Safe Query Processing for Pairwise Authorizations in Coalition Networks

Qiang Zeng[†], Jorge Lobo[*], Peng Liu[†], Seraphin Calo[*], Poonam Yadav[*]

[†]Penn State University [*]IBM Watson Research Center

***Abstract*—In a coalition network where database servers of multiple parties are linked to facilitate information sharing, a data owner usually wants to authorize different portions of its information, in an antonomous way, to be accessible by different peer parties; consequently, each party has a distinct authorized view over the data stored in the coalition network. The requirements, however, impose new challenges on safe query evaluation, because query planning has to resolve the view discrepancy between parties due to the independently defined authorizations, ensuring each server does not receive data it is not authorized to see, and meanwhile guarantees correct query results. We present an algorithm that, given a set of authorizations and a query plan, generates a safe execution strategy. The algorithm enables per-party access authorization and autonomous access control. Following the principle of efficient distributed computation, it explicitly supports distributed query evaluation. A new join method, named *split-join*, is proposed, significantly reducing the inter-server data transmission cost by pushing the join computation as close as to the data sources. The proofs of the correctness and safety of the algorithm are presented. The performance evaluation result is reported.**

I. Introduction

Emerging scenarios for collaborative missions in various areas require different organizations to exchange information by means of sharing data 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 links US troops and their counterparts from several allied nations, has fundamentally changed the way the multinational efforts have been conducted over the past several years. As information is flowed outside party boundaries, the need for data protection is as strong as data exchange.

In this work, we consider a coalition network linking relational databases of several parties.[1] In particular, through a research project jointly sponsored by the US Army and the UK DoD, we have collected a set of principal data protection needs of a multinational military mission in Afghanistan. Based on these protection needs, we conclude the following three access control requirements, which we believe are common for non-military coalition networks as well. (1) *Per-party authorization profile* (**R1**): in a collaboration involving multiple parties, various bilateral relations are maintained between parties. Hence, for example, the fact that party A discloses some specific data to party B does not imply that A feels necessary or comfortable 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 a distinct 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 in terms of its data. It is usually impractical to assume a centralized entity (e.g., a supervisor) defining authorizations on behalf of all parties. It is also inflexible and inefficient 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 the vertical and horizontal perspectives, respectively. While regulating the accessibility of attributes (*vertical access control*) is important in some scenarios, the ability to control the access to any given tuple (*horizontal access control*) is needed in most cases.

[1]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.

Although many authorization models and enforcement mechanisms have been proposed in the literature, none has addressed **R1–R3** simultaneously in a coalition network. Classic distributed database systems assume a single organization [1], [6], [14], [16], [20]. They do not consider inter-party access control at all. Federated databases usually define authorizations in terms of *roles* [5], [12], [15], [18], [19]. Users with the same role, even if they are from different parties, have identical access privileges. This violates **R1**. In addition, they usually assume a centralized entity who defines the authorizations, so **R2** is not satisfied either. The work by Sabrina et al. [8]–[10] defines an authorized view for each party, which starts to realize **R1**; however, they deal with the accessibility of columns rather than tuples (**R3**). Due to the orthogonal purposes, their model cannot be applied to solving our problem. Besides, in their model, authorizations regulating the join operations between relations belonging to multiple parties are supposed to be defined by these parties collaboratively, which does not meet **R2**.

We propose to adopt a simple, yet expressive, authorization specification, named *the pairwise authorization*, to address the access control requirements **R1–R3**, and we present a query planning algorithm enforcing such authorizations in coalition networks. Each pairwise authorization involves two parties: the data provider (i.e., the data owner) and the consumer. The data provider defines authorizations describing which tuples owned by the provider are authorized to be accessible by the consumer. This type of authorization specifications is not new; it essentially corresponds to a generic view definition, thus it is straightforward to be seamlessly integrated into current database systems.

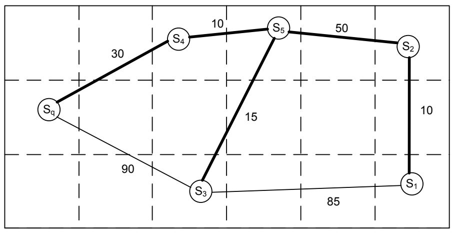
While pairwise authorizations define a distinct authorized view for each party, our algorithm, given a query, generates a safe query execution plan that ensures the query computation exposes to each party only data that the party is authorized to see. An important feature of the algorithm is its explicit support for the collaborated query computation. Instead of simply drawing all the base tables involved in a given query to some server to complete the query computation, the algorithm explores possibilities to execute the computation near the data sources, which is a principle for efficient query processing in a distributed setting. To simultaneously achieve two goals, namely (1) push query computation as close as possible to the data sources, and (2) keep compliant with pairwise authorizations, a new join method, named *split-join*, is designed. It exploits the overlapped authorized view between the join participants, significantly reducing the data transmission cost, while correctly dealing with the possible view discrepancy caused by the per-party authorized view. Leveraging the fact that servers belonging to the same party, named *buddy servers*, share the same authorized view, we involve buddy servers as third participants in join planning to further reduce query evaluation cost. Besides, a significant property of our algorithm is that the generated query execution plan preserves authorization confidentiality. That is, each server involved in the query computation only refers to its owner party's authorizations to execute the plan; therefore, the privacy how each party defines the authorizations is protected. We proved the correctness, safety, and authorization confidentiality property of the proposed algorithm.[2] The experiments show that the query plan generated by our algorithm saves the communication cost by 60%.

The remainder of the paper is organized as follows. Section II introduces the database system over a coalition network. Section III introduces pairwise authorizations and safe query processing. Section IV presents different join methods including the novel split-join. Section V illustrates the query plan generation algorithm, and Section VI discusses its correctness, safety. Section VII presents a performance evaluation of the algorithm. Section VIII and IX discuss related work and conclusions, respectively.

II. Queries in a Coalition Network

We consider a coalition network which links a set of database servers $N = \{S_1, \ldots, S_n\}$ belonging to different parties. Each server contains one or more relations involved in the information sharing in the coalition network. An overlay network $G(N, E)$ is constructed among the server 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 data along the path between two servers S_i and S_j is denoted as $p_{i,j}$, which is the sum of

[2]The proof is omitted in this paper due to space limitations.



S_1: Safehouse(<u>safehouseID</u>, type, district). S_2: Service(<u>employeeID</u>, <u>service</u>, district). S_3: Communication(<u>stationID</u>, function, district). The underlined fields above are the table keys. Party membership: $V_1 = \{S_1\}$, $V_2 = \{S_2\}$, $V_3 = \{S_3\}$, $V_4 = \{S_4\}$, $V_5 = \{S_5, S_q\}$. The numbers indicate the link costs. The bold lines mark the shortest path tree rooted at S_q.

Fig. 1. A coalition network for a disaster recovery mission.

the costs of individual links along the path. The shortest path is always adopted when transmitting data between servers. We assume that, to protect data confidentiality, data transmission is encrypted using the session key, which is unique for each pair of parties, so that when S_i sends data to S_j via S_k, S_k cannot reveal the data that has been encrypted using the session key only shared between S_i and S_j. Different from conventional distributed database systems, the servers in N do not belong to a single party but a number of parties $V_1, \ldots, V_m$. Each server belongs to one and only one party. The party that server S belongs to is denoted as $P(S)$. If two servers belong to the same party, they are *buddies* of each other; otherwise, *foreigners*. A *buddy*$^+$ of server S refers to a buddy server of S or S itself.

Example 2.1: Figure 1 represents a coalition network constructed for a disaster recovery involving multiple parties. The mission takes place in a large area, which has been 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. The coalition network involves five parties and six servers: $V_1 = \{S_1\}, V_2 = \{S_2\}, V_3 = \{S_3\}, V_4 = \{S_4\}, V5 = \{S_5, S_q\}$. S_5 and S_q are called buddies. 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 tables are as shown in Figure 1.

Parties in the coalition network share information by contributing and querying data. We consider simple yet typical queries of the form *"SELECT-FROM-WHERE"*. Given a query, the query optimizer compares the 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 query tree represents base relations accessed by the query, while each non-leaf node represents a relation obtained by applying one relational operator to its child nodes. A query tree may have different execution plans. The reason is that, given the operator in an non-leaf node, the locations where the computation is conducted and the execution algorithms lead to a lot of possibilities. we call a query tree whose non-leaf nodes contain the query operators but lack concrete query execution information a *preliminary* query tree, while a query tree that contains complete information for the query evaluation including the operator, the concrete method and the involved servers for each non-leaf node a *complete* query tree. How to enumerate the possible preliminary query trees for a given query has been extensively researched [13], [17], while our work is, given a preliminary tree, to generate a complete query tree which is compliant with the authorizations.

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. 2. Query.

Example 2.2: Figure 2 shows a query, *Query 1*, issued by server S_q belonging to party V_5, which is responsible for medical assistance. The query searches locations suitable for performing surgeries needing remote expert diagnosis, which means finding the districts that have disinfection services, safe houses, and satellite communication infrastructure.

Figure 3 (a) shows a preliminary query tree for Query 1. Nodes (n_2, n_5, n_6) contain operations filtering out tuples inaccessible by servers of V_5 according to the authorizations (Section III-A). The selection operations in n_3, n_4 are executed before joins to eliminate unnecessary tuples early. All nodes except n_0, n_1 specify a server, named the *consolidator*, indicating the location where the tables represented by the nodes are obtained. We assign alias names A, B, C, D for tables represented by n_2, n_3, n_4, n_1, respectively, so n_1 represents $A \bowtie B$ and n_0 $(A \bowtie B) \bowtie C$, which we also use to represent Query 1. Note that we cannot assign n_1's consolidator as, for example, S_1, because it may be not safe to allow S_1 to see $A \bowtie B$. The complete query tree in Figure 2 (b), which is obtained based on the preliminary query tree in Figure 2 (a), shows a safe query plan. We will interpret it after introducing the query planning algorithm (Section V).

III. Authorizations and Safe Query Planning

A. Pairwise Authorizations

In a collaboration, many types of alliances may be formed by parties, and what information can be shared between parties may depend on the type of alliance they are involved in. Thus, the need for discriminative information sharing can be common and critical in many multi-party collaborations. A simple, yet expressive, authorization specification, named pairwise authorizations, can be used to meet such access control requirements in collaborations.

Definition 3.1: An pairwise authorization is a rule of the form $V_i \xrightarrow{Q_{\{R\}}} V_j$, where:

(1) $Q_{\{R\}}$ describes a subset of the tuples in table R.

(2) A server of party V_i owns R and defines this rule.

(3) V_j is the party of the data consumer.

The semantics of an authorization is that the server that owns R authorizes the servers of V_j to access the set of tuples described by $Q_{\{R\}}$. Note how the simple authorization specification satisfies the data protection requirements. First, the data owner can define a different accessible tuple set per peer party; the authorized view of a given party (see Definition 3.2) is equal to the union of the views specified by the authorization whose data consumer is the party (**R1**). Second, it is the data owner who defines the authorizations releasing the access to its data (**R2**). Third, the specification has fine-granularity control over the visibility of any given tuple (**R3**).

The form of $Q_{\{R\}}$ is not specified. In practice, it can be a simple query describing a subset of tuples in R:

SELECT * FROM R **WHERE** *constraint*,

where *constraint* is a disjunction and/or conjunction of literals in the form of A_i op *value* where A_i is an attribute of R, $op \in \{<, >, =, \neq\}$ and $value \in dom(A_i)$.

This kind of tuple-set-based access release is not new. It essentially corresponds to a generic view definition, thus it is simple enough to be seamlessly integrated into existing database systems. Note that we do not consider multilevel security concepts [14], which we regard as an orthogonal issue of horizontal access control.

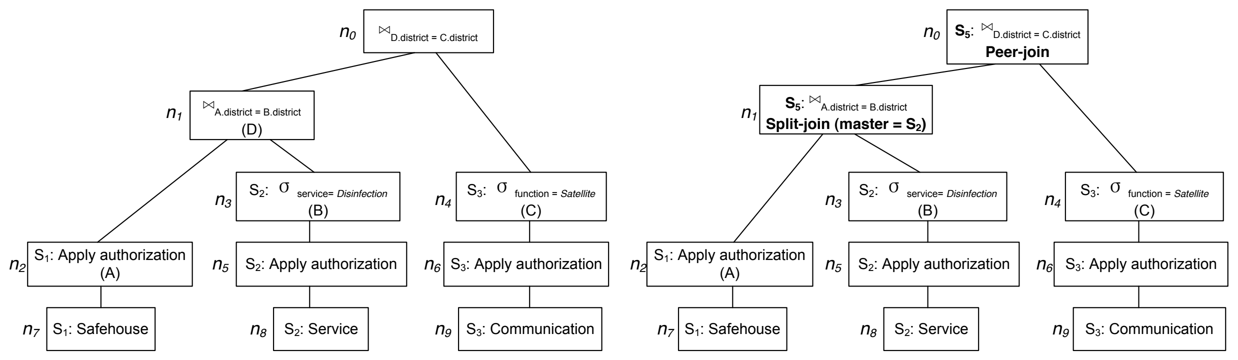
B. Safe Query Plan

Definition 3.2: Given a set of authorizations $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ where the data consumer is party V, the authorized view of V is $\Theta(V) = \{Q_1, \ldots, Q_n, R_1, \ldots, R_m\}$, where Q_i is the tuple set specified in $\mathcal{A}_i$, $i = 1, \ldots, n$, and $R_1, \ldots, R_m$ are the relations owned by servers belong to party V.

The attributes of a relation R are denoted as $attr(R)$. A projection on R using a subset α of $attr(R)$ is denoted as $R[\alpha]$ or $\pi_\alpha R$. A relation R' is a *subrelation* of a relation R if $attr(R') \subseteq attr(R)$ and $R' \subseteq R[attr(R')]$ (a relation is viewed as a set of tuples). Given a set of relations $\mathbb{R} = \{R_1, \ldots, R_n\}$, the cross product of the relations in $\mathbb{R}$ is denoted by $\Delta(\mathbb{R}) = R_1 \times \cdots \times R_n$.

Definition 3.3: A given operation in a query is safe, iff for all the steps due to the operation, each step is either executed inside a server, or it sends a relation, R, to a server belonging to party V and R is a subrelation of $\Delta(\Theta(V))$. A given query plan, which consists of a sequence of operations, is safe iff all its operations are safe.

Definition 3.3 is worth more explanation. Any data transmission due to a safe query plan should only expose information that the receiver is authorized to view. However, the information carried in an intermediate result may be more than it appears. For example the query plan $\pi_{attr(R_1)}(R_1 \bowtie R_2)$ executes the join between R_1 and R_2, then applies a projection preserving only R_1's attributes. Although the result only contains a subset of tuples in R_1, it contains the information that R_2 has at least the tuples with the attributes satisfying the join conditions. We assume the query intention is protected in a way that, for any given query plan, each involved server gets only the slice of the plan that guides the server to collaborate the query evaluation; therefore, without exposing


n_0 $\bowtie_{D.district = C.district}$
n_1 $\bowtie_{A.district = B.district}$ (D)
n_3 S2: σ service= Disinfection (B)
n_4 S3: σ function = Satellite (C)
n_2 S1: Apply authorization (A)
n_5 S2: Apply authorization
n_6 S3: Apply authorization
n_7 S1: Safehouse
n_8 S2: Service
n_9 S3: Communication
n_0 S5: $\bowtie_{D.district = C.district}$ Peer-join
n_1 S5: $\bowtie_{A.district = B.district}$ Split-join (master = S2)
n_3 S2: σ service= Disinfection (B)
n_4 S3: σ function = Satellite (C)
n_2 S1: Apply authorization (A)
n_5 S2: Apply authorization
n_6 S3: Apply authorization
n_7 S1: Safehouse
n_8 S2: Service
n_9 S3: Communication

(a) A preliminary query tree of Query 1

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

Fig. 3. Query trees.

the corresponding query intention, a set of tuples received from a remote server does not leak more information than the tuples themselves. In this example, the server that receives the result of $\pi_{attr(R_1)}(R_1 \bowtie R_2)$ cannot distinguish whether they were the result of "projection after join", "select in R_1" or other queries. Therefore, we consider a data transmission operation to be safe if the transmitted relation is a subrelation of the cross product of the relations the receiver is authorized to view.

IV. Join Methods

While a query operation with a unitary operator, e.g., selection and projection, is executed inside a server and is thus safe by definition, a join operation may involve inter-party data transmission. Hence, our interest is to look at and compare different join methods to determine a safe and efficient execution strategy. In this section, we first describe two straightforward join methods, named *peer-join* and *broker-join*,[3] then present a novel join method, named *split-join*. In all the three join methods, we explore the possibilities of adopting buddy servers to achieve efficient join evaluation. As communication cost is a most important performance metric in a distributed system, our optimization target is the communication cost during the query evaluation. The size of a relation is denoted as $|R|$ and, if we ignore the cost of initiating a data transmission, the cost of sending R from S_i to S_j is $p_{i,j} * |R|$. The cost of a query plan is thus the sum of the costs of all its operations.

We consider the equi-join operation contained in n_1 of Figure 3, which joins the two tables A and B represented by the child nodes n_2 and n_3, respectively. S_q is the query issuer, while A and B reside in S_1 and S_2. respectively. We consider various party relationships of the servers (instead of sticking to the party memberships 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.

A. Peer-join and Broker-join

If S_1 and S_2 are buddies, a standard *semi-join* can be applied [4]. One viable sequence of the steps as shown in Figure 4 (a) is (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 computes $A \bowtie (A[district] \bowtie B)$ to get the final join result. The consolidator here is S_1. Note how steps (1) and (2) help determine tuples in B needed for the join, such that the resultant data transmission can be dramatically reduced compared to simply sending B to S_1. Since semi-join has been extensively researched in the area of distributed query processing, we refer the readers to the literature [4]. We turn to consider the conditions where S_1 and S_2 do not belong to the same party.

If S_2 and S_q are buddies while S_1 and S_q are not, a straightforward join method, named *peer-join*, is to send A to S_2, which computes $A \bowtie B$ as the consolidator. It leverages the fact that buddy servers (S_2 and S_q) share the same authorized view, so the data transmission is safe. Figure 4 (b) shows an application of the peer-join.[4]

Another join method, named *broker-join*, involves the steps that S_1 and S_2 send A and B to the S_q, respectively, which then completes the join computation. Here, the consolidator is S_q. It can be generalized by allowing any buddy server of S_q to be the consolidator, such that we can exploit buddy servers of S_q that may scatter in the network and choose one near S_1 and S_2 as the consolidator to reduce the data transmission cost. For example, assume S_b is a buddy of S_q close to S_1 and S_2, the broker-join is executed as shown in Figure 4 (c). Compared to sending the tables to the query issuer S_q, it computes the result as close as to the data sources, which is a principal optimization in distributed computation. This method can be applied under any party relationship condition.

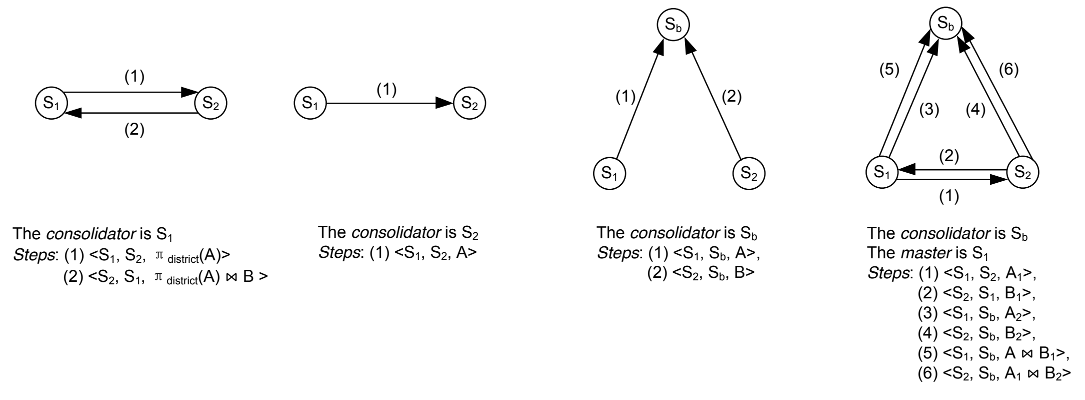
When both peer-join and broker-join are applicable, no method is better than the other in general conditions.

B. Split-join

We propose a new join method, named *split-join*, which can be applied when S_1, S_2 and S_q belong to three different parties. It explores the commonly accessible tuple sets to do partial computation near the data sources, and do the rest 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, while the remaining tuples

[3]We describe these join methods briefly for the sake of completeness; readers with the background of distributed query processing can jump to Section IV-B.

[4]Similarly, if S_1 and S_q are buddies while S_2 and S_q are not, S_2 sends B to S_1 (the consolidator) to apply the peer-join.



The *consolidator* is S_1
Steps: (1) <S_1, S_2, $\pi_{district}(A)$>
(2) <S_2, S_1, $\pi_{district}(A) \bowtie B$ >

The *consolidator* is S_2
Steps: (1) <S_1, S_2, A>

The *consolidator* is S_b
Steps: (1) <S_1, S_b, A>,
(2) <S_2, S_b, B>

The *consolidator* is S_b
The *master* is S_1
Steps: (1) <S_1, S_2, A_1>,
(2) <S_2, S_1, B_1>,
(3) <S_1, S_b, A_2>,
(4) <S_2, S_b, B_2>,
(5) <S_1, S_b, $A \bowtie B_1$>,
(6) <S_2, S_b, $A_1 \bowtie B_2$>

(a) Semi-join (b) Peer-join (c) Broker-join (d) Split-join

Only the data transmission steps are shown. Each step follows the form: *<sender, receiver, data>*.

Fig. 4. Join methods.

are in the other one. Specifically, we assume server S_1 is allowed to access tuple set B_1 of B, and $B_2 = B - B_1$(, i.e. $B_2 = \{x|x \in B \wedge x \notin B_1\}$); server S_2 is allowed to access tuple set A_1 of A, and $A_2 = A - A_1$, thus B_1 (A_1 resp.) can be safely sent to S_1 (S_2 resp.) So the join can be rewritten as

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

As in broker-join, a buddy$^+$ of S_q can be chosen to participate in the computation on behalf of S_q. Assume, for example, S_b, a buddy of S_q, is chosen. As shown in Figure 4 (d), a split-join is actually composed of two peer-joins (steps (1) and (2)) and one broker join (steps (3) and (4)). Each of the three joins is called a *sub-join*. The three sub-join results are unioned at S_b (the consolidator) to get the join result (steps (5) and (6)).

To illustrate the potential cost benefit of the split-join method, we compare the cost of broker-join and split-join methods.

$$\begin{aligned} cost(A \bowtie_{broker-join} B) =& p_{1,b} * |A| + p_{2,b} * |B| \\ cost(A \bowtie_{split-join} B) =& p_{1,2} * (|A1| + |A_2|) + \\ & p_{1,b} * (|A_2| + |A \bowtie B1|) + \\ & p_{2,b} * (|B_2| + |A_1 \bowtie B_2|) \end{aligned}$$

Both the path costs and relation sizes may affect the comparison result. Nevertheless, it can be shown that in some specific conditions, one method should be preferred over the other. For example, if $p_{1,2} << p_{1,b} = p_{2,b}$, $|A \bowtie B_1| < |B_1|$, and $|A_1 \bowtie B_2| < |A_1|$, then

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

That is, in this situation, the split-join method is cheaper. Similarly, we can prove that if $|A \bowtie B_1| \geq |B_1|$, and $|A_1 \bowtie B_2| \geq |A_1|$, the broker-join leads to less cost.

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 another parameter, *master*: when the master is S_1 (S_2, resp.), split 1 (2, resp.) is adopted. For example, the split-join shown in Figure 4 (d) adopts split 1 (and the master is S_1). We have proved that If $p_{1,b} < p_{2,b}$, the communication cost of split 1 is less (S_1 should be chosen as the master); otherwise, S_2 should be the master. The proof is omitted.

The four join methods can be represented in a uniform form: *join-method*$\{$*the consolidator, the master*$\}$, where the master is only applicable to the split-join. For example, the four join methods shown in Figure 4 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. Based on the definition of the four join methods, it is easy to see that in each step a server only receives data it is authorized to see; therefore, the join methods are safe.

V. Query Plan Generation

We present an algorithm that, given a set of pairwise authorizations and a preliminary query tree, generates a complete query tree consistent with the authorizations. Different join methods are considered and all viable strategies are compared to determine the final query tree. Since selections, projections and operations filtering tuples according to authorizations are executed inside servers, we ignore them in the algorithm and focus on joins.

Listing 1. Structures for query plan generation

```
struct Node{
    string server, relation; // Only used for leaf nodes
    string Operator; // Selection, projection, and join etc.
    Node *left, *right; // Pointers to child nodes
    Candidate *cand_list, * final_cand;
};

struct Candidate{
    Node * node; // The tree node that this candidate belongs to
    Candidate *lcand, *rcand; // Child candidates in
          consideration
    int method; // One of the four join methods
    string consolidator, master; // Parameters for the join method
    int cost; // Cost of executing the sub-tree rooted at this node
14 };
```

We first introduce the main structures *Node* and *Candidate* in Listing 1. The *Node* structure is used to represent the nodes in the query tree, while each *Candidate* instance is a join strategy, comprising the join method and the parameters, for generating the relation represented by the containing node. The *cand_list* field in the *Node* structure contain all candidates for the node. Other fields are explained in the comments.

Listing 2. Algorithm for query plan generation

```
15 // Input: Query issuer Sq, overlay network G(N, E), preliminary
       query tree T, and database profile;
16 // Output: A complete query plan and the estimated cost;
17 GeneratePlan(){
18     InitializeLeafNodes(root); // Argument is the root node
19     SearchCandidate(root);
20     cand = the Candidate in root−>cand_list with the least cost;
21     TraceBack(cand);
22 }
23
24 SearchCandidates(node){
25     if(node is leaf) return;
26     l = node−>left; r = node−>right;
27     if( l != null)  SearchCandidate(l);
28     if( r != null)  SearchCandidate(r);
29     for(each candidate lcand in l−>cand_list)
30         for(each_candidate rcand in r−>cand_list)
31             ExamineJoinOptions(node, lcand, rcand);
32 }
33
34 AddCand(node, lcand, rcand, method, consolidator, master){
35     cost = lcand−>cost + rcand−>cost + current join cost;
36     insert Candidate(node, lcand, rcand, method, consolidator,
           master, cost) into node−>cand_list;
37 }
38
39 ExamineJoinOptions(node, lcand, rcand){
40     lserver = lcand−>consolidator;
41     rserver = rcand−>consolidator;
42
43     // Intra−server join
44     if(lserver = rserver){
45         Choose an intra−server join method, e.g., sort−merge;
46         AddCand(node, lcand, rcand, sort−merge, consolidator,
               null);
47         return;
48     }
49     // semi−join. Recall that P(S) returns the party S belongs to.
50     if(P(lserver) = P(rserver)){
51         Determine the consolidator; // The details are omitted.
52         AddCand(node, lcand, rcand, semi−join, consolidator, null
               );
53         return; // Assume semi−join is superior to other joins.
54     }
55     // broker−join
56     for (each server S in N)
57         if(S != lserver and S != rserver)
58             if(P(S) = P(Sq))
59                 AddCand(node, lcand, rcand, broker−join, S, null);
60     // peer−join
61     if(P(lserver) = P(Sq))
62         AddCand(node, lcand, rcand, peer−join, lserver, null);
63     else if(P(rserver) = P(Sq))
64         AddCand(node, lcand, rcand, peer−join, rserver, null);
65     // split−join
66     else
67         for(each server S in N)
68             if(P(S) = P(Sq))
69                 if(path_cost(lserver, S) < path_cost(rserver, S))
70                     AddCand(node, lcand, rcand, broker−join, S,
                           lserver);
71                 else
72                     AddCand(node, lcand, rcand, broker−join, S,
                           rserver);
73 }
74
75 TraceBack(cand){
76     if(!cand) return;
77     cand−>node−>cand_final = cand;
78     TraceBack(cand−>lcand);
79     TraceBack(cand−>rcand);
80 }
```

The algorithm in Listing 2 takes the preliminary query tree along with the overlay network topology, authorization policies and the database profile as the input, and generates a complete query tree compliant with the authorizations. The generation of a query plan consists of three phases. (1) Initialize the query tree (Line 18; the code for *InitializeLeafNodes* is omitted): add a Candidate instance for each leaf node (*node* = the node, *lcand* = *rcand* = *method* = null, *consolidator* = the server storing the base table represented by the node, *master* = null, *cost* = zero). (2) Accumulate query strategies (Line 19): *SearchCandidates* walks the tree in a post-order and collects feasible join options into *cand_list*. The servers containing the relations to be joined are critical when considering which join methods are applicable. So it enumerates the combination of the candidates in the child nodes to collect join candidates for current node (Line 29–31). (3) Trace back from the candidate stored at the root node with the minimum cost, which represents the total cost of the plan, to determine the join strategy for each other node (Line 21).

Example 5.1: A preliminary query tree is shown in Figure 3 (a); the topology and the party membership are depicted in Figure 1. $A \bowtie B$ 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, its right child node n_4 contains only one candidate, while each of the candidates in the left child node n_1 is enumerated. For example, when the candidate *split-join*$\{S_5,$ $S_2\}$ of n_1 is enumerated, the peer-join that sends C to S_5 and the broker-join that sends both $A \bowtie B$ and C to S_q are inserted into the candidate list of n_0. By comparing the total costs of the candidate query plans, the most efficient one is chosen. Figure 3 (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.

VI. Correctness, Safety and Authorization Confidentiality

Definition 6.1: Given a preliminary query tree T, its *reference* complete query tree is generated by transforming the nodes of T: (1) for each leaf node n in T, a data transmission operation sending the relation represented by n to the query issuer is added to n; (2) for each non leaf node with a unitary operator, it remains the same except for the operation is executed in the query issuer; (3) for each non leaf node with a join operator, an intra-server join method (e.g., sort-merge) is adopted.

The query plan represented by a reference complete query tree sends all the base relations involved in the query to the query issuer, which then performs a local query computation. Note that we do not consider the safety issue due to the data transmission in the reference tree.

Definition 6.2: Let T be a preliminary query tree. T specifies a set $\mathbb{R}$ of base tables in its leaf nodes. A complete query tree of T is correct if, given any tuple instances contained in $\mathbb{R}$, it always computes the same query result as that is computed by 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 the algorithm (Listing 2) is correct and safe.

Definition 6.3: Given a query operation, if each server involved in the execution only needs the content of the authorizations defined by itself or its buddies, the operation satisfies *authorization confidentiality*. A query plan satisfies authorization confidentiality, if all its operations satisfy authorization confidentiality.

Theorem 6.2: Given a preliminary query tree, a set of authorizations, the complete query tree T generated by the algorithm (Listing 2) satisfies authorization confidentiality.

The proof of Theorem 6.1 and 6.2 is omitted due to space limitations.

VII. Experiments

We evaluated the performance of the query planning algorithm described in Listing 2, and particularly measured how buddy servers and split-joins can be exploited to save the communication cost.

We implemented the algorithm in Python and simulated the topology shown in Figure 1. We considered the query, $(A \bowtie B) \bowtie C$, issued by S_q, described in the end of Section II. Specifically, A, B and C are stored in S_1, S_2 and S_3, respectively, and all their tuples are accessibly by S_q. We assumed $|A| = |B| = |C|$. We measured the communication costs of the query plans generated by the algorithm under four setting cases (Table I). We ran the experiments 100 times for each case and each of the five different values of table sizes varying from 10 to 100k, and the average communication costs are reported. The size of each join result in each running was a random value uniformly distributed between zero and the average size of the two tables involved in the join operation. That is, $|T_1 \bowtie T_2|$ follows $\mu(0, (|T_1| + |T_2|)/2)$. The results are shown in Table II.

TABLE I
Settings cases.

Case	Setting
Case 1	Centralized evaluation.
Case 2	S_q, S_5 are buddies; foreigners do not share the views.
Case 3	S_q, S_5 are not buddiess; foreigners share part of the views.
Case 4	S_q, S_5 are buddies; foreigners share part of the views.

Case 1 enforces a centralized evaluation where A, B and C are sent to S_q to complete the query computation. The centralized query evaluation is widely used in federated databases [5], [12], [15], [18], [19], thus we regard its communication costs as the baseline.

Case 2 is to exploit buddy servers. As servers of different parties do not share the authorized views, [5] a broker-join is preferred for the first join. The algorithm generates the query plan $(A \bowtie_{broker-join\{S_5,null\}} B) \bowtie_{peer-join\{S_5,null\}}$. That is, for the first join, S_1 and S_2 send the tables to S_5, as opposed to S_q in Case1, to enforce a broker-join. In the second join, a peer-join where S_3 sends C to S_5 is applied. In both joins, the operations are executed near the data sources, largely saving the communication cost (42% on average).

Case 3 is to demonstrate the advantage of the split-join method. Each server belongs to a different party. However, the authorized views of different parties overlap. In each running the size of the shared view is a random value between zero and the table size, such that the split-join method as well as other join methods is considered when generating the query plan. Note that the consolidator in the split-join has to be S_q. It saves the communication cost by 39% on average.

Case 4 illustrates the combined benefits of exploiting buddy servers and the split-join method. S_5 and S_q are buddies and the size of the shared view is again a random value uniformly distributed between zero and the table size for each running. For the first join no matter the broker-join or the split-join is applied, the consolidator is S_5. The second-join applies the peer-join and sends C to S_5. The computation has been pushed towards the data sources as much as possible. The experiment results show that the combination of buddy servers and the split-join method improves the communication cost by 60% compared to the baseline (Case 1) under various table sizes.

TABLE II
Communication costs of different scenarios.

Table size	Case 1	Case 2	Case 3	Case 4
10	2.45e3	1.43e3	1.51e3	1.1e3
100	2.45e4	1.40e4	1.58e4	9.4e3
1k	2.45e5	1.38e5	1.42e5	9.6e4
10k	2.45e6	1.42e6	1.43e6	9.9e5
100k	2.45e7	1.41e7	1.47e7	9.3e6

VIII. Related Work

In conventional distributed database systems, it is common to grant access by defining different authorized views for different users or roles [1], [6], [14], [16], [20]. In such systems, however, servers belong to a single organization. Thus they do not consider inter-party information flows.

Safe query execution for federated database systems has been extensively researched. Most of them define authorizations in terms of roles and perform centralized query evaluation [5], [12], [15], [18], [19]. Some focus on resolving inconsistencies when merging security specifications from multiple independent data sources with the goal of establishing a common security policy set for regulating the federated system [7], [11]. Authorizations defined in our work do not

[5]It implies that $|A_1|$ and $|B_1|$ in the split-join are both zero.

need conflict resolution, as each of them is defined in terms of local resources of the authorization definer. Some works [8]–[10] start to explore per-party authorization profile. However, they investigate vertical access control, specifying how each attribute in relations can be accessed directly or referred to for join connection purposes directly. Their work complements the horizontal access control presented in this paper.

Sovereign joins [2], [3] enhance privacy and confidentiality by allowing queries to be computed without revealing source data other than the query result. The algorithms usually incur a high computational and communication cost due to the goal avoiding data leakage and information inference.

IX. Conclusions

In multiple-party collaborations, information leakage during sharing is a realistic concern. How to share information safely and flexibly is a common and important problem in many coalition scenarios. We formalized the problem in federated databases and presented a simple but effective solution to enforcing horizontal access control. It allows each party to share the information selectively and discriminatively, as a party can define an arbitrary set of accessible tuples per peer party independently. We have shown that our algorithm generates correct and safe query plans, protecting the confidentiality of both information and access control policies. The initial experiments show the advantage of the algorithm in largely saving communication cost. More experiments will be performed to further evaluate it. In short, the solution provides an effective access control measure for safe information sharing in federated databases with high communication efficiency, which is a critical criterion for wireless mobile and wide-area networks.

Acknowledgment

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] M. E. Adiba. Derived relations: a unified mechanism for views, snapshots, and distributed data. In *VLDB '81*, pages 293–305.

[2] R. Agrawal, D. Asonov, and M. K. Li. Sovereign joins. In *ICDE '06*.

[3] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *SIGMOD '03*, pages 86–97.

[4] 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 Trans. Database Syst.*, 6:602–625, December 1981.

[5] J. B. Bocca, M. Jarke, and C. Zaniolo. An approach for building secure database federations. In *VLDB '94*, pages 24–35.

[6] S. Castro, M. Fugini, and P. Samarati. *Database security*. 1995.

[7] S. Dawson, S. Qian, and P. Samarati. Providing security and interoperation of heterogeneoussystems. *Distrib. Parallel Databases*, 8:119–145, 2000.

[8] S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Assessing query privileges via safe and efficient permission composition. In *CCS '08*, pages 311–322.

[9] S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Controlled information sharing in collaborative distributed query processing. In *ICDCS '08*, pages 303–310.

[10] 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.

[11] S. D. C. di Vimercati and P. Samarati. Authorization specification and enforcement in federated database systems. *Journal of Computer Security*, 5(2):155–188, 1997.

[12] D. Heimbigner and D. Mcleod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3:253–278, 1985.

[13] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization. In *SIGMOD '91*, pages 168–177.

[14] S. Jajodia and R. Sandhu. Toward a multilevel secure relational data model. *SIGMOD Rec.*, 20:50–59, 1991.

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

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

[17] 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 *SIGMOD '79*, pages 23–34.

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

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