Cooperative Data Access in Multi-cloud Environments *

Meixing Le, Krishna Kant, Sushil Jajodia
{*mlep, kkant, jajodia*}*@gmu.edu*

Center for Secure Information Systems,
George Mason University, Fairfax, VA 22030

Abstract. In this paper, we discuss the problem of enabling cooperative query execution in a multi-cloud environment where the data is owned and managed by multiple enterprises. We assume that each enterprise defines a set of *allow rules* to facilitate access to its data, which is assumed to be stored as relational tables. We propose an efficient algorithm using join properties to decide whether a given query will be allowed. We also allow enterprises to explicitly forbid access to certain data via *deny rules* and propose an efficient algorithm to check for conflicts between allow and deny rules.
Keywords: Cloud; Rule Composition; Join Path

1 Introduction

With increasing popularity of virtualization, enterprises are deploying clouds to flexibly support the IT needs of their internal business units or departments while providing a degree of isolation between them. Enterprises may need to collaborate with one another in order to run their businesses. For example, an insurance company needs information from a hospital, and vice versa. Clouds remove the physical boundaries of enterprise data so that several enterprises can share the same underlying physical infrastructure. Physical location of the data is important when planning an optimal query plan with data cooperation among enterprises. However, in this work, it suffices to assume that each enterprise has access to a logically separate cloud. We assume that all data is stored in relational databases and accessed via relational queries. The enterprises disclose some information to others based on their collaboration requirements, but would like to avoid leakage of other information.

Similar data sharing scenarios arise in other contexts as well, including those between independently owned data centers and between the enterprise clouds and the underlying physical infrastructure. Figure 1 shows the latter situation more clearly where the enterprise clouds A and B run on top of the physical

* This material is based upon work supported by the National Science Foundation under grants CCF-1037987 and CT-20013A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

infrastructure I. Similarly, enterprise cloud C runs on top of a different physical infrastructure II. In this case, the enterprise clouds need to know suitable configuration information from the infrastructure providers and the providers may need to know the characteristics of the software deployed by the enterprise clouds. Given the standard CIM [10] (common information model) based storage of configuration data, it may even be necessary to consider access to information that is obtained by combining the stored data in some way (much like joins in normal databases). Thus, the collaboration requirements among these entities can be similar to those in the context of multiple enterprises sharing data.

If in Figure 1, enterprise A needs data from enterprise B to satisfy its business needs, A and B need to negotiate and establish policies regarding the accessibility of each other's data. This results in authorization rules for A and B to follow. With these authorization rules, A is able to answer some queries that require information from B but not others. If A also has authorization rules for cloud C, then A may be able to answer a query that requires some data from both B and C. In the first part of our work, we want to decide whether a given query against enterprise A is allowed according to all the authorization rules given to A.

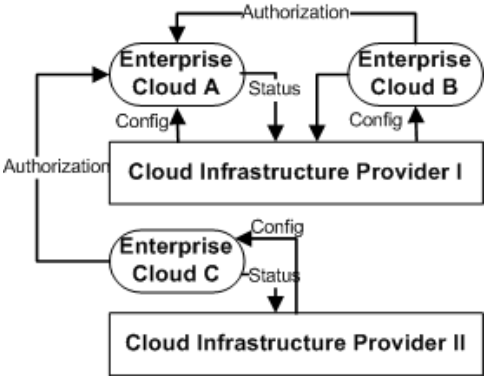

Fig. 1. Cooperative data access in cloud environment.

In general, there are two ways of specifying the authorizations: explicit (as in reference [1]), and implicit. The explicit method is easier in that any queries that do not match any explicit authorization rule will not be answered. However, the number of rules could become large and cumbersome to manage. In the implicit approach, the enterprises are only given some basic rules, and are free to compose them and thereby access more information than the rules imply directly. The implicit method can be more concise, and is the focus of this paper. The main problem with implicit method is that there is no way to exclude certain compositions. We fill this gap by introducing deny policies as well.

Deny policies are needed for two reasons. The first reason is simply to avoid certain combinations and thereby achieve the same level of expressiveness as the explicit authorizations. The second reason is that an enterprise may be able to do compositions locally after having obtained the desired data from other enterprises, but such compositions may not be intended.

In this paper, we also present an algorithm to verify whether a deny rule will be violated by the authorization rules. In other words, we check the conflict between allow rules and deny rules. In some cases, the deny rules may still be

difficult to enforce with existing parties. In such case, the conflicts found by our algorithm can be used to alert the data owners to change their authorizations or policies in order to remove the conflicts. On the other hand, if it is possible to implement deny rules using a third-party, then they should be given higher priority over the allow rules. Of course, if a deny rule does not conflict with the allow rules, it has no effect and can be ignored. Thus our consistency checking algorithm can be used to reduce the number of deny rules.

The main purpose of this paper is to come up with efficient algorithms for query permission and conflict checking. This paper does not address the next step of actually formulating a query plan as well as the problem of implementing and enforcing all the rules, which will be explored in a subsequent paper.

The outline of the rest of the paper is as follows. Section 2 discusses the related work. Section 3 presents the concepts related to join group and composable rules, and the intuition behind our approach. For checking whether a query originating from a cloud can be authorized, we propose a new two-step algorithm which first selects all the related given rules, and then tries to compose these rules to determine authorization of the query. This is discussed in Section 4. In section 5, we present an algorithm for checking whether the deny rules are consistent with the given authorization rules. Finally, in section 6 we conclude the discussion and outline future work.

2 Related Work

De Capitani di Vimercati, Foresti and Jajodia [1] studied the problem of authorization enforcement for data release among collaborating data owners in a distributed computation so as to make sure the query processing discloses only data that has been explicitly authorized. They proposed an efficient and expressive form of authorization rules which define on the join path of relations and they also devise an algorithm to check if a query with given query plan tree can be authorized using the explicit authorization rules. In our work, we follow the format of authorization rules they proposed. However, it is possible that these explicit authorization rules given to the same enterprise can be composed together to implicitly allow more information to be released through queries.

In another work [2], the same authors evaluate whether the information release the query entails is allowed by all the authorization rules given to a particular user, which is similar to the problem of query permission checking in our work. Their solution uses the graph model to find all the possible compositions of the given rules, and checks the query against all the given allow rules. In our work, the rules are given to different clouds instead of users, and we propose a more efficient algorithm to filter more unrelated rules first. Moreover, we deal with deny policies also.

Processing distributed queries under protection requirements has been studied in [9, 12, 14]. In these works, each relation/view is constrained by an access pattern, and their goals are to identify the classes of queries that a given set of access patterns can support. These works with access patterns only considers

two subjects, the owner of the data and a single user accessing it, whereas the authorization model considered in this work involves independent parties who may cooperate in the execution of a query. There are also classical works on the query processing in centralized and distributed systems [8, 13, 5], but they do not deal with constraints from the data owners. Superficially, the problem of checking queries against allow and deny rules is similar to checking packets against firewall allow and deny rules [6]. However, firewall rules are usually explicit, and one rule can contain another rule but not compose with another rule.

There are several services such as Sovereign joins [11] to enforce the authorization rule model we used, such a service gets encrypted relations from the participating data providers, and sends the encrypted results to the recipients. Also, there are some research works [3, 4, 7] about how to secure the data for out-sourced database services. These methods are also useful for enforcing the authorization rules in our work, and their primary purpose is to provide mechanisms for information sharing among untrusted parties.

3 Composing Rules for Query Checking

In order to check if a query is admissible according to the authorization rules, one naive idea is to generate all the possible compositions of the given basic rules, so as to convert each implicit rule into explicit one, and then check the query permission. The problem is that the compositions may generate too many rules, which make the approach very expensive.

Instead of generating all possible compositions, we organize the rules based on join attributes, and then use a two-step algorithm to check whether a given query can be authorized. In the first step, we filter as many rules as possible according to the given query. In the second step, we compose these rules based on their join attributes.

In this section we build up the machinery to enable this checking. In order to illustrate the various concept and models, we start with an e-commerce example that we will use throughout the paper. The example has the following schema:

1. Order (order_id, customer_id, item, quantity) as O
2. Customer (customer_id, name, creditcard_no, address) as C
3. Inventory (item, retail_price, date) as I
4. Warehouse (location, item, supplier_id, stock) as W
5. Supplier (supplier_id, supplier_name, cost_price) as S
6. Shipping (location, customer_id, days, ship_cost) as Sp

The underlined attributes indicate the primary keys of the relations. We assume that relations *Order* and *Customer* are stored at Cloud A, and other relations are on the other clouds. The authorization rules for Cloud A are given below. The first two rules define access to local relations, and the following rules define remote access cooperated with other clouds. Each authorization rule has an attribute set, and is defined on one relation or a join path; the rule is also applied to a specified cloud.

1. (order_id,customer_id,item,quantity),(Order) → Cloud A
2. (customer_id,name,creditcard_no,address),(Customer) → Cloud A
3. (item,supplier_id,supplier_name), (Warehouse, Supplier) → Cloud A
4. (item,order_id,retail_price), (Order, Inventory) → Cloud A
5. (location,supplier_id,retail_price,stock), (Warehouse, Inventory) → Cloud A
6. (location,item,customer_id,ship_cost), (Shipping, Warehouse) → Cloud A
7. (ship_cost,stock, cost_price), (Shipping, Warehouse, Supplier) → Cloud A

For simplicity, we assume identical attributes in different relations have the same name, and queries are in simple *Select-From-Where* form. In addition, relations satisfy the Boyce-Codd Normal Form (BCNF), and possible joins among the relations are all lossless joins. Also, we assume there is no collusion between clouds to bypass access limitations.

To illustrate query authorization, we shall consider two specific queries:

1. ***Select*** name, address, ship_cost, retail_price
   From Customer ***as*** C, Shipping ***as*** Sp, Warehouse ***as*** W, Inventory ***as*** I
   Where C.customer_id = Sp.customer_id ***and*** Sp.location = W.location ***and*** W.item = I.item

2. ***Select*** supplier_name, stock
   From Supplier ***as*** S, Warehouse ***as*** W, Inventory ***as*** I
   Where S.supplier_id = W.supplier_id ***and*** W.item = I.item
   and cost_price>'100'

3.1 Basic Concepts

In order to perform efficient authorization checking, we group relations according to their join capability. For this we define a ***Join Group*** as a set of relations that share the same set of attributes and any subset of them can be joined based on that attribute set. A relation can appear in several Join Groups. A Join Group is identified by the set of attributes that its relations can join over, and we call this as ***Joinable Attribute Set (JAS)*** for the group. In our example, relations $Shipping, Warehouse$ are in the same Join Group, and attribute set $\{location\}$ is the JAS of this group. Other JASes among these relations are: $customer\_id, supplier\_id, item$. In order to address information release by joining two or more relations, we define the notation of Join Path.

Definition 1 ***(Join Path)*** *Given a set of relations $T_1, T_2 ... T_n$, a Join Path $< T_1, T_2 ... T_n >$ is an ordered chain of these relations, where each pair of relations $< T_i, T_{i+1} >$ are joined with each other on the JAS.*

Each query itself has an associated Join Path called ***Query Join Path***. In contrast, join path associated with a rule is called ***Rule Join Path***. For instance, the Query Join Path of Query 1 is $< C, Sp, W, I >$, and Rule 7 is defined on the join path $< Sp, W, S >$. For each rule, the Join Path defines a

view, and the attribute set further refines the view. Therefore, a rule for a cloud defines a view that this cloud can access. Similar to relations, views (rules) can also be joined together. By joining two views, the resulting view is defined over a longer join path. Next, we define the concept of Sub-Path relationships between two join paths, which is useful for determining the relevant rules for checking the authorization.

Definition 2 ***(Sub-Path Relationship)*** *A Join Path A is a Sub-Path of another Join Path B if: 1) The set of relations in Join Path A is a subset of the relation set of Join Path B. 2) For each join pair* $< T_i, T_{i+1} >$ *joins on a JAS henceforth denoted as* JAS_i*, and* $< T_i, T_{i+1} >$ *also appears in Join Path B and joined on* JAS_i*.*

Given two join paths A and B, whether A is a Sub-Path of B cannot be determined by a simple linear matching of Join Paths. It is because the order of the relations may be interchanged in a join path, and JASes in the join path also need to be compared.

3.2 Graph Model to Determine Sub-Path Relationship

Here, we use a graph model to determine the Sub-Path relationships. We present Join Path via a labeled graph. $G =< V, E >$, where each node $v \in V$ represents a relation in the Join Path, and each labeled edge $e \in E$ connects two nodes if the two relations form a join pair in the Join Path, and the label indicates the JAS. The graph model applies to both queries and the authorization rules. To determine whether an authorization rule is defined on a Sub-Path of a query is equivalent to checking whether graph $G(r)$ of a rule r is a sub graph of query graph $G(q)$.

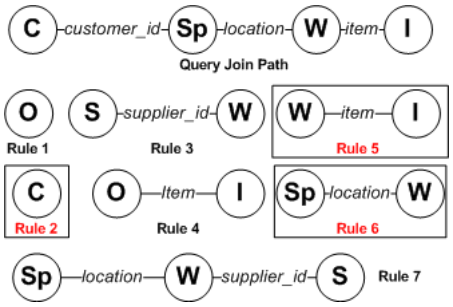

Fig. 2. Rules defined on the Sub-Paths of Query Join Path of Query 1 in example.

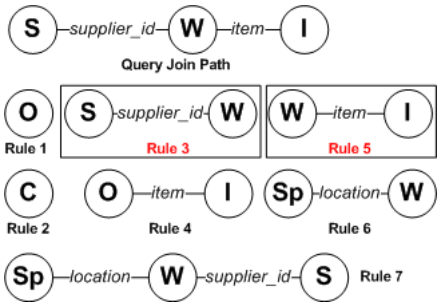

Fig. 3. Rules defined on the Sub-Paths of Query Join Path of Query 2 in example.

Figure 2 shows the query 1 in our example, and the rules in the boxes are the ones defined on the Sub-Path of the Query Join Path. Figure 3 does the same

for query 2. For query 1, rules 2, 5, 6 are defined on the Sub-Paths of the Query Join Path. For query 2, the rules are 3 and 5.

Determining Sub-Path relationship is an important step to figure out the composable rules as we shall show later in Theorem 1. However, a rule defined on a Sub-Path of the Query Join Path is not necessarily a composable rule of the query. Hence, we also look at the attributes that can be used to compose rules in the given query. We call the JAS in a Query Join Path as ***Query JAS***. Each Query JAS is also associated with the relation pair that join over it. For example, in query 2, the Query JASes are: $supplier\_id(S, W), item(W, I)$. As rules can be composed using join operations, we define the concept of *Composable rule* below.

Definition 3 ***(Composable rule)*** *An authorization rule is a composable rule, if the attribute set of the rule contains at least one JAS.*

According to the definition, only Rule 7 in our example is not a composable rule because its attribute set does not contain any of the four JASes. Similarly, with a given query, we define ***Query Composable rule*** as an authorization rule whose Join Path is defined on a Sub-Path of the Query Join Path and attribute set contains one Query JAS. For illustration, Rules 2, 5, 6 are the Query composable rules for query 1 since Rule 2 contains *customer_id* and Rule 5 and 6 contain *location*. As join operations can occur in rules, the concept of Join Groups can also be applied to rules instead of basic relations. Within each Join Group of rules, there are the rules whose attribute sets contain a common JAS.

Definition 4 ***(Join Group List)*** *Each entry in a Join Group List is a Join Group of composable rules. There is a unique JAS to identify each entry and within the entry there are composable rules whose attribute sets contain this JAS.*

It is clear that one rule may appear in multiple entries. The Join Group List can be generated with the given rules, and an example of Join Group List can be found in section 5.3. ***Query Join Group List*** is a Join Group List based on the given query. For each entry in such list, it is identified by a Query JAS, and within each entry are the Query composable rules whose attribute set contains this Query JAS. Only rules in the Query Join Group List are the relevant rules that will be considered in the composition step. In section 4.2 we show the Query Join Group Lists of queries in our example.

3.3 Rule Composition Rationale

Our mechanism first checks if a single rule can authorize the query. If not, we compose the relevant rules to see whether the given query can be authorized. All the rules within the same entry of the Query Join Group List can be composed together since they are all composable on that Query JAS. Therefore, rules within one entry can be composed into one single composed rule with longer join path and larger attribute set. If one rule appears in two or more entries of

the list, it indicates that this rule can be used to connect these Join Groups so that the composed rules from these entries can be further composed.

Such a composition is also transitive. If a rule r_a appears in entries of JAS_1 and JAS_2 and a rule r_b appears in entries of JAS_2 and JAS_3, then all the rules within these 3 entries can be composed into one rule. It is because r_a and r_b share JAS_2, these two rules can be composed by joining on JAS_2, and their connected entries can be further composed. Therefore, we group the entries in the Query Join Group List based on their connectivity. All the rules within a connected entry group can be composed into one rule. This procedure produces one or more maximally composed rules such that no further composition is possible. If there is more than one such composed rule, at most one of them can be defined on the Query Join Path. This follows from the fact that if two composed rules are defined on the same join path, then they can be further composed together. In addition, since the Query composable rules are all defined on the Sub-Path of the Query Join Path, composition of the rules will not have a join path longer than Query Join Path. Therefore, we only need to check the composed rule which includes the greatest number of relations (longest join path). If this rule is defined on the same join path as the query join path, then we check whether the attribute set of the composed rule is a superset of the attribute set in the query. The query can be authorized if and only if this is the case.

3.4 Theorems and Proofs

In this section, we prove a number of assertions regarding the rule composition and query checking which are useful in formulating the checking algorithm and proving their correctness.

Theorem 1 *All authorization rules that are not defined on a Sub-Path of query Join Path are not useful in the rule composition.*

Proof. Assume a query q has a Join Path of $< T_1, T_2 ... T_n >$. A rule r not defined on a Sub-Path of the Query Join Path will have two possibilities by definition. 1) The Join Path of r includes at least one relation T_m which is not in the set of $\{T_1, T_2 ... T_n\}$. 2) The Join Path of r is defined on the set of relations which is a subset of $\{T_1, T_2 ... T_n\}$, but join over different JASes. The composed rule that can authorize the query must have the same Join Path as Query Join Path. Otherwise, the query results will have incorrect tuples because the underlining views are joined differently, and such a case also means the query is not authorized. Thus, if an authorization rule r has T_m in its Join Path, then any composed rule using this rule will also have T_m in its Join Path which is different from Query Join Path. For the second case, such a rule generates a different view, and any composed rule containing this rule also have a Join Path different from Query Join Path. Therefore, both types of rules need not be included to compose a rule that will authorize the query.

Theorem 2 *Only Query Composable rules are useful in the rule composition.*

Proof. A rule that is not a Query composable rule can have two possibilities: 1) it is not defined on a Sub-Path of Query Join Path. Theorem 1 indicates these rules are not useful. 2) the rule is defined on a Sub-Path of query Join Path, but the attribute set of the rule does not contain any Query JAS. To compose a rule with others to authorize the query, it must join with other rules on Query JAS. Otherwise, either it cannot compose with any other rule, or the composed rule has a join path different from Query Join Path. Therefore, only the Query composable rules should be included for rule composition step.

Theorem 3 *The composition step can cover all the possible ways to authorize the query.*

Proof. From Theorem 2, we know that any composition including non-Query Composable rules will not authorize the query. Then the composition step looks for only possible compositions among Query Composable rules. According to the connectivity among the entries, if two rules are in two disconnected entries, then they cannot be composed into one rule. On the other hand, for rules within the connected entries, we compose them into a maximally composed authorization rule. Such a rule maybe more than enough to authorize the query, but the Join Path of the rule can be at most the same as the Query Join Path. From above two observations, all the possible compositions that may authorize the query are included in these composed rules from separate connected entry groups. Finally, only one composed rule that has the same join path as the Query Join Path can authorize the rule, and there is at most one such composed rule.

4 Verifying Query Admissibility

Our two-step algorithm first builds up the Query Join Group list, and then uses composition step to construct rules that can possibly authorize the query.

4.1 Algorithm For Checking Query Permission

In the first step, the algorithm examines all the given rules and builds the Query Join Group List as discussed above. Each Query composable rule is put into the entries based on its Query JAS. If one rule appears in multiple entries, these entries are connected. Also, each entry is augmented with the relations which are accessible from the rules in this entry. At the end of this step, the algorithm maintains the connected entry group with the greatest number of relations.

In the second step, the algorithm can compose rules efficiently with Query Join Group List. The algorithm only examines all the entries within the connected group that holds the largest number of relations (can be multiple), and entries with only one rule are also ignored. The rules within each connected entry group are composed into one rule as discussed above. As the algorithm examines the groups with most relations, if these composed rules cannot authorize the query, then the query is not authorized.

We assume the complexity of the basic operation that checks whether a given rule r can authorize the query q is C, and there are N given rules, and the query q is defined on a Join Path of m relations. In the algorithm, step one has the worst case complexity of $O(N*C*m)$. It is because the complexity of Sub-Path determination is lower than that of checking the query authorization; both of them need to compare the Join Paths and attribute set. If all the rules pass the Sub-Path checks, then the algorithm compares each rule with the $m-1$ Query JASes to decide which entries to put in. None of the rest operations is more expensive than C. Similarly, in step two, at most m entries and N rules are checked, and composing the rules is not expensive than C also, thus, the complexity of step two is $O(N*C)$. Therefore, the overall complexity of the algorithm is $O(N*C*m)$. Considering the fact that most join paths in practice involves less than 4 or 5 relations, the number of m is expected to be very small in most cases. Therefore, in average cases, we can expect the complexity of the algorithm close to $O(N*C)$.

Algorithm 1 Query Permission Checking Algorithm

Require: Set of authorization rules, the query q
Ensure: Query can be authorized or not
STEP ONE:
for each authorization rule r **do**
  if r authorizes q **then**
    q is authorized
    return true
  else if Sub-Path(r, q) **then**
    for each Query JAS in q **do**
      if r is composable on this JAS **then**
        Add r into the entry of this JAS in Query Join Group List
        Connect this entry with previous entry that r also appears
        Update the relation set associated with this entry
for each unvisited entry in Query Join Group List **do**
  Follow the link to the connected entries
  Update the relation set associated with each entries in the same group
Keep the largest connected entry groups with most relations
STEP TWO:
Construct an empty rule r_c
for each largest connected groups **do**
  Begin from one entry in the group
  Follow the link to the connected entries
  Compose the rules in entry with the existing composed rule r_c
  Generate a composed rule r_c
  if r_c authorizes q **then**
    q is authorized
    return true
q is denied
return false

4.2 Illustration with The Running Example

We begin with query 1. In the first step, the algorithm examines all the rules. As no single rule is defined on the Query Join Path, none of the given rule can authorize this query. Based on the definitions, rules 1, 3, 4, 7 are not defined on the Sub-Path of the Query Join Path, so that they are not useful to authorize the query. The Query Join Group List is:

1. customer_id (C, Sp) → {Rule 2, Rule 6}.
2. location (Sp, W) → {Rule 5, Rule 6}.
3. item (W, I) → {Rule 6}.

Since Rule 6 appears in all three entries, these three entries form the only connected entry group in this list. Then in second step of the algorithm, the entry *item* is ignored since there is only one rule in the group, and Rule 6 is composed with Rule 2 by joining on Query JAS *customer_id* which further composes with Rule 5 by joining on Query JAS *location*. Thus, the composed rule is "(*customer_id*, *name*, *creditcard_no*, *item*, *address*, *retail_price*, *stock*, *ship_cost*, *location*), ($Customer$, $Shipping$, $Warehouse$, $Inventory$) → Cloud A". This composed rule is defined on the Query Join Path of query 1, and the attribute set contains all the attributes required in query 1. Therefore, the query is authorized.

Query 2 has Query Join Path $< S, W, I >$, attribute set {*supplier_name*, *stock*, *cost_price*}, and Query JASes are {*supplier_id* (S,W), *item* (W,I)}. Here, attribute *cost_price* appears in ***Where*** clause is put into the attribute set, since the query needs the authorization on that attribute to do the select operation. As no single rule can authorize the query, the algorithm builds the Query Join Group List during the first step. Rules 1, 2, 4, 6, 7 are filtered as their Join Paths are not Sub-Paths of Query Join Path. The Query Join Group List is:

1. supplier_id (S, W)→ {Rule 3, Rule 5}.
2. item (W, I) → {Rule 3}.

Then the algorithm ignores entry *item*, and composes the Rule 3, 5 by joining on Query JAS *supplier_id*. The resulting composed rule is "(*item*, *supplier_name*, *supplier_id*, *retail_price*, *stock*), ($Supplier$, $Warehouse$, $Inventory$) → Cloud A". Since attribute *cost_price* is not in the attribute set of the composed rule, this query cannot be authorized.

5 Checking Consistency with Deny Policies

In addition to the authorization rules to allow access, cloud owners usually have deny rules to make sure that certain combinations of attributes are not accessible so that the information contained in such a relationship will not be released. We want to check using all the given authorization rules whether there exists any possible authorized query that violates the deny rules. For example, we can have a deny rule as below:

1. (Inventory.item, Inventory.retail_price, Supplier.cost_price) → Cloud A

This rule means the Cloud A does not allow to get these three attributes from two tables at the same time (in one tuple), however the appearance of two of the attributes at the same time is allowed. Unlike the authorization rule, deny rules are not defined on join paths because such a rule is more restrictive than the one defined on a join path from the perspective of deny. Without join path, a deny rule prohibits any composition result that make the attribute set appear together no matter which join path is used. Since they are not defined on join paths so that they cannot be composed, and we always check them one at a time. To make sure a deny rule is not violated, all the possible join paths and rule compositions that will allow the attribute set need to be checked. To do so, one naive idea is to generate all the possible authorization rules and check if any one of them violates the given deny rules. Again, this is highly inefficient and we need a better algorithm.

5.1 Join Group List Approach

If the attributes within one deny rule are not explicitly allowed by an authorization rule, then the only possible way to violate it is the composition of the given authorization rules. We use the Join Group List to check the possible rule compositions that may violate the deny rules. Unlike query authorization considered earlier, the rule composition here is not constrained by the Query Join Path, and any composition of the rules that may violate the deny rule should be considered. Similar to the above algorithm, rules in a connected entry group of the Join Group List can be composed into one rule. Beginning with one basic rule and following all the connected entries, we can get a maximally composed rule including that basic rule.

To test whether a given deny rule is violated, we begin with the deny rule by randomly pick an attribute from the rule. We can randomly pick the first attribute because that for the attributes in a deny rule to appear together in one tuple, there must exist a composed or given rule to include all these attributes. After picking the first attribute, we choose all the basic rules that include this attribute. It is because any composed rule that violates the deny rule must be composed with at least one of such rules. We then compose the rules much like that for the query authorization one. In addition, there is no need to generate the real composed rule, as we are only concerned with the attribute set of the composed rule. This can be achieved by taking the union of the attribute set from all the connected Join Group List entries.

5.2 Deny Rule Verification Algorithm

The deny rule verification algorithm first generates the Join Group List with given rule, and then composes rules to check violation. The first step of the algorithm can be treated as a pre-computation step since once the authorization rules are given, the list can be generated. According to the definition, by

examining the authorization rules with each JAS, putting the rules in the corresponding entries, and creating the connections among the entries, the list is generated. In the second step, the algorithm goes through all the rules containing the randomly picked attribute and tries to compose maximum possible rules to violate the rule. If and only if one of such rule is found, the deny rule is violated. Algorithm 2 is the detail description of Deny Rule Verification procedure.

Algorithm 2 Deny Rule Verification Algorithm

Require: Set of authorization rules, the deny rule d, the JAS set
Ensure: Deny rule can be violated or not

```
    STEP ONE(Join Group List Generation):
 1: for each authorization rule r do
 2:   for each JAS do
 3:     if JAS ⊆ Attribute set of r then
 4:       Add r into the entry of this JAS in Join Group List
 5:       Connect this entry with previous entry that r also appears
    STEP TWO(Verification):
 6: Pick one attribute A from deny rule d
 7: Create an empty attribute set UA
 8: for each rule r includes attribute A do
 9:   if r is in Join Group List and not visited then
10:     Get the attribute set from the rules in the entry that includes r
11:     Follow the links among the entries to get all connected entries
12:     Union all the attributes from the rules in these entries, get set UA
13:     if The attribute set of deny rule d ⊆ UA then
14:       Deny Rule can be violated
15:       return true
16:   else
17:     if The attribute set of deny rule d ⊆ The attribute set of r then
18:       Deny Rule can be violated
19:       return true
20: Deny Rule cannot be violated
21: return false
```

In order to examine its complexity, suppose that there are N given rules, and there are m possible JASes among them, and the cost of checking whether an attribute is included in a set is C. Then the complexity of step one is $O(N*C*m)$. If the largest number of rules in each entry in the list is t, and basic operation cost for getting the attribute set from a rule is C, the worst complexity of step two is $O(N * C * t)$. It is because in step two, at most N rules are examined, and for each entry, at most t rules are checked. Therefore, the overall complexity depends on the number of given rules and the relationships among them. On the other hand, since such verification can be done offline with all given authorization rules and deny rules, complexity is not a big concern here.

5.3 Illustration of Deny Rule Checking

Based on the definition in section 3, the Join Group List of our running example including all the relations is:

1. (customer_id) → (Rule 1, Rule 2, Rule 6)
2. (supplier_id) → (Rule 3, Rule 5)
3. (item) → (Rule 1, Rule 3, Rule 4, Rule 6)
4. (location) → (Rule 5, Rule 6)

For the verification, the algorithm randomly picks one attribute, let us say *retail_price*. Since *retail_price* appears in Rule 4 and Rule 5, the algorithm only needs to begin with these two rules. Starting with Rule 4, the algorithm first gets the attribute set of the rules within entry *item*, and then examines the connected entry group including entry *item*. Since entries *location* and *customer_id* connect to *item* with Rule 6, and entry *supplier_id* connects entry *location* with Rule 5, all the entries in this list are connected. Therefore, Rule 5 does not need to be checked again. Figure 4 depicts how the rules 1 to 6 are composed together with JASes to obtain the attribute set. The resulting composed rule will have the attribute set which is the union of the attribute sets from rule 1 to 6. Because this set is not a superset of $\{item, retail\_price, cost\_price\}$, the deny rule cannot be violated.

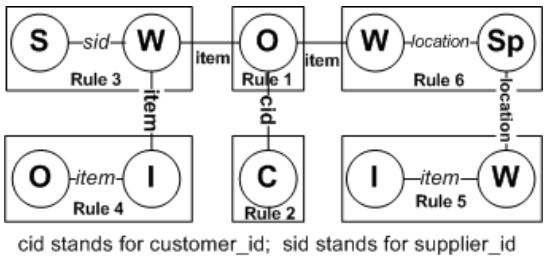

Fig. 4. Composition of the rules 1 to 6.

6 Conclusions and Future Work

In this paper, we examined the problem of cooperative data access in multi-cloud environments. Given the authorization rules for allow policies, using the join properties among the given rules, we presented an efficient algorithm to decide whether a given query can be authorized. In addition, we proposed an algorithm to check whether the given authorization rules are consistent with the deny rules that the enterprises may have specified to ensure that sensitive data is not released.

As stated earlier, we do not consider the generation of actual query plans in this paper. Generating a query plan may require the help of a trusted third-party in order to do the required join operations without violating the authorizations and deny rules. The query plan generation also involves performance considerations, which, in a multi-cloud environment would require consideration of

location of data. The implementation of authorization checks may need to be done at all the parties that contribute data to the query before the query execution can begin. The query execution itself must decide what operations are done where in order to avoid any unauthorized leakage of information.

It may be possible to formulate the query authorization problem formally with first-order logic so as to use traditional SAT based techniques; however, the feasibility and complexity of this approach remain to be investigated.

References

1. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati. Controlled Information Sharing in Collaborative Distributed Query Processing. In Proc. of ICDCS 2008. Beijing, China, Jun 2008.
2. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati. Assessing query privileges via safe and efficient permission composition. In Proc. of ACM Conference on Computer and Communications Security 2008. Alexandria, VA, U.S.A. Oct. 2008.
3. G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In Proc. of CIDR 2005. Asilomar, CA, USA, Jan 2005.
4. V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In Proc. of ESORICS 2009. Saint Malo, France, Sept. 2009.
5. D. Kossmann. The state of the art in distributed query processing. ACM CSUR, 32(4):422-469, Dec. 2000.
6. M. Gouda, A. Liu. Firewall Design: Consistency, Completeness, and Compactness. In Proc. of ICDCS 2004, Tokyo, Japan, 2004.
7. R. Sion. Query execution assurance for outsourced databases. In Proc. of VLDB 2005. Trondheim, Norway, 2005.
8. P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. J.B. Rothnie. Query processing in a system for distributed databases (SDD-1). ACM TODS, 6(4):602-625, Dec. 1981.
9. A. Cali, D. Martinenghi. Querying data under access limitations. In Proc. of ICDE 2008, Cancun, April 2008.
10. Common Information Model. http://dmtf.org/standards/cim
11. R. Agrawal, D. Asonov, M. Kantarcioglu, Y. Li. Sovereign joins. In Proc. of ICDE 2006, Atlanta, April 2006.
12. D. Florescu, A. Y. Levy, I. Manolescu, D. Suciu. Query optimization in the presence of limited access patterns. In Proc. of SIGMOD 1999, Philadelphia, PA, June 1999.
13. A. V. Aho, C. Beeri, J. D. Ullman. The theory of joins in relational databases. ACM TODS, 4(3):297-314, 1979.
14. C. Li. Computing complete answers to queries in the presence of limited access patterns. VLDB Journal, 12(3), 2003.