Consistency and Enforcement of Access Rules in Cooperative Data Sharing Environment

Meixing Le*, Krishna Kant, Sushil Jajodia

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

Abstract

In this paper we consider the situation where a set of enterprises need to collaborate to provide rich services to their clients. An enterprise may need information from several other collaborating parties to satisfy its business requirements. Such collaboration often requires controlled access to one another's data, which we assume is stored in standard relational form. We assume that a set of access rules is given to the parties to regulate the data sharing, and such rules are defined over the join operations over the relational data. It is expected that the access rules will be designed according to business needs of the involved enterprises and although some negotiation between them will be involved, only a comprehensive analysis of the rules can uncover all issues of consistency between rules and their adequacy in answering the authorized queries (which we call enforceability). In this paper, we provide such an analysis and provide algorithms for checking and removing inconsistency, checking for rule enforceability, and minimally updating the rules to ensure enforceability whenever possible using only the existing parties. The involvement of specialized third parties for consistency and enforcement purposes is not addressed in this paper.

Keywords: access rule consistency; cooperative data access; join path; rule enforcement; consistent join group

1. 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 e-commerce company needs to obtain data from a shipping company to arrange for automated shipping of merchandise and to enable automated status checking, and the shipping company requires the order information from the e-commerce company. The e-commerce company may have to further exchange data with warehouses and

*Corresponding author

Email addresses: mlep@gmu.edu (Meixing Le), kkant@gmu.edu (Krishna Kant), jajodia@gmu.edu (Sushil Jajodia)

suppliers to get the information about the products. In such an environment, information needs to be exchanged in a controlled way so that the desired business requirements can be met but other private information is never leaked. For example, a shipping company has all the information about its customers, however, only the information about the customers that deal with the e-commerce company in question should be visible to the e-commerce company. The information about the remaining customers should not be released to the e-commerce company. In addition, the data from shipping company may include other information such as which employee is delivering the order, and such information should not be released to the e-commerce company. Therefore, we need a mechanism to define the data access privileges in the cooperative data access environment.

We assume that each enterprise manages its own data and all data is stored in a standard relational form such as BCNF. We use relational data since it is the most used data form, but it is possible to extend the model to work with other data forms by defining suitable models for data composition. The data access privileges of the enterprises are regulated by a set of access rules. Each access rule is defined either on the original tables belonging to an enterprise or over the lossless joins of the data from several different parties. Using join operations, an access rule only releases the matched information from the parties. For instance, if the e-commerce company can only access the join result of its data and the shipping company's data, then only the tuples about the shipping orders from the e-commerce company can be visible to the e-commerce company. In addition, the attributes such as "delivery_person" are never released to the e-commerce company, so suitable projection operations are applied on the join results in access rules to further restrict the access privileges. To allow working at the schema level, the selection operation is not considered in this paper, although it is possible to extend the results to simple selection predicates. In the following, we expose and study various issues that arise in such an environment.

An enterprise typically needs data from several parties to provide services and answer queries. A query is authorized only if there is an 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 derive the result that is not authorized by any rule. To give a simple example, if an enterprise P is authorized to get relations E and S from the e-commerce and shipping companies respectively, then it can obtain the result of $E \bowtie S$ (over appropriate join attribute). If we say that there is no rule authorizing P to access the join result of $E \bowtie S$, there is a conflict among the access rules since $E \bowtie S$ is intended to be denied but is accessible to P.

In a complex environment, when the enterprises set up their rules, such conflicts may not be obvious, and making them aware of the issue via the kind of analysis presented in this paper is crucial for avoiding undesired leakage of their data. A more important issue is one of resolving the conflict. This can be done in two basic ways: (a) by explicitly addition of implicit privileges such as $E \bowtie S$ to the rules, or (b) by denying access to $E \bowtie S$ if the enterprises conclude

that the implicit access was really not intended. In this paper we focus only on (a) and rely on the enterprises to alter their rules suitably so that implicit accesses are no longer objectionable. We recognize that this may not always be possible. Individual implicit accesses can be denied by introducing trusted third parties as proxies. For example, if party P_E can operate on relation S only via a third party, it can no longer compute the full $E \bowtie S$. (As usual, some restrictions on how much data a query can retrieve is essential to ensure that a querier does not retrieve the entire relation.) Due to space limitations, a comprehensive treatment of third parties is outside the scope of this paper, and is addressed in an upcoming paper [20].

Given a set of access rules, we propose an algorithm to generate additional rules so as to remove all the conflicts. 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 due to fact that long chains of joins are rare in practice. In addition, the evaluation can be done prior to running queries, so its 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. The first type of change only affects the rules on a single party and the latter one involves multiple parties. Since any changes on the access rules may result in 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 of changes in order to 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. The main issue to address is to ensure that rule changes are introduced in such a way that minimum number of queries is affected.

Even if the set of access rules is conflict free, there are still many hurdles in properly implementing the access rules. Since the enterprises are allowed to specify an arbitrary set of access rules, it is possible that there is no way to derive a safe execution plan for queries allowed by certain rules. The simplest way to illustrate this problem is by considering the following situation: a rule specifies access to $R \bowtie S$ (where R and S are relations owned by two different parties); however, no party has access to both R and S and thus no party is able to do the join operation! The rules are consistent, but $R \bowtie S$ cannot be implemented anywhere. Thus, a basic problem is to determine enforceability of the given rules. If a rule is not enforceable, we should either remove it or make it enforceable. If not, this will cause problems for the queries. For instance, a query for the information of $R \bowtie S$ is authorized by the specified rule, but cannot be properly answered.

We address the rule enforcement checking problem in two steps. First of

all, we examine the enforceability of each access rule in a constructive bottom-up manner, and build a relevance graph that captures the relationships among the rules. In a collaborative environment, a rule can be enforced with not only the locally available information but also the remote information from the cooperative parties. If a rule is not totally enforceable, we consider two ways to deal with it. The first option is to remove the unenforceable part of the rule, so that only enforceable rules are retained. The second option is to modify related existing rules to make the inspected rule totally enforceable. We use the property of the graph to find the solution that has minimal impact on the existing rules.

The outline of the paper is as follows. Section 2 addresses the issue of consistency and enforcement of rules in a cooperative access environment. Section 3 describes an algorithm for rule consistency checking. Section 4 deals with the problem of changes in the rules. Section 5 describes the mechanism to check the rule enforceability. Section 6 discusses the algorithm modifying the rules to achieve rule enforceability. Section 7 discusses the related work. Finally, Section 8 concludes the discussion and lays out areas for future work.

2. Consistency of Access Rules and Rule Enforcement

In this section, we introduce basic concepts and the problems of access rule consistency and enforcement.

2.1. Preliminaries

In this work, we consider a group of cooperating parties, each of which maintains its data in a standard relational form such as Boyce-Codd Normal Form (BCNF). It is possible to consider more complex normal forms as well, but this is beyond the scope of this paper. We assume simple select-project-join queries, i.e., no cyclic join schemas or queries. 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 study the problems only involving the cooperating parties; 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 the order of joins in an actual implementation 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, an incoming query q can be represented as a pair $[A_q, J_q]$, which stands for the query attribute set and the query join path. The defined access rules are known by all the cooperating parties, and any party that has the sufficient permissions may answer the query. We also assume that each rule has all the key attributes of the basic relations in its join path.

2.2. A 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. In each of these relations, a single attribute happens to form the key, but this is not required in our analysis.

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

Below, 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 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).

We now define a set of access rules given to the party E as described in Table 1. (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.

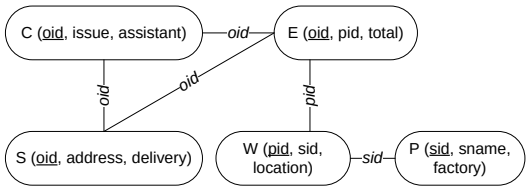

Figure 1: The given join schema for the example

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 1: access rules for e-commerce cooperative data access

2.3. 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.

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 discussed before, we choose to remove inconsistency by adding additional rules that allow for all potential compositions that have not been explicitly specified in this paper. In most cases, it is reasonable to allow the local computation results once the underlying information is authorized. For this, one must generate all possible compositions of the given rules and add any missing ones from the list. Therefore, 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 defined by 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 in a cooperative environment, the rule consistency property applies to each individual party separately. Thus, the mechanism for achieving consistent closure below only involves rules on one party.

2.4. Key attributes hierarchy

Since 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 and 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. 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 $R.K \rightarrow S.K$. If $R.K = S.K$, there is no hierarchy, and such key attribute of R and S is also the key attribute of the join result. For a given valid join path, the key attribute of the join path is always 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 and these join attributes form the hierarchal relationship. For instance, in the given example rules, 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. Since 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 **relevance graph**. Each node in such structure is a rule marked by its join path. Rules in such structure are put into different levels, and the level is determined by the length of its join path. Two nodes are connected if one is the relevant rule of

the other. For instance, Figure 2 shows a relevance graph. 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 as J_2 has length 2 and J_6 has length 3.

2.5. Query authorization and rule enforcement

When a query is given, it should be answered by one of the parties that have the authorization. Since our authorization model is based on attributes, any attribute appearing in the Selection predicate in an SQL query is treated as a Projection attribute. In other words, the authorization of a PSJ(Project, Select, Join) query is transformed into an equivalent Projection-Join query form. Therefore, a query q can be represented by a pair $[A_q, J_q]$, where A_q is the set of attributes appearing in the Selection and Projection predicates, and the query join path J_q is the FROM clause of an SQL query. For instance, there is an SQL query Q_1:

"Select $oid, total, address$ From E Join S On $E.oid = S.oid$ Where $delivery =$ 'ground'"

The query can be represented as the pair $[A_q, J_q]$, where A_q is the set $\{oid, total, address, delivery\}$; J_q is the join path $E \bowtie_{oid} S$.

Access rules define the set of queries that are authorized to retrieve information from the parties. A query q is called **authorized** if there exists a rule r_t such that $J_t \cong J_q$ and $A_q \subseteq A_t$. The join paths must be equivalent. Otherwise, the relation/view defined by the rule will have fewer or more tuples than the query asks for. Here we don't consider the situation where the projections on two different join paths get the same result (e.g., by joining on foreign keys) since data coming from different parties usually does not have foreign key constrains.

In fact, "authorized" is only a necessary condition for a query to be answered but not sufficient. To perform the required join operations to answer the query, we need to find appropriate parties that have the sufficient privileges to do these joins. Therefore, at least one legitimate query execution plan is required to answer a given query. A **query execution plan** or "query plan" for short, includes several ordered steps of operations over authorized and obtainable information and provides the composed results to a party. The result of a query plan pl is also relational, and it can also be presented with the triple $[A_{pl}, J_{pl}, P_{pl}]$. A valid query plan should be authorized by a given access rule r_t. A query plan pl answers a query q, if $J_{pl} \cong J_q \cong J_t$, $A_q = A_{pl} \subseteq A_t$ and $P_{pl} = P_t$. An access rule defines the maximal set of attributes that a query on the equivalent join path can retrieve. Thus, each rule can also be treated as a query. We call the query plan to enforce a rule as an **enforcement plan** or "plan" for short below.

Definition 4. *A rule r_t can be* totally enforced*, if there exists a plan pl such that $J_t \cong J_{pl}$, $A_t = A_{pl}$, $P_t = P_{pl}$. r_t is* partially enforceable*, if it is not totally enforceable and there is a plan pl that $J_t \cong J_{pl}$, $A_t \supset A_{pl}$, $P_t = P_{pl}$. Otherwise, r_t is not enforceable. A join path J_t is enforceable if there is a plan pl that $J_t \cong J_{pl}$.*

At the very beginning, only the rules indicating the data owners have their own data are known to be totally enforceable. As a plan contains steps bringing information together to enforce a rule, an enforcement plan can have following 3 operations over the enforceable information: A projection (π) is performed on a single party to select attributes; A join ($\bowtie$) operation is also performed at a single party, and it combines two pieces of information and generates information on a longer join path; Data transmission ($\rightarrow$) is an operation that happens between two parties, and one party sends information to the other. It is required that the two parties have two rules on the equivalent join paths and the information transmitted is based on such join path. In addition, the rule on the receiving party should have an attribute set that contains all the attributes of the information being transmitted. Otherwise, the transmission is not safe.

It is obvious that not all the rules are enforceable. Whether an enforcement plan exists depends on whether pieces of enforceable information on shorter join paths are available and whether they can be joined losslessly at some places. In cooperative environment, the enforceable information on remote cooperative parties may also be helpful to construct an enforcement plan. We will discuss the mechanism to check rule enforcement in later sections.

3. Consistency Checking Algorithm

To resolve the access rule inconsistency, we propose a rule consistency checking algorithm. Given a set of rules, the goal of the algorithm 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.

3.1. 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 equal to this attribute are called **optional** relations. Optional relations or join paths are associated with the dependent join paths. In relevance 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.

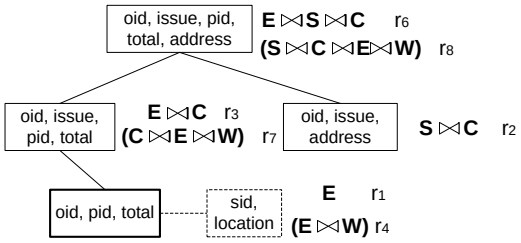

Figure 2: The consistent join group of *oid*

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 relevance graph 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 of 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 matching rules are being considered in the reverse order of join path length since the rule with longer join path includes all the attributes from its lower level 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. After the check, 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. Between 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 relevance graph, 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 is 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 it is listed in Table 2. 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 relevance graph 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 2: Generated consistent join group of *oid*

3.2. Iteration of key attributes

Based on the previous step, we take advantage of the key attributes hierarchy property to achieve the rule consistent closure. Since the key attribute hierarchy can be obtained based on the given join schema, 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 it 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 using the mechanism discussed above.

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. 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

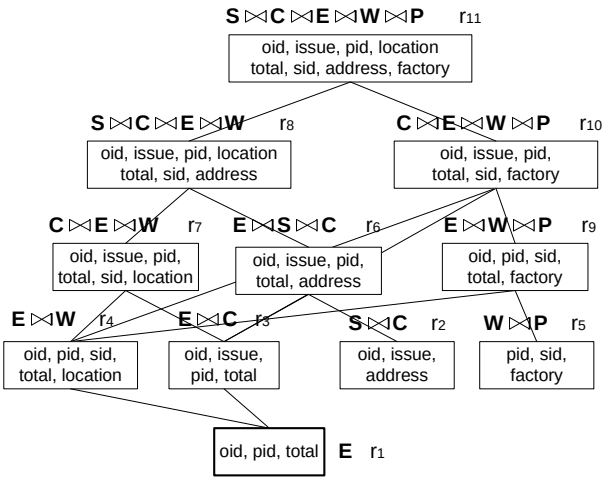

Figure 3: The relevance graph for the consistent closure.

attribute sets, and it adds these generated rules into the target rule set. If there is 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. At the time when rules are added to the target rule set, the algorithm also updates the relevance graph capturing the rule relevance relationships. If a new rule is generated, it is appended to the graph. The detail algorithm is described in Algorithm 1.

We 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 2 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. There is only one rule r_5 in the join group of *pid*, so this join group is already consistent. In the key attribute hierarchy, *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 3. The last three rules are generated in this process. Figure 3 shows the built relevance graph, 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.

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 3: Generated consistent closure based on given rule set

3.3. 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, there are $k - p$ nodes for initially given rules. As the largest number of different relations is 4, we have $k - p < 16$. Usually, k and p are small, and the number of rules in a consistent join group is less than 20 in most cases so that the complexity of generating it is low. We can consider 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 ($v < C$) including such dependent relation, then at most $v * C$ rules will be added at each iteration, 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 randomly chosen 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 can be given rules or the rules generated by the

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

```
1: Put rules from R into join groups based on their key
2: Put the key attributes of relations into a priority queue Q based on its level in hierarchy
3: for Each join group G do
4:     Generate the consistent join group G+
5:     for Length k ← 2 to 4 do
6:         Mark all rules unvisited
7:         for Each unvisited rule r_i length < k do
8:             if Exists r_m, where |J_j − J_i| + |J_i| = k then
9:                 Join r_i with r_m and get result r_j
10:                if There is no rule in R+ of join path J_j then
11:                    R+ ← r_j
12:                else
13:                    Get the rule and merge with r_j
14:                    R+ ← updated r_j
15:            Mark its relevant rules visited
16: while Q ≠ ∅ do
17:    Dequeue the key attribute, and get its associated G+
18:    if R+ ≠ ∅ then
19:        for Each rule r_r in R+ do
20:            if r_r includes the key attribute of G+ then
21:                for Each rule r_g in G+ do
22:                    Join r_r with r_g and get result r_j
23:                    if There is no rule in R+ of join path J_j then
24:                        R+ ← r_j
25:                    else
26:                        Get the rule and merge with r_j
27:                        R+ ← updated r_j
28:    R+ ← ⋃G+
```

algorithm. If r_i, r_j have the same key attribute, the two rules are in the same join group. When the algorithm generates the consistent join group, it tries all possible combinations of the dependent relations and optional relations are considered from bottom up, so r_k is always included in the generated consistent join group.

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 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 r_k is put into the target rule set. Therefore, all the rules are upwards closed, and the generated rule set is consistent. □

4. Consistent Access Rule Changes

Cooperative parties may change the access rules over time because of the evolving business needs. The change could either be grant more access privileges to a party or revoke some existing privileges. However, the change may cause new conflicts among the rules. A mechanism is needed to maintain the rule consistency while access rules are changed.

4.1. Access rule change

In general, a change of access rule that meet the new business requirement and also has minimal impact on the remaining access rules is the optimal solution. There are different factors can be take into consideration to best recover the rule consistency in the case of change. For instance, according to the business relationships, some access rules maybe more important than the others, so they may have different priorities. In such case, we always prefer to make changes on the less important rules first. Also, in a cooperative environment, some parties collaborate more intimately than the others, and there may also have priorities on different parties. Thus, it is preferred to grant privileges to the intimate parties and revoke privileges from the others. To keep the discussion simple, we propose our mechanism to find the solution that takes minimal changes to the existing access rules in terms of the number rules being modified. The priorities in access rules and parties can be considered by extending such a mechanism, and we leave them for future works.

A possible architecture for the authorization is that the access rules are stored at a central place different from any cooperative parties. An independent query optimizer then read the access rules and generates the query plans. In some cases, cooperative enterprises do not typically share a single independent query optimizer. Then, each party that answers the queries has to generate the query plan locally. Without a centralized party, each cooperative party should keep a copy of all access rules locally. We discuss rule changes under centralized model below, and the problem under de-centralized model can be solved by taking care of the synchronization issues among the cooperative parties.

4.2. Consistently grant more information

When more access privileges are granted to a party, we need a mechanism to maintain the rule consistency. There are two types of grants. The first is adding non-key (non-join) attributes to a rule. If a rule is granted with more attributes, then the algorithm examines the higher level relevant rules of the rule in the graph. We search upwards in the graph, and this can be done with a depth first search. 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 at a higher level from 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_{11} on the same path 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. 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 to maintain rule consistency.

In addition to that, there is another type of change, where a rule with 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 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 rules 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_i, the algorithm checks the join attributes in its join path (excluding its key attribute), and for each join attribute the algorithm combines r_i 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 relevance graph, 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. Table 4 lists these newly added rules.

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

Rule No.	Authorized attribute set	Join Path	Party
12	{oid, pid, total, address}	$E \bowtie_{oid} S$	P_E
13	{oid, pid, total, address, sid, location}	$S \bowtie_{oid} E \bowtie_{pid} W$	P_E
14	{oid, pid, total, address, sid, location, factory}	$S \bowtie_{oid} E \bowtie_{pid} W \bowtie_{sid} P$	P_E

Table 4: Access rules added together with a rule grant change

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 $O(C * u * s)$.

4.3. 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, the revocation can range from removing some non-key attributes to complete removal of a rule. We first discuss the situation where non-key attributes are revoked. The revocation of attributes usually causes inconsistency. Since its relevant rules may still have the revoked attribute, the party can still access these attributes through local computation. Therefore, we need to also revoke these attributes from all relevant rules. Based on the built relevance graph, 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.

If a rule with a join path is completely revoked from the rule set, we need to make sure that such a 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, only the **direct relevant rules** of the revoked rule r_v in the relevance graph are examined. The direct relevant rules of r_v are the relevant ones in the graph that directly connected to r_v with one edge. For each of the direct relevant 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 which will be discussed in later sections, we assume we already know what are the locally enforceable

rules. For each pair of rules being 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 appearances of the rules. We prefer to remove the rule with most appearances since removing one such rule can break most pairs. In worst case, half of the existing rules need to be removed from the rule set. 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. In addition, 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 as well. Since 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.

The above mechanism to remove a rule is complicated and it considers only one next level of rules. Thus, we also propose removing the rules in another way. Usually, a revocation is issued by a single party, and this party may revoke the access rules with its own data from a certain party. When a revocation is issued, it is reasonable for the party to revoke all the rules including its information. In such a case, the revocation involves a set of rules that all including the same basic relation, and the consistent closure is still maintained. Following this idea, if we want to remove a rule, we can remove a set of rules containing the same basic relation. The algorithm can first obtain all the relevant rules of r_v. For relevant rules, the algorithm records the numbers of appearances of the basic relations in the join paths. The basic relation associated with least number of rules is then selected, and rules including this basic relation are all removed from the set. Using our example, suppose that we want to revoke rule r_{10}. This mechanism first retrieves its relevant rules which are $\{r_4, r_5, r_9, r_3, r_1\}$. Then the appearances of 4 basic relations are checked and counted. Relation C appears once, E appears 4 times, W appears 3 times, and P appears twice. Thus, the algorithm tries to remove the rules whose join path has C. Thus, r_3 is removed, and this result is the same as the previous algorithm for this example. In general, these two mechanisms produce different results.

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.

5. Checking Rule Enforcement

As discussed before, even though the rules are consistent, we still need a mechanism to check if each individual rule can be enforced among the existing

parties. In this section, we first introduce some results, and then we present the algorithm that checks the enforceability of each given access rules. To enforce an access rule, a query plan is required, and we call a plan as **joinable plan** if it contains all the key attributes of the basic relations in its join path. In some cases, a rule does not have a total enforcement plan, but only some partial plans. A partial plan only enforces a rule with an attribute set that is a subset of the rule attribute set. We say that an attribute set is a **maximal enforceable attribute set** for a rule, if it is enforced by a plan of the rule, and there is no other plan of the same rule that can enforce a superset of these attributes. If a rule is totally enforceable, its maximal enforceable attribute set is the rule attribute set. Each rule has only one maximal enforceable attribute set.

Lemma 1. *A rule has only one maximal enforceable attribute set.*

PROOF. A rule defined on basic relation has one maximal enforceable attribute set. To get the maximal attribute set of a rule, we do not eliminate any attributes via projections and data transmission in the enforcement plans. If a rule is not totally enforceable, it can have several partial plans. These plans are on the same join path so they can be merged into one via join operations. Thus, a partially enforceable rule has one maximal enforceable attribute set. If a rule is not enforceable, it has empty enforceable attribute set. □

5.1. A new set of example access rules

When discussion the consistency problem, we check the rules on each party individually. In contrast, the enforcement of a rule may require the collaboration between the parties by exchanging authorized information. Hence, we give a new set of access rules as the example to illustrate the enforcement checking process. The set of rules are listed in Table 5, and given rules are already consistent. The join schema is the same as the one used in previous example.

5.2. Enforcement checking mechanism

In this section, we describe the rule enforcement checking mechanism, and it can tell which rules can be enforced and what are their maximal enforceable attribute sets. That also gives the answer of what are the set of authorized queries that can be safely answered. We have two options with the given rules that are not enforceable. The first choice is that we keep only the found enforceable rules with their maximal enforceable attribute sets, and rules that are not enforceable as well as the unenforceable attributes are removed from the rule definitions. In other words, the algorithm finds all the information that can be safely retrieved according to the given set of rules, and all inaccurate and unenforceable definitions are removed. This solution can be thought as a conservative one since it prohibits some authorized information to be released because of the enforceability. In contrast, we can also modify the rule configurations in an aggressive way. In such scenario, we think all the information regulated by the rules is authorized, and authorized information should be retrievable. Whenever any information in the defined rules cannot be enforced,

Rule No.	Authorized attribute set	Join Path	Party
1	{pid, location}	W	P_W
2	{oid, pid}	E	P_W
3	{oid, pid, location}	$E \bowtie_{pid} W$	P_W
4	{oid, pid, total}	E	P_E
5	{oid, pid, total, issue}	$E \bowtie_{oid} C$	P_E
6	{oid, pid, total, issue, address}	$S \bowtie_{oid} E \bowtie_{oid} C$	P_E
7	{oid, pid, location, total, address}	$S \bowtie_{oid} E \bowtie_{pid} W$	P_E
8	{oid, pid, issue, assistant, total, address, delivery}	$S \bowtie_{oid} E \bowtie_{oid} C \bowtie_{pid} W$	P_E
9	{oid, address, delivery}	S	P_S
10	{oid, pid, total}	E	P_S
11	{oid, pid, total, address, delivery}	$E \bowtie_{oid} S$	P_S
12	{oid, pid, total, location}	$E \bowtie_{pid} W$	P_S
13	{oid, location, pid, total, address, delivery}	$S \bowtie_{oid} E \bowtie_{pid} W$	P_S
14	{oid, pid}	E	P_C
15	{oid, issue, assistant}	C	P_C
16	{oid, pid, issue, assistant}	$E \bowtie_{oid} C$	P_C
17	{oid, pid, issue, assistant, total, address, location}	$S \bowtie_{oid} C \bowtie_{oid} E \bowtie_{pid} W$	P_C

Table 5: Access rules for e-commerce cooperative data access

we change the rule configurations by granting more privileges so as to make this information enforceable. Since there are different ways to modify the rules, we prefer to find the way that has minimum impact on the existing rules. That is, we try to find the minimum amount of information to release.

To that end, we first propose a constructive mechanism that checks the rules in a bottom-up manner. In general, an enforcement plan for a rule combines pieces of information available and generates the information authorized by the rule. For each rule, the mechanism checks its relevant information locally and remotely and indicates if it can be enforced and what is its maximal enforceable attribute set. The set of unenforceable attributes and the unenforceable rules are removed from the rule set. Since this algorithm works as the first option we discussed above, we also provide the algorithm for the second option. If the inspected rule cannot be totally enforced, we modify rules to make it enforceable. We discuss the possible ways to define what is the minimal impact on the current set of rules. We present the first algorithm below, and then we describe the second algorithm. In fact, the second algorithm just introduces extra steps into the first one when inspecting a given rule, and all the other steps are the same. Therefore, we discuss only these additional steps for the second algorithm.

5.3. Finding enforceable information

When examining a rule $[A_t, J_t, P_t]$, we call such a rule r_t as *Target Rule*, the attribute set A_t as *Target Set*, the join path J_t as *Target Join Path*, and the party P_t in the rule as *Target Party*. All the other parties are *Remote Parties*. To check the enforceability of r_t, we first find the relevant information that

can be obtained locally at P_t. If this is not enough, we check the information from remote parties. To enforce a rule with a long join path, we always need to retrieve information from relevant rules with short join paths. Thus, the algorithm works in the order of join path length, and it begins with rules on basic relations (length of 1 rules). As the mechanism works bottom-up, when examining a target rule with join path of length n, we can assume that all the rules on join paths with shorter lengths have already been examined, and only the maximal enforceable attribute sets of the rules are preserved.

Since the first task is to identify relevant information locally, we check the rules relevant to r_t at P_t. At party P_t, a joinable plan that is on a sub-join path of J_t is a *Relevant Plan*. Parties that have rules defined on the equivalent join path of J_t are called **J_t-cooperative parties**, and information on J_t is allowed to be exchanged only among these parties by data transmission operations. For instance, P_E and P_S are J_{13}-cooperative parties since $J_{13} \cong J_7$. We assume that each inspected rule is represented by an enforcement plan. When inspecting the target rule, we consider using these plans to enforce it. We say "join among rules" below, which means their enforcement plans.

The rule with join path length of 1 can be totally enforced by sending the data from owners to the authorized parties. From then on, the algorithm checks for rules defined on longer join paths. At the same time examining the rules, the algorithm also builds a relevance graph similar to the one discussed in the previous algorithm. Each node in such structure is a rule with its maximal enforceable attribute set. The only difference is nodes belong to different parties can be connected if they have the equivalent join paths. Figure 4 shows the built structure for our running example. The different parties are separated vertically. The bold boxes show the basic relations owned by different parties. The algorithm starts the iteration with the rules on basic relations r_1, r_2, r_4, r_{10}, and so on. As the algorithm iteratively checks all the rules, when a target rule r_t is examined, the algorithm first checks whether the join path J_t can be enforced using relevant rules on P_t. After that, all the rules with equivalent join path of J_t are checked respectively at J_t-cooperative parties. Then the algorithm checks the possible enforcement by exchanging information among these parties. In Figure 4, on the level of join path length 2, the algorithm checks the rules with the order of r_3, r_{12}, r_5, r_{16}, r_{11} because $J_3 \cong J_{12}$ and $J_5 \cong J_{16}$. J_t-cooperative parties such as P_W and P_S on J_3 will check the remote enforcement between r_3 and r_{12}, which will be described later.

To check local enforceability, the algorithm finds its local relevant rules in the currently built relevance graph since all its relevant rules have already been examined and added to the graph. It only checks with the *top level relevant rules* in the current graph, where top level rules are the nodes not connected to any higher level nodes (rules with longer join paths) in the currently built graph during the bottom-up procedure. In Figure 4, when the algorithm examines r_{13} on P_S, only r_{11}, r_{12} are top level rules. And when checking r_8, r_7 and r_5 are top level rules since r_6 is not enforceable. Here, we take advantage of the upwards closed property of the rules, so that the top level rules cover all possible join results among the lower level rules. If these top level rules cannot be composed

to enforce the J_t, there is no need to check lower level rules. When examining r_{13}, there is no need to consider the join between r_9 and r_{10}. Among the rules in the graph on P_t, a relevant rule r_r of r_t can be efficiently decided, if $JR_r \subset JR_t$.

The following step is to check whether the join path J_t can be enforced locally by performing joins among these top level relevant rules. The algorithm basically checks each pair of these rules. We check it pairwise because if a pair of them can join, the result must be able to enforce J_t. Otherwise, there must exist another relevant rule of r_t authorizing the join result, and such a rule is on higher level of the pair of rules being inspected, which is contradict to the fact that the pair of rules are top level rules. When checking whether a pair of rules (r_s, r_r) can join, the algorithm first tests their relation sets to see if $JR_s \bigcap JR_r = \emptyset$. If these two join paths have overlapped relations, they can join over the key attribute of the overlap part, and J_t can always be enforced. Otherwise, we need to further check the attributes of two rules to see if they have the required join attribute in common. If J_t can be locally enforced, we mark the target rule as **local enforceable** rule and add it to the graph by connecting it with top level relevant rules. Otherwise, it has to wait and see if J_t can be enforced on other parties. For instance, when checking r_3 in our example, it has top level relevant rules r_1 and r_2, since there is no overlapped relation for the pair of rules, the algorithm checks whether join attribute *pid* can be found in both rules. On the other hand, when checking the pair r_{11} and r_{12}, as E is the overlapped relation, the join path J_{13} can be locally enforced. r_{17} does not has a valid join pair, and it is not locally enforceable. Once a pair is found to enforce the join path, the algorithm proceeds to next steps.

Meanwhile, the algorithm also computes the union of the attributes from top level relevant rules regardless of the enforceability of J_t. The resulting attribute set A_r includes all attributes that can be obtained from party P_t if J_t can be enforced. It is always the case that $A_r \subseteq A_t$ as rules are upwards closed. If A_r not equals to A_t, we call the set of attributes $A_t \setminus A_r$ as **missing attribute set** A_m. The attributes in A_m are potentially obtainable from the J_t-cooperative parties. In the example, the attribute *delivery* in r_8 cannot be found in its top level rules r_7 and r_5, and it is a missing attribute after the local checking.

Next, the algorithm checks the remote information that a party can use to enforce a rule, and only J_t-cooperative parties are checked. As the previous steps of the algorithm tell which parties can locally enforce the join path J_t, if there exists any party that can enforce J_t, then all the J_t-cooperative parties can have joinable plans for their rules on J_t. Thus, the party P_t is able to get attributes from all its J_t-cooperative parties to enforce r_t. For instance, r_{17} is not locally enforceable, but J_8 can be enforced with a joinable plan at P_E. Thus, we can add a data transmission operation to such plan, and r_{17} also has a joinable plan. This plan can join with r_{16}, so that attributes $issue, assistant$ in r_{17} can be enforced. Consequently, these attributes in r_8 can also be enforced. Therefore, we take the union of the attribute sets from all J_t-cooperative parties to check if r_t can be totally enforced. If the missing attribute set $A_m \subset A_{r_1} \bigcup A_{r_2} ... A_{r_k}$ (where A_{r_i} is the relevant attribute set of a J_t-cooperative party P_i), then r_t can be totally enforced. Otherwise, A_m is updated by removing the attributes

Algorithm 2 Rule Enforcement Checking Algorithm

Require: All given access rule set R on all parties
Ensure: Find enforceable rules and build graph
Mark rules with length 1 as total enforceable rules
Get the maximal length of join path length N
for Join path of length 2 to N **do**
  for Each join path J_t length equal to i **do**
    $AJ_t \leftarrow \emptyset$, the set of shared attributes on J_t
    for Each party P_t has a rule r_t on J_t **do**
      Obtain the set of top level relevant rules R_v
      Add the node and connections to R_v in graph
      $A_v \leftarrow$ the union of attributes in R_v
      Missing attribute set $A_m \leftarrow A_t$
      for Each pair of relevant rule (r_s, r_r) **do**
        if The pair can locally enforce J_t **then**
          $A_m \leftarrow A_m \setminus A_v$ and break
      if $A_m \neq \emptyset$ **then**
        Put r_t with A_m into the Queue of J_t
      $AJ_t \leftarrow AJ_t \bigcup A_v$
    for Each rule r_t in the Queue of J_t **do**
      if J_t can be enforced on some party **then**
        Add connections among J_t-cooperative parties in graph
        $A_m \leftarrow A_m \setminus AJ_t$
        if $A_m \neq \emptyset$ **then**
          Replace A_t with $A_t \setminus A_m$ in graph
      else
        r_t cannot be enforced, remove rules on J_t from graph
  Join path length i++

appear in any A_{r_i}. In such case, r_t has a maximal enforceable attribute set on J_t without the attributes in A_m. The node r_t in the relevance graph is presented with the attribute set $A_t \setminus A_m$. Meanwhile, **connection edges** are added among the J_t-cooperative rules in the relevance graph. For example, attribute *delivery* of r_8 also cannot be found in its J_t-cooperative party P_C, so it cannot be enforced. r_8 in the graph is represented with the attribute set without *delivery*. We use bold font in Figure 4 to indicate this attribute is not enforceable. Also, since join path J_6 cannot be enforced at any party, r_6 is not enforceable, and it will not be included in the relevance graph. In Figure 4, we use the dashed box to show r_6 is removed. The local enforceable rules are marked with "L". The detailed algorithm is described in Algorithm 2.

In algorithm 2, each rule will be examined at most twice, with one local enforceability check and another one in checking the queue of J_t. In the step of local enforcement checking, only the top level relevant rules on party P_t are checked. Suppose that the total number of rules is N_t, the maximal number of relevant rules of a rule is N_o, and checking join condition takes constant C. Then the worst case complexity for algorithm 2 is $O(N_t * N_o^2 * C)$, where N_o is usually small. In addition, this algorithm can be used as a pre-compute step once all the rules are given.

Theorem 2. *The* **Rule Enforcement Checking Algorithm** *finds all enforceable information.*

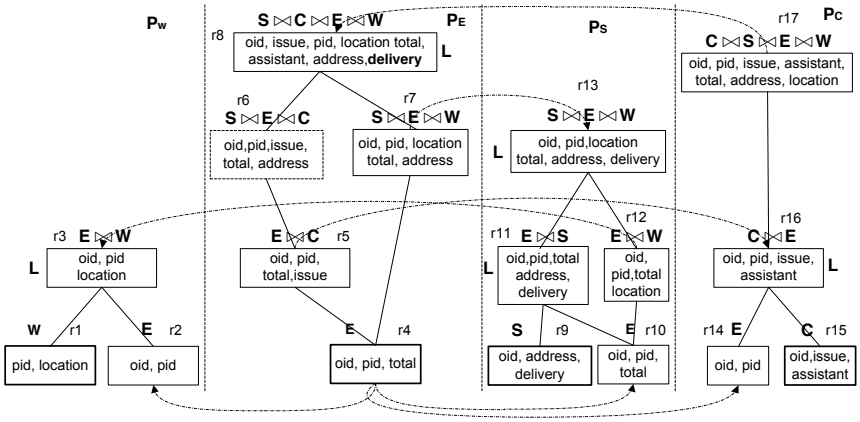

Figure 4: relevance graph built for the example

PROOF. As all the information on the join results comes from the basic relations, the algorithm works in bottom-up manner to capture the operation results. If the join path of a rule cannot be enforced, then rules on this join path cannot be enforced. The algorithm first finds a way to enforce the join path of the rule r_t, and it explores all possible ways to compose useful information on P_t. Since the only other information can be used to enforce r_t must come from J_t-cooperative parties, the algorithm also considers all the attributes that r_t can get from them. There is no other way to enforce more attributes for r_t. □

6. Augmenting Authorizations for Enforceability

In this section, we discuss the mechanism that modifies the rule definitions by granting more privileges to enforce the unenforceable rules. We can add such steps into the previous algorithm after concluding that the inspected rule cannot be totally enforced. Such a rule could be either partially enforceable or its join path is not enforceable at all. In the former case, according to Lemma 1, the missing attributes are all non-key attributes in the underlining basic relations. Therefore, there is no need to add new rules, and expanding the attribute sets of existing rules can make the rule totally enforceable. For the latter case, new rules must be added to make the join path enforceable. The problem of deciding new rules to add is extremely complex for a number of reasons. First, an addition could violate the consistency of the rules, and we need to maintain the rule consistency using the aforementioned algorithms. Second, an automated addition of a rule may be undesirable, and we may need to look for some manually assisted or guided process. Third, it would desire to define some notion of minimality in adding new rules that is nontrivial. In view of these challenges, we reserve this case for future work.

Thus, our goal is to grant more attributes to existing rules so as to turn partially enforceable rules into totally enforceable rules while maintaining min-

imal impact on the existing rules. Here we define the impact as the amount of new information granted to the parties, which can be measured in two slightly different ways as explained below.

We first consider the minimal impact in terms of the number of attributes being added to the different rules. Referring to the graph we built, we can turn the problem of minimal attribute addition into a shortest path problem with path length measured by the number of edges. We do it for each attribute M_i in the missing attribute set respectively. A shortest path from the target rule r_t to a rule with missing attributes M_i gives the minimal number of rules. For this, we perform a breadth first search starting from r_t where each visited node records its parent node. Once a rule has M_i is found, the breadth first search stops and the path between the two nodes is selected. For each node on the selected path, the attribute M_i is added to the corresponding rule.

Since the goal is to make the attribute M_i enforceable in r_t, the algorithm only checks the rules on the sub-paths of J_t and include the relation that M_i comes from. When r_t is inspected, it is at the top level of the graph, so the search is performed top-down. Each next visited node must have the join path length no longer than the current node. It is because that a plan has longer join path cannot be used in a plan for a shorter join path by valid operations, and the information of M_i cannot be transmitted from a higher level node to a lower one. All these selection conditions make the search more efficient than the general case. For instance, since rule r_8 has missing attribute *delivery*, the search begins from r_8. The search gives the shortest path r_8, r_7, r_{13}, and r_8 becomes totally enforceable by adding *delivery* to r_7.

However, because of the upwards closed property of the rules, once M_i is released to a party on a rule with shortest join path, all the rules that this rule is relevant to should also have M_i added to them. Since rules with longer join paths have not been examined by Algorithm 2 yet, it is difficult to know the exact number of attributes that will ultimately be added. That is, the above algorithm ensures minimality only in terms of initial addition, not the ultimate one. It has the worst case complexity the same as breadth first search which is related to the number of nodes and edges.

From another point of view, we can consider that once an attribute is granted to a party, this party will have the privilege to access such attribute. Therefore, no matter how many rules on this party are expanded with this attribute, we only count them as one attribute release. In this sense, our algorithm does accomplish its stated goal. The detailed algorithm is described in Algorithm 3. Different from the previous algorithm which stops when a rule having the attribute M_i is found, we do the breadth first search for all the qualified rules. Whenever a new rule r_n is visited, the algorithm checks if the party P_n has been visited before. If this is the first time P_n being visited, the algorithm records the rule r_n associated with the party P_n indicating a shortest path from P_n to P_t is found. After the breath first search, the parties that have the associated rules are examined. For each found shortest path from P_n to P_t, the algorithm counts the number of different parties along such path. At last, the path with the minimal number of parties will be selected as the best way to

Algorithm 3 Rule Enforcement Algorithm with Minimal Parties

Require: A partially enforceable rule r_t with missing attribute set A_m
Ensure: r_t is totally enforced with modified rules R'

```
1: Get the relevance graph G up to r_t
2: for Each M_i in A_m do
3:     Create a queue Q
4:     Enqueue r_t onto Q
5:     while Q is not empty do
6:         r_i ← Q.dequeue()
7:         if r_i has attribute M_i then
8:             if P_i is unvisited then
9:                 Mark P_i visited, record with r_i
10:        for Each edge e that includes r_i do
11:            r_n ← e.opposite(r_i)
12:            if r_n is unvisited && J_n is a sub-path of J_t &&
               JR_n ≤ JR_i && J_n includes the relation of M_i then
13:                Mark r_n visited, record r_i as its parent
14:                Enqueue r_n onto Q
15:    for Each visited party P_i except P_t do
16:        Get r_i associated with P_i
17:        Count the number of parties on path from r_i to r_t
18:        Keep the minimal party P_m
19:    Get r_m associated with P_m
20:    while r_m ≠ r_t do
21:        r_m ← parent(r_m)
22:        Add M_i to r_m
```

modify the rules, and rules on this path are expanded with the attribute M_i. All these modified rules are recorded. Similar to the previous discussion, when Algorithm 2 examines rules on longer join paths, if a modified rule is relevant to the rule being inspected, such a rule is also modified so as to include the missing attributes into its attribute set. The algorithm has the complexity of $O((|E|+|V|)*|A_m|)$, where $|E|$ is the number of the edges and $|V|$ is the number of nodes in the relevant subgraph, and $|A_m|$ is the number of missing attributes.

In the example, r_8 has the missing attribute *delivery*. After the breadth first search, only party P_S is visited, and the associated rule is r_{13}. Following the path from r_{13} to r_8, only r_7 needs to be updated with the attribute *delivery*.

7. Related Work

The problem of controlled data release among distributed collaborating parties has been studied in [11]. The authors propose an efficient and expressive form of authorization rules defined on the join path of relations. They devise an algorithm to check if a query with given query plan tree can be authorized using the explicit authorization rules. It assumes all the given rules are already upwards closed. However, this is not the case in reality since access rules are usually formulated 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 [10], 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 combination of rules and assume 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 authorizations are explicitly given. Data release is prohibited if there is no explicit authorization.

Processing distributed queries under protection requirements has been studied in [6, 13, 21]. In these works, data access is constrained by limited access pattern called binding patterns, and the goal is to identify the classes of queries that a given set of access patterns can support. These works only consider two subjects, the owner of the data and a single user accessing it, whereas the authorization model considered in our work involves independent parties that may cooperate in the execution of a query. There are also classical works on query processing in centralized and distributed systems [4, 17, 9, 25], but they do not deal with constraints from the data owners.

There are some works on the access control in collaborative environments. In [26], the authors examined existing access control models as applied to collaboration, and point the weaknesses of these models. In addition, [15, 23] applied RBAC in the collaborative environments. [8] discussed the access control problems in the popular social networks. [12] proposed a web services based mechanism for access control in collaboration situations. All these access control models are different from the one we are using. In [22], collaboration among enterprises was also studied, but that work focused on different application data and multilevel policies. The given access rules are also similar to the firewall rules, which indicates what types of queries can go through. As firewall rules are needed to be enforceable and accurate [3, 27], we have the same requirements in our situation.

Our authorization model is similar to the view based authorization, and it is related to the area of answering queries using views [16, 14, 24]. These techniques are useful for query optimization, data integration and so on. Although the given view definitions in these works is similar constraints to our access rules, they consider the queries and views in the form of conjunctive queries and they do not consider the collaboration relationships among different parties. These make our problem different from these works, and we may investigate our problem with conjunctive authorization model in the future. There are several services such as Sovereign joins [1] to enforce the access rule model we used, such a service gets encrypted relations from the participating data providers, and sends the encrypted results to the recipients. Join processing in outsourced databases is also discussed in [28, 7]. These methods are useful to enforce our access rules, but we discuss the problem without any involvement of third parties.

This paper is primarily based on our work on the rule consistency problem that was presented at the CollaborateCom conference [18], and we integrated it with additional materials on rule enforcement issues which were presented at the SafeConfig Symposium [19] (no proceedings).

8. 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 conflicts may arise among the rules made according to business requirements. In this paper, we proposed a mechanism to make the set of cooperative access rules consistent, and we also presented algorithms to maintain the rule consistency in the case of granting and revocation of access privileges. Since access rules are made based on business requirements, it is possible that some rules cannot be enforced among the cooperative parties. Therefore, we also proposed an algorithm to check the enforceability of the given rules among cooperative parties as well as the mechanisms to make rules totally enforceable by modifying the rule definitions.

In the future, we will look for mechanisms that maintain the rule consistency by removing some of the rules instead of simply adding rules. The Chinese wall policy [5] and access via third parties provides two ways of enforcing this. Moreover, we will further look into the more dynamic situation where not only the rules are changed from time to time, but also parties can join and leave the cooperative environment at different times. For the rule enforcement issue, we will study the problem where a trusted third party is available. In such a scenario, the problems of how to enforce the rules without modifications and what are the optimal ways to enforce such rules need to be investigated. We will also study how to efficiently handle select operations in our theoretical development and algorithms.

References

[1] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In *Proceedings of the 22nd International Conference on Data Engineering, (ICDE'06)*, 2006.

[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] E. Al-Shaer, A. El-Atawy, and T. Samak. Automated pseudo-live testing of firewall configuration enforcement. *IEEE Journal on Selected Areas in Communications*, 27(3):302–314, 2009.

[4] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, Dec. 1981.

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

[6] A. Calì and D. Martinenghi. Querying data under access limitations. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancun, Mexico*, pages 50–59, 2008.

[7] B. Carbunar and R. Sion. Toward private joins on outsourced data. *IEEE Transactions of Knowledge and Data Engineering(TKDE)*, 24(9):1699–1710, 2012.

[8] B. Carminati and E. Ferrari. Collaborative access control in on-line social networks. In *Proceedings of 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011*, pages 231 –240, oct. 2011.

[9] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM symposium on Principles of database systems(PODS'98)*, pages 34–43, 1998.

[10] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Assessing query privileges via safe and efficient permission composition. In *Proceedings of the 15th ACM conference on Computer and communications security(CCS '08)*, pages 311–322, 2008.

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

[12] A. El Kalam, Y. Deswarte, A. Baina, and M. Kaaniche. Access control for collaborative systems: A web services based approach. In *Proceedings of IEEE International Conference on Web Services(ICWS'07)*, pages 1064 –1071, july 2007.

[13] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 41–60, 1999.

[14] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data(SIGMOD'01)*, pages 331–342, 2001.

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

[16] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

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

[18] M. Le, K. Kant, and S. Jajodia. Access rule consistency in cooperative data access environment. In *8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2012.

[19] M. Le, K. Kant, and S. Jajodia. Rule configuration checking in secure cooperative data access. In *5th Symposium on Configuration Analytics and Automation (SafeConfig)*, 2012.

[20] M. Le, K. Kant, and S. Jajodia. Rule enforcement with third parties in secure cooperative data access. In *27th IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec)*, 2013.

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

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

[23] J. S. Park and J. Hwang. Role-based access control for collaborative enterprise in peer-to-peer computing environments. In *Proceedings of the eighth ACM symposium on Access control models and technologies(SACMAT '03)*, pages 93–99, 2003.

[24] R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, 2001.

[25] S. H. Roosta. Optimizing distributed query processing. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*, pages 869–875, Las Vegas, Nevada, USA, June 2005.

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

[27] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[28] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the ACM Conference on the Management of Data (SIGMOD'09)*, 2009.