Chapter 5
Distributed Query Processing under Safely Composed Permissions[1]

The integration of information sources detained by distinct parties, either for security or efficiency reasons, is becoming of great interest. A crucial issue in this scenario is the definition of mechanisms for the integration that correctly satisfy the commercial and business policies of the organization owning the data. In this chapter, we propose a new model based on the characterization of access privileges for a set of servers on the components of a relational schema. The proposed approach is based on three concepts: *i)* flexible permissions identify portions of the data being authorized, *ii)* relations are checked for release not with respect to individual authorizations but rather evaluating whether the information release they (directly or indirectly) entail is allowed by the permissions, and *iii)* each basic operation necessary for query evaluation entails different data exchanges among the servers. Access control is effectively modeled and efficiently executed in terms of graph coloring and composition. The query execution plan is checked against privileges to evaluate if it can or cannot be exploited for query evaluation.

5.1 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,

[1] Part of this chapter appeared under 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, October 2008 [42] ©2008 ACM, Inc. Reprinted by permission http://doi.acm.org/10.1145/1455770.1455810; and under ©2008 IEEE, reprinted, with permission, from S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, 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 [43].

where a centrally planned database design is 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 correct definition and management of protection requirements is therefore a crucial point for an effective collaboration and integration of heterogeneous 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 coherent with current mechanisms for the management of distributed computations, to be seamlessly integrated in current systems and fully exploit the availability of technical solutions that are the fruit of a large amount of research and development. To this aim and for the sake of concreteness, in this chapter we address the problem with specific consideration to distributed database systems, while noting that our approach can be extended to other data models.

Current approaches for the specification and enforcement of authorizations in relational databases claim flexibility and expressiveness because of the possibility of referring to views. Users can be given access to a specific portion of the data by the definition of the corresponding view (in the database schema) and the consequent granting of the authorization on the view to the user. It is then responsibility of the user to query the view itself. Queries on a table (base relation or view) are controlled with respect to authorizations specified on the table and granted only if authorized. When the diversity of users and possible views is considerable and dynamic such an approach clearly results limiting as it: *i)* requests to explicitly define a view for each possible access needed and *ii)* imposes on the user/application the burden of knowing and directly querying the view. The evaluation of query compliance in terms of existing authorization views has been considered in [71, 80, 81, 82].

We propose an expressive, flexible, and powerful, yet simple approach for the specification and enforcement of permissions that overcomes such limitations. Our permissions express privileges not on specific existing views but on stable components of the database schema, exploiting both relations and joins between them, effectively identifying the specific portion of the data whose access is being authorized. Another important aspect of our approach is that we do not limit ourselves to a simple relation-authorization control but allow data release whenever the information carried by the relation (either directly or indirectly due to the dependence of the attributes with other data not explicitly released) is legitimate according to the specified permissions. This is an important paradigm shift with respect to current solutions, departing from the need of specifying views to identify the portion of the data to be authorized but explicitly supporting such a specification in the permissions themselves.

A further 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, 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.

5.1.1 Chapter Outline

The remainder of the chapter is organized as follows. Section 5.2 introduces the preliminary concepts of distributed query evaluation, which are referred in our approach. Section 5.3 illustrates our security model. Section 5.4 illustrates a graph-based representation of the components of the proposed authorization model (database schema, relation profiles, and permissions). Section 5.5 describes a safe and efficient permission composition method, exploited for evaluating if a given release is to be authorized or denied. Section 5.6 discusses query planning and how protection requirements stated by permissions should impact its execution to ensure data are properly protected by the distributed computation. Section 5.7 proposes an algorithm for determining whether a query plan can be executed in the respect of the authorizations and determine, if it exists, a safe assignment of tasks to the distributed cooperating parties for the execution of the query plan. Finally, Sect. 5.8 concludes the chapter.

5.2 Preliminary Concepts

We consider a distributed system composed of different subjects, denoted $\mathscr{S}$, some of which act as servers storing different relations, denoted $\mathscr{R}$. In this section, we briefly introduce the basic concepts and assumptions on the data model and the distributed query execution.

5.2.1 Data Model

We refer in this chapter to the relational database model discussed in Sect. 3.2, which is basically composed of a set $\mathscr{R}$ of relations, each with a primary key, and of a set of referential integrity constraints.

Example 5.1. Consider a distributed system managing medical data, whose schema is represented in Fig. 5.1. The system is composed of four servers with one relation each: `Employee` stored at server S_E; `Patient` stored at server S_P; `Treatment` stored at server S_T, and `Doctor`, stored at server S_D. Underlined attributes denote primary keys. There are two referential integrity constraints: $\langle$`Treatment.SSN`,`Patient.SSN`$\rangle$, implying that treatments can only be given

$\mathscr{R}$	EMPLOYEE(SSN,Job,Salary)
	PATIENT(SSN,DoB,Race)
	TREATMENT(SSN,IdDoc,Type,Cost,Duration)
	DOCTOR(IdDoc,Name,Specialty)
$\mathscr{I}$	⟨Treatment.SSN,Patient.SSN⟩
	⟨Treatment.IdDoc,Doctor.IdDoc⟩
$\mathscr{J}$	⟨Employee.SSN,Patient.SSN⟩

Fig. 5.1 An example of relations, referential integrity constraints, and joins

to patients (values appearing for `SSN` in `Treatment` can be only values appearing for `SSN` in `Patient`), and ⟨`Treatment.IdDoc`,`Doctor.IdDoc`⟩, implying that treatments can only be prescribed by doctors (values appearing for `IdDoc` in `Treatment` can be only values appearing for `IdDoc` in `Doctor`).

Information in different relations can be combined by using the join operation, which allows the combination of tuples belonging to different relations imposing conditions on how tuples can be combined. For simplicity of exposition, we assume that attributes that can be joined appear with the same name in the different relations, and consider then all joins to be *natural joins*, that is, joins whose conditions are conjunctions of equality conditions that compare the value of two attributes with the same name. We denote a conjunction of equality conditions with a pair $\langle A_l, A_r \rangle$, where A_l (A_r, resp.) is the list of attributes of the left (right, resp.) operand of the join. Note that while possible joins obviously include all referential integrity constraints, other joins are possible; in the following we denote with $\mathscr{J}$ the set of pairs representing the equality conditions of such additional joins. As an example, with respect to the relations in Fig. 5.1, `Employee` and `Patient` can be joined over attribute `SSN` (retrieving all people that are both employees and patients). Like the set of relations and the referential integrity constraints, possible joins are also specified at the time of database design [49].

We assume all attributes in the different relations to have distinct names, apart from attributes that can be joined, which appear instead with the same name. The intuitive rationale behind such a homonymity is that attributes that can be joined actually represent the same concept of the real world. For instance, `SSN` denotes social security numbers of people, who can then appear, for example, as patients or employees. We adopt the usual dot notation when necessary to distinguish the attribute in a specific relation (to refer to the occurrence of its specific values). For instance, `Employee.SSN` denotes the social security numbers of employees and `Patient.SSN` denotes the social security numbers of patients.

Different join operations can also be used to combine tuples belonging to more than two relations. The following definition introduces a *join path* as a sequence of natural join conditions.

Definition 5.1 (Join path). A *join path* over a sequence of relation schemas $R_1, \ldots, R_n$ is a sequence of $n-1$ joins $J_1, \ldots, J_{n-1}$ such that $\forall i = 1, \ldots, n-1$,

$J_i = \langle J_{li}, J_{ri} \rangle \in (\mathscr{I} \cup \mathscr{J})$ and J_{li} are attributes of a relation appearing in a join J_k, with $k < i$.

Example 5.2. With reference to the relations in Fig. 5.1, an example of join path (combining more than two relations) is, $\{\langle$`Patient.SSN`,`Treatment.SSN`$\rangle$, $\langle$`Treatment.IdDoc`,`Doctor.IdDoc`$\rangle\}$, allowing combination of tuples of the relations `Patient`, `Treatment`, and `Doctor` to retrieve, for example, the specialty of the caring doctor of patients of a given race.

While noting that the permission model we propose in the next section can be applied to any schema, in this chapter we assume that the schema is *acyclic* and *lossless* [1, 5, 9]. Acyclicity implies that the join path over any subset of the relations $\{R_1,\ldots,R_n\}$ in the schema, denoted *joinpath*$(\{R_1,\ldots,R_n\})$, is unique. Acyclicity rules out schemas that present recursion or multiple independent join conditions among the same relations. Acyclicity can be immediately evaluated on the schema graph (see Sect. 5.4), considering arcs without orientation. Losslessness of the schema guarantees that joins among relations produce only correct information (according to the real world). Intuitively, two relations produce a *lossless join* if the join among them does not produce spurious tuples. Losslessness can be evaluated by means of attribute intersections and functional dependencies (see Sect. 5.4). Acyclicity and losslessness assumptions are often used in relational databases, because they permit the realization of simple and efficient procedures on the data, at the same time capturing the requirements of most real-word situations [9].

5.2.2 *Distributed Query Execution*

Since relations are distributed at different servers, query execution may require communication and data exchanges among the different servers involved in the query (i.e., on which the relations to be accessed are stored). We assume that each server implements a relational engine able to compute queries and that it can require the execution of queries to other servers. We assume communication relies on trusted channels and that servers use robust authentication mechanism (e.g., SSL/TLS with 2-way authentication using certificates).

We consider simple *select-from-where* queries of the form: "SELECT A FROM *Joined relations* WHERE C", corresponding to algebra expression $\pi_A(\sigma_C(R_1 \bowtie \ldots \bowtie R_n))$, where A is a set of attributes, C is the selection conditions, and $R_1 \bowtie \ldots \bowtie R_n$ 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, and, in particular, we assume that projections are "pushed down" the tree, to eliminate unnecessary

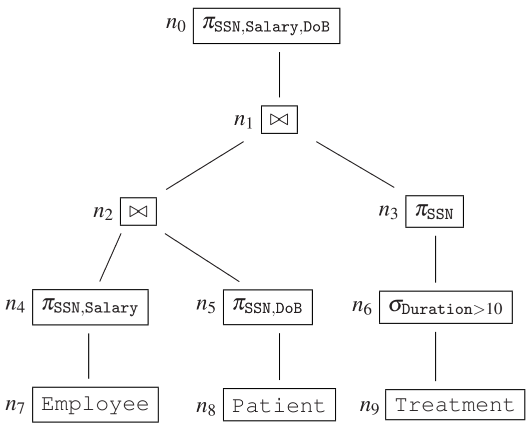

n_0 $\pi_{\texttt{SSN,Salary,DoB}}$
n_1 $\bowtie$
n_2 $\bowtie$
n_3 $\pi_{\texttt{SSN}}$
n_4 $\pi_{\texttt{SSN,Salary}}$
n_5 $\pi_{\texttt{SSN,DoB}}$
n_6 $\sigma_{\texttt{Duration}>10}$
n_7 Employee
n_8 Patient
n_9 Treatment

Fig. 5.2 An example of query tree plan

attributes as soon as possible. While usually adopted for efficiency, this assumption is also important for security purposes, as it restricts the attributes being potentially disclosed to those strictly needed for the computation.

Example 5.3. Consider the relations in Fig. 5.1, and consider the following query.

```
SELECT E.SSN, Salary, DoB
FROM   Employee AS E JOIN Patient AS P ON E.SSN=P.SSN
                     JOIN Treatment AS T ON P.SSN=T.SSN
WHERE Duration> 10
```

The corresponding relational algebra expression is $\pi_{\texttt{SSN,Salary,DoB}}$ $(\sigma_{\texttt{Duration}>10}$ $(\texttt{Employee} \bowtie \texttt{Patient} \bowtie \texttt{Treatment}))$. An example of tree representing the execution of this query is represented in Fig. 5.2, where the selection on Duration$>$ 10 on relation Treatment has been pushed down the tree (i.e., it is executed before the join operation). Also, projections on necessary attributes are added before join operations.

Queries may involve joins among relations stored at different servers, which therefore need to cooperate, and possibly exchange data, for performing the computation. We therefore propose an authorization model to regulate the view that each server (subject in general) can have on the data and ensure that query computation exposes to each server only data that the server can view.

We assume that each server is responsible for the definition of the access policy on its resources and permissions involving data stored at different servers are jointly specified and administered. A centralized query optimizer is responsible for the construction of the query plan, taking into account the schema and the permissions from each server. This is compatible with all the proposals for distributed databases aiming at a realization on concrete systems, which assume the use of a centralized optimizer; a purely distributed approach based on some form of negotiation protocol among the servers is considered impractical.

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, when the semantics is clear from the context.

5.3 Security Model

We first present our simple, while expressive, permissions 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.

5.3.1 Permissions

Consistently with standard practice in the security world, we assume a "closed" policy, where data can be made visible only to parties explicitly authorized for that.

Different subjects in the system may be authorized to view portions of the whole database content. We consider permissions in a simple, yet powerful form, specifying visibility *permissions* for subjects to view certain schema components. Formally, permissions are defined as follow.

Definition 5.2 (Permission). A *permission* p is a rule of the form $[\text{Att}, \text{Rels}]{\rightarrow}S$ where:

- *Att* is a set of attributes, belonging to one or more relations, whose release is being authorized;
- *Rels* is a set of relations such that for every attribute in *Att* there is a relation including it;
- S is a subject in $\mathscr{S}$.

Permission $[\text{Att}, \text{Rels}]{\rightarrow}S$ states that subject S can view the sub-tuples over the set of attributes *Att* belonging to the join among relations *Rels* (on conditions $\mathit{joinpath}(\mathit{Rels})$).

Note that, according to the definition, only attribute names (without indication of the relation) appear in the first component of the permission, whereas the relation (or relations) to which the attribute belongs is specified in the second component. This occurs even when the attribute appears in more than one relation (specified in *Rels*), consistently with the semantics that all the occurrences represent the same entity in the real world.

Example 5.4. Figure 5.3 illustrates some permissions on the relations in Fig. 5.1 that give `Alice` the visibility of:

- SSN, Date of Birth, and Race of all patients (p_1);

p_1: [(SSN,DoB,Race),(Patient)] $\rightarrow$Alice
p_2: [(SSN,Type,Cost,Duration),(Treatment)] $\rightarrow$Alice
p_3: [(Race,Specialty),(Treatment,Patient,Doctor)] $\rightarrow$Alice
p_4: [(SSN,Job,Salary),(Employee)] $\rightarrow$Alice
p_5: [(Name),(Treatment,Doctor)] $\rightarrow$Alice

Fig. 5.3 Examples of permissions

- SSN of treated patients, together with Type, Cost, and Duration of their treatments (p_2);
- Race of patients and Specialty of their caring doctors (p_3);
- SSN, Job, and Salary of all employees (p_4);
- Name of doctors who have prescribed some treatment (p_5).

Note that the *presence of a relation* (and therefore the enforcement of the corresponding join condition) in a permission may *decrease the set of tuples* that are made visible (to only those tuples that participate in the join). However, such an elimination of tuples does not correspond to less information, rather it *adds information* on the fact that the visible tuples actually join with (i.e., have values appearing in) other tuples of the joined relations. For instance, permission p_5 while restricting the set of doctor's names visible to `Alice` to only the names of the doctors who have prescribed treatments, it allows `Alice` to see that such doctors have prescribed treatments (i.e., they appear in relation `Treatment`).

The only case where including an additional relation in the permission does not influence the result, and therefore does not imply an indirect information disclosure, occurs when the additional relations are reachable via referential integrity constraints (from the foreign to the primary key it references) from a relation in *Rels*. For instance, permissions p_2 in Fig. 5.3 and a permission with the same first component as p_2 and having (`Treatment`,`Patient`,`Doctor`) as a second component, are completely equivalent as they permit (direct or indirect) release of exactly the same information. Indeed, given the existing referential integrity constraints (see $\mathscr{I}$ in Fig. 5.1), all `SSN` and all `IdDoc` appearing in `Treatment` also appear in `Patient` and `Doctor` respectively. The added joins are therefore ineffective.

Note how the simple form of permissions above, with the specification of the relations as a separate element, proves quite expressive. In particular, the *Rels* component may also include relations whose attributes do not appear in the set *Att* of attributes. This may be due to either:

- *connectivity constraints*, where these relations are needed to build the association among attributes of other relations (i.e., the relations are in the join path). For instance, in permission p_3 in Fig. 5.3, `Treatment` relation appears in the join path to establish the association between each patient and her caring doctors, but none of its attributes is released. Note how the permission allows `Alice` to view the speciality of patients' doctors without need of knowing their treatment.
- *instance-based restrictions*, where the relations are needed to restrict the attributes to be released to only those values appearing in tuples that can be asso-

ciated with such relations. For instance, permission p_5 in Fig. 5.3 allows `Alice` to view the names of all the doctors who prescribed at least a treatment (i.e., tuples in the `Doctor` relation satisfying `Doctor.IdDoc=Treatment.IdDoc` condition) but not of those doctors who never prescribed a treatment. 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, we can define a permission such that providing the employees' SSN, the company can retrieve their treatments.

5.3.2 *Relation Profiles*

Permissions restrict the data (view) that can be released to each subject. To determine whether a release should be authorized or not, we first need to capture the information content of the released relation, which can be either base or computed by a query. To this purpose, we introduce the concept of *relation profile* as follows.

Definition 5.3 (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, set of base relations joined for the definition/construction of R;
- R^{σ} is the, 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 $[\{a_1,\ldots,a_n\}, R, \emptyset]$.

The reason why both *i)*the attributes being returned as result (i.e., the attributes in the SELECT clause) and *ii)* the attributes on which the query imposes conditions (i.e., the attributes in the WHERE clause) appear in the profile reflects the fact that the query result returns indeed information on both (or, equivalently, the subject needs permissions to view both for accessing the relation to be released).

Note also that, like for permissions, only attribute names (without indication of the relation) appear in the first component of the query profile, while the relation (or relations) to which the attributes belong is specified in the second component. Indeed, if an attribute belongs to more than one relation (and therefore participates in the join), the common values of such an attribute in all relations are released by the query, regardless of the specific relation mentioned in the SELECT clause, which is needed for disambiguating attribute names. The consideration of the attribute names allows us to conveniently capture this aspect regardless of the specific way in which the query has been written. For instance, with respect to the query in Example 5.3, the set of social security numbers released by the query is the intersection of the set of `SSN` values of patients, employees, and treatments as captured in the profile:

	Profile		
Operation	R^{π}	$R^{\bowtie}$	R^{σ}
$R := \pi_A(R_l)$	A	$R_l^{\bowtie}$	R_l^{σ}
$R := \sigma_A(R_l)$	R_l^{π}	$R_l^{\bowtie}$	$R_l^{\sigma} \cup A$
$R := R_l \bowtie_j R_r$	$R_l^{\pi} \cup R_r^{\pi}$	$R_l^{\bowtie} \cup R_r^{\bowtie}$	$R_l^{\sigma} \cup R_r^{\sigma}$

Fig. 5.4 Profiles resulting from operations

[(SSN,Salary,DoB), (Employee,Patient,Treatment), (Duration)]. As a matter of fact, a query equal to the query in Example 5.3 but releasing P.SSN or T.SSN instead of E.SSN, while slightly different in the syntax, would carry exactly the same information content and, consequently, would have the same profile.

According to the semantics of the relational operators, the profile resulting from a relational operation, summarized in Fig. 5.4,[2] is as follows.

- *Projection* (π). A projection operation 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). A selection operation 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* ($\bowtie$). A join operation 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}, R^{π}, and $R^{\bowtie}$ of the resulting relation R are the union of those of the operands, implicitly capturing the join path *joinpath*($R^{\bowtie}$) among the relations composing $R^{\bowtie}$ and consequently the set of conditions that each tuple in R satisfies.

5.4 Graph-based Model

We model database schema, permissions, and queries via mixed graphs, that is, graphs with both undirected and directed arcs.

The *schema graph* of a set $\mathscr{R}$ of relations is a mixed graph whose nodes correspond to the different attributes of the relations, whose non-oriented arcs correspond to the possible joins ($\mathscr{J}$), and whose oriented arcs correspond to the referential integrity constraints ($\mathscr{I}$) and the functional dependencies between the primary key of a relation and its non-key attributes. Attributes appearing with the same name in more than one relation appear as different nodes. To disambiguate, nodes are identi-

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

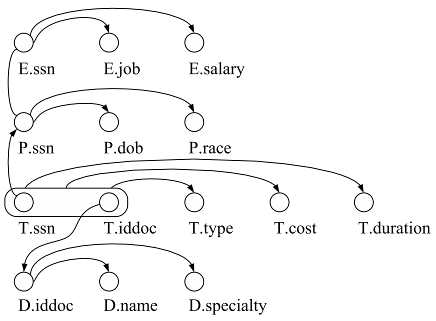


Fig. 5.5 Schema graph for the relations in Fig. 5.1

fied with the usual dot notation by the pair *relation.attribute*. This is formalized by the following definition.

Definition 5.4 (Schema graph). Given a set $\mathscr{R}$ of relations, a set $\mathscr{I}$ of referential integrity constraints over $\mathscr{R}$, and a set $\mathscr{J}$ of join conditions over $\mathscr{R}$, a *schema graph* is a graph $G(\mathscr{N},\mathscr{E})$ where:

- $\mathscr{N} = \{R_i.* : R_i \in \mathscr{R}\}$
- $\mathscr{E} = \mathscr{J} \cup \mathscr{I} \cup \{(R_i.K, R_i.a) : R_i \in \mathscr{R} \land a \notin K\}$

Figure 5.5 represents the schema graph corresponding to the set of relations, referential integrity constraints, and join conditions in Fig. 5.1 (for simplicity, we only report the initials of the relations).

Permissions and relation profiles correspond to *views* over the set $\mathscr{R}$ of relations and are characterized by a pair $[A,\mathbb{R}]$, corresponding to $[Att,Rels]$ appearing in the permissions, and to $[R^{\pi} \cup R^{\sigma}, R^{\bowtie}]$ in the relation profile of relation R, respectively.

Definition 5.5 (Entailed view). Given a set $\mathscr{R}$ of relations and a permission $p=[Att,Rels]$ over it, the view $V=[A,\mathbb{R}]$ entailed by p is defined as: $A=Att$ and $\mathbb{R}=Rels$. Given a set $\mathscr{R}$ of relations and a relation profile $[R^{\pi},R^{\bowtie},R^{\sigma}]$, the view $V=[A,\mathbb{R}]$ entailed by the profile is defined as: $A=R^{\pi} \cup R^{\sigma}$ and $\mathbb{R}=R^{\bowtie}$.

In the characterization of the view, we take into consideration the fact that referential integrity constraints can be used to extend the relations in $\mathbb{R}$ to include all relations reachable from the ones appearing in $\mathbb{R}$ by following referential integrity connections from a foreign key to the referenced primary key. We can then include such relations in the set $\mathbb{R}$. Given a set $\mathbb{R}$ of relations, $\mathbb{R}^*$ denotes the relations obtained by closing $\mathbb{R}$ with respect to referential integrity constraints. For instance, with respect to the schema graph in Fig. 5.5, the closure of $\mathbb{R}$={`Treatment`} is $\mathbb{R}^*$={`Treatment`, `Patient`, `Doctor`}.

Given a relation profile/permission, we graphically represent the view entailed through it as a *view graph* obtained by coloring the original schema graph with three colors: *black* for information that the view carries (i.e., it explicitly contains or indirectly conveys); *white* for all the non-black attributes belonging to relations in $\mathbb{R}^*$ and the arcs connecting them to the primary key; and *clear* for any other attribute or arc. Intuitively, clear nodes/arcs are attributes/arcs belonging to the original graph that are ineffective with respect to the evaluation and composition of permissions. The reason for maintaining them in the view graphs is so that every query/permission is a coloring (in contrast to a subgraph) of the schema graph. View graph is formally defined as follows.

Definition 5.6 (View graph). Given a set $\mathscr{R}$ of relations characterized by schema graph $G(\mathscr{N},\mathscr{E})$ and a view $V = [A,\mathbb{R}]$ entailed by a permission/relation profile on it, the *view graph* of V over G is a graph $G_V(\mathscr{N},\mathscr{E},\lambda_V)$, where $\lambda_V : \{\mathscr{N} \cup \mathscr{E}\} \rightarrow \{black,white,clear\}$ is a coloring function defined as follows.

$$\lambda_V(n)=\begin{cases} black, \ n{=}R.a,\ R \in \mathbb{R}^* \wedge a \in A \\ white, \ n{=}R.a,\ R \in \mathbb{R}^* \wedge a \notin A \\ clear, \ \text{otherwise} \end{cases}$$

$$\lambda_V(n_i,n_j)=\begin{cases} black, & (n_i,n_j) \in \text{joinpath}(\mathbb{R}^*) \vee \\ & (n_i{=}R.K,\ n_j{=}R.a,\ R \in \mathbb{R}^*,(a \in A \vee R.a \text{ appears in joinpath}(\mathbb{R}^*))) \\ white, & n_i{=}R.K,\ n_j{=}R.a,\ R \in \mathbb{R}^*, \\ & \neg(a \in A \vee R.a \text{ appears in joinpath}(\mathbb{R}^*)) \\ clear, & \text{otherwise} \end{cases}$$

According to this definition, a node is colored as: *black* if it appears in A, *white* if it is not black and it belongs to a relation appearing in $\mathbb{R}^*$, and *clear* otherwise. An arc is colored: *black* if either it belongs to *joinpath*$(\mathbb{R}^*)$ or it is an arc going from the key of a relation in $\mathbb{R}^*$ to an attribute which either belongs to A or appears in *joinpath*$(\mathbb{R}^*)$; *white* if it is an arc from the key of a relation in $\mathbb{R}^*$ to one of its attributes which neither belongs to A nor appears in *joinpath*$(\mathbb{R}^*)$; *clear* otherwise.

Figure 5.6 illustrates the **ColorGraph** function that given the schema graph G and a pair $[A,\mathbb{R}]$ denoting either the view entailed by a permission or by a relation profile, implements Definition 5.6 and returns the corresponding view graph. **ColorGraph**, whose interpretation is immediate, starts by assigning a clear color to all nodes and arcs and proceeds by coloring black and white arcs and nodes as prescribed by the definition.

Figure 5.7 reports the view graphs corresponding to the permissions in Fig. 5.3. Figure 5.8 reports some examples of relations obtained through queries over the schema in Fig. 5.5. The figure reports the queries originating the relations, the relation profiles, and the corresponding view graphs.

Before closing this section we introduce two dominance relationships between view graphs that will be used in the remainder of the chapter.

```
**COLORGRAPH**($G$,[$A$,$\mathbb{R}$])
$\mathcal{N}_V := \mathcal{N}$
$\mathcal{E}_V := \mathcal{E}$
for each $n \in \mathcal{N}_V$ do $\lambda_V(n)$ := clear
for each $(n_i,n_j) \in \mathcal{E}_V$ do $\lambda_V(n_i,n_j)$ := clear
for each $R \in \mathbb{R}^*$ do
    for each $a \in R.*$ do /* color nodes */
        if $a \in A$ then
            $\lambda_V(R.a)$ := black
        else
            $\lambda_V(R.a)$ := white
for each $(n_i,n_j) \in joinpath(\mathbb{R}^*)$ do /* color the join path */
    $\lambda_V(n_i,n_j)$ := black
for each $(n_i,n_j) \in \{(n_i,n_j)$: $\exists R \in \mathbb{R}^*$, $n_i = R.K \wedge n_j \subseteq R.*\}$ do
    if $\lambda_V(n_j)$=black $\vee$ $n_j$ appears in $joinpath(\mathbb{R}^*)$ then
        $\lambda_V(n_i,n_j)$ := black
    else
        $\lambda_V(n_i,n_j)$ := white
$G_V$ := $(\mathcal{N}_V,\mathcal{E}_V,\lambda_V)$
return($G_V$)
```

Fig. 5.6 Function for coloring a view graph

Definition 5.7 ($\preceq_N$, $\preceq_{NE}$). Given a schema graph $G(\mathcal{N},\mathcal{E})$, and two view graphs $G_{V_i}(\mathcal{N},\mathcal{E},\lambda_{V_i})$ and $G_{V_j}(\mathcal{N},\mathcal{E},\lambda_{V_j})$ over G, the following dominance relationships are defined:

- $G_{V_i} \preceq_N G_{V_j}$, when $\forall n \in \mathcal{N}$ and $\forall(n_h,n_k) \in (\mathcal{J} \cup \mathcal{I})$:
  - $\lambda_{V_i}(n)$ = black $\Longrightarrow \lambda_{V_j}(n)$=black, and
  - $\lambda_{G_i}(n_h,n_k)$ = black $\Longleftrightarrow \lambda_{G_j}(n_h,n_k)$ = black.
- $G_{V_i} \preceq_{NE} G_{V_j}$, when $\forall n \in \mathcal{N}$ and $\forall(n_h,n_k) \in \mathcal{E}$:
  - $\lambda_{V_i}(n)$ = black $\Longrightarrow \lambda_{V_j}(n)$=black, and
  - $\lambda_{G_i}(n_h,n_k)$ = black $\Longrightarrow \lambda_{G_j}(n_h,n_k)$ = black.

According to this definition, given two graphs G_{V_i} and G_{V_j} on the same database schema, $G_{V_i} \preceq_N G_{V_j}$ if they have exactly the same black referential integrity and join arcs and the black nodes of G_{V_i} are a subset of the black nodes of G_{V_j}. $G_{V_i} \preceq_{NE} G_{V_j}$ if the black arcs and nodes of G_{V_i} are a subset of the black arcs and nodes of G_{V_j}. For instance, with reference to the view graphs in Figs. 5.7 and 5.8, it is easy to see that: $G_{p_3} \preceq_N G_{Q_3}$ and that $G_{p_1} \preceq_{NE} G_{Q_2}$.

5.5 Authorized Views

To evaluate a query requested by a subject against her permissions and to determine if the query can be executed, we implement the following intuitive concept.

Principle 5.1 *A relation (either base or resulting from a query evaluation) can be released to a subject if she has permissions to view the information content carried by the relation.*

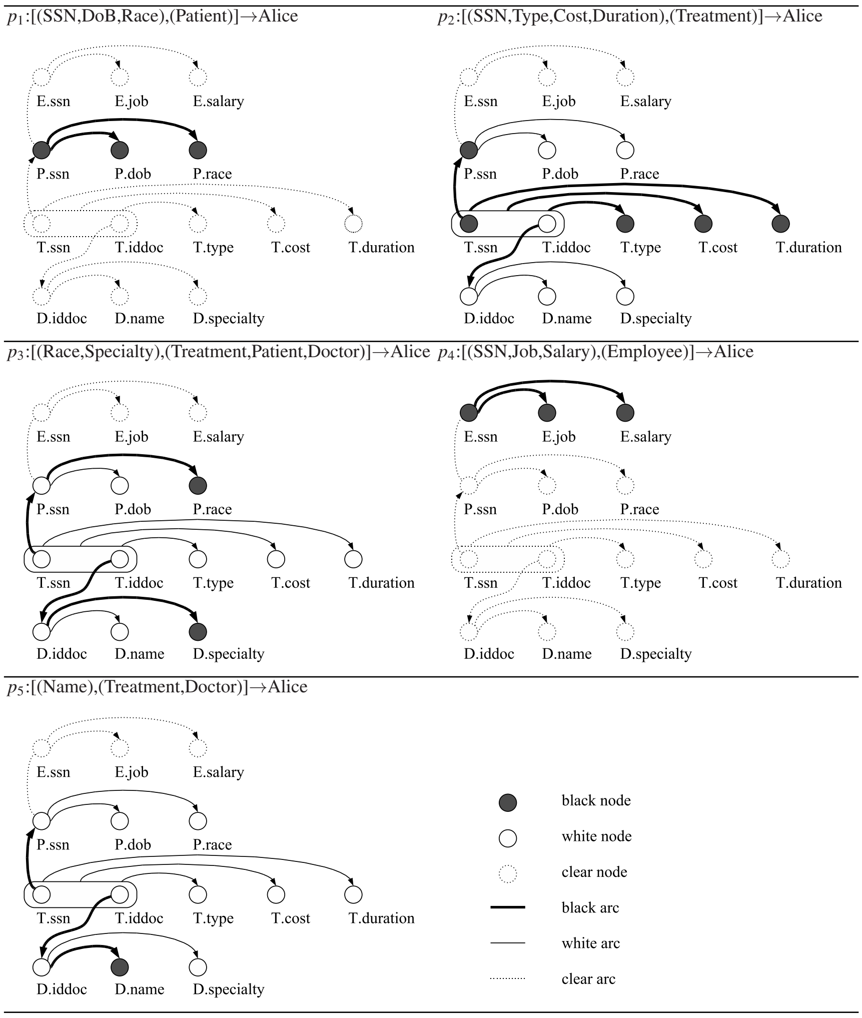

p1:[(SSN,DoB,Race),(Patient)]→Alice
p2:[(SSN,Type,Cost,Duration),(Treatment)]→Alice
p3:[(Race,Specialty),(Treatment,Patient,Doctor)]→Alice
p4:[(SSN,Job,Salary),(Employee)]→Alice
p5:[(Name),(Treatment,Doctor)]→Alice
E.ssn
E.job
E.salary
P.ssn
P.dob
P.race
T.ssn
T.iddoc
T.type
T.cost
T.duration
D.iddoc
D.name
D.specialty
black node
white node
clear node
black arc
white arc
clear arc

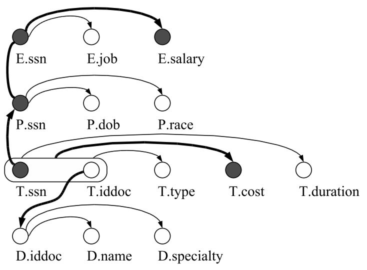
Fig. 5.7 Examples of permissions and their view graphs

We first discuss when a permission authorizes the release of a relation. We will then address permission composition and cooperation in query evaluation.

In the reminder of this section we refer our discussion to permissions and relation profiles of a *specific subject* and omit, for simplicity, the subject component of permissions in the formalization.

Q_1

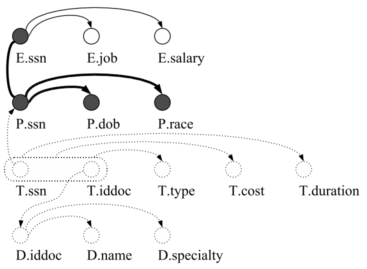
```
SELECT  E.SSN,Salary
FROM    Employee AS E
        JOIN Patient AS P
        ON E.SSN=P.SSN
        JOIN Treatment AS T
        ON T.SSN=P.SSN
WHERE   Cost> 250
```



[(SSN,Salary), (Employee,Patient,Treatment), (Cost)]

Q_2

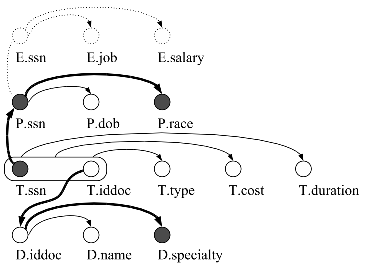
```
SELECT  P.SSN,DoB
FROM    Employee AS E
        JOIN Patient AS P
        ON E.SSN=P.SSN
WHERE   Race=‘asian’
```



[(SSN,DoB), (Employee,Patient), (Race)]

Q_3

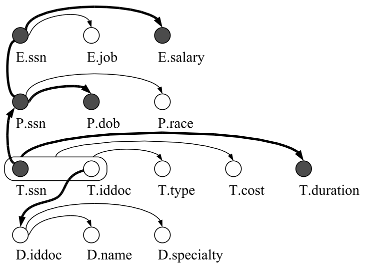
```
SELECT  P.SSN,Race
FROM    Patient AS P
        JOIN Treatment AS T
        ON T.SSN=P.SSN
        JOIN Doctor AS D
        ON T.IdDoc=D.IdDoc
WHERE   Specialty=‘cardiology’
```



[(SSN,Race), (Patient,Treatment,Doctor), (Specialty)]

Q_4

```
SELECT  E.SSN,Salary,DoB
FROM    Employee AS E
        JOIN Patient AS P
        ON E.SSN=P.SSN
        JOIN Treatment AS T
        ON P.SSN=T.SSN
WHERE   Duration> 10
```



[(SSN,Salary,DoB), (Employee,Patient,Treatment), (Duration)]

Fig. 5.8 Examples of queries, their relation profiles, and their view graphs

5.5.1 Authorizing Permissions

Intuitively, a permission authorizes a release if and only if the information (directly or indirectly) entailed by the relation profile is a subset of the information that the permission authorizes to view. Note that this is different from saying that the relation should contain only data that are a subset of the data authorized by the permission, as this denotes only the information directly released. A correct enforcement should also ensure that no indirect release occurs. There are two main sources of indirect release:

- the presence, in the query generating the relation, of conditions on attributes that are not returned (i.e., attributes that appear in the WHERE clause but do not appear in the SELECT clause);
- the presence of join conditions restricting the tuples returned by the query.

The first aspect is easily taken into consideration as it is already captured by the inclusion, in the relation profile (Definition 5.3), of R^{σ} component, which is included in A for the entailed view definition (Definition 5.5). To illustrate the problem of the second aspect, consider permission p_1 in Fig. 5.7, which allows Alice to view the complete information in Patient, and therefore the whole tuples representing all patients. Permission p_1 by itself is then sufficient to grant Alice the ability to view the data of all patients (i.e., relation obtained through query "SELECT P.SSN,DoB FROM Patient AS P WHERE Race='asian' "). Suppose instead that Alice is interested in the relation resulting from Q_2 in Fig. 5.8. This latter query returns a subset of all the tuples of patients, and therefore only tuples that Alice, according to p_1, is authorized to see. However, permission p_1 is not sufficient for granting Alice such visibility on data, since the query result conveys the additional information that the returned tuples refer to patients who are also employees of the considered company (information which permission p_1 does not authorize).

As already commented in Sect. 5.4, the only case when joins do not add information is when there is a referential integrity constraint among the involved relations. Consider, for example, permission p_2 authorizing the release of different attributes in Treatment. For instance, query "SELECT T.SSN FROM Treatment AS T" is clearly authorized by p_2. Consider then the same query containing, in the FROM clause, also relations Patient and Doctor with the corresponding joins. Despite the presence of the additional joins, such a query does not bear additional information (indirect release) and should therefore be authorized by p_2. As a matter of fact, because of the referential integrity constraints between the involved relations, all SSN's and IdDoc's appearing in Treatment also appear in Patient and Doctor, respectively, and therefore the joins do not impose restrictions. The consideration of the peculiar characteristics of joins due to referential integrity constraints is easily taken into account, since it is already captured by the coloring, in the view graph, of all the relations reachable from the ones appearing in the query, by following referential integrity constraints (Definition 5.6).

Let us then proceed to formally define when a permission authorizes the release of a relation. We start by identifying permissions applicable to a relation profile. Intuitively, a permission applies to a relation when it refers to the complete set of tuples composing the relation. Since tuple restriction is due to joins not following the direction from a foreign key to the referenced key in a referential integrity constraint (as commented above), this is equivalent to saying that the permission applies to a relation profile if it does not contain additional joins (apart from those corresponding to referential integrity constraints). This is formalized by the following definition.

Definition 5.8 (Applicable). A permission $[Att,Rels]$ is applicable to a relation profile $[R^{\pi},R^{\bowtie},R^{\sigma}]$ iff $Rels^{*}\subseteq R^{\bowtie *}$.

In terms of view graphs, this definition is equivalent to say that the black and white nodes of the view graph G_p of permission p should be a subset of the black and white nodes of the view graph G_R of the relation profile of R.

According to the discussion above, a permission authorizes the release of a relation if and only if the permission applies to the relation profile and authorizes the release, either direct of indirect, of the information in the profile. This means that the permission should include (at least) all attributes composing the relation or accessed for its definition/computation as well as all the join conditions. In terms of the view graphs, this is equivalent to say that the view graph G_R of the relation profile and the view graph G_p of the permission have exactly the same black referential integrity and join arcs and that all nodes that are black in the view graph of the relation profile are also black in the view graph of the permission, that is, $G_R \preceq_N G_p$. This is formally captured by the following definition.

Definition 5.9 (Authorizing permission). Given a permission p=$[Att,Rels]$ applicable to a relation profile R=$[R^{\pi},R^{\bowtie},R^{\sigma}]$, p *authorizes* the release of R iff $G_R \preceq_N G_p$.

As an example, with reference to the permissions in Fig. 5.7 and the relation computed through query Q_2 in Fig. 5.8, the set of permissions applicable includes p_1 and p_4. However, neither p_1 nor p_4 authorize the release of the query result. By contrast, considering query "SELECT P.SSN,DoB FROM Patient AS P WHERE Race='asian' ", with profile [(SSN,DoB), (Patient), (Race)] permission p_1 is the only applicable permission that also authorizes the query.

5.5.2 *Composition of Permissions*

Checking relation profiles against individual permissions is not sufficient for a true enforcement of Principle 5.1. Indeed, it might be that for a relation profile there is no permission that singularly taken authorizes the release of the relation, however information released (directly or indirectly) by the relation profile is authorized. As an example, consider permissions p_1 and p_4 in Fig. 5.3 and suppose that Alice requests the relation resulting from query Q_2 in Fig. 5.8, returning the tuples associated with patients whose SSN appears also in the Employee relation. While neither p_1 nor p_4 authorize the relation profile (as, for each of them, the relation profile has the additional join condition that the permission does not authorize), it is clear that the relation does not contain any information that Alice is not authorized to see. As a matter of fact, Alice could indeed separately query both relations and then join the two results. In the spirit of Principle 5.1, the release of the result of

```
COMPOSE($G$,$p_i$,$p_j$)
$p$ := [$Att_i \cup Att_j$,$Rels_i \cup Rels_j$]
$\mathcal{N}_p$ := $\mathcal{N}$
$\mathcal{E}_p$ := $\mathcal{E}$
for each $n \in \mathcal{N}_p$ do $\lambda_V(n)$ := clear
for each $(n_i,n_j) \in \mathcal{E}_p$ do $\lambda_V(n_i,n_j)$ := clear
for each $n \in \mathcal{N}_p$ do
    if $\lambda_{p_i}(n)$=black$\vee\lambda_{p_j}(n)$=black then
      $\lambda_p(n)$=black
    else
      if $\lambda_{p_i}(n)$=white$\vee\lambda_{p_j}(n)$=white then
        $\lambda_p(n)$=white
for each $(n_h,n_k) \in \mathcal{E}_p$ do
    if $\lambda_{p_i}(n_h,n_k)$=black$\vee\lambda_{p_j}(n_h,n_k)$=black$\vee$ ($\lambda_p(n_h)$=black$\wedge\lambda_p(n_k)$=black) then
      $\lambda_p(n_h,n_k)$=black
    else
      if $\lambda_{p_i}(n_h,n_k)$=white$\vee\lambda_{p_j}(n_h,n_k)$=white then
        $\lambda_p(n_h,n_k)$=white
return($p$)
```

Fig. 5.9 Function composing two permissions

query Q_2 to Alice should therefore be authorized. To enforce this principle, we compose permissions and consider a release of a relation authorized if there exists a composition of permissions that authorizes it.

Composition of permissions must however be performed carefully to ensure that composition does not authorize additional queries that were authorized by neither of the original permissions. To illustrate, consider again the permissions in Fig. 5.7 and suppose that Alice is interested in the relation resulting from query Q_3. One could think that such a release can be authorized by composing p_1 in Fig. 5.7 (authorizing the release of SSN's and Race's) and p_3 (authorizing the release of the race of patients together with the specialty of their caring doctor). However, such a composition does not authorize the relation release. Indeed, the relation profile conveys the associations between a patient and her caring doctor, which neither of the individual permissions authorize and which Alice would not be able to reconstruct by separately exploiting the privileges granted by the two permissions. The problem, in this case, is that the composition of the two permissions returns more information than that entailed by the two permissions individually taken. If this is the case, the two permissions should not be composed.

To determine when two permissions can be composed, we exploit one of the foundational results of the theory of joins for relational databases, expressed by the theorem presented in [5], which states that two relations produce a *lossless join* if and only if at least one of the two relations *functionally depends* from the intersection of their attributes. The relations that are considered in the theorem correspond to generic projections on the set of attributes that characterizes the "universal relation" obtained joining all the relations of our lossless acyclic schema; this means that each permission corresponds to a relation and that the composition of permissions is correct only if the above requirement is satisfied. For instance, consider the previous examples and the permissions in Fig. 5.7. Permissions p_1 and p_4 can be combined because their intersection is represented by attribute SSN, which is a key for all the attributes in p_1 (and p_4). Permissions p_1 and p_3 cannot be combined because their intersection is represented by attribute `Race`, and neither p_1 nor p_3 functionally depend on it.

The application of this basic result of the theory of joins in our scenario is slightly complicated by the fact that the views corresponding to given permissions may include attributes from different relations. (We note here that intersection of permissions is computed based only on the attribute names, without considering the relation they belong to, since attributes with the same name represent the same real world concept and natural joins impose them to be equal in all the resulting tuples.) Given two permissions p_i=[Att_i,$Rels_i$] and p_j=[Att_j,$Rels_j$] their composability depends on the intersection of their visible attributes (i.e., $Att_i \cap Att_j$) but the functional dependency of the visible attributes of one of the two permissions from the common attributes needs to be evaluated by taking into account also the referential integrity constraints. This concept can be easily captured by analyzing the view graphs G_{p_i} and G_{p_j} corresponding to the two permissions. The basic idea is that there is a dependence between p_i and p_j when there is a black path from nodes corresponding to the attributes that are listed both in Att_i and in Att_j to all the black nodes in G_{p_i} or in G_{p_j}. This intuitive concept of dependency is formalized as follows.

Definition 5.10 (Dependence). Given two permissions p_i=[Att_i,$Rels_i$] and p_j=[Att_j,$Rels_j$] with view graphs $G_{p_i}(\mathscr{N},\mathscr{E},\lambda_{p_i})$ and $G_{p_j}(\mathscr{N},\mathscr{E},\lambda_{p_j})$, respectively, let B_j be the set of nodes corresponding to $\{Att_i \cap Att_j\}$ in G_{p_j}. We say that p_j *depends* on p_i, denoted $p_i \rightarrow p_j$, iff $\forall n_j \in \mathscr{N}$ such that $\lambda_{p_j}(n_j)$=black, $\exists n \in B_j$ such that there is a path of only directed black arcs from n to n_j in G_{p_j}.

In the following, notation $p_i \leftrightarrow p_j$ denotes that both $p_i \rightarrow p_j$ and $p_j \rightarrow p_i$ hold. Similarly, $p_i \not\leftrightarrow p_j$ denotes that neither $p_i \rightarrow p_j$ nor $p_j \rightarrow p_i$ hold.

For instance, with reference to the permissions in Fig. 5.7, as already noted, $p_2 \rightarrow p_1$, since common attribute `SSN` is key for the `Patient` relation authorized by p_1, and $p_1 \not\rightarrow p_2$, since the attributes released by p_2 depend on the pair of attributes `SSN` and `IdDoc`. We also note that $p_1 \leftrightarrow p_4$, since the `SSN` attribute, common to the two permissions, is the key of both the `Patient` and `Employee` relations. On the contrary, as already pointed out, $p_1 \not\leftrightarrow p_3$.

If $p_i \rightarrow p_j$ (or $p_j \rightarrow p_i$, respectively), then the two permissions can be *safely composed*, as formally stated by the following definition.

Definition 5.11 (Safe composition). Given two permissions p_i=[Att_i,$Rels_i$] and p_j=[Att_j,$Rels_j$], p_i and p_j can be *safely composed* when $p_i \rightarrow p_j$, or $p_j \rightarrow p_i$, or both.

For instance, p_1 can be safely composed with p_2, since $p_2 \rightarrow p_1$. Also, since $p_1 \leftrightarrow p_4$, p_1 can be safely composed with p_4.

Similarly to the composition of relations presented in the theory of normal forms for relational databases, the composition of p_i with p_j generates a new permission that combines the viewing privileges of the two, as stated by the following definition.

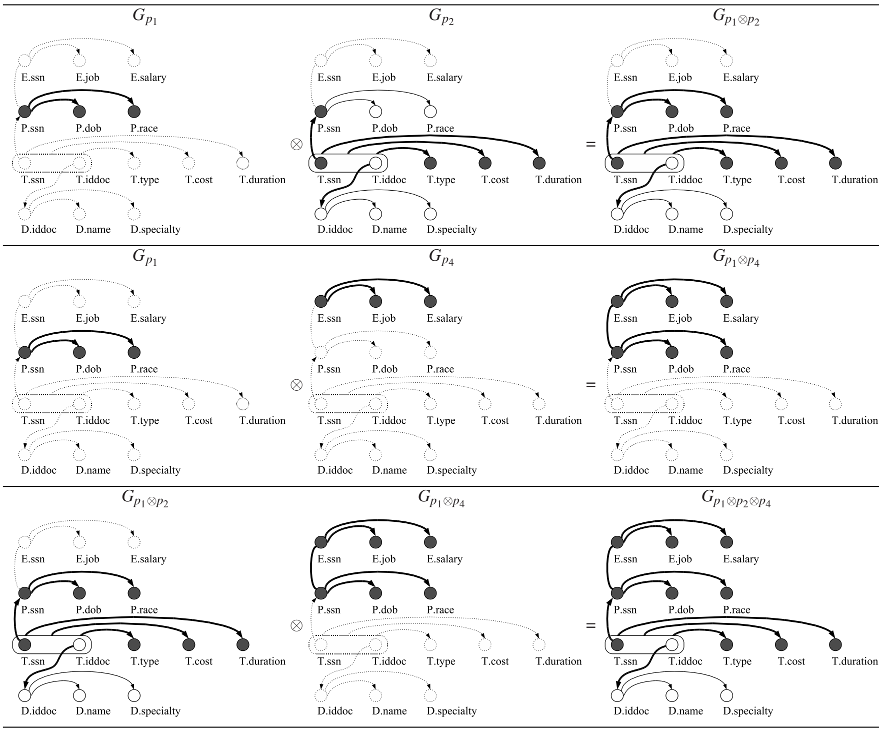


Fig. 5.10 Examples of permission compositions

Definition 5.12 (Composed permission). Given two permissions p_i=[Att_i,$Rels_i$] and p_j=[Att_j,$Rels_j$], their composition is the permission $p_i \otimes p_j$=[$Att_i \cup Att_j$,$Rels_i \cup Rels_j$].

It is easy to see that the view graph of the resulting composed permission is obtained from the view graphs of the components as follows. A node in $G_{p_i \otimes p_j}$ is: *black* if it is black in either G_{p_i} or G_{p_j}; *white* if it is not black and it is white in either G_{p_i} or G_{p_j}; it is *clear* otherwise. An arc in $G_{p_i \otimes p_j}$ is: *black* if it is black in either G_{p_i} or G_{p_j} or if it is incident on only black nodes in $G_{p_i \otimes p_j}$; *white* if it is not black and is white in either G_{p_i} or G_{p_j}; it is *clear* otherwise. Figure 5.10 represents the view graphs resulting from a subset of the safe compositions of the privileges in Fig. 5.7, that is, $p_1 \otimes p_2$, $p_1 \otimes p_4$, and $p_1 \otimes p_2 \otimes p_4$.

A permission obtained by composing permissions p_i and p_j ($p_i \otimes p_j$) can be composed with a permission p_k that did not satisfy the composition requirements with p_i nor with p_j. In general, each new permission produces new opportunities for composition that have to be considered. The consideration of all the potential compositions is modeled by the following concept.

Definition 5.13 (Composition closure). Given a set of permissions $\mathscr{P}$, the *closure on composition* of $\mathscr{P}$, denoted $\mathscr{P}^{\otimes}$, is the set of permissions obtained as a fixpoint by the procedure which repeatedly extends $\mathscr{P}$ with all permissions obtained by the safe composition of the permissions in $\mathscr{P}$.

For instance, with reference to the set of permissions in Fig. 5.7, their closure is $\mathscr{P}^{\otimes}=\{p_1, p_2, p_3, p_4, p_5, p_1{\otimes}p_2, p_1{\otimes}p_4, p_2{\otimes}p_4, p_1{\otimes}p_2{\otimes}p_4\}$.

The closure represents the greatest representation of the permissions available to a subject. This concept permits to identify in a complete way if a specific relation profile is authorized for a subject.

Definition 5.14 (Authorized release). Given a set $\mathscr{P}$ of permissions applicable to a relation profile $[R^{\pi}, R^{\bowtie}, R^{\sigma}]$, $\mathscr{P}$ authorizes R iff $\exists p \in \mathscr{P}^{\otimes}$ such that p authorizes R (according to Definition 5.9).

The computation of the closure on composition of permissions is potentially an expensive procedure. In the following, we present an efficient algorithm that avoids computing the whole set of permissions in the composition closure while ensuring completeness of the control, needed to evaluate if a release is authorized.

5.5.3 Algorithm

Given a set $\mathscr{P}$ of n permissions of a subject S applicable to a relation profile $[R^{\pi}, R^{\bowtie}, R^{\sigma}]$, the control for the authorized release does not require to compute all the possible $2^n - 1$ permission compositions, since given two permissions p_i and p_j, if $p_j{\rightarrow}p_i$ then p_j is subsumed by $p_i{\otimes}p_j$, and whenever a permission p_k can be composed with p_j, p_k can also be composed with $p_i{\otimes}p_j$, as stated by the following theorem.

Theorem 5.1 (Permission implication). *Given two permissions* p_i=[Att$_i$,Rels$_i$], p_j=[Att$_j$,Rels$_j$] $\in \mathscr{P}$ *such that* $p_j{\rightarrow}p_i$*,* $\forall p_k$=[Att$_k$,Rels$_k$] $\in \mathscr{P}$*:*

1. $p_j{\rightarrow}p_k \Rightarrow (p_i{\otimes}p_j){\rightarrow}p_k$*;*
2. $p_k{\rightarrow}p_j \Rightarrow p_k{\rightarrow}(p_i{\otimes}p_j)$*.*

Proof. Let us consider the two cases above.

1. Let $p_i{\otimes}p_j$=$[Att_{i,j}, Rels_{i,j}]$. Attributes in $Att_j{\cap}Att_k$ also appear in the intersection between $Att_{i,j}$ and Att_k. Therefore, there exists a path of only directed black arcs from a node corresponding to some attributes in $Att_j{\cap}Att_k$ to each black node in G_{p_k}.
2. From the hypothesis, we know that there is a path of only directed black arcs from a node corresponding to some attributes in $Att_j{\cap}Att_k$ to each black node in G_{p_j}. Also, we know that there is a path of only directed black arcs from a node corresponding to some attributes in $Att_i{\cap}Att_j$ to each black node in G_{p_i}. By combining these paths, it follows that also $p_k{\rightarrow}p_i$ and, therefore, that $p_k{\rightarrow}(p_i{\otimes}p_j)$.

```
**AUTHORIZED**($G_R$,$S$)
Let *Applicable* be the set of permissions [Att, Rels]$\rightarrow S_i$ such that:
$\{n\in\mathcal{N}_p:\lambda_p(n)=\text{black}\vee\text{white}\}\subseteq\{n\in\mathcal{N}_R:\lambda_R(n)=\text{black}\vee\text{white}\}\wedge S_i=S$
/* check individual permissions */
for each $p\in Applicable$ do
    if $G_R\preceq_N G_p$ then return(true)
/* compose permissions */
$maxid := |Applicable|$
$counter := 1$
for each $p\in Applicable$ do
    $p.id := counter$
    $p.maxcfr := counter$
    $counter := counter + 1$
$idmin_i := 1$
repeat
      Let $p_i$ be the permission with $p_i.id=idmin_i$
      $idmin_j :=$ Min($\{p.id{:}p\in Applicable\wedge p_i.maxcfr<p.id\}$)
      Let $p_j$ be the permission with $p_j.id=idmin_j$
      $dominated :=$ NULL
      if $(G_{p_i}\not\preceq_{NE}G_{p_j})\wedge(G_{p_j}\not\preceq_{NE}G_{p_i})$ then
        if $p_j\rightarrow p_i$ then $dominated := dominated\cup\{p_j\}$
        if $p_i\rightarrow p_j$ then $dominated := dominated\cup\{p_i\}$
      $p_i.maxcfr := p_j.id$
      if $dominated\neq$NULL then
        $maxid := maxid + 1$
        $p_{maxid}$ := Compose($G$,$p_i$,$p_j$)
        $p_{maxid}.id := maxid$
        $p_{maxid}.maxcfr := maxid$
        $Applicable := Applicable - dominated\cup\{p_{maxid}\}$
      $idmin_i :=$ Min($\{p.id{:}p\in Applicable\wedge p.maxcfr<maxid\}$)
until $idmin_i$=NULL
/* check resulting permissions */
for each $p\in Applicable$ do
    if $G_R\preceq_N G_p$ then return(true)
return(false)
```

Fig. 5.11 Function that checks if a release is authorized

This theorem implies that permission p_j can be removed from the set $\mathcal{P}$ without compromising the composition process. It is also easy to see that since the composed permission is again applicable to the relation profile $[R^{\pi},R^{\bowtie},R^{\sigma}]$, the set of permissions to be composed always contains at most n permissions (i.e., the composed permission substitutes one, or both, of the composing permissions). Function **Authorized** in Fig. 5.11 applies this observation to check whether a relation profile release is authorized. The function takes as input the view graph G_R representing the relation profile and the subjects requesting the release; on the basis of the set of applicable permissions, it returns *true* or *false*, depending on whether or not the query is authorized.

Initially, **Authorized** determines the set *Applicable* of applicable permissions and checks if one of these permissions dominates ($\preceq_N$) G_R. If this is the case, function **Authorized** returns *true*. Otherwise, the function starts the composition process that exploits Theorem 5.1 according to which permission p_i can be removed from set *Applicable* if $p_j\rightarrow p_i$. The applicable permissions are first ordered according to a numeric identifier *id*, ranging from 1 to $|Applicable|$, which is associated with each permission. In the **repeat until** loop, each permission p_i is compared with a permission p_j such that $p_i.id{<}p_j.id$. If the set of black nodes and arcs of G_{p_i} is not a subset of the set of black nodes and arcs of G_{p_j} (i.e., $G_{p_i}{\not\preceq}_{NE}G_{p_j}$, meaning that p_j has not been computed in a previous iteration by composing p_i with another authorization) and viceversa, function **Authorized** checks whether p_i and p_j can be composed (i.e., $p_j{\to}p_i$ or $p_i{\to}p_j$). If this is the case, the identifier of the resulting composed permission (if any) becomes equal to the current maximum identifier (*maxid*) incremented by one. Each permission p is also associated with a variable $p.maxcfr$ that keeps track of the highest identifier of the permissions compared to p. This variable avoids to check the same pair of permissions more than once. The composition process terminates when *maxcfr* of all permissions is equal to the highest identifier *maxid*. The function then checks if any of the permissions in *Applicable* dominates ($\preceq_N$) G_R. If this is the case, function **Authorized** returns *true*; otherwise it returns *false*.

Example 5.5. Consider the schema graph in Fig. 5.5, the set of permissions in Fig. 5.7, and the relation R_1 computed by query Q_1 in Fig. 5.8. As it is visible from the view graphs, all the five permissions are applicable to the profile of the relation resulting from Q_1. The table in Fig. 5.12 represents the execution, step by step, of function **Authorized** on G_{Q_1} by reporting the evolution of variable $p.maxcfr$ for both original and composed permissions. Each column in the table corresponds to a permission, whose identifier is the label of the column itself. Note that when a permission is removed from *Applicable*, its *maxcfr* is not reported anymore. Each row in the table represents an iteration of the **repeat until** loop, reporting both the dependence relationship between the composing permissions and the *maxcfr* for all permissions. Also, in each row the *maxcfr* of the permissions checked for a possible composition are reported in italic. When a permission is removed from *Applicable* (because subsumed by an added composed permission), its *maxcfr* is not reported anymore. Figure 5.10 represents the view graph of the permissions obtained by the composition. We then conclude that the relation resulting from the evaluation of query Q_1 can be released to `Alice`, since $p_1{\otimes}p_2{\otimes}p_4$ authorizes it.

The following theorems state the correctness and complexity of function **Authorized**.

Theorem 5.2 (Correctness). *Given a relation profile* $R{=}[R^{\pi},R^{\bowtie},R^{\sigma}]$ *and a set Applicable of applicable permissions, function* **Authorized** *terminates and returns* true *iff the release of R is authorized by Applicable*$^{\otimes}$.

Proof. Termination. All the **for** loops terminate, since *Applicable* (by Theorem 5.1) is composed of at most n permissions. At each iteration of the **repeat until** loop, function **Authorized** evaluates a pair of permissions $\langle p_i,p_j\rangle$ such that $p_i.maxcfr{<}p_j.id$. Two cases can occur: p_i and p_j cannot be composed, or p_i and p_j can be composed (and we suppose, without loss of generality, that $p_j{\to}p_i$). In the first case, in the subsequent iterations p_i and p_j are no more checked, since $p_i.id$ and $p_j.id$ do not change and $p_i.maxcfr$ is set to $p_j.id$. In the second case, p_i is removed from *Applicable*, while the composed permission $p{=}p_i{\otimes}p_j$ is added to *Applicable*.

id	1	2	3	4	5	6	7	8
	p_1	p_2	p_3	p_4	p_5			
initialization	1	2	3	4	5			
$p_2 \rightarrow p_1$	*1*	2	3	4	5	$p_1 \otimes p_2$		
$p_1 \not\leftrightarrow p_3$	*2*		*3*	4	5	6		
$p_1 \leftrightarrow p_4$	*3*		3	*4*	5	6	$p_1 \otimes p_4$	
$p_3 \not\leftrightarrow p_5$			*3*		*5*	6	7	
$p_3 \not\leftrightarrow (p_1 \otimes p_2)$			*5*		5	*6*	7	
$p_5 \not\leftrightarrow (p_1 \otimes p_2)$			6		*5*	*6*	7	
$p_3 \not\leftrightarrow (p_1 \otimes p_4)$			*6*		6	6	*7*	
$p_5 \not\leftrightarrow (p_1 \otimes p_4)$			7		*6*	6	*7*	
$(p_1 \otimes p_2) \rightarrow (p_1 \otimes p_4)$			7		7	*6*	*7*	$p_1 \otimes p_2 \otimes p_4$
$p_3 \not\leftrightarrow (p_1 \otimes p_2 \otimes p_4)$			*7*		7		7	*8*
$p_5 \not\leftrightarrow (p_1 \otimes p_2 \otimes p_4)$			8		*7*		7	*8*
$G_{p_1 \otimes p_4} \preceq_{NE} G_{p_1 \otimes p_2 \otimes p_4}$			8		8		*7*	*8*
			8		8		8	8

Fig. 5.12 An example of the execution of function **Authorized**

Since $p_j \preceq_{NE} p$, in the following iterations, when they are compared they do not generate new permissions. Since each possible combination is checked only once and the number of possible combination is finite, the **repeat until** loop terminates.

Correctness. If there exists a permission $p \in Applicable^{\otimes}$ that authorizes the release of R, two cases can occur: $p \in Applicable$, or p is a composed permission. In the first case, **Authorized** returns *true* since the first **for** loop iterates on all permissions in *Applicable*. In the second case, the **repeat until** loop removes from *Applicable* only non-necessary permissions (see Theorem 5.1) and checks all non-redundant pairs of permissions in *Applicable*. The **repeat until** loop terminates when, for all p in *Applicable*, $p.maxcfr{=}maxid$. Since $p.maxcfr$ is initialized to $p.id$ and updated to the minimum $p_i.id$ such that $p.maxcfr{<}p_i.id$, each permission is compared to all the other permissions following it in the order established by *id*. Also, for each new permission p_i, *maxid* increases by 1 and $p_j.id$ is set to the new value of *maxid*. Since, for each permission p but p_i in *Applicable*, $p.maxcfr$ is less than $p_j.id$, the subsequent iterations of the **repeat until** loop check the new permission with all the other permissions in *Applicable*. This means that the **repeat until** loop checks all possible pairs of permissions and therefore it finds the permission authorizing the release of R.

Note also that, if a permission p_i removed from *Applicable* (because $p_j \rightarrow p_i$) authorizes R, the composed permission $p_j \otimes p_i{=}[Att_{ij}, Rels_{ij}]$ belongs to *Applicable* and authorizes the release of R. In fact, $Att_{ij}{=}(Att_i \cup Att_j) \supset (R^{\pi} \cup R^{\sigma})$. Also, $Rels_{ij}{}^{*}{=}Rels_i{}^{*}{=}R^{\bowtie *}$.

Theorem 5.3 (Complexity). *Given a relation profile* $R{=}[R^{\pi}, R^{\bowtie}, R^{\sigma}]$ *and a set Applicable of n applicable permissions, the complexity of function* **Authorized** *is* $O(n^3)$ *in time.*

Proof. The function matches every permission with every other permission in the *Applicable* set, to verify if they can be composed. Any time $p_i \rightarrow p_j$, p_i is removed from *Applicable*, while $p_i \otimes p_j$ is added to the same. Since, thanks to the ordering among permissions, no match between pairs of permissions is repeated, each per-

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: $R_l \bowtie R_r$	R_r		$[R_r^\pi, R_r^\bowtie, R_r^\sigma]$
	[S_r,NULL]	S_l: $R_l \rightarrow S_r$ S_r: $R_l \bowtie R_r$		R_l	$[R_l^\pi, R_l^\bowtie, R_l^\sigma]$
	[S_l,S_r]	S_l: $R_{J_l} := \pi_J(R_l)$ S_l: $R_{J_l} \rightarrow S_r$ S_r: $R_{J_lr} := R_{J_l} \bowtie R_r$ S_r: $R_{J_lr} \rightarrow S_l$ S_l: $R_{J_lr} \bowtie R_l$	$\pi_J(R_l) \bowtie R_r$	$\pi_J(R_l)$	$[J, R_l^\bowtie, R_l^\sigma]$ $[R_r^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	[S_r,S_l]	S_r: $R_{J_r} := \pi_J(R_r)$ S_r: $R_{J_r} \rightarrow S_l$ S_l: $R_{lJ_r} := R_l \bowtie R_{J_r}$ S_l: $R_{lJ_r} \rightarrow S_r$ S_r: $R_{lJ_r} \bowtie R_r$	$\pi_J(R_r)$	$R_l \bowtie (\pi_J(R_r))$	$[J, R_r^\bowtie, R_r^\sigma]$ $[R_l^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$

Fig. 5.13 Execution of operations and required views with corresponding profiles

mission is compared to at most $n-1$ permissions generating, at most n versions of the same. Therefore the function makes at most n^3 comparisons.

5.6 Safe Query Planning

To determine whether and how a query can be executed over the distributed system, 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.

The table in Fig. 5.13 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 (natural) join operation $R_l \bowtie R_r$, 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 specifies the operand that serves as master and the second the operand that serves as slave. We briefly 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 (i.e., needs to release) its relation to S_l, and S_l computes the join. For execution, S_l needs to hold a permission (either base or composed) authorizing it to view R_r, which has profile [R_r^π,$R_r^\bowtie$,R_r^σ].
- [S_l,S_r]: the *semi-join* requires a longer sequence of steps. First, S_l computes the projection R_{J_l} of the attributes J in its relation R_l participating in the join. Second, 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 5.3) is [J,$R_l^\bowtie$,R_l^σ]. Third, S_r locally computes $R_{J_l r}$ as the join between R_{J_l} and its relation R_r. Fourth, S_r sends $R_{J_l r}$ to S_l; this operation entails a data release characterized by the profile of $R_{J_l r}$, namely [R_r^π,$R_l^\bowtie \cup R_r^\bowtie$,$R_l^\sigma \cup R_r^\sigma$] (note that the first component contains only R_r^π, since J must be a subset of R_r^π). Fifth, S_l computes the join between $R_{J_l r}$ 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 5.3. If the join at node n_2 in the tree is executed as a regular join, S_E sends the all the tuples in `Employee` relation, restricted to attributes `SSN` and `Salary`, to S_P (or vice versa). If the join is executed as a semi-join where S_E acts as a master, S_E sends to S_P the projection of the `Employee` relation on `SSN`. S_P then sends back to S_E the `SSN` and `DoB` values in `Patient` relation joined with the list of values of `SSN` received from S_E.

A function ε_T assigns to each node n of a query tree plan $T(N_T,E_T)$ a server or a pair of servers, called *executor*, responsible for the execution of the algebraic operation represented by n. To formally capture this intuitive idea, the definition of the *executor assignment* function ε_T is introduced as follows.

Definition 5.15 (Executor assignment). Given a query tree plan $T(N_T,E_T)$, an *executor assignment function* $\varepsilon_T : N_T \rightarrow \mathscr{S} \times \{\mathscr{S} \cup \text{NULL}\}$ is an assignment of pairs of servers to nodes such that:

1. each *leaf* node (corresponding to a 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* operation 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 *master*$\in \{S_l,S_r\}$, *slave*$\in \{S_l,S_r,\text{NULL}\}$, and *master*$\neq$*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 releases allowed by the permissions.

[m,s]	**Operation/Flow**	**Views(S_l)**	**Views(S_r)**	**Views(S_t)**	**View profiles**
$[S_t,\text{NULL}]$	S_l: $R_l \rightarrow S_t$			R_l	$[R_l^\pi, R_l^\bowtie, R_l^\sigma]$
	S_r: $R_r \rightarrow S_t$			R_r	$[R_r^\pi, R_r^\bowtie, R_r^\sigma]$
	S_t: $R_l \bowtie R_r$				
$[S_t,S_r]$	S_l: $R_l \rightarrow S_t$			R_l	$[R_l^\pi, R_l^\bowtie, R_l^\sigma]$
	S_t: $R_{J_l} := \pi_J(R_l)$				
	S_t: $R_{J_l} \rightarrow S_r$		$\pi_J(R_l)$		$[J, R_l^\bowtie, R_l^\sigma]$
	S_r: $R_{J_l r} := R_{J_l} \bowtie R_r$				
	S_r: $R_{J_l r} \rightarrow S_t$			$\pi_J(R_l) \bowtie R_r$	$[R_r^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_t: $R_{J_l r} \bowtie R_l$				
$[S_t,S_l]$	S_r: $R_r \rightarrow S_t$			R_r	$[R_r^\pi, R_r^\bowtie, R_r^\sigma]$
	S_t: $R_{J_r} := \pi_J(R_r)$				
	S_t: $R_{J_r} \rightarrow S_l$	$\pi_J(R_r)$			$[J, R_r^\bowtie, R_r^\sigma]$
	S_l: $R_{J_r l} := R_l \bowtie R_{J_r}$				
	S_l: $R_{J_r l} \rightarrow S_t$			$R_l \bowtie (\pi_J(R_r))$	$[R_l^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_t: $R_{J_r l} \bowtie R_r$				
$[S_l,S_t]$	S_l: $R_{J_l} := \pi_J(R_l)$				
	S_l: $R_{J_l} \rightarrow S_t$			$\pi_J(R_l)$	$[J, R_l^\bowtie, R_l^\sigma]$
	S_r: $R_r \rightarrow S_t$			R_r	$[R_r^\pi, R_r^\bowtie, R_r^\sigma]$
	S_t: $R_{J_l r} := R_{J_l} \bowtie R_r$				
	S_t: $R_{J_l r} \rightarrow S_l$	$\pi_J(R_l) \bowtie R_r$			$[R_r^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_l: $R_{J_l r} \bowtie R_l$				
$[S_r,S_t]$	S_r: $R_{J_r} := \pi_J(R_r)$				
	S_r: $R_{J_r} \rightarrow S_t$			$\pi_J(R_r)$	$[J, R_r^\bowtie, R_r^\sigma]$
	S_l: $R_l \rightarrow S_t$			R_l	$[R_l^\pi, R_l^\bowtie, R_l^\sigma]$
	S_t: $R_{J_r l} := R_l \bowtie R_{J_r}$				
	S_t: $R_{J_r l} \rightarrow S_r$		$R_l \bowtie (\pi_J(R_r))$		$[R_l^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_r: $R_{J_r l} \bowtie R_r$				
$[S_t,S_l S_r]$	S_l: $R_{J_l} := \pi_J(R_l)$				
	S_r: $R_{J_r} := \pi_J(R_r)$				
	S_l: $R_{J_l} \rightarrow S_t$			$\pi_J(R_l)$	$[J, R_l^\bowtie, R_l^\sigma]$
	S_r: $R_{J_r} \rightarrow S_t$			$\pi_J(R_r)$	$[J, R_r^\bowtie, R_r^\sigma]$
	S_t: $R_{J_l J_r} := R_{J_l} \bowtie R_{J_r}$				
	S_t: $R_{J_l J_r} \rightarrow S_l$	$(\pi_J(R_l)) \bowtie (\pi_J(R_r))$			$[J, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_t: $R_{J_l J_r} \rightarrow S_r$		$(\pi_J(R_l)) \bowtie (\pi_J(R_r))$		$[J, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_l: $R_{J_{lr} l} := R_l \bowtie R_{J_l J_r}$				
	S_r: $R_{J_{lr} r} := R_{J_l J_r} \bowtie R_r$				
	S_l: $R_{J_{lr} l} \rightarrow S_t$			$R_l \bowtie ((\pi_J(R_l)) \bowtie (\pi_J(R_r)))$	$[R_l^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_r: $R_{J_{lr} r} \rightarrow S_t$			$((\pi_J(R_l)) \bowtie (\pi_J(R_r))) \bowtie R_r$	$[R_r^\pi, R_l^\bowtie \cup R_r^\bowtie, R_l^\sigma \cup R_r^\sigma]$
	S_t: $R_{J_{lr} l} \bowtie R_{J_{lr} r}$				

Fig. 5.14 Different strategies for executing join operation, with the intervention of a third party

Definition 5.16 (Safe assignment). Given a query tree plan $T(N_T, E_T)$ and an executor assignment function ε_T, $\varepsilon_T(n)$ is said to be *safe* when 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 releases derived by the assignment are authorized.

ε_T is said to be *safe* iff $\forall n \in N_T$, $\varepsilon_T(n)$ is safe.

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

Definition 5.17 (Feasible query plan). A query plan $T(N_T, E_T)$ is said to be *feasible* iff there exists an executor assignment function ε_T on T such that ε_T is safe.

5.6.1 Third Party Involvement

As already discussed, the execution of joins necessarily requires some communication of information among the operands, which we check against permissions (base or composed) and allow only if authorized. It may happen that, for a given join, none of the four possible modes of execution corresponds to a safe assignment. In such a case, we envision a third party can participate in the operation acting either as a proxy for one of the two operands or as a coordinator for them. Table in Fig. 5.14 summarizes the different ways in which a third party can be involved. We briefly comment them here.

- $[S_t$,NULL$]$: the third party receives the relations from the operands and independently computes the (regular) join.
- $[S_t,S_l]$ and $[S_t,S_r]$: the third party replaces S_r (S_l, respectively) in the computation with the role of master with S_l (S_r, respectively) in the role of slave.
- $[S_l,S_t]$ and $[S_r,S_t]$: the third party replaces S_r (S_l, respectively) in the computation with the role of slave with S_l (S_r, respectively) in the role of master.
- $[S_t,S_lS_r]$: the third party 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 permissions necessary to view the relation of the party for which it acts as a 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 a relation as in the case it acts as a 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 list).

The consideration of a third party requires to slightly change the executor assignment definition (Definition 5.15) which becomes as follows.

Definition 5.18 (Executor assignment - with third party). Given a query plan $T(N_T,E_T)$, an *executor assignment function* $\varepsilon_T : N_T \rightarrow \mathscr{S} \times \{\mathscr{S} \cup [\mathscr{S} \times \mathscr{S}] \cup \text{NULL}\}$ is an assignment of pairs of servers to nodes such that:

1. each *leaf* node (corresponding to a 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 $[S_l,\text{NULL}]$.

p_6:	[(SSN,Job,Salary),(Employee)] $\rightarrow S_E$
p_7:	[(SSN),(Patient)] $\rightarrow S_E$
p_8:	[(SSN,DoB,Race),(Patient)] $\rightarrow S_P$
p_9:	[(SSN,Job,Salary),(Employee,Patient)] $\rightarrow S_P$
p_{10}:	[(SSN,IdDoc,Type,Cost,Duration),(Patient,Treatment)] $\rightarrow S_P$
p_{11}:	[(SSN,IdDoc,Type,Cost,Duration),(Treatment)] $\rightarrow S_T$
p_{12}:	[(IdDoc,Name,Specialty),(Doctor)] $\rightarrow S_D$
p_{13}:	[(SSN,Type,Duration),(Treatment)] $\rightarrow S_D$
p_{14}:	[(SSN,DoB,Race),(Employee,Patient)] $\rightarrow S_D$

Fig. 5.15 An example of servers' permissions

3. each non-leaf node n, corresponding to a *join* operation on operand 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 $master \in \mathscr{S}$, $slaves \in \{\mathscr{S} \cup [S_l,S_r] \cup \text{NULL}\}$, $master \neq slave$, and at least one of the elements is in $\{S_l,S_r,[S_l,S_r],\text{NULL}\}$.

The definitions of safe assignment and feasible query plan remain unchanged.

Example 5.6. Consider the scenario of Example 5.3 and the permissions held by servers storing data in Fig. 5.15. The outer join between (`Employee`⋈`Patient`) and `Treatment` can be safely assigned neither to S_E and S_P nor to S_T. It is then necessary to resort to the intervention of a third party. Specifically, a safe assignment for the given operation is $[S_P,S_D]$. As a matter of fact, S_D is authorized to access attributes `SSN`, `Type`, and `Duration` of relation `Treatment` and attributes `SSN`, `DoB`, and `Race` from the join of `Employee` with `Patient`. S_P is authorized to view the whole `Treatment` relation, provided join condition `P.SSN=T.SSN` holds.

We can now state the problem as follows.

Problem 5.1. Given a query plan $T(N_T,E_T)$ and a set of permissions $\mathscr{P}$: *1)* determine if T is feasible and *2)* retrieve a safe assignment ε_T for it.

In the next section we illustrate an algorithm for the solution of such a problem, which exploits permissions composition technique already introduced, and given a query plan and a set of base permissions determines if the plan is feasible and, if so, returns a safe assignment for it.

5.7 Build a Safe Query Plan

The determination of the safe assignment follows two basic principles, in order to minimize the cost of computation: *i)* we favor semi-joins (in contrast to regular joins); *ii)* if more servers are candidate to safely execute a join operation (at a given level in the tree), we prefer the server that is involved in a higher number of join

```
INPUT
$\mathscr{P}$
$G(\mathscr{N},e)$
$T(N_T,E_T)$

OUTPUT
$\varepsilon_T(n)$ /* as n.executor */

/* n.left, n.right: left and right children */
/* n.operator, n.parameter: operation and its parameters */
/* [$n.\pi$,$n.\bowtie$,$n.\sigma$]: profile */
/* n.leftslave, n.rightslave: left and right slaves */
/* n.leftthirdslave: third party acting as left slave */
/* n.rightthirdslave: third party acting as right slave */
/* n.candidates: list of records of the form [server,fromchild,counter] stating candidate servers, the child
   (left, 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.slaves: executor assignment */

MAIN
FindCandidates(root(T))
AssignExecutor(root(T), NULL)
return(T)
```

Fig. 5.16 Algorithm computing a safe assignment for a query plan

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 permissions, the schema graph, and the query plan $T(N_T,E_T)$, where each leaf node (base relation R) is already assigned to executor [*server*,NULL], where *server* is the server storing the relation. It returns, if it exists, a safe assignment for T.

The algorithm works by performing two traversals of the query tree 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 Fig. 5.4) 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 unique child. If the node is a join operation, procedure **Find_candidates** calls function **Authorized** in Fig. 5.11 whenever it is necessary to verify if a particular server can act as master, slave, or can calculate a regular join. **Authorized** is called on the view graph representing the profile of the views that should be made visible in the execution of an operation. 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 incre-

```
**FINDCANDIDATES**($n$)
$l := n.left$
$r := n.right$
if $l \neq$ NULL then FindCandidates($l$)
if $r \neq$ NULL then FindCandidates($r$)
case $n.operator$ of
    $\pi$: $n.\pi := n.parameter$; $n.\bowtie := l.\bowtie$; $n.\sigma := l.\sigma$
        for $c$ in $l.candidates$ do Add [$c.server$, left, $c.count$] to $n.candidates$

    $\sigma$: $n.\pi := l.\pi$; $n.\bowtie := l.\bowtie$; $n.\sigma := l.\sigma \cup n.parameter$
        for $c$ in $l.candidates$ do Add [$c.server$, left, $c.count$] to $n.candidates$

    $\bowtie$: $n.\pi := l.\pi \cup r.\pi$; $n.\bowtie := l.\bowtie \cup r.\bowtie \cup n.parameter$; $n.\sigma := l.\sigma \cup r.\sigma$
        $right\_slave\_view := [J_l, l.\bowtie, l.\sigma]$
        $left\_slave\_view := [J_r, r.\bowtie, r.\sigma]$
        $right\_master\_view := [l.\pi \cup J_r, l.\bowtie \cup r.\bowtie \cup n.parameter, l.\sigma \cup r.\sigma]$
        $left\_master\_view := [J_l \cup r.\pi, l.\bowtie \cup r.\bowtie \cup n.parameter, l.\sigma \cup r.\sigma]$
        $right\_full\_view := [l.\pi, l.\bowtie, l.\sigma]$
        $left\_full\_view := [r.\pi, r.\bowtie, r.\sigma]$
        /* check case [$S_r$,NULL] and [$S_r$,$S_l$] */
        $n.leftslave$ := NULL
        $c$ := GetFirst($l.candidates$)
        while ($n.leftslave$=NULL)$\wedge$($c\neq$NULL) do
                if Authorized($G_{left\_slave\_view}$, $c.server$) then $n.leftslave := c$
                $c := c.next$
        $regular$ := NULL
        $rightmasters$ = NULL
        for $c$ in $r.candidates$ do
            if Authorized($G_{right\_full\_view}$, $c.server$) then Add [$c.server$, right, $c.count$+1] to $regular$
            if Authorized($G_{right\_master\_view}$, $c.server$) then Add [$c.server$, right, $c.count$+1] to $rightmasters$
        if $n.leftslave\neq$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)$\wedge$($c\neq$NULL) do
                if Authorized($G_{right\_slave\_view}$, $c.server$) then $n.rightslave := c$
                $c := c.next$
        $regular$ := NULL
        $leftmasters$ = NULL
        for $c$ in $l.candidates$ do
            if Authorized($G_{left\_full\_view}$, $c.server$) then Add [$c.server$, left, $c.count$+1] to $regular$
            if Authorized($G_{left\_master\_view}$, $c.server$) then Add [$c.server$, left, $c.count$+1] to $leftmasters$
        if $n.rightslave\neq$NULL then
          Add $leftmasters$ to $n.candidates$
        else
          Add $regular$ to $n.candidates$
        /* check third party */
        if $n.candidates$=NULL then $n.candidates$ := FindThirdParty($n$,$leftmasters$,$rightmasters$)
        /* node cannot be executed */
        if $n.candidates$=NULL then exit($n$)
```

Fig. 5.17 Function that determines the set of safe candidates for nodes in T

mented 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,

```
**ASSIGNEXECUTOR**($n$, $from\_parent$)
**if** $from\_parent \neq$ NULL **then**
  $chosen$ := **Search**($from\_parent$, $n.candidates$)
**else**
  $chosen$ := **GetFirst**($n.candidates$)
$n.executor.master$ := $chosen.server$
**case** $chosen.fromchild$ **of**
    left: /* case [$S_l$,NULL], [$S_l$,$S_r$], [$S_l$,$S_t$] */
        **if** $n.left \neq$ NULL **then AssignExecutor**($n.left$, $n.executor.master$)
        **if** $n.right \neq$ NULL **then**
          **if**  $n.rightslave \neq$ NULL **then**
                $n.executor.slaves$ := $\{n.rightslave\}$
                **AssignExecutor**($n.right$, $n.rightslave$)
          **else** $n.executor.slaves$ := $\{n.rightthirdslave\}$
                 **AssignExecutor**($n.right$, NULL)

    right: /* case [$S_r$,NULL], [$S_r$,$S_l$], [$S_r$,$S_t$] */
        **if** $n.left \neq$ NULL **then**
          **if**  $n.leftslave \neq$ NULL **then**
                $n.executor.slaves$ := $\{n.leftslave\}$
                **AssignExecutor**($n.right$, $n.leftslave$)
          **else** $n.executor.slaves$ := $\{n.leftthirdslave\}$
                **AssignExecutor**($n.right$, NULL)
        **if** $n.right \neq$ NULL **then AssignExecutor**($n.right$, $n.executor.master$)

    third_left: /* case [$S_t$,$S_r$] */
        $n.executor.slaves$ := $\{n.rightslave\}$
        **if** $n.left \neq$ NULL **then AssignExecutor**($n.left$, NULL)
        **if** $n.right \neq$ NULL **then AssignExecutor**($n.right$, $n.rightslave$)

    third_right: /* case [$S_t$,$S_l$] */
        $n.executor.slaves$ := $\{n.leftslave\}$
        **if** $n.left \neq$ NULL **then AssignExecutor**($n.left$, $n.leftslave$)
        **if** $n.right \neq$ NULL **then AssignExecutor**($n.right$, NULL)
    third: /* case [$S_t$,NULL], [$S_t$,$S_l S_r$] */
        $n.executor.slaves$ := $\{n.leftslave, n.rightslave\}$
        **if** $n.left \neq$ NULL **then AssignExecutor**($n.left$, $n.leftslave$)
        **if** $n.right \neq$ NULL **then AssignExecutor**($n.right$, $n.rightslave$)
```

Fig. 5.18 Function that chooses one candidate for each node in T

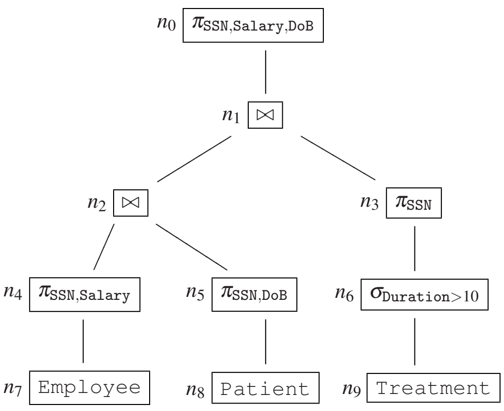
list *candidates* contains all the candidates coming from either the left or right child that can execute the join in any of the execution modes of Fig. 5.13. If no candidate was found, the algorithm determines whether the operation can be computed with the intervention of a third party by calling function **FindThirdParty** in Fig. 5.19 that similarly for the cases above, simply implements the controls according to the views that would be required for the execution (Sect. 5.6.1). If even such a call does not return any candidate, 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.

If **Find_candidates** completes successfully, the algorithm proceeds with the second traversal of the query tree plan. The second traversal (procedure **AssignExecutor**) 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 determined during the preceding post-order traversal. The other child (if existing) is pushed down the recorded candidate slave. If no slave was recorded as possible (i.e., *right-*

```
FINDTHIRDPARTY($n$,$leftmasters$,$rightmasters$)
$l$ := $n.left$; $r$ := $n.right$; $list$ := NULL
$right\_slave\_view$ := $[J_l, l.\bowtie, l.\sigma]$
$left\_slave\_view$ := $[J_r, r.\bowtie, r.\sigma]$
$right\_master\_view$ := $[l.\pi \cup J_r, l.\bowtie \cup r.\bowtie \cup n.parameter, l.\sigma \cup r.\sigma]$
$left\_master\_view$ := $[J_l \cup r.\pi, l.\bowtie \cup r.\bowtie \cup n.parameter, l.\sigma \cup r.\sigma]$
$right\_full\_view$ := $[l.\pi, l.\bowtie, l.\sigma]$
$left\_full\_view$ := $[r.\pi, r.\bowtie, r.\sigma]$
$two\_slave\_view$ := $[J_l \cup J_r, l.\bowtie \cup r.\bowtie \cup n.parameter, l.\sigma \cup r.\sigma]$
/* check if a third party can act as a slave */
if $leftmasters \neq$NULL then /* case $[S_l,S_t]$ */
  $n.rightthirdslave$ := NULL
  $i$ := 1
  while ($n.rightthirdslave$=NULL)$\wedge$($i$< $|\mathscr{S}|$) do
      if Authorized($G_{right\_slave\_view}$, $S_i$) $\wedge$ Authorized($G_{left\_full\_view}$, $S_i$) then $n.rightthirdslave$ := $S_i$
      $i$ := $i$+1
if $n.rightthirdslave \neq$NULL then
  for each $c \in leftmasters$ do Add [$c.server$, left, $c.count$] to $list$
if $rightmasters \neq$NULL then /* case $[S_r,S_t]$ */
  $n.leftthirdslave$ := NULL
  $i$ := 1
  while ($n.leftthirdslave$=NULL)$\wedge$($i$< $|\mathscr{S}|$) do
      if Authorized($G_{left\_slave\_view}$, $S_i$) $\wedge$ Authorized($G_{right\_full\_view}$, $S_i$) then $n.leftthirdslave$ := $S_i$
      $i$ := $i$+1
if $n.leftthirdslave \neq$NULL then
  for each $c \in rightmasters$ do Add [$c.server$, right, $c.count$] to $list$
if $list \neq$NULL then return($list$)
/* check if a third party can act as a master */
for $i$:=1...$|\mathscr{S}|$ do
   if $n.leftslave \neq$NULL then /* case $[S_t,S_l]$ */
     if Authorized($G_{right\_master\_view}$, $S_i$) $\wedge$ Authorized($G_{left\_full\_view}$, $S_i$) then Add [$S_i$, third_right, 1] to $list$
   else
     if $n.rightslave \neq$NULL then /* case $[S_t,S_r]$ */
        if Authorized($G_{left\_master\_view}$, $S_i$) $\wedge$ Authorized($G_{right\_full\_view}$, $S_i$) then Add [$S_i$, third_left, 1] to $list$
if $list \neq$NULL then return($list$)
/* check if a third party can execute the regular join: case [$S_t$,NULL] */
for $i$:=1...$|\mathscr{S}|$ do
   if Authorized($G_{left\_full\_view}$, $S_i$) $\wedge$ Authorized($G_{right\_full\_view}$, $S_i$) then Add [$S_i$, third, 1] to $list$
if $list \neq$NULL then return($list$)
/* check if a third party can act as a coordinator: case $[S_t,S_lS_r]$ */
$c$:= GetFirst($l.candidates$)
while ($n.leftslave$=NULL)$\wedge$($c \neq$NULL) do
     if Authorized($G_{two\_slave\_view}$, $c.server$) then $n.leftslave$ := $c.server$
       $c$ := $c.next$
if $n.leftslave \neq$NULL then
  $c$:= GetFirst($r.candidates$)
  while ($n.rightslave$=NULL)$\wedge$($c \neq$NULL) do
      if Authorized($G_{two\_slave\_view}$, $c.server$) then $n.rightslave$:= $c.server$
        $c$ := $c.next$
if $n.rightslave \neq$NULL then
  for $i$:=1...$|\mathscr{S}|$ do
    if Authorized($G_{left\_slave\_view}$, $S_i$) $\wedge$ Authorized($G_{right\_slave\_view}$, $S_i$)
      $\wedge$ Authorized($G_{left\_master\_view}$, $S_i$) $\wedge$ Authorized($G_{right\_master\_view}$, $S_i$)
      then Add $S_i$ to $masterlist$
if $masterlist \neq$NULL then for each $m \in masterlist$ do Add [$m$, third, 1] to $list$
if $list \neq$NULL then return($list$)
```

Fig. 5.19 Function that evaluates the intervention of a third party for join operations

slave/*leftslave*=NULL or the slave is a third party) a NULL value is pushed down. At each children, the master executor is determined as the server pushed down by the


n_0 $\pi_{\texttt{SSN,Salary,DoB}}$
n_1 $\bowtie$
n_2 $\bowtie$
n_3 $\pi_{\texttt{SSN}}$
n_4 $\pi_{\texttt{SSN,Salary}}$
n_5 $\pi_{\texttt{SSN,DoB}}$
n_6 $\sigma_{\texttt{Duration}>10}$
n_7 Employee
n_8 Patient
n_9 Treatment

Find_candidates

Node	Candidates	Slaves
n_7	$[S_E, \_, 0]^*$	
n_4	$[S_E$, left, 0]	
n_8	$[S_P, \_, 0]^*$	
n_5	$[S_P$, left, 0]	
n_2	$[S_P$, right, 1]	S_E
n_9	$[S_T, \_, 0]$	
n_6	$[S_T$, left, 0]	
n_3	$[S_T$, left, 0]	
n_1	$[S_P$, left, 2]	S_D
n_0	$[S_P$, left, 2]	

Assign_executor

Node	$\varepsilon_T(n)$	Calls to AssignExecutor
n_0	$[S_P, \_]$	(n_1, S_P)
n_1	$[S_P, S_D]$	(n_2, S_P) $(n_3$, NULL)
n_2	$[S_P, S_E]$	(n_4, S_E) (n_5, S_P)
n_4	$[S_E, \_]$	(n_7, S_E)
n_7	$[S_E, \_]^*$	
n_5	$[S_P, \_]$	(n_8, S_P)
n_8	$[S_P, \_]^*$	
n_3	$[S_T, \_]$	(n_6, S_T)
n_6	$[S_T, \_]$	(n_9, S_T)
n_9	$[S_T, \_]$	

Fig. 5.20 An example of execution of the algorithm in Fig. 5.16

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 5.7. Consider the query plan in Fig. 5.2 of query Q_4, reported in Fig. 5.20 for convenience, requested by Alice, who is authorized to view the query result (see composed permission $p_1 \otimes p_2 \otimes p_4$ in Fig. 5.10). Consider also the set of servers' permissions in Fig. 5.15. Figure 5.20 illustrates the working of procedures **Find_candidates** and **Assign_executor** 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, for example, the call **Find_candidates**(n_2). Among the candidates of the children (S_E from left child n_4 and S_P from right child n_5) only the right child candidate S_P survives as candidate for the join, which is executed as a semi-join since S_E can act as a slave. When **Assign_executor** 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 $[S_P,\_]$, where S_P was recorded as coming from the left (and only) child n_1, to which S_P is then pushed with a recursive call. At n_1 the master is set as S_P and, combining this with the slave field, the executor is set to $[S_P,S_D]$. Hence, S_P is further pushed down to the left child (from where it was taken by **Find_candidates**) n_3, while S_D is not pushed down to the left child n_2, since it was a third party helping in finding a correct assignment.

We conclude this section with a note regarding the integration of our approach with existing query optimizers. Optimization of distributed queries operates in *two-steps* [64]. First, the query optimizer identifies a good plan, analogous to the one it would produce for a centralized system; second, it assigns operations to the distinct servers in the system. Our algorithm nicely fits in such a two phase structure. In particular, while in the illustration of the algorithm we have assumed the complete query plan to be provided as input, we note that our algorithm could be nicely merged with the optimizers and perform its pre-order visit in conjunction with the construction of the tree by the query optimizer, computing candidates while the optimizers builds the plan, and its post-order visit for computing executors for the optimizers in the second phase.

5.8 Chapter Summary

We presented a simple, yet powerful, approach for the specification and enforcement of permissions 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 has been captured by means of profiles associated with each data computation that describe the information carried by the released relation. Allowed data releases have instead been captured by means of simple permissions, which can be efficiently composed without privacy breaches. In this chapter we presented a simple graphical representation of both permissions and profiles, allowing to easily enforce our secure chasing process. We also presented an algorithm that, given a query plan, determines whether it can be safely executed and produces a safe query planning for it. 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.