15

Specification and Enforcement of Access Policies in Emerging Scenarios

Sabrina De Capitani di Vimercati

Università degli Studi di Milano

Sara Foresti

Università degli Studi di Milano

Pierangela Samarati

Università degli Studi di Milano

15.1 Introduction

Information sharing and data dissemination are at the basis of our digital society. Users as well as companies access, disseminate, and share information with other parties to offer services, to perform distributed computations, or to simply make information of their own available. Such a dissemination and sharing process however is typically selective, and different parties may be authorized to view only specific subsets of data. Exchanges of data and collaborative computations should be controlled to ensure that authorizations are properly enforced and that information is not improperly accessed, released, or leaked. For instance, data about the patients in a hospital and stored at one provider might be selectively released only to specific providers (e.g., research institutions collaborating with the hospital) and within specific contexts (e.g., for research purposes). This situation calls for the definition of a policy specification and enforcement framework regulating information exchange and access in the interactions among parties. This problem has been under the attention of the research and development communities and several investigations have been carried out, proposing novel access control solutions for emerging and distributed scenarios. In particular, attention has been devoted to the development of powerful and flexible authorization languages and frameworks for open environments, policy composition techniques, privacy-enhanced access control and identity management solutions, policy negotiation and trust management strategies, fault tolerant policies based on user's requirements, and access control models and policies for regulating query execution in distributed multi-authority scenarios (e.g., [4][6][7][11][12][13][14][15][22] [23][26][27][28][37]). Other works have addressed the problem of private and secure multi-party computation, where different parties perform a collaborative computation learning only the query results and nothing on the inputs (e.g., [36]). In this chapter, we focus on a scenario where different parties (data owners or providers) need to collaborate and share information for performing a distributed query computation with selective disclosure of data. For the sake of simplicity, we will assume that the data stored at each provider are modeled by a relational table $r(a_1,\ldots,a_m)$, with r the name of the relation and $a_1,\ldots,a_m$ its attributes. In the following, we refer our examples to a set of four different providers, each storing one relation (Figure 15.1): Insurance company S_I with relation `Insurance`, Hospital S_P with relation `Patient`, Research Center S_T with relation `Treatment`, and a Pharmaceutical Company S_M with relation `Medicine`. In such a scenario, the problem of executing distributed query computations while ensuring that information is not improperly leaked, can be translated into the problem of producing query plans with data sharing constraints. Traditional query optimizers aim at optimizing query plans by pushing down selection and projection operations, and by choosing, for each operation in the query plan, the provider in charge of its evaluation and how the operation should be executed (e.g., they decide which join evaluation algorithm should be adopted and/or which index should be used). Query optimizers do not take into consideration possible share restrictions that data owners may wish to enforce over their data. For instance, the hospital may want to keep patients' diseases confidential and allow the insurance company to access the data of their customers only. In the definition of efficient query plans, the query optimizer should therefore consider also access privileges to guarantee that query evaluation does not imply flows of information that should be forbidden. In the remainder of this chapter, we survey the following existing approaches that address the above-mentioned problems.

S_I: `Insurance`($\underline{ssn}$, *type*, *premium*)
S_P: `Patient`($\underline{ssn}$, *name*, *dob*, *disease*)
S_T: `Treatment`($\underline{ssn}$, $\underline{mid}$, *date*, *result*)
S_M: `Medicine`($\underline{mid}$, *principle*, *auth_date*)

Figure 15.1 An example of four relations stored at four different providers

- *View-based access control*: in the relational database context, it is necessary to define authorizations that provide access to portions of the original relations. In Section 15.2, we describe solutions that address this problem by defining views, which are used to both grant access privileges to users and to enforce them at query evaluation time.
- *Access patterns*: in many scenarios (e.g., in the Web context) data sources may have limited capabilities, meaning that data can be accessed only by specifying the values for given attributes according to some patterns. In Section 15.3, we summarize approaches

that associate a profile with each relation to keep track of the attributes that should be provided as input to gain access to the data.

- *Sovereign join*: when relations are owned by different parties, the evaluation of join operations among them may reveal sensitive information to both the server in charge of the evaluation and to the two providers owning the operands. In Section 15.4, we illustrate a join evaluation strategy that reveals to the server evaluating the join neither the operands nor the result.
- *Coalition networks*: in coalition networks, different parties are aimed at sharing their data for efficiency in query evaluation while protecting data confidentiality. In Section 15.5, we describe a solution based on the definition of pairwise authorizations to selectively regulate data release.
- *User-based restrictions*: besides providers, also users may wish to define privacy restrictions in query evaluation to protect the objective of their queries to the providers' eyes. In Section 15.6, we illustrate a proposal that permits a user to specify preferences about the providers in charge of the evaluation of her queries.
- *Authorization composition and enforcement in distributed query evaluation*: in distributed scenarios where data release is selective, it is necessary to define an authorization model that, while simple, guarantees that parties cannot improperly access data. In Section 15.7, we describe an authorization model regulating the view that each provider can have on the data and illustrate an approach for composing authorizations.

15.2 View-based access control

In the relational database context, access restrictions can be defined over views that provide access to only certain portions of the underlying relations [25][31][34][35]. *Authorization views* represent a powerful and flexible mechanism for controlling what information can be accessed, and can be distinguished between traditional relational views and *parameterized views*. A parameterized view makes use of input parameters (e.g., *$user_id*, *$time*) in its conditions to possibly change the authorized subset of data depending on the execution context (e.g., the identity of the subject performing the access). *Access pattern views* are parameterized views whose parameters are bounded at access time to any value. For instance, Figure 15.2(a-c) illustrates three authorization views over the relations in Figure 15.1. The first view (*AvgPremium*) is a traditional relational view that authorizes the release of the average premium for each insurance type. The second view (*MyData*) is a parameterized view that allows each user to access her data (variable *$user_id*) in relation `Insurance`. The third view (*Customers*) is an access pattern view that allows the access to the information about treatments using medicines whose active principles are provided as input (variable *$$values*).

```
CREATE AUTH VIEW AvgPremium AS
    SELECT type, AVG(premium) AS avg
    FROM Insurance
    GROUP BY type
```

(a)

```
CREATE AUTH VIEW MyData AS
    SELECT *
    FROM Insurance
    WHERE ssn=$user_id
```

(b)

```
CREATE AUTH VIEW Customers as
    SELECT ssn, date, result
    FROM Treatment T
         JOIN Medicine M
         ON T.mid=M.mid
    WHERE M.principle IN $$values
```

(c)

```
SELECT AVG(premium)
FROM Insurance
```

(d)

Figure 15.2 An example of traditional view (a), parameterized view (b), access pattern view (c), and valid query (d)

The main disadvantage of a view-based solution is that it forces requesters (which may be final users as well as providers) to know and directly query authorization views. To overcome such a limitation, more recent models operate in an *authorization-transparent* way (e.g., [31][34][35]). These solutions permit requesters to formulate their queries over base relations. The access control system will then be in charge of checking whether such queries should be permitted or denied. Two models can be used to determine whether a query *q* satisfies the authorization views granted to the requester [25][34].

- *Truman model*: query *q* is rewritten substituting the original relations with the authorization views and base relations that the requester is authorized to access. This rewriting aims at ensuring that the requester does not obtain information that she cannot access. The advantage of this solution is that it always provides an answer to every query formulated by a requester. The drawback is that this approach may return misleading results. As an example, assume that a user is authorized to access view *MyData* and submits the query in Figure 15.2(d). Before evaluation, the query is reformulated as "SELECT AVG(*premium*) FROM *MyData*," which will return the premium of the user. The user will then have the impression that her premium is exactly equal to the average premium of all the customers of the insurance company.
- *Non-Truman model*: query *q* is subject to a *validity check* that aims at verifying whether the query can be answered using only the information contained in the authorization views and base relations that are accessible to the requester. If the query is valid, it is executed as it is without any modification. Otherwise, the query *q* is rejected. To check its validity, query *q* is compared against the authorization views of the requester. For instance, the query in Figure 15.2(d) is valid with respect to the authorization views in Figure 15.2(a-c). In fact, the query can be evaluated over view *AvgPremium*. On the

contrary, query "SELECT AVG(*premium*) FROM Insurance JOIN Patient ON I.*ssn*=P.*ssn* GROUP BY *disease*" is not valid.

View-based access control solutions have been developed for centralized scenarios, but they can be adapted to operate also in distributed database systems. However, when the diversity of the providers involved and of their views is considerable and dynamic, view-based access control approaches result limiting, since they require to explicitly define a view for each possible access need. This aspect is particularly critical in distributed scenarios, where inter-organizational collaborations occur on a daily basis, and where the heterogeneity of the providers and of their access restrictions can be high.

15.3 Access patterns

In many scenarios, data sources can be accessed only providing the values of certain attributes as input. These values are used to properly bound query results. For instance, to access data available on the web, users are often required to fill in a form that includes mandatory fields. The provider can then bound the returned data to the tuples matching the values specified in the form. As another example, a research center may be willing to share the results of the testing of medicines with a pharmaceutical company only if the company provides as input the identifier of the medicines it produces. *Access patterns* [21] are used to formally define these kinds of access restrictions, which have to be properly enforced by query evaluation engines. Each relation schema $r(a_1,\ldots,a_m)$ in a distributed database is then assigned an access pattern α, which is a string of *m* symbols, one for each attribute in the schema, as formally defined in the following.

Definition 1 **(Access Pattern)** Given a relation *r* defined over relational schema $r(a_1,\ldots,a_m)$, an *access pattern* α associated with *r*, denoted r^α, is a sequence of *m* symbols in $\{i, o\}$.

If the j-th symbol of the access pattern is i, the j-th attribute a_j in the relation schema is said to be an *input* attribute; it is an *output* attribute, otherwise. Input attributes are those that must be provided as input to gain access to a subset of tuples in relation r. Output attributes are instead not subject to constraints for access to the data. (Note that input and output attributes can also be referred as bounded and free attributes, denoted b and f, respectively.) Figure 15.3 illustrates an example of access patterns defined over the relations in Figure 15.1 where, for example, `Insurance`^{ioo}(*ssn*, *type*, *premium*) indicates that the *ssn* of customers must be provided as input to access attributes *type* and *premium* of their insurance contracts.

`Insurance`^{ioo} ($\underline{ssn}$, *type*, *premium*)
`Patient`^{iooi} ($\underline{ssn}$, *name*, *dob*, *disease*)
`Treatment`^{oioo} ($\underline{ssn}$, $\underline{mid}$, *date*, *result*)
`Medicine`^{oio} ($\underline{mid}$, *principle*, *auth_date*)

Figure 15.3 An example of access patterns

The presence of access patterns may complicate the process of query evaluation. In fact, the execution of a query q under access restrictions may require the evaluation of a *recursive query plan* where the values extracted from a relation (say r_y), which may even not be explicitly mentioned in the query itself, have to be used to access another relation (say r_x) in q. Clearly, the schema of relations r_x and r_y must include attributes characterized by the same domain (e.g., join attributes). For instance, with reference to the access patterns in Figure 15.3, the result of the projection over attribute *ssn* of relation `Treatment` can be used as input for relation `Insurance`, to obtain the plans subscribed by patients subject to a treatment.

The enforcement of access restrictions modeled by access patterns requires a revision of the traditional query evaluation strategies. In fact, classical solutions do not take into consideration the fact that query plans may need to operate recursively.

Most of the proposed solutions for the definition of query plans with access patterns consider conjunctive queries (e.g., [2][5][16][21][24][30][32]), that is, queries that include selection, projection, and join operations only and aim at identifying the tuples that satisfy all the conditions implied by the values given as input to the query. An effective (although non optimized) approach to determine a query plan that satisfies all the access restrictions operates according to the following three steps.

- Initialize a set *B* of constant values with the constant values in *q* and a local cache to the empty set.
- Iteratively access relations according to their access patterns using values in *B* and, for each accessed relation, update the cache with the tuples obtained and *B* with the corresponding values.
- Evaluate *q* over the tuples in the local cache.

For instance, consider query *q* in Figure 15.4 and the access patterns in Figure 15.3. Condition `M`.*principle*='paracetamol' provides the required input value to access the tuples in relation `Medicine` and, in particular, to extract the list of identifiers *mid* of the medicines that contain this active principle. This list of *mid* values can in turn be provided as input for accessing the tuples of interest in relation `Treatment`, which include the *ssn* of the patients treated with these medicines. The list of *ssn* values, together with value flu for attribute *disease*, finally permit to get access to the tuples in relation `Patient`, which form the result of query *q*. The above approach has been subsequently enhanced by considering, for example, run-time optimization techniques for the generation of a query plan and integrity constraints (e.g., [2][3][5][16][24][32]).

```
SELECT P.ssn, P.name, P.dob
FROM Treatment T JOIN Medicine M ON T.mid=M.mid
                 JOIN Patient P ON T.ssn=P.ssn
WHERE M.principle='paracetamol' AND P.disease='flu'
```

Figure 15.4 An example of query over relations in Figure 15.1

15.4 Sovereign joins

When operating with different relations owned by different providers, the operation that most of all may reveal sensitive information to non-authorized subjects is the *join* operation, which combines tuples from different relations. In fact, the evaluation of the join between two relations r_x and r_y reveals to the server S evaluating it the content of the two operands. In many scenarios, however, the content of the relations involved in the join operation should be kept confidential, even if the join result can possibly be revealed to the requester who submitted the query. As an example, suppose that we need to extract the collateral effects of a medicine that depend on the age of the patients treated with that medicine. However, both the hospital and the research center conducting the experimentation want (or are legally forced) to keep their own data private. *Sovereign join* [1] has been proposed as a join evaluation strategy aimed to solve this privacy issue, permitting the evaluation of join operations without revealing the operands to the server in charge of the join computation, which is assumed to not be the owner of one of the operands. The goal of sovereign join is to evaluate join operation $r_x \bowtie_J r_y$, with J an arbitrary join condition, in such a way that: *i)* only the party that requested the join can access the join result; and *ii)* no other party should be able to learn the content of relations r_x, r_y, and $r_x \bowtie_J r_y$. Sovereign join solution relies on a *secure coprocessor* located at server S, which is the only trusted component in the system. The secure coprocessor can access r_x, r_y, and the join result.

To prevent unauthorized parties, including the server S, to access the content of r_x, r_y, and of the join result, all the information flows between provider P_x (P_y, respectively) storing r_x (r_y, respectively) and S, and between S and the requester are encrypted with a key shared between the coprocessor and each of the providers owing an operand relation, and between the coprocessor and the requester.

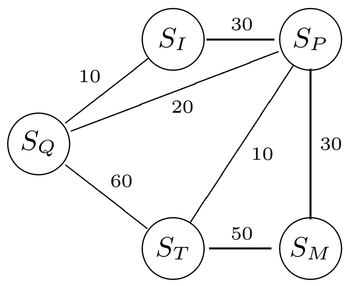
Note that even if *S* has a secure coprocessor onboard, the evaluation of the join operation should be performed carefully. In fact, secure coprocessors have limited resources and, in particular, limited memory. Hence, the join operands cannot be completely loaded in memory. The join evaluation algorithm should then guarantee that any observation of the interactions between the coprocessor and *S* (i.e., read and write operations by the coprocessors) do not reveal any information about the join operands and the result. As an example, consider the following straightforward adaptation of the traditional nested-loop algorithm for join evaluation. *S* receives from P_x and P_y the encrypted version of r_x and r_y, respectively. Iteratively, the coprocessor reads one encrypted tuple from r_x and decrypts it, obtaining t_x. For each tuple t_x, the coprocessor iteratively reads each tuple in r_y, decrypts it obtaining t_y, and checks whether it matches with t_x. If tuples t_x and t_y join, the coprocessor encrypts the pair $<t_x, t_y>$ and writes the resulting ciphertext in the join result. It then passes to the next tuple in r_y. The join evaluation terminates when all the pairs of tuples in r_x and r_y have been evaluated by the coprocessor. By observing the sequence of read and write operations, *S* (as well as any observer) can infer which encrypted tuples in r_x join with which encrypted tuples in r_y. To prevent this leakage of sensitive information, sovereign join guarantees that every join computation satisfies the following two properties:

- *fixed time*: the time for the evaluation of the join condition and for the composition of tuples is the same independently from the result;
- *fixed size*: the size of the result obtained when comparing tuples is the same independently from the result.

To guarantee the satisfaction of both these properties, the sovereign join solution adopts a variation of the nested-loop algorithm. This join computation strategy burns CPU cycles to maintain a fixed computation time, and relies on decoys (i.e., fake tuples) to maintain a fixed size of the join result. The algorithm is then designed to return an encrypted join tuple if the input tuples t_x and t_y satisfy the join condition, and an encrypted decoy of the same size, otherwise. Since decoys are indistinguishable from original tuples, server S cannot draw any inference observing information flows.

15.5 Pairwise authorizations

Emerging scenarios where data need to be exchanged and shared among different parties are represented by *coalition networks*. A coalition network is a distributed system characterized by a set of providers that wish to collaborate and share their data to reach a common goal (e.g., coalition networks often combine organizations cooperating for military, scientific, or emergency purposes) [38][39]. Each provider P in a coalition network owns one or more relations, as well as one or more servers for both computation and data storage purposes. The servers that belong to a same provider are said to be *buddies* and typically share the same privileges. A coalition network is traditionally modeled as an undirected graph $G(N,E)$ representing the corresponding overlay network among servers. Each server in the coalition network is represented by a node in N, and connections among servers are represented by weighted edges in E, where the weight of edge (S_i,S_j) represents the cost of transmitting a data unit between servers S_i and S_j. Figure 15.5(a) illustrates an example of weighted graph representing the overlay network among the servers storing the relations in Figure 15.1 and an additional server S_Q that does not store any relation and is a buddy of S_P.



Research Center $\xrightarrow{r_1=\sigma(\mathrm{T})}$ Hospital

```
r1 = SELECT *
     FROM Treatment
```

Pharmaceutical Company $\xrightarrow{r_2=\sigma(\mathrm{M})}$ Hospital

```
r2 = SELECT *
     FROM Medicine
     WHERE principle IN
           {paracetamol, antacid}
```

Research Center $\xrightarrow{r_3=\sigma(\mathrm{T})}$ Pharmaceutical Company

```
r3 = SELECT *
     FROM Treatment
     WHERE result = ‘success’
```

Pharmaceutical Company $\xrightarrow{r_4=\sigma(\mathrm{M})}$ Research Center

```
r4 = SELECT *
     FROM Medicine
     WHERE auth_data > 01/01/1995
```

(a) (b)

Figure 15.5 An example of a graph modeling a coalition network (a) and its pairwise authorizations (b)

Given a query *q*, the goal of the query optimizer is to minimize data transmission costs among the servers involved in query evaluation. For instance, consider a query that requires to join relations `Patient` (S_P), `Treatment` (S_T), and `Medicine` (S_M). A plan that minimizes data transmission costs would evaluate the join operations at server S_P. In fact, the shortest path between S_T, storing `Treatment`, and S_M, storing `Medicine`, passes through S_P, which stores `Patient`. This plan may however imply unauthorized data releases. In fact, in a coalition network not all the servers can perform all the operations in a query plan. The access control model regulating accesses to data in coalition networks must provide the data owner with the possibility to: *i)* authorize different parties for different portions of its dataset; *ii)* maintain full and autonomous control over who can access its data; and *iii)* define access control restrictions operating at tuple level. *Pairwise authorizations* satisfy all these requirements and are formally defined as follows [38].

Definition 2 (Pairwise authorization). Given two providers P_i and P_j and a relation r_i owned by P_i, a *pairwise authorization* defined by P_i over r_i is a rule of the form $P_i \xrightarrow{r_x=\sigma(r_i)} P_j$, with r_x the subset of tuples in r_i that satisfy a selection condition. A pairwise authorization $P_i \xrightarrow{r_x=\sigma(r_i)} P_j$ allows provider P_j to access a subset of the tuples in r_i, according to $\sigma(r_i)$. In fact, r_x is the result of a selection restricting the tuples visible to P_j to all and only the tuples in r_i that satisfy the selection condition. Note that all the servers belonging to P_j have the same visibility over r_i, that is, they can access the tuples granted by the pairwise authorization. A server S_j that belongs to provider P_j is then authorized to access: *i)* all the relations owned by P_j, and *ii)* the subsets of tuples of any relation r_i for which there exists a pairwise authorization $P_i \xrightarrow{r_x=\sigma(r_i)} P_j$. Server S_j can also view any subset of tuples and/or attributes in the Cartesian product among the authorized relations, also when these views are the result of the evaluation of a (sub-)query. Figure 15.5(b) illustrates an example of a set of pairwise authorizations for the coalition network in Figure 15.5(a). According to these authorizations, for example, server S_Q, which is owned by Hospital, can access relation `Patient`, relation `Treatment`, and the tuples in relation `Medicine` associated with values paracetamol and antacid for attribute *principle*. S_Q can also access the result of any query operating on these relations.

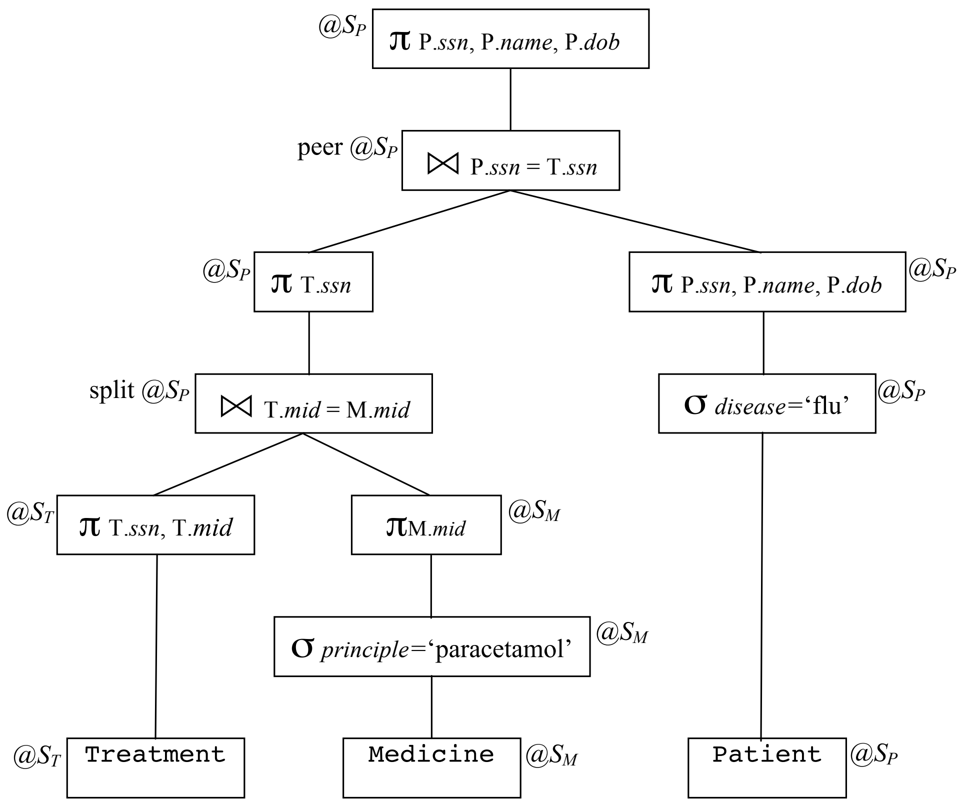
Given a query q, a coalition network $G(N,E)$, and a set of pairwise authorizations, a *safe query plan* for q has to be determined, that is, a query plan that entails only authorized data exchanges (i.e., the server receiving some data must be authorized to see them). Such a plan should also minimize data transfers, according to the costs represented by the weight of edges in G. Unary operators (i.e., selection and projection) clearly do not require data transmission for their evaluation. In fact, the server that knows the operand can evaluate the operator with no risk of violation of pairwise authorizations. Join operations may instead require the cooperation of different servers (at least the ones knowing the two operands). The server in charge of computing the join is called *master* and the server that cooperates with the master is called *slave*. The data transmitted between the two servers for the execution of the join vary depending on the specific strategy adopted. For each join in the query plan, it is important to choose the evaluation strategy that minimizes data transfers and implies only authorized flows. In the following, we summarize four join strategies (see Figure 15.6 for more details about the operations performed at each server and the corresponding information flows) that can be applied for join evaluation. For concreteness, we consider join operation $r_x \bowtie_{ax=ay} r_y$ required by server S_Q, where relations r_x and r_y are stored at S_x and S_y, respectively.

```
broker join
$S_x$: $r_x \rightarrow S_Q$
$S_y$: $r_y \rightarrow S_Q$
$S_Q$: $r_J := r_x \bowtie_{ax=ay} r_y$

peer join
$S_y$: $r_y \rightarrow S_x$
$S_x$: $r_J := r_x \bowtie_{ax=ay} r_y$
    $r_J \rightarrow S_Q$

semi-join
$S_x$: $r_{Jx} := \pi_{ax}(r_x)$
    $r_{Jx} \rightarrow S_y$
$S_y$: $r_{Jxy} := r_{Jx} \bowtie_{ax=ay} r_y$
    $r_{Jxy} \rightarrow S_x$
$S_x$: $r_J := r_{Jxy} \bowtie_{ax=ay} r_x$
    $r_J \rightarrow S_Q$

split-join
$S_x$: $r_{x1}$ := authorized tuples
    $r_{x2} := r_x - r_{x1}$
    $r_{x1} \rightarrow S_y$
$S_y$: $r_{y1}$ := authorized tuples
    $r_{y2} := r_y - r_{y1}$
    $r_{y1} \rightarrow S_x$
$S_x$: $r_{Jxy1} := r_x \bowtie_{ax=ay} r_{y1}$
    $\{r_{x2}, r_{Jxy1}\} \rightarrow S_Q$
$S_y$: $r_{Jx1y2} := r_{x1} \bowtie_{ax=ay} r_{y2}$
    $\{r_{y2}, r_{Jx1y2}\} \rightarrow S_Q$
$S_Q$: $r_{Jx2y2} := r_{x2} \bowtie_{ax=ay} r_{y2}$
    $r_J := r_{Jxy1} \cup r_{Jx1y2} \cup r_{Jx2y2}$
```

Figure 15.6 Working of the different join evaluation strategies

- *Broker-join*: both S_x and S_y send their relations to S_Q, which computes the join result. This approach can be applied independently on whether S_x, S_Q, and S_y are buddies or not.
- *Peer-join*: server S_y sends relation r_y to S_x, which computes the join and sends the result to S_Q. This approach works well when S_x and S_Q are buddies, while S_y is not. In fact, S_x and S_Q have the same privileges and therefore any result computed by S_x can always be sent to S_Q.

- *Semi-join*: servers S_x and S_y interact to compute the join result, which operates in four steps. Assuming that S_x acts as master, it first sends the projection over the join attribute of relation r_x to S_y. As a second step, S_y computes the join between the relation received from S_x and r_y, and sends the result back to S_x. In the third step, S_x computes the join between the received relation and r_x, obtaining the join result. In the fourth step, S_x sends the join result to S_Q. This approach works well when S_x and S_y are buddies as they need to exchange attributes and/or tuples of their relations.
- *Split-join*: let r_{x1} be the set of tuples in r_x that server S_y can access, and r_{y1} be the set of tuples in r_y that server S_x can access. To evaluate the join between r_x and r_y, the operation is rewritten as the union of three joins: $(r_x \bowtie_{ax=ay} r_{y1}) \cup (r_{x1} \bowtie_{ax=ay} r_{y2}) \cup (r_{x2} \bowtie_{ax=ay} r_{y2})$, with r_{x2} the set of tuples in r_x that S_y cannot access, and r_{y2} the set of tuples in r_y that S_x cannot access. The computation of the join result operates in three steps. First, S_x and S_y compute $r_x \bowtie_{ax=ay} r_{y1}$ as a peer-join with S_x acting as master. Second, S_x and S_y compute $r_{x1} \bowtie_{ax=ay} r_{y2}$ as a peer-join with S_y acting as master. Third, S_Q cooperates with both S_x and S_y and acts as a master for the evaluation of $r_{x2} \bowtie_{ax=ay} r_{y2}$ as a broker join, and computes the union of the three partial results. This approach can be applied independently on whether S_x, S_y, and S_Q are buddies or not. Then it is also suited to scenarios where S_x, S_y, and S_Q belong to three different providers.


@S_P π P.*ssn*, P.*name*, P.*dob*
peer @S_P ⋈ P.*ssn* = T.*ssn*
@S_P π T.*ssn*
π P.*ssn*, P.*name*, P.*dob* @S_P
split @S_P ⋈ T.*mid* = M.*mid*
σ *disease*='flu' @S_P
@S_T π T.*ssn*, T.*mid*
πM.*mid* @S_M
σ *principle*='paracetamol' @S_M
@S_T Treatment
Medicine @S_M
Patient @S_P

(a)

S_T: $T := \pi_{T.ssn,T.mid}$ (Treatment)
$T_{r1} := \sigma_{result='success'}$(Treatment)
$T_1 := \pi_{T.ssn,T.mid}(T_{r1})$
$T_2 := T - T_1$
$T_1 \rightarrow S_M$

S_M: $Mp := \sigma_{principle='paracetamol'}$(Medicine)
$Mp_1 := \sigma_{authdata>1/1/1995}(Mp)$
$M := \pi_{M.mid}(Mp)$
$M_1 := \pi_{M.mid}(Mp_1)$
$M_2 := M - M_1$
$M_1 \rightarrow S_T$
$TM_1 := T_1 \bowtie_{T.mid=M.mid} M_2$
$\{TM_1, T_2\} \rightarrow S_P$

S_T: $TM_2 := T \bowtie_{T.mid=M.mid} M_1$
$\{TM_2, M_2\} \rightarrow SP$

S_P: $TM := (T_2 \bowtie_{T.mid=M.mid} M_2) \cup TM_1 \cup TM_2$
$Pd := \sigma_{disease='flu'}$ (Patient)
$P := \pi_{P.ssn,P.name,P.dob}(Pd)$
$J := P \bowtie_{P.ssn=T.ssn} TM$
$Res := \pi_{P.ssn,P.name,P.dob}(J)$
$Res \rightarrow S_Q$

(b)

Figure 15. 7 An example of safe query tree plan for the query in Figure 15.4 (a) and corresponding information flow (b)

As

an example, consider the pairwise authorizations in Figure 15.5(b) and the query in Figure 15.4. Figure 15. 7(a) illustrates a safe query plan for the query, which is represented as a tree where the leaf nodes are the relations appearing in the FROM clause, and each non-leaf node corresponds to a relational operator. In this figure, the server acting as master for each operation is reported on the side of each node. The deepest join in the tree is evaluated as a split join, while the other join is evaluated as a peer join. The operations evaluated at each server and the corresponding information flows are detailed in Figure 15. 7(b).

15.6 Preferences in query optimization

Besides the parties owning the data in a distributed database system, also requesters (e.g., end users) accessing such data may be interested in specifying confidentiality requirements that the query evaluation process should take into consideration. In particular, a requester authorized to access different data sources may want to keep secret to the involved providers that she is joining their data to possibly find hidden correlations. As an example, suppose that Alice works for Hospital, which is involved in the experimentation of a new medicine, and that she suspects that this medicine has serious side effects on people suffering from diabetes. To verify her assumption, she formulates query "SELECT T.*result* FROM Treatment T JOIN Medicine M ON T.*mid*=M.*mid* JOIN Patient P ON T.*ssn*=P.*ssn* WHERE M.*principle*='expz01' AND P.*disease*= 'diabetis' ". Alice however wants to keep her intention secret from both Hospital (which may fire her) and Pharmaceutical Company (to not rise suspects). In this case, the *intension of a query* (i.e., the goal of the requester) has to be protected from the eyes of some servers [17][18][19][20][33]. The query plan may then need to satisfy constraints (i.e., requirements and preferences) specified by the requester formulating the query (e.g., certain operations cannot be revealed to, and hence also executed by, a given provider). In particular, a requester associates conditions with those portions of the query that need to be handled in a specific way during the query evaluation process. Such requirements and preferences can be effectively expressed through the following specific clauses that extend the traditional SQL syntax [19].

1. REQUIRING *condition* HOLDS OVER *node_descriptor*

   expresses a mandatory condition that must be satisfied by the query evaluation plan

2. PREFERRING *condition* HOLDS OVER *node_descriptor*

   expresses a non-mandatory condition representing user's preferences.

The *node_descriptor* is used to identify the portion of the query to which condition applies and represents a node in the query tree plan. A *node_descriptor* is a triple of the form <*operation*, *parameters*, *master*>, where *operation* is the operation represented by the node in the query plan, *parameters* are its input parameters, and *master* is the provider in charge of its evaluation. Each of the three components in a node descriptor can include a free variable (denoted with symbol @) or wild character * (representing any possible value for the corresponding element). The *condition* in a REQUIRING or PREFERRING clause imposes restrictions on the values of the free variables appearing in the *node_descriptor*. For instance, node descriptor <*, {(`Treatment`.*ssn*)}, @*p*> refers to the evaluation by an arbitrary provider @*p* of any operation over attribute *ssn* in relation `Treatment`. Condition @*p* <> S_P implies that Hospital cannot operate over the *ssn* attribute of patients who are subject to a treatment.

Both REQUIRING and PREFERRING clauses may include multiple conditions. While the conditions in the REQUIRING clause can be connected only through the AND operator and must all be satisfied, the conditions in the PREFERRING clause can be combined also using the CASCADE operator. The CASCADE operator defines a precedence among preferred conditions, thus imposing a partial order relationship among them. Consider query *q* in the example above formulated by

Alice. To prevent Hospital and Pharmaceutical Company to infer *Alice'*s intention, she can add a REQUIRING clause to her query as illustrated in Figure 15.8(a).

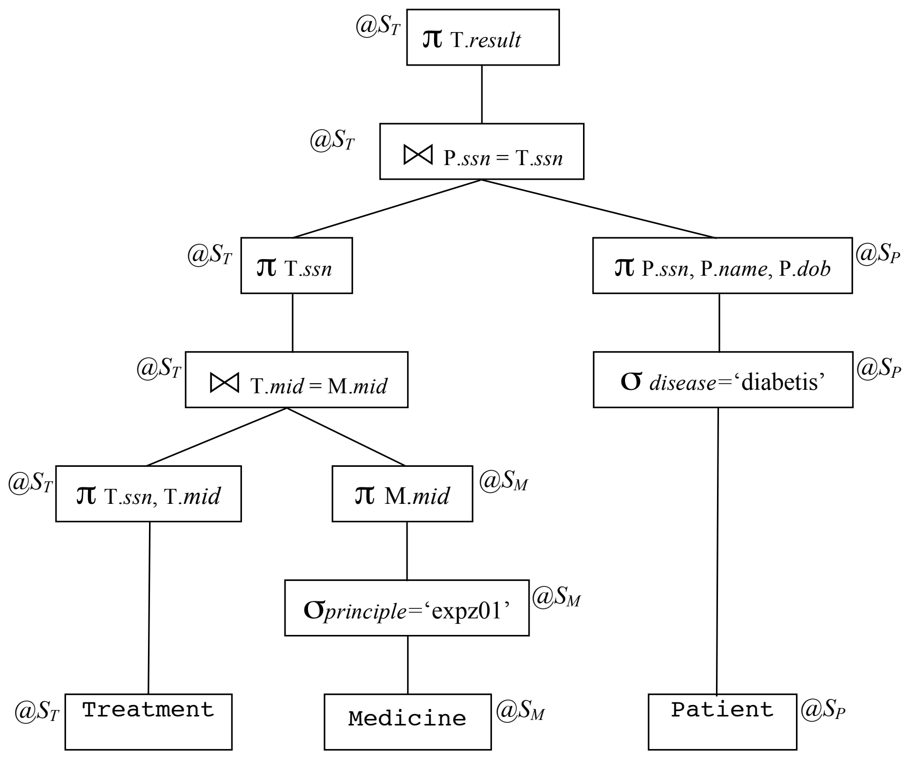
Given a query *q* including REQUIRING and/or PREFERRING clauses, the corresponding query plan has to satisfy all the mandatory conditions in the REQUIRING clause and maximize the preferences for the conditions in the PREFERRING clause. To this aim, the approach in [19] proposes to modify traditional query optimizers. The proposed solution adopts a bottom-up dynamic programming approach, which iteratively builds a safe query tree plan involving a larger subset of relations in the query at each iteration. Figure 15.8(b) illustrates a safe query tree plan for the query in Figure 15.8(b). We note that: *i)* the deepest join in the tree can only be evaluated by S_T because S_M cannot operate over attribute *mid* (as demanded by the REQUIRING clause in *q*); *ii)* the other join operation can only be evaluated by S_T because S_P cannot operate over attribute *ssn* (as demanded by the REQUIRING clause in *q*).

15.7 Collaborative query execution with multiple providers

Data stored and managed by different parties may need to be selectively shared and processed in a collaborative way to support distributed query evaluation. In this scenario, the correct definition and enforcement of access privileges ensuring that data are not improperly accessed and shared are crucial points for an effective collaboration and integration of large-scale distributed systems (e.g., [8][9][10][29]). In this section, we present an approach for collaborative distributed query execution in presence of access restrictions [8][9][10].

```
SELECT      T.result
FROM          Treatment T JOIN Medicine M ON T.mid=M.mid
            JOIN Patient P ON  T.ssn=P.ssn
WHERE         M.principle='expz01' AND P.disease='diabetis'
REQUIRING   @p <> S_P HOLDS ON <*, {(T.ssn)}, @p>
              AND @p <> S_M HOLDS ON <*, {(T.mid)}, @p>
```

(a)



(b)

Figure 15.8 An example of query with privacy preferences (a) and a corresponding safe query tree plan

15.7.1 Scenario and data model

Given a set of collaborating providers, the set of all relations they store, denoted $\mathcal{R}$, is assumed to be acyclic and lossless. Acyclicity means that the join path over any subset of the relations is unique. Lossless means that the join among relations produces only correct information. At the instance level, each relation *r* is a finite set of tuples, where each tuple *t* is a function mapping attributes to values in their domains and $t[A]$ denotes the mapping for the set A of attributes in t. Each relation r has a *primary key* and a set of *referential integrity constraints*. The primary key K of a relation $r(a_1,\ldots,a_m)$ is a subset of attributes in $\{a_1,\ldots,a_m\}$ that univocally identifies the tuples of r, meaning that there is a *functional dependency* between the primary key of a relation and all the other attributes.[1] A referential integrity constraint is a pair $<F_j,K_i>$, with F_j a subset of the attributes in relation r_j and K_i the primary key of relation r_i, stating that the set F_j of attributes, called *foreign key*, can assume only values that K_i assumes in the tuples of r_i. Notation $\mathcal{I}$ is used to denote the set of all referential integrity constraints between relations in $\mathcal{R}$.

Tuples of different relations can be combined through a *join operation*, working on the attributes with the same name, which represent the same concept in the real world. In particular, the considered approach focuses on *natural joins* where the join conditions are conjunctions of expressions of the form $a_x=a_y$, with a_x an attribute of the left operand and a_y an attribute of the right operand. In the following, the conjunction of join conditions between r_x and r_y will be represented as a pair $J=<A_x,A_y>$, with A_x (A_y, respectively) the attributes in r_x (r_y, respectively) involved in join conditions. Notation $\mathcal{J}$ will be used to denote the set of all possible joins not implied by referential integrity constraints between relations in $\mathcal{R}$. Figure 15.9 illustrates an example of referential integrity constraints and of joins defined over the relations in Figure 15.1, which have been reported in the figure for the sake of readability. A sequence of join operations that combine tuples belonging to more than two relations is called *join path* and is formally defined as follows.

[1]A functional dependency between two subsets A_i and A_j of attributes means that for each pair of tuples t_x, t_y in r such that $t_x[A_i]=t_y[A_i]$, also $t_x[A_j]=t_y[A_j]$ holds.

$\mathcal{R}$	Insurance(*ssn*, *type*, *premium*) Patient (*ssn*, *name*, *dob*, *disease*) Treatment (*ssn*, *mid*, *date*, *result*) Medicine (*mid*, *principle*, *auth_date*)
$\mathcal{I}$	<Treatment.*ssn*, Patient.*ssn*> <Treatment.*mid*, Medicine.*mid*>
$\mathcal{J}$	<Insurance.*ssn*, Patient.*ssn*>

Figure 15.9 An example of relations, referential integrity constraints, and joins

Definition 3 (Join path). Given a sequence of relations $r_1,\ldots,r_n$, a *join path* over it, denoted *joinpath*$(r_1,\ldots,r_n)$, is a sequence of n-1 joins $J_1,\ldots,J_{n-1}$ such that $\forall\, i=1,\ldots,n-1$, $J_i=<A_k,A_i> \in(\mathcal{J}\cup\mathcal{I})$, with A_k attributes in J_k, $k<i$, and A_i attributes of relation r_i.

15.7.2 Security model

The security model regulating access to data in the distributed system relies on the definition of permissions, stating which party can access which portion of the dataset, and on relation profiles, which represent the information content of relations. In the following of this section, we introduce permissions, relation profiles, and their graphical representation.

Permission

A permission defines a view over data that a given subject can access and is formally defined as follows.

Definition 4 (Permission). A *permission* is a rule of the form $[A,R]\rightarrow P$ where A is a set of attributes belonging to one or more relations, R is a set of relations such that for each attribute in A there is a relation in R including it, and P is the subject of the permission.

Permission $[A,R]\rightarrow P$ states that provider P (and hence also any server or user in its authorization domain) can view the sub-tuples over the set A of attributes belonging to the join among relations in R. Since the set $\mathcal{R}$ of relations is acyclic, the join over relations in R is unique. Note that only attribute names appear in the set A while the relations to which they belong are specified in R. This applies also to the attributes appearing in more than one relation, consistently with the fact that these attributes represent the same entity in the real word. Figure 15.10 illustrates a set of permissions for the relations in Figure 15.9. It is important to note that while the presence of a relation in the set *R* of a permission possibly implies the release of fewer tuples (only the tuples matching the join conditions are released), it does not imply the release of less information. In fact, the tuples whose release is authorized by a permission $[A,R] \rightarrow P$ implicitly give information on the fact that they satisfy the join path *joinpath*(*R*), meaning that they match tuples of other relations. For instance, permission p_5 in Figure 15.10 allows Alice to access the identifier and the authorization date of a subset of medicines used to treat patients. The inclusion of a relation *r* in the set *R* does not disclose any additional information only if there is a referential integrity constraint from a foreign key of a relation in *R* referencing attributes in *r*. For instance, permission p_2 in Figure 15.10 and a permission with the same set of attributes and the set (`Treatment`, `Patient`) of relations allows Alice to access the same information as p_2. Note also that the set *R* of relations may include relations that do not have any attribute in *A*. This may occur when a relation is needed to: *i)* build a correct association among tuples belonging to different relations (*connectivity constraint*); or *ii)* restrict the values of the attributes in *A* to only those values appearing in tuples that can be associated with such a relation (*instance-based restriction*). For instance, permission p_3 includes relation `Treatment` that is needed only to correctly associate tuples in `Patient` with tuples in `Medicine`, and permission p_5 includes relation `Treatment` that is only needed to restrict the information on released medicines.

p_1: [(*ssn*, *name*, *dob*, *disease*), (`Patient`)] $\rightarrow$ Alice
p_2: [(*ssn*, *tid*, *date*, *result*), (`Treatment`)] $\rightarrow$ Alice
p_3: [(*name*, *principle*), (`Patient`, `Treatment`, `Medicine`)] $\rightarrow$ Alice
p_4: [(*ssn*, *type*, *premium*), (`Insurance`)] $\rightarrow$ Alice
p_5: [(*mid*, *auth_date*), (`Treatment`, `Medicine`)] $\rightarrow$ Alice

Figure 15.10 Examples of permissions for the relations in Figure 15.1

Relation profile

The *relation profile* of a base or derived (i.e., computed through a query) relation *r* characterizes its information content and is necessary to determine whether a provider can access the relation. The profile of a relation *r* is a triple $[r^{\pi}, r^{\bowtie}, r^{\sigma}]$, where r^{π} is the set of attributes in *r*, $r^{\bowtie}$ is the set of relations used in the definition/construction of *r*, and r^{σ} is the set of attributes involved in the selection conditions in the definition/construction of *r*. Intuitively, the meaning of a relation profile $[r^{\pi}, r^{\bowtie}, r^{\sigma}]$ is that the base or derived relation *r* brings information on attributes in $r^{\pi} \cup r^{\sigma}$ appearing in the set $r^{\bowtie}$ of joined relations. For instance, the profile of the relation resulting from the query in Figure 15.4 is [(*ssn*, *name*, *dob*), (`Patient`, `Treatment`, `Medicine`), (*principle*, *disease*)].

Schema and view graph

A set $\mathcal{R}$ of relations can be represented through a *schema graph*, which is a mixed graph with one node for each attribute of the relations in $\mathcal{R}$, one non-oriented arc for each join in $\mathcal{J}$, one oriented arc for each referential integrity constraint in $\mathcal{I}$ and functional dependency between the key of a relation and its non-key attributes. Figure 15.11(a) illustrates the schema graph representing relations, referential integrity constraints, and joins in Figure 15.9.

Each permission $[A,R] \rightarrow P$ and each relation profile $[r^{\pi}, r^{\bowtie}, r^{\sigma}]$ can be seen as a view over $\mathcal{R}$ that

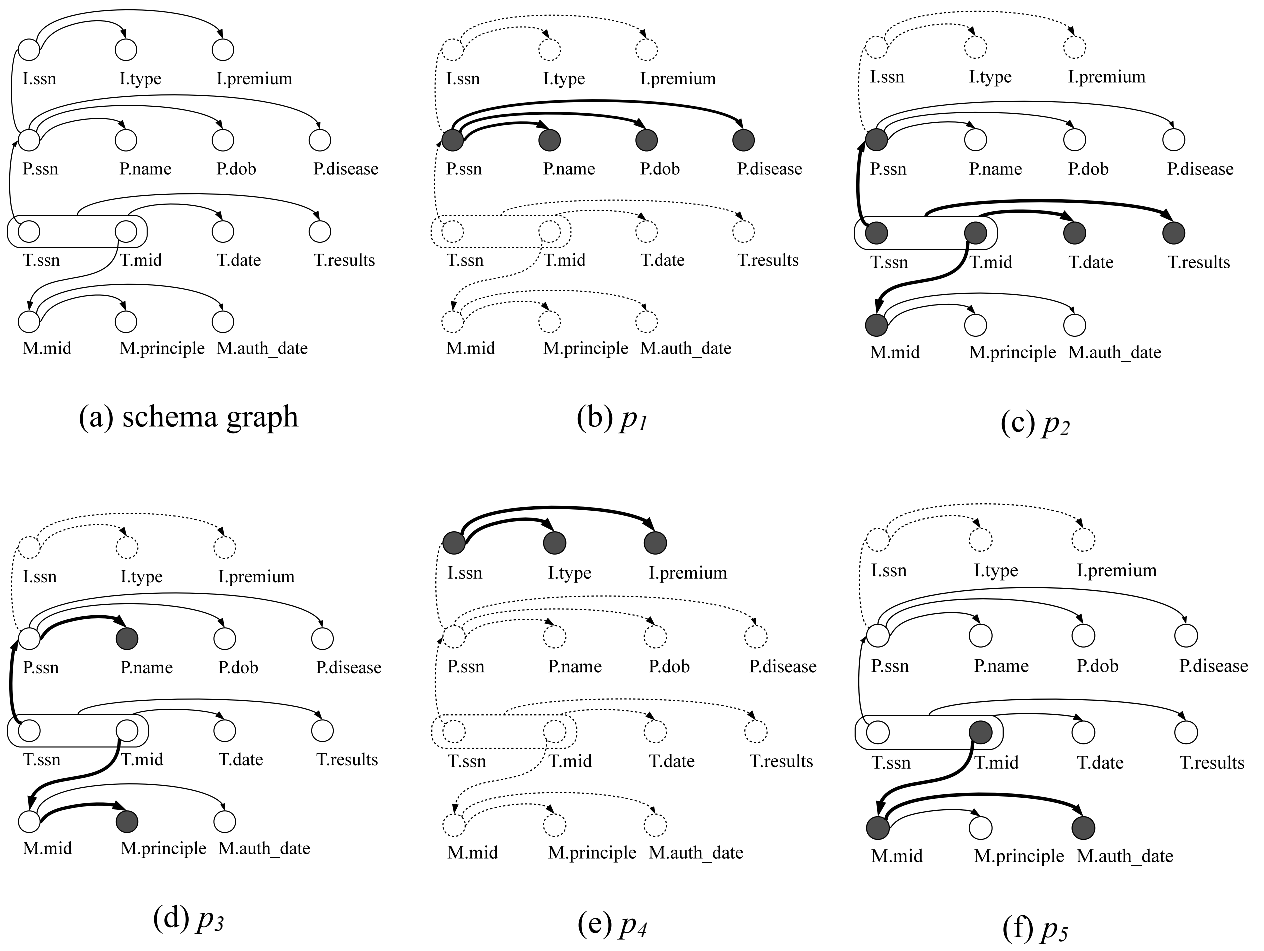


Figure 15.11 Schema graph for the relations in Figure 15.9 (a) and view graphs of the permissions in Figure 15.10(b-f)

is modeled as a pair [*Attr*, *Rel*] where: *Attr* corresponds to the attributes in the permission/relation profile (i.e, $A/r^{\pi} \cup r^{\sigma}$), and *Rel* corresponds to the relations in the permission/relation profile (i.e., $R/r^{\bowtie}$). In the characterization of views, we take into consideration the fact that the set *Rel* of relations can be extended by inserting all relations reachable from those already in *Rel* via referential integrity constraints without adding information. Given a set *R* of relations, we then denote with *R** the set of relations obtained closing *R* via the set $\mathcal{I}$ of referential integrity constraints. For instance, the closure of

R={`Treatment`} is *R**={`Treatment`, `Patient`, `Medicine`}. In fact, all the values of attribute *ssn* in `Treatment` also appear in `Patient`; analogously, all the values of attribute *mid* in `Treatment` also appear in `Medicine`.

A view *V*=[*Attr*, *Rel*] can be graphically represented as a *view graph* G_V obtained coloring the schema graph with three colors: white, black, and clear. The graph coloring is performed according to the following rules [8]: *i)* all nodes appearing in *Attr*, and all arcs belonging to *joinpath*(*Rel**) or going from the key of a relation in *Rel** to an attribute in $Attr \cup joinpath(Rel^*)$ are black; *ii)* all nodes belonging to a relation in *Rel** that are not black and all arcs going from the key of a relation in *Rel** to one of its attributes that neither belongs to *Attr* nor appears in *joinpath*(*Rel**) are white; *iii)* the remaining nodes and arcs are clear. Figure 15.11(b-f) illustrates the view graphs corresponding to the permissions in Figure 15.10. In the figure, black nodes and arcs are represented by filled nodes and bold lines, white nodes and arcs are represented by continuous nodes and lines, and clear nodes and arcs are represented by dashed nodes and lines.

15.7.3 Authorized views

Given a subject and the set $\mathcal{P}$ of her permissions, the release of a base or derived relation to her is authorized when the information directly or indirectly conveyed by the relation is included in a permission. (In the following discussion, we refer to permissions of a specific subject and therefore we omit it). The indirect information release that a relation *r* computed through a query *q* may cause is related to: *i)* the attributes used in the WHERE clause but not appearing in the SELECT clause of *q* (i.e., the attributes not appearing in *r*), which are however captured by the relation profile (r^{σ}); and *ii)* the presence of join conditions in *q* that restrict its set of tuples. A permission *p*=[*A*,*R*] authorizes the release of a relation *r* if and only if *p* includes: *i)* at least all the attributes that directly or indirectly belong to r (i.e., $(r^{\pi} \cup r^{\sigma}) \subseteq A$); and *ii)* all and only the join conditions evaluated to determine r (i.e., $R^* = r^{\bowtie *}$). Note that the set of joins (extended to consider those corresponding to referential integrity constraints) must be exactly the same for the authorizing permission and the authorized relation. This guarantees that p and r refer to the same set of tuples (i.e., the tuples belonging to the join result). As an example, consider the set of permissions in Figure 15.10 and suppose that Alice submits a query for retrieving the name of all patients. Permission p_1 authorizes the execution of the query. In terms of the view graphs, this is equivalent to say that the view graph G_r of the derived relation and the view graph G_{p1} of the permission have exactly the same black arcs among attributes in different relations, and that all nodes that are black in the view graph of the query are also black in the view graph of the permission.

Note that while a subject may not have a single permission p authorizing the release of a relation r, she may be able to compute r by joining other authorized relations. For instance, consider query "SELECT *name* FROM `Patient` JOIN `Insurance` ON `Patient`.*ssn*=`Insurance`.*ssn*". Even if no permission in Figure 15.10 authorizes Alice for this query, such a query does not provide any information that she cannot access (Alice could execute two separate queries on `Patient` and `Insurance` and join their results). The release of a relation r should therefore be allowed whenever there is a permission or a composition thereof that authorizes it. However, the *composition of permissions* has to be carefully performed to avoid that the composed permission authorizes releases that the original permissions do not authorize. In particular, two permissions p_i=[A_i,R_i] and p_j=[A_j,R_j] can be composed if and only if the join between the two corresponding views over $\mathcal{R}$ is *lossless* (i.e., the join produces a correct result w.r.t. $\mathcal{R}$), meaning (in our scenario) that the attributes in the intersection $A_i \cap A_j$ form the key of one of the two views. For instance, permissions p_1 and p_4 in Figure 15.10 can be composed because the common attribute *ssn* is the key for relation `Patient` (and also for relation `Insurance`). On the contrary, p_1 and p_3 cannot be composed, because *name* is not the key of the views corresponding to the two permissions. In terms of the view graphs, two permissions p_i and p_j can be composed if and only if there is a path of black edges from a node *n* that is black in both G_{p_i} and G_{p_j} to each black node in G_{p_i} (or to each black node in G_{p_j}). The composition of two permissions p_i=[A_i,R_i] and p_j=[A_j,R_j] is a new permission $p_i \otimes p_j$ = [$A_i \cup A_j$, $R_i \cup R_j$]. Figure 15.12 illustrates some of the permissions resulting from the composition of the permissions in Figure

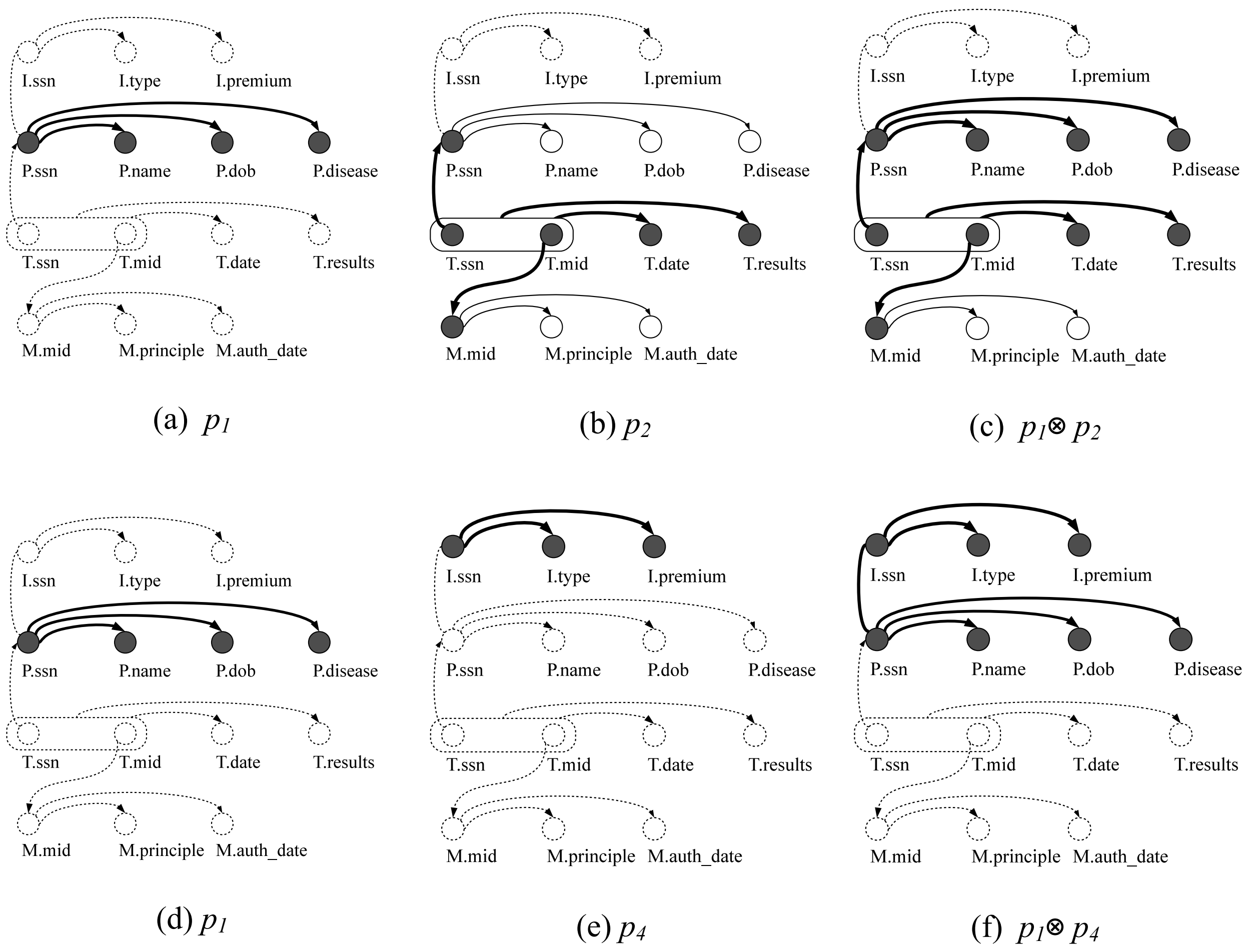


Figure 15.12 Examples of composed permissions

15.10. Note that permission $p_i \otimes p_j$ may in turn be composed with another permission p_k that could be composed with neither p_i nor p_j. Notation $\mathcal{P}^{\otimes}$ denotes the closure of $\mathcal{P}$ with respect to the composition operation. For instance, the closure of the permissions in Figure 15.10 is $\mathcal{P}^{\otimes} = \{p_1, p_2, p_3, p_4, p_5, p_1 \otimes p_2, p_1 \otimes p_4, p_2 \otimes p_4, p_2 \otimes p_5, p_1 \otimes p_2 \otimes p_4, p_1 \otimes p_2 \otimes p_5, p_1 \otimes p_2 \otimes p_4 \otimes p_5\}$. Given the set $\mathcal{P}$ of permissions granted to a subject, she is authorized for r if there is a permission p in $\mathcal{P}^{\otimes}$ that authorizes r. The work in [8] presents an efficient algorithm to verify whether a relation is authorized by a set of permissions without computing all possible compositions of permissions in $\mathcal{P}$.

15.7.4 Safe query plan

Given a query tree plan for a query q, it is necessary to assign each operation to a server responsible for its execution. Such an assignment should be safe, meaning that the server should be authorized to execute the corresponding operation. Since each server is authorized to view the relations it holds, every unary operation (i.e, selection and projection) can be executed by the server holding the relation itself. Join operations instead require cooperation between the servers that hold the relations to be joined. Given a join operation $r_x \bowtie_J r_y$, with r_x a relation of server S_x and r_y a relation of server S_y, the join can be executed as a *regular join* or as a *semi-join*. Regular join means that the slave sends to the master its relation, and then the master computes the join. Semi-join means that the master sends to the slave the projection of its relation over the attributes involved in the join, and the slave computes the join with its relation. The slave then returns the result of such join operation to the master that in turn computes the final result. Figure 15.13 summarizes the data exchanges occurring during the execution of a relational operation as well as the profile of the relation communicated at each exchange. In the figure, before each

Oper.	[m, s]	Operation/Flow	Views (S_x)	Views (S_y)	View Profiles
$\pi_X(r_x)$	$[S_x$, NULL$]$	$S_x : \pi_X(r_x)$			
$\sigma_X(r_x)$	$[S_x$, NULL$]$	$S_x : \sigma_X(r_x)$			
$r_x \bowtie_{Jxy} r_y$	$[S_x$, NULL$]$	$S_y : r_y \rightarrow S_x$ $S_x : r_x \bowtie_J r_y$	r_y		$[r_y^{\pi}, r_y^{\bowtie}, r_y^{\sigma}]$
	$[S_y$, NULL$]$	$S_x : r_x \rightarrow S_y$ $S_y : r_x \bowtie_J r_y$		r_x	$[r_x^{\pi}, r_x^{\bowtie}, r_x^{\sigma}]$
	$[S_x, S_y]$	$S_x : r_{Jx} := \pi_{Jx}(r_x)$ $S_x : r_{Jx} \rightarrow S_y$ $S_y : r_{Jxy} := r_{Jx} \bowtie_J r_y$ $S_y : r_{Jxy} \rightarrow S_x$ $S_x : r_{Jxy} \bowtie_J r_x$	$\pi_{Jx}(r_x) \bowtie_J r_y$	$\pi_{Jx}(r_x)$	$[J_x, r_x^{\bowtie}, r_x^{\sigma}]$ $[J_x \cup r_y^{\pi}, r_x^{\bowtie} \cup r_y^{\bowtie}, r_x^{\sigma} \cup r_y^{\sigma}]$
	$[S_y, S_x]$	$S_y : r_{Jy} := \pi_{Jy}(r_y)$ $S_y : r_{Jy} \rightarrow S_x$ $S_x : r_{xJy} := r_x \bowtie_J r_{Jy}$ $S_x : r_{xJy} \rightarrow S_y$ $S_y : r_{xJy} \bowtie_J r_y$	$\pi_{Jy}(r_y)$	$r_x \bowtie_J \pi_{Jy}(r_y)$	$[J_y, r_y^{\bowtie}, r_y^{\sigma}]$ $[r_y^{\pi} \cup J_y, r_x^{\bowtie} \cup r_y^{\bowtie}, r_x^{\sigma} \cup r_y^{\sigma}]$

Figure 15.13 Execution of relational operations and required views and profiles [9]

operation, we report the server S_i executing it. Column **[m,s]** reports the assignment as a pair, where the first element is the server serving as a master and the second element is the server serving as a slave. For an unary operation applied over relation *r*, the master is the server where *r* is stored and the slave is NULL. In [9] the authors present an approach that, given a query tree plan, computes a *safe assignment* (if it exists), meaning that each node of a query tree plan is assigned to a pair of servers so that there are only authorized information flows.

As an example, consider the additional permissions in Figure 15.14 and assume that Alice submits query *q* in Figure 15.4. The algorithm verifies whether Alice is authorized for the relation profile resulting from *q*. In this case, it is immediate to see that the profile of *q*, [(*ssn*, *name*, *dob*), (`Patient`, `Treatment`, `Medicine`), (*principle*, *disease*)], is authorized by the permission resulting from $p_1 \otimes p_2$ = [(*ssn*, *name*, *dob*, *disease*, *mid*, *date*, *results*), (`Patient`, `Treatment`, `Medicine`)]. The algorithm then determines a safe assignment for all operations appearing in the query tree plan. Figure 15.15 illustrates the relation profile associated with each node in the corresponding query tree plan, and a safe executor assignment for the same.

p_6:	[(*ssn*, *type*, *premium*), (`Insurance`)] → Insurance
p_7:	[(*ssn*, *name*, *dob*, *disease*),(`Patient`)] → Hospital
p_8:	[(*ssn*, *result*, *principle*), (`Patient`, `Treatment`, `Medicine`)] →Hospital
p_9:	[(*ssn*, *mid*, *date*, *result*), (`Treatment`)] → Research Center
p_{10}:	[(*mid*, *principle*, *auth_date*), (`Medicine`)] → Pharmaceutical Company
p_{11}:	[(*ssn*, *mid*, *results*), (`Treatment`)] → Pharmaceutical Company
p_{11}:	[(*ssn*), (`Patient`)] → Pharmaceutical Company

Figure 15.15 Examples of permissions for the relations in Figure 15.9

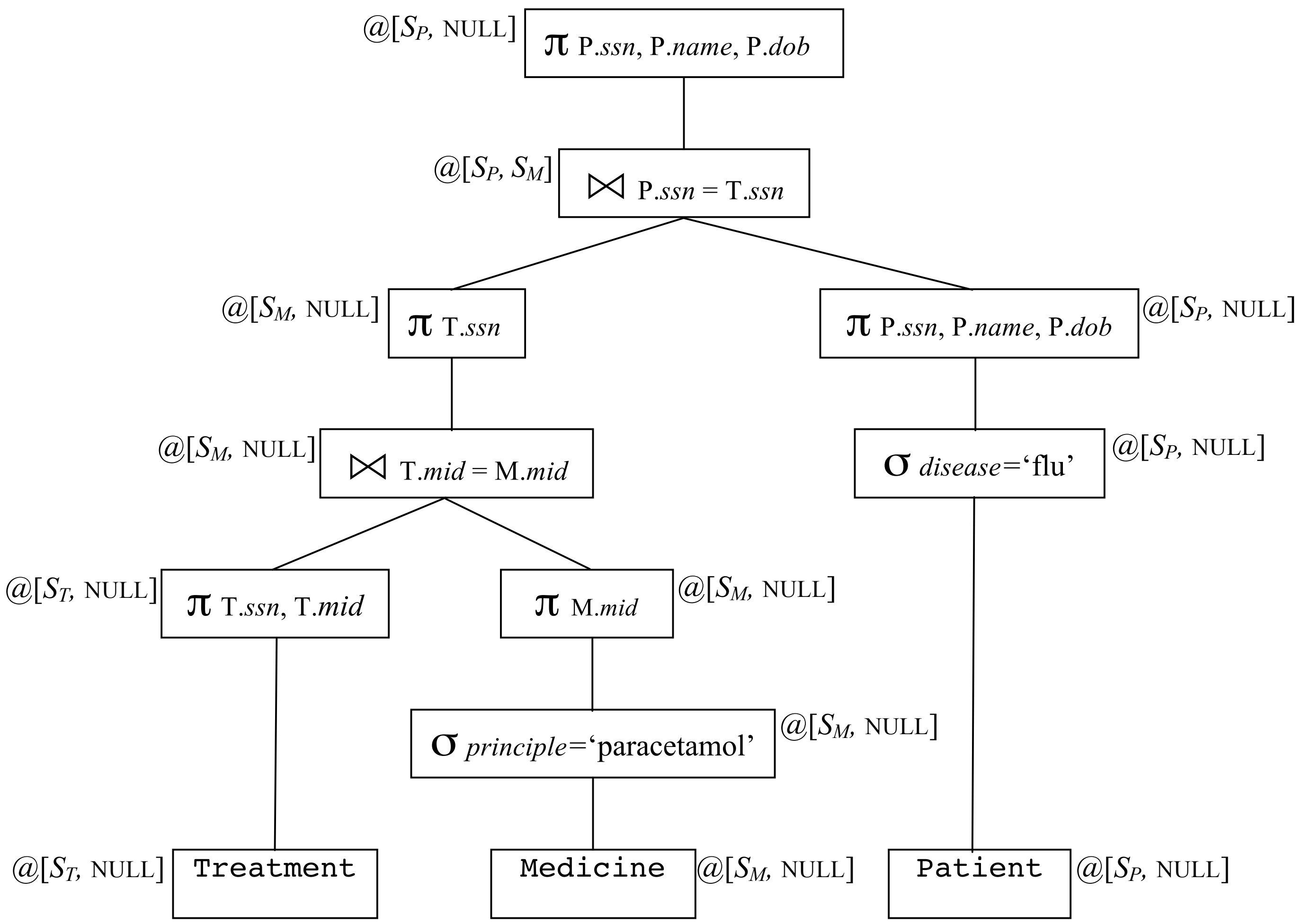

@[S_P, NULL] π P.ssn, P.name, P.dob
@[S_P, S_M] ⋈ P.ssn = T.ssn
@[S_M, NULL] π T.ssn
π P.ssn, P.name, P.dob @[S_P, NULL]
@[S_M, NULL] ⋈ T.mid = M.mid
σ disease='flu' @[S_P, NULL]
@[S_T, NULL] π T.ssn, T.mid
π M.mid @[S_M, NULL]
σ principle='paracetamol' @[S_M, NULL]
@[S_T, NULL] Treatment
Medicine @[S_M, NULL]
Patient @[S_P, NULL]

Figure 15.14 An example of a safe assignment for the query in Figure 15.4

15.8 Summary

The need of a party to share information and to cooperate with others is growing every day. This situation requires the definition of approaches for easily defining and effectively enforcing the selective sharing requirements of information stored at different providers, possibly also crossing administrative and enterprise domains. In this chapter, we have surveyed recent solutions aimed at providing effective control to data owners interested in selectively sharing their data for collaborative distributed computations. We have also illustrated approaches for defining query evaluation plans that satisfy all the restrictions to data release defined by the different collaborating parties.

Acknowledgements

This work was supported in part by the EC within the 7FP under grant agreement 312797 (ABC4EU) and within the H2020 program under grant agreement 644579 (ESCUDO-CLOUD), and by the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH).

References

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

[2] V. Barany, M. Benedikt, and P. Bourhis. "Access patterns and integrity constraints revisited." In *Proc. of the 16th International Conference on Database Theory (ICDT 2013)*, Genoa, Italy, March 2013.

[3] M. Benedikt, J. Leblay, and E. Tsamoura. "Querying with access patterns and integrity

constraints." *Proceedings of the VLDB Endowment*, 8(6):690-701, February 2015.

[4] P. Bonatti and P. Samarati. "A uniform framework for regulating service access and information release on the web." *Journal of Computer Security (JCS)*, 10(3):241-271, 2002.

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

[6] E. Damiani, S. De Capitani di Vimercati, and P. Samarati. "New paradigms for access control in open environments." In *Proc. of the 5th IEEE International Symposium on Signal Processing and Information*, Athens, Greece, December 2005.

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

[8] 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 on Computer and Communications Security (CCS 2008)*, Alexandria, VA, USA, October 2008.

[9] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. "Authorization enforcement in distributed query evaluation." *Journal of Computer Security (JCS)*, 19(4):751-794, 2011.

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

Beijing, China, June 2008.

[11] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and P. Samarati. "Access control policies and languages." *International Journal of Computational Science and Engineering (IJCSE)*, 3(2):94-102, 2007.

[12] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and P. Samarati. "Access control policies and languages in open environments." *Secure Data Management in Decentralized Systems*, T. Yu, S. Jajodia (eds.), Springer-Verlag, 2007.

[13] S. De Capitani di Vimercati and P. Samarati. "Access control in federated systems." In *Proc. of the ACM SIGSAC New Security Paradigms Workshop*, Lake Arrowhead, CA, USA, September 1996.

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

[15] S. De Capitani di Vimercati, P. Samarati, and S. Jajodia. "Policies, models, and languages for access control." In *Proc. of the Workshop on Databases in Networked Information Systems*, Aizu-Wakamatsu, Japan, March 2005.

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

[17] N.L. Farnan, A.J. Lee, P.K. Chrysanthis, and T. Yu. "Don't reveal my intension: Protecting user privacy using declarative preferences during distributed query processing." In *Proc. of the 16th European Symposium On Research In Computer Security (ESORICS 2011)*, Leuven, Belgium, September 2011.

[18] N.L. Farnan, A.J. Lee, P.K. Chrysanthis, and T. Yu. "PAQO: A preference-aware query

optimizer for PostgreSQL," *Proceedings of the VLDB Endowment*, 6(12):1334-1337, August 2013.

[19] N.L. Farnan, A.J. Lee, P.K. Chrysanthis, and T. Yu. "PAQO: Preference-aware query optimization for decentralized database systems." In *Proc. of the 30th IEEE International Conference on Data Engineering (ICDE 2014)*, Chicago, IL, USA, March-April 2014.

[20] N.L. Farnan, A.J. Lee, and T. Yu. "Investigating privacy-aware distributed query evaluation." In *Proc. of the 9th ACM Workshop on Privacy in the Electronic Society (WPES 2010)*, Chicago, IL, USA, October 2010.

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

[22] S. Foresti. *Preserving Privacy in Data Outsourcing*, Springer, 2011.

[23] M. Gamassi, V. Piuri, D. Sana, and F. Scotti. "Robust fingerprint detection for access control." In *Proc. of the Workshop RoboCare*, Rome, Italy, May 2005.

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

[25] M. Guarnieri and D. Basin. "Optimal security-aware query processing." *Proceedings of the VLDB Endowment*, 7(12):1307-1318, August 2014.

[26] R. Jhawar and V. Piuri. "Fault tolerance management in IaaS clouds." In *Proc. of the 2012 IEEE Conference in Europe about Space and Satellite Telecommunications (ESTEL 2012)*, Rome, Italy, October 2012.

[27] R. Jhawar, V. Piuri, and P. Samarati. "Supporting security requirements for resource

management in cloud computing." In *Proc. of the 2012 IEEE International Conference on Computational Science and Engineering (CSE 2012)*, Paphos, Cyprus, December 2012.

[28] R. Jhawar, V. Piuri, and M. Santambrogio. "A comprehensive conceptual system-level approach to fault tolerance in cloud computing." In *Proc. of the 2012 IEEE International Systems Conference (SysCon 2012)*, Vancouver, BC, Canada, March 2012.

[29] M. Le, K. Kant, and S. Jajodia. "Consistency and enforcement of access rules in cooperative data sharing environment." *Computers and Security*, 41:3-18, March 2014.

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

[31] A. Motro. "An access authorization model for relational databases based on algebraic manipulation of view definitions." In *Proc. of the 5th International Conference on Data Engineering (ICDE 1989)*, Los Angeles, CA, USA, February 1989.

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

[33] N.R. Ong, S.E. Rojcewicz, N.L. Farnan, A.J. Lee, P.K. Chrysanthis, and T. Yu. "Interactive preference-aware query optimization." In *Proc. of the 31st IEEE International Conference on Data Engineering (ICDE)*, Seoul, Korea, April 2015.

[34] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. "Extending query rewriting techniques for fine-grained access control." In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*, Paris, France, June 2004.

[35] A. Rosenthal and E. Sciore. "Administering permissions for distributed data: Factoring

and automated inference." In *Proc. of the 15th IFIP Annual Working Conference on Database and Application Security (DBSec 2001)*, Niagara on the Lake, Ontario, Canada, July 2001.

[36] A.C.C. Yao. "How to generate and exchange secrets." In *Proc. of the 27th Annual Symposium on Foundations of Computer Science*, Toronto, Canada, October 1986.

[37] T. Yu, M. Winslett, and K.E. Seamons. "Supporting structured credentials and sensitive policies trough interoperable strategies for automated trust." *ACM TISSEC*, 6(1):1-42, February 2003.

[38] Q. Zeng, M. Zhao, P. Liu, P. Yadav. S. Calo , and J. Lobo. "Enforcement of autonomous authorizations in collaborative distributed query evaluation." *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 27(4):979-992, April 2015.

[39] M. Zhao, P. Liu, and J. Lobo. "Towards collaborative query planning in multi-party database networks." In *Proc. of the 29th Working Conference on Data and Applications Security and Privacy (DBSec 2015)*, Fairfax, VA, USA, July 2015.