© IOS Press, (2011). This is the author's version of the work. It is posted here by permission of IOS Press for your personal use.
Published in Journal of Computer Security (JCS), Volume 19, Issue 4, Pag 751-794, 2011.

Authorization Enforcement in Distributed Query Evaluation*

Sabrina De Capitani di Vimercati,[1] Sara Foresti,[1] Sushil Jajodia,[2] Stefano Paraboschi,[3] Pierangela Samarati[1]

[1]DTI - Università degli Studi di Milano - 26013 Crema, Italy
firstname.lastname@unimi.it
[2]CSIS - George Mason University - Fairfax, VA 22030-4444
jajodia@gmu.edu
[3]DIIMM - Università degli Studi di Bergamo - 24044 Dalmine, Italy
parabosc@unibg.it

Corresponding author: Pierangela Samarati
DTI - Università degli Studi di Milano
Via Bramante 65 - 26013 Crema, Italy
pierangela.samarati@unimi.it
phone: +39-0373-898061, *fax:* +39-0373-898010

Abstract

We present a simple, yet powerful, approach for the specification and enforcement of authorizations regulating data release among data holders collaborating in a distributed computation, to ensure that query processing discloses only data whose release has been explicitly authorized. Data disclosure is captured by means of profiles, associated with each data computation, that describe the information carried by a base or a derived (i.e., computed by a query) relation. We present an algorithm that, given a query plan, determines whether it can be safely executed and produces a safe execution strategy for it. For each operation in a safe query plan, the algorithm determines the server(s) responsible for the execution, based on the entailed information flows, considering different strategies for the execution of joins. Finally, we discuss the architecture of a distributed database system based on the proposed model, illustrating possible design choices and their impact.

keywords: Distributed query evaluation, authorized views, safe query planning

*A preliminary version of this paper appeared under the title "Controlled Information Sharing in Collaborative Distributed Query Processing," in *Proc. of the 28th International Conference on Distributed Computing Systems*(ICDCS 2008), Beijing, China, June 17-20, 2008 [11].

1 Introduction

More and more emerging scenarios require different parties, each withholding large amounts of independently managed information, to cooperate with other parties in a large distributed system to the aim of sharing information and perform distributed computations. Such scenarios range from traditional distributed database systems, where a centrally planned database design is then distributed to different locations; to federated systems, where independently developed databases are merged together; to dynamic coalitions and virtual communities, where independent parties may need to selectively share part of their knowledge towards the completion of common goals. Regardless of the specific scenario, a common point of such a merging and sharing process is that it is selective: if on one hand there is a need to share some data and cooperate, there is on the other hand an equally strong need to protect those data that, for various reasons, should not be disclosed.

The correct definition and management of protection requirements is therefore a crucial point for an effective collaboration and integration of large-scale distributed systems. The problem calls for a solution that must be expressive to capture the different data protection needs of the cooperating parties as well as simple and consistent with current mechanisms for the management of distributed computations, to be seamlessly integrated in current systems. To this aim and for the sake of concreteness, in this paper we address the problem with specific consideration to distributed database systems. While noting that our approach can be extended to other data models, we also note that the emphasis on relational databases must not be considered a limitation. First, relational database technology currently dominates the management of data in most scenarios where collections of sensitive information have to be integrated over a network; even if a system offers access to the data using Web technology, the data offered by the system are extracted from a relational database and a description of the access policy in terms of the underlying relational structure offers a high degree of flexibility. Second, for integration solutions based on Web technology, and in particular systems relying on the use of Web services, it is always possible to model the structure of the exported data in terms of a relational representation. In this situation a description of the access policy according to our model, in contrast to a policy description on service invocations, typically provides a more robust and flexible identification of the security requirements of the application.

We consider a scenario where relations are distributed at different servers, query execution may require cooperation, and data are exchanged among the different servers involved in the query. Each server is responsible for the definition of the access policy on its resources. We propose an authorization model to regulate the view that each server can have on the data and ensure that query computation exposes to each server only data that the server can view. In our approach, authorizations regulate not only the data on which parties have explicit visibility, but also the visibility of possible associations that such data convey. Our simple authorization

form essentially corresponds to generic view patterns, thus nicely meeting both expressiveness and simplicity requirements. A novel aspect of the model is the definition of distinct authorization profiles for different parties in the system and the explicit support for cooperative query evaluation. This is an important feature in distributed settings, where the minimization of data exchanges and the execution of a query step in locations where it can be less costly is a crucial factor in the identification of an execution strategy characterized by good performance. In [11] we presented an early version of our proposal that here has been extended to possibly consider a third party (belonging to the system) as responsible for the execution of one or more operations composing a query plan, when none of the involved parties has the authorizations necessary for their computation. We then revise both the definition of safe executor assignment and of feasible query plan. The algorithm originally proposed in [11], which determines whether and how a given query plan can be safely executed, has been changed to compute a safe executor assignment eventually involving a third party. In addition, we formally analyze the correctness and computational complexity of the novel algorithm and describe the architecture of a distributed database system that adopts the proposed model. The description of the architecture focuses on the possible design choices and their impact.

The remainder of the paper is organized as follows. Section 2 introduces preliminary concepts of the distributed data model and distributed query evaluation. Section 3 illustrates our security model. Section 4 discusses query planning and how protection requirements stated by authorizations must be considered in the query execution to ensure that data are properly protected within the distributed computation. Section 5 proposes an algorithm for determining whether a query plan can be executed in the respect of the authorizations and, if it exists, producing a safe assignment of tasks to the distributed cooperating parties for the execution of the query plan. Section 6 describes the architecture of a distributed database system implementing our security model. Section 7 discusses related work. Finally, Section 8 draws our conclusions. The proofs of the theorems are reported in Appendix A.

2 Preliminary concepts

We consider a distributed system composed of different servers, storing different relations. Each relation is represented as $R(A_1, \ldots, A_n)$, where R is the name of the relation and $A_1, \ldots, A_n$ are its attributes. At the instance level, a relation is a set of tuples, where each tuple associates with each attribute in the relation a value in the attribute's domain. The *primary key* of a relation is the attribute, or set of attributes, that uniquely identifies each tuple in the relation. For the sake of simplicity, we assume all relations to have distinct names and all attributes in the different relations to have distinct names. While simplifying the notation, this assumption does not limit in any way our approach since relations/attributes with the same name can be made distinct by

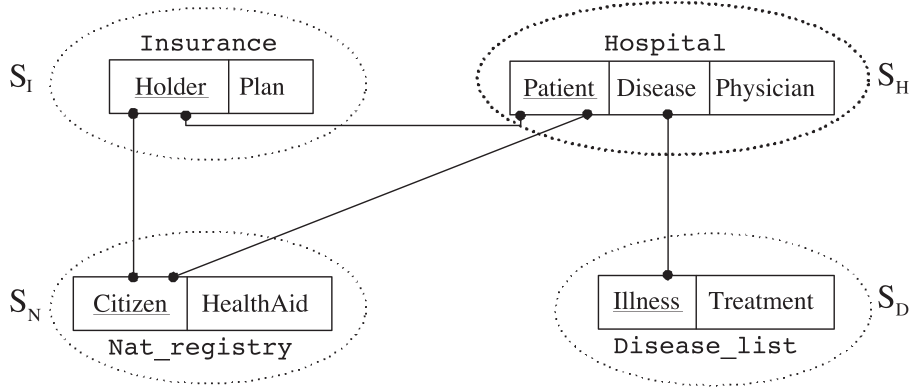


Figure 1: Schema of a distributed system

using the usual dot notation (*server.relation.attribute*).

A fundamental operation in the relational model is the *join* ($\bowtie$) between relations. A join combines tuples belonging to two different relations based on specified conditions. We consider equi-joins, that is, joins whose conditions are conjunctions of expressions of the form A_l=A_r, where A_l is an attribute of the relation appearing as left operand and A_r an attribute of the relation appearing as right operand. In the following, we denote a conjunction of equi-join conditions simply as a pair $\langle J_l, J_r\rangle$, where J_l (J_r, resp.) is the list of attributes of the left (right, resp.) operand. Different join operations can be used to combine tuples belonging to more than two relations. The following definition introduces a *join path* as a sequence of equi-join conditions.

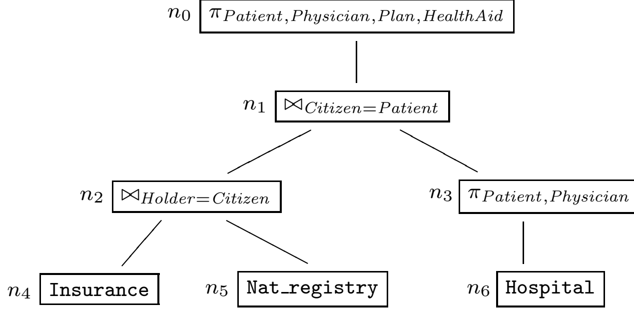
Definition 2.1 (Join path)**.** *A join path over a sequence of relations $R_1, \ldots, R_n$ is a sequence of $n-1$ equi-joins $J_1, \ldots, J_{n-1}$, with $J_i = \langle J_{l_i}, J_{r_i}\rangle$, $i = 1, \ldots, n-1$, where J_{r_i} is a list of attributes of relation R_{i+1} and J_{l_i} a list of attributes of a relation R_k, with $k \leq i$.*

Example 2.1. *Figure 1 represents a distributed system managing medical data. The system is composed of four relations each stored at a different server:* `Insurance` *stored at server S_I;* `Hospital` *stored at server S_H;* `Nat_registry` *stored at server S_N, and* `Disease_list` *stored at server S_D. Underlined attributes denote primary keys, while lines represent possible joins. An example of join path is* {⟨Holder,Patient⟩*,* ⟨Disease,Illness⟩}*, combining tuples of relations* `Insurance`*,* `Hospital`*, and* `Disease_list` *to retrieve the insurance plan of patients using a given treatment.*

We consider simple select-from-where queries of the form: "SELECT A FROM *Joined relations* WHERE C", corresponding to the algebra expression $\pi_A(\sigma_C(R_1 \bowtie_{J_1} \ldots \bowtie_{J_{n-1}} R_n))$, where A is a set of attributes, C is the selection conditions, and $R_1 \bowtie_{J_1} \ldots \bowtie_{J_{n-1}} R_n$ are the joins in the FROM clause. Each query execution can be represented as a binary tree, called *query tree plan*, and denoted $T(N,E)$, where leaves correspond to the physical relations accessed by the query (appearing in the FROM clause), each non-leaf node is a relational

q_1:

SELECT *Patient*, *Physician*, *Plan*, *HealthAid*
FROM `Insurance` JOIN `Nat_registry` ON *Holder*=*Citizen*
JOIN `Hospital` ON *Citizen*=*Patient*



q_2:

SELECT *Patient*, *Physician*, *Plan*
FROM `Insurance` JOIN `Hospital` ON *Holder*=*Patient*
WHERE *Disease* = "hypertension"

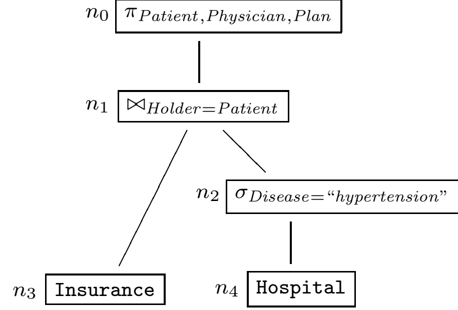


Figure 2: Two examples of query tree plan on the schema of Figure 1

operator receiving in input the result produced by its children and producing a relation as output, and the root corresponds to the last operation and returns the result of the query evaluation. For simplicity and without loss of generality, we assume the query plan to satisfy the usual minimization criteria. In particular, projections are "pushed down" the tree to eliminate unnecessary attributes as soon as possible in the query execution. While usually adopted for efficiency, this assumption is also important for security purposes, as it permits to discard (i.e., not make visible) attributes not needed for the computation.

Example 2.2. *Figure 2 illustrates two examples of queries, namely q_1 and q_2, corresponding to the relational algebra expressions* $\pi_{Patient,Physician,Plan,HealthAid}$ (`Insurance` $\bowtie_{Holder=Citizen}$ `Nat_registry` $\bowtie_{Citizen=Patient}$ `Hospital`)*, and* $\pi_{Patient,Physician,Plan}$ ($\sigma_{Disease=\text{"hypertension"}}$ (`Insurance` $\bowtie_{Holder=Patient}$ `Hospital`))*, respectively. On the left-hand side of Figure 2 there is a query tree plan for query q_1, where the projection on attributes* Patient *and* Physician *of relation* `Hospital` *has been pushed down. On the right-hand side of Figure 2 there is a query tree plan for query q_2, where the selection on attribute* Disease *of relation* `Hospital` *has been pushed down.*

In the following, given an operation involving a relation stored at a server, we will use the term *operand* to refer indistinguishably to either the relation or the server storing it, when the semantics is clear from the context.

3 Security model

We first present our simple, while expressive, authorizations, regulating how data can be released to each server. We then introduce the concept of relation profile that characterizes the information content of a relation.

3.1 Authorizations

Consistently with standard security practice, we assume a "closed policy", where data can be made visible only to parties explicitly authorized for that.[1] Authorizations have the following form.

Definition 3.1 (Authorization)**.** *An* authorization *is a rule of the form* [Attributes, Join Path]→Server *where:*

1. Attributes *is a set of attributes belonging to one or more relations;*

2. Join Path *is a join path including* (*at least*) *all the relations with attributes in* Attributes*, that is, whose release is authorized* (*the join path can be empty when all the attributes in* Attributes *belong to the same relation*)*;*

3. Server *is a server in the distributed system.*

The semantics of an authorization is that *Server* can be released (i.e., is authorized to view) the set *Attributes* of attributes of tuples resulting from the join operation described by *Join Path*. This form of authorizations, with the specification of the join path as a separate element, is quite expressive, yet simple and easily usable by DBAs.

Note that the join path may also include attributes appearing in relations that do not have any attribute in *Attributes*. This may be due to two different reasons.

1. *Connectivity constraints*. The additional relations are needed to build a correct association among the attributes of other relations (i.e., the relations that are in the join path). For instance, in authorization 3 in Figure 3 attribute `Patient` in `Hospital` appears in the join path to establish the association between insurance holders and their treatments, but none of the attributes of `Hospital` are released. Note how the authorization allows the insurance company (server S_I) to view the treatment of its subscribers without need of knowing their illness.

2. *Instance-based restrictions*. The additional relations are needed to restrict the values of attributes to be released to only those values appearing in tuples that can be associated with such relations. For instance, authorization 15 in Figure 3 allows server S_N to view the illnesses of all the citizens appearing

[1] While we assume a closed policy, we note that our approach can be adapted to an "open policy" scenario, where data are visible by default and negative rules specify restrictions on the visibility that parties may have on the data [22].

	Attributes	**Join Path**	**Server**
1	{Holder, Plan}	-	S_I
2	{Holder, Plan, Patient, Physician}	{⟨Holder,Patient⟩}	S_I
3	{Holder, Plan, Treatment}	{⟨Holder,Patient⟩, ⟨Disease,Illness⟩}	S_I
4	{Patient, Disease, Physician}	-	S_H
5	{Patient, Disease, Physician, Holder, Plan}	{⟨Patient,Holder⟩}	S_H
6	{Patient, Disease, Physician, Citizen, HealthAid}	{⟨Patient,Citizen⟩}	S_H
7	{Patient, Disease, Physician, Holder, Plan, Citizen, HealthAid}	{⟨Patient,Citizen⟩, ⟨Citizen,Holder⟩}	S_H
8	{Citizen, HealthAid}	-	S_N
9	{Holder, Plan}	-	S_N
10	{Patient, Disease}	-	S_N
11	{Citizen, HealthAid, Patient, Disease}	{⟨Citizen,Patient⟩}	S_N
12	{Citizen, HealthAid, Holder, Plan}	{⟨Citizen,Holder⟩}	S_N
13	{Patient, Disease, Holder, Plan}	{⟨Patient,Holder⟩}	S_N
14	{Citizen, HealthAid, Patient, Disease, Holder, Plan}	{⟨Citizen,Patient⟩, ⟨Citizen,Holder⟩}	S_N
15	{Citizen, Illness}	{⟨Citizen,Patient⟩, ⟨Disease,Illness⟩}	S_N
16	{Illness, Treatment}	-	S_D

Figure 3: Examples of authorizations for the distributed system in Figure 1

in the National Registry (i.e., tuples in `Nat_registry` satisfying *Citizen=Patient* and *Disease=Illness* conditions) but not of those patients who are not registered. Note how instance-based restrictions can also be used to support situations where some information can be released only if explicit input is requested (the input is viewed in this case as a relation to be joined). For instance, with reference to the example just mentioned, providing the citizen identifier, server S_N can retrieve the corresponding illness.

It is important to note that the presence of a join path in an authorization implies the release of fewer tuples (only those for which the conditions in the join path are satisfied), but it does not imply the release of less information. Indeed, releasing a tuple implicitly gives information on the fact that the tuple satisfies the join path, that is, that its values have an association with another relation (which might not be released). For instance, authorization 2 in Figure 3 gives S_I not only the values of attribute *Physician* for its subscribers, but also the additional information about the fact that the subscriber has been hospitalized. We will come back to this observation when discussing access control evaluation.

Note also that, while expressive, our authorizations remain at the schema level, that is, they do not allow restricting access only to certain tuples, depending on the satisfaction of some conditions (apart from joins). The reason for this is that including selection conditions in the authorizations would increase the complexity and reduce the applicability of the model.

Operation	Profile R^{π}	$R^{\bowtie}$	R^{σ}
$R := \pi_X(R_l)$	X	$R_l^{\bowtie}$	R_l^{σ}
$R := \sigma_X(R_l)$	R_l^{π}	$R_l^{\bowtie}$	$R_l^{\sigma} \cup X$
$R := R_l \bowtie_j R_r$	$R_l^{\pi} \cup R_r^{\pi}$	$R_l^{\bowtie} \cup R_r^{\bowtie} \cup j$	$R_l^{\sigma} \cup R_r^{\sigma}$

Figure 4: Profiles resulting from relational operations

3.2 Profiles and authorized views

Authorizations restrict the data (views) that can be released to each server. To determine whether a release should be authorized or not, we first need to capture the information content of a relation, either *base* or *derived* (i.e., computed by a query). To this purpose, we introduce the concept of relation profile.

Definition 3.2 (Relation profile). *Given a relation R, the* relation profile *of R is a triple $[R^{\pi}, R^{\bowtie}, R^{\sigma}]$, where: R^{π} is the set of attributes in R (i.e., R's schema); $R^{\bowtie}$ is the, possibly empty, join path used in the definition/construction of R; R^{σ} is a, possibly empty, set of attributes involved in selection conditions in the definition/construction of R.*

According to the definition above, the relation profile of a base relation $R(A_1, \ldots, A_n)$ is $[\{A_1, \ldots, A_n\}, \emptyset, \emptyset]$. Also, according to the semantics of the relational operators, the profile resulting from a relational operation, summarized in Figure 4,[2] is as follows.

- *Projection* (π). It returns a *subset of the attributes* of the operand. Hence, $R^{\bowtie}$ and R^{σ} of the resulting relation R are the same as the ones of the operand, while R^{π} contains only those attributes being projected.
- *Selection* (σ). It returns a *subset of the tuples* of the operand. Hence, $R^{\bowtie}$ and R^{π} of the resulting relation R are the same as the ones of the operand, while R^{σ} needs to include also the attributes appearing in the selection condition.
- *Join* ($\bowtie$). It returns a relation that contains the *association of the tuples of the operands*, thus capturing the information in both operands as well as the information on their association (conditions in the join). Hence, R^{σ} and R^{π} of the resulting relation R are the union of those of the operands, while $R^{\bowtie}$ is the union of the join paths of the operands and the one of the operation.

A generic query of the form "SELECT A FROM *Joined Relations* WHERE C" then produces a derived relation whose profile is $[A, J_q, Att_C]$, where A is the set of projected attributes, J_q is the join path connecting the relations in the FROM clause of the query, and Att_C are the attributes appearing in the WHERE clause. In the

[2]For the sake of simplicity, with a slight abuse of notation, we write $\sigma_X(R)$ as a short hand for any expression $\sigma_{condition}(R)$, where X is the set of attributes of R involved in *condition*.

following, when clear from the context, we use the term relation to refer to either a base relation or a derived relation.

According to the semantics of authorizations and of profiles, the visibility on the different relations by a server is regulated as follows.

Definition 3.3 (Authorized view). *Given a set $\mathcal{A}$ of authorizations, a relation R with profile $[R^\pi, R^{\bowtie}, R^\sigma]$, and a server S, we say that S is* authorized to view R *iff* $\exists$ $[A, J]\rightarrow S \in \mathcal{A}$ *such that both the following conditions hold:* 1) $R^\pi \cup R^\sigma \subseteq A$, *and* 2) $R^{\bowtie} = J$.

Definition 3.3 states that a relation can be released to a server only if there is an authorization permitting the release of (at least) all the attributes, either explicitly contained in the relation or appearing in the selection condition in its definition, and which has exactly the same join path as the relation. The reason for the subset in the first condition is that an authorization to view a superset of attributes implies the authorization to view a subset of them. One could think that a similar implication could also hold for join paths, and therefore that an authorization containing a join path could also authorize all relations that simply include the path (as the addition of further conditions simply restricts the set of tuples in the result). However, this implication cannot hold. In fact, while decreasing the set of tuples belonging to the result, any additional join condition adds information on the fact that the tuples join with (i.e., have values appearing in) other tuples of relations whose content should not be released. For instance, consider the relation obtained as "SELECT *Illness*, *Treatment* FROM `Disease_list` JOIN `Hospital` ON *Illness*=*Disease*", characterized by profile [{*Illness*,*Treatment*},{⟨*Illness*,*Disease*⟩},∅]. S_D cannot access such a relation because its only authorization for the attributes (authorization 16) does not have the join path that appears in the relation profile. While including only some of the tuples of the `Disease_list` relation, the query result bears also the information of which illnesses have a correspondence in the `Hospital` relation, which S_D is not authorized to view.

While we check each data release with respect to individual authorizations, we note that a server could be authorized to view some data in the case where, even if not explicitly authorized for the specific data view, it holds the authorizations for all the underlying relations and therefore would be able to independently compute the view. For instance, with reference to the example just mentioned, suppose S_D has, besides authorization 16 allowing it to view relation `Disease_list`, also the authorization to view relation `Hospital`. The two authorizations clearly imply the (derived) authorization for the query above, which represents a view on them. Derived authorizations can be obtained by means of a "chase" procedure that computes all the authorizations implied directly or indirectly by those explicitly specified. Such closing process goes beyond the functionalities of the SQL standard, but is based on known results [2, 10]. Its treatment is outside the scope of the paper; we therefore do not discuss it further, and assume that authorizations are closed with respect to these derivations.

Oper.	[m,s]	Operation/Flow	Views(S_l)	Views(S_r)	View profiles
$\pi_X(R_l)$	[S_l,NULL]	S_l: $\pi_X(R_l)$			
$\sigma_X(R_l)$	[S_l,NULL]	S_l: $\sigma_X(R_l)$			
$R_l \bowtie_{J_{lr}} R_r$	[S_l,NULL]	S_r: $R_r \rightarrow S_l$ S_l: $R_l \bowtie_{J_{lr}} R_r$	R_r		$[R_r^\pi, R_r^\bowtie, R_r^\sigma]$
	[S_r,NULL]	S_l: $R_l \rightarrow S_r$ S_r: $R_l \bowtie_{J_{lr}} R_r$		R_l	$[R_l^\pi, R_l^\bowtie, R_l^\sigma]$
	[S_l,S_r]	S_l: $R_{J_l} := \pi_{J_l}(R_l)$ S_l: $R_{J_l} \rightarrow S_r$ S_r: $R_{J_lr} := R_{J_l} \rtimes_{J_{lr}} R_r$ S_r: $R_{J_lr} \rightarrow S_l$ S_l: $R_{J_lr} \bowtie R_l$	$\pi_{J_l}(R_l) \rtimes_{J_{lr}} R_r$	$\pi_{J_l}(R_l)$	$[J_l, R_l^\bowtie, R_l^\sigma]$ $[J_l \cup R_r^\pi, R_l^\bowtie \cup R_r^\bowtie \cup J_{lr}, R_l^\sigma \cup R_r^\sigma]$
	[S_r,S_l]	S_r: $R_{J_r} := \pi_{J_r}(R_r)$ S_r: $R_{J_r} \rightarrow S_l$ S_l: $R_{lJ_r} := R_l \ltimes_{J_{lr}} R_{J_r}$ S_l: $R_{lJ_r} \rightarrow S_r$ S_r: $R_{lJ_r} \bowtie R_r$	$\pi_{J_r}(R_r)$	$R_l \ltimes_{J_{lr}} (\pi_{J_r}(R_r))$	$[J_r, R_r^\bowtie, R_r^\sigma]$ $[R_l^\pi \cup J_r, R_l^\bowtie \cup R_r^\bowtie \cup J_{lr}, R_l^\sigma \cup R_r^\sigma]$

Figure 5: Execution of operations and required views with corresponding profiles

4 Safe query planning

Goal of this work is to determine whether and how the query can be computed without violating the defined authorizations. A necessary condition for the query execution is clearly that the requester be authorized to view the profile associated with the relation corresponding to the query result. As a matter of fact, there is no need of determining a plan for a query execution if its result cannot be made visible to the requester. In the following, we therefore consider queries whose profile can be viewed by the requester (queries for which such condition does not hold are discarded upfront). The definition of a plan for the query execution requires determining the data releases that each execution step entails, so that only executions implying authorized releases are performed. Since we can assume each server to be authorized to view the relation it holds, each unary operation (projection and selection) can be executed by the server itself, while a join operation can be executed if all the data communications correspond to authorized releases. Figure 5 summarizes the operations and the data exchanges needed to perform a relational operation, reporting, for every data communication, the profile of the relation being communicated (and hence the information exposure implied by it); data access by a server on its own relation is implicit. For each operation/communication we also show, before the ":", the server executing it. We note that a join operation $R_l \bowtie_{J_{lr}} R_r$, where R_l and R_r represent the left and right input relations, respectively, can be executed either as a *regular join* or a *semi-join*. We call *master* the server in charge of the join computation and *slave* the server that cooperates with the master during the computation. We then distinguish four different cases depending on whether the join is executed as a regular join or as a semi-join and on which operand serves as master (slave, respectively). The master/slave assignment is specified

as a pair, where the first element is the operand that serves as master and the second element is the operand that serves as slave. We discuss the cases where the left operand serves as master (denoted $[S_l,\text{NULL}]$ for the regular join and $[S_l,S_r]$ for the semi-join), with the note that the cases where the right operand serves as master ($[S_r,\text{NULL}]$ and $[S_r,S_l]$) are symmetric.

- $[S_l,\text{NULL}]$: in the *regular join* processed by S_l, server S_r sends its relation to S_l, and S_l computes the join. For execution, S_l needs to be authorized to view R_r, which has profile $[R_r^\pi,R_r^\bowtie,R_r^\sigma]$.
- $[S_l,S_r]$: the *semi-join* requires a sequence of five steps. *1)* S_l computes the projection R_{J_l} of the attributes in its relation R_l participating in the join. *2)* S_l sends R_{J_l} to S_r; this operation entails a data release characterized by the profile of R_{J_l}, which (according to Definition 3.2) is $[J_l,R_l^\bowtie,R_l^\sigma]$. *3)* S_r locally computes R_{J_lr} as the join between R_{J_l} and its relation R_r. *4)* S_r sends R_{J_lr} to S_l; this operation entails a data release characterized by the profile of R_{J_lr}, namely $[J_l\cup R_r^\pi,R_l^\bowtie\cup R_r^\bowtie\cup J_{lr},R_l^\sigma\cup R_r^\sigma]$. *5)* S_l computes the natural join between R_{J_lr} and its own relation R_l.

Semi-joins allow reducing the flow of information: the slave server needs only to send those tuples that participate in the join, instead of its complete relation.

Example 4.1. *Consider query q_1 in Example 2.2. If the join between* `Insurance` *and* `Nat_registry` (*node n_2 in Figure 2*) *is executed as a regular join, S_N sends the whole* `Nat_registry` *relation to S_I* (*or, vice versa, S_I sends the whole* `Insurance` *relation to S_N*)*, who computes the result of the join. If the join is executed as a semi-join, where S_I acts as master, S_I sends to S_N the projection of* `Insurance` *on* Holder*. S_N then sends back to S_I* `Nat_registry` *joined with the list of values of* Holder *received from S_I. Finally, S_I computes the result of the join between the relation received from S_N and its relation* `Insurance`.

We assign to each node of a query tree plan a server or a pair of servers, called *executor*, responsible for the execution of the algebraic operation represented by the node. To formally capture this intuitive idea, we introduce the definition of the *executor assignment* function λ_T as follows.

Definition 4.1 (Executor assignment)**.** *Given a query tree plan $T(N,E)$, an* executor assignment function $\lambda_T : N \rightarrow \mathcal{S} \times (\mathcal{S}\cup\{\text{NULL}\})$ *assigns to each node a pair of servers such that:*

1. *each leaf node* (*corresponding to a base relation R*) *is assigned the pair $[S,\text{NULL}]$, with S the server where R is stored;*
2. *each non-leaf node n, corresponding to unary operation* op *on operand R_l* (*left child*) *at server S_l, is assigned the pair $[S_l,\text{NULL}]$;*

3. *each non-leaf node n, corresponding to a join on operands R_l (left child) at server S_l and R_r (right child) at server S_r, is assigned a pair* [master,slave] *such that* master $\in \{S_l, S_r\}$, slave $\in \{S_l, S_r, \text{NULL}\}$, *and* master$\neq$slave.

Given a query plan, we need to determine an assignment of the computation steps to different servers, in such a way that the execution given by the assignment entails only views allowed by the authorizations. This concept is captured by the following definition of safe assignment.

Definition 4.2 (Safe assignment). *Given a query tree plan $T(N,E)$, a node $n \in N$, and an* executor assignment function λ_T, *$\lambda_T(n)$ is said to be* safe *if one of the following conditions hold:* 1) *n is a leaf node;* 2) *n corresponds to a unary operation;* 3) *n corresponds to a join and all the views derived by the assignment are authorized. The executor assignment λ_T is said to be* safe *iff $\forall n \in N$, $\lambda_T(n)$ is safe.*

A query plan is feasible iff there exists a safe assignment for it, as captured by the following definition.

Definition 4.3 (Feasible query plan). *A query plan $T(N,E)$ is said to be* feasible *iff there exists an executor assignment function λ_T on T such that λ_T is safe.*

4.1 Third party involvement

As already discussed, the execution of joins necessarily requires some communication of information among the operands, which we check against authorizations and allow only if authorized. It may happen that, for a given join, none of the four possible modes of execution in Figure 5 corresponds to a safe assignment, but the join could be safely executed with the consideration of a third party acting either as proxy for one of the two operands or as coordinator for the join evaluation. To find a safe executor assignment for the join, it is therefore necessary to verify whether one or more third parties can be involved in the computation of the join. The following theorem proves that the maximum number of third parties that may need to be involved is equal to the number of conditions in the join path.

Theorem 4.1. *Given two relations $R_l(A_{l_1}, A_{l_2}, \ldots, A_{l_n}, A_{l_{n+1}})$ and $R_r(A_{r_1}, A_{r_2}, \ldots, A_{r_n}, A_{r_{n+1}})$, the maximum number of parties necessary to evaluate an equi-join $R_l \bowtie_{J_{lr}} R_r$, with $J_{lr} = \bigwedge_{i=1}^{n} \langle A_{l_i}, A_{r_i} \rangle$, is n.*

Proof: See Appendix.

While the consideration of a third party can bring great advantage in many cases, it is instead clearly impractical, as well as undesirable, to consider an arbitrary number of parties (and check all the possible ways for executing a join that could therefore arise). In this paper, we therefore consider the case where only one third party can participate in the join operation. This assumption keeps the computational cost of the algorithm limited, since the number of ways in which a join operation can be safely evaluated remains low.

[m,s]	Operation/Flow	Views(S_l)	Views(S_r)	Views(S_t)	View profiles
$[S_t$,NULL$]$	S_l: $R_l \rightarrow S_t$ S_r: $R_r \rightarrow S_t$ S_t: $R_l \bowtie_{J_{lr}} R_r$			R_l R_r	$[R_l^{\pi}, R_l^{\bowtie}, R_l^{\sigma}]$ $[R_r^{\pi}, R_r^{\bowtie}, R_r^{\sigma}]$
$[S_t,S_l]$	S_r: $R_r \rightarrow S_t$ S_t: $R_{J_r} := \pi_{J_r}(R_r)$ S_t: $R_{J_r} \rightarrow S_l$ S_l: $R_{lJ_r} := R_l \bowtie_{J_{lr}} R_{J_r}$ S_l: $R_{lJ_r} \rightarrow S_t$ S_t: $R_{lJ_r} \rtimes R_r$	$\pi_{J_r}(R_r)$		R_r $R_l \bowtie_{J_{lr}} (\pi_{J_r}(R_r))$	$[R_r^{\pi}, R_r^{\bowtie}, R_r^{\sigma}]$ $[J_r, R_r^{\bowtie}, R_r^{\sigma}]$ $[R_l^{\pi} \cup J_r, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$
$[S_t,S_r]$	S_l: $R_l \rightarrow S_t$ S_t: $R_{J_l} := \pi_{J_l}(R_l)$ S_t: $R_{J_l} \rightarrow S_r$ S_r: $R_{J_lr} := R_{J_l} \bowtie_{J_{lr}} R_r$ S_r: $R_{J_lr} \rightarrow S_t$ S_t: $R_{J_lr} \rtimes R_l$		$\pi_{J_l}(R_l)$	R_l $\pi_{J_l}(R_l) \bowtie_{J_{lr}} R_r$	$[R_l^{\pi}, R_l^{\bowtie}, R_l^{\sigma}]$ $[J_l, R_l^{\bowtie}, R_l^{\sigma}]$ $[J_l \cup R_r^{\pi}, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$
$[S_l,S_t]$	S_l: $R_{J_l} := \pi_{J_l}(R_l)$ S_l: $R_{J_l} \rightarrow S_t$ S_r: $R_r \rightarrow S_t$ S_t: $R_{J_lr} := R_{J_l} \bowtie_{J_{lr}} R_r$ S_t: $R_{J_lr} \rightarrow S_l$ S_l: $R_{J_lr} \rtimes R_l$	$\pi_{J_l}(R_l) \bowtie_{J_{lr}} R_r$		$\pi_{J_l}(R_l)$ R_r	$[J_l, R_l^{\bowtie}, R_l^{\sigma}]$ $[R_r^{\pi}, R_r^{\bowtie}, R_r^{\sigma}]$ $[J_l \cup R_r^{\pi}, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$
$[S_r,S_t]$	S_r: $R_{J_r} := \pi_{J_r}(R_r)$ S_r: $R_{J_r} \rightarrow S_t$ S_l: $R_l \rightarrow S_t$ S_t: $R_{lJ_r} := R_l \bowtie_{J_{lr}} R_{J_r}$ S_t: $R_{lJ_r} \rightarrow S_r$ S_r: $R_{lJ_r} \rtimes R_r$		$R_l \bowtie_{J_{lr}} (\pi_{J_r}(R_r))$	$\pi_{J_r}(R_r)$ R_l	$[J_r, R_r^{\bowtie}, R_r^{\sigma}]$ $[R_l^{\pi}, R_l^{\bowtie}, R_l^{\sigma}]$ $[R_l^{\pi} \cup J_r, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$
$[S_t,S_lS_r]$	S_l: $R_{J_l} := \pi_{J_l}(R_l)$ S_r: $R_{J_r} := \pi_{J_r}(R_r)$ S_l: $R_{J_l} \rightarrow S_t$ S_r: $R_{J_r} \rightarrow S_t$ S_t: $R_{J_lJ_r} := R_{J_l} \bowtie_{J_{lr}} R_{J_r}$ S_t: $R_{J_lJ_r} \rightarrow S_l$ S_t: $R_{J_lJ_r} \rightarrow S_r$ S_l: $R_{lJ_{lr}} := R_l \ltimes R_{J_lJ_r}$ S_r: $R_{J_{lr}r} := R_{J_lJ_r} \rtimes R_r$ S_l: $R_{lJ_{lr}} \rightarrow S_t$ S_r: $R_{J_{lr}r} \rightarrow S_t$ S_t: $R_{lJ_{lr}} \bowtie R_{J_{lr}r}$	$(\pi_{J_l}(R_l)) \bowtie_{J_{lr}} (\pi_{J_r}(R_r))$	$(\pi_{J_l}(R_l)) \bowtie_{J_{lr}} (\pi_{J_r}(R_r))$	$\pi_{J_l}(R_l)$ $\pi_{J_r}(R_r)$ $R_l \ltimes ((\pi_{J_l}(R_l)) \bowtie_{J_{lr}} (\pi_{J_r}(R_r)))$ $((\pi_{J_l}(R_l)) \bowtie_{J_{lr}} (\pi_{J_r}(R_r))) \rtimes R_r$	$[J_l, R_l^{\bowtie}, R_l^{\sigma}]$ $[J_r, R_r^{\bowtie}, R_r^{\sigma}]$ $[J_l \cup J_r, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$ $[J_l \cup J_r, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$ $[R_l^{\pi} \cup J_r, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$ $[J_l \cup R_r^{\pi}, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$

Figure 6: Different strategies for executing a join operation with the intervention of a third party

Figure 6 summarizes the different ways considered in the paper in which a third party S_t can be involved. We briefly comment them here.

- $[S_t$,NULL$]$: the third party S_t receives the relations from the operands and independently computes the (regular) join.
- $[S_t$,$S_l]$ and $[S_t$,$S_r]$: the third party S_t replaces S_r (S_l, resp.) in the computation with the role of master, with S_l (S_r, resp.) in the role of slave.
- $[S_l$,$S_t]$ and $[S_r$,$S_t]$: the third party S_t replaces S_r (S_l, resp.) in the computation with the role of slave, with S_l (S_r, resp.) in the role of master.
- $[S_t$,$S_lS_r]$: the third party S_t takes the role of master in charge of computing the join with S_l and S_r both working as slaves. In this case, each of the operands computes the projection of its attributes that participate in the join and sends it to the third party. The third party computes the join between the two inputs and sends back the result to each of the operands, each of which joins the input with its relation

and returns the result to the third party. The third party can now join the relations received from the operands and compute the result.

Note that the first five scenarios are a simple adaptation of those already seen in the previous section, with the third party only acting as proxy, which therefore needs to have the authorization to view the relation of the party for which it acts as proxy as well as the view required by its role (master/slave). The latter scenario [S_t,S_lS_r] is instead a little more complex and, as it can be easily seen from the table, entails different data views. In this scenario, the third party is required to only view the tuples of the operands that participate in the join (it does not need to have the complete view on one of the operand relations as it is when it acts as proxy). Also, each of the slaves is required only to view the attributes of the other relation that joins with itself (instead of the complete set of attributes).

The consideration of a third party requires to slightly change the executor assignment definition (Definition 4.1) which becomes as follows.

Definition 4.4 (Executor assignment (with third party))**.** *Given a query plan $T(N,E)$, an* executor assignment function $\lambda_T : N \rightarrow \mathcal{S} \times (\mathcal{S} \cup \{[\mathcal{S} \times \mathcal{S}]\} \cup \{\text{NULL}\})$ *is an assignment of pairs of servers to nodes such that:*

1. *each leaf node (corresponding to a base relation R) is assigned the pair $[S, \text{NULL}]$, where S is the server where R is stored;*
2. *each non-leaf node n, corresponding to unary operation* op *on operand R_l (left child) at server S_l, is assigned a pair $[S_l,\text{NULL}]$;*
3. *each non-leaf node n, corresponding to a join on operands R_l (left child) at server S_l and R_r (right child) at server S_r, is assigned a pair* [master,slaves] *such that* master $\in \mathcal{S}$, slaves $\in (\mathcal{S} \cup \{[S_l, S_r]\} \cup \{\text{NULL}\})$, master$\neq$slaves*, and at least one between master and slaves is in $\{S_l,S_r,[S_l,S_r],\text{NULL}\}$.*

The definition of views required by an execution remains as before (with the note that the views of the third party are now also included). The definition of safe assignment and feasible query plan also remain unchanged.

Example 4.2. *Consider query q_2 in Example 2.2 and the set of authorizations in Figure 3. The join between* `Insurance` *and* `Hospital` *(node n_1 in Figure 2) can be safely assigned neither to S_I nor to S_H. It is then necessary to resort to the intervention of a third party, which is S_N. A safe assignment for the given operation is $[S_H,S_N]$. In fact, S_N can act as slave, as a proxy for S_I, since it is authorized to access the whole content of the* `Insurance` *relation (authorization 9) and attribute* Patient *(authorization 10). S_H can act as master since it is authorized to view the* Holder, Plan*, and* Patient *attributes, provided prior enforcement of join condition* $\langle$Holder,Patient$\rangle$ *(authorization 5).*

We can now state our problem as follows.

Problem 4.1. *Given a query plan $T(N,E)$ and a set of authorizations $\mathcal{A}$:* 1) *determine if T is feasible and* 2) *retrieve a safe assignment λ_T for it, assuming possible involvement of third parties.*

In the next section, we illustrate an algorithm that, given a query tree plan and a set of authorizations, determines if the plan is feasible and, if so, returns a safe assignment for it.

5 Algorithm

We follow two basic principles in the determination of a safe assignment: *i)* we favor semi-joins (in contrast to regular joins); *ii)* if more servers are candidate to safely execute a join operation, we prefer the server that is involved in a higher number of join operations. To this aim, we associate, with each candidate server, a counter that keeps track of the number of join operations for which the server is a candidate.

The algorithm in Figure 7 receives in input the set $\mathcal{A}$ of authorizations and a query tree plan $T(N, E)$ and, if T is feasible, returns a safe executor assignment λ_T for it (solving Problem 4.1). Each node n in the query tree plan is associated with the following variables: *n.left* and *n.right* are the left and right children; *n.operator* and *n.parameter* are the operation and its parameters; *n.leftslave* and *n.rightslave* are the left and right slaves; *n.candidates* is a list of records of the form [*server*,*fromchild*,*counter*] stating candidate servers, the child (left, right, left_right, third, third_left, third_right) it comes from, and the number of joins for which the server is a candidate in the subtree rooted at n; *n.executor.master* and *n.executor.slave* represent the executor assignment computed by the algorithm for n (i.e., $\lambda_T(n)$); and [$n.\pi$,$n.\bowtie$,$n.\sigma$] is the profile of the node. We assume that for each leaf node n (base relation R) the list *n.candidates* of candidate executors is initialized to [*server*,_,0], where *server* is the server storing the relation. Also, variables *n.executor.master* and *n.executor.slave* are set to *server* and NULL, respectively.

The algorithm performs two traversals of tree T. The first traversal (procedure **Find_candidates**) visits the tree in *post-order*. At each node n, the profile of the node is computed (as described in Figure 4) based on the profiles of its children and on the operation associated with the node. Then, the set *n.candidates* of possible candidate executors for the node is computed by analyzing the candidate executors of its children. If the node is a unary operation, the candidate executors for the node are all the candidate executors of its only child (we suppose that if a node has only one child, it is the left one). If the node is a join operation, procedure **Find_candidates** proceeds by checking whether the candidate executors of its left and right children can act as slave and master, respectively, and vice versa (i.e., it then verifies whether the left and right candidate executors can act as master and slave, respectively). To this purpose, the procedure considers the candidate executors of

Algorithm 5.1 (Safe assignment computation).

```
/* Input: 𝒜, T(N,E) */
/* Output: λ_T(n) as n.executor */

/* n.left, n.right: left and right children */
/* n.operator, n.parameter: operation and its parameters */
/* [n.π,n.⋈,n.σ]: profile */
/* n.leftslave, n.rightslave: left and right slaves */
/* n.leftthirdslave, n.rightthirdslave: third parties acting as left and right slaves */
/* n.candidates: list of records of the form [server,fromchild,counter] stating candidate servers, the child
   (left, right, left_right, third, third_left, third_right) it comes or proxies for, and the number of joins for which the server
   is candidate in the subtree */
/* n.executor.master, n.executor.slave: executor assignment */

MAIN
Find_candidates(root(T))
Assign_ex(root(T), NULL)
return(T)

FIND_CANDIDATES(n)
l := n.left; r := n.right
if l ≠NULL then Find_candidates(l) /* recursive call on the left child */
if r ≠NULL then Find_candidates(r) /* recursive call on the right child */
case n.operator of
  π: n.π := n.parameter; n.⋈ := l.⋈; n.σ := l.σ
     /* the candidates of the node are the candidate of its left child */
     for c in l.candidates do Add [c.server, left, c.count] to n.candidates
  σ: n.π := l.π; n.⋈ := l.⋈; n.σ := l.σ ∪ n.parameter
     /* the candidates of the node are the candidate of its left child */
     for c in l.candidates do Add [c.server, left, c.count] to n.candidates
  ⋈: n.π := l.π ∪ r.π; n.⋈ := l.⋈ ∪ r.⋈ ∪ n.parameter; n.σ := l.σ ∪ r.σ
     right_slave_view := [J_l, l.⋈, l.σ]
     left_slave_view := [J_r, r.⋈, r.σ]
     right_master_view := [l.π ∪ J_r, l.⋈ ∪ r.⋈ ∪ n.parameter, l.σ ∪ r.σ]
     left_master_view := [J_l ∪ r.π, l.⋈ ∪ r.⋈ ∪ n.parameter, l.σ ∪ r.σ]
     right_full_view := [l.π, l.⋈, l.σ]
     left_full_view := [r.π, r.⋈, r.σ]
     /* check case [S_r,NULL] and [S_r,S_l] */
     n.leftslave := NULL
     c := GetFirst(l.candidates)
     while (n.leftslave=NULL)AND(c≠NULL) do
          /* candidate for the left child that can serve as a slave */
          if Can_view(left_slave_view, c.server) then n.leftslave := c.server
          c := c.next
     regular := NULL; rightmasters = NULL
     for c in r.candidates do
        /* candidates of the right child that can execute a regular join */
        if Can_view(right_full_view, c.server) then Add [c.server, right, c.count+1] to regular
        /* candidates of the right child that can serve as a master */
        if Can_view(right_master_view, c.server) then Add [c.server, right, c.count+1] to rightmasters
     if n.leftslave≠NULL then Add rightmasters to n.candidates
       else Add regular to n.candidates
     /* check case [S_l,NULL] and [S_l,S_r] */
     n.rightslave := NULL
     c := GetFirst(r.candidates)
     while (n.rightslave=NULL)AND(c≠NULL) do
          /* candidate for the right child that can serve as a slave */
          if Can_view(right_slave_view, c.server) then n.rightslave := c.server
          c := c.next
     regular := NULL; leftmasters = NULL
     for c in l.candidates do
        if ∃c′∈n.candidates : c.server=c′.server then /* candidate for both left and right children */
          c′.count := c′.count + c.count
          c′.fromchild := left_right
        else
          /* candidates of the left child that can execute a regular join */
          if Can_view(left_full_view, c.server) then Add [c.server, left, c.count+1] to regular
          /* candidates of the left child that can serve as a master */
          if Can_view(left_master_view, c.server) then Add [c.server, left, c.count+1] to leftmasters
     if n.rightslave≠NULL then Add leftmasters to n.candidates
       else Add regular to n.candidates
     if n.candidates=NULL then n.candidates := Find_third_party(n,leftmasters,rightmasters) /* check third party */
     /* there does not exists a safe executor assignment for the node */
     if n.candidates=NULL then exit(n)

CAN_VIEW(profile,S)
for each [A, J] ∈view(S) do /* check the profile with all the authorizations of the server */
  if ((profile.π∪profile.σ)⊆ A) AND (profile.⋈=J) then return(TRUE)
return(FALSE)
```

Figure 7: Pseudocode of the algorithm computing a safe assignment for a query tree plan

the left child in decreasing order of join counter (**GetFirst**) and stops at the first candidate executor found that can serve as left slave (inserting it into local variable *leftslave*). The procedure then examines all the candidate executors of the right child to determine if they can work as master for a semi-join (if a left slave was found) or for a regular join (if no left slave was found). Note that while we need to identify all the servers that can act as master, as we need to consider all possible candidate executors for propagating them upwards in the tree, it is sufficient to identify one slave (a slave is not propagated upward in the tree). For each right candidate executor *server* identified for node n, the triple [*server*,right,*counter*] is added to the $n.candidates$ list, where *counter* is the counter that was associated with *server* in the right child of the node incremented by one (as candidate executor also for the join of the father, the server would execute one additional join compared to the number it would have executed at the child level). Then, the procedure proceeds symmetrically to determine whether there is a candidate from the right child (considering the candidates in decreasing order of counter) that can work as slave and, as before, stops at the first candidate found that can serve as right slave (inserting it into local variable *rightslave*). After that, the procedure examines all the candidate executors of the left child of the node. In particular, for each left candidate executor *server*, the procedure first checks if *server* already appears as a candidate executor for the node itself, that is, [*server*,right,*counter*] is in *candidates*, meaning that the left child and the right child have a common candidate server. If this is the case, [*server*,right,*counter*] is updated by adding to *counter* the value of the variable *counter* associated with *server* in the candidate list of the left child. Also, the *right* value in [*server*,right,*counter*] is updated to *left_right*. If *server* does not already appear as a candidate for the node, the algorithm determines if *server* can work as master for a semi-join (if a right slave was found) or for a regular join (if no right slave was found) and updates the *candidates* list accordingly. Note that procedure **Find_candidates** calls function **Can_view** whenever it is necessary to verify whether a server can work as slave/master for a semi-join or can execute a regular join. **Can_view** takes in input the profile of the view that should be made visible in the execution of an operation along with a server and returns true if the view is authorized.

At the end of this process, list *candidates* contains all the candidate servers coming from either the left or right child that can execute the join in any of the execution modes involving only the two operand parties (Figure 5). If no candidate executor was found, the algorithm determines whether the operation can be executed with the intervention of a third party (different from the candidate executors of the left and right children of the node) by calling function **Find_third_party** (Figure 8). Similarly for the cases above, function **Find_third_party** enforces the controls necessary to verify whether there are third parties authorized to evaluate the join operation according to the strategies reported in Figure 6. The function receives in input the current node n for which *candidates* list is empty, along with the corresponding sets *leftmasters* and *rightmasters*, and returns a possible *list* of candidate executors. The function checks the following four cases in the order, applying two main

```
FIND_THIRD_PARTY(n,leftmasters,rightmasters)
l := n.left; r := n.right; list := NULL
right_slave_view := [J_l, l.⋈, l.σ]
left_slave_view := [J_r, r.⋈, r.σ]
right_master_view := [l.π ∪ J_r, l.⋈ ∪ r.⋈ ∪ n.parameter, l.σ ∪ r.σ]
left_master_view := [J_l ∪ r.π, l.⋈ ∪ r.⋈ ∪ n.parameter, l.σ ∪ r.σ]
right_full_view := [l.π, l.⋈, l.σ]
left_full_view := [r.π, r.⋈, r.σ]
two_slave_view := [J_l∪J_r, l.⋈ ∪ r.⋈ ∪ n.parameter, l.σ ∪ r.σ]
/* check if a third party can act as slave */
if leftmasters≠NULL then /* case [S_l,S_t] */
  n.rightthirdslave := NULL
  cand_slave := 𝒮\(l.candidates∪r.candidates)
  while (n.rightthirdslave=NULL)AND(cand_slave≠∅) do
      S := Extract(cand_slave)
      /* server that can work as a right slave */
      if Can_view(right_slave_view, S) AND Can_view(left_full_view, S) then n.rightthirdslave := S
if n.rightthirdslave≠NULL then
  for each c ∈ leftmasters do Add [c.server, left, c.count] to list
if rightmasters≠NULL then /* case [S_r,S_t] */
  n.leftthirdslave := NULL
  cand_slave := 𝒮\(l.candidates∪r.candidates)
  while (n.leftthirdslave=NULL)AND(cand_slave≠∅) do
      S := Extract(cand_slave)
      /* server that can work as a left slave */
      if Can_view(left_slave_view, S) AND Can_view(right_full_view, S) then n.leftthirdslave := S
if n.leftthirdslave≠NULL then
  for each c ∈ rightmasters do Add [c.server, right, c.count] to list
if list≠NULL then return(list)
/* check if a third party can act as master */
for each S∈𝒮\(l.candidates∪r.candidates) do
  if n.leftslave≠NULL then /* case [S_t,S_l] */
    /* server that can proxy for the right child */
    if Can_view(right_master_view, S) AND Can_view(left_full_view, S) then Add [S, third_right, 1] to list
  else
    if n.rightslave≠NULL then /* case [S_t,S_r] */
      /* server that can proxy for the left child */
      if Can_view(left_master_view, S) AND Can_view(right_full_view, S) then Add [S, third_left, 1] to list
if list≠NULL then return(list)
/* check if a third party can execute the regular join: case [S_t,NULL] */
for each S∈𝒮\(l.candidates∪r.candidates) do
  /* server that can execute a regular join */
  if Can_view(left_full_view, S) AND Can_view(right_full_view, S) then Add [S, third, 1] to list
if list≠NULL then return(list)
/* check if a third party can act as coordinator: case [S_t,S_lS_r] */
c:= GetFirst(l.candidates)
while (n.leftslave=NULL)AND(c≠NULL) do
    if Can_view(two_slave_view, c.server) then n.leftslave := c.server
      c := c.next
if n.leftslave≠NULL then
  c:= GetFirst(r.candidates)
  while (n.rightslave=NULL)AND(c≠NULL) do
      if Can_view(two_slave_view, c.server) then n.rightslave:= c.server
        c := c.next
if n.rightslave≠NULL then
  masterlist := NULL
  for each S∈𝒮\(l.candidates∪r.candidates) do
    if Can_view(left_slave_view, S) AND Can_view(right_slave_view, S)
      AND Can_view(left_master_view, S) AND Can_view(right_master_view, S)
      then Add S to masterlist /* server that can act as coordinator */
if masterlist≠NULL then
  for each m∈masterlist do Add [m, third, 1] to list
if list≠NULL then return(list)
```

Figure 8: Pseudocode of the function that evaluates the intervention of a third party for join operations

principles: *i)* solutions where the third party acts as slave are preferred to solutions where it acts as master to avoid the propagation of the third party up in the tree; *ii)* semi-joins are preferred to regular joins. Note that the case where the third party acts as coordinator is the last option, since it is less efficient than all the other execution modes. The function stops at the first case that produces a candidate *list* that is not NULL.

Case 1. The function verifies whether a server can work as left/right slave, depending on whether *rightmasters*/*leftmasters* is not NULL. In particular, the function first verifies whether there exists a server in the system that can work as right slave and inserts it into local variable *rightthirdslave*. If such a server exists (i.e., *rightthirdslave* is not NULL), for each candidate *server* in *leftmasters*, a triple [*server*,left,*counter*] is added to the candidates *list*. Then, the function proceeds symmetrically to determine whether there exists a server in the system that can work as left slave (inserting it into local variable *leftthirdslave*). If *leftthirdslave* is not NULL, for each candidate *server* in *rightmasters*, a triple [*server*,right,*counter*] is added to the candidate *list*.

Case 2. The function verifies whether there is a *server* that can work as master. For each *server* in the system, if *leftslave* is not NULL and *server* can work as right master, triple [*server*,third_right,1] is added to the candidate *list*. Otherwise (i.e., *leftslave* is NULL), if *rightslave* is not NULL, for each *server* in the system that can work as left master, triple [*server*,third_left,1] is added to the candidate *list*.

Case 3. The function verifies whether there is a *server* in the system that can execute the join represented by node n as regular join. For each *server* in the system, if *server* can execute the regular join, triple [*server*,third,1] is added to the candidate *list*.

Case 4. The function verifies whether there is a *server* that can act as coordinator. The function considers candidate executors of the left child in decreasing order of join counter (**GetFirst**) and stops at the first candidate server found that can serve as slave in the coordinator join evaluation mode, inserting it into local variable *leftslave*. If a candidate left slave has been determined (i.e., *leftslave* is not NULL), the function repeats the same process on the candidate executors of the right child thus checking the existence of a candidate right slave that is inserted in variable *rightslave*. If also *rightslave* is not NULL, for each *server* in the system, the function verifies whether it can work as coordinator for the join operation. If so, the candidate *list* is updated by adding the triple [*server*,third,1].

If the call to function **Find_third_party** does not return any candidate executor, the algorithm exits returning the node at which the process was interrupted (i.e., for which no safe assignment exists) signaling that the tree plan is not feasible.

If **Find_candidates** completes successfully, the algorithm proceeds with the second traversal of the query tree plan. The second traversal (procedure **Assign_ex** in Figure 9) recursively visits the tree in *pre-order*. At the root node, if more assignments are possible, the candidate server with the highest join counter is chosen.

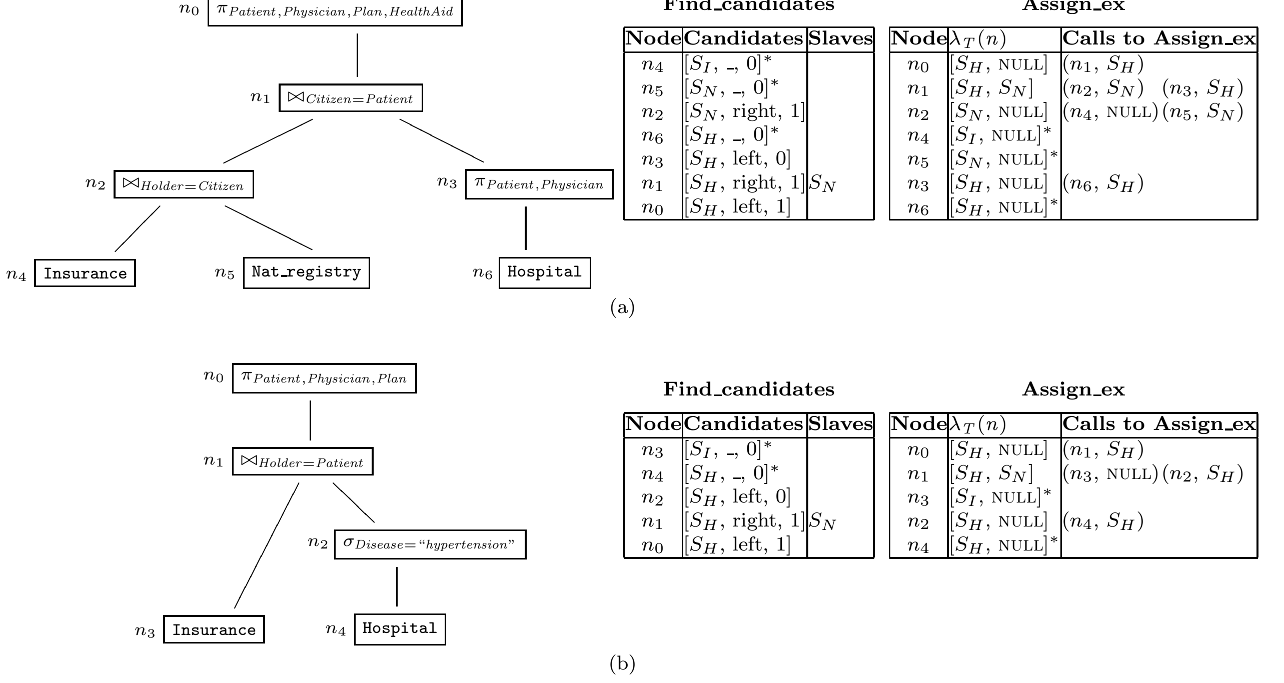
```
ASSIGN_EX(n, from_parent)
if from_parent≠NULL then
  chosen := Search(from_parent, n.candidates)
else chosen := GetFirst(n.candidates)
n.executor.master := chosen.server /* assign master executor */
case chosen.fromchild of
  left: /* case [Sl,NULL], [Sl,Sr], [Sl,St] */
    if n.left≠NULL then Assign_ex(n.left, n.executor.master)
    if n.right≠NULL then
      if   n.rightslave≠NULL then
           n.executor.slave := {n.rightslave} /* the slave is a candidate for the right child */
           Assign_ex(n.right, n.rightslave)
      else n.executor.slave := {n.rightthirdslave} /* the slave is either a third party or does not exist */
            Assign_ex(n.right, NULL)
  right: /* case [Sr,NULL], [Sr,Sl], [Sr,St] */
    if n.left≠NULL then
      if   n.leftslave≠NULL then
           n.executor.slave := {n.leftslave} /* the slave is a candidate for the left child */
           Assign_ex(n.left, n.leftslave)
      else n.executor.slave := {n.leftthirdslave} /* the slave is either a third party or does not exist */
            Assign_ex(n.left, NULL)
    if n.right≠NULL then Assign_ex(n.right, n.executor.master)
  left_right: /* case [Sl,NULL], [Sr,NULL] when Sl=Sr */
    if n.left≠NULL then Assign_ex(n.left, n.executor.master)
    if n.right≠NULL then Assign_ex(n.right, n.executor.master)
  third_left: /* case [St,Sr] */
    n.executor.slave := {n.rightslave}
    if n.left≠NULL then Assign_ex(n.left, NULL)
    if n.right≠NULL then Assign_ex(n.right, n.rightslave)
  third_right: /* case [St,Sl] */
    n.executor.slave := {n.leftslave}
    if n.left≠NULL then Assign_ex(n.left, n.leftslave)
    if n.right≠NULL then Assign_ex(n.right, NULL)
  third: /* case [St,NULL], [St,SlSr] */
    n.executor.slave := {n.leftslave, n.rightslave} /* the slave is either a pair of servers or does not exist */
    if n.left≠NULL then Assign_ex(n.left, n.leftslave)
    if n.right≠NULL then Assign_ex(n.right, n.rightslave)
```

Figure 9: Pseudocode of procedure **Assign_ex**

Hence, the chosen candidate executor is pushed down to the child(s) from which it comes in the post-order traversal, and the selected slave is pushed down to the other child. If no slave was recorded as possible (i.e., *rightslave*/*leftslave*=NULL or the slave is a third party) a NULL value is pushed down. At each child, the master executor is determined as the server pushed down by the parent (if it is not NULL) or the candidate server with the highest join counter and the process is recursively repeated, until a leaf node is reached.

Example 5.1. *Consider the two query plans in Figure 2 (reported also in Figure 10 for convenience) and the set of authorizations in Figure 3. Figure 10 illustrates the working of procedures* **Find_candidates** *and* **Assign_ex** *reporting the nodes in the order they are considered by the procedures and the candidates/executors determined. Candidates/executors with a "*" are those of the leaf nodes (already given in input). To illustrate the working, let us look at some sample calls.*

Consider query q_1 and, in particular, call **Find_candidates**(n_2)*. Among the candidates of the children (S_I from left child n_4, and S_N from right child n_5) only the right child candidate S_N survives as candidate for the join, which needs to be executed as a regular join since the only candidate from the left child cannot serve as slave. When* **Assign_ex** *is called, the set of candidates at each node is as shown in the table summarizing the results of* **Find_candidates***. Starting at the root node, the only possible choice consists in assigning to n_0 executor $[S_H, \text{NULL}]$, where S_H was recorded as coming from the left (and only) child n_1, to which S_H is then*



(a)

Find_candidates

Node	Candidates	Slaves
n_4	$[S_I$, _, 0]*	
n_5	$[S_N$, _, 0]*	
n_2	$[S_N$, right, 1]	
n_6	$[S_H$, _, 0]*	
n_3	$[S_H$, left, 0]	
n_1	$[S_H$, right, 1]	S_N
n_0	$[S_H$, left, 1]	

Assign_ex

Node	$\lambda_T(n)$	Calls to Assign_ex
n_0	$[S_H$, NULL]	(n_1, S_H)
n_1	$[S_H, S_N]$	(n_2, S_N) (n_3, S_H)
n_2	$[S_N$, NULL]	(n_4, NULL)(n_5, S_N)
n_4	$[S_I$, NULL]*	
n_5	$[S_N$, NULL]*	
n_3	$[S_H$, NULL]	(n_6, S_H)
n_6	$[S_H$, NULL]*	

(b)

Find_candidates

Node	Candidates	Slaves
n_3	$[S_I$, _, 0]*	
n_4	$[S_H$, _, 0]*	
n_2	$[S_H$, left, 0]	
n_1	$[S_H$, right, 1]	S_N
n_0	$[S_H$, left, 1]	

Assign_ex

Node	$\lambda_T(n)$	Calls to Assign_ex
n_0	$[S_H$, NULL]	(n_1, S_H)
n_1	$[S_H, S_N]$	(n_3, NULL)(n_2, S_H)
n_3	$[S_I$, NULL]*	
n_2	$[S_H$, NULL]	(n_4, S_H)
n_4	$[S_H$, NULL]*	

Figure 10: An example of algorithm execution over queries q_1 (a) and q_2 (b) of Example 2.2

pushed down with a recursive call. At n_1 the master is set to S_H and, combining this with the slave field, the executor is set to $[S_H, S_N]$. Hence, S_H is further pushed down to the right child n_3 (from where it was taken by **Find_candidates***), while S_N is pushed down to the left child n_2.*

Consider query q_2 and, in particular, call **Find_candidates**(n_1)*. Among the candidates of the children (S_I from left child n_3, and S_H from right child n_2) only the right child candidate S_H survives as candidate for the join, which needs to be executed as a semi-join with the collaboration of third party S_N working as slave. When* **Assign_ex** *is called, the set of candidates at each node is as shown in the table summarizing the results of* **Find_candidates***. Note that call* **Assign_ex**(n_1) *assigns to n_1 the pair $[S_H, S_N]$ as an executor. Then, S_H is further pushed down to the right child n_2 (from where it was derived by* **Find_candidates***): by contrast, S_N is not pushed down in the tree, since it is a third party.*

The executor assignment computed by procedure **Assign_ex** implicitly determines the flows of information among cooperating servers necessary for query execution. These data exchanges practically translate into *send* and *receive* operations, which permit to transfer a set of tuples from a server (the sender) to another (the receiver).

Example 5.2. *Consider the query plans and executor assignments in Figure 10. Figure 11 illustrates the server responsible for the evaluation of each operation in the plan and the data exchanges these assignments imply. Each box delimits the operations for which each server is responsible and edges across boxes represent flows of*

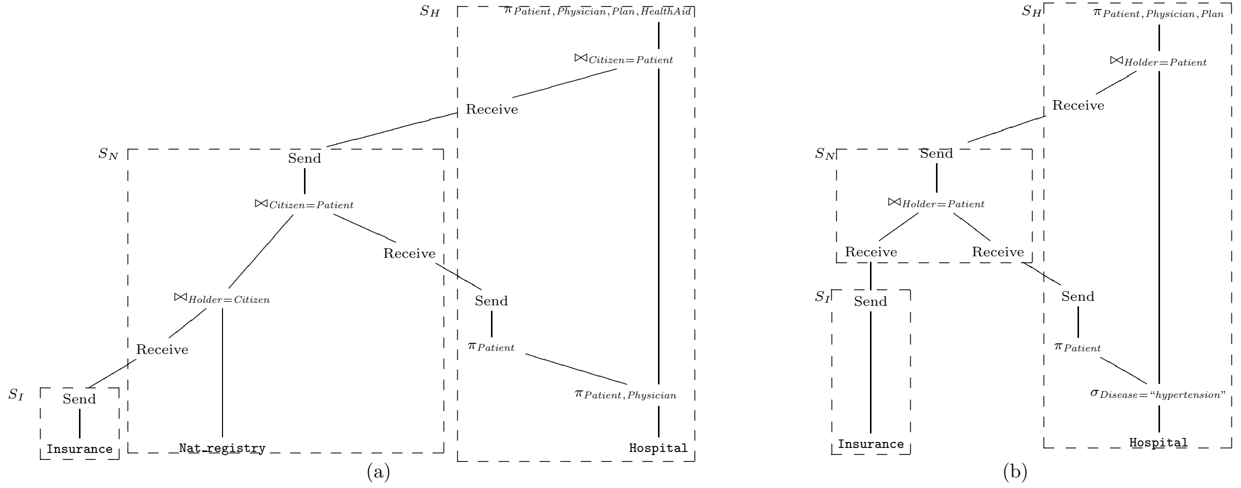


Figure 11: Operations and data flows implied by the execution of queries q_1 (a) and q_2 (b), following the assignment in Figure 10

information between servers.

For the evaluation of query q_1, S_N is responsible for the evaluation of the (regular) join of relations `Insurance` *and* `Nat_registry`*. Therefore, S_I sends to S_N its relation. The evaluation of the join represented by node n_1 requires instead the cooperation of S_H, working as master, and S_N, working as slave. As a consequence, S_H sends the projection of attribute* Patient *to S_N, which can then restrict the set of tuples resulting from the evaluation of n_2 and send back the result to S_H. S_H locally holds all the information to complete the evaluation of the query. The evaluation of query q_2 is analogous.*

The correctness of the algorithm is proved by showing that: *i)* the algorithm always terminates, *ii)* the returned executor assignment (if any) is safe, and *iii)* if there exists a safe assignment for a query tree plan, the algorithm finds it. The following theorem states the correctness of the algorithm.

Theorem 5.1 (Correctness)**.** *Given a query tree plan $T(N,E)$, a set $\mathcal{A}$ of authorizations, and a set $\mathcal{S}$ of servers, Algorithm 5.1:* i) *terminates,* ii) *computes an executor assignment for T that is safe;* iii) *is always able to find a safe executor assignment for T, if T is feasible.*

PROOF: See Appendix.

The time complexity of the algorithm is linear in the size of its input parameters, as stated by the following theorem.

Theorem 5.2 (Complexity)**.** *Given a query tree plan $T(N,E)$, a set $\mathcal{A}$ of authorizations, and a set $\mathcal{S}$ of servers, the time complexity of Algorithm 5.1 is $O(|N| \cdot |\mathcal{S}| \cdot |\mathcal{A}|)$.*

PROOF: See Appendix.

6 Distributed architecture

We briefly discuss the integration of our approach with distributed query processing architectures. In distributed databases the processing of queries is typically supported by a distributed query optimizer, which has access to a complete description of the relation schemas and statistic profiles. The distributed query optimizer, available on a location known by clients, receives the query request and produces a plan for the distributed query. The distributed optimizer is essentially an extension of a traditional centralized SQL query optimizer. Compared to a centralized optimizer, it also has to take into account all the parameters of the distributed system that have an impact on query execution. The use of a single query optimizer is supported by the experience in the construction of distributed relational systems, which has shown that a fully distributed approach, using negotiation among the servers to identify the query plan, would be too complex and considerably impact performances, since it would require a complex protocol for the exchange of the costs of the different alternatives and the identification of a globally optimal solution.

The main impact of the model proposed in this paper on the distributed system architecture is the need for the query optimizer to consider the authorizations defined by each server to determine a proper assignment compliant with the authorizations. As discussed in Section 4.1, depending on authorizations, the join operators can be realized in a variety of ways, characterized by different assignments of operators to servers. The portion of the database catalog accessed by the optimizer to identify a query plan has then to be extended with the description of all the authorizations.

The services of the distributed query optimizer are invoked by any client that desires to execute a query (see Figure 12). The Access control and server assignment module receives from the client a query (step 2 in Figure 12) that requires distributed computation. It first checks if the user is authorized for the profile resulting from the query and, if the user is not authorized, the Access control and server assignment module returns an error message. Otherwise, it identifies an efficient execution plan able to return to the requesting user the result of the query (steps 2-5 in Figure 12). The client can then execute the query, by dispatching to the servers in the network the assigned portions of the plan (step 6 in Figure 12).

It is important to note that, to satisfy basic security requirements, an adequate identity infrastructure has to be available to guarantee the satisfaction of the authorizations defined on the data. The identity infrastructure will support the authentication of all the parties involved: clients, servers, and the distributed query optimizer. Security also requires to consider the integrity of the catalog, describing the database schemas and the associated authorizations. The query optimizer is a relatively large software component, typically designed without paying attention to security requirements: it is important that security does not rely on it and the architecture in Figure 12 satisfies this requirement by isolating the optimizer from the evaluation of security aspects. To

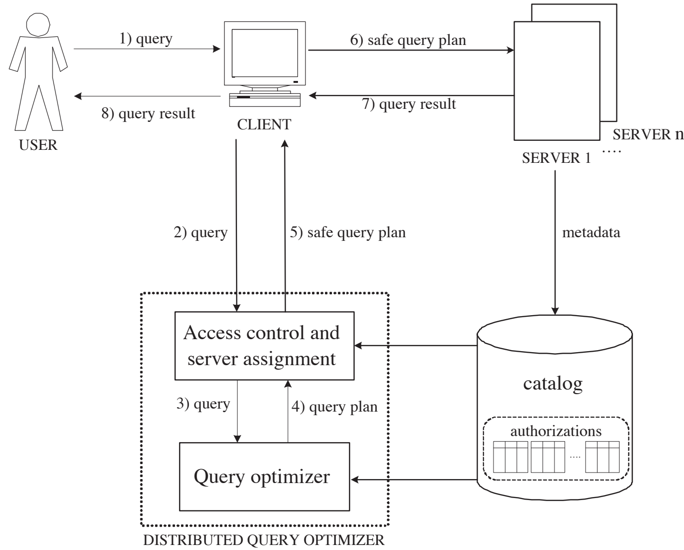


Figure 12: Architecture of the distributed system

guarantee query safety, it is sufficient to assume trust in the catalog, which contains the user authorizations, and in the Access control and server assignment module, which computes the safety checks as described by procedure **Find_candidates**. Signatures on the plan will support each server in determining that the portion of the plan the server is responsible to execute complies with all the specified authorizations.

We consider the integration with an optimizer using a 2-phase process for the identification of the distributed query execution plan. This is an approach currently employed by several distributed query optimizers [17]: *1)* the query optimizer identifies an efficient query plan, analogous to the one it would produce for a centralized system, e.g., using dynamic programming for the identification of the plan; *2)* the query optimizer assigns operations to the distinct servers in the system, realizing the *site selection* phase. Our algorithm nicely fits into such a 2-phase structure. Specifically, the first phase of the optimization process produces a query plan that can be immediately used as input to the **Find_candidates** procedure, which labels every node with the identifiers of the servers authorized to execute the operator. The second phase of the optimization process represents instead the activity realized by procedure **Assign_ex**, which selects a server for each node among those that have been identified as candidates by the first phase. Procedure **Assign_ex** applies a greedy selection aiming to minimize information flow and parties involvement (favoring, among the candidate executors of a node, the server assigned to its ancestors). The procedure guarantees to identify a safe assignment, if one exists. Other site assignment techniques [8, 16] can be easily adapted in this scenario.

Finally, we observe that our approach is applicable to both *dynamic* and *static* queries. Dynamic queries are queries that are created at run-time by a user or an application. Dynamic queries are processed by the query optimizer and executed immediately after optimization; the execution plan is typically discarded after the

query has completed. Dynamic queries always consider the current state of every component of the system and therefore updates to the policy are automatically taken into consideration. Static queries are instead queries that appear within the code of an application, within SQL procedures or in programs using SQL Embedded. These queries are typically processed by the query optimizer at compile-time, creating an object representation of the query plan that will be directly executed at every query invocation. The specific issue that needs to be taken into account for static queries is the impact that changes on policies have on a query plan, because an authorized query plan can become invalid, for example, if a policy is modified by dropping a privilege required at some point in the plan. Fortunately, the mechanisms within relational databases already support the representation of dependencies for query plans, making it easy to introduce an explicit management of dependencies of compiled query plans on policy updates. In other words, by specifying a dependency between a query plan and the authorizations supporting it, possible changes to the policy will automatically invalidate the query plan forcing its recomputation next time the query is executed.

7 Related work

The lines of research that have an impact on the presented proposal can be classified into two categories. On one hand we have classical works on the management of queries in centralized and distributed systems [3, 6, 7, 17, 19, 23, 25]; these approaches describe how efficient query plans can be obtained, but do not take into consideration constraints on attribute visibility for servers. On the other hand, in light of the crucial role that security has in the construction of future large-scale distributed applications, a significant amount of research has recently focused on the problem of processing distributed queries under protection requirements. Most of these works [5, 13, 14, 15, 18, 21] are based on the concept of *access pattern*, a profile associated with each relation/view, where each attribute has a value that may either be i or o (i.e., input or output). When accessing a relation, the values for all i attributes must be supplied to obtain the corresponding values of o attributes. Also, queries are represented in terms of Datalog, a query language based on the logic programming paradigm. The main goal of all these works is that of identifying the classes of queries that a given set of access patterns can support; a secondary goal is the definition of query plans that match the profiles of the involved relations, while minimizing some cost parameters (e.g., the number of accesses to data sources [5]). To this aim, relations not explicitly belonging to the query are possibly involved. The main difference between the work on access patterns and the approach presented in this paper is that our proposal can be considered a natural extension of the approach normally used to describe database privileges in a relational schema. Our solution essentially introduces only a mechanism to define access privileges on join paths; instead, access patterns describe authorizations as special formulas in a logic programming language for data access. In our view, the

two models are complementary and both needed, in the same way as both procedural and declarative languages are typically considered in the design of a relational database engine. The model we propose is certainly easier to integrate with the mechanisms and approaches that are used by current database servers (and are familiar to database administrators). Also, the model presented in this paper explicitly manages a scenario with different independent subjects who may cooperate in the execution of a query, whereas the work done on access patterns only considers two actors, the owner of the data and a single user accessing the data. The problem of guaranteeing data security in the distributed database scenario has also been studied in [4], where the authors propose a view-based access control system for restricting both access to data and information flows. The access control policies regulating access to a view are derived from the policies regulating access to the base tables used in the view definition. Access to a view is permitted only if the view is *valid*, meaning that the server that defined the view holds all the privileges for its computation and granted the requester the right to access the content of the view itself. The main goal of this work is that of identifying the server where to define, store, and enforce access control rules both on base tables and on views. The approach in [4] addresses therefore issues complementary to those presented in this paper, which illustrates an approach for expressing authorizations regulating information exchanges among servers and for executing queries entailing only information exchanges allowed by the authorizations.

Our proposal also exploits the results on authorization composition presented in [10]. The novelty of this proposal is that data release is not subject to the simple relation profile-authorization control, but a relation (base or resulting from the evaluation of a query) can be released whenever the information carried by the relation (either directly or indirectly due to the dependence of the query result with other data not explicitly released) is legitimate according to the authorizations explicitly specified for the requesting subject. Therefore, the definition of safe authorization composition is proposed. Also, the process of composing authorizations is proved to be efficient, since it runs in polynomial time in the number of authorizations.

Sovereign joins [1] are an alternative solution for secure information sharing. This method, differently from our, is based on a secure coprocessor, which is involved in query execution, and exploits cryptography to grant privacy. The advantage of sovereign joins is that they extend the plans that allow an execution in the scenario we present; the main obstacle is represented by their high computational cost, due to the use of specific asymmetric cryptography primitives, which make them currently not applicable when large collections of sensitive information must be combined (this is the reason we chose not to consider them in the construction of the plan).

Other related work is represented by approaches enforcing access control in federated systems [9, 12, 20, 24]. Such approaches however mostly focus on the merging of security specifications, and therefore solving possible inconsistencies among them, towards the establishment of a common security policy for regulating the federated

system. Furthermore, they do not address the problem of processing distributed queries under protection requirements. Such solutions are therefore not applicable to our problem where the goal is to maintain complete independence of the different parties that need only to participate in the distributed computation.

8 Conclusions

Adequate support for the integration of information sources detained by distinct parties is an important requirement for future information systems. A crucial issue in this respect is the definition of integration mechanisms that correctly satisfy the commercial and business policies of the organizations owning the data. To solve this problem, we propose a new model based on the characterization of access privileges for a set of servers on the components of a relational schema. Compared to other proposals, the model promises to be more directly applicable to current infrastructures and presents a greater compatibility with the basic features of current DBMSs, which are at the heart of all modern information systems.

Acknowledgments

This work was supported in part by the EU within the 7FP project "PrimeLife" under grant agreement 216483 and by the Italian Ministry of Research within the PRIN 2008 project "PEPPER" (2008SY2PH4). The work of Sushil Jajodia was partially supported by the National Science Foundation under grants CT-20013A, CT-0716567, CT-0716323, CT-0627493, and CCF-1037987.

References

[1] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In *Proc. of the 22nd International Conference on Data Engineering (ICDE 2006)*, Atlanta, GA, USA, April 2006.

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

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

[4] E. Bertino and L.M. Haas. Views and security in distributed database management systems. In *Proc. of the 1st International Conference on Extending Database Technology (EDBT 1988)*, Venice, Italy, March 1988.

[5] A. Calì and D. Martinenghi. Querying data under access limitations. In *Proc. of the 24th International Conference on Data Engineering (ICDE 2008)*, Cancun, Mexico, April 2008.

[6] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.

[7] D.M. Chiu and Y.C. Ho. A methodology for interpreting tree queries into optimal semi-join expressions. In *Proc. of the 1980 ACM SIGMOD International Conference on Management of Data (SIGMOD 1980)*, Santa Monica, CA, USA, May 1980.

[8] D.W. Cornell and P.S. Yu. On optimal site assignment for relations in the distributed database environment. *IEEE Transactions on Software Engineering*, 15(8):1004–1009, August 1989.

[9] S. Dawson, S. Qian, and P. Samarati. Providing security and interoperation of heterogeneous systems. *Distributed and Parallel Databases*, 8(1):119–145, January 2000.

[10] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Assessing query privileges via safe and efficient permission composition. In *Proc. of the 15th ACM Conference Conference on Computer and Communications Security (CCS 2008)*, Alexandria, VA, USA, October 2008.

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

[12] S. De Capitani di Vimercati and P. Samarati. An authorization model for federated systems. In *Proc. of 4th European Symposium on Research in Computer Security (ESORICS 1996)*, Rome, Italy, September 1996.

[13] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. In *Proc. of the 10th International Conference on Database Theory (ICDT 2005)*, Edinburgh, Scotland, January 2005.

[14] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD 1999)*, Philadelphia, PA, USA, June 1999.

[15] G. Gottlob and A. Nash. Data exchange: Computing cores in polynomial time. In *Proc. of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2006)*, Chicago, IL, USA, June 2006.

[16] A. Helal, Y.-S. Kim, M.H. Nodine, A.K. Elmagarmid, and A.A. Heddaya. Transaction optimization techniques. In S. Jajodia and L. Kerchberg, editors, *Advanced Transaction Models and Architectures*. Kluwer, 1997.

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

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

[19] G.M. Lohman, D. Daniels, L.M. Haas, R. Kistler, and P.G. Selinger. Optimization of nested queries in a distributed relational database. In *Proc. of the 10th International Conference on Very Large Data Bases (VLDB 1984)*, Singapore, August 1984.

[20] P. Mitra, C. Pan, P. Liu, and V. Atluri. Privacy-preserving semantic interoperation and access control of heterogeneous databases. In *Proc. of the ACM Symposium on Information, Computer and Communications Security (ASIACCS 2006)*, Taipei, Taiwan, March 2006.

[21] A. Nash and A. Deutsch. Privacy in GLAV information integration. In *Proc. of the 10th International Conference on Database Theory (ICDT 2005)*, Barcelona, Spain, January 2007.

[22] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer-Verlag, 2001.

[23] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD 1979)*, Boston, MA, USA, May - June 1979.

[24] J. Warner, V. Atluri, and R. Mukkamala. A credential-based approach for facilitating automatic resource sharing among ad-hoc dynamic coalitions. In *Proc. of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2005)*, Storrs, CT, USA, August 2005.

[25] C.T. Yu and C.C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.

A Proofs of theorems

A.1 Maximum number of third parties

Theorem 4.1. *Given two relations $R_l(A_{l_1}, A_{l_2}, \ldots, A_{l_n}, A_{l_{n+1}})$ and $R_r(A_{r_1}, A_{r_2}, \ldots, A_{r_n}, A_{r_{n+1}})$, the maximum number of parties necessary to evaluate an equi-join $R_l \bowtie_{J_{lr}} R_r$, with $J_{lr} = \bigwedge_{i=1}^{n} \langle A_{l_i}, A_{r_i}\rangle$, is n.*

PROOF: Suppose that R_l is stored at S_l, R_r is stored at S_r, and that S_l and S_r are characterized by the following authorizations.

- For $i = 1, \ldots, n$: $[\{\bigcup_{j=1}^{i} A_{l_j}, A_{r_j}\}, \{\bigcup_{j=1}^{i} \langle A_{l_j}, A_{r_j}\rangle\}] \rightarrow S_l$
- For $i = 1, \ldots, n$: $[\{\bigcup_{j=1}^{i} A_{l_j}, A_{r_j}\}, \{\bigcup_{j=1}^{i} \langle A_{l_j}, A_{r_j}\rangle\}] \rightarrow S_r$

These authorizations state that servers S_l and S_r are authorized to access the relations they store and each of the attributes in the other relation, but only if joined with their relation.

Let $S_1, \ldots, S_{n+1} \in \mathcal{S}$ be a set of servers in the system characterized by the following authorizations.

- For $i = 1, \ldots, (n+1)$: $[\{\bigcup_{j=1}^{i} A_{l_j}\}, \{\bigcup_{j=1}^{i-1} \langle A_{l_j}, A_{r_j}\rangle\}] \rightarrow S_i$ and $[\{\bigcup_{j=1}^{i} A_{r_j}\}, \{\bigcup_{j=1}^{i-1} \langle A_{l_j}, A_{r_j}\rangle\}] \rightarrow S_i$

These authorizations state that each S_i is authorized to access the i-th attribute in R_l and in R_r, projected from the join between R_l and R_r evaluated on $\bigwedge_{j=1}^{i-1} \langle A_{l_j}, A_{r_j}\rangle$. Note that S_{n+1} only is authorized to access the table resulting from the join between R_l and R_r, on condition $\bigwedge_{i=1}^{n} \langle A_{l_i}, A_{r_i}\rangle$.

The equi-join between R_l and R_r on J_{lr} cannot be directly evaluated by S_l and S_r. However, it can be safely evaluated by resorting to $S_1, \ldots, S_{n+1}$. First, server S_1 acts as coordinator for the join operation, receiving from S_l and S_r the relations $\pi_{A_{l_1}}(R_l)$ and $\pi_{A_{r_1}}(R_r)$, respectively. Server S_1 then computes the join $J_1 = \pi_{A_{l_1}}(R_l) \bowtie_{A_{l_1}=A_{r_1}} \pi_{A_{r_1}}(R_r)$ between the received relations and sends the result back to both S_l and S_r. Then, S_l computes the join $R_{l_1} = R_l \bowtie J_1$ and S_r computes the join $R_{r_1} = R_r \bowtie J_1$, which are used in the following step of computation in place of R_l and R_r, respectively. At the i-th step (for $i = 1, \ldots, (n-1)$) of the computation, server S_i acts as coordinator for the join operation, receiving from S_l and S_r the projection of attribute A_{l_i} and A_{r_i}, respectively, on the partial result computed at step $i-1$ (i.e., $R_{l_{i-1}}$ and $R_{r_{i-1}}$). Server S_i then computes the join between the received relations and sends the result back to both S_l and S_r, which compute the join between the data received from S_i and the relation obtained by the previous $i-1$ computation steps. Finally, server S_{n+1} receives from both S_r and S_l their partial results computed at step n, and computes the final result of the equi-join operation between R_l and R_r on condition $\bigwedge_{i=1}^{n} \langle A_{l_i}, A_{r_i}\rangle$. □

A.2 Correctness of Algorithm 5.1

Before proving the correctness of Algorithm 5.1 (Theorem 5.1), we introduce three lemmas that will be used in the proof of the main theorem.

Lemma A.1 establishes the correctness of the relation profiles associated with the nodes in a query tree plan and computed by the algorithm.

Lemma A.1. *Given a query tree plan $T(N,E)$, a set $\mathcal{A}$ of authorizations, and a set $\mathcal{S}$ of servers, Algorithm 5.1 associates with each node $n \in N$ a relation profile that correctly reflects the information content of the relation resulting from the evaluation of the operator at node n.*

PROOF: The prove is by induction on the nodes of T.

Base case. If n is a leaf node, the corresponding profile $[n.\pi, n.\bowtie, n.\sigma]$ is initialized to the value $[R, \emptyset, \emptyset]$, where R is the relation represented by n. Therefore, the profile correctly represents the implied information release.

Induction. Consider an internal node n and suppose, by induction, that both the profile $[l.\pi, l.\bowtie, l.\sigma]$ of the left child and the profile $[r.\pi, r.\bowtie, r.\sigma]$ of the right child (if any) correctly represent the information content of the left R_l operand and of the right R_r operand of *n.operator*, respectively. The profile of node n is always computed by modifying the profiles of their children according to the semantics of the relational operator as illustrated in Figure 4. It is then easy to see that by hypothesis the profiles of the children of n are correct, it is also correct the profile of n. □

Lemma A.2 shows that for each node n in a query tree plan, the servers appearing in *n.candidates* can access the relation resulting from the execution of the operation that the node represents.

Lemma A.2. *Let $T(N,E)$ be a query tree plan and $\mathcal{S}$ be the set of servers in the system. At the end of the execution of* **Find_candidates**, *n.candidates contains all and only the servers authorized to access the relation resulting from the execution of the operation represented by n, for each $n \in N$.*

PROOF: The prove is by induction on the nodes of T.

Base case. If n is a leaf node, *n.candidates* is initialized to $[S, _, 0]$, where S is the server storing the base relation R represented by the leaf node. Therefore, S is authorized to access R.

Induction. Consider an internal node n and suppose, by induction, that *l.candidates* contains only servers authorized to access the left operand R_l, and that *r.candidates* contains only servers authorized to access the right operand R_r. We prove that also *n.candidates* contains only servers authorized to access the relation R_n

resulting from the evaluation of the operation represented by n. To this purpose, we consider three cases, depending on $n.operator$.

- $n.operator = \pi$. Each candidate server S appearing in $l.candidates$ is inserted in $n.candidates$. S appears in $l.candidates$ because there is an authorization $[A, J] \rightarrow S$ such that $(l.\pi \cup l.\sigma) \subseteq A$ and $l.\bowtie = J$. Since $n.\pi \subseteq l.\pi$, $n.\sigma = l.\sigma$, and $n.\bowtie = l.\bowtie$, then $(n.\pi \cup n.\sigma) \subseteq A$ and $n.\bowtie = J$. Therefore, each S in $n.candidates$ can access R_n.

- $n.operator = \sigma$. Each candidate server S appearing in $l.candidates$ is inserted in $n.candidates$. S appears in $l.candidates$ because there is an authorization $[A, J] \rightarrow S$ such that $(l.\pi \cup l.\sigma) \subseteq A$ and $l.\bowtie = J$. Since $n.\pi = l.\pi$, $n.\sigma \subseteq l.\pi \cup l.\sigma$, and $n.\bowtie = l.\bowtie$, then $(n.\pi \cup n.\sigma) \subseteq A$ and $n.\bowtie = J$. Therefore, each S in $n.candidates$ can access R_n.

- $n.operator = \bowtie$. A server S in $\mathcal{S}$ is inserted in $n.candidates$ by both procedure **Find_candidates** and function **Find_third_party**. This may happen due to the following different cases that the procedure/function may verify.

  1. S appears in *rightmasters* and is inserted in $n.candidates$ by procedure **Find_candidates**. The list *rightmasters* of candidates is filled in with a **for** loop iterating on the servers appearing in $r.candidates$. A server S is inserted in *rightmasters* only if it has the *right_master_view*. By hypothesis, S is authorized to access R_r. Also, since profiles are correctly computed (Lemma A.1) and S is authorized for the *right_master_view*, S can also access the tuples in R_l participating in the join between R_l and R_r (i.e., $\pi_{n.parameter}(R_r) \bowtie_{n.parameter} R_l$). As a consequence, S is also authorized to access R_n.

  2. S appears in *regular* and is inserted in $n.candidates$ by procedure **Find_candidates**. The list *regular* of candidates is filled in with a **for** loop scanning the servers appearing in $r.candidates$. A server S is inserted in *regular* only if it has the *right_full_view*. By hypothesis, S is authorized to access R_r. Also, since profiles are correctly computed (Lemma A.1) and S is authorized for the *right_full_view*, S can also access the whole content of R_l. As a consequence, S is also authorized to access R_n.

  3. S appears in *leftmasters* and is inserted in $n.candidates$ by procedure **Find_candidates**. Symmetric to Case 1.

  4. S appears in *regular* and is inserted in $n.candidates$ by procedure **Find_candidates**. Symmetric to Case 2.

  5. S appears in *leftmasters* and is inserted in $n.candidates$ by procedure **Find_third_party**. Since list *leftmasters* is filled in by procedure **Find_candidates** and is not updated by function **Find_third_party**, this case is analogous to Case 3.

6. S appears in *rightmasters* and is inserted in *n.candidates* by procedure **Find_third_party**. Since list *rightmasters* is filled in by procedure **Find_candidates** and is not updated by function **Find_third_party**, this case is analogous to Case 1.

7. S is inserted in *n.candidates* by procedure **Find_third_party** and is authorized both for the *right_master_view* and *left_full_view*. Since profiles are correctly computed (see Lemma A.1), and S is authorized for the *right_master_view*, S can access the tuples in R_l participating in the join with R_r. Since S is also authorized for the *right_master_view*, S can access the whole content of R_l. As a consequence, S is also authorized to access R_n.

8. S is inserted in *n.candidates* by procedure **Find_third_party** and is authorized both for the *left_master_view* and *right_full_view*. This case is symmetric to Case 7.

9. S is inserted in *n.candidates* by procedure **Find_third_party** and is authorized both for the *left_full_view* and the *right_full_view*. Since profiles are correctly computed (see Lemma A.1), and S is authorized for both the *left_master_view* and the *right_master_view*, S can access the whole content of both R_l and R_r. As a consequence, S is also authorized to access R_n.

10. S appears in *masterlist* and is inserted in *n.candidates* by procedure **Find_third_party**. The list *masterlist* of candidates is filled in with a **for** loop iterating on the servers appearing in $\mathcal{S}\setminus(l.candidates \cup r.candidates)$. A server S is inserted in *masterlist* only if it is authorized for the *left_slave_view*, the *right_slave_view*, the *left_master_view*, and the *right_master_view*. Since profiles are correctly computed (Lemma A.1) and S is authorized for the *left_master_view*, S can access the tuples in R_r participating in the join between R_l and R_r. Analogously, since S is authorized for the *right_master_view*, S can access the tuples in R_l participating in the join between R_l and R_r. Therefore, S is also authorized for $R_n = R_l \bowtie_{n.parameter} R_r$. □

Lemma A.3 shows that for each node n in a query tree plan, list *n.candidates* is finite and does not contain duplicate entries (i.e., a server appears at most once in the list).

Lemma A.3. *Let $T(N,E)$ be a query tree plan and $\mathcal{S}$ be the set of servers in the system. At the end of the execution of* **Find_candidates**, *for each $n \in N$ n.candidates does not contain duplicate entries (i.e., each $S \in \mathcal{S}$ is inserted at most once in n.candidates).*

PROOF: The prove is by induction on the nodes of T.

Base case. If n is a leaf node, then $l=r=$NULL. Also, *n.candidates* is initialized to $[S,_,0]$, where S is the server storing the relation represented by n. Since *n.operator* is NULL, then *n.candidates* is composed of one element only and therefore cannot include duplicate entries.

Induction. Consider an internal node n and suppose that, by induction, both l.*candidates* and r.*candidates* contain at most one element for each $S \in \mathcal{S}$. We prove that also n.*candidates* contains at most one element for each $S \in \mathcal{S}$. To this purpose, we consider three cases depending on the value of n.*operator*.

- n.*operator*=π. n.*candidates* contains exactly the same elements as l.*candidates*. Therefore n.*candidates* contains at most one element for each $S \in \mathcal{S}$.
- n.*operator*=σ. n.*candidates* contains exactly the same elements as l.*candidates*. Therefore n.*candidates* contains at most one element for each $S \in \mathcal{S}$.
- n.*operator*=$\bowtie$. Procedure **Find_candidates** first inserts in n.*candidates* either the elements in *rightmasters* or the elements in *regular*. However, both *rightmasters* and *regular* contain only servers in r.*candidates*, which are inserted in *rightmasters* and/or in *regular* depending on their ability of acting as masters in a regular and/or semi-join operation. Then, procedure **Find_candidates** proceeds in the same way with the servers in l.*candidates*, which are inserted in *leftmasters* and/or in *regular* depending on their ability of acting as masters in a regular and/or semi-join operation. Either *leftmasters* or *regular* are then inserted in n.*candidates*. Note that both *leftmasters* and *regular* cannot contain a server that already appears in n.*candidates* since in this situation such a server is inserted neither in *leftmasters* nor in *regular*. Since both l.*candidates* and r.*candidates* contain at most one element for each $S \in \mathcal{S}$, either n.*candidates* contains at most one element for each $S \in \mathcal{S}$, or it is empty. In the first case, the lemma is proved. In the second case (n.*candidates*=$\emptyset$), procedure **Find_candidates** calls function **Find_third_party**, which returns a set *list* of servers, used to fill in n.*candidates*. Variable *list* may be filled in by extracting one server at time either from *leftmasters* and *rightmasters*, which do not have common candidate servers, or from the set $\mathcal{S} \setminus (l.candidates \cup r.candidates)$ of servers. Therefore, if function **Find_third_party** returns a set *list* that is not empty, then *list* does not contain duplicate entries. □

Theorem 5.1 (Correctness)**.** *Given a query tree plan $T(N,E)$, a set $\mathcal{A}$ of authorizations, and a set $\mathcal{S}$ of servers, Algorithm 5.1:* i) *terminates,* ii) *computes an executor assignment for T that is safe;* iii) *is always able to find a safe executor assignment for T, if T is feasible.*

PROOF: We separately prove the three properties of the algorithm.

i) Termination

The algorithm terminates iff both procedure **Find_candidates** and procedure **Assign_ex** terminate.

Find_candidates. Procedure **Find_candidates** performs a post-order traversal of the tree T representing the query plan. Since each node of the tree is visited only one time, the number of recursive calls to **Find_candidates** is finite and is equal to the number of nodes in the tree. We therefore need to show that the visit of each node (i.e., the computations performed in correspondence of a node after the recursive calls) terminates. The visit consists in executing **for** and **while** loops that iterate on different subsets of servers (i.e., *l.candidates* and *r. candidates*). Since the number of servers in the system is finite and by Lemma A.3, also these subsets of servers are finite. Furthermore, the loops do not add new elements in the considered sets and therefore they terminate. Each visit also calls function **Can_view** and function **Find_third_party**. It is easy to see that these two functions terminate. As a matter of fact, function **Can_view** is composed of a unique **for** loop iterating on *view*(S). Since the set $\mathcal{A}$ of authorizations in the system is finite and since *view*(S) is a subset of $\mathcal{A}$, the function terminates. Also function **Find_third_party** terminates since its **for** and **while** loops iterate on sets of servers for which it is easy to see that they contain a finite number of servers.

Assign_ex. Procedure **Assign_ex** performs a pre-order traversal of the tree T representing the query plan. Since each node of the tree is visited only one time, the number of recursive calls of **Assign_ex** is finite. Procedure **Assign_ex** then terminates if the visit of each node terminates. The procedure is characterized by a search operation over *n.candidates* list. Since the set $\mathcal{S}$ of servers in the system is finite and by Lemma A.3, we also know that *n.candidates* is composed of a finite number of elements. We can conclude that procedure **Assign_ex** always terminates.

ii) Computation of a safe executor assignment

The prove is by contradiction. Suppose that the executor assignment computed by the algorithm is not safe. Let n be the only node of T for which the corresponding assignment is not safe. We now show that the algorithm cannot associate with n an unsafe assignment, thus contradicting the hypothesis.

- $\lambda_T(n)$ = [*chosen.server*,*n.rightslave*]. In this case, *n.rightslave*≠NULL since otherwise the assignment would be [*chosen.server*,NULL]. Server *n.rightslave* is a server S extracted from *r.candidates* by procedure **Find_candidates**(n) only if S is authorized for the *right_slave_view*, that is, if S is authorized to view $\pi_{n.parameter}(R_l)$. This $\lambda_T(n)$ is obtained in two cases.

  1. *chosen.fromchild*=left. *chosen* is an element of *n.candidates*, and *n.candidates* is a list of servers that procedure **Find_candidates**(n) fills in by extracting from *leftmasters* the servers that are authorized for the *left_master_view* (note that *leftmasters* is a subset of the servers in *l.candidates* authorized to access R_l by Lemma A.2). By Lemma A.1 and Lemma A.2, we can then conclude that $\lambda_T(n)$ is

safe since *chosen.server* is authorized for the *left_master_view* and *n.rightslave* is authorized for the *right_slave_view*.

2. *chosen.fromchild*=third_left. *chosen* is an element of *n.candidates*, and *n.candidates* is a list of servers that function **Find_third_party** fills in by extracting from $\mathcal{S}\backslash(l.candidates \cup r.candidates)$ the servers authorized for both the *left_master_view* and the *right_full_view*. By Lemma A.1 and Lemma A.2, $\lambda_T(n)$ is safe since *chosen.server* is authorized for the *left_master_view* and the *right_full_view* and *n.rightslave* is authorized for the *right_slave_view*.

- $\lambda_T(n)$ = [*chosen.server*,*n.rightthirdslave*]. In this case, *chosen.fromchild*=left and *n.rightthirdslave*≠NULL since otherwise the assignment would be [*chosen.server*,NULL]. The third party *n.rightthirdslave* working as right slave is determined by function **Find_third_party**. In particular, *n.rightthirdslave* is a server that function **Find_third_party** extracts from $\mathcal{S}\backslash(l.candidates \cup r.candidates)$ only if it is authorized for both the *right_slave_view* and the *left_full_view*. Since *n.rightthirdslave*≠NULL, we know that *n.candidates* has been determined by function **Find_third_party** and that *leftmasters*≠NULL. Variable *chosen* then represents a server extracted from *leftmasters* (note that *leftmasters* is a subset of *l.candidates*) that contains servers authorized for the *left_master_view*. By Lemma A.1 and Lemma A.2, $\lambda_T(n)$ is safe since *chosen.server* is authorized for the *left_master_view* and *n.rightthirdslave* is authorized for the *right_slave_view* and for the *left_full_view*.

- $\lambda_T(n)$ = [*chosen.server*,*n.leftslave*] is symmetric to the case $\lambda_T(n)$ = [*chosen.server*,*n.rightslave*].

- $\lambda_T(n)$ = [*chosen.server*,*n.leftthirdslave*] is symmetric to the case $\lambda_T(n)$ = [*chosen.server*,*n.rightthirdslave*].

- $\lambda_T(n)$ = [*chosen.server*,NULL]. Procedure **Assign_ex**(n) assigns NULL to variable *n.executor.slave* in four cases.

1. *chosen.fromchild*=left and both *n.leftslave* and *n.leftthirdslave* are NULL. *chosen* is a server inserted in *n.candidates* by procedure **Find_candidates**(n). In particular, if *n.operator*=π or *n.operator*=σ, *chosen* is also a server that appears in *l.candidates* and, by Lemma A.2, it can access the whole content of both R_l and R_n. This implies that $\lambda_T(n)$ is a safe assignment. Otherwise, if *n.operator*=$\bowtie$, *chosen* is a server that procedure **Find_candidates**(n) has inserted in *n.candidates* when it has added to *n.candidates* all servers in *regular*. Set *regular* is a subset of *l.candidates* and contains all servers that are authorized for the *left_full_view*. By Lemma A.1 and Lemma A.2 we can then conclude that $\lambda_T(n)$ is a safe assignment since *chosen.server* is authorized for the *left_full_view*.

2. *chosen.fromchild*=right is symmetric to the case *chosen.fromchild*=left when *n.operator*=$\bowtie$.

3. *chosen.fromchild*=left_right. Variable *chosen* is inserted in *n.candidates* by procedure **Find_candidates**(n) and *chosen.fromchild* is set to left_right when *chosen.server* appears both in *l.candidates* and in *r.candidates*. It is then easy to see that *chosen.server* can access both R_l and R_r and, consequently it can compute the join operation. We can then conclude that $\lambda_T(n)$ is a safe assignment.

4. *chosen.fromchild*=third. In this case, *chosen* is inserted in *n.candidates* by function **Find_third_party** only if the server is authorized for both the *left_full_view* and the *right_full_view*. By Lemma A.1 and Lemma A.2, we can conclude that $\lambda_T(n)$ is a safe assignment.

- $\lambda_T(n)$ = [*chosen.server*,{*n.leftslave*,*n.rightslave*}]. In this case, since the pair {*n.leftslave*,*n.rightslave*} has been assigned to *n.executor.slave*, *chosen.fromchild* must be equal to "third" and *chosen* has been inserted in *n.candidates* by function **Find_third_party**. More precisely, *chosen* is a server that belongs to list *masterlist* and therefore is authorized for the *left_slave_view*, the *right_slave_view*, the *left_master_view*, and the *right_master_view*. If *n.rightslave* is not NULL when calling function **Find_third_party**, *n.rightslave* is a server extracted from *r.candidates* by procedure **Find_candidates**(n) and that is authorized for the *right_slave_view*. Otherwise, *n.rightslave* is a server extracted from *r.candidates* by function **Find_third_party** and that is authorized for the *two_slave_view*. We note here that a server authorized for the *right_slave_view* is also authorized for the *two_slave_view*. Analogously, *n.leftslave* is either authorized for the *left_slave_view* or the *two_slave_view*. By Lemma A.1 and Lemma A.2, we can conclude that $\lambda_T(n)$ is a safe assignment since servers *chosen.server*, *n.leftslave*, and *n.rightslave* are correctly authorized.

Since any possible assignment that the algorithm can compute for n is safe, we obtain a contradiction.

iii) Determination of a safe assignment if T is feasible

The prove is by contradiction. Suppose that T is feasible and that the algorithm does not find a safe executor assignment. Let n be the only node of T for which the algorithm does not find a safe assignment. For simplicity and without loss of generality, suppose that there exists exactly one safe executor assignment for each node n in the tree (i.e., there is exactly one safe executor assignment for T).

The algorithm cannot determine a safe assignment for n only if *n.candidates* remains empty since otherwise procedure **Assign_ex** would be able to compute a safe assignment. If *n.candidates* is empty, the algorithm terminates after the call to **Find_candidates**(n). However, *l.candidates* and *r.candidates* cannot be empty, since otherwise the algorithm terminates after the call to **Find_candidates**(l) or after the call to

Find_candidates(r). Consider now the different cases that may cause *n.candidates* to remain empty. We now prove that if *n.candidates* remains empty, there is not a safe assignment for n, thus contradicting the hypothesis.

- If *n.operator*=π or *n.operator*=σ, *n.candidates* is empty only if *l.candidates* is empty, thus obtaining a contradiction.
- If *n.operator*=$\bowtie$, *n.candidates* remains empty when the following cases happen.
  1. Procedure **Find_candidates** tries to add *rightmasters* to *n.candidates*. If *n.candidates* remains empty, this happens because either *rightmasters* (i.e., the servers in *r.candidates* authorized for the *right_master_view*) is empty or *n.leftslave* (i.e., the servers in *l.candidates* authorized for the *left_slave_view*) is NULL. We can conclude that there is not a safe assignment $[S_r, S_l]$ for n.
  2. Procedure **Find_candidates** tries to add *regular* to *n.candidates*. If *n.candidates* remains empty, this happens because also *regular*, which should contain the servers in *r.candidates* authorized for the *right_full_view*, remains empty. We can conclude that there is not a safe assignment $[S_r, \text{NULL}]$ for n.
  3. Procedure **Find_candidates** tries to add *leftmasters* to *n.candidates*. If *n.candidates* remains empty, this happens because either *leftmasters* (i.e., the servers in *l.candidates* authorized for the *left_master_view*) is empty or *n.rightslave* (i.e., the servers in *r.candidates* authorized for the *right_slave_view*) is NULL. We can conclude that there is not a safe assignment $[S_l, S_r]$ for n.
  4. Procedure **Find_candidates** tries to add *regular* to *n.candidates*. If *n.candidates* remains empty, this happens because also *regular*, which should contain the servers in *l.candidates* authorized for the *left_full_view*, remains empty. We can conclude that there is not a safe assignment $[S_l, \text{NULL}]$ for n.
  5. If *n.candidates* is still empty, procedure **Find_candidates** calls function **Find_third_party**. Function **Find_third_party** tries to find possible candidates for n by considering the following cases in the order with which we discuss them.
     (a) Function **Find_third_party** first checks whether a third party can act as right slave. If however *list* remains empty, this happens because either *leftmasters* is empty or *n.rightthirdslave* is NULL. Note that *leftmasters* is computed by procedure **Find_candidates** and should contain all servers in *l.candidates* authorized for the *left_master_view*. Therefore, if *leftmasters* is empty, we can immediately conclude that there is not a safe assignment $[S_l, S_t]$ for n. Otherwise, function **Find_third_party** tries to determine server *n.rightthirdslave* by checking all servers in $\mathcal{S}\backslash(l.candidates \cup r.candidates)$ authorized for the *right_slave_view* and the *left_full_view*. Note that it is correct to exclude from this evaluation servers in *r.candidates* since they have been

already evaluated for the *right_slave_view* by procedure **Find_candidates** without any positive result. Analogously, servers in l.*candidates* can be excluded since they have been already evaluated for the *left_full_view* by procedure **Find_candidates** without any positive result. Therefore, if n.*rightthirdslave* remains NULL, we can conclude that there is not a safe assignment $[S_l, S_t]$ for n.

(b) Function **Find_third_party** checks whether a third party can act as left slave. This case is symmetric to Case 5a. We can then conclude that if *list* remains empty, there is not a safe assignment $[S_r, S_t]$ for n.

(c) Function **Find_third_party** proceeds by checking whether a third party can act as right master. To this purpose, the function checks whether there are servers in $\mathcal{S}\setminus(l.candidates \cup r.candidates)$ that are authorized for the *right_master_view* and the *left_full_view*. Note that it is correct to exclude from this evaluation servers in r.*candidates* since they have already been evaluated for the *right_master_view* by procedure **Find_candidates** without any positive result. Analogously, servers in l.*candidates* can be excluded since they have been already evaluated for the *left_full_view* by procedure **Find_candidates** without any positive result. Therefore, if *list* remains empty, there is not a safe assignment $[S_t, S_l]$ for n.

(d) Function **Find_third_party** proceeds by checking whether a third party can act as left master. This case is symmetric to Case 5c. We can conclude that if *list* remains empty, there is not a safe assignment $[S_t, S_r]$ for n.

(e) Function **Find_third_party** proceeds by checking whether a third party can execute a regular join. To this purpose, the function checks whether there are servers $\mathcal{S}\setminus(l.candidates \cup r.candidates)$ authorized for the *left_full_view* and the *right_full_view*. Note that again it is correct to exclude from this evaluation servers in r.*candidates* since they have been already evaluated for the *right_full_view* by procedure **Find_candidates** without any positive result. Analogously, servers in l.*candidates* can be excluded since they have been already evaluated for the *left_full_view* by procedure **Find_candidates** without any positive result. Therefore, if *list* remains empty, there is not a safe assignment $[S_t, \text{NULL}]$ for n.

(f) As last attempt, function **Find_third_party** checks whether there is a third party that can act as coordinator. In this case, if *list* remains empty, then either n.*leftslave*, n.*rightslave*, or *masterlist* are NULL. n.*rightslave* is determined by extracting from r.*candidates* a server authorized for the *two_slave_view*. n.*leftslave* is determined by extracting from l.*candidates* a server authorized for the *two_slave_view*. *masterlist* is computed extracting from $\mathcal{S}\setminus(l.candidates \cup r.candidates)$

all servers authorized for the *left_slave_view*, the *right_slave_view*, the *left_master_view*, and the *right_master_view*. Note that it is correct to exclude from this evaluation servers in l.*candidates* and in r.*candidates* since they have been already evaluated for at least one of these views without positive results. Therefore, *list* remains empty and there is not a safe assignment $[S_t, S_l S_r]$ for n.

Since there are no other possible safe assignments for n, we can conclude that n.*candidates* remains empty only if there is not a safe assignment for n, thus obtaining a contradiction. □

A.3 Complexity analysis

Theorem 5.2 (Complexity)**.** *Given a query tree plan $T(N,E)$, a set $\mathcal{A}$ of authorizations, and a set $\mathcal{S}$ of servers, the time complexity of Algorithm 5.1 is $O(|N| \cdot |\mathcal{S}| \cdot |\mathcal{A}|)$.*

PROOF: The complexity in time of the algorithm is the sum of the complexities of procedures **Find_candidates** and **Assign_ex**.

Find_candidates. Procedure **Find_candidates** performs a post-order traversal of T and therefore each node of the tree is visited only one time. The complexity of the procedure is $|N|$ times the complexity of each visit. The visit of a node in T consists in executing **for** and **while** loops that iterate on sets l.*candidates* and r.*candidates* and that contain calls to function **Can_view**. By Lemma A.3, the number of elements in both l.*candidates* and r.*candidates* is $O(|\mathcal{S}|)$. Function **Can_view** is composed of a unique **for** loop iterating on *view*$(S) \subseteq \mathcal{A}$. The complexity in time of **Can_view** is then $O(|\mathcal{A}|)$. Procedure **Find_candidates** also calls function **Find_third_party**, which is characterized by both **for** and **while** loops iterating on l.*candidates*, r.*candidates*, *leftmasters*, *rightmasters*, *cand_slave*, and *masterlist* lists, that contain calls to function **Can_view**. By Lemma A.3, we know that the number of elements in both l.*candidates* and r.*candidates* is $O(|\mathcal{S}|)$. Also, since *leftmasters* $\subseteq$ l.*candidates* and *rightmasters* $\subseteq$ r.*candidates*, the number of servers in *leftmasters* and in *rightmasters* is $O(|\mathcal{S}|)$. Analogously, *cand_slave* is initialized to $\mathcal{S} \setminus (l.candidates \cup r.candidates)$ and *masterlist* $\subseteq$ $\mathcal{S} \setminus (l.candidates \cup r.candidates)$ and then also the number of servers in *cand_slave* and *masterlist* is $O(|\mathcal{S}|)$. The complexity in time of function **Find_third_party** is $O(|\mathcal{S}| \cdot |\mathcal{A}|)$, where $O(|\mathcal{A}|)$ is the complexity of **Can_view**. Since in the worst case function **Find_third_party** is called for each node n in T, we can conclude that the time complexity of procedure **Find_candidates** is $O(|N| \cdot |\mathcal{S}| \cdot |\mathcal{A}|)$.

Assign_ex. Procedure **Assign_ex** performs a pre-order traversal of T and therefore each node of the tree is visited only one time. The complexity of the procedure is $|N|$ times the complexity of each visit. The procedure

is characterized by a search of *from_parent* in *n.candidates*. The time complexity of this search is $O(|\mathcal{S}|)$, since by Lemma A.3, *n.candidates* includes $O(|\mathcal{S}|)$ servers. The time complexity of procedure **Assign_ex** is then $O(|N| \cdot |\mathcal{S}|)$.

We can conclude that the overall complexity of Algorithm 5.1 is $O(|N| \cdot |\mathcal{S}| \cdot |\mathcal{A}|)$ in time. □