Controlled Information Sharing in Collaborative Distributed Query Processing*

Sabrina De Capitani di Vimercati
DTI - Università di Milano
26013 Crema - Italy
decapita@dti.unimi.it

Sara Foresti
DTI - Università di Milano
26013 Crema - Italy
foresti@dti.unimi.it

Sushil Jajodia
CSIS - George Mason University
Fairfax, VA 22030-4444
jajodia@gmu.edu

Stefano Paraboschi
DIIMM - Università di Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Pierangela Samarati
DTI - Università di Milano
26013 Crema - Italy
samarati@dti.unimi.it

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 the result. We also present an algorithm that, given a query plan, determines whether it can be safely executed and produces a safe execution strategy. The main advantage of our approach is its simplicity that, without impacting expressiveness, makes it nicely interoperable with current solutions for collaborative computations in distributed database systems.

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 larger 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 the 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 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 coherent with current mechanisms for the management of distributed computations.To this aim and for the sake of concreteness, in this paper we address the problem with specific consideration to distributed database systems. This must not be considered a limitation as relational databases are the core of any Web service.

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 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 access profiles for the users in the system, with explicit support for a cooperative management of queries. This is an important feature in distributed settings,

*This work was supported in part by the EU, within the 7FP project "PrimeLife" and by the Italian MIUR, within PRIN 2006, under project 2006099978. The work of Sushil Jajodia was partially supported by NSF under grants CT-0716567, CT-0627493, IIS-0242237, and IIS-0430402 and by the Army Research Office under grant W911NF-07-1-0383.

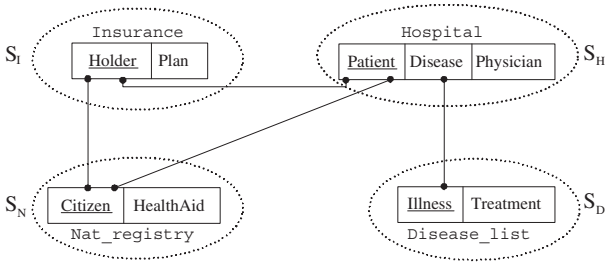

Figure 1. Schema of a distributed system

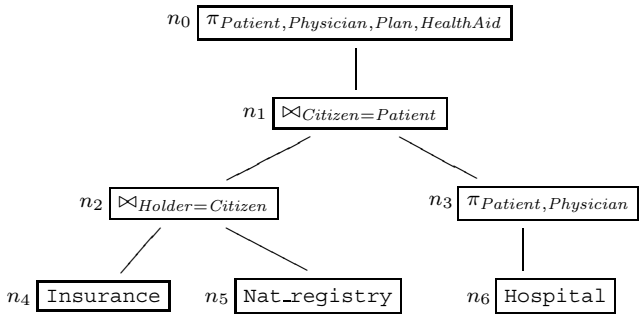

Figure 2. An example of query tree plan

where the minimization of data exchanges and the execution of steps of the queries in locations where it can be less costly, is a crucial factor in the identification of an execution strategy characterized by good performance.

2. Preliminary concepts

We consider a distributed system composed of different servers, storing different relations. Each relation is characterized 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. 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 as relations/attributes with the same name can be made distinct by 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 one 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) *Given a set of relations $R_1,\ldots,R_{n+1}$ and a sequence of n join operations on them, we define a* join path *as a set of pairs $\langle J_{l_1}, J_{r_1} \rangle, \ldots, \langle J_{l_n}, J_{r_n} \rangle$, where $\langle J_{l_i}, J_{r_i} \rangle$ are the equi-join conditions of the i-th join.*

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* $\{\langle$Holder,Patient$\rangle, \langle$Disease,Illness$\rangle\}$*, 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 algebra expression $\pi_A(\sigma_C(R_1 \bowtie_{JC_1} \ldots \bowtie_{JC_n} R_{n+1}))$, where A is a set of attributes, C is the selection conditions, and $R_1 \bowtie_{JC_1} \ldots \bowtie_{JC_n} R_{n+1}$ are the joins in the FROM clause. Each query execution can be represented as a binary tree (called query tree plan) where leaves correspond to the physical relations accessed by the query (appearing in the FROM clause), each non-leaf node is a relational 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. To simplify 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. While usually adopted for efficiency, this assumption is also important for security purposes, as it discloses only the attributes needed for the computation.

Example 2.2 *Consider query*

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

corresponding to relational algebra expression $\pi_{Patient,Physician,Plan,HealthAid}$ (`Insurance` $\bowtie_{Holder=Citizen}$ `Nat_registry` $\bowtie_{Citizen=Patient}$

	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	{Illness, Treatment}	_	S_D

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

`Hospital`). *Figure 2 illustrates a tree representing the query execution, where the projection on* Patient *and* Physician *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 independently to the relation or to the server storing it.

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]

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 relations with attributes in* Attributes*, i.e., 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 is that *Server* can be released (i.e., is authorized to view) the set of *Attributes* for which the join operations of the involved relations satisfy the conditions given in the *Join Path*.

Note how the simple form of authorizations above, with the specification of the join path as a separate element, proves quite expressive. In particular, the join path may also include relations that do not have any attribute appearing in the set *Attributes*. This may be due to either:

- *connectivity constraints*, where these relations are needed to build a correct association among the attributes of other relations (i.e., the relations are in the join path). For instance, in authorization 3 in Figure 3 `Hospital` appears in the join path to establish the association between insurance holders and their treatments, but none of its attributes 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.
- *instance-based restrictions*, where the 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 5 in Figure 3 allows server S_H to view the insurance plans of all the patients of the Hospital (i.e., tuples in `Insurance` satisfying *Patient=Holder* condition) but not of those insurance holders who are not treated at the Hospital. 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, providing the patients' SSN, the hospital can retrieve the plan.

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

[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 [17].

fact that the tuple satisfies the join path, i.e., that its values have an association with another relation (possibly not 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.

3.2. Profiles and authorized views

Authorizations restrict the data (view) 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 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.

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) $R^\bowtie = J$.

[2] For the sake of simplicity, with a slight abuse of notation, in the table 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*.

Operation	Profile R^π	Profile $R^\bowtie$	Profile 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 operations

According to the definition above, 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 clearly an authorization to view a superset of attributes implies the authorization to view a subset of them. Note that a similar implication (an authorization containing a join path authorizing all relations that include the path) 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 authorization for the attributes (authorization 15) 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 should 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 15 allowing it to view relation `Disease_list`, also the authorization to view relation `Hospital`. The two authorizations clearly imply the authorization for the query above, which represents a view on them. We assume authorizations are closed with respect to such derivations by means of a "chase" procedure [2] that derives all the authorizations implied directly or indirectly by those explicitly specified. For space reasons, we do not further discuss such a process.

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: $R_{J_lr} := R_{J_l} \bowtie_{J_{lr}} R_r$ S_r: $R_{J_lr} \rightarrow S_l$ S_l: $R_{J_lr} \bowtie R_l$	$\pi_{J_l}(R_l) \bowtie_{J_{lr}} R_r$	$\pi_{J_l}(R_l)$	$[J_l, R_l^{\bowtie}, R_l^{\sigma}]$ $[J_l \cup R_r^{\pi}, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$
	[S_r,S_l]	S_r: $R_{J_r} := \pi_{J_r}(R_r)$ S_r: $R_{J_r} \rightarrow S_l$ S_l: $R_{lJ_r} := R_l \bowtie_{J_{lr}} R_{J_r}$ S_l: $R_{lJ_r} \rightarrow S_r$ S_r: $R_{lJ_r} \bowtie R_r$	$\pi_{J_r}(R_r)$	$R_l \bowtie_{J_{lr}} (\pi_{J_r}(R_r))$	$[J_r, R_r^{\bowtie}, R_r^{\sigma}]$ $[R_l^{\pi} \cup J_r, R_l^{\bowtie} \cup R_r^{\bowtie} \cup J_{lr}, R_l^{\sigma} \cup R_r^{\sigma}]$

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

4. Safe query planning

To determine whether and how an operation can be executed, we need first to determine the data releases that the execution 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 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. For join operations, we first 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 resulting from whether the join is executed as a *regular* join or as a *semi-join* and from which operand serves as master (slave, respectively). The assignment is specified as a pair, where the first element is the operand that serves as master and the second the operand that serves as slave. We discuss the cases where the left operand serves as master (denoted [S_l,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,NULL] and [S_r,S_l]) are symmetric.

- [S_l,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 longer 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 are usually more efficient than regular joins as they minimize communication, which also benefits security: the slave server needs only to send those tuples that participate in the join, instead of its complete relation.

For instance, consider the query in Example 2.2. If the join at node n_2 in the tree 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). If the join is executed as a semi-join where S_I acts as a 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.

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, the definition of the *executor assignment* function λ_T is introduced 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* $[S, \text{NULL}]$*, where S is the server where R is stored;*

2. *each non-leaf node n, corresponding to unary operation* op *on operands 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 operand 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* $\text{master} \in \{S_l, S_r\}$, $\text{slave} \in \{S_l, S_r, \text{NULL}\}$, *and* master≠slave.

Given a query plan, our algorithm determines 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.

Definition 4.2 (Safe assignment) *Given a query tree plan T(N,E) 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.*
λ_T *is said to be* safe *iff* $\forall n \in N$, $\lambda_T(n)$ *is safe.*

A query plan is then feasible iff there exists a safe assignment for it.

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.*

We can now state the problem as follows.

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

In the next section we illustrate an algorithm for the solution of such a problem.

5. Algorithm

To minimize the cost of computation, 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 receives in input the set of authorizations and the query plan $T(N, E)$, where each leaf node (base relation R) is already assigned executor [*server*,NULL], where *server* is the server storing the relation. Each node of the tree is associated with a set of variables: *n.left* and *n.right* are the left and right children; *n.operator* and

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

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)
if r ≠NULL then Find_candidates(r)
case n.operator of
 π : n.π := n.parameter; n.⋈ := l.⋈; n.σ := l.σ
     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
     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
         if CanView(left_slave_view, c.server) then n.leftslave := c
         c := c.next
     for c in r.candidates do
       if n.leftslave≠NULL then
         if CanView(right_master_view, c.server) then
           Add [c.server, right, c.count+1] to n.candidates
       else if CanView(right_full_view, c.server) then
           Add [c.server, right, c.count+1] 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
         if CanView(right_slave_view, c.server) then n.rightslave := c
         c := c.next
     for c in l.candidates do
       if n.rightslave≠NULL then
         if CanView(left_master_view, c.server) then
           Add [c.server, left, c.count+1] to n.candidates
       else if CanView(left_full_view, c.server) then
           Add [c.server, left, c.count+1] to n.candidates
     if n.candidates=NULL then exit(n) /* node cannot be executed */

CAN_VIEW(profile,S)
for each [A, J] ∈view(S) do
  if ((profile.π∪profile.σ)⊆ A) AND (profile.⋈=J) then return(TRUE)
return(FALSE)

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
case chosen.fromchild of
  left: n.executor.slave := n.rightslave   /* case [S_l,NULL], [S_l,S_r] */
        if n.left≠NULL then Assign_ex(n.left, n.executor.master)
        if n.right≠NULL then Assign_ex(n.right, n.executor.slave)
  right: n.executor.slave := n.leftslave   /* case [S_r,NULL], [S_r,S_l] */
        if n.left≠NULL then Assign_ex(n.left, n.executor.slave)
        if n.right≠NULL then Assign_ex(n.right, n.executor.master)
```

Figure 6. Pseudo-code of the algorithm

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) it comes from, and the number of joins for which the server is candidate in the subtree; and *n.executor.master* and *n.executor.slave* are the executor assignment; $[n.\pi,n.\bowtie,n.\sigma]$ is the profile of the node.

The algorithm performs two traversals of the tree query plan. The first traversal (procedure **Find_candidates**) visits the tree in *post-order*. At each node, the profile of the node is computed (as in Figure 4) based on the profile of the children and of the operation associated with the node. Also, the set of possible candidate assignments for the node is determined based on the set of possible candidates for its children as follows. If the node is a unary operation, the candidates for the node are all the candidates for its child. If the node is a join operation, procedure **Find_candidates** calls function **CanView** whenever it is necessary to verify if a particular server can act as master, slave, or can calculate a regular join. **CanView** takes as input the profile of the views that should be made visible in the execution of an operation and a server; it returns true if the result is authorized. The algorithm considers candidates of the left child in decreasing order of join counter (**GetFirst**) and stops at the first candidate found that can serve as left slave (inserting it into local variable *leftslave*). The algorithm proceeds examining all the candidates of the right child to determine if they can work as master for a semi-join (if a left slave was found) or as a regular join (if no left slave was found). Note that while we need to determine all servers that can act as master, as we need to consider all possible candidates for propagating them upwards in the tree, it is sufficient to determine one slave (a slave is not propagated upward in the tree). For each of such *server* candidates a triple [*server*,right,*counter*] is added to the *candidates* list, where counter is the counter that was associated with the server in the right child of the node incremented by one (as candidate 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 algorithm 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 then determining all the left candidates that can work as master, adding them to the set of candidates. At the end of this process, list *candidates* contains all the candidates coming from either the left or right child that can execute the join in any of the execution modes of Figure 5. If no candidate was found, the algorithm exits returning the node at which the process was interrupted (i.e., for which no safe assignment exists) signaling that the tree is not feasible.[3]

[3]We note that a safe assignment could exist in case of a third party acting either as a proxy for one of the two operands or as a coordinator for them. Due to space restrictions, we omit the discussion on how to integrate the use of a third party in the query execution.

Find_candidates

Node	Candidates	Slave
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]*	

Figure 7. Algorithm execution

If **Find_candidates** completes successfully, the algorithm proceeds with the second traversal of the query tree plan. The second traversal (procedure **Assign_ex**) recursively visits the tree in *pre-order*. At the root node, if more assignments are possible, the candidate server with the highest join count is chosen. Hence, the chosen candidate is pushed down to the child from which it was derived in the post-order traversal. The other child (if existing) is pushed down the recorded candidate slave. If no slave was recorded as possible (i.e., *rightslave/leftslave*=NULL) a NULL value is pushed down. At each children, 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 count and the process is recursively repeated, until a leaf node is reached.

Example 5.1 *Consider the query plan in Figure 2, and the set of authorizations in Figure 3. Figure 7 illustrates the working of procedures* **Find_candidates** *and* **Assign_ex** *reporting the nodes in the order they are considered by them 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 call* **Find_candidates**(n_2). *Among the candidates of the children (S_N from left child n_4 and S_I from right child n_5) only the left 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 right 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 assigns to n_0 executor [S_H,NULL], where S_H was recorded as coming from the left (and only) child n_1, to which S_H is then pushed with a recursive call. At n_1 the master is set as 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 (from where it was taken by* **Find_candidates**$)$ n_3, *while S_N is pushed down to the left child n_2.*

We conclude this section with a note regarding the integration of our approach with existing query optimizers. Optimization of distributed queries often operates in *two-steps* [12]. First, the query optimizer identifies a good plan; second, it assigns operations to the servers in the system. Our algorithm nicely fits in such a two phase structure.

6. Related work

Previous work is related to classical proposals on the management of queries in centralized and distributed systems [3, 5, 6, 12, 14, 18] that describe how efficient query plans can be obtained, but do not take into account constraints on attribute visibility for servers. Also, a significant amount of research has recently focused on the problem of processing distributed queries under protection requirements [4, 9, 10, 11, 13, 15]. These works are typically based on the definition of an *access pattern* associated with each relation/view that defines how it can be accessed. Here, the main goal is the identification of 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 parameter (e.g., the number of accesses to data sources [4]). While our proposal can be considered a natural extension of the approach normally used to describe database privileges in a relational schema, access patterns describe authorizations as special formulas in a logic programming language for data access. The model we propose is certainly easier to integrate with the mechanisms and approaches that are used by current database servers. Also, our model 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 it. Sovereign joins [1] are an alternative solution based on a secure coprocessor, which is involved in query execution, and exploits cryptography thus bearing a high computational cost. Other related work is represented by approaches merging security specifications coming from different independent data sources (e.g., [7, 8]). Such approaches however mostly focus on the merging of specifications, and possible solving inconsistencies among them, towards the establishment of a common security policy for regulating the federated system.

7. 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.

References

[1] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In *Proc. of ICDE'06*, Atlanta, April 2006.

[2] A. Aho, C. Beeri, and J. Ullman. The theory of joins in relational databases. *ACM TODS*, 4(3):297–314, 1979.

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

[4] A. Calì and D. Martinenghi. Querying data under access limitations. In *Proc. of ICDE'08*, Cancun, April 2008.

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

[6] D. Chiu and Y. Ho. A methodology for interpreting tree queries into optimal semi-join expressions. In *Proc. of SIGMOD 1980*, Santa Monica, CA, May 1980.

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

[8] S. De Capitani di Vimercati and P. Samarati. Authorization specification and enforcement in federated database systems. *Journal of Computer Security*, 5(2):155–188, 1997.

[9] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. In *Proc. of ICDT 2005*, Edinburgh, Scotland, Jan. 2005.

[10] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of SIGMOD 1999*, Philadelphia, PA, June 1999.

[11] G. Gottlob and A. Nash. Data exchange: Computing cores in polynomial time. In *PODS'06*, Chicago, June 2006.

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

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

[14] G. Lohman, D. Daniels, L. Haas, R. Kistler, and P. Selinger. Optimization of nested queries in a distributed relational database. In *Proc. of VLDB 1984*, Singapore, August 1984.

[15] A. Nash and A. Deutsch. Privacy in GLAV information integration. In *Proc. of ICDT 2007*, Barcelona, Jan. 2007.

[16] A. Rosenthal and E. Sciore. Who-or what-do you trust: Nondisclosure policies for distributed computations. Technical report, MITRE, May 2005.

[17] P. Samarati and S. De Capitani di Vimercati. Access control: policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri (eds), LNCS 2171, Springer-Verlag 2001.

[18] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational DBMS. In *Proc. of SIGMOD'79*, Boston, May 1979.