Access Rule Consistency in Cooperative Data Access Environment

Meixing Le, Krishna Kant, Sushil Jajodia
Center for Secure Information Systems
George Mason University, Fairfax, VA

***Abstract*—In this paper we consider the situation of a set of cooperating enterprises where each enterprise provides others controlled access to its data in order to implement rich services for the clients. In such an environment, various access rules must be mutually consistent and it should be possible efficiently check the queries for any violations of access policies. In this paper we specifically consider the handling of policy changes that may occur occasionally or routinely as a function of the overall context. We address the issues of ensuring consistency in the face of such changes. We also consider the impact of such change on query planning and execution and devise efficient mechanisms to handle dynamic changes to policies.**

I. Introduction

Providing rich services to clients with minimal manual intervention or paper documents requires the enterprises involved in the service path to collaborate and share data in an orderly manner. For instance, an automated determination of patient coverage and costs requires that a hospital and insurance company be able to make certain queries against each others' databases. Similarly, to arrange for automated shipping of merchandise and to enable automated status checking, the e-commerce vendor and shipping company should be able to exchange relevant information, perhaps in form of database queries. In such environments, data must be released only in a controlled way among cooperative parties, subject to the authorization policies established by them. In this paper, we expose and study various facets of problem of such a collaboration problem.

In general, enterprise data may appear in a variety of forms, including the simplistic key-value forms like Google's BigTable. However, for concreteness, we assume that all data is stored in relational form, with all tables in a standard normal form. In such a model, data access privileges are given by a set of access rules, each of which is defined either on original tables belonging to an enterprise or over the lossless joins of two or more of these. The join operations, coupled with appropriate projection and selection operations define the access restrictions; although in order to enable working with only the schemas, we do not consider selection operation. For each query, the system must ensure that it is allowed only if it satisfies access rules.

A query is authorized only if there is a given access rule providing enough privileges. However, as the access rules are defined on the join results of basic relations, a party can get information from several cooperative parties and perform local computation to obtain the result that is not authorized by any rule. To give a simple example, if an enterprise P is authorized to get relations R and S from P_R and P_S respectively, then it can obtain the result of $R \bowtie S$ (R and S can have a lossless join). As the access rules are made according to business requirements among the cooperative parties, it is possible that there is no rule authorizing P to access that join result. Consequently, there is a conflict among the access rules. To avoid such conflict, one solution is to add an additional rule to allow P accessing the join result. The alternative is to constrain the party so that it cannot access both R and S at the same time.

In this work, we explore the first approach, and remove the conflicts among the rules by adding more access rules. Given a set of access rules made according to business requirements, we propose an algorithm to generate the needed rules so as to remove all the conflicts among the rules. When a new rule is added, we need to further consider the new conflicts caused by this rule. To achieve that, the algorithm takes advantage of the functional dependencies among the basic relations to add all needed rules. Although the worst case complexity of the algorithm is exponential, the real world complexity is generally quite acceptable particularly due to fact that it is rare to have long chains of joins in practice. In addition, such a process can be done as pre-computation so that complexity is not critical.

Since the business relationships among the cooperative parties may change from time to time, the access rules also change correspondingly. We consider two types of changes on the access rules: independent change and cooperative change, where the first type of change only affects the rules on one single party and the latter one involves multiple parties. Since any changes on the access rules may invoke new conflicts, we also propose algorithms to remove conflicts in the cases of a new access rule is granted or an existing rule is revoked. In both cases, a single change can lead to a series changes in order ensure consistency. For the cooperative access rule changes, we assume that the enterprises negotiate and agree to the necessary changes in advance. This means that the actual changes must be introduced simultaneously for all the parties. We propose a mechanism to deal with the required synchronization in this case.

In addition to ensuring consistency of rules under change, we also need to address the issue of implementing changes to rules while the system is executing queries. The issues to address are to ensure that rule changes are introduced in such a way that minimum number of queries are affected. We devise

mechanisms for both individual and coordinated changes to rules in this case.

The outline of the paper is as follows. Section II addresses the issue of consistency of rules in a cooperative access environment and Section III describes an algorithm for consistency checking. Section IV deals with the problem of changes in the rules. Section V addresses the issue of query planning and execution under access rule changes. Section VI discusses the related work. Finally, Section VII concludes the discussion and lays out areas for future work.

II. Consistency of Access Rules

A. preliminaries

We consider a group of cooperating parties, each of which maintains its data in a standard relational form such as such as Boyce-Codd Normal Form (BCNF). It is possible to consider more complex normal forms as well, but is beyond the scope of this paper. We also assume simple select-project-join queries, i.e., no cyclic join schemas or queries. The query may be answered by any of the parties that has the required permissions. We assume that the join schema is given – i.e., all the possible join attribute sets between any two relations are known. Each join in the schema is lossless so that a join attribute is always a key attribute of some relations. We also assume that the rules are composeable, which means each rule has all the key attributes of the basic relations in its join path. We study the problems only involving the cooperating enterprises; no "helper" third parties are considered here.

Each cooperative party is given a set of access rules that are defined over the join results of basic relations owned by these parties. We call a sequence of joins as a join path. An access rule is further defined with the attribute set authorized on a specified join path.

Definition 1: A **join path** is the result of a series of join operations over a set of relations $R_1, R_2 ... R_n$ with the specified equi-join predicates $(A_{l1}, A_{r1}), (A_{l2}, A_{r2}) ... (A_{ln}, A_{rn})$ among them, where (A_{li}, A_{ri}) are the join attributes from two relations. We use the notation J_t to indicate the join path of rule r_t. We use JR_t to indicate the set of relations in a join path J_t. The **length** of a join path is the cardinality of JR_t.

An **access rule** r_t is a triple $[A_t, J_t, P_t]$, where J_t is called the join path of the rule, A_t is the set of authorized attributes, and P_t is the party authorized to access these attributes. (Note that projection over the authorized set of attributes is implicit here but may be done according to performance considerations.) Each access rule defines a new relation, and we can perform the relational operations such as join on them as well. Correspondingly, a query q can be represented as a pair $[A_q, J_q]$, and any party has the authorized rule can answer the query.

B. An running example

Our running example models an e-commerce scenario with five parties: (a) *E-commerce*, denoted as E, is a company that sells products online, (b) *Customer_Service*, denoted C, is another entity that provides customer service functions (potentially for more than one company), (c) *Shipping*, denoted S, provides shipping services (again, potentially to multiple companies), (d) *Supplier*, denoted P, is the party that stores products in the warehouses, and finally (e) *Warehouse*, denoted W, is the party that provides storage services. To keep the example simple, we assume that each party has but one relation for its local database described below. The attributes should be self-explanatory; the key attributes are indicated by underlining them. In each of these relations, a single attribute happens to form the key, but this is not required in our analysis.

1) E-commerce (order_id, product_id, total) as E
2) Customer_Service (order_id, issue, assistant) as C
3) Shipping (order_id, address, delivery_type) as S
4) Warehouse (product_id, supplier_id, location) as W
5) Supplier (supplier_id, supplier_name, factory) as P

In the following, we use oid to denote $order\_id$ for short, pid stands for $product\_id$, sid stands for $supplier\_id$, and $delivery$ stands for $delivery\_type$. The possible join schema is also given in figure 1. Relations E, C, S can join over their common attribute oid; relation E can join with W over the attribute pid, and W can join with P on sid. In the example, relations are in BCNF, and the only functional dependency (FD) in each relation is the one implied by the key attribute (i.e., key attribute determines everything else).

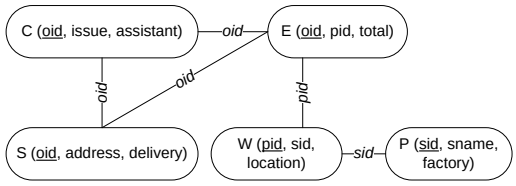

Fig. 1. The given join schema for the example

We now define a set of access rules given to the party E as described in Table I. (Suitable rules must also be defined for other parties, but are not shown here for brevity.) The first column of the table is the rule numbers, and the second column shows the attribute sets of the rules. The third column lists the join paths on which the rules are defined. The last column (redundant in this example) indicates the party to which the rules are given.

Rule No.	Authorized attribute set	Join Path	Party
1	{oid, pid, total}	E	P_E
2	{oid, issue, address}	$S \bowtie_{oid} C$	P_E
3	{oid, pid, total, issue}	$E \bowtie_{oid} C$	P_E
4	{oid, pid, sid, location, total}	$E \bowtie_{pid} W$	P_E
5	{pid, sid, factory}	$W \bowtie_{sid} P$	P_E

TABLE I
ACCESS RULES FOR E-COMMERCE COOPERATIVE DATA ACCESS

C. Rule conflicts and consistency

There are two styles in which rules can be given. An implicit specification means any valid compositions of the given rules are also considered as valid rules. In contrast, an explicit specification lists out all the allowed accesses and any access not included in the list is not allowed. Given our chosen method of conflict resolution (i.e., by adding rules), the distinction between implicit and explicitly specification is not significant, as we shall see shortly.

For a query q to be authorized by explicit rules, there must be an access rule r_t whose join path J_t is equivalent to J_q and A_t is a superset of A_q. In general, it is possible that a party obtains two pieces of information, say R and S according to two different explicit rules. It is then free to join these locally and obtain $R \bowtie S$ even if no rule authorizes access to this composition. Such a situation creates a **conflict** since access to $R \bowtie S$ is not allowed by the rules but is still possible. We say the set of rules are **inconsistent** if an access conflict exists with respect to any join path. As stated earlier, the inconsistency can be removed in one of two ways: (a) By adding additional rules that allow for all potential compositions that have not been explicitly specified, or (b) by actually denying access to $R \bowtie S$. The latter can be done via the well-known Chinese Wall policy [4] whereby the party can either access R or S but not both simultaneously (and hence cannot compute $R \bowtie S$).

In this paper we adopt solution (a). To do so, one must generate all possible compositions of the given rules and add any missing ones from the list. In this case whether we start out with an implicit or explicit specification, the result will be the same. We now define the notion of closure to make the rules consistent.

Definition 2: If two rules r_i, r_j of party P can be joined losslessly according to the given join schema, and the resulting information $[A_i \bigcup A_j, J_i \bowtie J_j]$ is also authorized by another rule r_k of party P, then we say the two rules are **"upwards closed"**. For a set of rules, if any two rules that can be joined losslessly are "upwards closed", we say the set of rules is **consistent**, and the rules form a **consistent closure**.

As access rules are usually designated among the parties based on their business needs, the given set of rules is usually inconsistent. Therefore, it is desired to have a mechanism to add the necessary rules so as to make the rule set a consistent closure. Although we are discussing the problem under cooperative environment, the rule consistency property only applies to each individual parties only. It is because that the inconsistency of rules is caused by local computations. In other words, it is only required that the rules given to one party form a closure, and the rules on other different cooperative parties are considered separately. Thus, we inspect one party at a time, and the mechanism for achieving consistent closure below only involves rules on one party.

D. Key attributes hierachy

Since we assume all the basic relations are in BCNF, and the join paths are the results of lossless join operations, the key attributes of basic relations in the given join schema form a hierarchal relationship. For instance, suppose that the relations R, S have their key attributes $R.K$ and $S.K$ respectively. If these relations can join losslessly, then the joining attribute must be the key attribute in at least one of them [2]. That is, either the join is performed on $R.K$, $S.K$, or $R.K$ is the same attribute as $S.K$. In either case, one key attribute from a basic relation is also the key attribute of the join result of the two relations. Therefore, if the join is performed over the attribute $S.K$ ($R.K \neq S.K$), then the attribute $R.K$ can functionally determine the relation S. In such case, we say $R.K$ is at a higher level than $S.K$, denoted $R.K \rightarrow S.K$. Thus, for a given valid join path, the key attribute of the join path is a key attribute from a basic relation. We call the key attribute of the join path in an access rule as **key** of the rule. Also, the join attributes in the join paths are always key attributes of some basic relations so these join attributes form the hierarchal relationship. For instance, in the running example, the key attribute oid is at the top level, and we have the hierarchal relationship for three key attributes, where $oid \rightarrow pid \rightarrow sid$.

For each key attribute of basic relation, we create a group for the rules that take this attribute as their key attribute. As the rules within this group share the same key attribute, any two of them can join over their key attributes.

Definition 3: A **join group** is a group of access rules associated with a key (join) attribute, where all the attributes in these rules functionally depend on this attribute. If a join group is **consistent**, then it is called a **consistent join group**.

Since some rules can be the result of local computation over other rules, there also exist relationships among the rules. In fact, the relationships are based on the join paths of the rules as they present the possibilities of join operations. Given a rule r_t with join path J_t, we call a join path as a **sub-join path** of J_t if it is a join path that contains a proper subset of relations of JR_t. We say a rule defined on a sub-join path of J_t is a **relevant rule** to r_t. A rule r_t can be locally generated only by combining the information from its relevant rules, otherwise, the generated rule contains extra information from relations not in J_t. Based on the relevance relationship, the rules are organized in a **graph structure**. Such a graph has different levels according to the different lengths of the join paths. Each node in such structure is a rule marked by its join path. Two nodes are connected if one is the relevant rule of the other. For instance, figure 2 shows a graph structure. J_2 is a sub-path of J_6, and r_2 is a relevant rule to r_6. They are connected in the graph, and they are on different levels.

III. Consistency Checking Algorithm

Given a set of rules, our goal is to generate the consistent closure of it. Our algorithm uses the join attribute hierarchy property and join groups to efficiently generate the consistent closure. The rules are first divided into different join groups and consistent join groups are generated. Next, based on the join attribute hierarchy, each join attribute is considered for deriving further rules, and any such rules are added to the rule closure. When this procedure terminates, we have the entire consistent closure.

A. Consistent join group generation

The first step is to generate the consistent join group. With the input as a join group of some given rules, the algorithm considers each derived rule in the order of join path length. When counting the join path length for a group, we only include the basic relations whose key attributes are the attribute associated with the join group, and we call these relations as **dependent** relations of the group. A join path that involves only dependent relations is called a **dependent** join path. Relations whose key attributes are not this attribute are

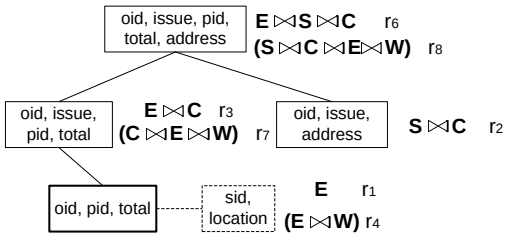

Fig. 2. The consistent join group of oid

called **optional** relations. Optional relations or join paths are associated with the dependent join paths. In the graph, we only assign one node for each dependent join path. If the given rule set includes two or more rules that have the same dependent join path, they are assigned to the same node in the graph but identified with their optional relations. When generating the consistent join group on the higher level parent nodes of this node, the algorithm needs to generate corresponding rules using each of the rule associated with this node. We will use our running example to illustrate this.

The join paths discussed below to generate the consistent join group are all dependent join paths. The algorithm looks for each join path length to check if a pair of rules can be joined to form a join path of desired length. Starting from the length of 2, the algorithm takes rules with length less than 2 and generates all the pairs of them. If the resulting rule is not present in the given join group, the algorithm adds it to the group. Otherwise, the resulting rule is merged with the existing rule on their attribute sets. Meanwhile, the graph structure is also built and edges are added between the resulting rule and the rules being examined.

Next, the algorithm checks join path length of 3 to k where k is the number of dependent relations in the join group. When inspecting the length i join-path, the algorithm first takes the rule r_m with maximal length ($m < i$) in the current join group. The algorithm then looks for possible pairs including r_m, so the other rule r_j whose dependent join path should have the property that $|JR_j \setminus JR_m| + |JR_m| = i$. The rules are chosen in the reverse order of join path length since the rule with longer join path includes all the attributes from its relevant rules. All the rules with join paths that do not satisfy this property will not be considered in pair with r_m, and a rule is never paired with its own relevant rules. By iterating over all the join path lengths, the consistent join group can be generated.

To illustrate the process, we use the running example. The first 4 rules have the same key attribute oid, and they are put into the same join group of oid. Within these rules, r_4 has an optional relation W which does not depend on oid. It is only counted as join path of length 1 and is associated with the node of r_1 since its dependent join path is the same as J_1. Then the algorithm begins with join path length of 2. As the only rule with join path length less than 2 is r_1, no pair is found. However, the given rules r_2 and r_3 are both of length 2, so they are checked with r_1 to see the relevance relationship. Thus, r_3 is connected with r_1 in the graph. Next, the algorithm checks the length of 3. Since this join group only includes 3 different relations $\{E, C, S\}$, this is the maximal length to check. The algorithm first takes r_2 and looks for the rule can pair with it. Among the join path J_1 and J_3, J_3 is selected since its length is longer, and there is no need to further check with J_1 as it is relevant to J_3. Therefore, a rule r_6 with join path $E \bowtie C \bowtie S$ is added to the join group with the attribute set $A_2 \bigcup A_3$. In the graph structure, this rule is connected with both r_2 and r_3.

In addition, rule r_4 has the optional relation W, and it is associated with r_1 in the group. Therefore, all the rules that r_1 relevant to also have this optional relation. In such case, based on r_6 and r_3, another two rules are added into the join group. This makes join group consistent and is listed in Table II. Here the first 4 rules are given and rule 6 to 8 are added by the algorithm to make the join group consistent. The built graph structure is shown in Figure 2. In the figure, the rule numbers are indicated beside the rule join paths, and the dashed box shows the optional relation of W. Since r_4 has the optional relation E and overlaps with r_1 on dependent join path, all the parent rules of r_1 which are r_3, r_6 should also have corresponding rules including the optional relation W, which are the rules r_7, r_8.

Rule No.	Authorized attribute set	Join Path	Party
1	{oid, pid, total}	E	P_E
2	{oid, issue, address}	$S \bowtie_{oid} C$	P_E
3	{oid, pid, total, issue}	$E \bowtie_{oid} C$	P_E
4	{oid, pid, sid, location, total}	$E \bowtie_{pid} W$	P_E
6	{oid, pid, total, issue, address}	$E \bowtie_{oid} S \bowtie_{oid} C$	P_E
7	{oid, pid, total, issue, location, sid}	$C \bowtie_{oid} E \bowtie_{pid} W$	P_E
8	{oid, pid, total, issue, location, sid, address}	$S \bowtie_{oid} C \bowtie_{oid} E \bowtie_{pid} W$	P_E

TABLE II
GENERATED CONSISTENT JOIN GROUP OF oid

B. Iteration of key attributes

We take advantage of the key attributes hierarchy property to develop a mechanism that can achieve the consistent closure. As the key attribute hierarchy can be obtained based on the given join schema, and we assume this information is available when the algorithm is being executed.

At the beginning, the algorithm makes an empty set called **target rule set**, and the algorithm keeps adding rules into this set. At the end, the target rule set is the rule closure we need. For the given set of rules, the algorithm first puts each rule into different join groups based on its key attribute, and it will only be assigned into one join group. Then, for each join group, the algorithm generates the consistent join group respectively.

Next, the algorithm iterates each join group according to the level of its associated attribute in the key attribute hierarchy. To begin with, the algorithm inspects the join group of the top level attribute. All the rules in the group being inspected are put into the target rule set first. Then, the algorithm checks the lower level groups one by one. For each join group being checked, all the rules in the current target rule set are iterated.

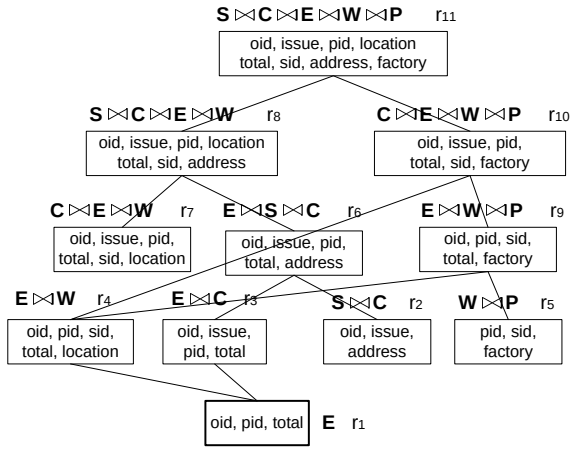

Fig. 3. The graph structure for the consistent closure.

Rule No.	Authorized attribute set	Join Path	Party
1	{oid, pid, total}	E	P_E
2	{oid, issue, address}	$S \bowtie_{oid} C$	P_E
3	{oid, pid, total, issue}	$E \bowtie_{oid} C$	P_E
4	{oid, pid, sid, location, total}	$E \bowtie_{pid} W$	P_E
5	{pid, sid, factory}	$W \bowtie_{sid} P$	P_E
6	{oid, pid, total, issue, address}	$E \bowtie_{oid} S \bowtie_{oid} C$	P_E
7	{oid, pid, total, issue, location, sid}	$E \bowtie_{oid} C \bowtie_{pid} W$	P_E
8	{oid, pid, total, issue, location, sid, address}	$S \bowtie_{oid} C \bowtie_{oid} E \bowtie_{pid} W$	P_E
9	{oid, pid, sid, factory, total}	$E \bowtie_{pid} W \bowtie_{sid} P$	P_E
10	{oid, pid, total, issue, sid, factory}	$C \bowtie_{oid} E \bowtie_{pid} W \bowtie_{sid} P$	P_E
11	{oid, pid, total, issue, location, sid, factory, address}	$C \bowtie_{oid} S \bowtie_{oid} E \bowtie_{pid} W \bowtie_{sid} P$	P_E

TABLE III
GENERATED CONSISTENT CLOSURE BASED ON GIVEN RULE SET

If the rule r_t from the current target rule set contains the join attribute that is associated with the join group being checked, then each rule in the join group being checked can join with r_t. The algorithm generates all these rules by making the union of join paths and the attribute sets, and it adds these generated rules into the target rule set. If there are already a rule in the target rule set with the same join path, the generated rule is merged with the existing rule by making union of the attribute sets from the rules.

As the algorithm iterates all the join groups, the target rule set will keep grow and eventually form the consistent closure. As rules are added to the target rule set, the algorithm also updates the graph structure capturing the rule relevance relationships. If a new rule is generated, it is appended to the graph. Connection edges are added between the rule and the pair of rules that generate it, and the attribute set can be updated. The detail algorithm is described in Alg. 1.

We can use the running example to illustrate the process of join group iteration. According to the key attribute hierarchy, oid is the top level attribute. Thus, the consistent join group of oid which is listed in Table II is copied to the target rule set. The only remaining join group is the group of pid since there is no given rule takes sid as key attribute. Also, there is only one rule r_5 in the join group of pid, and this join group is already consistent. As in the key attribute hireachy, pid is on the next level of oid, the algorithm checks each rule in the current target rule set to see if it contains the attribute pid. The set of rules $\{r_1, r_3, r_4, r_6, r_7, r_8\}$ all have this attribute, so 6 rules joining with r_5 are generated and added to the target rule set. However, some of these rules have the same join paths and they are merged with existing rules, so only 3 new rules are added to the target rule set. Finally, we generate the consistent closure as listed in Table III. The last three rules are generated in this process. Figure 3 shows the built graph structure, where relevant rules are connected by edges. The attribute sets of the rules are shown in boxes and the join paths together with rule numbers are shown above. The rules are put into 5 levels based on their join path length.

C. Average case complexity

The complexity of the algorithm depends on the given join schema and given rules. In worst case, generating a consistent join group takes exponential time. However, in real cases, usually a join group will not include more than 4 dependent relations. We make the assumption that the maximal number of dependent relations in a join group is 4. In addition, we assume there are at most k given rules in a join group. Within a join group, there are some given rules overlap on their dependent join paths. Assuming the number of overlapped rules is p, then there are $k - p$ nodes for initially given rules. As the most number of relations is 4, we have $k - p < 16$. For the algorithm, at most 22 pairs of nodes will be examined, and there are at most 11+8p rules are added into the consistent join group. As k and p are usually small, the number of rules in a consistent join group is usually less than 20 and the complexity of generating it is also low. we can think the generation of consistent join groups takes constant time and there are at most C rules in a consistent join group.

If there are m join groups in total, it looks like we have the complexity of C^m in worst case. However, within a join group, there is only one dependent relation that can join with the rules in the next join group to be inspected. If at most v rules including such dependent relation, then at each step only $v*C$ rules will be added, and the complexity is $O(v*C*(m-1))$. In many cases, a join group contains only one or no rule such as the join group of pid and sid in the example, so C is fairly small for many join groups. Also, the length of a valid join path m is usually very small as a join of 5 relations from different enterprises should be a rare case. Therefore, the complexity of the algorithm in real scenario is much lower than the theoretical worst case one.

Theorem 1: Given a rule set, the algorithm generates its consistent closure.

Proof: Assuming there are two random rules r_i, r_j, we check whether the consistent closure generated by the algorithm always have r_k, which is the join result of them. r_i, r_j

Algorithm 1 Rule Closure Generation Algorithm

Require: Given access rule set R on one party
Ensure: The set of rules R^+ that is a consistent closure

```
Put rules from R into join groups based on their key
Put the key attributes of relations into a priority queue Q
  based on its level in hierarchy
for Each join group G do
  Generate the consistent join group G+
  for Length k ← 2 to 4 do
    Mark all rules unvisited
    for Each unvisited rule r_i length < k do
      if Exists r_m, where |J_j − J_i| + |J_i| = k then
        Join r_i with r_m and get result r_j
        if There is no rule in R+ of join path J_j then
          R+ ← r_j
        else
          Get the rule and merge with r_j
          R+ ← updated r_j
        Mark its relevant rules visited
while Q ≠ ∅ do
  Dequeue the key attribute, and get its associated G+
  if R+ ≠ ∅ then
    for Each rule r_r in R+ do
      if r_r includes the key attribute of G+ then
        for Each rule r_g in G+ do
          Join r_r with r_g and get result r_j
          if There is no rule in R+ of join path J_j then
            R+ ← r_j
          else
            Get the rule and merge with r_j
            R+ ← updated r_j
  R+ ← ⋃ G+
```

can be given rules or the rules generated by the algorithm. Firstly, if r_i, r_j have the same key attribute, the two rules will be in the same join group. When the algorithm generates the consistent join group, it tries all possible combinations of the dependent relations. In addition, optional relations are considered from bottom up, so there is always a rule in the generated consistent join group that has the same join path as r_k. When checking the rule relevance in the graph, the attributes from the relevant rules are added to the higher level rules so the rule has the same join path as r_k also has all the attributes from r_i and r_j. Since the algorithm examines each join path length in ascending order, it does not matter if r_i, r_j are given rules or generated rules, and r_i, r_j are always upwards closed.

If r_i and r_j are not in the same join group, then we assume the key attribute of r_i is on the higher level than the key of r_j. If both rules are the given rules and r_i includes the key attribute of r_j, when the algorithm iterates the join group of r_j, r_i is already in the target rule set, and their join result is put into the target rule set. On the other hand, if r_i is a generated rule, it is always added into the target rule set by the algorithm. if it can join with r_j, the result is added to the target rule set also. Thus, after checking the join group of r_j, all the possible joins over that join attribute are examined. If r_j is a generated rule, it is in its consistent join group, so the algorithm adds the result of r_i and r_j into the rule set. Therefore, all the rules are upwards closed, and the generated rule set is consistent.

■

IV. Consistent access rule changes

As time goes, cooperative parties may change the access rules because of the change of business needs. A change of access rule can either be granting more access privileges to a party or revoking some existing privileges. In the case of access rule are changed on a party, it may cause new conflicts among the rules. Thus, a mechanism is needed to maintain the rule consistency while access rules are changed.

A. Two types of rule changes

In cooperative environment, the access rules can be changed at each party individually and can also be changed between several parties at the same time. One possible architecture for the authorization is that the rules are stored at the central place different from any cooperative parties. There exists an independent query optimizer reading the authorization rules and generating the query plans. However, usually cooperative enterprises do not share a single independent query optimizer. Instead, each party that answers the queries usually generates the query plan locally. Therefore, without a centralized party, each cooperative parties should keep a copy of the access rules locally. Each party needs to know more rules than the ones given to itself as it can exchange information with parties that have the rules on the same join paths. Therefore, we assume all the access rules are duplicated at different cooperative parties. We discuss two types of rule changes below.

1) Independent change: This type of rule change only applies to a single party. Although a join path may involve authorizations from several parties, the change is only happened to one party. It may because of one party no longer trusts the other or their business relationships get changed. Such changes usually affects only a small set of rules. Even if the change only takes on a single rule, to maintain the consistency of the rule set, usually a set of rules need to be changed accordingly. The discussions below about the granting and revoking of access rules can directly applied to this type of change. After the party changes its access rules, it board-casts the change to other cooperative parties.

2) Cooperative change: Sometimes a group of parties may want to update the access rules among them at the same time. These parties may negotiate the rules together and apply the changes on multiple parties at the same time. The group of rules need to be updated as a whole, and we call this type of change as **cooperative change**. In such case, the updates on several parties need to be synchronized. We call the parties involved in a cooperative rule change as **change cooperative parties**. A cooperative change need to be performed among these parties atomically from a temporal perspective.

To achieve that, we can 2PC protocol for the rule updating. Among the change cooperative parties, one party is selected

as the master party, which we can also call it **coordinator**. In real scenario, usually the cooperative change is invoked by a single party, and in such case that party can work as the coordinator. All the other change cooperative parties are called slave parties.

Since we assume the rule changes do not happen frequently, each party can only be involved in one cooperative change process at a time. Therefore, if a slave party is under an updating process, it will have a lock on the current rules given to itself and other rule updating requests received are not accepted and it acknowledges the master node about the status.

Therefore, to perform a cooperative change on the rules, the mechanism works as below. According to the 2PC protocol, the update process is divided into a voting phase and a commit phase. In the voting phase, the master party (coordinator) sends messages to all slave parties indicating the set of rules being changed, and each slave party is required to update the rules related to itself. If the slave party can update its rules, which means there is no ongoing rule updates at this party, the agreement will be sent back to the coordinator. Only if the agreements from all salve parties are received, the coordinator will go into the commit phase. In the commit phase, the coordinator sends a commit message to slave parties to finish the rule update and locks are released. Otherwise, the updating transaction is aborted, and the coordinator will try it later.

B. Consistently grant more information

When more access privileges are granted to a party, we need a mechanism to maintain the consistency among the rules. There are also two types of grants. The first is adding non-key attributes (non-join attributes) to a rule. If a rule is granted with more attributes, then the algorithm first selects the higher level parent rules of this rule in the graph. We search upwards in the graph, and this can be done with a depth first search. Along the path it searches, if the rule being inspected does not have these expanded attributes, then the algorithm adds these attributes to the rule. If the rule being inspected already has these attributes, the search along this path will stop and another path will be picked. Consequently, the added attributes will be propagated to all the related rules that are on the higher level of the rule being changed. For instance, in our running example, if the attribute $delivery$ is added to r_2, then the rules $r_6, r_8, r_1 1$ on the same path all need to add this attribute.

In some cases, the attribute added is not the key attribute of the rule being modified, but the attribute is the key attribute for other rules. Therefore, by adding this attribute, the modified rule can possibly further join with other rules. To deal with this situation, once a join attribute is added to a rule (non-key attribute for the rule being modified), the algorithm checks if there exists a join group associated with this attribute. If that is the case, rules which use this attribute as the key attribute are selected from the generated consistent closure. Each rule selected is then joined with the rule being modified, and the resulting rule is added to the rule set or merged with existing rule. Only these rules need to be added to the rule set.

On the other hand, there is another type of change of rules, where a rule on a new join path is granted to a party. In such case, we need to check if this rule can join with existing rules to generate legitimate new rules. The mechanism is similar to the previous approach for generating the consistent closure. As the newly added rule r_n has a new join path, we first obtain the key attribute of r_n, and then r_n is put into the join group whose associated attribute is the key attribute of r_n. Within this group, as a new rule is added, the algorithm recomputes the consistent join group. This can be done efficiently since these rule all can join over their key attributes. In fact, the rule r_n is checked with existing rules in the consistent join group. r_n is inserted into the graph of the join group, and its relevant rules and the rules it relevant to are not checked with it. All the other rules are checked and r_n can join with each of them to form a new rule and put into the consistent join group. The algorithm then keeps the set of newly added rules for the following rule generation.

In the next step, each of the newly added rules is iterated to see what are the other rules that can be generated based on it. For each newly added rule r_n, the algorithm checks the join attributes in its join path (excluding its key attribute), and for each join attribute the algorithm combines r_n with the rules in the join group and add them into the newly added rule set. This process actually finds all needed rules which has the same key attribute as the key of r_n. After that, the algorithm looks for existing rules that include the key attribute of r_n but not using it as their key attributes. Each such rule can join with the newly added rules in the group of r_n over the key attribute of r_n. The algorithm adds all these generated rules into the rule set so as to complete it as a consistent closure. The attribute set of the rules should also be considered. If there exists a rule on the same join path, the attribute sets of the two rules are merged.

In our running example, we can think a new rule r_{12} with join path $E \bowtie_{oid} S$ is added whose attribute set is $\{oid, pid, total, address\}$. In this case, the algorithm will put the rule into the join group of oid. In the graph structure, such a rule has relevant rule r_1, and it is the relevant rule of r_6, r_8. Therefore, other rules in the join group are paired with r_{12}. However, most of these generated rules already exist in the current join group, so the only new rule r_{13} need to be added is on the join path of $S \bowtie_{oid} E \bowtie_{pid} W$. Next, the algorithm checks the rules r_{12}, r_{13}. Since both of them include pid as non-key attribute, and there is no join group of sid, both rules are paired with the join group of pid. This results in only one additional rule r_{14} on the join path of $S \bowtie_{oid} E \bowtie_{pid} W \bowtie_{sid} P$. Since oid is the top level join attribute, by adding this rule to the rule set, the consistent rule closure is achieved.

In worst case, if there are already n rules exist in the closure, and there are C rules in the join group. Adding one more rule will need adding additional $C - 1$ rules to maintain the consistency. For the above mechanism, the recompilation of the join group will take C steps since each existing rule need to be checked. The remaining complexity depends on the join groups associated with the added rules. If the total number of levels is u, and assuming at most s rules in a join group has the join attribute of the inspected group, then the number of pairs to examine in for one join group is $s * C$. The total

complexity can be $O(C * u * s)$.

C. Revocation of existing access rules

Besides grant of more access privileges, the changes on the rules can also be the revocation of some existing access rules. Similar to the grant case, we consider the revocation operation can be just on some non-key attributes or completely removal of one rule. We first discuss the situation that non-key attributes are revoked. The revocation of attributes on just one rule usually causes inconsistency. It is because that its relevant rules may still have the revoked attribute, and the party can still access to these attributes through local computation. Therefore, we need to also revoke these attributes from such relevant rules. Based on the built graph structure, the algorithm retrieves the relevant rules of the rule being modified, if any relevant rules include such revoked attributes, these attributes are also revoked from these rules.

For instance, we can take the example of Figure 3. Let's assume the modification is made on the rule r_{10}, and the attribute $factory$ is revoked. In such case, its relevant rules r_9, r_5, r_4, r_1 are checked. Attribute $factory$ should also be revoked from these rules. Therefore, r_9, r_5 are modified to keep the rule closure consistent.

On the other hand, if one rule with a join path is completely revoked from the rule set, we need to make sure that such join path can no longer be generated from the remaining relevant rules. Therefore, each possible ways to enforce the join path need to be obtained and the possible pairs should be taken apart. To achieve that, the algorithm uses the graph structure built before. In the graph, only the **direct relevant rules** of the revoked rule r_v are examined. The direct relevant rules of r_v are the relevant ones in the graph that directly connected with r_v with one edge. For each of the direct connect rule r_d, the algorithm computes its matching join path J_m for J_v. The **matching join path** J_m is a join path that $J_m \bowtie J_d = J_v$, $J_m \neq J_v$, and $|J_m|$ is the minimal one among such join paths. Given the join schema, J_m can be efficiently determined by computing the minimal set of $JR_m = JR_v - JR_d$. If such set does not form a join path that is a sub-path of J_m, then the matching join path of r_d does not exist. Otherwise, the matching join path J_m is obtained. In the graph, if a rule containing J_m is not found, higher level rules connecting to it are examined, and the one with minimal join path length is selected as J_m.

As we can check the enforceability of the rules [1], we assume what are the locally enforceable rules is already known. Thus, for each pair of rules selected, the algorithm needs to remove one rule from it so as to make the join path no longer enforceable. If a rule in the pair is not locally enforceable, we prefer to remove it since it does not cause cascade revocations. In contrast, if a rule in the pair is locally enforceable, by removing this rule, we need to make sure all the rules that can compose this one are taken apart. Thus, a cascade of revocation will occur. In addition, when iterating each pair, the algorithm also records the number of appearance of the rules. The rule with most appearance is more preferable to be removed since removing one such rule can break several pairs. For the locally enforceable rules that are being removed, the algorithm puts them into a queue so that they are processed in a cascade manner.

For instance, in figure 3, the rule r_{10} is completely removed. This rule has three direct relevant rules $\{r_4, r_9, r_3\}$. r_9 is first examined, and its matching join path is $\{C\}$. As $\{C\}$ is not available, r_3 is paired with r_9. On the other hand, r_3 can pair with r_5, r_9, and r_4 cannot pair with any other rule. Therefore, the algorithm needs to break all the pairs of rules $\{(r_3, r_5), (r_3, r_9)\}$. Since r_3 appears in both pairs, the algorithm will revoke it also, and it is put into the queue. As r_3 is not locally enforceable, we do not need further revocation. Finally, revoking r_{10} with r_3 will keep the rule closure consistent.

Since the above mechanism to remove a rule is complicated and it cannot guarantee the minimal number of rules being removed (because it considers only for one next level and do not know further levels). We also consider to remove the rules in another way. Because a revocation is usually issued by a single party, and this party usually revokes the access rules with its own data. Therefore, when a revocation is issued, it is general for the party to revoke all the rules including that basic relation. If this is the case, the revocation can involves a set of rules and all including that basic relation, and the consistent closure is still maintained.

According to this idea, if we want to remove a rule, we can also remove a set of rules containing the same basic relation. Thus, another possible way to consistent revoke a rule can be found. The algorithm can first obtain all the relevant rules of r_v. For each relevant rule, the algorithm records the basic relations appeared in the join path. As last, the basic relation associated with fewest number of rules is selected, and rules including this basic relation are removed from the set.

Back to our example, and we want to revoke rule r_{10}. This mechanism first retrieve its relevant rules which are $\{r_4, r_5, r_9, r_3, r_1\}$. These join paths are examined, and the appearances of 4 basic relations are checked counted. Therefore, relation C appears once, E appears 3 times, W appears 3 times, and P appears twice. Thus, the algorithm tries to remove the rule whose join path has C. r_3 is removed, and this result is the same as the previous algorithm. In general, these two mechanisms produce different results.

Here, we argue that the rule closure property is different from the rule enforcement issue. Though removing a set of rules will affect the enforceability of other rules, we only focus on maintaining the rule consistency property here. For the second approach, the complexity is $O(n * t)$, where n is the number of relevant rules, and t is the maximal number of relations in a join path.

V. Query planning with access rule changes

In this section, we consider the problem of access rule changes when queries are being processed. We assume that we have an existing algorithm [1] to generate a query plan among the cooperative parties based on the consistent closure. In addition, as not all the rules can be enforced among the party, we have an algorithm [1] to check the rule enforceability.

For instance, if there is an access rule saying party P can access the join result of $R \bowtie S$, but there is no existing party can access the two relations at the same time, then there is no way to enforce the rule. For each rule in the consistent closure, the algorithm tells if it can be enforced or not, and only rules that can be enforced are useful for query planning. While a generated query plan is being executed, if the access rules are updated, then some steps in the plan may not be able to be executed. As grant of access does not affect the query plan execution, the rule changes here are revocations on access privileges. We discuss how to adapt the query execution with the access rule changes below.

A. Snapshot solution

Since the biggest concern of this problem is that the access rules are changed during the query plans are being executed, one possible solution is that at the time each generated query plan is executed, we make sure the access rules related to this plan are not changed. Thus, in this solution, we first obtain a snapshot of a consistent state of the rules before doing the query planning, and the query plan is generated under such snapshots rules so that the execution of the query plan is not affected by the rule changes.

Assuming the query q is received by a party P which has the rule authorizing the answer of the query, and the party is going to do the query planning locally. Therefore, the party P first sends the access rules it caches to other parties with the mark of the query q identifying this snapshot is exclusively for q. Each cooperative party that receives the message compares its rules with the received rules from P. Each party should either agrees on the rules regarding itself or sends its updated rules to the party P. If a party is performing a rule updating, it may send a notification back to P, and P have to defer the query planning and try again later.

Therefore, party P can always obtain and plan with the updated rules, and the party receiving the messages will take a snapshot on the rule set it acknowledged to P with the mark of the query q. Later, even if the party updates its rules, it keeps these snapshotted rules until the query is answered. Then, the party P will generate the query plan using the query planning algorithm, and the plan is executed on these cooperative parties. If there is no rule update between the time of P making the snapshot and the query is answer, the query plan can be executed without problems. In the case that the rules on a party are updated and the step in the query plan no longer executable, the party may lack the authorization to perform the steps in the query plan. However, it can use the snapshotted rules with the marks of q to retain the authorizations from other parties. Other cooperative parties will also honer the snapshotted rules even if currently the access rules are updated. Therefore, the query plan generated from the snapshot can always be executed.

Finally, as soon as the query q is answered, party P sends finish messages to other parties indicating that the execution of the query q has finished. A party receives this message will remove the snapshotted rules associated with query q. Thus, any following queries cannot take advantage of these rules and they can only be processed according to updated rules.

To give an example, we assume there are three parties P_R, P_S, P_T. Each party can access their own relations R, S, T. These relations share the same key attribute so that they can always be joined. In addition, P_T is allowed to get the relation R, and the result of $R \bowtie T$. P_S is allowed to get $R \bowtie T$, and of course the join of $R \bowtie S \bowtie T$. The incoming query q asks for the information on the join path of $R \bowtie S \bowtie T$ and it is authorized by the rule on P_S. As P_S is going to generate the query plan, it first send messages to snapshot the current rules. Based on these rules, a query plan will first let P_T to get the relation R and generate the result $R \bowtie T$, and this result is send to P_S to perform a further join and answer the query q. At the time the plan is generated, the rules given to P_T are modified, and the rule that authoring P_T to access R is revoked. In such case, when P_T execute the query plan, it will use the snapshotted rule associated with q. Therefore, party P_R knows the access for R is to answer the query q, and the access is allowed. After q is answered, P_S sends the finish messages to remove the snapshot. At the end, party P_T can no longer access the relation R.

B. Dynamic planning

In stead of the snapshot approach, as we assume the access changes is infrequent, we can use the mechanism to adjust the query plan dynamically. The idea is to execute the query plan first, and when a step cannot be executed because of the access change, the algorithm looks for alternative ways to replace the step and continue the query plan. However, sometimes the access change makes the query no longer answerable, and therefore the existing plan may be aborted, and it should be checked again to see if there still exists a valid query plan to answer the query. As discussed above, we have two types of revocations.

1) Revocation of non-key attributes: If only the non-key attributes in the rules are revoked, the generated query plan should still be able to run. However, these revoked attributes cannot be retrieved at these steps in the plan. Thus, the algorithm first checks if the following steps in the plan can access these attributes efficiently. If this is not the case, as the plan can still be executed, the result will only have the partial answer of the query without these revoked attributes. Therefore, the algorithm constructs another query to retrieve these missing attributes. The new query q' contains only only these attributes as well as the key attributes from their relations which are used to join with the previously got partial results.

Since the planning party P can no longer retrieve these attributes, such a query must be answered by another party, and party P will send the obtained partial result to that party. Thus, the party to answer q' must have a rule on the same join path as the one in query q. If such a party exists, the query planning algorithm is re-executed to get a plan for q'. Otherwise, the query q cannot be answered. If a re-planning is possible, the planning party will first collect rules from cooperative parties to get the most up to date rules, and then run the query planning algorithm. Finally, the result of q' is joined with the partial results of q, and that answer is obtained.

2) Revocation of join paths: In the previous case, the enforceability of the join paths are not affected. Consequently, the query plan can continue to execute. In contrast, if a rule is totally revoked, it is possible that the current plan cannot be executed any more. In this situation, the algorithm looks for an alternative party that can perform these steps. If the same intermediate result can be obtained from the alternative party, the following steps in the plan can still be executed. If the replacement of the steps cannot be found, the algorithm has to abort the current plan. Since it is not clear whether the rules and their join paths are still enforceable, the rule enforcement checking algorithm needs to be performed to determine the set of enforceable rules. If query q no longer be able to answer, the algorithm finishes. Otherwise, the query planning algorithm is re-executed, and a new plan is generated if possible. Since we assume the access changes are infrequent, this dynamic adaption mechanism works if the new plan is generated.

VI. Related Work

The problem of controlled data release among collaborating parties has been studied in [8]. The authorization model in this paper is identical to ours and provides the motivation for our work. Its main contribution is an algorithm to check if a query with a given query plan tree can be safely executed. It assumes all the given rules are already upwards closed. However, this is not the case in real, and access rules are usually made without consideration of consistency. Therefore, maintaining the consistency of the set of given rules is a crucial problem, that we address in this work.

In another work [7], the same authors evaluate whether the information release the query entails is allowed by all the authorization rules given to a particular user, which considers the possible combinations of rules and assumes that the rules are defined in an implicit way. Their solution uses a graph model to find all the possible compositions of the given rules, and checks the query against all the generated authorization rules. In our work, we assume access rules are explicitly given. Data release is prohibited if there is no matching rules.

There are some works on the access control in collaborative environments. In [16], the authors examined existing access control models as applied to collaboration, and pointed the weaknesses of these models. In addition, [10], [15] applied RBAC in the collaborative environments. As social network get popular, [6] discussed the problems in these situations, and [11] proposed a web services based mechanism for access control in collaboration. All these access control models are different from the one we are using. In [14], collaboration among enterprises was also studied, but that work focused on different application data and multilevel policies.

Processing distributed queries under protection requirements has been studied in [5], [9], [13]. In these works, data access is constrained by a limited access pattern called binding pattern, and the goal is to identify the classes of queries that a given set of access patterns can support. There are also classical works on query processing in centralized and distributed systems [3], [12], but they do not deal with constraints from the data owners.

VII. Conclusions and future works

As more and more enterprises work cooperatively to perform computations, securely providing access to cooperative data is important. We use an authorization model for cooperative data access based on the join results of the relational data. However, in the cooperative environment, access rules conflicts may arise among the rules made according to business requirements. We proposed a mechanism to make the set of cooperative access rules consistent. In addition, we also presented algorithms to maintain the rule consistency in the cases of granting and revocation of access privileges. At last, as running queries may be affected by the rule changes in the cooperative situation, we discuss the mechanisms for query planning that can adapt to the access rule changes.

In the future, we plan to perform experiments with real world cases to extensively evaluate the complexity of the algorithms. In addition, Chinese wall policies will be considered to remove the inconsistency among the rules, and how to implement it need to be studied. Moreover, we will further look into the more dynamic situation where parties can be added and removed from the cooperative environment.

References

[1] Consistent query plan generation in secure cooperative data access. *http://mason.gmu.edu/ mlep/submission.pdf*.

[2] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database Systems*, 4(3):297–314, Sept. 1979.

[3] 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(4):602–625, Dec. 1981.

[4] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[5] A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE*, pages 50–59. IEEE, 2008.

[6] B. Carminati and E. Ferrari. Collaborative access control in on-line social networks. pages 231–240. IEEE, 2011.

[7] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Assessing query privileges via safe and efficient permission composition. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 311–322. ACM, 2008.

[8] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Controlled information sharing in collaborative distributed query processing. In *28th IEEE International Conference on Distributed Computing Systems (28th ICDCS'08)*, Beijing, China, June 2008. IEEE Computer Society.

[9] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. 15èmes Journées Bases de Données Avancées, BDA*, pages 41–60, 1999.

[10] A. Gouglidis and I. Mavridis. domRBAC: An access control model for modern collaborative systems. *Computers & Security*, 31(4):540–556, 2012.

[11] A. A. E. Kalam, Y. Deswarte, A. Baïna, and M. Kaâniche. Access control for collaborative systems: A web services based approach. pages 1064–1071. IEEE Computer Society, 2007.

[12] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv*, 32(4):422–469, 2000.

[13] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB Journal*, 12(3):211–227, 2003.

[14] E. Y. Li, T. C. Du, and J. W. Wong. Access control in collaborative commerce. *Decision Support Systems*, 43(2):675–685, 2007.

[15] J. S. Park and J. Hwang. Role-based access control for collaborative enterprise in peer-to-peer computing environments. pages 93–99. ACM, 2003.

[16] Tolone, Ahn, Pai, and Hong. Access control in collaborative systems. *CSURV: Computing Surveys*, 37, 2005.