SECURE COOPERATIVE DATA ACCESS IN MULTI-CLOUD ENVIRONMENT

by

Meixing Le
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____	Dr.Sushil Jajodia, Dissertation Co-Director
_____	Dr.Krishna Kant, Dissertation Co-Director
_____	Dr.Angelos Stavrou, Committee Member
_____	Dr.Massimiliano Albanese, Committee Member
_____	Dr.Duminda Wijesekera, Committee Member
_____	Dr.Stephen Nash, Senior Associate Dean
_____	Dr.Kenneth S. Ball, Dean, The Volgenau School of Engineering

Date: _____ Spring Semester 2013
George Mason University
Fairfax, VA

Secure Cooperative Data Access in Multi-Cloud Environment

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Meixing Le
Master of Science
Fudan University, 2007
Bachelor of Science
Fudan University, 2004

Co-Director: Dr.Sushil Jajodia, Professor
Co-Director: Dr.Krishna Kant, Professor
Center for Secure Information Systems

Spring Semester 2013
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my wonderful family for their continuous support.

Acknowledgments

I would like to thank the following people who made this possible.

First, I would like to express my deepest gratitude to my advisors, Professor Sushil Jajodia and Professor Krishna Kant. I would like to thank Professor Sushil Jajodia, who gave me the opportunity to work in the Center for Secure Information Systems (CSIS), advised me and financially supported me through my whole Ph.D study. His wisdom, knowledge and commitment inspire and motivate me. I am also very grateful to my co-adivsor, Professor Krishna Kant. During these years, he patiently provided the vision, encouragement and advice for me to go through my research. He taught me how to find, analyze and solve the research problems. I thank him for his great helps.

I would like to thank Professor Angelos Stavrou. During the first several years of my study, he taught me how to do research. I accumulated hands on experiences on system and network security under his advice. I would like to thank my other committee members: Professor Massimiliano Albanese and Professor Duminda Wijesekera. They spent time and provided me with valuable advice.

I thank my colleagues who graduated from or currently in the Center for Secure Information Systems: I thank Professor Brent Kang for his valuable suggestions. I thank my friends Quan Jia, Zhan Wang, Dr. Zhaohui Wang and Li Wang for many helpful discussions for research ideas. Special thanks go to Patricia Carcamo who helped me on administrative issues.

Table of Contents

List of Tables

List of Figures

Abstract

SECURE COOPERATIVE DATA ACCESS IN MULTI-CLOUD ENVIRONMENT

Meixing Le, PhD

George Mason University, 2013

Dissertation Co-Director: Dr.Sushil Jajodia

Dissertation Co-Director: Dr.Krishna Kant

In this dissertation, we discuss the problem of enabling cooperative query execution in a multi-cloud environment where the data is owned and managed by multiple enterprises. Each enterprise maintains its own relational database using a private cloud. In order to implement desired business services, parties need to share selected portion of their information with one another. We consider a model with a set of authorization rules over the joins of basic relations, and such rules are defined by these cooperating parties. The accessible information is constrained by these rules. It is assumed that the rest of the information is well protected but those mechanisms are not addressed here.

It is expected that the authorization rules are formulated based on business needs and agreements, and may suffer from several issues. First, the rules may be inconsistent in that they release more information than the parties may realize or agree to. We formalize the notion of consistency of authorization rules and devise an algorithm to augment rules to maintain rule consistency. We also consider the possibility of occasional changes in authorization rules and address the problem of maintaining consistency in the face of such changes. We propose algorithms for both changes with new privileges grants and revocations on existing privileges. Instead of augmentation, conflicts may be resolved by introducing

negative rules. We discuss the mechanism to check if the negative rules can be violated and the possible way of enforcing them.

The second issue is that the parties may possess inadequate access to basic data to implement the operations required for providing the stated access to the composed data. In other words, the rules cannot be enforced or implemented in reality. Therefore, we propose an algorithm to systematically check the enforceability for each given authorization rule in order to determine the set of queries that can be safely executed. We also present mechanisms to generate a query execution plan which is consistent with the authorization rules for each incoming authorized query. Since finding the optimal query plan can be very expensive, our algorithm attempts to find good query plans using a greedy approach. We show that the greedy approach provides plans that are very close to optimal but can be obtained with a far lower cost.

The third issue to consider is handling of situations where rules cannot be enforced among existing parties. For this, we propose the introduction of trusted third parties to perform the expected operations. Since interactions with the third party can be expensive and there maybe risk of data exposure/misuse, while the data is held by third party, it is important to minimize their use. We define a cost model and formulate the minimization problem. We show that this problem is NP-hard, so we use greedy algorithms to generate solutions. With extensive simulation evaluations, the results show the effectiveness of our approach. Furthermore, we discuss different types of third parties, and the need for multiple third parties. We examine the problem of how to use minimal number of third parties to meet the given security requirements. This problem thus out to be strongly related to the graph coloring problems. We propose some heuristics to find near optimal answers.

Chapter 1: Introduction

1.1 Secure cooperative data access

Enterprises increasingly need to collaborate to provide rich services to clients with minimal manual intervention and without paper documents. This requires the enterprises involved in the service path to 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. To achieve collaborative computation, one scenario could be that needed information is shared among all the parties. In such a scenario, each individual party manages its own data, but one party can always access the data of another party. However, from data security point of view, sharing information with all parties releases more information than needed, and it is not desired by the data owners in many cases. Usually, data owners release their data to other parties only based on their collaboration requirements. In such environments, data must be released only in a controlled way among cooperative parties, subject to the authorization policies established by them. The data authorization should be flexible and fine-grained so that the data owners can easily determine which portion of the information is released to which party. On the other hand, the rest of the information that is not released needs to be protected. In the rest of the chapter, we will discuss the model of the cooperative data access, the problems in such an environment and the contributions of this thesis.

1.1.1 Cooperative data access model

Without loss of generality, we assume each collaborative party or enterprise maintains its own data in its private cloud. Such a party may have its own data center running the private cloud or possibly running the cloud on infrastructures rented from a provider. We assume that all data is stored in relational form. It is because relational model is the most popular data model. However, it is possible to extend our model to more general data models. We also assume the relational data is in a standard form such as BCNF so that data from different sources can be joined in a lossless manner. As the enterprises need to collaborate with one another to fulfill the desired business requirements, they will negotiate with each other to make data access permissions according to their agreements. For instance, an insurance company may require certain data from hospital and if the hospital thinks it is reasonable and safe to give insurance company access to the requested data, it may agree to release this data. We define the data access privileges using a set of authorization rules. Since we are dealing with the relational model, the authorization rules are made over the original tables belonging to some enterprises or over the lossless joins ($\bowtie$) over two or more of the relational tables. 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. We use the join operation over the relations because it can implicitly constrain the tuples being released to the authorized party and it meets the requirement of cooperative data access. For example, if the hospital thinks the insurance company should be able to obtain the patient information but only these patients who have plans with this insurance company, then the authorization given to the insurance company is defined only on the join result of hospital data and the insurance data. In this way, the insurance company can only retrieve corresponding patients' data instead of all the data from the hospital. We leave the formal definition of the authorization rules in later chapters.

To be able to apply the desired permissions among the collaborating parties, the authorization rules are visible to all the parties. It is worth noting that only the rules are exposed

to all the parties but no data. We can assume that the rules and related metadata are stored at a central place, and it can be either a party among the collaborating ones or a separate party that is trusted by all the collaborating parties. It is also possible to consider the situation that authorization rules are stored in a distributed manner, and each party only knows the permissions it is given. However, in this thesis, we consider our model as a centralized one. The purpose of cooperative data access is to provide services to the clients, so the clients will issue queries against the parties to obtain desired information. For instance, a client may want to retrieve the information about a patient in the hospital and insured with the insurance company. Thus, we first need to check if the information requested by the client is authorized to be released. If the information is authorized, a query execution plan is required to retrieve and compose the data from the owners to answer the query. Such a query execution plan must follow the given authorization rules so as to make sure no unauthorized information is released to unauthorized parties during the whole process. The tasks of query authorization check and query planning are fulfilled by a centralized party that knowns all the authorization rules. Figure 1.1 shows a possible architecture for such environment. Although the data are managed by different owners with their private clouds, the authorization rules are put at a central place to be referenced by all the parties. As clients initiate queries, the queries are first handled by the query planner. The planner checks the query authorizations and generates safe query plans based on the rules. After that, the client can use such safe distributed query plan to execute the query against the cooperative parties.

1.1.2 Problems

The data authorization rules are supposed to satisfy the requirements laid down by each enterprise, but without a careful global organization, the rules may not be either mutually consistent or adequate to allow all the desired data sharing. There are several reasons for the possible inconsistency. First of all, enterprise may make some rules independently so that they are unaware of the rules made by others. For example, when a hospital releases data to

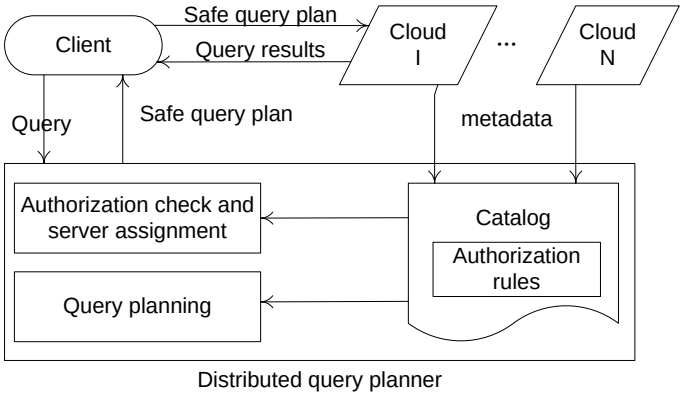

Figure 1.1: Centralized authorization rule control

an insurance company, it may not know what other information the insurance company can get from other parties such as a credit card company. Furthermore, some authorization rules involve information from multiple data owners (e.g., join over data belonging to multiple parties). Authorized information is generated through a series of relational operations, where join is the most important operation. As authorized information is given in the form of join results, a party getting pieces of information can also perform join operations over it. Consequently, the party can derive some information by itself based on the information it receives from cooperative parties. However, there is no rule to authorize such information release. In this case, we say such a set of rules is inconsistent relative to the set of intended authorization. For instance, if a party gets information both from insurance company and hospital, the party can derive the information about a patient with his/her plan. However, there is no given permission allowing this party to obtain such information. In such case, we say the rules are inconsistent. The rule inconsistency creates conflicting results for query authorization, so it needs to be properly handled. The rule inconsistency problem arises because the information that is not regulated by any rule is considered not authorized. However, it is not the case that all the local computation results are undesired. In the extreme case, if the enterprises find out there is no restriction on the local computation results, it is possible that a large number of authorization rules need to be specified. The

overwhelming number of rules can be difficult to manage, so we may want to use a concise way to present the data access permissions. In fact, if all the local computation results are allowed, we can use the initial set of rules made by the cooperative parties and consider an implicit way of information authorization. We call it as the implicit semantics of the rules. By doing that, we have a small number of rules compared to the explicit way of authorization, but the price is that we may need additional cost to determine the query authorization. In addition, when interpreting the rules using implicit semantics, we need to introduce negative rules to explicitly prohibit the undesired local computations.

Even if the rules are mutually consistent with one another, there are still other problems to be considered. In addition to inconsistency, it is possible that a query can be authorized but not implementable. Since the rules are individually specified based on business requirements of the cooperative parties, there may not exist a complete set of rules that can authorize all the steps in the process of answering an authorized query. 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 individually and thus no party is able to do the join operation! In such case, a query requesting the data on the join result of R and S is authorized by the rule, but the query cannot be answered. We say that a rule can be enforced among the cooperative parties if there exists a series of operations among the cooperative parties that is allowed by the rule permissions and the final result is exactly the information conveyed by the rule. Obviously, authorization rules are not always enforceable among the parties, and a basic problem is to determine implementability of them. This needs to be done as efficiently as possible. Furthermore, if a query is authorized and the corresponding rule can be enforced, we still need a safe query execution plan to answer the query. Such a plan must be consistent with the given set of rules so that no privilege restriction is violated. In addition, we need to deal with the unenforceable information. There are several different ways to handle such a problem.

One way to enforce a rule is to introduce a trusted third party. A trusted third party is an

entity other than the existing collaborating parties, and it is trusted by some collaborating parties so it is authorized to get information from these parties. In the above example, if there is a trusted third party, then the third party can be authorized to access both R and S, then it can do the computation of R join S for the desired party. Therefore, we discuss the possibilities of different third party models to enforce the rules and answer queries. A third party may either act as an opaque service provider that does not retain any data or provide richer functionality such as caching of data or query results. Multiple third parties maybe needed to provide data isolation and/or to improve performance. The use of a third party can be expensive both computationally and economically. Third parties may charge money for the services they provide, and the parties need to consider the potential risk of data exposure. Therefore, when using the third party to enforce the rules, we want to interact with the third party in a minimal way. In some cases, multiple third parties are needed because of potential conflicts among the data obtained by the third party as well as the performance considerations. For example, if we the computation result of $R \bowtie S$ needs to be prohibited on a given party, we also want to ensure it will not be generated at any third party. To achieve that, parties own the data of R and S should not share the same third party to enforce their rules. Such a constraint at the third party can even cause multi-way conflicts among the data. To give an example, the result of $R \bowtie S \bowtie T$ is undesired, but any combination of the two relations is allowed. In such scenarios, using minimal number of third parties to satisfy all the requirements is desired.

1.1.3 Contributions

We study the problems mentioned above, and propose mechanisms to solve these problems. We study the rule inconsistency problem and discuss possible ways to resolve it. We first consider the explicit semantics of the authorization rules. In this situation, if there is no explicit rule specifying the data release, then access to such information is prohibited. To resolve the inconsistency among the rules, one solution is to add additional privileges allowing the party to access the local computation results. To give a simple example, if

an enterprise P is authorized to get relations R and S from two parties, but no rule is authorizing P to access the join result of $R \bowtie S$, we can add a rule allowing P to access $R \bowtie S$ to resolve consistency. In contrast, the alternative way is to constrain the existing privileges of the party so that it cannot access all pieces of information and local computation is prevented.

By exploring the first approach, we propose a mechanism that analyzes the given set of rules and automatically adds necessary rules to authorize the access to all derivable information. Basically, we build a closure of rules based on the initial given set of rules. The mechanism takes advantage of the join properties among the relations and systematically generates the rules, and it ensures the rule inconsistency can be removed. Since the business relationships among the cooperative parties may change from time to time, the authorization rules may change correspondingly. Because any changes on the rules may result in new conflicts, we propose algorithms to maintain rule consistency in the cases of a new 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. The main issue to address is to ensure that changes are introduced in such a way that minimum number of queries is affected.

Considering the explicit semantics of the rules, checking whether a query is authorized is straightforward. The complexity of generating the rule closure to achieve consistency is generally quite acceptable due to fact that long chains of joins are rare in practice. However, it can be exponential in the worst case. Therefore, we also consider the implicit way of interpreting rules so that the rules are concise and local computations are allowed by default. In this case, to decide whether an incoming query can be authorized to run is not trivial, and the check needs to be done on the fly for each individual query. A query is authorized if a party has all the information that the query requests and the exact answer for the query can be generated with legitimate computations over this information. We propose a mechanism that efficiently checks the query authorization under implicit scenarios. Since local computation is allowed by default under implicit semantics, there is a need to introduce negative rules to prohibit undesired local computation results. Compared

with the authorization rules, the negative rules define which information a party cannot access. However, negative rules may be difficult to enforce. One possible way is to revoke some authorization rules from the party so the local computation cannot be performed. The other option is to use Chinese wall security policy [1]. In Chinese wall policy, data is categorized into conflicting classes. The data access privilege of a party depends on the previous data access history of the party, and it is only allowed to access one piece of data from each class. For instance, if data R and S are within the same conflicting class, a party P is free to visit either R or S at the beginning. However, once party P already has the data R it can never get S, and vice versa. Using such a mechanism, undesired results can never be generated.

Regarding the rule enforcement issues, 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. We propose a mechanism that checks the authorization rules in a bottom-up manner and finds all the information that is authorized by the rules and can be enforced among collaborating 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. Furthermore, we examine the problem of query plan generation in controlled collaborating environment, and it can be different from the classical distributed query processing. To find the optimal query plan under data access constraints is not an easy task due to the large search space for possible optimal query plans. We show the difficulty of finding optimal answer in such scenarios, and how it is significantly more complex than classical query processing. Thereby, we also propose an efficient greedy algorithm that in most cases provides either the optimal solution or a solution that is very

close to optimal. We prove that our algorithms are both correct and complete. That is, the rule enforcement checking algorithm finds all information that can be enforced among existing parties. The query planning algorithm is guaranteed to generate a query plan for each query against these enforceable information, and such a plan is consistent with the given authorization rules so that no data access restriction is violated.

Considering the using one global trusted third party to enforce rules, we aim to minimize the costs of using the third party. Therefore, we first model the costs of using the third party, which includes communication and computing costs. We show that minimizing both types of costs is NP-hard. Therefore, we propose efficient greedy algorithms and evaluate their performance against brute force algorithms. Since the brute force algorithm is exponential in time complexity, we compare the results only when the optimal solution can be generated in a reasonable amount of time. We find that the greedy algorithm finds cost in all cases that is very close to the optimal cost. In the cases where a single trusted third party is not adequate, we examine the problem of finding the minimal number of third parties to meet all security requirements. Specifically, we show such a problem can be reduced to the graph coloring problem which is also a NP-hard problem. Hence, instead of giving a solution that finds the optimal answer, we provide with heuristics taking advantage of the property of cooperative data access, which is complementary to the existing greedy algorithm for graph coloring problem.

1.2 Related work

The problem of controlled data release among distributed collaborating parties has been studied in [2]. The authors propose an efficient and expressive form of authorization rules defined on the join path of relations. Under such model, only explicitly authorized information can be safely retrieved. They devise an algorithm to check if a query with given query plan tree can be authorized using the explicit authorization rules. However, this work does not address the problem of determining the query plan. We note in this regard that regular query optimizers do not comprehend access restrictions and thus cannot be used to generate

query plans. Thus, determining an appropriate query plan is also a crucial problem that we address. In our work, we follow the format of authorization rules proposed in [2].

In another work [3], 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. The architecture and rule implementation issues are discussed in [4], where they proposed a centralized server to enforce the access control.

Processing distributed queries under protection requirements has been studied in [5–7]. 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 [8–11], but they do not deal with constraints from the data owners. Using a classical query optimizer may miss possible query plans in the cooperative scenario.

Answering queries using views [12] is close to our work also since each rule can be thought as a view. Answering queries using views can be used for query optimization [13,14], maintaining physical data independence [15], data exchange [16] and data integration [17, 18]. Different methods can be applied, materialized views can be treated as new options and put into the conventional query plan enumeration to find better query plan, queries can also be rewritten using given views with query rewriting techniques [19, 20], and sometimes conjunctive queries are used to evaluate the query equivalence and information containment. In addition, query optimization using cached results in distributed environment is studied in [21], and the query rewriting technique can also be used for view based access control [22].

Our scenario is different from all these works. To answer a query using given views, at least we know all the information provided by the given views can be obtained. In our situation, without knowing how the rules can be enforced, we cannot decide which views should be used. Most of these works consider conjunctive queries, and the decision problem of whether a query can be answered by the given views is NP-Complete [23,24] in general. However, the authorization rules in our model do not have selection operations and we consider lossless joins of relational data in standard form, whether a query is authorized can be determined in polynomial time.

Security issues in cloud computing have been discussed in [25–28], and there are some proposed secure architectures in cloud [29–31]. There are several works that focus on verifying data ownership in the clouds [32,33]. In addition, information leakage problem in clouds is discussed in [34]. A mechanism for efficient and exclusive access to outsourced data is proposed in [35], and some other works [36,37] consider the different security protections of the clouds and selectively outsource less sensitive data. On the other hand, attribute based access control is getting more attention, and a new encryption framework is proposed in [38] to combine the attributed based access control with the encryption mechanism.

There are some works on the access control in collaborative environments. In [39], the authors examined existing access control models as applied to collaboration, and point the weaknesses of these models. In addition, [40, 41] applied RBAC in the collaborative environments. [42] discuss the access control problems in the popular social networks. [43] 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 [44], collaboration among enterprises was also studied, but that work focused on different application data and multilevel policies.

There are several services such as Sovereign joins [45] to enforce the authorization rule model we used, such a service gets encrypted relations from the participating data providers, and sends the encrypted results to the recipients. Join processing in outsourced databases is also discussed in [46, 47]. In addition, there are some research works [48–51] about how

to secure the data for out-sourced database services. These methods are also useful for enforcing the authorization rules in our work.

Theoretically speaking, a trusted third party can be replaced using the secure multiparty computation (SMC) [52–54]. However, the generic solution of a SMC problem can be very complicated depending on the input data, it is usually unrealistic in practical. Particularly, the join operation among several parties with huge relational tables does not have an efficient SMC solution. Therefore, we consider using the third parties to enforce the rules.

1.3 Thesis Outline

In this thesis, the first chapter discusses the general challenges in secure cooperative data access scenarios, the problems studied and the contribution of this thesis and the previous related research works. In chapter 2, we study the problem of inconsistent authorization rule set and efficient query authorization check. In chapter 3, we discuss the problem of authorization rule enforcement among collaborative parties, where we provide a mechanism to check the rule enforceability as well as how to generate query plans for authorized queries. In chapter 4, we study the issue of minimal information exchanged with trusted third parties so as to answer the queries and enforce the rules. In chapter 5, we discuss the cooperative data access problem in cloud environments and the new challenges and opportunities together with conclusions and future works.

Chapter 2: Cooperative data authorization and rule consistency

A cooperative enterprise environment involves access to shared data that is defined by a set of authorization rules based on negotiations among these cooperating enterprises. In this chapter, we first formally introduce our authorization model and preliminary of the basic concepts. After that, we discuss the query authorization checking problem as well as the rule consistency problem and give corresponding solutions. At last, we introduce negative rules and discuss the possible ways of enforce negative rules.

2.1 Preliminaries

First, we describe the assumptions for our context. We assume simple select-project-join queries (e.g., no cyclic join schemas or queries). In general, the join operation cannot be done on any two arbitrary attributes, and the possible joins between different relations are usually limited. We assume that the join schema is given – i.e., all the possible join attributes between relations are known. Each join in the schema is lossless so a join attribute is always a key attribute of some relations. We assume all the existing parties depend on each other to work collaboratively, so that they all strictly follow the given rules and they are all not malicious. In addition, only one authorization rule can be given on each distinct join result for each party.

2.1.1 Notations and definitions

An **authorization 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 authorized attribute set, and P_t is the party authorized to access the data.

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*.*

We can consider a join path as the result of join operations without limitations on the attributes. Thus, A_t is the set of attributes projection on the join path that is authorized to be accessed by party P_t. Table 2.2 shows an example set of rules given to the cooperative parties. The first column is the rule number, the second column gives the attribute set of the rules, join paths of the rules are shown in the third column, and the last column shows the authorized parties of the rules. On each different join path, only one rule can be given to a party. We assume that each given authorization rule always includes all of the key attributes of the relations that appear in the rule join path. In other words, a rule has all the join attributes on its join path. We argue that this is a reasonable assumption as in many cases when the information is released; it is always released along with the key attributes.

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 query is transformed into an equivalent Projection-Join query authorization. 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. In fact, each join path defines a new relation/view. To better understand the authorization relationships between the queries and the rules, we give the definition for join path equivalence below.

Definition 2. *Two join path* J_i *and* J_j *are equivalent, noted as* $J_i \cong J_j$*, if any tuple in* J_i *appears in* J_j *and any tuple in* J_j *appears in* J_i*.*

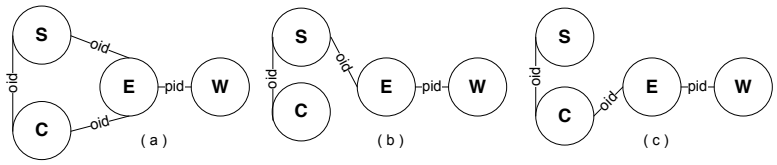

Figure 2.1: Join path equivalence

Based on the two parts of information conveyed by join paths, one necessary condition for equivalence is $JR_i = JR_j$. However, if several relations joins over the same attributes, then the join predicates (join orders) among the join paths can be different, but they are still equivalent. To decide join path equivalence, we put join paths into join graphs. A **join graph** is a graph $G = (V, E)$, where each vertex v in the graph indicates a relation, and each edge e is the join attribute for the possible join between two vertices. The given join schema is an example of join graph. For a given join path, we can also put its relations and join predicates into a graph. A valid join path is a spanning tree of a join graph, and two join paths are equivalent if they are both spanning trees of the same join graph. Figure 2.1 (a) shows the join graph of the given join schema, (b) is the graph representation of the join path of example rule r_8, and (c) is for the join path of rule r_{17}. Since figure 2.1 (b) and (c) are both spanning trees of (a), these two join paths are equivalent.

2.2 Online query authorization check

In this section, we discuss the problem under the implicit semantic of the authorization rules where the local computations are allowed by default. Given a set of authorization rules R in an implicit way, we want to check whether an incoming query q is authorized according to given rules R. Under the implicit semantic, we say a query is **authorized** if the query is authorized by any local computation results.

Definition 3. *A query q is* **authorized** *if there is a subset of rules R' of R, and there exists a resulting rule r_t which is the local computation results over R' such that $J_t \cong J_q$*

and $A_q \subseteq A_t$

2.2.1 A running example

To better illustrate the problem, we first give an example scenario of cooperative data access. Since query authorization and consistency problems are mainly related to the local computations, our discussion in this section focuses on the rules given to a single party. Therefore, we only give the sample rules of one party in our example to illustrate the problems.

The 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

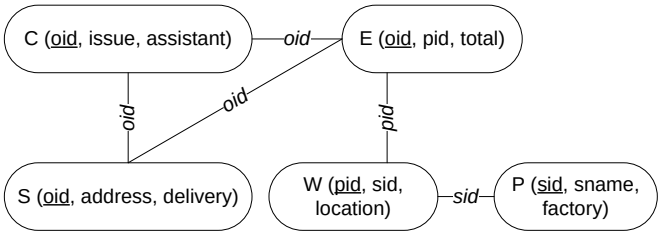

Figure 2.2: The given join schema for the example

also given in figure 2.2. 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 authorization rules given to the party E as described in Table 2.1. (Suitable rules must also be defined for other parties, but are not shown here for brevity.) The first column is the rule number, the second column gives the attribute set of the rules, join paths of the rules are shown in the third column, and the last column shows the authorized parties of the rules. The first rule define access to the relation owned by E, and the following rules define remote access cooperated with other parties.

To see if a query can be authorization, we need to check if the given authorization rules can directly allow the query permission. In the case that no existing rule can authorize the query, we need to check if the local compositions of the authorized information can allow the query. To illustrate query authorization, we shall consider two specific queries:

1. ***Select*** oid, total, location, sname ***From*** E-commerce ***as*** E, Warehouse ***as*** W, Supplier ***as*** P ***Where*** E.pid = W.pid ***and*** W.sid = P.sid

2. ***Select*** issue, pid, location ***From*** Customer ***as*** C, E-commerce ***as*** E, Warehouse ***as*** W ***Where*** C.oid = E.pid ***and*** E.pid = W.pid ***and*** total>‘100’

Since the authorization form is based on attributes, any attribute appearing in the Selection predicate in an SQL query are treated as Projection attributes. Thus, Query 2

Table 2.1: Authorization rules given to party P_E

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

can be put into a pair $[A_q, J_q]$, here A_q is the set $\{issue, pid, location, total\}$, and J_q is the join path $C \bowtie_{oid} E \bowtie_{pid} W$.

2.2.2 Join groups and Sub-Path relationship

In order to perform efficient authorization checking, we group relations according to their join capability. For this we define a **Join Group** as a set of relations that share the same join attribute and they can possibly join over that attribute. A relation can appear in several Join Groups. A Join Group is identified by the **join attribute** that its relations can join over which is the key attribute of some relation. Each query itself has an associated Join Path called **query join path**. Similarly, join path associated with a rule is called **Rule Join Path**. For instance, the query join path of Query 1 is $E \bowtie W \bowtie P$, and Rule 6 has the rule join path $W \bowtie P$. Similar to relations, the rules can be joined together. We first study the problem that given a query q to a party P_E, whether the given set of rules R on P_E can be composed using joins to authorize the query q. The first step is to figure out the rules that carry relevant information to the query. Hence, we define the concept of Sub-Path relationships between join paths, which is useful for determining the relevant rules for checking the authorization.

Definition 4. ***(Sub-Path Relationship)*** *A Join Path J_i is a Sub-Path of another Join Path J_j if: 1) The set of relations JR_i is a subset of the relation set JR_j. 2) J_i is equivalent to the join results in J_j which only involves relations in JR_i.*

The sub-path relationships can be determined using the graphs. Figure 2.3 shows the

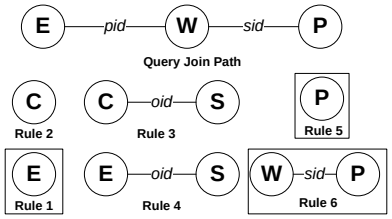

Figure 2.3: Rules defined on the Sub-Paths of Query 1

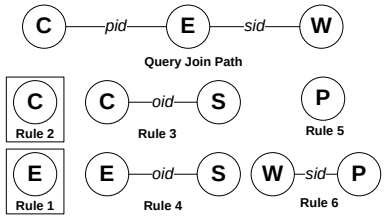

Figure 2.4: Rules defined on the Sub-Paths of Query 2

query 1 in our example, and the rules in the boxes are the ones defined on the Sub-Path of the Query Join Pquery join pathath. Figure 2.4 does the same for query 2. For query 1, rules r_1, r_5, r_6 are defined on the Sub-Paths of the query join path. For query 2, the rules are r_1, r_2. A rule defined on a Sub-Path of the query join path is called a **Relevant rule** to the query.

For the incoming query, we construct a **Query Join Group List** for it, and each entry in the list is a join group of rules. The unique join attribute appearing in the query is to identify each entry. Only rules in the list are the relevant rules that will be considered in the composition step. To compose the rules, we take advantage of the properties of the lossless joins in the join schema.

2.2.3 Join patterns and key attributes hierarchy

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 [55]. 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 $R.K \rightarrow S.K$. 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. 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 authorization 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 $oid \rightarrow pid \rightarrow sid$.

Chain join type

In a chain type of join, each relation has at most two join attributes. It is allowed to have multiple tables joining on the same attribute. Based on the lossless join condition discussed above, for each join between two tables, the key attribute of one relation is also the key attribute of the join result. For a chain type of join, the join path can be presented as a chain of relations, and the relations can be ordered according to the key attributes hierarchy. For a group of relations joining on the same attribute their positions can be changed in the ordered join path.

Star join type

For the star type of join, there is a central relation with more than two different join attributes to join with other tables. Other relations join with the central one on their key attributes, and these join attributes are non-key attributes of the central relation. This join type cannot be presented as an ordered chain where each relation appears only once. However, there is key hierarchy among the relations. The key of the central relation is still the key of the join result.

In above cases, each join between two tables involves key attribute of one relation, so that the equi-join operation usually reduces the number of tuples. In a cooperative environment, data from different parties usually does not have foreign key constraints.

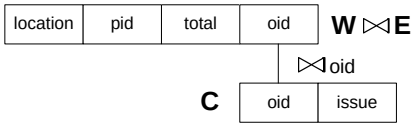

Figure 2.5: Join with no overlapping join paths

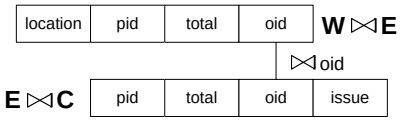

Figure 2.6: Join on the key of overlapping join paths

We do not consider cardinality-preserving joins [14] and join paths with different lengths contains different number of tuples in general.

2.2.4 Lossless join conditions over authorization rules

Based on the hierarchal join attributes in a join path, we discuss the possible lossless join scenarios between a pair of rules r_i and r_j. Given two rules r_i and r_j, we can think their join paths J_i and J_j stand for two different relations. There are several situations that the two join paths can join in a lossless way.

1. To form a join path of $R \bowtie_{s.k} S \bowtie_{t.k} T$, one situation is that two join paths do not have overlapped relations. For instance, it can be $r_i = R \bowtie_{s.k} S, r_j = T$. Since they do not have overlapped relations, the join attribute is clear. The two join paths must join on $t.k$, and we need to check their attribute set. If r_i does not have $t.k$, the join cannot be done. Figure 2.5 shows this join scenario. In the figure, we use relation C in the example to represent R, E represents S and W represents T. The join is performed on *oid*, which is the key of E.

2. Another case is that the two join paths J_i and J_j have overlapped relations. In figure 2.6, the two join paths are in the form of $R \bowtie_{s.k} S$ and $S \bowtie_{t.k} T$. The overlapped relation is S. In general, S can also be a join path. Here, the join condition is not clear as S can have multiple attributes such as $s.k$ and $t.k$. If they join over the key attribute of S, then $(R \bowtie_{s.k} S) \bowtie_{s.k} (S \bowtie_{t.k} T) \cong R \bowtie_{s.k} (S \bowtie_{s.k} S) \bowtie_{t.k} T \cong R \bowtie_{s.k} S \bowtie_{t.k} T$. The join is lossless in both chain and star type of schema.

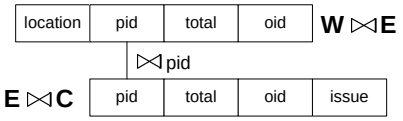

Figure 2.7: Join on join attribute of overlapping join paths

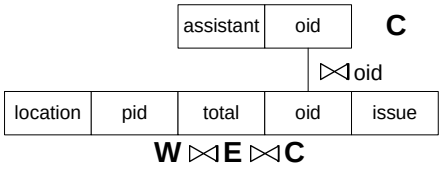

Figure 2.8: Join to extend attribute set

3. If the join attribute between two join paths is not the key attribute for the overlapping relation S, then a join operation is lossy in general. However, if the join is done on the join attribute among the target join path, the join is lossless. For example in Figure 2.7, $t.k$ is the join attribute between S and T, and it is not the key attribute of the overlapping relation S. However, we can first apply projection $\pi_T(S \bowtie_{t.k} T)$, then join it with the other join path. The join is also lossless in both chain and star type of schema.

4. In addition to the above scenarios where join operations generates a longer join path, another join scenario is the back join that extends the attribute set. We assume $J_i = R \bowtie_{s.k} S \bowtie_{t.k} T$ such as the example in Figure 2.8. Then it can join with $J_j = S \bowtie_{t.k} T$ on the common attribute $t.k$. Since an FD (Functional dependency) exists in a sub join path is always valid in the longer join path, the FD $t.k \rightarrow T$ can extend the attribute set of the previous rule with more attributes.

According to above discussion, when checking query permission, we can compose the relevant rules according to the key hierarchy of the query join path. As rules are organized in the Query Join Group List according to their join attributes, the rules having common join attribute will appear in the same entry of the list. Therefore, for the given query join path, we can check the possible compositions beginning from the highest level of the join attributes. We iterate all the rules including the highest level relation to start the composition of rules. For each of them, we check if join operation can be performed on the

next level of the join attributes. For each entry in the list we checked, if there are other rules can be composed with the existing rule, we expand the composed rule with more relations in the join path or more attributes. Finally, a set of maximally composed rules on the query join path can be obtained, and the query is checked with these rules to see if it is authorized. Here, we give a number of assertions regarding the rule composition and query checking which are useful in formulating the checking algorithm and proving their correctness.

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

Proof. Assume a query q has a Join Path of J_q. A rule r not defined on a Sub-Path of the query join path will have two possibilities by definition. 1) The Join Path J_r includes at least one relation T_m which is not in the set of JR_q. 2) The Join Path J_r is defined on the set of relations which is a subset of JR_q, but join over different join attributes. The composed rule that can authorize the query must have the equivalent Join Path as query join path. Otherwise, the query results will have incorrect tuples, and such a case also means the query is not authorized. Thus, if an authorization rule r has T_m in its Join Path, then any composed rule using this rule will also have T_m in its Join Path which is different from query join path. For the second case, such a join result is different from the query join path. □

2.2.5 Algorithm for checking query permission

In the first step, the algorithm examines all the given rules and builds the Query Join Group List as discussed above. In the second step, the algorithm composes rules efficiently with Query Join Group List. The algorithm examines the list entries according to the join attribute hierarchy. For rules in the top level entry, the algorithm begins the composition to generate a maximal composed rule. Each rule in the entry group is applied with projection operation first so only these relations and attributes that functional depend on this join attribute are preserved. All these projected attributes are merged into the composing rule.

Thus, the composing rule increases join path length as well as the attribute set. Once a rule has been merged with the composing rule, the algorithm will ignore it when iterating the lower level entries in the list. Eventually, one composed rule will be generated by the algorithm. The query will be checked with these rules to see if there exists a composed rule that authorizes the query. The detailed algorithm is presented in Algorithm 1.

We assume the complexity of the basic operation that checks whether a given rule can authorize the query is C, and there are N given rules, and the query q has a Join Path length of m. In the algorithm, step one has the worst case complexity of $O(N * C * m)$. Similarly, in step two, at most m entries and N rules are checked, and composing the rules is not expensive than C also, thus, the complexity of step two is $O(N * C * m)$. Therefore, the overall complexity of the algorithm is $O(N * C * m)$. Considering the fact that most join paths in practice involves less than 4 or 5 relations, the number of m is expected to be very small in most cases. Therefore, in average cases, we can expect the complexity of the algorithm close to $O(N * C)$.

Theorem 2. *The composition step can cover all the possible ways to authorize the query.*

Proof. The composition step looks for only possible compositions among Query Composable rules. According to the key attributes and relations' hierarchy, the composed rules that can authorize the query must include the top level relations. Also, all the information that can be included into the composed rule is through join operations, and only the portion of a rule functionally depends on the join attribute can be added into the composed plan in a lossless manner. As we consider all join attributes hierarchy and possible lossless join cases. The composition steps generate the maximal possible composed rule. □

Illustration with the running example

We begin with query 1. In the first step, the algorithm examines all the rules. As no single rule is defined on the query join path, none of the given rule can authorize this query. After figuring out relevant rules, the query join group list can be built:

oid → {Rule 1}, pid → {Rule 6}, sid → {Rule 5}

Algorithm 1 Online Query Permission Checking Algorithm

Require: Set of authorization rules, the query q
Ensure: Query can be authorized or not

```
    STEP ONE:
 1: for each authorization rule r do
 2:     if r authorizes q then
 3:         q is authorized
 4:         return true
 5:     else if Sub-Path(r, q) then
 6:         for each join attribute in q do
 7:             if r is functionally determined by this attribute then
 8:                 Add r into the entry of this join attribute in Query Join Group List
    STEP TWO:
 9: Construct an empty rule r_c
10: for Each rule r_t include the top level relations do
11:     r_c ← r_t
12:     Initialize a priority queue Q
13:     Enqueue all the join attributes in r_c
14:     while The queue Q is not empty do
15:         Dequeue the join attribute A_j
16:         Find the entry associated with A_j
17:         for Each rule r_r in this entry do
18:             if r_r has not been visited then
19:                 Apply projection to r_r with attributes functionally depends on A_j
20:                 Merge the projected rule with r_c
21:                 Enqueue new join attributes in r_c
22:     if r_c authorizes q then
23:         q is authorized
24:         return true
25: q is denied
26: return false
```

The the algorithm iterates each entry of the list, and attributes in each entries are added to the composed rule. Consequently, we have the composed rule $\{oid, pid, total, sid, location, facorty\}, (E \bowtie W \bowtie P) \rightarrow P_E$. Such a rule authorizes Query 1. In contrast, the composed rule for Query 2 is $\{oid, pid, total, issue\}, (C \bowtie E) \rightarrow P_E$. Since it is not on the query join path, the query is not authorized.

2.3 Offline authorization and rule consistency

When considering the explicit semantic of the authorization rules, an authorized query must have a matching authorization rule given on a join path that is equivalent to the one of the query which is the following definition for **Authorized query**.

Definition 5. *A query q is* **authorized** *if there exists a rule r_t such that $J_t \cong J_q$ and $A_q \subseteq A_t$*

The query authorization checking is straightforward in this case. However, due to the possible local computation, the inconsistency among the rules needs to be resolved. To achieve that, we propose an algorithm taking advantage of the functional dependencies among the basic relations to generate all derivable rules to ensure rule consistency. Such a process can be done as pre-computation.

2.3.1 Consistency of rules

To remove inconsistency by adding rules, one must generate all possible compositions through lossless joins of the given rules and add any missing ones from the list. We first define the notion of closure to make the rules consistent.

Definition 6. *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**.

Although we are discussing the problem under cooperative environment, the rule consistency property only applies to each individual party separately. 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 we create a group for the rules based on their key attributes which is similar to the previous join group. As the rules within this group share the same key attribute, any two of them can join over their key attributes.

Definition 7. *A* **rule group** *is a group of authorization rules associated with a key (join) attribute, where all the attributes in these rules functionally depend on this attribute. If a rule group is* **consistent**, *then it is called a* **consistent rule group**.

We follow the above definitions for Sub-path relationship and relevant rules. A rule r_t

Table 2.2: Authorization rules for e-commerce cooperative data access

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

can be locally generated only by combining the information from its relevant rules. Based on the relevance relationship, the rules are organized in a **Relevance graph**. Such a graph has different levels according to the corresponding 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 2.9 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.

2.3.2 Another set of example rules

To better understand the algorithm, we give a different set of example rules in this section. The set of authorization rules given to the party E are listed in Table 2.2. (Suitable rules must also be defined for other parties, but are not shown here for brevity.) Columns are the same as Table 2.1.

Given a set of rules, our goal is to generate the consistent closure of it. Our algorithm uses the join attribute hierarchy property and rule groups to efficiently generate the consistent closure. The rules are first divided into different rule groups and consistent rule 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.

2.3.3 Consistent rule group generation

The first step is to generate the consistent rule group. With the input as a rule 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 rule 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 attribute is not this attribute are 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 rule 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 rule 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 rule 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 rule group. When inspecting the length i join-path, the algorithm first takes the rule r_m with maximal length ($m < i$) in the current rule 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

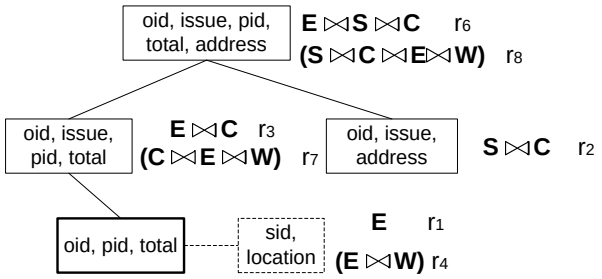

Figure 2.9: The consistent rule group of *oid*

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 rule 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 rule 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 rule 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 rule 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 rule group. This makes

Table 2.3: Generated consistent rule group of *oid*

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

rule group consistent and is listed in Table 2.3. Here the first 4 rules are given and rule 6 to 8 are added by the algorithm to make the rule group consistent. The built relevance graph is shown in Figure 2.9. 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.

2.3.4 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 rule groups based on its key attribute, and it will only be assigned into one rule group. Then, for each rule group, the algorithm generates the consistent rule group respectively.

Next, the algorithm iterates each rule group according to the level of its associated attribute in the key attribute hierarchy. To begin with, the algorithm inspects the rule group of the top level attribute. All the rules in the group being inspected are put into the

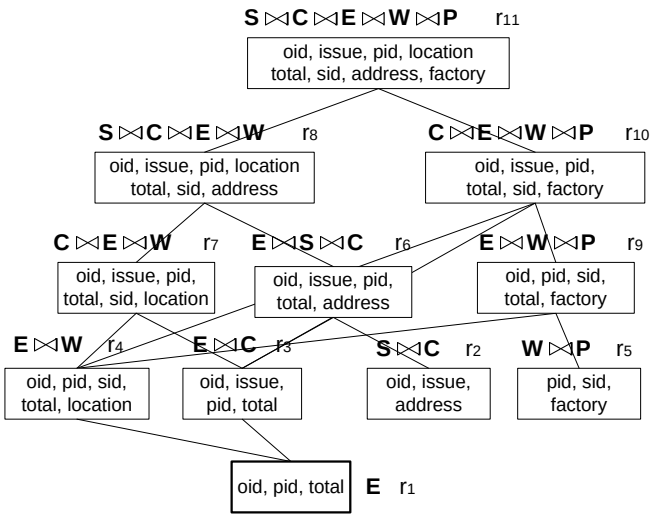

Figure 2.10: The relevance graph for the consistent closure.

target rule set first. Then, the algorithm checks the lower level groups one by one. For each rule 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 rule group being checked, then each rule in the rule 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 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 rule 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 relevance graph 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. 2.

We can use the running example to illustrate the process of rule group iteration. According to the key attribute hierarchy, *oid* is the top level attribute. Thus, the consistent rule group of *oid* which is listed in Table 2.3 is copied to the target rule set. The only remaining

Table 2.4: Generated consistent closure based on given rule set

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, location}	$E \bowtie_{pid} W \bowtie_{sid} P$	P_E
10	{oid, pid, total, issue, sid, factory, location}	$C \bowtie_{oid} E \bowtie_{pid} W \bowtie_{sid} P$	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

rule 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 rule group of *pid*, and this rule group is already consistent. As 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 2.4. The last three rules are generated in this process. Figure 2.10 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.

2.3.5 Average case complexity

The complexity of the algorithm depends on the given join schema and given rules. In worst case, generating a consistent rule group takes exponential time. However, in real cases, usually a rule group will not include more than 4 dependent relations. We make the assumption that the maximal number of dependent relations in a rule group is 4. In addition, we assume there are at most k given rules in a rule group. Within a rule 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 rule group. As k and p are usually small, the number of rules in a consistent rule group is usually less than 20 and the complexity of generating it is also low. We can think the generation of consistent rule groups takes constant time and there are at most C rules in a consistent rule group.

If there are m rule groups in total, it looks like we have the complexity of C^m in worst case. However, within a rule group, there is only one dependent relation that can join with the rules in the next rule 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 rule group contains only one or no rule such as the rule group of *pid* and *sid* in the example, so C is fairly small for many rule 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 3. *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 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 rule group. When the algorithm generates the consistent rule 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 rule 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.

Algorithm 2 Rule Closure Generation Algorithm

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

```
1: Put rules from R into rule 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 rule group G do
4:     Generate the consistent rule 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+
```

If r_i and r_j are not in the same rule 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 rule 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 rule group of r_j, all the possible joins over that join attribute are examined. All the rules generated afterwards are joined over the attribute of lower level of r_j, and rules from these rule groups never include the key attribute of r_j. If r_j is a generated rule, it is in its consistent rule 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.

□

2.3.6 Consistent authorization rule changes

Cooperative parties may change the authorization 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. In addition, the change may cause new conflicts among the rules. Thus, a mechanism is needed to maintain the rule consistency while authorization rules are changed.

In general, a change of authorization rule that meet the new business requirement and also has minimal impact on the remaining authorization 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 authorization 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 authorization rules in terms of the number rules being modified. The priorities in authorization rules and parties can be considered by extending such a mechanism, and we leave them for future works.

Two types of rule changes

A possible architecture for the authorization is that the authorization rules are stored at a central place different from any cooperative parties. An independent query optimizer then reads the authorization rules and generates the query plans. However, cooperative enterprises do not typically 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 party should keep a copy of all authorization rules locally. We discuss two types of rule changes below.

Independent change: This type of rule change only applies to a single party. Even though a join path involves authorizations from several parties, the change may occur because of a party no longer trusts some other party or their business relationships changed. Such changes usually affect only a small set of rules. Even if the change only takes on a single rule, to maintain the consistency of the rule set, a set of rules may need to be changed accordingly. The discussions below about the granting and revoking of authorization rules can be directly applied to this type of change. After the party changes its authorization rules, it broadcasts the change to other cooperative parties.

Cooperative change: Sometimes a group of parties may want to update the authorization 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 needs 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 needs to be performed among these parties atomically from a temporal perspective.

To achieve that, we use 2PC protocol for the rule update. Among the change cooperative parties, one party is selected as the master party, which we call as **coordinator**, and all the other change cooperative parties are called slave parties. Since we assume that 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 updating its rules, it will have lock them and other rule update requests received are rejected.

Overall, the mechanism works as follows. 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 it. If the slave party can update its rules, which means there is no ongoing rule update at this party, an agreement

is sent back to the coordinator, else the request is rejected. Only if the agreements from all slave 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.

Consistently grant more information

In the case of rule change, when more access privileges are granted to a party, we need a mechanism to maintain the rule consistency. 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. 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. 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 re-computes 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_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

Table 2.5: Authorization rules added together with a rule grant change

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

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 2.5 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 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 $O(C * u * s)$.

Revocation of existing authorization rules

Besides grant of more access privileges, the changes on the rules can also be the revocation of some existing authorization 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 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 2.10. 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 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, 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 which will be discussed in later sections, we assume we already know what are the locally enforceable rules. 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 appearances of the rules. We prefer to remove the rule with most appearances 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 cascaded manner. In worst case, it checks exponential number of pairs, and half of the existing rules need to be removed from the rule set.

For instance, in figure 2.10, 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.

The above mechanism to remove a rule is complicated and it considers only one next level of rules. Thus, we also consider removing the rules in another way. A single party usually issues a revocation, and this party usually revokes the authorization rules with its own data. Therefore, when a revocation is issued, it is common for the party to revoke all the rules including its basic relation. If this is the case, the revocation involves a set of rules that 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 consistently 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 appearing in the join path. The basic relation associated with least number of rules is then selected, and rules including this basic relation are removed from the set.

Back to our example, suppose that we want to revoke rule r_{10}. This mechanism first retrieves its relevant rules which are $\{r_4, r_5, r_9, r_3, r_1\}$. These join paths are examined, and the appearances of 4 basic relations are checked and counted. Therefore, relation C appears once, E appears 4 times, W appears 3 times, and P appears twice. Thus, the algorithm tries to remove the rule whose join path has C. Consequently, 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.

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.

2.4 Negative rules to prevent undesired results

The consistent rule set always allows the local computation results. However, the data owners may not desire such results. Therefore, negative rules are required so as to prevent a party getting such results. Negative rules are easy to be made, but they are difficult to enforce. In this section, we first discuss the way to check if a negative rule is being violated. Based on that, we use the Chinese wall security policy to enforce negative rules. As introduced before, the access privilege of a party is based on the access history of it under Chinese wall policy. Once releasing new information to the party will let the party get the sensitive results by local computation, and then the party is not allowed to get such new information. For instance, if data owners want to protect the information of $R_A \bowtie R_B$, and party P_C already get the information of R_A, then it is not allowed to access R_B anymore. In such case, we can guarantee that P_C cannot obtain the information of $R_A \bowtie R_B$.

Under explicit rule semantic, anything not specified in the authorization rules are prohibited. In this section, we focus our discussion with the implicit semantic of authorization rules, and negative rules need to be given in explicit way. A negative rule is similar to an authorization rule, but it may not have the join path information. In such case, a negative rule takes priority over others so that the associations among certain attributes are not released anywhere. We can also present a **negative rule** denoted as nr_t as a triple $[NA_t, NJ_t, NP_t]$, where NA_t is the attribute associations that is not allowed to be obtained by the party NP_t, and access to the subset of NA_t. NJ_t is the join path defining where

NA_t comes from, and it can also be $\emptyset$ as discussed. Once a negative rule on a join path NJ_t is specified, all the information on the join paths that are the longer paths including NJ_t is prohibited as well. We want to check using all the given authorization rules under implicit semantic to see if there exists any possible composition of rules that violates the negative rules. For example, we can have a negative rule denoted as nr_1 as below:

1. (E.oid, E.total, W.location), $\emptyset \rightarrow$ Party E

This rule means the Party E is not allowed to get these three attributes from two tables at the same time (in one tuple), however the appearance of any two of them is allowed. Before discussing Chinese wall policy, we need a mechanism to tell if a given negative rule is potentially violated. Negative rules cannot be composed by join operations since such operations do not make sense. Thus, we always check negative rules one at a time. To make sure a negative rule without a join path is not violated, all the possible join paths and rule compositions that may generate such attribute set need to be checked. To do so, one naive idea is to generate all the possible authorization rules and check if any one of them violates the given negative rules. Again, this is highly inefficient and we need a better algorithm. To check a negative rule with a join path, we can use the aforementioned mechanisms as discussed below.

2.4.1 Negative rule violation detection algorithm

If there is no single given authorization rule that violates the attribute set of the negative rule, then the negative rule can only be violated via the composition of the given authorization rules. To check a negative rule with a join path, one option is to use algorithm 1 to see if there will be a violation. We can put the negative rule denoted as nr_t being inspected as a query denoted as Q_n against the party. Thus, we run the online query permission checking algorithm to test if query Q_n is authorized by the given authorization rules. If that is the case, it means the negative can be violated potentially. However, as the information on the longer join paths should also be prohibited, we may need to run the algorithm several

times with negative rules on these longer join paths as well. In some cases, this becomes inefficient. We give the alternative way to do the check below.

We consider the negative rules without specified join paths as example. In fact, even if the join path is not explicitly specified, we can still figure it out through the attribute set. For instance, the above negative rule 1 has the join path $E \bowtie W$ which can be inferred from its attribute set. Similar to the query authorization checking mechanism, we use the Join Group List to check the possible rule compositions that may violate the negative rules. Different from the query authorization check where only the rules relevant to the query need to be checked, we need to consider all the given authorization rules here. Thus, we create the Join Group List by putting all the given authorization rules into the list. Each entry in the list is identified by the key attribute, and the content of the entry is an attribute set. We only need to list the key attributes in the rule join path. A rule is put into an entry if it contains such attribute, and we do the projection on the rule so that only the attributes functionally depends on this key attribute will be added into the entry. After that, we begin with the negative rule check according to property of the key attributes hierarchy. The entry with the highest level in the hierarchy is examined first. If its attribute set contains any key attribute of the other entries, we connected the current entry with these entries. Then, we examine the entry which connected with the current entry and has the key attribute on the next level in the hierarchy. Finally, we merge all the connected entries beginning with the one with highest level key. If such an attribute set is not a superset of the attribute set in the negative rule, the negative is not violated.

For instance, if the negative rule is nr_1 above, and we consider the set of rules in section 2.2.1. Then only two entries will be created which are identified by the key attributes of $\{oid, pid\}$. As rule r_1 to r_4 all have *oid* and *oid* is the key attributes of these rules, attributes appear in these rules are all put into the entry of *oid* and attributes of r_6 are put into the entry of *pid*. Since these two entries are connected, we can obtain the final attribute set with all attributes from the two entries, and that is a superset of attribute set of nr_1. Therefore, nr_1 is violated in this example. Algorithm 2 is the detail description of the process.

Algorithm 3 Negative Rule Violation Detection Algorithm

Require: Set of authorization rules, the negative rule nr, the join attribute set
Ensure: negative rule can be violated or not

```
    STEP ONE(Join Group List Generation):
 1: for each authorization rule r do
 2:     for each join attribute A_i in NJ_t do
 3:         if A_i ⊆ Attribute set of r then
 4:             Project r to keep attributes functionally depend on A_i
 5:             Merge these attributes into the entry of A_i in Join Group List
    STEP TWO(Verification):
 6: Pick the entry with highest key attribute A_s
 7: Create an empty attribute set UA
 8: Put A_s in a priority queue Q
 9: while Q is not empty do
10:     Get A_j with highest key attribute
11:     for each key attribute k in entry A_j do
12:         if entry k is not visited then
13:             Push entry k to Q
14:             Obtain the attribute set from the entry of A_j
15:             Union all the attributes with set UA
16: if The attribute set of the negative rule NA_t ⊆ UA then
17:     return true
18: return false
```

To examine the complexity, suppose that there are N given rules, and there are m join attributes in the join path. The cost of checking an attribute in a set takes constant time C, and the union operation takes constant time S, so the complexity of step one is $O(N * C * S * m)$ which is $O(N^2)$.

2.4.2 Applying Chinese wall policy

From the previous mechanism, we can know if a given negative rule can be violated. We apply Chinese wall policy only on these rules. To efficiently enforce negative rules using Chinese wall policy, we need to modify the architecture a little bit. As Chinese wall policy needs to be implemented using the access history of a party, we introduce a **centralized authority** CA that keeps the access histories for all the parties, and the CA controls the privileges of all the access requests. Each party needs to fetch data from others must go through CA first.

To answer a received query, a party issues an access request to CA so as to retrieved the related information from cooperative parties. Such access request is similar to a query that has the attribute set information along with the join path. As the access request denoted

as rq_a is received by CA, it first checks the access history of the requesting party P_r. Since each access history entry is similar to an authorization rule, CA first adds the new request rq_a into the access history of P_r. After that, for each of the negative rules given to P_r, CA tests if the access history of P_r violates the negative rule. The testing mechanism is the same as the above two algorithms we discussed depending on whether the join path of the negative rule is defined or not. If the testing result shows there is a violation, then it means allowing the access request of rq_a will violate at least one negative rule. Consequently, the access request rq_a is denied by CA, and it is removed from the access history of P_r. Otherwise, the request rq_a is approved, and P_r is allowed to access such information. With the coordination of CA, we enforce the negative rules with Chinese wall policy, and the undesired local computation results can be avoided.

Chapter 3: Authorization rule enforcement and query planning

In last chapter, we discussed the query authorization and rule consistency problems. In this chapter, we assume our discussion is under the explicit semantics of authorization rules, and we assume we already have a set of authorization rules that are consistent. We study the problems only involving existing cooperative parties, without any trusted third parties. We discuss the rule enforcement and query planning problems in this chapter.

Under explicit semantic, a query authorized by an authorization rule should be answered. However, "authorized" is only a necessary condition for a query to be answered but not sufficient. It is due to the fact that each join operation requires the operating party has the access privileges on join attributes from two pieces of data, and this requirement may not be always satisfies. In fact, the basic problem is to determine enforceability of the given rules.

To actually answer a query, we need at least one query execution plan. A **query execution plan** or "query plan" for short, includes several ordered steps of operations over authorized and obtainable information and provides the retrieved information to a party. The result returned by a query execution plan pl is also relational data, 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 authorization rule r_t. Therefore, a 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 authorization rule defines the maximal set of attributes that a query on the equivalent join path can retrieve. Therefore, a rule can also be viewed as a query. We call the query plan to enforce a rule as an **enforcement plan** or "plan" for short below.

Definition 8. *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* $J_t \cong J_{pl}$.

In general, we can use a query optimizer to find a query plan tree, and check if the plan tree can be safely assigned to the parties. However, regular query processing techniques usually focus on the performance perspective, and they do not understand the data access restrictions among parties. Consequently, some possible query plans to answer an authorized query may be missed by them. Therefore, it is desired to have an efficient mechanism to decide whether an authorized query can be answered with a consistent query execution plan which involves only cooperative parties. In addition, we should find an efficient query plan for an answerable query.

We address such problems separately. First, we examine each authorization rule and check the possible enforcement of each individual rule, and build a relevance graph that captures the relationships among the rules. This can be done once all the rules are given, and we can do it as a precomputing step. Once we know the enforceability of the rules and the built relevance graph, we can do efficient query planning according to them. Queries that are authorized by enforceable rules are guaranteed to have a safe execution plan. However, a rule may have many possible partial plans. Although we can systematically generate all partial plans and determine the optimal query plan; the worst case complexity of the process could be very high. Under such scenario, we propose an efficient algorithm that works in a bottom up manner to check the enforceability of each given rule, and we also give algorithms to generate efficient query plans for authorized queries.

3.1 Authorization enforcement among existing parties

Similar to the previous chapter, we consider a group of cooperating parties, each of which maintains its data in a standard relational form. The basic problems considered here are as follows: Given a set of authorization rules R on N cooperating parties, (a) identifies the

Table 3.1: Authorization rules for e-commerce cooperative data access

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

subset of R that can be enforced along with a consistent plan, and determines the maximal portion of other rules that can also be enforced. (b) Derives a query execution plan pl for an incoming authorized query q which is consistent with the rules R.

3.1.1 Example rule

We use a running example which is slightly different from the one in last chapter by removing the party of *Supplier* to keep the discussion clear. The set of example rules are listed in Table 3.1.

3.1.2 Consistent query plan

In this section, we first define the query plan consistency. For a query plan to be consistent with the rules, the result of the query plan must be authorized by a rule. In fact, a query plan recursively contains other sub query plans until the sub plans are access plans getting information from basic relations. Sub plans constitute the resulting plan with operations. Thus, a query plan contains a series of operations over sub plans, and each operation takes sub plans as input and generates another plan as output. In our context, the possible

operations on plans are projection, join and data transmission. For instance, there is an enforcement plan for r_3, and such a plan contains a join over two sub plans on the data authorized by r_1 and r_2 respectively. Information authorized by r_1 is owned by P_W, and the sub plan for it is an access plan reading the table W. The sub plan for r_2 includes an access plan reading table S at P_S, and another operation transmitting the data from P_S to P_w. The example plan for r_3 has the $J_{pl} = E \bowtie_{pid} W$, and $A_{pl} = \{oid, pid, location\}$. r_3 authorizes this plan.

Definition 9. *An operation in a query plan is* **consistent** *with the given rules R, if for the operation, there exist rules that authorize access to the input tuples of the operation and to the resulting output tuples.*

For the three types of operations in our scenario, we give the corresponding conditions for consistent operation.

1. Projection (π) is a unary operation. For a projection to be consistent with the rules, there must be a rule r_p authorizes ($\succeq$) the input information.

2. Join ($\bowtie$) is a binary operation, and two input sub plans pl_{i1} and pl_{i2} do a join operation and the resulting plan $pl_o = pl_{i1} \bowtie pl_{i2}$. Therefore, for a join operation to be consistent with R, all the three plans need to be authorized by rules. Since join is performed at a single party, and rules are upwards closed, if the input plans are authorized by rules, the join operation is consistent.

3. Data transmission ($\rightarrow$) is an operation involves two parties. The input is a plan pl_i on a party P_i, and the output is a plan pl_o for a party P_o, where $pl_i \rightarrow pl_o$. In our scenario, data cannot be freely transmitted between parties. The receiving party must be authorized to get the part of information that the party sends out. As each join path defines a different relation, the receiving party must have a rule that is defined on the equivalent join path as the information being sent. Otherwise, the transmission is not safe. The reason is similar as the query authorization discussed above. Therefore,

a data transmission operation to be consistent with R, if $\exists r_i, r_o \in R, J_i \cong J_o, P_i \neq P_o$ and $r_i \succeq pl_i, r_o \succeq pl_o$. In the case that P_i is sending information with attributes not in A_o, P_i should do a projection operation $\pi_{A_o}(pl_i)$ first followed by the data transmission operation.

In our example, r_8 authorizes P_E to get information on $(S \bowtie E \bowtie C \bowtie W)$. If P_S sends the information of r_{11} to P_E, it will not be allowed. Although the attribute set of r_{11} is contained by r_8, there is no rule for P_E to get data on the join path of $(E \bowtie S)$, and the data transmission is disallowed.

Definition 10. *A query execution plan pl is* **consistent** *with the given rules R, if for each step of operation in the plan is consistent with the given rule set R.*

3.2 Checking rule enforcement

It is not always the case that any authorized query has a consistent query plan. To perform a join operation, a single party must access the join attributes from two join operands. For instance, if we remove r_2 from the rule set, then r_3 that requires a join between E and W cannot be enforced since P_W is not allowed to read relation E and vice versa. Thus, we check the rule enforceability below.

3.2.1 Method overview

In this section, we first introduce some concepts and results, and then we present the algorithm that works from bottom-up to check the enforceability of each given authorization rules. According to the key hierarchy property, there always exists a key attribute from one basic relation that is also the key attribute of a join path. We call this attribute as the key attribute of the join path (or the key of the rules defined on such join path). If the join result of two join paths forms a valid longer join path, the join operation is always lossless. We call a plan as **joinable plan** if such a plan contains all the key attributes of the basic relations in its join path.

Lemma 1. *If a join path J_t is enforceable, there exists a joinable plan pl that $J_t \cong J_{pl}$.*

Proof. As we assume all the rule definitions contain the key attributes of the relations in its join path, these attributes are always authorized in data transmission operations. All the plans start from the rules on basic relations which are totally enforceable, and a longer join path is enforced by join operations over plans on shorter join paths. Therefore, if there is a plan for join path J_t, there always exist one plan that never project out any of these key attributes through the different operations in the entire plan, and such plan is joinable. □

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 proper 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, and we have the following lemma.

Lemma 2. *A rule has only one maximal enforceable attribute set.*

Proof. Firstly, a totally enforceable rule only has one maximal enforceable attribute set. Thus, a rule defined on basic relation only has one such set. To get the maximal attribute set, we do not eliminate any attributes via projections of plans, and maximal information is exchanged in data transmission operations, and such plans are always joinable. Therefore, if a rule is not totally enforceable, even it has several partial plans, these joinable plans are on the same join path and can always be merged by joining over the key attributes of the join path. Consequently, a partially enforceable rule has one maximal enforceable attribute set. At last, if a rule is not enforceable, its enforceable attribute set is empty. □

As discussed above, the required mechanism should tell which rules can be enforced and what are their maximal enforceable attribute sets. 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 set. 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 leave the given set of rules. In such scenario, we consider using trusted third party to enforce the unenforceable portion of the rules. Therefore, we discuss the first option in this chapter, and leave the second one to next chapter.

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 identified in the rule set.

3.2.2 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. Rules can be enforced by performing consistent operations over the information that is known enforceable. In addition, it is always the case that information from short join paths is put together to enforce a rule of longer join path. Therefore, we propose the algorithm to work in a bottom-up way 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 having 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 checking process iterates in the order of the join path lengths beginning with the rules defined on the basic relations on various parties. These rules can be totally enforced as the data owners sending their data 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. Each node in such structure is a rule with its maximal enforceable attribute set. The nodes in the graph are put in different levels based on their join path lengths. Different from the relevance graph discussed in last chapter, we consider the relationships among parties here. Among different parties, nodes can be connected if they have the equivalent join paths. Figure 3.1 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 3.1, 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. For example, in figure 3.1, 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.

After that, the algorithm gets the most efficient pair of rules that can enforce J_t. The cost of the plan can be quantified by the number of the join and data transmission operations

involved. For the selected pair of top level rules, the algorithm further searches for a more efficient join pair. The algorithm fixes one of the top level rule and search for the other matching rule whose join path has minimal overlapped relations. This can be done by following the path downwards beginning from the other top level rule in the pair. Finally, the best found pair of rules are recorded as the plan to enforce this join path J_t. Our algorithm is not designed to find a globally optimal way to enforce a join path due to extremely high complexity of doing so. Nevertheless, the algorithm attempts to make local (greedy) optimizations wherever possible. The topic of eventual effectiveness of local optimizations is beyond the scope of this work and is not addressed here. For instance, the join path J_{13} can be enforced with the pair of rules r_{11} and r_{12}, which is the only found pair. The algorithm further searches for pairs with fewer overlapping. Along the path, pairs r_{12} and r_9 is checked, and this is a better one than r_{11} and r_{12}, and the algorithm keeps this result $J_9 \bowtie J_{12}$ as the plan to enforce J_{13}. Only the join attributes in relations are preserved in such plans.

Meanwhile, the algorithm 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

Algorithm 4 Rule Enforcement Checking Algorithm

Require: All given authorization 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 ← ∅, 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 ← the union of attributes in R_v
            Missing attribute set A_m ← A_t
            for Each pair of relevant rule (r_s, r_r) do
                if The pair can locally enforce J_t then
                    A_m ← A_m \ A_v and break
            if A_m ≠ ∅ then
                Put r_t with A_m into the Queue of J_t
            AJ_t ← AJ_t ⋃ 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 ← A_m \ AJ_t
                if A_m ≠ ∅ then
                    Replace A_t with A_t \ A_m in graph
            else
                r_t cannot be enforced, remove rules on J_t from graph
    Join path length i++
```

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 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 3.1 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 3.1,

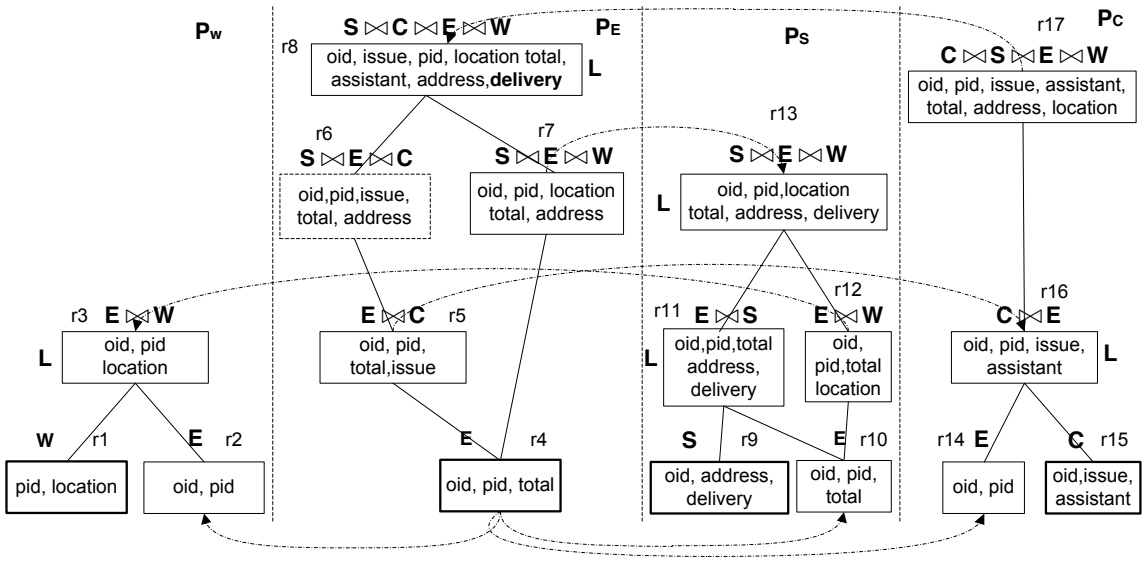

Figure 3.1: relevance graph built for the example

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 4.

In algorithm 4, 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 4 is $O(N_t * N_o^2 * C)$, where N_o is usually very small. In addition, this algorithm can be used as a pre-compute step once all the rules are given.

Theorem 4. *The Rule Enforcement Checking Algorithm finds all enforceable information.*

Proof. As all the information can be obtained on 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 all the rules on this join path cannot be enforced and can be discarded. The algorithm first finds a way to enforce the join path of the rule r_t. The check on local relevant rules 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 attribute for r_t. □

3.3 Complexity of query planning

The above mechanism tells us which queries can be answered with consistent plans. However, we still need to generate such consistent plans. From performance perspective, we always want optimal plans with minimal costs. Unfortunately, finding the optimal query plan is *NP-hard* in our scenario.

Theorem 5. *Finding the optimal query plan to answer an authorized query is NP-hard.*

Proof. The optimization of set covering problem is know to be *NP-hard*. The set covering problem can be described as follows: there is a set of elements $U = \{A_1, A_2, ..., A_n\}$ (called the universe), and theres is also a set of sets $S = \{S_1, S_2, ...S_m\}$ where S_i is a set of elements from U and is assigned a cost. Given an input pair (U, S), the task is to find a set covering that has minimal total cost. We can convert a set covering problem into a cooperative query planning problem. Assuming that there are two basic relations R and S which can join together, and we map each element in the universe U into an attribute for relation R, and add another attribute A_0 to R and S as their key attributes. Thus, R has the schema $\{\underline{A_0}, A_1, A_2, ..., A_n\}$, and we also assign the relation S with the schema $\{\underline{A_0}, A_{n+1}\}$. Next, we then consider a query requesting the attribute set $\{A_1, A_2, ..., A_n\}$ on the join path of $R \bowtie S$. Based on that, we can construct rules on $m + 1$ parties according to the given set covering problem. Party P_0 has the rule with join path $R \bowtie S$ that authorizing the query. For each other party P_i, it has the rule r_i on $R \bowtie S$ with the attribute set $S_i \bigcup \{A_0\}$. P_0 cannot locally do the join $R \bowtie S$, but other parties can enforce their rules r_i locally, and their costs are known. Therefore, for P_0 to answer the query, it needs a plan bringing attributes from other parties and merging them at P_0 (multi-way join operation on attribute A_0) to answer the query. The optimal plan needs to choose the rules with minimal costs,

and the union of their attribute sets must cover the query attribute set. In such case, if the optimal query plan can be found in polynomial time, the set covering problem also has a polynomial solution. Thus, finding the optimal query plan in our scenario is *NP-hard*. □

3.3.1 Query plan cost model

It is reasonable to assume that the numbers of tuples in the relations are known. In addition, the join selectivity between the relations is also known so that the size of join paths can be estimated as well. The cost of a query plan mainly includes two parts: 1) cost of the join operations, 2) cost of data transmission among the parties. We assume joins are done by nested loop and indices on join attributes are available. The cost of a join operation between R and S can be estimated as: $\alpha(Size(R) * Size(S) * P_{(R,S)}) + (Access(R) + Size(R))$, where $Size()$ is the number of tuples in the relation and $Access()$ is the cost of retrieving the relation. R is the smaller relation, α is the cost of generating each tuple in the results, and $P_{(X,Y)}$ is the known join selectivity. The costs of data transmission are only decided by the size of the data being shipped. The cost of moving $R \bowtie S$ from a party to another is $\beta(Size(R) * Size(S) * P_{(R,S)})$, where β is the per tuple cost for data transmission. Under such assumption, we can compare the costs of different query plans.

3.3.2 Upper bound complexity of plan enumeration

Although finding an optimal query plan is *NP-hard*, we want to see if it is possible to enumerate all possible query plans and compare them to get the optimal one as the join path length in our scenario is usually limited. We assume the longest join path is 5, and we begin the process based on the graph produced by the previous enforcement checking algorithm. When a query comes in, we first filter unrelated rules (rules not connected to the target node in the graph) and unrelated attributes (non-key attributes that are not in the query attribute set). In the classical query processing, the query attributes are always retrievable from the corresponding relations and usually the generated plan does not contain repeated joins (two join operands have overlapped relations). However, as authorization

rules put constraints in our scenario, additional join operations may be required so as to answer the query. Moreover, to enumerate the query plans, we should not only consider the different ways to perform the join operations, but also the different paths to retrieve the query attributes.

To generate a consistent plan for a query, we first need a plan that enforces the query join path. Once we have such a plan, it can further join with relevant plans to get all requested attributes to achieve the final query plan. Therefore, we first enumerate different possible ways to enforce the query join path. As we are interested in the worst case scenario, we assume the query join path is length of 5, and we consider the possible last join operations among a pair of relevant rules to enforce the target join path. In the worst case, the possible pairs of rules with different join path lengths are $(3, 2), (3, 3), (3, 4), (4, 1), (4, 2)$, where two numbers in parenthesis are the lengths of two relevant rules. We cannot discard pairs with overlapped relations such as $(3, 3), (3, 4), (4, 2)$ because in our scenario, enforcing a longer join path can be more efficient than a shorter join path. Next, we need to recursively search for the possible ways to enforce the join paths in each possible join pair listed in the previous step. We give the possible combinations for this recursive process below. Moreover, instead of counting the possible ways locally, we need to further consider the possibility that the join path is enforced via a remote party. Remote parties can enforce the join paths and sends the results to the target party.

We need to enumerate all possible ways of join path enforcement instead of finding only the optimal way to enforce the join path because the optimal query plan does not necessarily have the optimal enforcement plan for the query join path in our scenario. In fact, finding the optimal way of enforcing a join path can be somewhat easier, and we will discuss that later. Besides the join path enforcement, additional steps are still needed to retrieve missing attributes which are requested by the query but not enforced by the selected join path enforcement plan. Given various authorization rules, different join path enforcement plans may result in different sets of missing attributes. Therefore, for each different missing attribute set, we need to further enumerate the steps that retrieve these

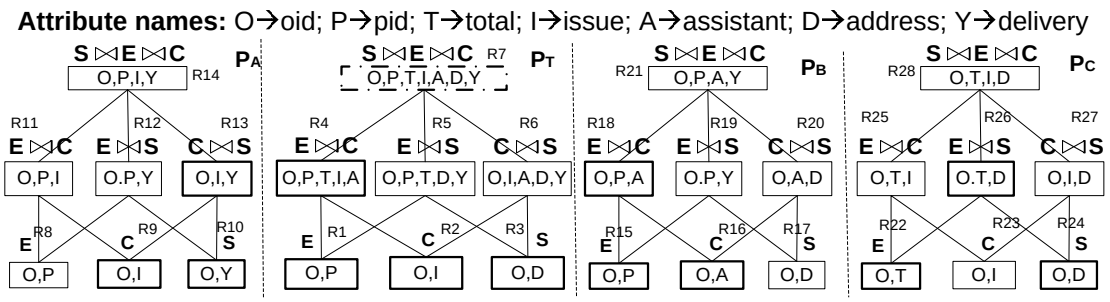

Figure 3.2: A simple worst case example

attributes via relevant rules on the cooperative parties. As a missing attribute may appear in multiple relevant rules, choosing the optimal set of relevant rules is similar to a set covering problem. Thus, to find the optimal answer, we need to enumerate all the possible sets of relevant rules that cover the missing attributes. Since the problem is similar to set covering problem, given N_r relevant rules that have the missing attributes, there are 2^{N_r}-1 combinations to check. In our scenario, once we select a rule cover, we still need to further find plans enforcing these rules to known their costs.

To illustrate the complexity, we construct a simple example with join path length of 3. In figure 3.2, there are four parties and they all have rules on different join paths. The attribute names are simplified to save space, and edges connecting the rules with equivalent join paths across the parties are also omitted to keep the graph clear. In the example, the query asks for all the attributes and only r_7 (dashed box) can authorize the query. For various plans enforcing the target join path, none of them can enforce all the query attributes. The possible ways to enforce the join path locally on P_t is $3 * (1 + 2) = 9$. Considering other 3 parties, we have $(3 * 4 * (1 + 2 * 4)) * 4 = 432$ different ways of enforcing the join path, and these plans result in 6+4 =10 different missing attribute sets. For each of them, we need to check the ways to get missing attributes. For example, if the missing attribute set is $\{total, assistant, delivery\}$. Then, there are 12 relevant rules having the missing attributes, and the possible combinations to consider is 2^{12}-1.

In table 3.2, we list the maximum numbers of possible join path enforcement plans for

Table 3.2: Maximum number of plans for each join path length

JP Length	**Maximum number of join path enforcement plans**
1	1
2	T_n
3	$(C_3^2 * S_2 * (1 + 2 * S_2)) * T_n$
4	$(C_4^3 * S_3 * (1 + C_3^1 * S_2 + C_3^2 * S_3) + C_4^2 * S_2 * S_2) * T_n$
5	$(C_5^4 * S_4 * (1 + C_4^1 * S_2 + C_4^2 * S_3) + C_5^3 * S_3 * (S_2 + C_3^1 * S_3)) * T_n$

each join path length. Notation S_i indicates the number of plans for a join path of length i. We assume there are at most T_n parties having the rules on equivalent join paths.

To sum up, in the worst case, to enumerate all the possible plans for a query, there are $S_5 + N_e * (2^{C_m} * -1) * C_m * S_4$ possible cases, where S_5 is the maximal number of ways to enforce a join path of length 5 and it is similar for S_4. C_m is the number of missing attributes which should be the number of all non-key attributes in the worst case. N_e is the different number of missing attribute sets based on all the join path enforcement plans. In above example, this number is 10. Because of the difficulties mentioned above in enumerating all possible ways of enforcing join paths and attributes, we consider a greedy algorithm in the following.

3.4 Consistent query planning

As finding optimal plan is very difficult, instead of giving an optimal query plan, we focus on generating an efficient consistent query plan in this section.

3.4.1 Query planning algorithm

When generating a plan for the query, we always choose the optimal join path enforcement plan first, and then apply the set covering greedy mechanism on the missing attributes to find required relevant rules. The optimal enforcement plan for each join path on a specified party can be pre-determined by extending the rule enforcement checking mechanism in a dynamic programming way. When checking a rule r_t, instead of inspecting only top level relevant rules, all the possible join pairs at P_t are inspected. These possible plans are

compared and only the one with minimal cost is kept. As J_t-cooperative parties find their optimal local enforcement plans respectively, these plans are further compared among the parties so that each party finds its optimal way of enforcing J_t. As discussed, the selected join path enforcement plan usually results in a missing attribute set. To get these attributes, we explore the relevance graph to decompose r_t into a set of relevant rules that can provide these attributes. We record the required operations among these rules, and then recursively find ways to enforce these rules so as to generate a query plan.

Firstly, as r_t uses the optimal join path enforcement plan, it can be extended to get missing attributes that appear in the relevant rules of basic relations on all J_t-cooperative parties. This can be done through semi-join operations. In such cases, the party P_t can send the plan with only the join attributes to the J_t-cooperative party, and the receiving party does a local join with its relevant rules of basic relations to get these attributes. Such information is sent back to P_t, and P_t performs another join to add these attributes to the query plan. In this way, we can reduce the missing attribute set by removing these attributes.

The remaining missing attributes can always be found in the relevant rules on J_t-cooperative parties. However, these relevant rules are defined on join paths instead of basic relations. Similar to the above case, the missing attributes carried by these relevant rules on P_t can be brought to the final plan by a local join, and these on remote J_t-cooperative parties can also be added to the plan by semi-join operations. Thus, our next effort is to determine these relevant rules. Here, we always pick the relevant rule that covers the most attributes in the missing attribute set until all the missing attributes are covered by the picked rules. This is a greedy approach, and is similar in spirit to the approximate algorithms used for the set covering problem. The relevant rules effectively allow us to decompose the rule (i.e., express in terms of) rules with smaller join paths. The missing attributes are also reduced in the process by considering the rules involving basic relations. During the decomposition, the algorithm associates the set of attributes with the decomposed rule which are the missing attributes expected to be delivered by this rule.

The attribute set of the rule is projected onto these attributes only, but the join attributes are always preserved. In addition, we also records the operations between the existing plan and these decomposed ones. If they are on the same party, a join operation between them is recorded. Otherwise, a semi-join operation is recorded. Since each decomposed rule also needs a subplan to enforce, we can iterate the process of decomposition, and the algorithm uses a queue to iteratively decompose rules until all the rules are on basic relations. This decomposition process gives the hierarchal relationships among rules which indicate how related attributes can be added to the final plan.

The decomposition process gives a set of rules, but we also need the subplans to enforce the join paths of these rules so as to generate a complete plan. To achieve that, we inspect the join paths of these decomposed rules from bottom-up. We use another priority queue to keep all the join paths from the decomposed relevant rules, and the shortest join path is always processed first. This allows the use of results from the enforcement plans of sub join paths as much as possible. The algorithm uses the best enforcement plan for each join path as discussed. When an enforcement plan of a join path is retrieved, the algorithm combines previously recorded operations to generate the subplan for the decomposed rule on such join path. Finally, the algorithm finds the plans for each join paths in the queue, and generates the final query plan with a series ordered operations starting from the basic relations, and it is described in Algorithm 5.

3.4.2 Illustration with example

Figure 3.1 gives an example of the relevance graph that is built by our algorithm. Non-composable rules, non-enforceable rules and attributes that cannot be enforced are all removed in the structure. Any remaining rule in the structure can be answered by a consistent query plan.

For instance, if the query q is on the join path of $S \bowtie C \bowtie E \bowtie W$ and the attribute set A_q is {oid, pid, issue, assistant, total, address, location}. Such a query is authorized by the rule r_8 on the party P_E. Then the algorithm begins from this node to build the query

Algorithm 5 Query Planning Algorithm

Require: The structure of rule set R, Incoming query q
Ensure: Generate a plan answering q.

```
1: if There is a rule r_t, J_t ≅ J_q and A_q ⊆ A_t then
2:     Missing attribute set A_m ← A_q
3:     Initialize queue Q, and priority queue P
4:     Enqueue r_t to Q with A_m
5:     while Queue Q is not empty do
6:         Dequeue rule r_t and the associated A_m
7:         for Each J_t-cooperative party do
8:             Finds the attribute set A_b from basic relations
9:             A_m ← A_m \ A_b
10:            Record connections between r_b and r_t
11:        while A_m ≠ ∅ do
12:            for Each relevant rule r_s on P_co do
13:                Find the rule with max A_m ⋂ A_s
14:            Enqueue the rule r_s with π(A_m)
15:            Enqueue the join path J_s to priority queue P
16:            Record connections between r_s and r_t
17:            A_m = A_m \ A_s
18:    while The priority queue P is not empty do
19:        Dequeue the rule r_s with join path J_s
20:        Add the path to enforce J_s to plan
21:        for Each J_s-cooperative party do
22:            if The party has recorded A_b on J_s then
23:                Add (⋈ / →) operations between r_b and r_s
24:        for Each decomposed rule r_d from r_s do
25:            Add (⋈ / →) operations between r_d and r_s
26: else
27:    The query q cannot be answered
```

plan.

First of all, rule r_4 of basic relations E on P_E is picked, and the attribute set {oid, pid, total} in A_q can be found in this rule. Next, the J_8-cooperative party P_C is inspected. It has relevant basic rules r_{14}, r_{15}, but the attributes in r_{14} is found in r_4 already, so it is discarded. Attributes found in r_{15} are removed from the missing attribute set just like the ones in r_4. Also, the algorithm records a join of r_8 with r_4, and a semi-join between r_8 and r_{15}. Since now the missing attribute set is $\{location, address\}$, the algorithm looks for relevant rules to start decomposition. As the relevant rule r_7 includes both attributes, the algorithm picks this rule and record a join between r_7 and r_8, but the attribute set associated with r_7 is reduced as $\{oid, pid, location, address\}$.

Next, the algorithm iterates to decompose the rule r_7 with missing attribute set {*location*, *address*}. As the attribute *address* can be obtained from r_9 at party P_S, the algorithm

records a semi-join between r_7 and r_9. Because of the attribute *location*, r_{12} is found as a decomposed rule with missing attribute set $\{location\}$. At next iteration, the J_{12}-cooperative party P_W is examined, and the related rule r_1 is recorded as a semi-join with r_{12} as *location* is found. Consequently, the decomposition process is completed, and the join paths J_3, J_{13} and J_8 are pushed into the priority queue.

Next, the algorithm checks the join paths from the priority queue in the order of join path length to build the plan. First, J_3 is checked. As the best plan to enforce J_3 is a join between r_1 and r_2, such a plan is chosen. Since r_1 has a semi-join with r_{12}, now the plan adds this operation to generate a sub plan enforcing r_{12} with the attribute set $\{oid, pid, location\}$, which is the attribute set associated with r_{12}. Next, J_7 is examined. The recorded plan to enforce J_7 is a join between J_{12} and J_9. Now the algorithm can take the advantage that J_{12} is already enforced in previous step. As r_9 has a recorded semi-join with r_7, the algorithm adds this operation and a plan enforcing r_7 with the attribute set $\{oid, pid, location, address\}$ is generated.

Finally, the algorithm takes the similar steps to generate the final plan for r_8 with the attribute set equivalent to A_q. Therefore, such a plan can answer the example query and it is consistent with the given rules. It is worth noticed that such a plan avoided enforcing the join paths r_5 and r_{11}, and the plan for the longer join paths use the intermediate results generated for its relevant join paths. Thus, such a plan is efficient under the restriction of the rules. Table 3.3 shows the steps of the query plan generated by the algorithm. Each step in the table generates a subplan of the final plan. In these steps, we assume projection are pushed to basic relations, so only join attributes and missing attributes are retrieved from the relevant rules, and we omit such steps in Table 3.3.

3.4.3 Theorem Proofs

In this subsection, we prove the correctness and completeness of the proposed algorithms.

Theorem 6. *A query plan generated by* **Query Planning Algorithm** *is consistent with the set of rules R.*

Table 3.3: A consistent plan for the example query

Step Number	**Performed operation**
$Step_1$	P_E sends oid, pid of E to P_W
$Step_2$	P_W does a join on $Step_1$ and relation W
$Step_3$	P_W sends $Step_2$ to P_S
$Step_4$	P_W does a join on $Step_3$ and relation S
$Step_5$	P_S sends $Step_4$ to P_E
$Step_6$	P_E does a join on $Step_5$ and relation E
$Step_7$	P_E sends $Step_6$ with projection on $\{oid, pid\}$
$Step_8$	P_C does a join on $Step_7$ and relation C
$Step_9$	P_C sends $Step_8$ to P_E
$Step_{10}$	P_C does a join on $Step_9$ and $Step_6$

Proof. The **Query Planning Algorithm** works on a subset of rules of R. In the generated plan, the subplans to enforce join paths are consistent. They are generated during the rule enforcement checking. Each join operation in such a plan is added according to a legitimate local join over the relevant rules, and each data transmission operation happens only among J_t-cooperative parties. Thus, these subplans are consistent. In the iteration of decomposing rules, there are join and semi-join operations between the decomposed rules and the original rule. A join operation between a rule and its local relevant rule is always consistent. A semi-join between a rule and a relevant rule on its J_t-cooperative party is also consistent. It is because the attributes in the relevant rule can be obtained by the rule with J_t on the same party, and the data transmission between two parties is consistent as they are J_t-cooperative parties and the original rule is always authorized to access these missing attributes. Since each operation in the plan is consistent, the plan generated by **Query Planning Algorithm** is consistent with the rule set R. □

Theorem 7. *The* **Rule Enforcement Checking Algorithm** *finds all consistently enforceable information.*

Proof. As all the information can be obtained on join results comes from the basic relations, the algorithm works in bottom-up manner to capture all possibilities. If the join path of a rule cannot be enforced, then all the rules on this join path cannot be enforced at all. Therefore, the algorithm first searches for all the possible ways for a join path to be enforced

at any party. As a join between a rule and its local relevant rule is alway consistent, the corresponding step in the algorithm finds all the locally enforceable attributes. Since the 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 other parties. There is no other way to enforce more attribute for r_t. □

Theorem 8. *For a query q and a set of given rules R, if the* **Query Planning Algorithm** *does not give a query plan, then there does not exist a consistent query plan.*

Proof. According to the above lemma, all the enforceable information is given by **Rule Enforcement Checking Algorithm**. Thus, if there is an enforceable rule r_t to authorize q, the **Query Planning Algorithm** can always generate a consistent query plan. Otherwise, q cannot be answered safely. □

3.4.4 Preliminary performance evaluation of the algorithm

Since our query planning algorithm works in a greedy way, we want to evaluate the output results. However, the optimal plan cannot be found in general, so we cannot compare our results with the optimal ones. Thus, we use simple examples, where manually finding the optimal plans becomes possible, and we perform preliminary evaluation on these cases. In the following, we assume the selected join path enforcement plan carries the maximal attributes along with it.

Case 1

Firstly, we can take a look at the example in figure 3.2. For simplicity, we assume all the relations have the same sizes. Given the same query discussed before which only r_7 can authorize, the optimal plan should be as follows: join two relations at P_t first, and then join with the third one at P_t to enforce the join path of $S \bowtie E \bowtie C$. Then P_t sends the *oid* on the join path of $S \bowtie E \bowtie C$ to other parties, and do semi-joins with each of the party to obtain the missing attributes $\{total, assistant, delivery\}$. Finally, P_t does a local join with these information got from remote parties and such a plan answering the query.

In this case study, our greedy algorithm generates the same optimal plan. As the optimal way to enforce join path $S \bowtie E \bowtie C$ is the local enforcement at P_t, our plan also gets the missing attributes via semi-join operations.

Case 2

In the example shown in figure 3.1, assuming the query has the join path $S \bowtie C \bowtie E \bowtie W$, and the attribute set includes all the attributes in r_8 except *delivery*. For such a query, our algorithm first find the optimal way to enforce the join path, which can be represented as $(((r_1 \bowtie r_2 \rightarrow P_S) \bowtie r_9) \rightarrow P_E) \bowtie (r_{14} \bowtie r_{15} \rightarrow P_E)$. This plan results in a missing attribute set $\{total, assistant\}$. Next, the algorithm adds a local join with r_4 to retrieve *total*, and a semi-join with r_{15} to obtain the attribute *assistant*. In fact, there are only two ways to enforce the query join path in this example. The other way is to perform $r_9 \bowtie r_{10}$ first and then join with r_{12} at party P_S. By doing that, the plan can carry the attribute *total* and only has *assistant* as missing attribute. However, if we compare the two plans, the difference is that our plan gets the attribute *total* via a join among relation E and join path $S \bowtie C \bowtie E \bowtie W$, and the latter plan perform the join among E and S on P_S. As the longer join path usually has much fewer tuples, and no matter the sizes of relation S and E, the former plan is better than the latter one in this example case. For the missing attribute *assistant*, as it can only be retrieved from party P_S, getting it from r_{15} is better than r_{16}. Therefore, the query plan generated by our algorithm is actually the optimal plan is this example case.

Case 3

However, our algorithm cannot guarantee the generated plan is always optimal. In figure 3.3, we consider a query which is the same as r_4. The way to enforce the query join path $S \bowtie E \bowtie C$ in our generated plan is labeled with bold boxes. The other way to enforce it is to enforce r_7 at P_A first, and send the results to P_T to enforce R_2 and join with R_3. As the latter plan requires one more join and data transmission operation, our plan to enforce

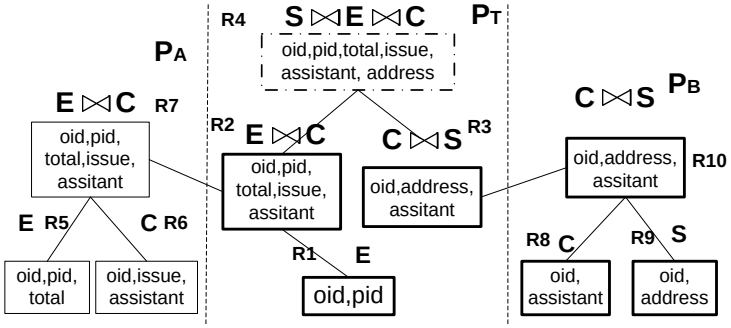

Figure 3.3: An non-optimal example

the query join path is better. However, the latter plan has no missing attribute, and our plan still need to enforce r_7 again to retrieve attributes $\{total, issue\}$. Therefore, our plan is not optimal in this case. Compared to the optimal plan, our generated plan just has one extra step which is r_1 join with r_3. Only in extreme situations, where the sizes of $E \bowtie C$ and $C \bowtie S$ are very large, but $S \bowtie E \bowtie C$ is very small, our plan can be better.

Complexity of the algorithm

As the algorithm 5 iteratively decomposes rules, if there are N_q rules relevant to the query q, the algorithm needs to examine all of them in the worst case. In addition, when finding the decomposition of relevant rules, it needs to find the cover of the missing attributes. If the maximum number of relevant rules on J_t-cooperative parties is N_r, the worst case complexity is $O(N_r^2 * C)$, where C is the constant to record operations among rules. The loop of the priority queue to find the enforcement of required join paths has at most N_q join paths to be explored, and it is no more expensive than the previous part. Therefore, the overall worst case complexity is $O(N_q * N_r^2 * C)$. Usually N_r is much smaller than N_q.

To sum up, these simple example cases show our query planning algorithm is effective to find a good query plan for an authorized query.

3.5 Query planning with authorization rule changes

In this section, we consider the problem of authorization rule changes when queries are being processed. We have the above algorithm to generate a query plan among the cooperative parties based on the consistent closure and the algorithm to check the rule enforceability. While a generated query plan is being executed, if the authorization 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 discussed here are revocations on access privileges. We discuss how to adapt the query execution with the authorization rule changes below.

3.5.1 Snapshot solution

Since the biggest concern of this problem is that the authorization 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 authorization 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 authorization rules it caches to other parties with the mark of 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 as usual. If the authorization rules on a party are updated, the party may lack the authorization to perform the steps in the query plan. In this case, it can use the snapshotted rules with the marks of q to retain the authorizations from other parties. Other cooperative parties will also honor the snapshotted rules even if currently the authorization 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 executes 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.

3.5.2 Dynamic planning

Instead 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 whenever a step cannot be executed because of the rule 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. Here, we also have two types of revocations.

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 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.

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 is not answerable anymore, 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.

Chapter 4: Rule enforcement with trusted third parties

In previous chapters, we studied the problems under the assumption that there is no trusted third party available for secure computation over sensitive data. In this chapter, we assume trusted third parties are available for rule enforcement and query execution. We discuss various problems by considering the third parties.

4.1 Enforcement of rules with third party as a join service

As discussed above, we knew there are some rules and attributes in the given set of rules cannot be enforced by any consistent query plan. In such case, we may need a trusted third party (TP) to be involved in the rule enforcement. We assume that a **third party (TP)** is not among the cooperative parties and can receive authorized information from any cooperative party. We assume that the TP always performs required operations honestly, and does not leak information to any other party. In our model, we assume the trusted third party works as a service. That is, each time we want to enforce a rule, we need to send all relevant information to the third party, and the third party is only responsible for returning the join results. After that, the third party does not retain any information about the completed join requests.

We investigate the problem of rule enforcement. With the existence of a third party, it is always possible to find an enforcement plan for a given authorization rule. We aim to minimize the amount of data to be released to the third party because of the overhead of sending data to the third party, latency/expense of third party computations, and potential exposure of data during transmit or while at the third party. We start with consideration of communication cost, and then discuss the overall cost of enforcing an authorization rule.

We can formulate the problem as follows: Given a *target rule* r_t to be enforced at

the third party, and a set of cooperative parties $\{P_1, P_2...P_m\}$ together with a set of rules $R = \{r_1, r_2...r_n\}$ given to these parties, we look for the minimal amount information from set R to be sent to TP to enforce r_t and the minimal overhead of using TP. We assume there exists a pre-processing step to tell which rules can be enforced without the third party. We prefer to enforce authorization rules among cooperative parties directly. We assume these directly enforceable rules are materialized which means the data regulated by the rules is always available so we do not have to enforce them from the beginning. We say the "rules" in the following refers to the enforced information authorized by these rules.

First of all, we filter out the rules in R which are not relevant to target ruler_t or not enforceable. Next, we look for which relevant rules should be selected. To minimize information release to TP, we prefer a rule with a longer join path since it usually carries less information. For instance, if the target join path J_t is $C \bowtie E \bowtie S$, it is desired to send rules on join paths $C \bowtie E$ and $E \bowtie S$ to TP instead of $C \bowtie E$ and S because join path $E \bowtie S$ usually has fewer tuples than S. In addition, as the given rules are "upwards closed", a higher level relevant rule always includes all the attributes of the lower level rules that are relevant to it. Therefore, if we only need certain attributes, instead of selecting a low level rule, we can select a high level rule with projection operation. For example, assuming we need to enforce a rule r_t with join path $R \bowtie S \bowtie T$ on the third party, all the attributes in relations S and T are already provided by a relevant rule r_s, and we look for the next rule. There are two relevant rules to choose from where one is on the basic relation R and the other is on the join path of $R \bowtie S$. We only need attributes in relation R, but we prefer to use the rule on $R \bowtie S$ because of fewer tuples. Hence, we apply $\pi_R(R \bowtie S)$ to get needed attributes. According to these principles, we only look for the *Top level relevant rules* on cooperative parties to enforce the target rule t_t. If a relevant rule of r_t is not relevant to any other higher level relevant rules, it is a top level relevant rule. We call the top level relevant rules as **Candidate Rules** in the following. By considering only candidate rules, we can largely reduce the number of rules to be considered to choose from. We have the following finding between the maximum number of candidate rules on a single party and

the join path length of the target rule.

Theorem 9. *Given a target rule r_t with join path length n_t, the maximum number of candidate rules on one party is n_t.*

Proof. Firstly, we assume the join path of r_t is a chain type of join so that each join operation between relations R_i and R_j is performed on a different join attribute. In such scenario, the maximum number of candidate rules is n_t. It happens when all the relevant rules on the party P are rules on basic relations, and none of them can join with each other, so there are at most n_t such rules. If there is one more relevant rule r_e on the same party, then the join path of r_e at least includes two relations. Thus, r_e is a candidate rule, but the total number of candidate rules decreases as the two rules with relations in J_e are no longer candidate rules. If two relevant rules can join, the resulting rule is also a relevant rule, and it can replace these two rules as the new candidate rule. Therefore, if there are more than n_t rules relevant to r_t, the number of candidate rules is less than n_t. If the join schema of the join path J_t is not chain type, then at least two relations in JR_t joining on the same attribute. Consequently, the relevant rules of r_t have more chances to be joined together to form higher level relevant rules. In the extreme case, where all the relations in JR_t joining over the same attribute, then there is only one candidate rule since all the relevant rules can join with each other. To conclude, the maximum number of candidate rules on a cooperative party is n_t. □

4.2 Minimizing communication cost

In this section, we consider the problem of choosing the proper candidate rules to minimize the amount of information sent to the third party. As these information is supposed to be sent to TP through the network, we call it as the *communication cost* of enforcing the target rule. In our cost model, we assume attributes are of the same lengths, so the amount of information is quantified by sum of the number of attributes picked from each rule multiplied by the number of tuples in that selected rule. Thus, we want to minimize

$Cost = \sum_{i=1}^{k} \pi(r_i) * w(r_i)$, where r_i is a selected rule, k is the number of selected rules, and $\pi(r_i)$ is the number of attributes selected to be sent, and $w(r_i)$ is the number of tuples in r_i. We begin with the model that all candidate rules have the same number of tuples to obtain useful theoretical results. Later, we discuss the model where the numbers of tuples are different which is realistic.

4.2.1 Rules with same number of tuples

Our goal is to minimize the total amount of information sent to the third party. In our cost model, only the numbers of the attributes and tuples influence the cost of a selected rule. However, it is usually difficult to know the exact number of tuples in the join paths. To begin with, we first assume all the candidate rules have the same number of tuples no matter what are their join path lengths. Under such assumption, we can convert our problem into an unweighted set covering problem.

We first assume the candidate rules have the same $w(r_i)$ value. An authorization rule has two things need to be concerned for enforcement (a) the join path and (b) the attribute set. Assuming the target rule r_t has the attribute set $A_t = \{A_1, A_2...A_m\}$, and the join path J_t with $JR_t = \{R_1, R_2...R_k\}$, we treat the join attributes from different relations in A_t as different attributes marked by their relations. Other attributes are also labeled by their relation names. To find the candidate rules that can provide enough information to enforce r_t, we map each labeled attribute to only one candidate rule so that all of these attributes can be covered. Once we get such a mapping, we have one solution including the picked rules and projections on desired attributes. Among these solutions, we want the minimal cost solution according to our model. Since we assume all the top level rules have the same number of tuples, it seems that the total cost of each candidate solution should always be the same.

However, it is not true because the join attributes appearing in different relations are merged into one attribute in the join results. We can consider the example in Figure 4.1. The boxes in the figure show the attribute set of the rules, and the join paths and rule

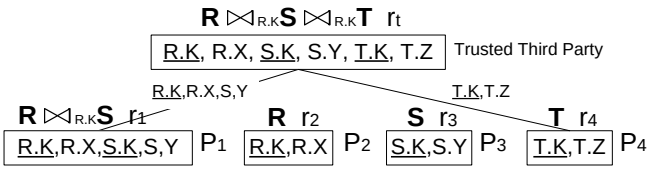

Figure 4.1: An example of choosing candidate rules

numbers are indicated above the boxes. There are four cooperating parties indicated by P_i and one TP, and the three basic relations are joining over the same key attributes $R.K$. Among the 4 candidate rules, if we select r_2, r_3 to retrieve the attributes $R.X$ and $S.Y$ (non-key attributes), we need to send $R.K$ and $S.K$ which are their join attributes to the third party as well. Whereas, if we choose r_1, then we only need to send 3 attributes as $R.K$ and $S.K$ are merged into one attribute in r_1. Thus, choosing a candidate rule with longer join path may reduce the number of attributes actually being transferred. Fewer rules mean fewer overlapped join attributes to be sent (e.g., $R.K$ in r_1 and $T.K$ in r_4 are overlapped join attributes). In addition, selecting fewer rules can result in fewer join operations performed at the third party. Since we assume the numbers of tuples in candidate rules are the same, the problem is converted to identify minimal number of candidate rules that can be composed to cover the target attribute set. It is basically an unweighted set covering problem and hence NP-hard.

Theorem 10. *Finding the minimal number of rules sent to the third party to enforce a target rule is NP-hard.*

Proof. Consider a set of elements $U = \{A_1, A_2, ..., A_n\}$ (called the universe), and a set of subsets $S = \{S_1, S_2, ...S_m\}$ where S_i is a set of elements from U. The unweighted set covering problem is to find the minimal number of S_i so that all the elements in U are covered. We can turn it into our rule selection problem. For this we start with the attribute set $\{\underline{A_0}, A_1, A_2, ..., A_n\}$, where A_0 is the key attribute of some relation R and A_i's are non-key attributes of R. For each $S_i \in S$, we construct a candidate rule r_i on R with the

attribute set $S_i \bigcup \{A_0\}$ and assign it to a separate cooperative party. Therefore, if we can find the minimal set of rules to enforce some target rule r_t in polynomial time, the set covering problem can also be solved in polynomial time. □

4.2.2 Rules with different number of tuples

In general, the numbers of tuples in the relations/join paths are different, and they depend on the length of the join paths and the join selectivity among the different relations. *Join selectivity* [9] is the ratio of tuples that agree on the join attributes between different relations, and it can be estimated using the historical and statistical data of these relations. In classical query optimization, a large number of works assume such values are known when generating the query plans. We also assume that this is the case. Therefore, we can assign each candidate rule r_i with a cost $cst_i = w(J_i) * \pi(r_i)$, where $\pi(r_i)$ is the per tuple cost of choosing rule r_i, and $w(J_i)$ is the number of tuples in join path J_i.

The problem is similar to (but not identical to) the weighted set covering problem. Given the universe $U = \{A_1, A_2, ..., A_n\}$ and the set of subsets $S = \{S_1, S_2, ...S_m\}$, we try to find a cover C of U which has the minimal overall cost $cost(C) = \sum_{S_i \in C} cst(S_i)$. Although the weighted set covering problem is also NP-hard to solve, the well known greedy algorithm can achieve a ln^n-approximation and it is proved to be the best-possible polynomial time approximation algorithm for the problem [56].

In our problem, once some attributes are covered by previously chosen rules, the following chosen rules should project out these attributes so as to reduce cost. This is closer to the pipelined set covering problem [57], which is to minimize the $cost(C) = \sum_{i=1}^{k} w(S_i)|U - \bigcup_{j=1}^{i-1} S_j|$. Unfortunately, our problem is still slightly different. Firstly, the costs of previously chosen rules are not applied on the elements that are not covered by these rules in our problem. Secondly, the key (join) attribute of a rule always needs to be selected with the rule whether or not it is previously covered by other rules. Therefore, our cost function should be as follows.

$$cost(C) = \sum_{i=1}^{k} w(S_i)\pi(S_i), \pi(S_i) = \begin{cases} |S_j \bigcap (U \setminus \bigcup_{j=1}^{i-1} S_j)|, & \text{if } (key(S_i) \notin \bigcup_{j=1}^{i-1} S_j) \\ |S_j \bigcap (U \setminus \bigcup_{j=1}^{i-1} S_j)| + 1, & \text{if } (key(S_i) \in \bigcup_{j=1}^{i-1} S_j) \end{cases} \tag{4.1}$$

Corollary 1. *Finding the minimal amount of information sent to the third party to enforce a target rule is NP-hard.*

Proof. Based on Theorem 10, if we have a polynomial algorithm to find the minimal amount of information with rules of different costs, we can assign the same cost to each candidate rule so as to solve the unweighted version of the problem. □

In the weighted set covering problem, the best known greedy algorithm finds the most effective subset by calculating the number of missing attributes it contributes divided by the cost of the subset. In our case, we also want to select the most effective attributes from the available subsets. Similar to the weighted set covering algorithm which selects the subset S_i using the one with minimal $\frac{w(S_i)}{|S_i \setminus U|}$, we select the rule with the minimal value of $\frac{w(S_i) * \pi(S_i)}{|S_j \bigcap (U \setminus \bigcup_{j=1}^{i-1} S_j)|}$, where $\pi(S_i)$ is defined in equation (4.1).

In our problem, with one more rule selected, the third party need to perform one more join operation, and possibly one more join attribute need to be transferred to the third party. Therefore, when selecting a candidate rule, we examine the number of attributes this rule can provide and the costs of retrieving these attributes. In the second case of $\pi(S_i)$ in equation (4.1), the cost of one one extra attribute is added. However, if this selected rule can provide many attributes to the uncovered set, the cost of this additional attribute can be amortized. This makes the algorithm prefer rules providing more attributes and results in less number of selected rules which is consistent with our goal. We present our greedy algorithm in Algorithm 6.

Algorithm 6 Selecting Minimal Relevant Data For Third Party

Require: The set R of candidate rules of r_t on cooperative parties
Ensure: Find minimal amount of data being sent to TP to enforce r_t

for Each candidate rule $r_i \in R$ **do**
  Do projection on r_i according to the attributes in r_t
  Assign r_i with its estimated number of tuples t_i
The set of selected rules $C \leftarrow \emptyset$
Target attribute set $U \leftarrow$ merged attribute set of r_t
while $U \neq \emptyset$ **do**
  Find a rule $r_i \in R$ that minimize $\alpha = \frac{w(S_i)*\pi(S_i)}{|S_j \bigcap (U \setminus \bigcup_{j=1}^{i-1} S_j)|}$
  $R \leftarrow R \setminus r_i$
  for Each attribute $A_i \in (r_i \bigcap U)$ **do**
    $cost(A_i) \leftarrow w(S_i)$
  $r_i \leftarrow \pi_U(r_i) * w(S_i)$
  $U \leftarrow U \setminus r_i$
  $C \leftarrow C \bigcup r_i$
Return C

Since the extra cost for a rule is at most one join/key attribute for a rule r_i, our greedy algorithm has a very tight bound. We show it in the following theorem.

Theorem 11. *The greedy algorithm finds the amount of information that is 2-approximation of the minimal amount of information to enforce a target rule.*

Proof. If we do not consider the issue of join attributes, then for each A_i, we select it from the subset r_j where $A_i \in r_j$ with minimal $w(S_j)$. As we select each element with minimal possible cost, the total cost that covers U is minimal. We denote this cost as MIN. Next, we denote the optimal solution of finding minimal information to enforce a rule as OPT, and it is obvious $MIN <= OPT$ because at least we need to cover all the required attributes. Then if we use the greedy algorithm 6, in the worst case each attribute A_i is retrieved with cost of $2^*w(S_j)$. That is, each attribute is selected from a separate rule and the associated join attribute is always redundant. We denote our solution as GRD, and $MIN <= OPT <= GRD <= 2MIN <= 2OPT$. Therefore, our solution is 2-approximation. □

Further reducing the communication cost

Instead of send all the relevant relational data to the third party and let the third party does all the join operations to enforce the target rule, we can further reduce the amount

Table 4.1: Example rules for enforcement

Rule No.	Authorized attribute set	Join Path	Party
1	{oid,pid,total}	E	P_E
2	{oid, pid, total}	$E \bowtie_{oid} S$	P_E
3	{oid, address, delivery}	S	P_S
4	{oid, pid, total, address, delivery}	$E \bowtie_{oid} S$	P_S

of information being transferred in some special cases. For instance, suppose we have the following set of rules on the two parties.

In Table 4.1, neither of the party can perform the join operation of $E \bowtie_{oid} S$. If r_4 needs to be enforced, a third party is required. According to the above discussion, both party P_E and P_S can send their own data of r_1 and r_3 to the third party, and the third party can perform the join operation and return the required result to party P_S so as to enforce r_4.

However, in this example, we can further reduce the involvement with the third party. In fact, both parties can send only the attribute *oid* to the third party. The third party perform the join operation only with two columns of *oid* and keeps only the values that appear in both columns. Such information is sent back to P_E, and P_E performs another local join to eliminate the tuples in its basic relation E whose value of *oid* do not appear in the result got from the third party. As a matter of fact, this step enforces r_2 and such information is then sent to P_S and the party P_S can perform another join operation with r_3 to enforce r_4. In this scenario, only two columns of data are sent to the third party.

In this example, such operation is possible because of the rule r_2, P_E is authorized to get the join result of $\pi_{oid}(E \bowtie_{oid} S)$ from the third party. Otherwise, this operation violates the authorizations. This type of operation can also happen on a multi-way join where several cooperative parties send their candidate rules to the third party. If a party P_C is similar to P_E which is also a J_t-cooperative party of r_t, then it is allowed to get only the join attribute of the resulting join path J_t from the third party. On the contrary, if P_C is not a J_t-cooperative party, then it is not allowed to get the join result on J_t. In such case, P_C has to send whole related attributes of its candidate rule to the third party.

This type of join operation is similar to semi-joins [58]. The requirement is that the

cooperative party must be a J_t-cooperative party. In such cases, the party has a given rule on the join path J_t. However, such a rule cannot be enforced among cooperative parties. If we want to minimize the information being sent using this type of operation, then we can assign new cost values for such candidate rules on J_t. Compare to other relevant rules with the value $\alpha = \frac{w(S_i)*\pi(S_i)}{|S_j \bigcap (U \backslash \bigcup_{j=1}^{i-1} S_j)|}$, the metric value α of a J_t top level rule is just $\alpha = \frac{w(S_i)}{|S_j \bigcap (U \backslash \bigcup_{j=1}^{i-1} S_j)|}$. This type of optimization may largely reduce the amount of information need to be sent to the third party, but it also brings overheads for the cooperative parties to perform more operations. If we choose to use such join mechanism, the algorithm prefers to choose from these rules because of the smaller α value. Again, depending on the given rules, this type of candidate rules may not exist.

4.3 Minimizing the overall cost

In this section, we consider the overall cost of using the third party, which includes both the communication cost and the computing cost at the third party. The *computing cost* is the cost of the third party to perform all required join operations. Given a set of selected candidate rules, minimization of computing cost is similar to the classical query optimization problem which is an NP-complete problem even if considering only the nested loops join method [59]. The computing cost is difficult to estimate because of the different access methods which can be index scan or sequential scan, and the different join methods such as nested loop, sort-merge, hash-join, etc. Moreover, the join orders and the size of the input data and join results also influence the computing cost. Therefore, we first make some assumptions in this section so that we can estimate the computing cost.

We assume the number of tuples of the selected candidate rules are known as $w(J_i)$, and we also assume all the joins are done with nested loop method. Given n rules, the third party always does $n-1$ sequential join operations. In addition, we assume the relations have indices on their join attributes. For a nested loop join with two input relations, the cost can be estimated as: $Access(Outer) + (Card(Outer) * Match(Inner))$, where $Access(R)$

is the cost of access the relation R which is usually a function of the number of data blocks in R. We can have $Access(R) = Card(R)/Page(R)$, where $Page(R)$ is the number of tuples of R in one page. $Card(R)$ is the number of tuples in R which is $w(J_R)$, and $Match(Inner)$ is the cost of finding a matching tuple in the inner relation which is usually a small constant if the indices are available. Obviously, we always prefer using the smaller input relation as the outer relation. In addition, as we need to perform $n-1$ joins, we keep the intermediate join results of the previous joins. The result of a join can be estimated as $Access(Result) = Card(Outer) * Card(Inner) * SelectivityFactor * Page(Result)$, where $SelectivityFactor$ is the estimate of what fraction of input tuples will be in the result, and it is known. Therefore, the total computing cost including $n-1$ join operations can be estimated as:

$$CompCost = Access(R_1) + \sum_{i=1}^{n-1}(((Card(JR_i) * Card(R_{i+1}) * SF_{JRi,R_{i+1}} * Page(JR_{i+1}))$$

In the above equation, R_1 is the selected rule with least cost $w(J_i)$. JR_i is the join results of the rules from R_1 to R_i, and $SF_{JRi,R_{i+1}}$ is the selectivity factor. Thus, to minimize $CompCost$, it is preferred to have fewer number of operations, and for each operation, the relation with smaller cardinality should be used as the $Outer$ relation. To that end, we process join operations in the ascending order of the cardinality of the selected rules. In fact, this only gives us a lower bound of the $CompCost$, because these rules cannot be processed in an arbitrary order due to the join schema. Given a set of candidate rules selected by the previous algorithm, we calculate the computing cost using the above model. In most cases, the communication cost and computing cost are closely related and with less information and fewer rules being sent to the third party, the computing cost can also be reduced.

Thereby, the overall cost can be calculated as $OverCost = CommCost + \alpha * CompCost$.

We put α here since the unit cost of the two are usually different. In communication cost the unit cost can be the network transmission cost and computing cost is mainly the disk I/O and CPU costs, which can be smaller than network cost but may not always be true. If the communication cost is the dominant cost, we still use the above discussed algorithm to pick up the candidate rules. However, if the computing cost is dominant, we may prefer minimal number of rules instead of minimal amount of information (fewer join operations). To that end, when finding the minimal overall cost, we run two greedy algorithms. One is the Algorithm 6, and the other is the greedy algorithm for unweighted set covering problem, where the algorithm always picks the rule that covers the maximal number of uncovered elements. After running the two algorithms and calculating the overall cost with the two algorithms, we pick the solution that gives the smaller overall cost.

4.4 Evaluation

We evaluated the effectiveness of our greedy algorithm against brute force via simulations. For the brute force algorithm, we first filter the rules to obtain the candidate rules and apply projections on these rules. In order to select the candidate rules, the brute-force algorithm tests the power set of the filtered candidate rules, and for each rule combination, uses the minimal possible cost value for each attribute in the target set. Therefore, we obtain the minimal costs for each possible rule combination.

4.4.1 Minimal communication costs comparison

In this simulation evaluation, we use a join schema with 8 parties. Three of the parties share the same key and join attributes, and the remaining parties are assumed to join in a chain type of schema with all different join attributes. According to the join schema, we generate the rules with various join paths and attribute sets, and each party could have one rule on each distinct join path. There are 8 basic rules corresponding to the basic relations owned by the parties. We generate total number of rules varying in the range from 9 to 208, and we randomly generate a target rule with join path length of 4. We define the cost

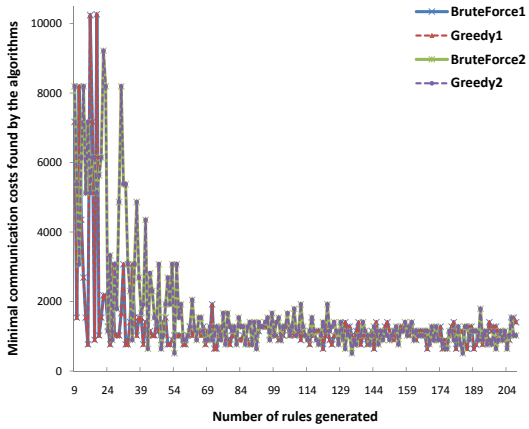

Figure 4.2: Minimal communication costs with different schemas

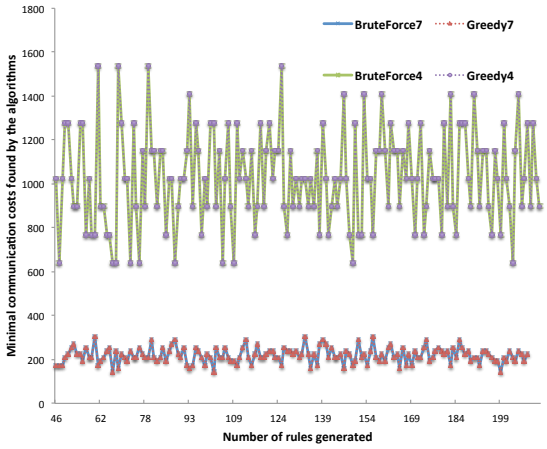

Figure 4.3: Minimal communication costs found by two algorithms

as a function of the join path length, basically $w(J_i) = 1024/2^{length(J_i)-1}$. In other words, we assume as the join path length increases by one, the number of tuples in the results decreases by half. We use this model because (a) the numbers of tuples in join paths highly depend on the real data and difficult to be modeled, (b) such a model is an extreme case that favors the candidate rules with longer join paths. In contrast, we also use the model where all the rules have same number of tuples (Section 4.1) as the other extreme case in later simulations to compare the differences. We use a Mac Pro machine with 4 cores 2.2G Hz with 6GB memory to perform all the simulations.

We then perform the similar simulations using a different join schema to see if schema affects the results. For one such case, 6 of the relations can join with each other and the other 2 relations form a chain. This is more like a star type of join. Figure 4.2 shows the results of the simulations. The legend of "BruteForce1" means the brute force algorithm is tested with the original join schema, and "Greedy2" means the greedy algorithm is tested using the new join schema. Figure 4.2 also shows the trends of the costs as number of rules increases. At the beginning, when the number of rules is small, usually the costs are high. It is because we have to choose the rules with shorter join paths which have higher costs. As the number of rules goes larger, the costs become less and stable. To better show the

results, in figure 4.3 and figure 4.7, we show the number of rules beginning from 46. The results show the different join schemas do not affect much.

Using the original schema, we then randomly generate a target rule with join path length of 7. Figure 4.3 shows the comparison between two algorithms. In fact, the two algorithms generate almost the same results. In Figure 4.3, the legend of "BruteForce4" indicates the target rule has the join path length of 4, and brute force algorithm is used. Among these solutions, in less than 2% of the cases the two algorithms produce different answers. In addition, the maximal difference between them is just 5%. The results also indicate the join path length of the target rule affect the costs, but the algorithms give similar solutions independent of the join path length.

4.4.2 Running time comparison

As the two algorithms have different complexity, we test both with different number of candidate rules. When the number of rules is small, the brute-force algorithm is also quick, but as the number goes over 16, it shows the exponential nature of the algorithm. In contrast, the greedy algorithm always runs fast. Figure 4.4 shows the comparison results in log_2 scale. We can learn from this result that if the total number of candidate rules is more than 16 rules, the greedy algorithm is required.

In Figure 4.5 we show the relationship between the total number of generated rules and the number of candidate rules. We run the simulations with various parameters. In the figure, bar "len7,node12" means the target join path length is 7 and there are 12 cooperative parties in total. As we can see this data series have a high probability of getting more than 16 candidate rules. The greedy algorithm is required in these cases.

4.4.3 Minimal overall costs comparison

We also conducted simulations to evaluate the solutions produced by two algorithms considering the minimal overall costs. As the significance of the communication cost and computing cost can be different, we test with α. $\alpha = 0$ means that the communication cost

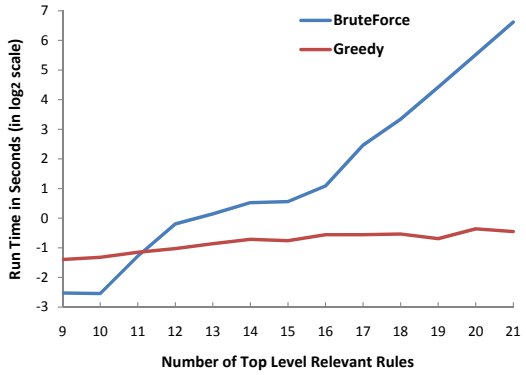

Figure 4.4: Time comparison between two algorithms

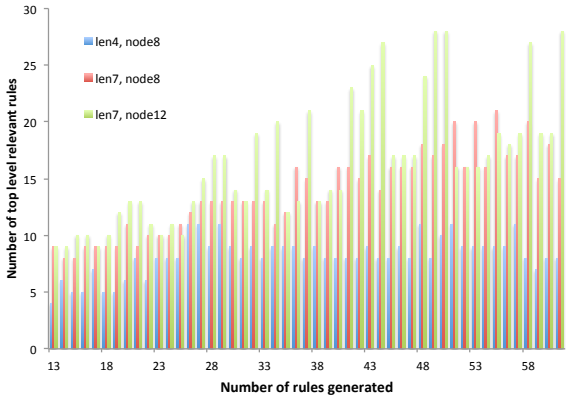

Figure 4.5: Numer of candidate rules

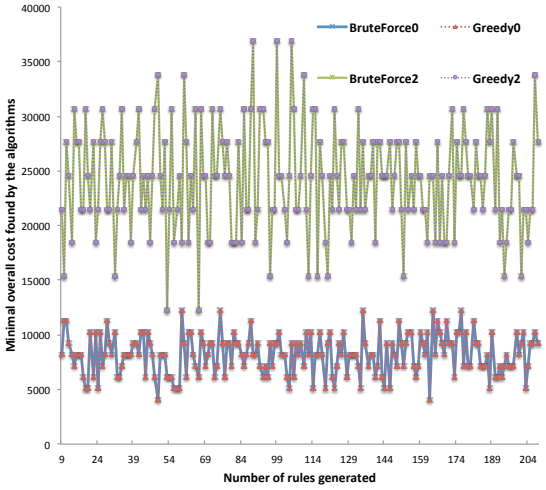

Figure 4.6: Minimal overall costs with same cost model

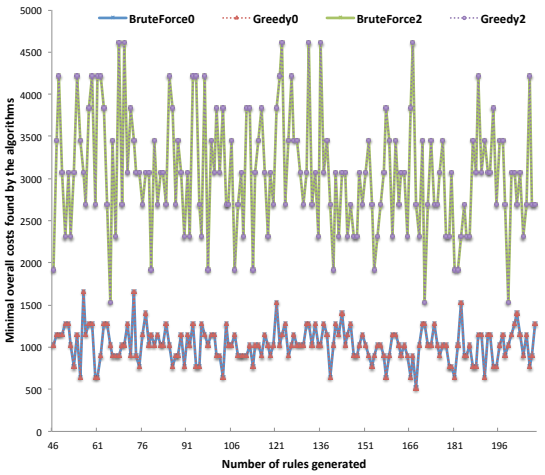

Figure 4.7: Minimal overall costs with half cost model

is dominant and results are similar to the above. In contrast, $\alpha = 2$ means the computing cost was dominant. Figure 4.6 depicts the comparison between two algorithms. In the legends, "Greedy0" means greedy algorithm is tested and $\alpha = 0$. For this simulation, we use the model that all the rules have the same number of tuples. Not surprisingly, the cost values are high under such a model.

We again perform similar simulations using the original half cost model, and Figure 4.7

shows the results. In both figures, the solutions found by the two algorithms are very close. In fact, there are only 27 different cases among around 800 runs of the comparisons where the two algorithms give different results. Moreover, we find most of the different cases happen when the number of rules is less than 110. That is, with plenty number of generated rules, the greedy algorithm is very close to give the optimal answer with the maximal difference of only 9%. In most different cases, the greedy algorithm gives answer that is 3 to 5 percent more compared to the optimal one. In addition, both figures show when $\alpha = 2$, the fluctuations become larger. It indicates that two types of costs are closely related and the overall costs show the accumulated effects.

4.5 Third party with storage capability

In previous sections, we consider the third party works as a join service. Under such assumption, the third party takes various input information from the cooperative parties and does the join operation and gives the join results to the target party, and we assume the third party does not retain any computation results. However, in many cases, the third party may have the ability to cache its computation results, and the cooperative parties can take advantage of it so as to reduce the cost of enforcing other rules.

In this section, we consider the model where the third party can store its computation results for the future tasks. As the cached data can become obsolete after some time period, we assume there are proper mechanisms [60, 61] for synchronize the data between the data owners and the trusted third party. Hence, the third party can distinguish which data is out of date. In addition, the storage capability of a third party is usually limited. Therefore, different cache replacement mechanisms can be applied. In fact, the optimal caching policy for the third party can be different from the file cache [62] and process cache [63] mechanisms, and it can be complicated. For instance, the third party prefers to keep the results of enforced rules with shorter join paths since they have better chances to be reused. We leave the problem of finding optimal caching policy as the future work, and we assume a good cache mechanism is applied at the third party. When a request for

enforcing rule r_t comes in, the third party first examines its cached data to see which is useful for the rule enforcement. Then the selected data is locked and cannot be swapped out from the third party storage until r_t is enforced.

By taking the advantage of the intermediate results at the third party, we can send less data to the third party to enforce a target rule. Similar to previous mechanisms, the third party filters its cached data to find the data from the relevant rules of r_t. Based on that, we try to find out the minimal information that is need to be enforced by the third party needs. We call the remaining set of attributes that are need to be enforced by the third party as **minimal required attribute set**. To obtain that set, we organize the cached information at the third party. All such information is put into a relevance graph, and we can use the rule consistency algorithm to generate the closure of these available data. This step results in the obtainable information of one or several candidate rules on the third party. These candidate rules represent the maximal existing information that a third party can use to enforce r_t. Since the rule consistency algorithm usually finishes in a very short time, we do not consider the cost of finding this useful information on the third party. However, performing the join operations over these data incurs computing costs.

The next task is to use the third party to enforce minimal required attribute set so that r_t can be totally enforced. Although the minimal required attribute set may not form a valid join path, we can still the same algorithm to find what is the minimal information needed from the cooperative parties. In fact, we can think the problem in the following way: to find the minimal costs of rules to enforce r_t, there are relevant rules with extra small costs which are the cached information at the third party. To enforce the remaining attribute set, what is a minimal cost of rule selection. Consequently, it becomes a sub problem of the problem discussed in last section. Thus, the problem is also NP-hard, and we can use the same greedy algorithm. We have the following assertion as well.

Theorem 12. *To minimize the cost of enforce a rule via third party, the data being sent from cooperative parties should only cover the minimal required attribute set.*

Proof. The third party takes advantage of all the cached information it has, so if an attribute

is available at the third party, it does not appear in the minimal required attribute set denoted as A_M. That is, the minimal required attribute set is unique and determined. Then we take A_M as the new target attribute set to find the minimal relevant information to cover it. For the set of relevant rules on cooperative parties denoted as S_t which is transferred to the third party to enforce r_t, it is adequate to project each rule in S_t on the attribute set A_M. The amount of resulting set of information is no more than S_t. Thus, choosing the set of information only covering A_M minimizes the total cost of enforcing the rule.

□

4.5.1 Partially trusted third parties

In the above, we assumed that third party is fully trusted, and each party can send all of their data to the third parties in plaintext. However, it may be not easy to find a third party that is totally trusted by all of the cooperative parties. We can assume a public third party service is "honest but curious", which means the third party does work correctly but may look into the data it receives. Cooperative enterprises usually do not want to reveal the information to the third party. In such a scenario, cryptography can be used in these operations. Before sending data to these third parties, data need to be encrypted so that the third party cannot read the information.

Assuming full trust between cooperative parties, we can assume that all cooperative parties share the same encryption key, and the cooperative parties encrypt all the data sent to the third party first. The relational data is encrypted on a per tuple basis so that tuples can be matched based on the join attribute. To that end, the join attribute of the tuple need to be encrypted separately from the rest of the attributes. Thus, the same values on the join attribute have the same cipher text, so there is no need for the third party to decrypt the tuples and it can decide whether two tuples from two data owners should be merged and output as a resulting tuple in the join result. In this way, a partially trusted third party performs the join operation over the cipher text and it cannot reveal the information

from the process. Similarly, the J_t-cooperative parties can use semi-join mechanism and send only encrypted join attributes to the third party.

4.6 Multiple third parties

Usually, one third party is good enough to do all the join operations if it acts as a join service. However, in some cases, one trusted third party is inadequate and undesired. The cooperative parties may not share the same third party in terms of business relationship such as cost or trust issues. In addition, one third party may be inadequate from performance perspective. A single third party may be unable to provide efficient service because it is remotely located from some of the cooperative parties.

From security perspective as well, we may need to isolate the data from different cooperative parties. Usually it is not the case that all the parties are cooperating with each other, and a cooperative party may have conflicts of interest with others. In other words, the parties may form several collaboration groups with conflicts between them. For example, an e-commerce company A may collaborate with a shipping company S, but it does not want another e-commerce company B (its competitor) to see any of its data. In such case, if the party A sends its data to a trusted third party, it expects that the competing party B never has any access or interaction with the same third party, and vice versa. It is also the third party's responsibilities to enforce these policies so that whenever it holds any data from party A, it denies any access (read/write) from party B. Therefore, if both A and B need to be sent to the third party to accomplish join operations with other data, the data from these conflicting parties should never be put on the same party including the third party. Hence, we may need multiple third parties to isolate the data. Here the conflict between A and B is just pairwise, but the conflicts can happen between multiple parties which are multi-way conflicts.

Similar to the cooperative parties, the third party may also need to be prevented from obtaining certain sensitive associations via local computation. In such case, if a third party gets various data from several cooperative parties and such data can be potential combined

to derive the sensitive association, then these pieces data from several cooperative parities forms a multi-way conflict. Multiple third parties are also required. For instance, the association in the join result of $R \bowtie S \bowtie T$ is undesired but any join results of two tables from $\{R, S, T\}$ is allowed, then data from $\{R, S, T\}$ forms a three-way conflict.

4.6.1 Multiple third parties for data isolation

First of all, we have the following definition for the data isolation requirement in our scenario.

Definition 11. *A piece of data* **conflicts** *with the other, if they are owned by two competing parties or the derivable information from them is undesired. The* **data isolation** *requirement is satisfied, if there is no conflict exists on any single party.*

From the definition, it is clear that there is no conflict for each given authorization rule. That is, for relations appear in a join path, these data do not conflict with each other. Otherwise, such a rule will not be made since all the rules are desired by the business requirements.

Basically, the conflicts happen for two reasons: competing parties and undesired derived results. The competing relationships can be obtained when the parties and join schema are given. In contrast, the undesired derived results are only given in the similar form of negative rules and only the prohibited results are explicitly given. Therefore, there is a need to figure out these multi-way conflicts. Considering the possible conflict at the third party, we interpret the conflicts using Chinese wall policy. If a third party obtains a piece of data from one party, it can never get data from its competitors. Similarly, for the undesired computation results, no third party can get all information that causes a multi-way conflict. Under the semantic of Chinese wall policy, if there is a n-way conflict, then a third party can obtain up to $n - 1$ pieces of data but it cannot access the last piece of the data.

Conflict graph

To minimize the costs of using the third party usage, a basic problem is what is the minimal number of third parties required to satisfy the security requirements. In the following, we

examine this problem in details.

We represent the conflict relationships using a graph. In such a graph $G_C =< V, E >$, each vertex $v \in V$ represents a rule, and an edge $e \in E$ connects two vertices if the information regulated by the two rules conflict with each other, and such a graph is called **conflict graph**. The data from competing parties obvious creates pairwise conflicts between them. However, the difficult problem is to use the conflict graph to represent the multi-way conflicts.

The multi-way conflict is usually given in the form like a negative rule discussed in Chapter 2. To simplify the discussion, we consider the conflicts in terms of join paths below, and the attribute set can be considered by extending the mechanism. Similar to negative rules, if a join path is prohibited at third parties, the other join paths it is relevant to are prohibited as well. For example, if $R \bowtie S$ contains sensitive association, then a join path $R \bowtie S \bowtie W$ is also prohibited. To present multi-way conflicts using a graph, we need to first understand the relationship among the rules. Similar to the situation in Chapter 2, we consider the potential local computations that can be happened at the third party. We need to consider the rules on all of the cooperative parties together since all these rules can be potentially sent to the third party. Therefore, we run the rule consistency checking algorithm against all the authorization rules given to the cooperative parties and we have the relevance graph. During this process, some new rules are generated and these rules represent the potentially derivable information at the third party. Each rule in the relevance graph is also put into the conflict graph as a vertex. For a given prohibited join path, we check the relevance graph using the similar mechanism to the rule enforcement checking algorithm, so that we know all pairs of rules that can be composed to obtain the undesired join path. Hence, we put an edge between each found pair of the rules. In this way, we include all the conflict information in the conflict graph.

4.6.2 Static data allocation

To be conservative, once the third party obtains some data, the cooperative parties may assume the third party will always retain that data so the conflict data will never be sent the same party. In addition, we assume all the data on the cooperative party may potentially send their data to the third party. Under such assumptions, we want to figure out the minimal number of third parties required.

Once we create the conflict graph, we can relate our problem with the graph coloring problem. Each edge in the graph denotes the two connecting vertices should not be put on the same third party, and we assign colors to all the vertices in the graph. The vertices connected by the same edge cannot have the same color. We can consider each color represents a third party, and we can avoid all the conflicts by properly coloring the graph. Therefore, finding the minimal number of third parties is finding the chromatic number of the conflict graph. However, find a chromatic number of a graph is known to be NP-hard.

Corollary 2. *Finding the minimal number of third parities to resolve all conflicts among data is NP-hard.*

Finding the chromatic number

Finding the chromatic number of a given graph is well studied, and it is NP-hard. The best known approximation gives the bound of $O(n(\log\log n/\log n)^3)$ [64], where n is the number of the vertices in the graph. Such an approximation is based on randomized algorithm and gives the theoretical bounds of the problem, but it cannot be directly applied to the graphs. In general, another well known greedy algorithm which is called Welsh-Powell algorithm [65] is used to find the chromatic number of a graph. Given the conflict graph, we can first order the vertices according to their degrees, and we give different numbers to various colors. The greedy algorithm considers the vertices in the order $v_1,,v_n$ and assigns to v_i the smallest available color not used by v_i's neighbors among $v_1,,v_{i-1}$, and it adds a new color to v_i if needed. The solution generated by the algorithm uses at most $min(d(x_i)+1,i)$ colors, where $d(x_i)$ is the maximum degree of the graph. Using this greedy algorithm gives us the

static allocation to avoid conflicts at the third parties.

The above algorithm gives us a coloring result of the conflict graph. However, in our scenario, we may be able to improve the bound of the algorithm. In the previous chapters, we discussed how to organize rules in a relevance graph. As rules are given to different cooperative parties, the set of rules on the same party will never conflict with each other. Furthermore, the rules have the relevant relationships do not conflict with each other either. Based on these properties, we can conclude that the other upper-bound for the chromatic number is the number of the different parties, and we assume there are N cooperative parties. Therefore, the new upper-bound is $min(d(x_i) + 1, i, N)$. If the result of Welsh-Powell algorithm uses more than N colors, we redo the coloring of the graph by simply assign a different color to each individual party. The rules on the same party are marked using the same color. This gives us a colored graph to avoid conflicts as well.

Optimization with relevant rules

By taking advantage of the rule relevance relationship, we may further reduce the number of the colors used. In the relevance graph, we have edges across the different parties that connecting the rules with the equivalent join paths. If two rules have the equivalent join paths, then they cannot conflict with each other, and they can be applied with the same color. In addition, all the lower level rules connected to these rules in the relevance graph do not conflict with each other either. For instance, we have r_t, r_o on the join path J_t and a set of lower level rules $r_m ... r_n$ are connected to them in the relevance graph, then these rule never conflict with each other. It is because of the fact that if any of these rules form conflict, then there will be a prohibited join path given which is a sub path of r_t which is impossible since r_t is a given authorization rule.

Therefore, we can use the same color for J_t-cooperative rules as well as their connected lower level rules. To that end, we examine the relevance graph in a top-down manner. Beginning with the rules having longest join paths, we count the number of cooperative top level rules on each party. A **cooperative top level rule** is a rule that has an equivalent

join path as a rule on another cooperative party, and there is no higher level rule it relevant to that is also a cooperative rule.

We prefer the parties whose top level rules are all cooperative rules. If so, these cooperative top level rules can be colored using the colors of other parties. To that end, we sort the cooperative parties according to the number of top level rules they have. We begin with the party whose top level rules are all cooperative rules. Since each rule is cooperative with some other rules on remote parties, we can color this rule using the color of the remote party. Thus, we can color all the rules on this party using only the colors of other parties and reduce the chromatic number by one.

However, this coloring mechanism is not transitive. It only works among J_t-cooperative rules. For instance, if the party P_R has two rules on R and S respectively, and another party P_S has two rules S and W. We can color P_R with red, and P_S is blue. As two parties both have the rules on S, the rule S on P_S can also be colored as red. However, even if S does not conflict with W, we cannot transitively color W using red. It is because these three relations may potentially form a conflict group. Therefore, we iteratively examine the parties covered by cooperative top level rules, and try to color such parties use the colors of other parties. The process proceeds until all such parties are checked, and not all such parties can be colored using the colors of remote parties. To conclude, the process helps to reduce the number of colors needed, but it is not aimed to provide a optimal answer. Using this approach, we just have another alternative way to give a coloring solution, which is complementary to the existing approximate coloring algorithms. This approach is specific to our problem situation compared to general graph coloring algorithms.

To illustrate the mechanism, we use figure 4.8. Assuming there is a restriction that the join path of $R \bowtie S \bowtie W$ is not allowed to be generated. As there are three parties, 3 colors are enough to solve all conflicts and we assign 3 different colors to parties at the beginning. Since party P_R only has one top level rule and it is on the same join path as rule r_7 on P_S, we can use the color of P_s which is color 2 to color r_6. As r_1 is relevant to r_6, all of the rules on P_R can be colored using color 2. Thus, we reduce the number of colors by one. On

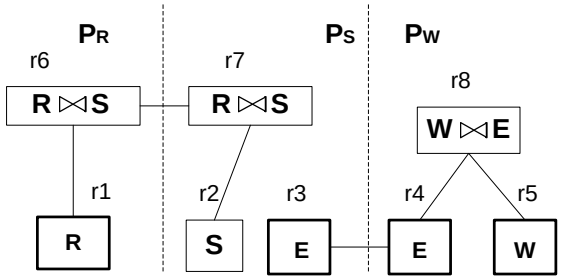

Figure 4.8: An example of reducing number of colors

party P_W, r_4 can be colored using color 2 as it is on the same join path as r_3. However, we cannot further reduce the number of colors.

4.6.3 Dynamic data allocation

Instead of assigning data statically to parties to avoid conflicts, a dynamic assignment may be desired in practice. When discussing the static allocation, we assume all the cooperative parties will send their data to the third party and the third party always retains the data it receives. However, these assumptions may not always be true. In some situations, the third party are trusted by the cooperative parties, and because of the limited storage capability of third party it swaps out some old data from the cache after providing the service. Then, we only need to worry about the possible conflict during a period of time on each third party. Moreover, different rules may need to be enforced with different frequencies. During certain amount of time, we can only focus on optimizing certain number of rules instead of all the rules. If we use the solution of static allocation, we may use extra number of third parties than we actually needed at any time point. In fact, the static data allocation assumes the worst case scenario, and we may want to allocate the data to the third parties on the fly which depends on the current allocation of data at the third parties.

As we assume the number of conflicts at a certain time point is small, we begin with only one third party. To enforce the first target rule, all the relevant data is sent to the third party. After that, for each new data sent to the third party, we check the previously built conflict graph to see if the rule regulating the new data will cause conflict on the third

party. If the new rule causes a new conflict, we move such data to a new third party. In the case that there is already a list of third parties, we check the newly added rule with each of these parties. We prefer to put the rule along with the rules from the same cooperative party since they never conflict with each other. We assign the new rule to the first found such third party, or we assign it to a new third party if needed. By allocating the data in this way, we may use more third parties than the number of parties currently involved in some bad cases. Whereas, under the assumption that there is not so many enforcement requests during a period of time, the number of third parties needed is usually smaller than the static allocation. The dynamic data allocation provides another option for solving the conflicts among the data.

Chapter 5: Conclusions and future work

In this chapter, we first give the conclusions of the works, and then discuss the future directions for the research.

5.1 Conclusions

In this thesis, we consider the scenarios that require different parties and enterprises to cooperate with each other to perform computations and meet business requirements. Each of these enterprises owns and manages its data independently using a private cloud, and these parties need to selectively share some information with one another. We consider the authorization model where authorization rules are used to constrain the access privileges based on the results of join operations over relational data.

We can interpret the authorization rules in two ways. With implicit authorization, we presented an efficient algorithm to decide whether a given query can be authorized using the join properties among the given rules. Under the explicit semantic, access conflicts may arise among the rules made according to business requirements. Therefore, we proposed a mechanism to make the set of cooperative authorization rules consistent. In addition, we also presented algorithms to maintain the rule consistency in the case of granting and revocation of access privileges. To prevent undesired computation results, negative rules are introduced. We proposed an algorithm to check whether the given authorization rules will violate the negative rules.

Since authorization rules are made based on business requirements, it is possible that some rules cannot be enforced among the cooperative parties which arises the problem of rule enforcement. To enforce a rule on join results, it is required that involved parties have sufficient authorizations to perform the join (or semi-join) operations. Therefore, given a set

of rules among some cooperative parties, it is desired to know whether each individual rule can be enforced among these parties. We proposed an algorithm that uses a constructive method to check the rules in a bottom-up manner based on the number of the relations involved, and the mechanism finds all the information that can be enforced among existing collaborating parties. There are some rules cannot be enforced, and we give a mechanism to modify the rules so as to make them totally enforceable. We proved the correctness and completeness of the algorithm.

In addition, a query execution plan is needed to answer an incoming query, and a regular query optimizer may miss possible plans under the constraints of these rules. Based on the rule enforceability, we presented a mechanism to generate a query execution plan which is consistent with the authorization rules for each incoming authorized query. Since finding the optimal query plan is NP-hard, our algorithm works in a greedy way to find a good solution. We compared the query plans generated by our algorithm with the optimal ones through case studies (where optimal plans can be identified). The results showed the effectiveness of our approach.

If there is no consistent plan to enforce a rule among existing parties, trusted third parties are needed. We think various models for the third party. A third party can work as a secure join service provider, where it does not store any information. Also, it can work as an aggregator where it can cache the input data and join results. We first examined the problem under the model of one third party works as a join service. We discussed how to minimize the communication cost and the computing cost for interaction with third parties. Since these problems are NP-hard, and we proposed greedy algorithms to generate solutions. With extensive simulation evaluations, we concluded that our algorithms were efficient and the solutions were close to the optimal ones. Sometimes, multiple third parties are needed for data isolation and performance purpose, and there is the problem of data allocation with multiple third parties. We described both static and dynamic data allocation mechanisms to meet the data isolation requirements with minimal number of third parties.

5.2 Future work

In the future, it is possible to formulate the query authorization problem as well as the negative rule checking problem with first-order logic so as to use traditional SAT based techniques [66, 67]; however, the feasibility and complexity of this approach remain to be investigated. For the enforcement of negative rules using Chinese wall policy, there exist a question of how to enforce it in a distributed manner. If the access history of the parties is not maintained at a central place, the enforcement of such negative rules becomes challenging. As we considered the rule consistency problem with respect to rule changes, it is worth to look into the more dynamic situation where not only the rules are changing from time to time, but also parties can join and leave the cooperative environment at different times.

We assume the collaborating parties first make the rules via negotiations, and then check whether a query is authorized and the safe ways to answer the query. It is possible to consider reversing the process. That is, we may want to figure out the complete set of queries that should be answered to meet business requirements, and after that we design authorization rules for cooperative parties so that only these wanted queries can be answered. However, due to the local computation, we may authorize extra information when granting privileges for this set of queries. Thereby, the problem becomes to figure out the best way of making rules so that minimal amount of extra information will be released together with the rules. To achieve that, we may also assume a limited number of third parties are given, and there is a problem of finding the optimal solution under such a scenario.

We studied the rule consistency problem with infrequent rule changes. In a military or workflow scenario, the permissions as well as the data may change on a per mission basis so that an authorization rule given to a party applies only for a short period of time. Since the relevant data also changes frequently in this case, it will become useless after some time. In such environments, the authorization rule can be granted dynamically based on the demands. For instance, for each step in a query, we can grant permissions to authorize the operation on the fly. Once such a step is executed, the authorizations are revoked.

This is similar to the workflow scenario. By granting privileges for a short time period, the extra information that is obtainable via local computation can be limited. The challenging problem becomes finding a way to schedule the queries as well as the time points to grant the authorizations so that minimal amount of extra information is released.

In our current model, access privileges are specified at the attribute level. Once a party can access an attribute, it can get all the tuples projected on that attribute. Since certain tuples can be more sensitive than the others, restrictions on the tuple level is also desired to prevent undesired data release. This is a simplified selection operation over the authorized data. However, it is expected that adding such a restriction should not complicate the current authorization model. In addition, it is also interesting to consider the write permissions. Our current models assume only the data owners may change their data and other parties just read the data from these owners. In some situations, it is desirable that a collaborating party can also modify the data owned by others. In addition to the synchronization problems, there is also a challenging problem of organizing the privileges and correctly granting and revoking write privileges to certain parties.

Our current model does not assume any malicious insiders and all the parties are expected to strictly follow the given authorization rules. In practice, a party may not behave honestly during the collaboration. For instance, a party may be authorized to obtain some information from a data owner, and then it may leak this information to some unauthorized parties. As another example, a party that receives data from the data owner and sends it to another party according to the generated query plan may change some of data it should transfer or choose not to send all the required data. Thus, it is required to have a mechanism that can verify the integrity of the received data. One possibility is to use the existing mechanisms such as hash values, merkle trees and signatures to ensure the data integrity [68, 69]. Considering the properties in the collaboration environment, it may be possible to check the data integrity through collaboration. In cooperative data access, there may exist more than one legitimate data transmission path beginning from the data owner to the authorized party. Therefore, parties can exchange the information they have.

By doing that, if the number of misbehaving parties is limited, it is very possible to detect them. It is possible to defined rules in such a way that each query be answered in at least k ways, and misbehaviors can be detected if only fewer than k/2 ways are behaving irregularly. Furthermore, existing mechanisms such as reputation systems [70] and trust management [71] can be considered to ensure the data integrity in the cooperative data access environment.

In previous discussion about the third party with storage ability, we do not consider the concrete cache mechanism used by the third party. The data cached by the third party should depend on the frequency of the use, size of the data, the update frequency of the relational data and so on. The optimal cache policy of the third party can be different from the file cache and process cache because the relational data is structured and we can cache the data on a per tuple or column basis. Therefore, it is not trivial to find the optimal cache policy. Furthermore, the data owned by the enterprises need to be synchronized with the data at the third party. The synchronization problem needs to be resolved.

To build the private cloud, different parties may rent the cloud infrastructure from the same service provider. It is also possible for an enterprise to build a hybrid cloud where the data owner manages the sensitive data locally, but the data for sharing is put in a public cloud. These emerging trends create new challenges and opportunities for secure cooperative data access. If cooperative parties use the same cloud provider, then the cloud provider could be used as a partially trusted third party to help enforce the security policies. In addition, it may be possible to perform privacy preserving join operations in such an environment. The expected mechanism can be a hybrid of using a trusted third party and the secure multiparty computation. Also, the cost model should also be revised under such situations.

At last, we want to apply and evaluate all the proposed mechanisms in real world scenarios. We already implemented the algorithms and tested them using generated simulation data. However, that is not adequate. We expect to look into the more concrete example in the real world. For instance, we may want to study the relational schemas used by the

e-commerce companies, shipping companies such as Amazon, ebay, UPS and on. We also need to analyze the collaboration relationships among these entities and see how to make the authorization rules in an optimal way so that only information required by business requirement is released. In addition, we want to evaluate how the query plan generation mechanism works under such environments, and to what extent the trusted third parties are needed to satisfy all security requirements.

Bibliography

Bibliography

[1] D. F. C. Brewer and M. J. Nash, "The chinese wall security policy," in *IEEE Symposium on Security and Privacy*, 1989, pp. 206–214.

[2] 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, Jun. 2008.

[3] 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)*, 2008, pp. 311–322.

[4] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Authorization enforcement in distributed query evaluation," *Journal of Computer Security*, vol. 19, no. 4, pp. 751–794, 2011.

[5] 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*, 2008, pp. 50–59.

[6] 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*, 1999, pp. 41–60.

[7] C. Li, "Computing complete answers to queries in the presence of limited access patterns," *The VLDB Journal*, vol. 12, no. 3, pp. 211–227, Oct. 2003.

[8] 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*, vol. 6, no. 4, pp. 602–625, Dec. 1981.

[9] D. Kossmann, "The state of the art in distributed query processing," *ACM Computer Survery*, vol. 32, no. 4, pp. 422–469, 2000.

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

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

[12] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[13] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *Proceedings of the Eleventh International Conference on Database Engineering*. IEEE, 1995, pp. 190–200.

[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)*, 2001, pp. 331–342.

[15] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis, "The GMAP: A versatile tool for physical data independence," *The VLDB Journal*, vol. 5, no. 2, pp. 101–118, 1996.

[16] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa, "Data exchange: semantics and query answering," *Theoretical Computer Science*, vol. 336, no. 1, pp. 89 – 124, 2005.

[17] R. Pottinger and A. Y. Halevy, "Minicon: A scalable algorithm for answering queries using views," *The VLDB Journal*, vol. 10, no. 2-3, pp. 182–198, 2001.

[18] Ullman, "Information integration using logical views," *TCS: Theoretical Computer Science*, vol. 239, 2000.

[19] H. Yang and P. A. Larson, "Query transformation for PSJ-queries," in *Proceedings of the International Conference on Very Large Data Bases (VLDB'87)*, Brighton, England, 1987, p. 245.

[20] F. Afrati, M. Chandrachud, R. Chirkova, and P. Mitra, "Approximate rewriting of queries using views," in *Proceedings of the 13th East European Conference on Advances in Databases and Information Systems(ADBIS'09)*, 2009, pp. 164–178.

[21] D. Kossmann, M. J. Franklin, G. Drasch, and W. Ag, "Cache investment: integrating query optimization and distributed data placement," *ACM Transactions on Database Systems*, vol. 25, no. 4, pp. 517–558, 2000.

[22] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *Proceedings of the ACM International Conference on Management of Data(SIGMOD'04)*, 2004.

[23] S. Abiteboul and O. M. Duschka, "Complexity of answering queries using materialized views," in *Proceedings of the 17th ACM Symposium on Principles of Database Systems(PODS'98)*, Seattle, WA, 1998, pp. 254–263.

[24] A. Y. Halevy, "Theory of answering queries using views," *SIGMOD Record*, vol. 29, no. 4, pp. 40–47, 2000.

[25] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, 2011.

[26] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[27] L. M. Vaquero, L. Rodero-Merino, and D. Moran, "Locking the sky: a survey on iaaS cloud security," *Computing*, vol. 91, no. 1, pp. 93–118, 2011.

[28] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref, "Secureblox: customizable secure distributed data processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 723–734.

[29] Y. Mao, C. Liu, J. E. van der Merwe, and M. Fernandez, "Cloud resource orchestration: A data-centric approach," in *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR'11)*, 2011, pp. 241–248.

[30] S. Das, D. Agrawal, and A. E. Abbadi, "Elastras: An elastic transactional data store in the cloud," in *Proceedings of USENIX HotCloud 2009*, 2010.

[31] R. Andersen, D. F. Gleich, and V. S. Mirrokni, "Overlapping clusters for distributed computation," in *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, (WSDM'12)*, 2012, pp. 273–282.

[32] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Proofs of ownership in remote storage systems," *IACR Cryptology ePrint Archive*, p. 207, 2011.

[33] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud computing," in *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09),*.

[34] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, (CCS'09)*, 2009.

[35] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati, "Efficient and private access to outsourced data," in *Proceedings of the 31st IEEE International Conference on Distributed Computing Systems (ICDCS'11)*, 2011, pp. 710–719.

[36] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Selective data outsourcing for enforcing privacy," *Journal of Computer Security*, vol. 19, no. 3, pp. 531–566, 2011.

[37] K. Zhang, X. yong Zhou, Y. Chen, X. Wang, and Y. Ruan, "Sedic: privacy-aware data intensive computing on hybrid clouds," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, (CCS'11)*, 2011, pp. 515–526.

[38] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *IEEE Symposium on Security and Privacy*, 2007, pp. 321–334.

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

[40] A. Gouglidis and I. Mavridis, "domRBAC: An access control model for modern collaborative systems," *Journal of Computers and Security*, vol. 31, no. 4, pp. 540–556, 2012.

[41] 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)*, 2003, pp. 93–99.

[42] 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*, oct. 2011, pp. 231 –240.

[43] 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)*, july 2007, pp. 1064 –1071.

[44] E. Y. Li, T. C. Du, and J. W. Wong, "Access control in collaborative commerce," *Decision Support Systems*, vol. 43, no. 2, pp. 675–685, 2007.

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

[46] 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.

[47] B. Carbunar and R. Sion, "Toward private joins on outsourced data," *IEEE Transactions of Knowledge and Data Engineering(TKDE)*, vol. 24, no. 9, pp. 1699–1710, 2012.

[48] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu, "Two can keep A secret: A distributed architecture for secure database services," in *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR'05)*, 2005, pp. 186–199.

[49] V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Keep a few: Outsourcing data while maintaining confidentiality," in *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09),*.

[50] R. Sion, "Query execution assurance for outsourced databases," in *Proceedings of the International Conference on Very Large Data Bases (VLDB'05)*, 2005, pp. 601–612.

[51] V. Ciriani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Combining fragmentation and encryption to protect privacy in data storage," *ACM Transactions on Information and System Security*, vol. 13, no. 3, Jul. 2010.

[52] L. Kissner and D. Song, "Privacy-preserving set operations," *Advances in Cryptology (CRYPTO 2005)*, vol. 3621, pp. 241–257, 2005.

[53] D. D. K. Mishra, P. Trivedi, and S. Shukla, "A glance at secure multiparty computation for privacy preserving data mining," in *International Journal on Computer Science and Engineering*, 2009.

[54] S. S. M. Chow, J.-H. Lee, and L. Subramanian, "Two-party computation model for privacy-preserving queries over distributed databases," in *Proceedings of the 16th Annual Network and Distributed System Security Conference (NDSS'09)*, 2009.

[55] A. V. Aho, C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM Transactions on Database Systems*, vol. 4, no. 3, pp. 297–314, Sep. 1979.

[56] Alon, Moshkovitz, and Safra, "Algorithmic construction of sets for k-restrictions," vol. 2, 2006.

[57] Munagala, Babu, Motwani, and Widom, "The pipelined set cover problem," in *ICDT: 10th International Conference on Database Theory*, 2005.

[58] P. Valduriez, "Semi-join algorithms for multiprocessor systems," in *Proc. ACM SIGMOD Conf.*, Orlando, FL, Jun. 1982, p. 225.

[59] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing n-relational joins," *ACM Trans. Database System*, vol. 9, no. 3, pp. 482–502, Sep. 1984.

[60] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[61] E. Rahm, "Primary copy synchronization for db-sharing," *Information Systems*, vol. 11, no. 4, pp. 275–286, 1986.

[62] S. Jiang and X. Zhang, "Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, 2002, pp. 31–42.

[63] T. R. Puzak, "Analysis of cache replacement-algorithms," 1985.

[64] M. M. Hallórsson, "A still better performance guarantee for approximate graph coloring," *Inf. Process. Lett.*, 1993.

[65] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[66] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and applications of satisfiability testing*, 2004, pp. 502–518.

[67] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design*, 2000, pp. 127–144.

[68] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume, "Merkle signatures with virtually unlimited signature capacity," in *Applied Cryptography and Network Security*, 2007, pp. 31–45.

[69] C. Wang, Q. Wang, K. Ren, and W. Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *INFOCOM, 2010 Proceedings IEEE*, 2010, pp. 1–9.

[70] K. Hoffman, D. Zage, and C. Nita-Rotaru, "A survey of attack and defense techniques for reputation systems," *ACM Computing Surveys (CSUR)*, vol. 42, no. 1, p. 1, 2009.

[71] R. K. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, "Trustcloud: A framework for accountability and trust in cloud computing," in *Services (SERVICES), 2011 IEEE World Congress on*, 2011, pp. 584–588.

Curriculum Vitae

Meixing Le is a PhD student in the Center for Secure Information Systems at George Mason University. His major is Information Technology. Before joining GMU, he received a Bachelor degree in Computer Science from Fudan University, China, in 2004, and a Master degree in Computer Science from Fudan University, China, in 2007. His research focuses on information security, privacy, and database systems.