This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

Enforcement of Autonomous Authorizations in Collaborative Distributed Query Evaluation

Qiang Zeng, *Member, IEEE,* Mingyi Zhao, Peng Liu, *Member, IEEE,* Poonam Yadav, *Member, IEEE,* Seraphin Calo, *Senior Member, IEEE,* and Jorge Lobo, *Member, IEEE*

Abstract—In a federated database system, each independent party exports some of its data for information sharing. The information sharing in such a system is very inflexible, as all peer parties access the same set of data exported by a party, while the party may want to authorize different peer parties to access different portions of its information. We propose a novel query evaluation scheme that supports differentiated access control with decentralized query processing. A new efficient join method, named *split-join*, along with other safe join methods is adopted in the query planning algorithm. The generated query execution reduces the communication cost by pushing partial query computation to data sources in a safe way. The proofs of the correctness and safety of the algorithm are presented. The evaluation demonstrates that the scheme significantly saves the communication cost in a variety of circumstances and settings while enforcing autonomous and differentiated information sharing effectively.

Index Terms—Query processing, Security and Authorization, Distributed databases

1 Introduction

Emerging scenarios for collaborative missions in various areas require different organizations to exchange information in a large distributed system. Such scenarios range from multinational military tasks; to international scientific cooperation; to *ad hoc* coalition formation for humanitarian emergency operations. A recent example is that a coalition network infrastructure in Afghanistan, which linked US troops and their counterparts from several allied nations, fundamentally changed the way multinational efforts had been conducted [17].

As information is flowed outside party boundaries, the need for data protection is as strong as data exchange. In this paper, we consider information sharing among multiple parties through a distributed database system. We focus on relational databases, which should not be considered as a limitation, as the relational model is the *de facto* standard used by virtually all mainstream database systems. In particular, through a research project jointly sponsored by the US Army and the UK Ministry of Defense, we have gathered a set of principal data protection needs during information sharing in a multinational military mission in Afghanistan. Based on these needs, we propose the following three access control requirements, which we believe are common for information sharing in non-military collaborations as well. (1) *Per-party authorized view* (**R1**): in a collaboration involving multiple parties, various bilateral relations between parties exist. Hence, for example, the fact that party A discloses some specific data to party B does not imply that A feels necessary or safe to share it with party C. A party should be able to authorize access to different portions of its data to different peer parties, as a result each party has its own view over the data set stored in the coalition network. (2) *Authorization autonomy* (**R2**): each party should have full and autonomous control over the authorization definition. It is usually impractical to assume a centralized entity (e.g., a supervisor) defining authorizations. It is also inflexible to require multiple parties reach consensus before an authorization modification can be made. (3) *Fine-granularity access control over tuples* (**R3**): access control over the data stored in a table has two dimensions: columns (attributes) and rows (tuples), corresponding to a vertical perspective and a horizontal one, respectively. While regulating the accessibility of attributes (*vertical access control*) is important in some scenarios, access control on tuples (*horizontal access control*) is a general need in most cases. There are many other requirements depending on practical circumstances, but **R1–R3** commonly exist in most multi-party information sharing scenarios.

Although many authorization models and enforcement mechanisms have been proposed in the literature, none supports such information sharing flexibility and capability as specified in **R1–R3**. Classic distributed database systems assume a single party/organization [4], [11], [13], [21]. They do not consider inter-party access control at all. In a federated database system each party exports information accessible to the whole system [3], [9], [12], [18], [20]. Thus, all the parties share the same data access, which does not satisfy **R1**.

- *Qiang Zeng is with the Department of Computer Science and Engineering, Penn State University. E-mail: qzeng@cse.psu.edu.*
- *Mingyi Zhao and Peng Liu are with the College of Information Sciences and Technology, Penn State University. E-mail: {muz127, pliu}@ist.psu.edu.*
- *Poonam Yadav and Seraphin Calo are with IBM T. J. Watson Research Center. E-mail: {yadavp, scalo}@us.ibm.com.*
- *Jorge Lobo is with ICREA-Universitat Pompeu Fabra. E-mail: jorge.lobo@upf.edu.*

A preliminary version of this work appeared in the 6th Annual Conference of International Technology Alliance, Southampton, UK, September 2012 [23].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

The first piece of work that targets differentiated privacy control between parties in distributed query processing is [5] (see also [6]); it enables a distinct authorized view for each party, which starts to realize **R1**. However, they deal with the accessibility of columns rather than tuples. Due to the orthogonal perspectives, their scheme cannot be applied to satisfying **R3**. Besides, in their model authorizations that regulate the joins among multiple parties are supposed to be defined by these parties collaboratively, which does not meet **R2**. One may propose to restrict the authorizations to be defined within each party to meet **R2**; however, it would significantly degrade the access control capability provided by the original proposal. Further discussion can be found in Section 9.

In order to address **R1–R3** we propose to adopt a simple, yet expressive, authorization specification, named *pairwise authorization*. A pairwise authorization involves two parties: the data provider (i.e., the data owner) and the consumer; it specifies a set of the data provider's tuples accessible to the consumer party. It is the data provider that defines the authorization, so each party has autonomous control of defining and modifying authorizations in terms of its own data.

Due to pairwise authorizations, each party has a distinct view of the data shared by other parties, which differs from the identical view among parties in a conventional federated database system. The view disparity among parties imposes new challenges on query evaluation. A straightforward solution is to pull all the data involved in a query to the querier, which then performs the query computation. Although the centralized query evaluation is safe, it usually leads to high communication costs.

We present a query planning algorithm that supports pairwise authorizations. Given a query, the algorithm generates a distributed query execution plan compliant with the specified pairwise authorizations. Following standard principles for query processing in distributed databases it explores possibilities to evaluate partial queries along the path the data travels from the data repositories to the querier, with the enhancement that the access control specified by the pairwise authorizations is obeyed.

One of the challenges due to the differentiated per-party access control is that the *semi-join* method [2], a widely used join method in conventional distributed database systems, is not generally applicable any more. To simultaneously achieve two goals, namely (1) push query computation as close as possible to data sources as the semi-join does, and (2) keep compliant with pairwise authorizations, a new join method, named *split-join*, is designed. It preserves distributed query computation between data sources while correctly dealing with the view disparity. The query planning algorithm adopts split-join and other join methods for a safe and efficient query evaluation. Multiple servers of the same party are called *buddy servers*, which share the same authorized view. The algorithm exploits the presence of buddy servers by delegating partial queries to the buddy servers near data sources or intermediate results to further reduce communication cost.

A notable property of the algorithm is that it preserves the confidentiality of authorizations. That is, during query evaluation a server only refers to the authorizations defined by its own party; therefore, the privacy how each party defines authorizations is protected.

We proved the correctness, safety, and authorization confidentiality property of the proposed algorithm. A general simulation platform is developed to experimentally evaluate the algorithm. The experiments show that the query planning algorithm reduces the communication cost by half or more on average under a variety of settings and differentiated access control for multi-party information sharing is achieved by enforcing autonomous authorizations.

The remainder of the paper is organized as follows. Section 2 describes the multi-party distributed database system over a coalition network. Section 3 introduces pairwise authorizations and safe query processing. Section 4 presents various join methods including the novel split-join. Section 5 illustrates the query plan generation algorithm, and Section 6 presents the proofs of its correctness, safety and authorization confidentiality. Section 7 presents the evaluation results. Section 8 discusses replication and fragmentation. Section 9 and 10 discuss related work and conclusions, respectively.

2 Queries in a Coalition Network

In this section, we describe the network constructed by multiple database servers, introduce the query procedure, and define query trees.

2.1 Coalition Network

We consider a coalition network which links a set of database servers $N = \{S_1, \ldots, S_n\}$ belonging to different parties. An overlay network $G(N, E)$ is constructed among the servers in N, and E is the set of links between the servers. $l_{i,j}$ represents the cost of transmitting a unit data over the link between S_i and S_j. We assume $l_{i,j} = l_{j,i}$ and the link cost is non-negative. The cost of transmitting a unit of data along the path between two servers S_i and S_j is denoted as $p_{i,j}$, which is the sum of the costs of individual links along the path. The shortest-path routing is adopted; that is, the shortest path is used to transmit data between any two servers.

We assume that, to protect data confidentiality, data transmission is encrypted. Each pair of parties share a unique key. For example, when S_i sends data to S_j via S_k, S_k cannot decipher the passing traffic, for it is encrypted using the key shared between S_i and S_j.

Each server and all the base relations stored in the server belong to one and only one party. Each relation is stored in a single server (fragmentation and replication are discussed in Section 8). The party that server S

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

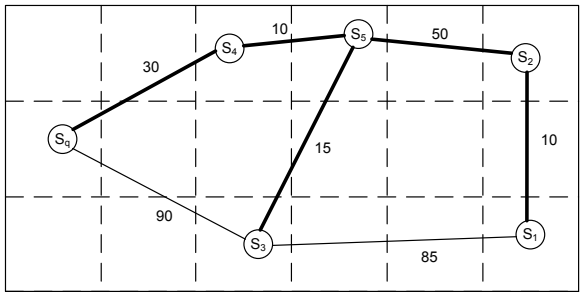


Fig. 1. A coalition network for a disaster recovery mission. It takes place in a large area divided into smaller districts, as marked by the dashed lines. The numbers indicate the link costs. The bold lines mark the shortest-path tree rooted at S_q; S_q communicates with other servers using the shortest paths.

TABLE 1
Relations

Server	Relation (*Italic* attributes are the keys)
S_1	Safehouse(*safehouseID*, type, district)
S_2	Service(*employeeID*, *service*, district)
S_3	Communication(*stationID*, function, district)

belongs to is denoted as $P(S)$. If two servers belong to the same party, they are *buddies* of each other; otherwise, they are *foreigners*. A *buddy*$^+$ of server S refers to a buddy server of S or S itself. Therefore, buddies and foreigners are used to describe the relationship of any pair of servers in a coalition network.

Example 2.1: Figure 1 represents a coalition network constructed for a disaster recovery mission involving multiple parties. The coalition network involves five parties, denoted as V_i, and six servers. $P(S_1) = V_1$, $P(S_2) = V_2$, $P(S_3) = V_3$, $P(S_4) = V_4$, and $P(S_5) = P(S_q) = V_5$ (that is, S_5 and S_q are buddies). Specifically, party V_1 is responsible for constructing safe houses, such as camps and shelters; a table named Safehouse describing the types and locations of the safe houses is stored at S_1. Party V_2, a contractor company providing various services including disinfection, air conditioning and transportation; information about employee duties and working districts is recorded at the table Service, which is stored at S_2. Party V_3 deploys and manages the communication infrastructure, for example, base stations for local area or satellite communication; a table named Communication is stored at server S_3. The attributes and keys of the relations are listed in Table 1.

2.2 Query Procedure

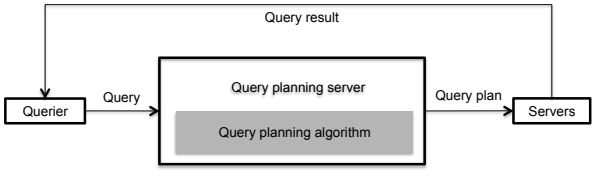


Fig. 2. Query procedure.

Query 1:
SELECT district, safehouseID, type, employeeID, stationID
FROM Safehouse, Service, Communication
WHERE Service.service=''Disinfection'' **and**
Communication.function = ''Satellite'' **and**
Safehouse.district = Service.district **and**
Service.district = Communication.district

Fig. 3. A query searching for surgery locations.

Figure 2 illustrates the query procedure. When a query is received, the query planning server generates a query plan and sends query plan slices to involved servers. Note that we do not hide query intension from the planning server [8], which is beyond the scope of this paper, and the planning server is trusted not to conspire with any party. Note that such a query planning server is commonly used in a distributed database system [5], [9], [18], [20], as the complexity of a distributed query planning is too high [6]. The query is then executed among the servers in a distributed way, and the final result is sent to the querier. Query planning plays a key role in query evaluation, and is the focus of this paper.

2.3 Query Trees

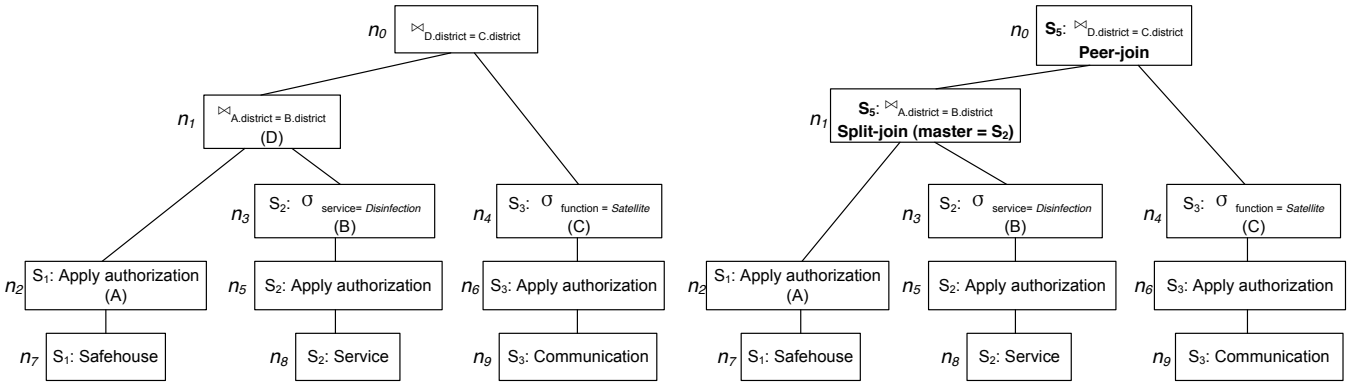
We consider simple yet typical queries of the form *"SELECT-FROM-WHERE"* in which the *WHERE* clause is limited to disjunctions and conjunctions of equalities and inequalities $\{<, >, \neq\}$. Given a query, the query optimizer compares possible query plans to determine the most efficient query strategy. A query plan can be represented as a query tree, where intermediate results flow from the bottom of the tree to the top during query execution. The leaves of a tree represents base relations, while each non-leaf node represents a relation obtained by applying one of the relational operators {select σ, project π, join $\bowtie$ } to its child nodes.

A *preliminary* query tree is a query tree that lacks information about how inter-server query operations are conducted, while a *complete* query tree corresponds to a complete query plan, which specifies all the query operations to execute a query. The function of a preliminary tree is mainly to indicate a query order. How to enumerate all possible preliminary query trees for a given query has been extensively researched [10], [16], while our work is, given a preliminary tree, to generate an efficient complete query tree compliant with authorizations.

Example 2.2: Query 1 shown in Figure 3 is issued by server S_q belonging to party V_5, which is responsible for medical assistance. The query searches for locations suitable for performing surgeries needing remote expert diagnosis, i.e., districts with disinfection services, safe houses, and satellite communication infrastructure.

Figure 4 (a) shows a preliminary query tree for Query 1. Nodes n_2, n_5, n_6 contain operations filtering out tuples

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering



(a) A preliminary query tree of Query 1

(b) A complete query tree based on the preliminary tree in (a)

Fig. 4. Query trees.

inaccessible to V_5 according to the authorizations (Section 3.1). The selection operations in n_3, n_4 are executed before joins to eliminate unnecessary tuples early. Each node except for n_0 and n_1 specifies a server, named the *consolidator*, indicating the location where the table represented by the node is generated. We assign alias names A, B, C, D for the tables represented by n_2, n_3, n_4, n_1, respectively. So n_0 represents $D \bowtie C = (A \bowtie B) \bowtie C$, that is, the result of Query 1. The query plan represented by the query tree is incomplete, because it lacks query execution information for n_0 and n_1.

One of the complete query trees for the preliminary query plan is shown in Figure 4 (b), which contains query execution information for n_0 and n_1. We will interpret it and explain how it is generated after introducing the query planning algorithm (Section 5).

3 Authorizations and Safe Query Plans

3.1 Pairwise Authorizations

In a collaboration a variety of alliances may be formed among parties. The information shared between two parties depends on the type of alliance they are involved in. Thus, the need for flexible information sharing is usually common in a multi-party collaboration. A simple, yet expressive, authorization specification, named *pairwise authorization*, is proposed to meet the access control requirements **R1–R3**.

Definition 3.1 (Pairwise authorization): A pairwise authorization is a rule of the form $V_i \xrightarrow{r=\sigma(R)} V_j$, where party V_i owns R and defines this rule.

The semantics is that party V_i authorizes party V_j to access r, which is specified using a selection $\sigma(R)$. Hence, from the point of view of V_i, the content of R is r.

Note how the authorization specification satisfies the data protection requirements. First, the tuple set described in an authorization is accessible to a specific party, thus overall each party has its own authorized view (**R1**). Second, it is the data owner who defines the authorization (**R2**). Third, it allows access control over any given tuple (**R3**).

Different from the conventional distributed database system, which defines authorizations in terms of roles and users (instead of parties) with the assumption that all servers hold an identical view, the pairwise authorization implies that servers of different parties may have different views. Therefore, it imposes new challenges on safe query planning when coping with inter-server information flow control.

3.2 Safe Query Plan

Definition 3.2 (Authorized view): Assume party V owns relations $R_1, \ldots, R_m$; pairwise authorizations specify $r_{m+1} = \sigma(R_{m+1}), \ldots, r_n = \sigma(R_n)$ accessible to V. The authorized view of V is the database $\Theta(V) = \{R_1, \ldots, R_m, r_{m+1}, \ldots, r_n\}$

Definition 3.3 (Subrelation): The set of attributes of a relation R is denoted as $attr(R)$. $R[\alpha]$, or $\pi_\alpha(R)$, denotes a projection on R using a subset attributes α of $attr(R)$ A relation R' is a subrelation of a relation R, if $attr(R') \subseteq attr(R)$ and $R' \subseteq R[attr(R')]$.

Given a set of relations $\mathbb{R} = \{R_1, \ldots, R_n\}$, the cross product of the relations in $\mathbb{R}$ is denoted by $\Delta(\mathbb{R}) = R_1 \times \cdots \times R_n$. In distributed query processing, there are two types of query operations in a query plan: data transmission operations and those without data transmission such as selection and projection.

Definition 3.4 (Safe query plan): A data transmission that sends R to a server belonging to party V is safe if R is a subrelation of $\Delta(\Theta(V))$. A complete query plan is safe if all the data operations specified by the plan are safe.

For example, assume $R \in \Theta(V)$ and a server S belonging to V queries $\pi_\alpha(R)$, where $\alpha \subseteq attr(R)$. The query contains only one potential data transmission that sends $\pi_\alpha(R)$ to S. It is safe because $\pi_\alpha(R)$ is a subrelation of R, and hence a subrelation of $\Delta(\Theta(V))$.

4 Safe Join Methods

A query operation with a unitary operator, e.g., selection and projection, is executed inside a server and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

is thus safe because there is no data transmission involved, while a join operation may involve inter-party data transmission. Hence, our interest is to look at and compare different join methods to determine safe and efficient execution strategies. In this section, we first examine how the widely used *semi-join* method in distributed query processing may violate the pairwise authorizations. We then propose a new join method, named *split-join*, which preserves the communication efficiency advantage of semi-joins and is compliant with the access control. Other safe join methods are also discussed in the section.

Cost model. Communication efficiency is usually a most important performance metric in distributed systems. Our optimization target is thus the communication cost. The size of a relation R is denoted as $|R|$. If we ignore the cost of initiating a data transmission, the cost of sending R from S_i to S_j is $p_{i,j} * |R|$ [2], [14], [15]. The cost of a query plan is thus the sum of the cost due to the data transmission operations in the plan.

4.1 Why Semi-join Breaks

We consider the equi-join operation contained in n_1 of Figure 4, which joins the two tables A and B represented by the child nodes n_2 and n_3, respectively. S_q is the querier, while A and B reside in S_1 and S_2. respectively. We consider various party membership of the servers (instead of sticking to that described in Figure 1) showing the conditions under which the join methods can be applied. In each join method, there is a parameter, the consolidator, indicating the server that obtains (consolidates) the join result. If a party has buddy servers, we explore them to save communication costs.

The semi-join method was proposed for efficient join processing in distributed database systems and has been widely used [2]. It assumes the servers have the same view over the database and saves the communication cost by pushing the query processing towards the data sources. Figure 5 (a) shows the operations in a semi-join: (1) S_1 sends $A[district]$ to S_2; (2) S_2 computes $A[district] \bowtie B$ and sends it to S_1. S_1, the consolidator here, computes $A \bowtie (A[district] \bowtie B)$ to get the final join result. Note how steps (1) and (2) help identify tuples in B needed for the join, such that the resultant data transmission can be reduced significantly compared to simply sending B to S_1. Finally no extra data but the join result is sent to S_q.

The semi-join method performs well in distributed systems where all servers hold an identical view. It breaks in our scheme where each party may have a distinct view. Assume S_q, S_1 and S_2 belong to different parties and have different authorized views. The data transmission operations due to a semi-join may result in access control violation. Specifically, some tuples in $A[district]$ may not be accessible to S_2 and similarly some tuples in $A \bowtie (A[district] \bowtie B)$ may be inaccessible to S_1. The pairwise authorization makes the semi-join method not applicable in general.

4.2 Split-join and Other Safe Methods

When S_2 and S_q are buddies, we can consider a safe join method, named *peer-join*, where S_1 simply sends A to S_2. It leverages the fact that buddy servers share the same authorized view, so the data transmission is safe. Figure 5 (b) shows an application of the peer-join. Symmetrically, if S_1 and S_q are buddies while S_2 and S_q are not, S_2 sends B to S_1 to apply the peer-join. Note that if S_1 and S_2 are buddies, the semi-join method is applicable and should be preferred.

Another straightforward and safe join method, named *broker-join*, is to retrieve the tables A and B to S_q, which then performs the join evaluation locally. When S_q is far away from the data sources, the transmission may lead to poor communication efficiency. We can improve the method by exploring buddy servers of the querier near data sources. Figure 5 (c) shows that a broker-join delegated to a buddy S_b of Sq, which may be closer to the data sources S_1 and S_2.

These join methods, except the semi-join method, do not follow the principle of distributed query processing, while the semi-join cannot be applied when S_1, S_2 and S_q belong to three different parties. We propose a new join method, named *split-join*, which applies distributed query processing safely when the three servers belong to different parties. It explores the commonly accessible tuple sets to do partial computation at the data sources, and does the rest of the computation at S_q.

The method splits tuples of a relation into two sets: tuples that can be commonly accessed by the servers involved in the join are in one set, which form the *overlapped authorized view*, while the remaining ones are in the other. Specifically, assume server S_1 is allowed to access tuple set B_1 of B (thus B_1 is the overlapped authorized view of $P(S_q)$ and $P(S_1)$ on B); and $B_2 = B - B_1$ (i.e. $B_2 = \{x | x \in B \wedge x \notin B_1\}$). Assume server S_2 is allowed to access tuple set A_1 of A (thus A_1 is the overlapped authorized view of $P(S_q)$ and $P(S_2)$ on A); and $A_2 = A - A_1$. Then B_1 (A_1 resp.) can be safely sent to S_1 (S_2 resp.) Accordingly the join can be rewritten as

$$A \bowtie B = A \bowtie (B_1 \cup B_2) = (A \bowtie B_1) \cup (A \bowtie B_2)$$
$$= (A \bowtie B_1) \cup (A_1 \bowtie B_2) \cup (A_2 \bowtie B_2)$$

The join is split into three *sub-joins*; the union of the three sub-joins equals the join result.

Figure 5 (d) shows the execution of a split-join where S_q delegates the join to its buddy server S_b. Step (1) transmits A_1 for the sub-join $A_1 \bowtie B_2$, while step (2) transmits B_1 needed in $A \bowtie B_1$. Steps (3) and (4) transmit A_2 and B_2 to S_b, which executes the third sub-join $A_2 \bowtie B_2$, receives the results of the other two sub-joins (steps (5) and (6)), and finally perform a union to get the join result. All the data transmissions are safe according to Definition 3.4. Thus, it is a safe join strategy.

Both the broker-join and the split-join can be applied safely when the three servers involved in the join belong to different parties. To illustrate the potential benefit of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

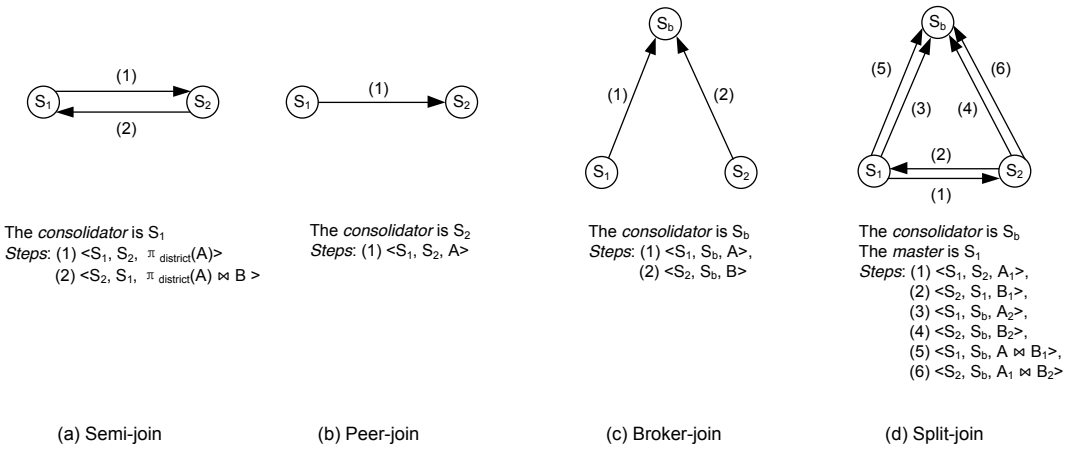


Fig. 5. Join methods. The two relations to be joined, A and B, are at S_1 and S_2, respectively. If S_1 and S_2 are buddies, semi-join is used. Otherwise, if S_2 (S_1 resp.) is a buddy$^+$ of S_q, peer-join is used and S_2 (S_1 resp.) is the consolidator. Finally, if S_1, S_2 and S_q belong to three different parties, broker-join and split-join are applicable and the consolidator S_b is a buddy$^+$ of S_q.

the split-join method, we compare the cost due to a broker-join and a split-join.

$$\begin{aligned} cost(A \bowtie_{broker-join} B) =& p_{1,b} * |A| + p_{2,b} * |B| \\ cost(A \bowtie_{split-join} B) =& p_{1,2} * (|A_1| + |B_1|) + \\ & p_{1,b} * (|A_2| + |A \bowtie B_1|) + \\ & p_{2,b} * (|B_2| + |A_1 \bowtie B_2|) \end{aligned}$$

It can be seen that under some conditions the split-join method saves the communication cost. Specifically, providing that $p_{1,2} << p_{1,b} = p_{2,b}$, i.e., the data sources are close to each other but far from the querier, and the selectivity factor satisfies $|A \bowtie B_1| < |B_1|$ and $|A_1 \bowtie B_2| < |A_1|$),

$$\begin{aligned} & cost(A \bowtie_{split-join} B) \\ & \quad \doteq p_{1,b} * (|A_2| + |A \bowtie B_1| + |B_2| + |A_1 \bowtie B_2|) \\ & \quad < p_{1,b} * (|A_2| + |B_1| + |B_2| + |A_1|) \\ & \quad = p_{1,b} * (|A| + |B|) = cost(A \bowtie_{broker-join} B) \end{aligned}$$

Compared to the broker-join, the split-join method conducts partial join processing at the data sources, so that it does not require transmitting the whole tables as the broker-join does following the principle of distributed query processing. However, we can similarly identify conditions where the broker-join leads to less cost. We will return to this issue later.

A join can be split in two ways:
(1) $A \bowtie B = (A \bowtie B_1) \cup (A_1 \bowtie B_2) \cup (A_2 \bowtie B_2)$.
(2) $A \bowtie B = (A_1 \bowtie B) \cup (A_2 \bowtie B_1) \cup (A_2 \bowtie B_2)$.
To distinguish the two, we use a parameter, named the *master*. When the master is the owner of S_1 (S_2, resp.), split 1 (2, resp.) is adopted.

Theorem 4.1: If $p_{1,b} < p_{2,b}$, split 1 is less costly; otherwise, split 2 is less or equally costly.

Proof:

$$\begin{aligned} cost(A \bowtie_{split1} B) =& p_{1,2} * (|A_1| + |B_1|) + \\ & p_{1,b} * (|A_2| + |A \bowtie B_1|) + \\ & p_{2,b} * (|B_2| + |A_1 \bowtie B_2|) \\ cost(A \bowtie_{split2} B) =& p_{1,2} * (|A_1| + |B_1|) + \\ & p_{1,b} * (|A_2| + |A_2 \bowtie B_1|) + \\ & p_{2,b} * (|B_2| + |A_1 \bowtie B|) \end{aligned}$$

$$\begin{aligned} & cost(A \bowtie_{split1} B) - cost(A \bowtie_{split2} B) \\ & \quad = p_{1,b} * (|A \bowtie B_1| - |A_2 \bowtie B_1|) \\ & \quad + p_{2,b}(|A_1 \bowtie B_2| - |A_1 \bowtie B|) \\ & \quad = p_{1,b} * (|A_1 \bowtie B_1| + |A_2 \bowtie B_1| - |A_2 \bowtie B_1|) \\ & \quad + p_{2,b} * (|A_1 \bowtie B_2| - |A_1 \bowtie B_1| - |A_1 \bowtie B_2|) \\ & \quad = (p_{1,b} - p_{2,b}) * |A_1 \bowtie B_1| \end{aligned}$$

Therefore, if $p_{1,b} < p_{2,b}$, split 1 leads to a less communication cost and should be chosen; otherwise, split 2 should be chosen. □

The join methods can be represented in a uniform form: *join-method*{*consolidator, master*} , where the master is only applicable to the split-join. For example, the four join methods shown in Figure 5 can be represented as semi-join{$S_1, null$}, peer-join{S_2, null}, broker-join{S_b, null} and split-join{S_b, S_1}, respectively.

All applicable join methods should be considered to achieve good overall communication efficiency. A query planning algorithm that considers different query strategies is presented in Section 5.

5 Query Plan Generation

We present an algorithm that, given a set of pairwise authorizations and a preliminary query tree, generates a complete query tree compliant with the authorizations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

Various join methods are considered and all viable strategies are collected and compared to determine the query plan with the minimum communication cost. Since selections, projections and operations filtering tuples according to authorizations are executed inside servers, we ignore them in the algorithm and focus on joins.

We first introduce two structures: *Node* and *Candidate*. The *Node* structure represents the node in the query tree. Its *baseRelation* and *ownerServer* fields are only used for the leaf node to refer to the base relation and the server that stores the relation, respectively. The *operator* field is the query operator taking the relations represented by *lchild* and *rchild* as operands. The *Candidate* structure represents a viable join strategy for generating the relation represented by *Candidate.node*; *Candidate.cost* is the estimated total cost of executing the sub-tree rooted at the node. The *lcand* and *rcand* fields are used to trace back the candidates of the child nodes. The *candList* filed in the *Node* structure contains all the candidate join strategies for the node, while the *chosen* is the chosen one in the final complete query tree.

```
struct Node{
    String baseRelation, ownerServer;
    String operator;
    Node *lchild, *rchild;
    Candidate *candList, *chosen;
};

struct Candidate{
    Node *node;
    Candidate *lcand, *rcand;
    String method, consolidator, master;
    int cost;
};
```

Algorithm 1 takes the querier S_q, the overlay network $G(N, E)$, the preliminary query tree T, and the database profile including the authorizations as the input, and generates a complete query tree that is efficient in communication cost and compliant with the authorizations. The algorithm consists of three steps:

1) Initialize the leaf nodes (Line 2. The code for *InitializeLeafNodes* is omitted). Add a *Candidate* instance for each leaf node. Since a leaf node simply represents a base relation, all the fields except *node* in the *Candidate* instance are *null* or 0.
2) Accumulate candidate query strategies (Line 3). *SearchCandidates* traverses the tree in a post-order and collects feasible join strategies. For each node, it enumerates the combination of the candidates in the child nodes to collect join candidates for the current node (Line 17–21).
3) Trace back. By comparing the cost of each candidate in $candList$ of the root node we can identify the candidate with the least cost. From the chosen candidate we trace back through the tree to assign the candidate in each of the other non-leaf nodes (Line 5).

Algorithm 1 The algorithm for query plan generation

Input: S_q, $G(N, E)$, T, and the database profile.
Output: a complete query plan.

```
1:  procedure GeneratePlan
2:      InitializeLeafNodes(T)
3:      SearchCandidates(T.root)
4:      cand ← the one with least cost in T.root.candList
5:      TraceBack(cand)
6:  end procedure

7:  procedure SearchCandidates(node)
8:      if node is leaf then return
9:      lchild ← node.lchild
10:     rchild ← node.rchild
11:     if lchild ≠ null then
12:         SearchCandidates(lchild)
13:     end if
14:     if rchild ≠ null then
15:         SearchCandidates(rchild)
16:     end if
17:     for all lcand ∈ lchild.candList do
18:         for all rcand ∈ rchild.candList do
19:             ExamineJoins(node, lcand, rcand)
20:         end for
21:     end for
22: end procedure

23: procedure ExamineJoins(node, lcand, rcand)
24:     lserver ← lcand.consolidator
25:     rserver ← rcand.consolidator
26:     if lserver = rserver then
27:         AddCand(intra-server-join, otherInfo)
28:         return
29:     end if
30:     if lserver and rserver are buddies then
31:         AddCand(semi-join, otherInfo)
32:         return
33:     end if
34:     for all buddy+ of S_q do
35:         AddCand(broker-join, otherInfo)
36:     end for
37:     if lserver or rserver is a buddy of S_q then
38:         AddCand(peer-join, otherInfo)
39:     else
40:         for all buddy of S_q do
41:             AddCand(split-join, otherInfo)
42:         end for
43:     end if
44: end procedure

45: procedure TraceBack(cand)
46:     if cand = null then return
47:     cand.node.chosen ← cand
48:     TraceBack(cand.lcand)
49:     TraceBack(cand.rcand)
50: end procedure
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

A variety of join methods may be applicable depending on the party membership of the servers involved in the join. They are considered in *ExaminJoins*. Intra-server join, whenever it is applicable, is superior to any method discussed in Section 4.2. Similarly, if a semi-join is applicable, it is chosen without further considering other join strategies. Otherwise, *broker-join* is examined by taking each buddy$^+$ of S_q into consideration. Finally, either *peer-join* or *split-join* is examined depending on the party membership of the servers.

In *AddCand* (the code is omitted), a *Consolidate* instance is generated; the join method and *otherInfo* (including the consolidator and master parameters for the join operation, the node, and the child candidates) are then filled in. The *Consolidate.cost* is the sum of the cost of the join operation and those specified in the child candidates.

Example 5.1: A preliminary query tree is shown in Figure 4 (a); the topology and the party membership are depicted in Figure 1.

In the following candidate accumulation step, $A \bowtie B$ at n_1 can apply either broker-join or split-join; the consolidator can be S_q or S_5 in either case. Therefore, n_1 has four candidates: *split-join*$\{S_q, S_2\}$ (split 2 is adopted because $p_{2,q} < p_{1,q}$), *broker-join*$\{S_q, null\}$, *split-join*$\{S_5, S_2\}$, and *broker-join*$\{S_5, null\}$. As the former (latter resp.) two candidates both obtain $A \bowtie B$ at S_q (S_5 resp.), an optimization is to compare the cost of the two and keep the better one. Next, when collecting candidates for the root node n_0, each of the candidates in the left child node n_1 is enumerated (the right child node n_4 contains only one candidate). For example, when the candidate *split-join*$\{S_5, S_2\}$ of n_1 is enumerated, the peer-join candidate that sends C to S_5 and the broker-join candidate that sends both $A \bowtie B$ and C to S_q are inserted into the candidate list of n_0.

Finally, by comparing the total costs of the candidate query plans, the most efficient one is chosen. Figure 4 (b) shows a safe complete query tree which applies *split-join*$\{S_5, S_2\}$ to n_1 and *peer-join*$\{S_5, null\}$ to n_4.

6 PROOFS

6.1 Correctness and Safety

In the presence of pairwise authorizations the answer to a query from S_q belonging to party V is correct if the result is equivalent to answering the query in $\Theta(V)$. In order to prove the correctness of Algorithm 1, we first define a reference complete query tree, whose result is obviously correct by planning *a centralized query computation* at the querier. Hence, we can say that a query tree is correct if it always generates the same query result as the reference one.

Definition 6.1 (Reference query tree): Given a preliminary query tree T, its *reference* complete query tree is generated by transforming the nodes of T through a post-order traversal; i.e., starting from the root node, a node is visited after its left and right subtrees are traversed: when a node n with a join operator is visited, if the relations represented by the two child nodes are in the same server, an intra-server join method is assigned to n; otherwise, for the two child nodes, if the relation represented by the child node is not in the querier S_q, a data transmission operation sending the relation to S_q is added to the child node, then an intra-server join method is assigned to n.

Definition 6.2 (Correct query tree): Let T be a preliminary query tree, which specifies a set $\mathbb{R}$ of base tables in its leaf nodes. A complete query tree of T is correct if, given any instance of $\mathbb{R}$, it always computes the same query result as the reference complete query tree of T.

Theorem 6.1: Given a preliminary query tree T, a set of authorizations, the complete query tree T_1 generated by Algorithm 1 is correct and safe according to Definition 6.2 and 3.4.

Proof: We first prove T_1 is correct. Let T_2 be the reference complete query tree of T. T_1 and T_2 have the same tree structure, for they are based on the same preliminary tree T, which implies that for each node n in T there exists a counterpart node in T_1 and one in T_2, denoted by $n_{<1>}$ and $n_{<2>}$, respectively.

The proof is by induction on the nodes of T, showing that for any given node n of T, its counterpart nodes $n_{<1>}$ and $n_{<2>}$ represent the same relation.

Base case. It is trivial that, when n is a leaf node of T, $n_{<1>}$ and $n_{<2>}$ represent the same relation as represented by n.

Induction. Consider a non-leaf node n of T and suppose, by induction, the left (right resp., if any) child node of $n_{<1>}$ represents the same relation as the left (right resp.) child node of $n_{<2>}$. Since our algorithm only impacts nodes whose operators are joins, we consider node n with a join operator. The input relations of the join operations are the same for $n_{<1>}$ and $n_{<2>}$. According to the definition of the join methods (i.e., semi-join, peer-join, broker-join and split-join), no matter which of the join methods is applied, $n_{<1>}$ generates the same join result as $n_{<2>}$, which applies an intra-server join.

We then prove the safety of T_1 by induction on its nodes, showing each data transmission step involved in each node of T_1 is safe, that is, the receiver is allowed to see the transmitted data. Note that tuples of any base table inaccessible by the querier are filtered out by the table owner. Thus, according to Definition 3.4 any query results computed based on these filtered tables are accessible by the querier.

Base case. The case when n is a leaf node is trivial, for it does not involve data transmission.

Induction. Consider an non-leaf node n of T_1 with a join operator and suppose, by induction, that the consolidator $lcon$ of the left child and the consolidator $rcon$ of the right child are authorized to see the relations represented by the two child nodes, respectively. Now we consider case by case when each of the four join methods is applied.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

Semi-join. In this case, $lcon$ and $rcon$ are buddies and have the same authorized view, so the data transmission is safe.

Peer-join and broker-join. The only data receiver, which is also the consolidator, is a buddy$^+$ of S_q. It is authorized to see the intermediate query results represented by child nodes of n.

Split-join. According to split1 and split2, $lconf$ ($rconf$ resp.) only sends tuples accessible by $rconf$ ($lconf$ resp.) to $rconf$ ($lconf$ resp.). The consolidator is a buddy$^+$ of S_q, which can access intermediate query results. □

6.2 Authorization Confidentiality

Definition 6.3 (Authorization confidentiality): Given a query operation, if each server involved in the execution only needs to refer to authorizations defined by its own party, the operation satisfies *authorization confidentiality*. A query plan satisfies authorization confidentiality, if all its operations satisfy authorization confidentiality.

During query execution, authorizations are used in two cases. One is when a server that owns a base table involved in the query filters out tuples inaccessible by the querier according to the local authorizations; hence it does not need authorizations from foreign servers. The other case is to guide table splits when executing split-joins. The execution of other operations including selections, projections, peer-joins and broker joins do not need to refer to any authorizations. We prove that servers involved in a split-join only need to refer to authorizations defined by its own party. We first introduce a lemma.

Lemma 6.2: Given a complete query tree T generated by the algorithm, when T is executed information is never flowed from servers in the party of the querier to other parties' servers.

Proof: Let S_q be the querier. We examine the information flow for each inter-server join method. (1) semi-join: information is flowed between buddy servers belonging to the same party. (2) peer-join, broker-join and split-join: if we examine operations in each of the join methods (Figure 5), data is never transmitted from a server of $P(S_q)$ to servers not belonging to $P(S_q)$. □

Lemma 6.2 can be interpreted in an intuitive way using the query tree. Each join operation is associated with a non-leaf node in T. The tree edges linking a non-leaf node and its child nodes indicate how the information flows from the consolidators of child nodes to the consolidators of the non-lead node. Lemma 6.2 is equivalent with the conclusion that a node with a buddy$^+$ of S_q as the consolidator does not have any ancestor node associated with a foreigner of S_q as the consolidator; otherwise, it violates the lemma by flowing information from a buddy$^+$ of S_q to a foreigner of S_q.

Theorem 6.3: Given a preliminary query tree, the complete query tree T generated from the preliminary query tree by Algorithm 1 satisfies authorization confidentiality.

Proof: Let S_q be the querier. The proof is by induction on the nodes of T, showing the sub-tree rooted at any node n of T satisfies authorization confidentiality.

Base case. The case when n is a leaf node satisfies authorization confidentiality trivially, for n represents a base relation and does not need any authorizations.

Induction. Consider a non-leaf node n and suppose, by induction, that the sub-tree T_{n_l} rooted at n's left child n_l and the sub-tree T_{n_r} rooted at n's right child n_r (if any) satisfy authorization confidentiality. As mentioned above, if the join operation associated with n is not a split-join, it does not need to consult any authorizations. Suppose the split-join method is adopted in n and the consolidators of n, n_l and n_r are S, S_l and S_r, respectively. According to the definition of the split-join method, S is a buddy$^+$ of S_q, while S_l and S_r are not. Therefore, none of the consolidators of the nodes in T_{n_l} is a buddy$^+$ of S_q; otherwise, it violates Lemma 6.2 by flowing information from a server in $P(S_q)$ to S_l.

Considering that broker-joins, peer-joins and split-joins all require a buddy$^+$ of S_q as the consolidator, the only inter-server join method that can be adopted by nodes of T_{n_l} is semi-join, which means that all the consolidators of the nodes of T_{n_l} belong to the same party, $P(S_l)$. Thus, all the base tables in T_{n_l} are owned by servers of $P(S_l)$. So, S_l only needs authorizations defined by servers of $P(S_l)$ to determine how to split the relation represented by n_l in the split-join operation.

Similarly, we can prove that S_r only needs authorizations from servers of $P(S_r)$. Therefore, the operation associated with n satisfies authorization confidentiality. □

7 EVALUATION

Parameters	Values
Number of parties	9
Number of servers per party	$\{1, 3, 5, 7\}$
Link cost	$[1, 50]$
Number of relations in a server	100
Cardinality of a relation	$[100, 50000]$
Number of attributes in a relation	$[3, 10]$
Overlapping ratio of the authorized views	$\{0.25, 0.5, 0.75\}$

TABLE 2
Evaluation parameters

In order to evaluate Algorithm 1, we have coded a simulator in Java.[1] Parameters used in the evaluation are listed in Table 2. Specifically, 9 parties are involved; the number of servers per party in the coalition network is 1, 3, 5 or 7 between experiments. Each network is generated based on the Erdös and Renyi random graph model [7] by connecting any two servers in the network at a probability of 0.1. The cost of each link is a random

1. The simulator has more than 6000 lines of code and could be made available upon request. Similar to work in the literature [2], [19], the simulator focuses on query planning and does not implement any part of the query executer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

number between 1 and 50, and servers communicate using the shortest-path routing, unless otherwise specified. The overlapping ratio of the authorized views of parties is a value randomly picked from the listed set; e.g., a ratio= 0.25 means that, given any two parties and a table not owned by the two parties, the subset of the tuples in the table that is commonly accessible by the two parties occupies 25% of the table.

Each data point in the figures of curves in this section is obtained by issuing 1000 join queries and calculating the average communication cost. Each join query is generated by randomly choosing the querier server and servers storing the base relations involved in the query from the coalition network; we force that these servers belong to different parties in order to avoid trivial intra-party queries. Each server owns 100 relations, each with the cardinality and the number of attributes randomly assigned within the listed ranges in Table 2. Each join operand, i.e., base relation, is also randomly picked from the randomly chosen server. Unless otherwise specified, the size of the resulting relation due to a join is defined as $0.001 * (|A| + |B|)$, where A and B are the operands of the join; the join selectivity that leads to such a join size is referred to as a *low* join selectivity.

To our knowledge, the centralized query evaluation, which pulls all authorized data involved in a query to the querier to perform query execution, is the only safe query approach compliant with pairwise authorizations on current distributed database systems. We call it *the baseline query evaluation*. This approach has been used in federated database systems such as GaianDB [1]. To better present the results, *the communication cost of each query in our experiments is normalized by the cost due to the baseline query evaluation*.

Our evaluation first demonstrates the benefit of the split-join method (Section 7.1), then the advantage of the planning algorithm (Section 7.2). Next, we investigate the impact of a series of factors, including the overlapping ratio of the authorized views (Section 7.3), join selectivity (Section 7.4), link costs (Section 7.5), and network types (Section 7.6). We finally investigate the query response time (Section 7.7) and the effect of pairwise authorizations (Section 7.8).

7.1 Benefit of Split-Joins

In order to evaluate the benefit of the new split-join method, we set the number of servers per party as 1 to avoid the effect of buddy servers. In addition, as all base relations involved in a query are from different parties, semi-joins cannot be applied, while broker-joins and peer-joins in this case actually send base relations to the querier, as the baseline query evaluation does. Therefore, experiments with this setting are able to demonstrate the benefit solely due to split-joins.

We vary the number of join operands from 2 to 8. As shown by the 1 *server per party* line in Figure 6, the normalized communication cost decreases from 86% to 66% as the number of join operands grows. The average communication cost saving is 26%.

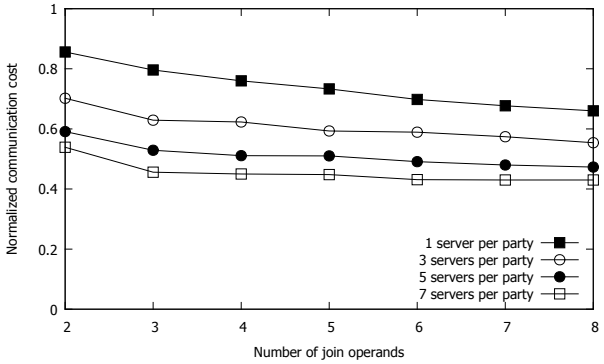


Fig. 6. Algorithm performance under different numbers of servers per party.

7.2 Communication Cost Saving

We then investigate the combined advantage of exploiting various join methods and buddy servers. Three sets of experiments are performed with 3, 5, and 7 servers per party, respectively. Figure 6 illustrates the three cases in contrast to the case of one server party described in Section 7.1. It shows that by combining various join methods and buddy servers the communication cost can be further reduced. The minimum communication cost reduction is 30% in the setting of two join operands and three servers per party. The communication saving keeps increasing when there are more buddy servers and join operands. The cost reduction is up to 57%, while the average communication cost saving of all the three cases is nearly half (48%).

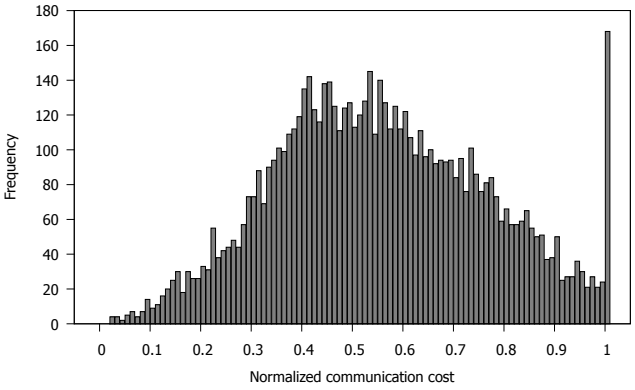


Fig. 7. The distribution of query costs.

In addition to the average cost, it is useful to see the costs of individual queries. We take the case of 5 *server per party* as an example. 1000 randomly generated queries are issued for each different number of join operands from 2 to 8; thus there are totally 7000 queries. Figure 7 shows the distribution of the costs of the 7000 queries. The randomly generated queries lead

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

to a diversity of communication cost savings. For only 2.3% of the queries (168 of 7000) our algorithm generates a plan with the cost equal to the baseline cost, while for all the other queries our algorithm saves communication costs. In the best case our algorithm saves 98% of the communication cost.

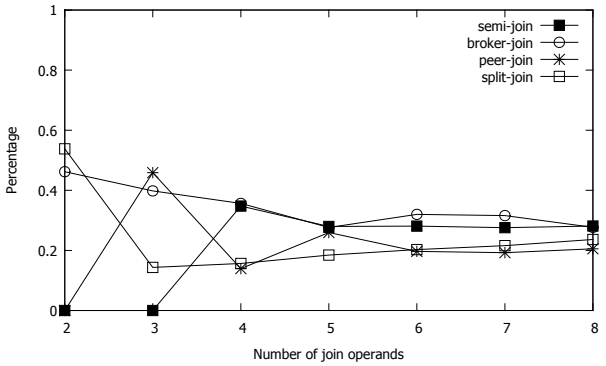


Fig. 8. Percentages of different join methods.

To better understand the query plans generated by our algorithm, we calculate the percentage of each join method used in the queries above. As shown in Figure 8, all the four inter-server join methods are considered in our query planning. Semi-joins are not used when a query only involves 2 or 3 join operands; the reason is that a safe use of a semi-join requires that the two servers involved are buddies, while our setting forces that base relations belong to different parties. As the number of join operands grows, cases where various join methods can be used increase, e.g., the semi-join is applicable when joining two intermediate relations in two buddy servers; hence, the percentages of different join methods become closer.

7.3 Impact of the Overlapping Ratio

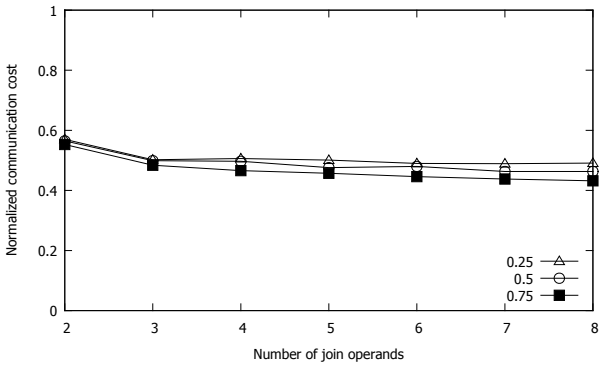


Fig. 9. Algorithm performance under different overlapping ratios of authorized views.

Next we measure how the overlapping ratio of authorized views between parties affects the performance of split-joins. So we perform three sets of experiments with the ratio equal to 0.25, 0.5 and 0.75, respectively. There are 5 servers per party in all the experiments. As shown in Figure 9, a larger ratio improves the communication cost slightly. For example, in the case of 8 join operands with a ratio 0.25 and 0.75 the normalized communication cost is 0.48 and 0.42, respectively. Note that the overlapping ratio only affects split-joins, and a higher overlapping ratio leads to a larger A_1 and B_1 (Figure 5 (d)). Since the split-join is usually adopted when $p_{1,2}$ is relatively small, the sizes of A_1 and B_1 affect the total join cost slightly.

7.4 Impact of Join Selectivities

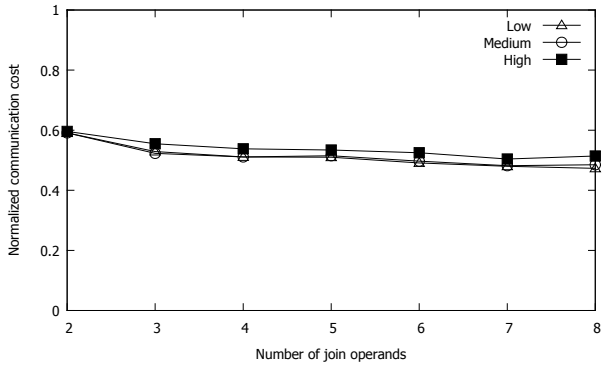


Fig. 10. Algorithm performance under different join selectivities.

Next, we evaluate the impact of the join selectivity. We consider three types of join selectivities: low, medium and high. The low join selectivity is defined above, while the medium and high join selectivities are 10 and 100 times of the low join selectivity, respectively. We perform experiments with 5 servers per party and a varying number of join operands.

Even with the *high* join selectivity, the size of the join result is much smaller than the size of the relations to be joined. The cost of a join query is thus mainly determined by the cost of transmitting the base relations to be joined, while the cost of sending the join result is relatively small. Hence the change of the join selectivity does not affect the total cost much, although a larger join selectivity leads to a slightly higher communication cost, as shown in Figure 10.

7.5 Impact of Link Costs

So far, the cost of each link in the network is a randomly assigned value between 1 and 50. We then evaluate the impact of links costs by randomly picking them from different ranges. As illustrated in Figure 11, with a greater link cost variance, our algorithm performs better due to the increased variance of *path* costs that favors distributed query processing. The performance becomes stable when the range changes from 1–50 to 1–100. The reason is that due to the use of the shortest-path routing the variance of path costs stops growing when further enlarging the range.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

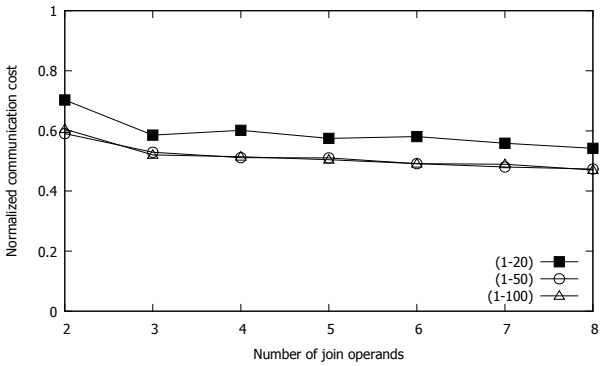


Fig. 11. Algorithm performance under different ranges of link costs.

7.6 Impact of Network Types

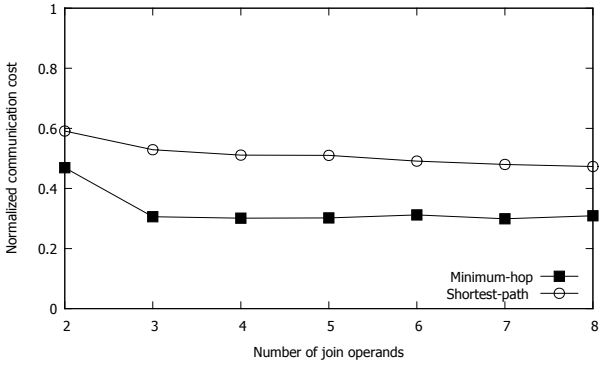


Fig. 12. Algorithm performance under two types of networks.

The experiments above run on networks using the shortest-path routing. We then evaluate our algorithm using the minimum-hop routing. In the network each pair of servers is connected by a link, the cost of which is a randomly assigned value between 1 and 50.

Figure 12 shows the algorithm performance under the two different networks. The settings in both cases are the same and both have 5 servers per party. Compared with the cost saving of half on average in the case of the shortest-path routing network, the cost reduction in the case of the minimum-hop routing is 68%. The results demonstrate that our algorithm saves communication costs significantly for database systems on both shortest-path and minimum-hop routing networks.

7.7 Query Response Time

Another important aspect of query planing is to reduce query response time. Query response time is mainly determined by two factors, data transmission between servers and computation inside servers. Conventional distributed query optimization work usually considers data transmission only [14], [15], as network delays are considered orders of magnitude larger than computation time. We also consider delay due to data transmission only when estimating the response time.

The time due to a single data transmission is the product of the data size and the time to send a unit of data, which is proportional to the communication cost. Based on parallel query processing, given a query plan tree, its query response time is estimated using a bottom-up approach [15], [19]. Specifically, the response time of a node is equal to the maximum of its children's response time plus the data transmission time involved in the query operation due to that node. When multiple servers send data to one server, e.g., in the baseline query, the response time is the sum of each data transmission time [19]. Compared to the baseline query plan, the query plan generated by our algorithm reduces the response time, not only because of the smaller communication cost, but also due to data transmission among many different pairs of nodes allowing parallel processing.

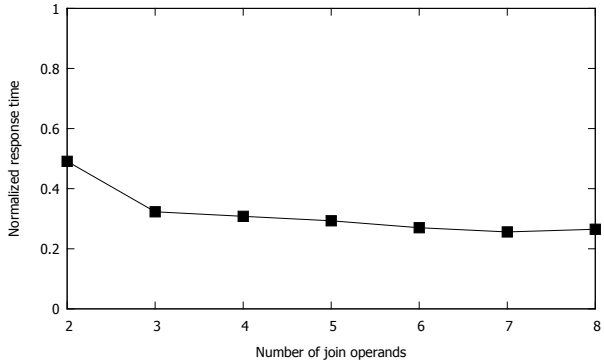


Fig. 13. Query response time.

Figure 13 shows the response time results, each of which is normalized to the response time due to the baseline query plan. The query plans generated by our algorithm reduce the response time by 69% on average.

7.8 Effect of Pairwise Authorization

Due to pairwise authorizations, parties have different views on tables. We are interested to evaluate the effect of pairwise authorizations by measuring the difference of query results when servers of different parties issue the same query. So the following case is considered. We have 2 coalition parties, P_1 and P_2, and 8 tables, $R_1...R_8$. Each party can randomly access X of the tuples in each table, where X is 40%, 60% or 80%. Given a query $Q = R_i \bowtie ... \bowtie R_j$, we calculate the overlapping value of P_1's view $V_1(Q)$ and P_2's view $V_2(Q)$ on Q as follows:

$$overlapping\ value = \frac{|V_1(Q) \cap V_2(Q)|}{|Q|}$$

We then run a simulation based on the case. For each number of join operands, we issue 1000 randomly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2014.2357018, IEEE Transactions on Knowledge and Data Engineering

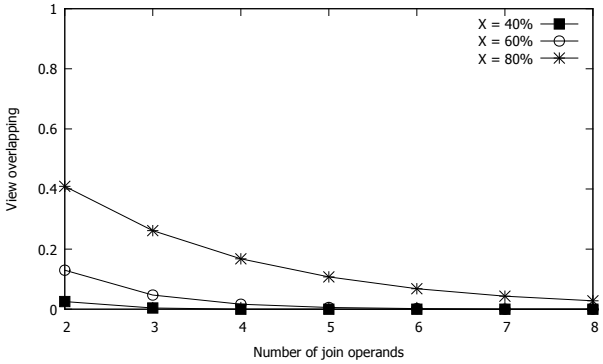


Fig. 14. Effect of pairwise authorizations.

generated queries and use the geometric mean of overlapping values as the result. The results shown in figure 14 illustrate that when X is 40% the overlapping is near zero. This means that the two parties obtain very different results for the same query. In the case of X=80%, the overlapping value is 40.9% when the number of join operand is 2, and it decreases to 2.8% quickly when the number increases to 8. In summary, pairwise authorizations cause significant query result differences between parties.

8 Discussion

Data replication and fragmentation are common in distributed databases. Relation replication increases the search space for query plan generation; for a given relation, multiple copies stored in different servers should be considered. As our algorithm supports multiple candidates for each operation, it is straightforward to adapt the algorithm to addressing relation replication.

In practice, it is not uncommon that the fragments of one relation are saved in accordance with the placement of another, which satisfies *placement dependency* [22]. In this case, given a join query, fragments of relations get joined at servers storing the fragments separately, then the join result is obtained by unioning the fragments.

In general, there is no correlation between the locations of the fragments of multiple relations. Many join strategies are possible. For example, given a join query, the join can be conducted among the fragments then the join results are unioned to deliver the final result; or, relations are obtained from the fragments first and then get joined. Research on how to deal with an explosion of possibilities is needed.

9 Related Work

In a distributed system owned by a single organization, servers belong to the same party. The view disparity between the servers does not exist; therefore, no access control is enforced for the inter-server information transmission [4], [11], [13], [21], while in our setting the inter-server information flow control has to be enforced due to the view disparity.

A variety of schemes have been proposed for information sharing in coalitions and collaborations. In a federated database system each party exports some information [3], [9], [12], [18], [20]. All parties involved in the federated system share the same federated view, so that the query methods and the query planning algorithms used in the single-party distributed system are applicable, which is the advantage of the federated system due to its simplicity. The drawback is the lack of authorization flexibility: given a piece of information, the owner either shares it with all peer parties or none of them. Our scheme, however, allows a party to authorize different peer parties to access different portions of its information.

Similar authorization flexibility can be achieved by maintaining one mini federated system for each pair of parties, so that a party can decide the information to be exported in each federated system. Nevertheless, it is costly and tedious to manage so many federated systems. In addition, query optimization is limited in such small systems. Our scheme achieves multi-party information sharing in a single distributed system.

The scheme proposed by Vimercati et al. allows each party to have a different view in such multi-party information sharing and performs distributed query processing [5], [6]. The core difference between that scheme and our scheme is the targets of access control: they target access control over attributes of tables, while our scheme enables access control over tuples. How to combine the two approaches in a single system is an interesting research question. In their scheme to define or alter an access control rule may need agreement of multiple parties; hence it does not satisfy the requirement of authorization autonomy. In addition, their scheme does not consider the network structure, while we consider a coalition network of multiple servers per party and propose to exploit *buddy servers* during query planning to save communication costs.

10 Conclusions

In multiple-party collaborations, how to share information safely and flexibly is an important problem. We presented an effective solution to enforcing safe and flexible access control. It allows each party to independently define policies that describe which of its tuples can be accessed by which peer party. Our algorithm generates query plans enforcing such access control policies. We have proved the safety and correctness of the algorithm. The experiments show the advantage of the algorithm in saving the communication cost. Therefore, the solution provides specification and enforcement for safe and autonomous information sharing in multi-party collaborations with high communication efficiency.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive suggestions and comments.

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

REFERENCES

[1] G. Bent, P. Dantressangle, D. Vyvyan, A. Mowshowitz, and V. Mitsou. A dynamic distributed federated database. In *2nd Annual Conference of International Technology Alliance*, 2008.

[2] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 6:602–625, December 1981.

[3] J. B. Bocca, M. Jarke, and C. Zaniolo. An approach for building secure database federations. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 24–35, 1994.

[4] S. Castro, M. Fugini, and P. Samarati. *Database Security*. 1995.

[5] S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Controlled information sharing in collaborative distributed query processing. In *Proceedings of The 28th International Conference on Distributed Computing Systems*, pages 303–310. IEEE Computer Society, 2008.

[6] S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Authorization enforcement in distributed query evaluation. *Journal of Computer Security*, 19(4):751–794, 2011.

[7] P. Erdos and A. Renyi. On random graphs. *Publicationes Mathematicae*, pages 290–297, 1959.

[8] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. 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*, pages 628–647, 2011.

[9] D. Heimbigner and D. Mcleod. A federated architecture for information management. *ACM Transactions on Information Systems*, 3:253–278, 1985.

[10] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 168–177.

[11] S. Jajodia and R. Sandhu. Toward a multilevel secure relational data model. *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 50–59, 1991.

[12] W. Kim, N. Ballou, J. F. Garza, and D. Woelk. A distributed object-oriented database system supporting shared and private databases. *ACM Transactions on Information Systems*, 9(1):31–51, 1991.

[13] T. Lunt and E. Fernandez. Database security. *SIGMOD Rec.*, 19:90–97, 1990.

[14] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[15] S. Pramanik and D. Vineyard. Optimizing join queries in distributed databases. *Software Engineering, IEEE Transactions on*, 14(9):1319–1326, 1988.

[16] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34.

[17] J. Serbu. DoD learns important lesson in developing coalition network, 2011. http://www.federalnewsradio.com.

[18] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22:183–236, 1990.

[19] R. Taylor. Query optimization for distributed database systems, 2010. University of Oxford.

[20] M. Templeton, E. Lund, and P. Ward. Pragmatics of access control in mermaid. *IEEE Data Eng. Bull.*, 10(3):33–38, 1987.

[21] P. Wilms and B. Lindsay. A database authorization mechanism supporting individual and group authorization, 1981. Research Report RJ 3137, IBM Almaden Research Laboratory.

[22] C. T. Yu, C. C. Chang, M. Templeton, D. Brill, and E. Lund. Query processing in a fragmented relational distributed system: Mermaid. *IEEE Trans. Softw. Eng.*, 11:795–810, 1985.

[23] Q. Zeng, J. Lobo, P. Liu, S. Calo, and P. Yadav. Safe query processing for pairwise authorizations in coalition networks. In *Annual Conference of International Technology Alliance*, 2012.

Qiang Zeng is a Ph.D. candidate in CSE at Penn State University. He received the B.E. and M.E. degrees from Beihang University, Beijing. He is interested in software security.

Mingyi Zhao is a Ph.D. candidate in the College of Information Sciences and Technology in Pennsylvania State University. He received his B.E. degree in computer science and technology from the University of Science and Technology of China in 2011. His research interests include distributed database security and information flow security.

Peng Liu is a Full Professor of Information Sciences and Technology at Penn State University. He received his Ph.D. degree from George Mason University in 1999. His research interests are in all areas of computer and network security.

Poonam Yadav is a research associate at Computing Department at Imperial College London, where she received her Ph.D. in 2011. Her research interests include Wireless and Distributed Networks.

Seraphin Calo is a Research Staff Member at IBM Research and currently manages the Network Science group within that organization. He received the M.S., M.A., and Ph.D. degrees in electrical engineering from Princeton University.

Jorge Lobo is an ICREA Research Professor in the Department of Information and Communication Technologies at UPF since October 2012 - on leave from IBM T. J. Watson Research Center. Jorge received a Ph.D. in Computer Science from the University of Maryland at College Park. He is an ACM Distinguished Scientist.