SMCQL need an honest and trusted third party to support the query.

Unlike the above-mentioned techniques, the privacy-aware query processing proposed in our work does not require...
a trusted third party and as shown in the evaluation, the approach is highly scalable under a wide range of experimental conditions. To the best of our knowledge, the work presented in this paper is the first significant effort in developing a highly scalable and efficient privacy-aware and latency optimized query processing in edge computing framework without requiring the use of a trusted third party entity.

VII. Conclusion

In this paper, we propose QueryGuard, a privacy-preserving latency-aware query optimization framework, to tackle privacy-aware and latency optimized query processing in edge computing environments. While edge computing provides a unique ability to store and process large amounts of data on servers and computing units located close to the data sources such as sensors and mobile devices, conventional query processing techniques applied in an edge computing environment can lead to higher risk of disclosure of private information from the edge nodes. Our proposed work deals with both the privacy concerns as well as query latency optimization for distributed join query processing. The proposed query optimization mechanism generates optimal query execution plans that ensure users privacy preferences on their sensitive data stored in edge nodes during the query execution. We evaluate the proposed techniques in terms of execution time and memory usage and our results show that the proposed methods perform better than conventional techniques while achieving the intended privacy goals.

REFERENCES


