Chapter 4
Combining Fragmentation and Encryption to Protect Data Privacy[1]

Traditional solutions for granting data privacy rely on encryption. However, dealing with encrypted data makes query processing expensive. In this chapter, we propose a solution to enforce privacy over data collections combining *data fragmentation* with *encryption*. We model privacy requirements as confidentiality constraints expressing the sensitivity of the content of single attributes and of their associations. We then use encryption as an underlying (conveniently available) measure for making data unintelligible, while exploiting fragmentation to break sensitive associations among attributes. We introduce both exact and heuristic algorithms computing a fragmentation that tries to minimize the impact of fragmentation on query efficiency.

4.1 Introduction

Information is probably today the most important and valued resource. Private and governmental organizations are increasingly gathering vast amounts of data, which are collected and maintained, and often include sensitive personally identifiable information. In such a scenario guaranteeing the privacy of the data, be them stored in the system or communicated to external parties, becomes a primary requirement.

Individuals, privacy advocates, and legislators are today putting more and more attention on the support of privacy over collected information. Regulations are increasingly being established responding to these demands, forcing organizations to provide privacy guarantees over sensitive information when storing, processing or

[1] Part of this chapter appeared under V. Ciriani, S. De Capitani di Vimercati, S. Jajodia, S. Foresti, S. Paraboschi, P. Samarati, "Fragmentation and Encryption to Enforce Privacy in Data Storage," in ACM Transactions on Information and System Security (TISSEC), Vol. 13:3, July, 2010 [29] ©2010 ACM, Inc. Reprinted by permission http://doi.acm.org/10.1145/1805974.1805978; and under ©2009 IEEE, reprinted, with permission, from V. Ciriani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, "Fragmentation Design for Efficient Query Execution over Sensitive Distributed Databases," in Proc. of the 29th International Conference on Distributed Computing Systems (ICDCS 2009), Montreal, Canada, June 2009 [28].

S. Foresti, *Preserving Privacy in Data Outsourcing*, Advances in Information Security 51,
DOI 10.1007/978-1-4419-7659-8_4, © Springer Science+Business Media, LLC 2011

sharing it with others. Most recent regulations (e.g., see [22] and [78]) require that specific categories of data (e.g., data disclosing health and sex life, or data such as ZIP and date of birth that can be exploited to uniquely identify an individual [83]) to be either *encrypted* or *kept separate* from other personally identifiable information (to prevent their association with specific individuals). Information privacy guarantees may also derive from the need of preventing possible abuses of critical information. For instance, the "Payment Card Industry (PCI) Data Security Standard" [77] forces all the business organizations managing credit card information (e.g., VISA and MasterCard) to apply encryption measures when storing data. The standard also explicitly forbids the use of storage encryption as natively offered by operating systems, requiring access to the encryption keys to be separated from the operating system services managing user identities and privileges.

This demand for encryption is luckily coupled today with the fact that the realization of cryptographic functions presents increasingly lower costs in a computer architecture, where the factor limiting system performances is typically the capacity of the channels that transfer information within the system and among separate systems. Cryptography then becomes an inexpensive tool that supports the protection of privacy when storing or communicating information.

From a data access point of view, however, dealing with encrypted information represents a burden since encryption makes it not always possible to efficiently execute queries and evaluate conditions over the data. In fact, a straightforward approach to guarantee privacy to a collection of data could consist in encrypting all the data. This technique is, for example, adopted in the database outsourcing scenario [35, 55], as discussed in Chaps. 2 and 3. The assumption underlying approaches applying such an encryption wrapper is that all the data are equally sensitive and therefore encryption is a price to be paid to protect them. This assumption is typically an overkill in many scenarios. As a matter of fact, in many situations data are not sensitive per se; what is sensitive is their association with other data. As a simple example, in a hospital the list of illnesses cured or the list of patients could be made publicly available, while the association of specific illnesses to individual patients is sensitive and must be protected. Hence, there is no need to encrypt both illnesses and patients if there are alternative ways to protect the association between them.

A promising approach to protect sensitive data or sensitive associations among data is represented by the combined use of fragmentation and encryption. Fragmentation and encryption provide protection of data in storage or when disseminated ensuring no sensitive information is disclosed neither directly (i.e., present in the database) nor indirectly (i.e., derived from other information present in the database). With this design, the data can be outsourced and stored on an untrusted server, typically obtaining lower costs, greater availability, and more efficient distributed access. The advantage of having only part of the data encrypted is that all the queries that do not require to reconstruct confidential information will be managed more efficiently and securely. Also, the idea that the higher-level privilege is only used when strictly necessary represents a concrete realization of the "least privilege" principle.

We frame our work in the context of relational databases. The reason for this choice is that relational databases are by far the most common solution for the management of the data subject of privacy regulations; also, they are characterized by a clear data model and a simple query language that facilitate the design of a solution. We note, however, that our model could be easily adapted to the protection of data represented with other data models (e.g., records in files or XML documents).

As discussed in Chap. 2, the combined use of fragmentation and encryption to protect confidentiality has been initially proposed in [2], where information is stored on two separate servers and protection relies on the hypothesis that the servers cannot communicate. This assumption is clearly too strong in any practical situation. Our solution overcomes the above limitations: it allows storing data even on a single server and minimizes the amount of data represented only in encrypted format, therefore allowing for efficient query execution.

This chapter, after introducing confidentiality constraints as a simple, yet powerful, way to capture privacy requirements, presents three different approaches for the design of a fragmentation that looks carefully at performance issues. The first approach tries to minimize the number of fragments composing the solution, the second is based on the affinity between pairs of attributes, and the third exploits a complete query workload profile of the system. Then, we introduce a complete search algorithm that computes an optimal fragmentation satisfying confidentiality constraints, which can be adapted to each of the three optimization models. Also, for each cost model considered, we propose an ad hoc heuristic algorithm working in polynomial time. Our approach also manages encrypted indexes, trying to analyze the vulnerability of sensitive data due to their introduction. The experimental results support the quality of the solutions produced by the three heuristics, with respect to the result computed by the complete search strategy.

4.1.1 Chapter Outline

The remainder of the chapter is organized as follows. Section 4.2 formally defines confidentiality constraints. Sections 4.3 presents our model for enforcing confidentiality constraints by combining fragmentation and encryption. Section 4.4 introduces the definition of minimal fragmentation and shows that it is a NP-hard problem. Section 4.5 describes a complete search approach that efficiently visits the solution space lattice. Section 4.6 introduces the definition of vector-minimal fragmentation and presents a heuristic algorithm for computing a fragmentation satisfying such a definition. Section 4.7 introduces the concept of attribute affinity. Section 4.8 presents a heuristic algorithm for computing a fragmentation guided by the affinity. Section 4.9 introduces the cost model based on query workload. Section 4.10 presents an algorithm for computing a fragmentation guided by the cost of query execution. Section 4.11 illustrates how queries formulated on the original data are mapped into equivalent queries operating on fragments. Section 4.12 discusses the introduction of indexes on encrypted attributes. Section 4.13 presents the

experimental results obtained by the implementation of both complete search and heuristic algorithms. Finally, Sect. 4.14 presents our concluding remarks.

4.2 Confidentiality Constraints

We consider a scenario where, consistently with other proposals (e.g., [2, 83]) the data to be protected are represented with a single relation r over a relation schema $R(a_1,\ldots,a_n)$, containing all the information that need to be protected. For simplicity, when clear from the context, we will use R to denote either the relation schema R or the set of attributes in R (instead of using $R.*$).

We model in a quite simple and powerful way the privacy requirements through *confidentiality constraints*, which are sets of attributes, as follows.

Definition 4.1 (Confidentiality constraint). Let $\mathscr{A}$ be a set of attributes, a *confidentiality constraint* c over $\mathscr{A}$ is:

1. a singleton set $\{a\} \subset \mathscr{A}$, stating that the values of the attribute are sensitive *(attribute visibility)*; or
2. a subset of attributes in $\mathscr{A}$, stating that the association among values of the given attributes is sensitive *(association visibility)*.

While simple, a confidentiality constraint supports the definition of different confidentiality requirements that may need to be expressed, such as the following.

- *The values assumed by some attributes are considered sensitive and therefore cannot be stored in the clear*. For instance, phone numbers or email addresses can be considered sensitive values (even if not associated with any identifying information).
- *The association among values of given attributes is sensitive and therefore should not be released*. For instance, while the list of (names of) patients in a hospital as well as the list of illnesses are by themselves not confidential, the association of patient's names with illnesses is considered sensitive.

Note that constraints specified on the association among attributes can derive from different requirements: they can correspond to an association that explicitly needs protection (as in the case of names and illnesses above) or to associations that could cause inference on other sensitive information. As an example of the latter, consider a hospital database, suppose that the names of patients are considered sensitive, and therefore cannot be stored in the clear, and that the association of the `Occupation` together with the `ZIP` code can work as a quasi-identifier (i.e., `Occupation` and `ZIP` can be used, possibly in association with external information, to help identifying patients and therefore to infer, or reduce uncertainty about, their names) [30, 83]. This inference channel can be simply blocked by specifying a constraint protecting the association of the `Occupation` with the `ZIP` code. As

PATIENT

SSN	Name	Occupation	Sickness	ZIP
123-45-6789	A. Smith	Nurse	Latex al.	94140
987-65-4321	B. Jones	Nurse	Latex al.	94141
246-89-1357	C. Taylor	Clerk	Latex al.	94140
135-79-2468	D. Brown	Lawyer	Celiac	94139
975-31-8642	E. Cooper	Manager	Pollen al.	94138
864-29-7531	F. White	Designer	Nickel al.	94141

(a)

c_0={SSN}
c_1={Name,Occupation}
c_2={Name,Sickness}
c_3={Occupation,Sickness,ZIP}

(b)

Fig. 4.1 An example of plaintext relation (a) and its well defined constraints (b)

another example, consider the case where attribute Name is not considered sensitive, but its association with Sickness is. Suppose again that the Occupation together with the ZIP code can work as a quasi-identifier (then potentially leaking information on names). In this case, an association constraint will be specified protecting the association among Occupation, ZIP, and Sickness, implying that the three attributes should never be accessible together in the clear.

We are interested in enforcing a set of *well defined* confidentiality constraints, formally defined as follows.

Definition 4.2 (Well defined constraints). A set of confidentiality constraints $\mathscr{C} = \{c_1,\ldots,c_m\}$ is said to be *well defined* iff $\forall c_i, c_j \in \mathscr{C}, i \neq j$, $c_i \not\subset c_j$ and $c_j \not\subset c_i$.

According to this definition, a set of constraints $\mathscr{C}$ over $\mathscr{A}$ cannot contain a constraint that is a subset of another constraint. The rationale behind this property is that, whenever there are two constraints c_i, c_j and c_i is a subset of c_j (or vice versa), the satisfaction of constraint c_i implies the satisfaction of constraint c_j (see Sect. 4.3), and therefore c_j is redundant.

Example 4.1. Consider the Patient relation in Fig. 4.1(a), containing the information about the patients of a hospital. The privacy requirements that the hospital needs to enforce, either due to legislative or internal restrictions, are illustrated in Fig. 4.1(b):

- c_0 is a singleton constraint stating that the list of SSN of patients is considered sensitive;
- c_1 and c_2 state that the association between Name and Occupation, and the association between Name and Sickness, respectively, are considered sensitive;
- c_3 states that the association among Occupation, ZIP, and Sickness is considered sensitive (the rationale for this is that Occupation and ZIP are a quasi-identifier *[83]*).

Note that also the association of patients' Name and SSN is sensitive and should be protected. However, such a constraint is not specified since it is redundant, given that SSN by itself has been declared sensitive (c_0). As a matter of fact, protecting SSN as an individual attribute implies automatic protection of its associations with any other attribute.

4.3 Fragmentation and Encryption for Constraint Satisfaction

Our approach to satisfy confidentiality constraints is based on the use of two techniques: encryption and fragmentation.

- *Encryption.* Consistently with how the constraints are specified, encryption applies at the attribute level, that is, it involves an attribute in its entirety. Encrypting an attribute means encrypting (tuple by tuple) all its values. To protect encrypted values from frequency attacks [88], we assume that a *salt*, which is a randomly chosen value, is applied to each encryption (similarly to the use of nonces in the protection of messages from replay attacks).
- *Fragmentation.* Fragmentation, like encryption, applies at the attribute level, that is, it involves an attribute in its entirety. Fragmenting means splitting sets of attributes so that they are not visible together, that is, the associations among their values are not available without access to the encryption key.

It is straightforward to see that attribute visibility constraints can be solved only by encryption. By contrast, an association visibility constraint could be solved by either: *i)* encrypting any (one suffices) of the attributes involved in the constraint, so to prevent joint visibility, or *ii)* fragmenting the attributes involved in the constraint so that they are not visible together. Given a relation r over schema R and a set of confidentiality constraints $\mathscr{C}$ on it, our goal is to fragment R granting constraints satisfaction. However, we must also ensure that no constraint can be violated by recombining two or more fragments. In other words, there cannot be attributes that can be exploited for linking. Since encryption is differentiated by the use of the salt, the only attributes that can be exploited for linking are the plaintext attributes. Consequently, ensuring that fragments are protected from linking translates into requiring that no attribute appears in clear form in more than one fragment. In the following, we use the term *fragment* to denote any subset of a given set of attributes. A fragmentation is a set of non overlapping fragments, as captured by the following definition.

Definition 4.3 (Fragmentation). Let R be a relation schema, a *fragmentation* of R is a set of fragments $\mathscr{F}=\{F_1,\ldots,F_m\}$, where $F_i \subseteq R$, for $i = 1,\ldots,m$, such that $\forall F_i, F_j \in \mathscr{F}, i \neq j : F_i \cap F_j = \emptyset$ (fragments do not have attributes in common).

In the following, we denote with F_i^j the i-th fragment in fragmentation $\mathscr{F}_j$ (the superscript will be omitted when the fragmentation is clear from the context). For instance, with respect to the plaintext relation in Fig. 4.1(a), a possible fragmentation is $\mathscr{F}$=\{\{`Name`\},\{`Occupation`\},\{`Sickness`,`ZIP`\}\}.

At the physical level, a fragmentation translates to a combination of fragmentation and encryption. Each fragment F is mapped into a physical fragment containing all the attributes of F in the clear, while all the other attributes of R are encrypted. The reason for reporting all the original attributes (in either encrypted or clear form) in each of the physical fragments is to guarantee that any query can be executed by querying a single physical fragment (see Sect. 4.11). For the sake of simplicity and

$\hat{f}_1$

salt	enc	Name
s_1	α	A. Smith
s_2	β	B. Jones
s_3	γ	C. Taylor
s_4	δ	D. Brown
s_5	ε	E. Cooper
s_6	ζ	F. White

(a)

$\hat{f}_2$

salt	enc	Occupation
s_7	η	Nurse
s_8	θ	Nurse
s_9	ι	Clerk
s_{10}	κ	Lawyer
s_{11}	λ	Manager
s_{12}	μ	Designer

(b)

$\hat{f}_3$

salt	enc	Sickness	ZIP
s_{13}	ν	Latex al.	94140
s_{14}	ξ	Latex al.	94141
s_{15}	π	Latex al.	94140
s_{16}	ρ	Celiac	94139
s_{17}	σ	Pollen al.	94138
s_{18}	τ	Nickel al.	94141

(c)

Fig. 4.2 An example of physical fragments for the relation in Fig. 4.1(a)

efficiency, we assume that all attributes not appearing in the clear in a fragment are encrypted all together (encryption is applied on subtuples). Physical fragments are then defined as follows.

Definition 4.4 (Physical fragment). Let R be a relation schema, and $\mathscr{F}=\{F_1,\ldots,F_m\}$ be a fragmentation of R. For each $F_i=\{a_{i_1},\ldots,a_{i_n}\} \in \mathscr{F}$, the *physical fragment* of R over F_i is a relation schema $\hat{F}_i(\underline{salt},enc,a_{i_1},\ldots,a_{i_n})$, where *salt* is the primary key, *enc* represents the encryption of all the attributes of R that do not belong to the fragment, XORed (symbol $\oplus$) before encryption with the salt.

At the level of instance, given a fragment $F_i=\{a_{i_1},\ldots,a_{i_n}\}$, and a relation r over schema R, the physical fragment $\hat{F}_i$ of F_i is such that each plaintext tuple $t \in r$ is mapped into a tuple $\hat{t} \in \hat{f}_i$ where $\hat{f}_i$ is a relation over $\hat{F}_i$ and:

- $\hat{t}[enc] = E_k(t[R-F_i] \oplus \hat{t}[salt])$
- $\hat{t}[a_{i_j}] = t[a_{i_j}]$, for $j = 1,\ldots,n$

Figure 4.2 illustrates an example of physical fragments for the relation schema in Fig. 4.1(a) that does not violate the well defined constraints in Fig. 4.1(b).

The algorithm in Fig. 4.3 shows the construction and population of physical fragments. When the size of the attributes exceeds the size of an encryption block, we assume that encryption of the protected attributes uses a Cipher Block Chaining (CBC) mode [88], with the salt used as the Initialization Vector (IV); in the CBC mode, the clear text of the first block is actually encrypted after it has been combined in binary XOR with the IV. Note that the salts, which we conveniently use as primary keys of physical fragments (ensuring no collision in their generation), need not be secret, because knowledge of the salts does not help in attacking the encrypted values as long as the encryption algorithm is secure and the key remains protected.

4.4 Minimal Fragmentation

We first formally discuss the properties we require to candidate fragmentations to ensure efficient query execution.

INPUT
A relation r over schema R
$\mathscr{C} = \{c_1, \ldots, c_m\}$ /* well defined constraints */

OUTPUT
A set of physical fragments $\{\hat{F}_1, \ldots, \hat{F}_i\}$
A set of relations $\{\hat{f}_1, \ldots \hat{f}_i\}$ over schemas $\{\hat{F}_1, \ldots, \hat{F}_i\}$

MAIN
$\mathscr{C}_f := \{c \in \mathscr{C} : |c| > 1\}$ /* association visibility constraints */
$\mathscr{A}_f := \{a \in R: \{a\} \notin \mathscr{C}\}$
$\mathscr{F}$:= **Fragment**($\mathscr{A}_f$, $\mathscr{C}_f$)
/* define physical fragments */
for each $F = \{a_{i_1}, \ldots, a_{i_l}\} \in \mathscr{F}$ **do**
  define relation $\hat{F}$ with schema: $\hat{F}(\underline{salt}, enc, a_{i_1}, \ldots, a_{i_l})$
/* populate physical fragments instances */
  for each $t \in r$ **do**
    $\hat{t}[salt]$:= **GenerateSalt**(F,t)
    $\hat{t}[enc] := E_k(t[a_{j_1} \ldots a_{j_p}] \oplus \hat{t}[salt])$ /* $\{a_{j_1} \ldots a_{j_p}\} = R - F$ */
    for each $a \in F$ **do** $\hat{t}[a] := t[a]$
    insert $\hat{t}$ in $\hat{f}$

Fig. 4.3 Algorithm that correctly fragments R

4.4.1 Correctness

Given a schema R and a set of confidentiality constraints $\mathscr{C}$ on it, a fragmentation satisfies all constraints if no fragment contains in the clear all the attributes which visibility is forbidden by a constraint. The following definition formalizes this concept.

Definition 4.5 (Fragmentation correctness). Let R be a relation schema, $\mathscr{F}$ be a fragmentation of R, and $\mathscr{C}$ be a set of well defined constraints over R. $\mathscr{F}$ *correctly enforces* $\mathscr{C}$ iff $\forall F \in \mathscr{F}, \forall c \in \mathscr{C} : c \not\subseteq F$ (each individual fragment satisfies the constraints).

Note that this definition, requiring fragments not to be a superset of any constraint, implies that attributes appearing in singleton constraints do not appear in any fragment (i.e., they are always encrypted). Indeed, as already noted, singleton constraints require the attributes on which they are defined to appear only in encrypted form.

In this chapter, we specifically address the fragmentation problem and therefore focus only on the association visibility (i.e., non singleton) constraints $\mathscr{C}_f \subseteq \mathscr{C}$ and on the corresponding set $\mathscr{A}_f$ of attributes to be fragmented, defined as $\mathscr{A}_f = \{a \in R : \{a\} \notin \mathscr{C}\}$.

4.4.2 Maximal Visibility

The availability of plaintext attributes in a fragment allows an efficient execution of queries. Therefore, we aim at minimizing the number of attributes that are not

represented in the clear in any fragment, because queries using those attributes will be generally processed inefficiently. In other words, we prefer fragmentation over encryption whenever possible and always solve association constraints via fragmentation.

The requirement on the availability of a plain representation for the maximum number of attributes can be captured by imposing that any attribute not involved in a singleton constraint must appear in the clear in at least one fragment. This requirement is formally represented by the definition of maximal visibility as follows.

Definition 4.6 (Maximal visibility). Let R be a relation schema, $\mathscr{F}$ be a fragmentation of R, and $\mathscr{C}$ be a set of well defined constraints over R. $\mathscr{F}$ *maximizes visibility* iff $\forall a \in \mathscr{A}_f$: $\exists F \in \mathscr{F}$ such that $a \in F$.

Note that the combination of maximal visibility together with the definition of fragmentation (Definition 4.3) imposes that each attribute that does not appear in a singleton constraint must appear in the clear in exactly one fragment (i.e., at least for Definition 4.6, at most for Definition 4.3). In the following, we denote with $\mathfrak{F}$ the set of all possible fragmentations maximizing visibility. Therefore, we are interested in determining a fragmentation in $\mathfrak{F}$ that satisfies all the constraints in the system.

4.4.3 Minimum Number of Fragments

Another important aspect to consider when fragmenting a relation to satisfy a set of constraints is to avoid excessive fragmentation. In fact, the availability of more attributes in the clear in a single fragment allows a more efficient execution of queries on the fragment. Indeed, a straightforward approach for producing a fragmentation that satisfies the constraints while maximizing visibility is to define as many (singleton) fragments as the number of attributes not appearing in singleton constraints. Such a solution, unless demanded by the constraints, is however undesirable since it makes any query involving conditions on more than one attribute inefficient.

A simple strategy to find a fragmentation that makes query execution efficient consists in finding a *minimal fragmentation*, that is, a correct fragmentation that maximizes visibility, while minimizing the number of fragments. This problem can be formalized as follows.

Problem 4.1 (Minimal fragmentation). Given a relation schema R, a set $\mathscr{C}$ of well defined constraints over R, find a *fragmentation* $\mathscr{F}$ of R such that all the following conditions hold:

1. $\mathscr{F}$ correctly enforces $\mathscr{C}$ (Definition 4.5);
2. $\mathscr{F}$ maximizes visibility (Definition 4.6);
3. $\nexists \mathscr{F}'$ satisfying the two conditions above such that the number of fragments composing $\mathscr{F}'$ is less than the number of fragments composing $\mathscr{F}$.

The minimal fragmentation problem is *NP-hard*, as formally stated by the following theorem.

Theorem 4.1. *The minimal fragmentation problem is NP-hard.*

Proof. The proof is a reduction from the NP-hard problem of minimum hypergraph coloring [50], which can be formulated as follows: *given a hypergraph $\mathscr{H}(V,E)$, determine a minimum coloring of $\mathscr{H}$, that is, assign to each vertex in V a color such that adjacent vertices have different colors, and the number of colors is minimized*.

Given a relation schema R and a set $\mathscr{C}$ of well defined constraints, the correspondence between the minimal fragmentation problem and the hypergraph coloring problem can be defined as follows. Any vertex v_i of the hypergraph $\mathscr{H}$ corresponds to an attribute $a_i \in \mathscr{A}_f$. Any edge e_i in $\mathscr{H}$, which connects $v_{i_1},\ldots,v_{i_c}$, corresponds to a constraint c_i=$\{a_{i_1},\ldots,a_{i_c}\}$, $c_i \in \mathscr{C}_f$. A fragmentation $\mathscr{F}$=$\{F_1(a_{1_1},\ldots,a_{1_k}),\ldots,F_p(a_{p_1},\ldots,a_{p_l})\}$ of R satisfying all constraints in $\mathscr{C}$ corresponds to a solution S for the corresponding hypergraph coloring problem. Specifically, S uses p colors and $\{v_{1_1},\ldots,v_{1_k}\}$, corresponding to the attributes in F_1, are colored using the first color, vertices $\{v_{i_1},\ldots,v_{i_j}\}$, corresponding to the attributes in F_i, are colored with the i-th color, and vertices $\{v_{p_1},\ldots,v_{p_l}\}$, corresponding to the attributes in F_p, are colored using the p-th color. As a consequence, any algorithm finding a minimal fragmentation can be exploited to solve the hypergraph coloring problem.

The hypergraph coloring problem has been extensively studied in the literature, reaching interesting theoretical results. In particular, assuming $NP \neq ZPP$, there are no polynomial time approximation algorithms for coloring k-uniform hypergraphs with approximation ratio $O(n^{1-\varepsilon})$ for any fixed $\varepsilon > 0$ [60, 65].[2]

4.4.4 Fragmentation Lattice

To characterize the space of possible fragmentations and the relationships among them, we first introduce the concept of fragment vector as follows.

Definition 4.7 (Fragment vector). Let R be a relation schema, $\mathscr{C}$ be a set of well defined constraints over R, and $\mathscr{F}=\{F_1,\ldots,F_m\}$ be a fragmentation of R maximizing visibility. The *fragment vector* $V_{\mathscr{F}}$ of $\mathscr{F}$ is a vector of fragments with an element $V_{\mathscr{F}}[a]$ for each $a \in \mathscr{A}_f$, where the value of $V_{\mathscr{F}}[a]$ is the unique fragment $F_j \in \mathscr{F}$ containing attribute a.

Example 4.2. Let $\mathscr{F}$ = {{`Name`},{`Occupation`},{`Sickness`,`ZIP`}} be a fragmentation of the relation schema in Fig. 4.1(a). The fragment vector is the vector $V_{\mathscr{F}}$ such that:

- $V_{\mathscr{F}}$[`Name`]={`Name`};
- $V_{\mathscr{F}}$[`Occupation`]={`Occupation`};

[2] In a minimization framework, an approximation algorithm with approximation ratio p guarantees that the cost C of its solution is such that $C/C^* \leq p$, where C^* is the cost of an optimal solution [50]. On the contrary, we cannot perform any evaluation on the result of a heuristic.

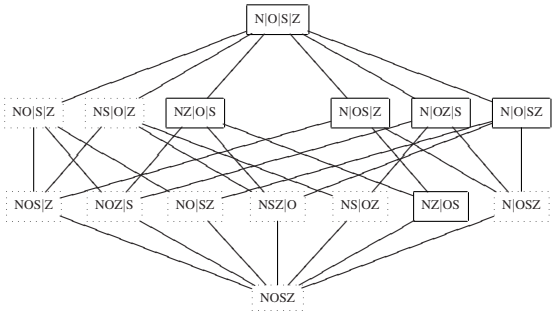


Fig. 4.4 An example of fragmentation lattice

- $V_{\mathscr{F}}[\texttt{Sickness}]=V_{\mathscr{F}}[\texttt{ZIP}]=\{\texttt{Sickness},\texttt{ZIP}\}$.

Fragment vectors allow us to define a partial order between fragmentations as follows.

Definition 4.8 (Dominance). Let R be a relation schema, $\mathscr{C}$ be a set of well defined constraints over R, and $\mathscr{F}$, $\mathscr{F}'$ be two fragmentations of R maximizing visibility. We say that $\mathscr{F}'$ *dominates* $\mathscr{F}$, denoted $\mathscr{F} \preceq \mathscr{F}'$, iff $V_{\mathscr{F}}[a] \subseteq V_{\mathscr{F}'}[a]$, $\forall\, a \in \mathscr{A}_f$. We say $\mathscr{F} \prec \mathscr{F}'$ iff $\mathscr{F} \preceq \mathscr{F}'$ and $\mathscr{F} \neq \mathscr{F}'$.

Definition 4.8 states that fragmentation $\mathscr{F}'$ dominates fragmentation $\mathscr{F}$ if $\mathscr{F}'$ can be computed from $\mathscr{F}$ by merging two (or more) fragments composing $\mathscr{F}$.

Example 4.3. Let $\mathscr{F}_1=\{\{\texttt{Name},\texttt{ZIP}\}, \{\texttt{Occupation},\texttt{Sickness}\}\}$ and $\mathscr{F}_2=\{\{\texttt{Name}\}, \{\texttt{Occupation},\texttt{Sickness}\}, \{\texttt{ZIP}\}\}$ be two fragmentations of the relation schema in Fig. 4.1(a). Since $\mathscr{F}_1$ can be obtained from $\mathscr{F}_2$ by merging fragments $\{\texttt{Name}\}$ and $\{\texttt{ZIP}\}$, it results that $\mathscr{F}_2 \prec \mathscr{F}_1$.

The set $\mathfrak{F}$ of all possible fragmentations maximizing visibility, together with the dominance relationship just introduced, form a *lattice*, as formally stated in the following definition.

Definition 4.9 (Fragmentation lattice). Let R be a relation schema, and $\mathscr{C}$ be a set of well defined constraints over R. The *fragmentation lattice* is a pair $(\mathfrak{F}, \preceq)$, where $\mathfrak{F}$ is the set of all fragmentations of R maximizing visibility and $\preceq$ is the dominance relationship among them, as defined in Definition 4.8.

The top element $\mathscr{F}_\top$ of the lattice represents a fragmentation where each attribute in $\mathscr{A}_f$ appears in a different fragment. The bottom element $\mathscr{F}_\bot$ of the lattice represents a fragmentation composed of a single fragment containing all attributes in $\mathscr{A}_f$. As an example, Fig. 4.4 illustrates the fragmentation lattice for the example in Fig. 4.1, with $\mathscr{A}_f=\{\texttt{Name}, \texttt{Occupation}, \texttt{Sickness}, \texttt{ZIP}\}$. Here, attributes are represented with their initials and fragments are divided by a vertical line. Furthermore, fragmentations that correctly enforce (Definition 4.5) constraints

in Fig. 4.1(b) appear as solid boxes, while fragmentations that violate at least a constraint appear as dotted boxes.

An interesting property of the fragmentation lattice is that given a non correct fragmentation $\mathscr{F}_i$, any fragmentation $\mathscr{F}_j$ such that $\mathscr{F}_j \preceq \mathscr{F}_i$ is non correct.

Theorem 4.2. *Given a fragmentation lattice* $(\mathfrak{F}, \preceq)$, $\forall \mathscr{F}_i, \mathscr{F}_j \in \mathfrak{F}$ *such that* $\mathscr{F}_j \preceq \mathscr{F}_i$, $\mathscr{F}_i$ *non correct* $\Longrightarrow$ $\mathscr{F}_j$ *non correct.*

Proof. If $\mathscr{F}_i$ is not correct, then $\exists c \in \mathscr{C}_f$ and $\exists F^i \in \mathscr{F}_i$ such that $c \subseteq F^i$. Since $\mathscr{F}_j \preceq \mathscr{F}_i$, by Definition 4.8, $\exists F^j \in \mathscr{F}_j$ such that $F^i \subseteq F^j$. Then $c \subseteq F^i \subseteq F^j$, and $\mathscr{F}_j$ is not correct.

By construction, each path in the lattice is characterized by a *locally minimal fragmentation*, which is the fragmentation such that all its descendants in the path correspond to non correct fragmentations. Intuitively, such locally minimal fragmentations can be determined either via a top-down visit or via a bottom-up visit of the lattice. The number of fragmentations at level i (i.e., the solutions composed of $(n-i)+1$ fragments) of the lattice is $\left\{ {n \atop n-i} \right\}$, which is the *number of Stirling of the second kind* [53]. As a consequence, $|\mathfrak{F}| = \sum_i = 0^n \left\{ {n \atop n-i} \right\} = B_n$, which is the *Bell number* [53]. The second level of the lattice then contains a quadratic number of solutions ($O(n^2)$), and an exponential number of fragmentations ($O(2^n)$) resides in the first to last level. The top-down strategy, exploiting the fact that the number of fragments increases while going down in the lattice, seems then to be more convenient. In the following section, we then propose an exact algorithm that performs a top-down tree traversal of the lattice (i.e., each fragmentation is visited at most once) and that generates only a subset of all possible fragmentations.

4.5 A Complete Search Approach to Minimal Fragmentation

Although the number of possible fragmentations in $\mathfrak{F}$ is exponential in $|\mathscr{A}_f|$, the set of attributes to be fragmented is usually limited in size and therefore a complete search evaluating the different fragmentations maximizing visibility could be acceptable. To ensure the evaluation of each correct fragmentation maximizing visibility exactly once, we define a *fragmentation tree* as follows.

Definition 4.10 (Fragmentation tree). Let $(\mathfrak{F}, \preceq)$ be a fragmentation lattice. A *fragmentation tree* of the lattice is a spanning tree of $(\mathfrak{F}, \preceq)$ rooted in $\mathscr{F}_\top$.

We propose here a method for building a fragmentation tree over a given fragmentation lattice. To this aim, we assume the set $\mathscr{A}_f$ of attributes to be totally ordered, according to a relationship, denoted $<_A$, and assume that in each fragment F attributes are maintained ordered, from the smallest, denoted *F.first*, to the greatest, denoted *F.last*. We then translate the order relationship among attributes into an order relationship among fragments within a fragmentation, by considering fragments to be ordered according to the order dictated by their smallest (*.first*) attribute.

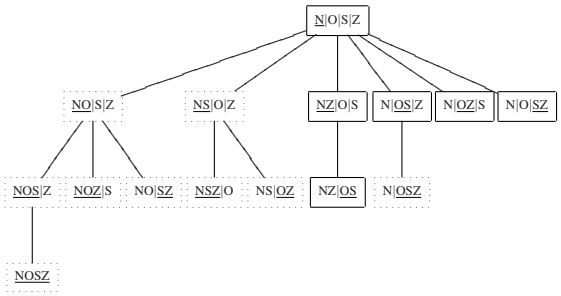


Fig. 4.5 A fragmentation tree for the fragmentation lattice in Fig. 4.4

Since, within a fragmentation, each attribute appears in exactly one fragment, the fragments in each fragmentation are totally ordered. Each fragmentation $\mathscr{F}$ is then a sequence, denoted $\mathscr{F} = [F_1, \ldots, F_n]$, of fragments, where $\forall i, j = 1, \ldots, n : i < j$, $F_i.first <_A F_j.first$. In this case, we say that fragment F_i *precedes* fragment F_j in fragmentation $\mathscr{F}$. Given two fragments F_i, F_j with $i < j$, we say that F_i *fully precedes* F_j iff all attributes of F_i are smaller than all attributes in F_j, that is, $F_i.last <_A F_j.first$. Note that *full precedence* is only a partial ordering.

To ensure tree traversal and therefore to avoid computing a fragmentation twice, we exploit the precedence relationship among fragments and associate with each fragmentation $\mathscr{F} = [F_1, \ldots, F_n]$ a *marker* F_i that is the non singleton fragment such that $\forall j > i$, F_j is a singleton fragment. For the root, the marker is its first fragment. Intuitively, the marker associated with a fragmentation denotes the starting point for fragments to be combined to obtain children of the fragmentation (as a combination with any fragment preceding it will produce duplicate fragmentations). We then define an order-based cover for the lattice as follows.

Definition 4.11 (Order-based cover). Let $(\mathfrak{F}, \preceq)$ be a fragmentation lattice. An *order-based cover* of the lattice, denoted $\mathscr{T}(V, E)$, is an oriented graph, where $V = \mathfrak{F}$, and $\forall \mathscr{F}_p, \mathscr{F}_c \in V$, $(\mathscr{F}_p, \mathscr{F}_c) \in E$ iff, being F_m^p the marker of $\mathscr{F}_p$, there exists i, j with $m \leq i$ and F_i^p fully preceding F_j^p, such that:

- $\forall l < j, l \neq i, F_l^c = F_l^p$;
- $F_i^c = F_i^p F_j^p$;
- $\forall l \geq j, F_l^c = F_{l+1}^p$.

As an example, consider the order-based cover in Fig. 4.5, where $<_A$ is the lexicographic order. It is built on the fragmentation lattice in Fig. 4.4 and the underlined fragments are the markers. Given fragmentations $\mathscr{F}_p$=[N|O|S|Z] and $\mathscr{F}_c$=[N|OS|Z], edge $(\mathscr{F}_p, \mathscr{F}_c)$ belongs to $\mathscr{T}$ since for $i = 2$ and $j = 3$ we have that F_1^c=F_1^p=N; F_2^c=$F_2^p F_3^p$=OS; and F_3^c=F_{3+1}^p=Z. The order-based cover so defined corresponds to a fragmentation tree for the lattice, as stated by the following theorem.

Theorem 4.3. *The order-based cover $\mathscr{T}$ of a lattice $(\mathfrak{F}, \preceq)$ is a fragmentation tree for $(\mathfrak{F}, \preceq)$ with root $\mathscr{F}_\top$.*

Proof. $\mathcal{T}$ is a fragmentation tree for $(\mathfrak{F}, \preceq)$ if: (1) each vertex at level i (but the root $\mathcal{F}_\top$) has *exactly one* parent at level $i-1$, and (2) *each edge of* $\mathcal{T}$ *is an edge in* $(\mathfrak{F}, \preceq)$.

1. *Each vertex has at most one parent.* Suppose, by contradiction, that a vertex $\mathcal{F}$=[F_1,...,F_{n-1}] is a child of two different vertices in $\mathcal{T}$, say $\mathcal{F}_1$=[F_1^1,...,F_n^1] and $\mathcal{F}_2$=[F_1^2,...,F_n^2]. Therefore, there exists a fragment F_{i_1} in $\mathcal{F}$ obtained as $F_{i_1}^1 F_{j_1}^1$. Analogously, there exists a fragment F_{i_2} in $\mathcal{F}$ obtained as $F_{i_2}^2 F_{j_2}^2$.
   Suppose also, without loss of generality, $i_1 < i_2$. By Definition 4.11, for each F_k in $\mathcal{F}$, $k \neq i_1$, there exists a fragment $F_{k_1}^1$ in $\mathcal{F}_1$ such that $F_{k_1}^1$=F_k and $k_1 \geq k$ (either $k_1 = k$ or $k_1 = k+1$). Therefore, there exists a non singleton fragment F_l^1=F_{i_2} with $l \geq i_2$. As a consequence, $l > i_1$, thus the marker for $\mathcal{F}_1$ must be greater than or equal to i_1, by definition. This generates the contradiction.
   Each vertex has at least one parent. Let $\mathcal{F}$ be a vertex at level i ($i \neq 1$) in $\mathcal{T}$ ($\mathcal{F} \neq \mathcal{F}_\top$), F_m be its marker, and F_m.*last* be the highest attribute in F_m. Consider fragmentation $\mathcal{F}_p$, containing all the fragments in $\mathcal{F}$ but F_m and the two fragments obtained by splitting F_m into $F_m - \{F_m.last\}$ and $\{F_m.last\}$. The marker of $\mathcal{F}_p$ precedes m, since all the fragments following F_m in $\mathcal{F}$ are singleton in $\mathcal{F}_p$ as well. Also, the additional fragment $\{F_m.last\}$ is singleton and it follows F_m^p, according to relationship $<_A$ (since it is the maximum attribute). Therefore, by Definition 4.11, there is an edge $(\mathcal{F}_p, \mathcal{F})$ in $\mathcal{T}$, then $\mathcal{F}_p$ is parent of $\mathcal{F}$ and $\mathcal{F}_p$ has exactly one fragment more than $\mathcal{F}$ (i.e., $\mathcal{F}_p$ is at level $i-1$).
2. *Each edge in* $\mathcal{T}$ *is an edge in* $(\mathfrak{F}, \preceq)$. Let $(\mathcal{F}_p, \mathcal{F}_c)$ be an edge in $\mathcal{T}$. By Definition 4.11 it follows that $\mathcal{F}_p \preceq \mathcal{F}_c$, then $(\mathcal{F}_p, \mathcal{F}_c)$ is an edge of $(\mathfrak{F}, \preceq)$.

4.5.1 Computing a Minimal Fragmentation

Our complete search function, function **Fragment** in Fig. 4.6, performs a *depth first search* on the fragmentation tree $\mathcal{T}$ built as an order-based cover. Besides exploiting the tree structure, our proposal takes advantage of the result of Theorem 4.2 by pruning the fragmentation tree to avoid the visit of subtrees composed only of fragmentations violating constraints (i.e., the children of a non correct parent).

The function takes as input the set $\mathcal{A}_f$ of attributes to be fragmented and the set $\mathcal{C}_f$ of well defined non singleton constraints. The function uses variables: *marker*[$\mathcal{F}$], representing the position of the marker within fragmentation $\mathcal{F}$; *Min*, representing the current minimal fragmentation; and *MinNumFrag*, representing the number of fragments composing *Min*. First, the function initializes variable *Min* to $\mathcal{F}_\top$ and variable *MinNumFrag* to the number of fragments in $\mathcal{F}_\top$. Then, it calls function **SearchMin** on $\mathcal{F}_\top$ that iteratively builds the children of $\mathcal{F}_\top$ according to Definition 4.11. Function **SearchMin**($\mathcal{F}_p$) is then recursively called on each fragmentation $\mathcal{F}_c$, child of $\mathcal{F}_p$, only if $\mathcal{F}_c$ satisfies all the constraints (i.e., if function **SatCon** returns *true*). The function exploits the fact that the number of fragments decreases while going down the lattice and compares *Min* with a fragmentation only if it does not have correct children (i.e., it is a candidate minimal fragmentation).

```
FRAGMENT(𝒜_f,𝒞_f)
for each a_i∈𝒜_f do F_i^⊤ := {a_i} /* root of the search tree ℱ_⊤ */
marker[ℱ_⊤] := 1
Min := ℱ_⊤ /* current minimal fragmentation */
MinNumFrag := Evaluate(Min)
SearchMin(ℱ_⊤) /* recursive call that builds the search tree */
return(Min)

SEARCHMIN(ℱ_p)
localmin := true /* minimal fragmentation */
for i:=marker[ℱ_p]...(|ℱ_p|-1) do
   for j:=(i+1)...|ℱ_p| do
      if F_i^p.last<_A F_j^p.first then /* F_i^p fully precedes F_j^p */
        for l:=1...|ℱ_p| do
           case:
                (l<j ∧ l≠i): F_l^c := F_l^p
                (l>j):       F_{l-1}^c := F_l^p
                (l=i):       F_l^c := F_i^p F_j^p
        marker[ℱ_c] := i
        if SatCon(F_i^c) then
          localmin := false
          SearchMin(ℱ_c) /* recursive call on correct fragmentation */
if localmin then
  nf := Evaluate(ℱ_p)
  if nf<MinNumFrag then
    MinNumFrag := nf
    Min := ℱ_p

SATCON(F)
for each c∈𝒞_f do
   if c⊆F then return(false)
return(true)
```

Fig. 4.6 Function that performs a complete search

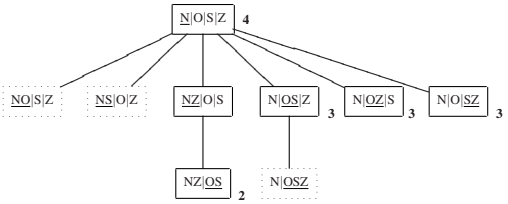
It is interesting to note that, by substituting the definition of the **Evaluate** function with any other cost function monotonic with respect to the dominance relationship, the given function **Fragment** can determine the minimum cost/maximum gain fragmentation in $\mathfrak{F}$.

The fragmentation tree generated by function **Fragment** in Fig. 4.6 according to the order-based cover introduced in Definition 4.11 is not balanced. Indeed, the fragmentation tree is built by inserting the vertices in a specific order, starting from $\mathscr{F}_\top$ and inserting, at each level of the tree, the vertices from left to right. This implies that each vertex in the tree at the i-th level has, as parent, the leftmost vertex in the $(i-1)$-th level that satisfies Definition 4.11. Consequently, as it is visible from Fig. 4.5 the length of the paths from $\mathscr{F}_\top$ to the leaves of the fragmentation lattice decreases when moving from the left to the right in the tree.

Example 4.4. Figure 4.7 illustrates the execution, step by step, of function **SearchMin** applied to Example 4.1. The columns of the table in Fig. 4.7(a) represent the call to **SearchMin** with its parameter $\mathscr{F}_p$; the fragments F_i^p and F_j^p merged; the resulting fragmentation $\mathscr{F}_c$; the value of **SatCon** on F_i^c; the possible recursive call to **SearchMin**$(\mathscr{F}_c)$; the result of function **Evaluate**$(\mathscr{F}_p)$ (i.e., the number of fragments in $\mathscr{F}_p$), when computed; the updates to *Min*. Figure 4.7(b) illustrates the tree built by the recursive calls of function **SearchMin** on the considered example, with the number of fragments necessary for comparison with *Min*

SearchMin($\mathscr{F}_p$)	F_i^p	F_j^p	$\mathscr{F}_c$	**SatCon**(F_i^c)	**SearchMin**($\mathscr{F}_c$)	**Evaluate**($\mathscr{F}_p$)	*Min*
N|O|S|Z	N	O	NO|S|Z	*false*	–		
		S	NS|O|Z	*false*	–		
		Z	NZ|O|S	*true*	NZ|O|S		
	O	S	N|OS|Z	*true*	N|OS|Z		
		Z	N|OZ|S	*true*	N|OZ|S		
	S	Z	N|O|SZ	*true*	N|O|SZ		
NZ|O|S	NZ	O	–	–	–		
		S	–	–	–		
	O	S	NZ|OS	*true*	NZ|OS		
NZ|OS	–	–	–	–	–	2	NZ|OS
N|OS|Z	OS	Z	N|OSZ	*false*	–	3	
N|OZ|S	OZ	S	–	–	–	3	
N|O|SZ	–	–	–	–	–	3	

(a)



(b)

Fig. 4.7 An example of the execution of function **Fragment** in Fig. 4.6

at the right of the corresponding fragmentations. At the beginning, variable *Min* is initialized to [N|O|S|Z] and the corresponding *MinNumFrag* is set to 4. The function then calls function **SearchMin** on [N|O|S|Z]. At the first iteration of the two **for** loops in **SearchMin**([N|O|S|Z]), fragments F_1^p=*N* and F_2^p=*O* are merged, thus generating the fragmentation [NO|S|Z] that violates constraint c_1. The second fragmentation generated is [NS|O|Z], which violates c_3. The third fragmentation [NZ|O|S] is correct and **SearchMin**([NZ|O|S]) is recursively called, which in turn calls **SearchMin**([NZ|OS]). Since the two fragments in [NZ|OS] cannot be merged ($Z \not<_A O$), **SearchMin** is not further called. Therefore, the function compares the number of fragments composing [NZ|OS], which is 2, with *MinNumFrag* and updates *Min* accordingly. The recursive calls on the other fragmentations are processed in an analogous way. The final minimal fragmentation computed by the function is [*NZ*|*OS*] with 2 fragments only.

4.5.2 Correctness and Complexity

Before proving the complexity of function **Fragment** in Fig. 4.6, we introduce a lemma, proving that function **Fragment** computes all correct fragmentations, while it never generates more than once the same solution.

Lemma 4.1. *Function* **Fragment** *in Fig. 4.6 visits all correct fragmentations in* $\mathscr{T}$ *exactly once.*

Proof. The function starts from the root of $\mathscr{T}$ and recursively visits it with a depth-first strategy. At each call of **SearchMin**($\mathscr{F}_p$) it generates all the children of $\mathscr{F}_p$, according to Definition 4.11, by the first two **for** loops and the following **if** instruction. Since **SearchMin** is recursively called only on correct solutions, the subtrees rooted at non correct children are not visited. However, by Theorem 4.2, no correct solution belongs to these subtrees.

Theorem 4.4 (Correctness). *Function* **Fragment** *in Fig. 4.6 terminates and finds a minimal fragmentation* (*Problem 4.1*).

Proof. Function **Fragment** in Fig. 4.6 always terminates since, at each recursive call, it combines two of the fragments in the parent to compute its children. Therefore, the maximum reachable depth is $|\mathscr{A}_f|$.

We now prove that a solution $\mathscr{F}$ computed by this function over $\mathscr{A}_f$ and $\mathscr{C}_f$ is a minimal fragmentation. According to Problem 4.1, a fragmentation $\mathscr{F}$ is *minimal* if and only if (1) it is correct, (2) it maximizes visibility, and (3)$\nexists \mathscr{F}'$ composed of less fragments than $\mathscr{F}$ and satisfying the two conditions above. A fragmentation $\mathscr{F}$ computed by function **Fragment** in Fig. 4.6 satisfies these three properties.

1. The computed fragmentation $\mathscr{F}$ is correct since function **SearchMin** is recursively called only on correct fragmentations $\mathscr{F}_p$ (i.e., when **SatCon** is *true*). Therefore only correct solutions are assigned to the returned solution $\mathscr{F}$ (i.e., *Min*).
2. $\mathscr{F}$ is a fragmentation of R maximizing visibility, since any solution generated by the function is obtained by merging fragments in $\mathscr{F}_\top$. $\mathscr{F}_\top$ is a fragmentation maximizing visibility, since it contains all attributes in $\mathscr{A}_f$ and each $a \in \mathscr{A}_f$ appears exactly in one fragment. The merge operation in the **SearchMin** function simply concatenates two fragments into a single one, thus producing a fragmentation $\mathscr{F}$ such that the condition of maximal visibility is satisfied.
3. $\mathscr{F}$ has minimum number of fragments, since the function visits all the correct solutions in $\mathscr{T}$ and compares *MinNumFrag* with the number of fragments in solutions having only non correct children. By Definition 4.8, the correct solutions that are not compared with $\mathscr{F}$ have a number of fragments greater or equal than $\mathscr{F}$.

Therefore the solution $\mathscr{F}$ computed by function **Fragment** in Fig. 4.6 is a minimal fragmentation.

Theorem 4.5 (Complexity). *Given a set* $\mathscr{C}=\{c_1,\ldots,c_m\}$ *of constraints and a set* $\mathscr{A}=\{a_1,\ldots,a_n\}$ *of attributes the complexity of function* **Fragment**$(\mathscr{A},\mathscr{C})$ *in Fig. 4.6 is* $O(B_n \cdot m)$ *in time.*

Proof. The proof comes directly from Lemma 4.1. In the worst case, each fragmentation in $\mathfrak{F}$, which are $O(B_n)$ in number, is generated exactly once by function

Fragment in Fig. 4.6. Also, function **SatCon** is called once for each solution generated and checks if all constraints, which are m in number, are satisfied. The overall time complexity is therefore $O(B_n \cdot m)$.

4.6 A Heuristic Approach to Minimize Fragmentation

In this section, we present a heuristic algorithm for Problem 4.1 to be applied when the number of attributes in the schema does not allow a complete exploration of the solution space. The heuristic is based on the definition of *vector minimality*, which is then exploited to efficiently find a correct fragmentation maximizing visibility.

A *vector-minimal fragmentation* is formally defined as a fragmentation $\mathscr{F}$ that is correct, maximizes visibility, and all fragmentations that can be obtained from $\mathscr{F}$ by merging any two fragments in $\mathscr{F}$ violate at least one constraint.

Definition 4.12 (Vector-minimal fragmentation). Let R be a relation schema, $\mathscr{C}$ be a set of well defined constraints, and $\mathscr{F}$ be a fragmentation of R. $\mathscr{F}$ is a *vector-minimal fragmentation* iff all the following conditions are satisfied:

1. $\mathscr{F}$ correctly enforces $\mathscr{C}$ (Definition 4.5);
2. $\mathscr{F}$ maximizes visibility (Definition 4.6);
3. $\nexists \mathscr{F}'$ satisfying the two conditions above such that $\mathscr{F} \prec \mathscr{F}'$.

According to this definition of minimality, it easy to see that while a minimal fragmentation is also vector-minimal, the vice versa is not necessarily true.

Example 4.5. Consider fragmentations $\mathscr{F}_1$ and $\mathscr{F}_2$ of Example 4.3, and the set of constraints in Fig. 4.1(b). Since $\mathscr{F}_2 \prec \mathscr{F}_1$, $\mathscr{F}_2$ is not vector-minimal. By contrast, $\mathscr{F}_1$ is vector-minimal. As a matter of fact, $\mathscr{F}_1$ contains all attributes of relation schema `Patient` in Fig. 4.1(a) but `SSN` (maximal visibility); satisfies all constraints in Fig. 4.1(b) (correctness); and no fragmentation obtained from it by merging any pair of fragments satisfies the constraints.

4.6.1 Computing a Vector-minimal Fragmentation

The definition of vector-minimal fragmentation allows us to design a heuristic approach for Problem 4.1 that works in polynomial time and computes a fragmentation that, even if it is not necessarily a minimal fragmentation, it is however near to the optimal solution, as the experimental results show (see Sect. 4.13).

Our heuristic method starts with an empty fragmentation and, at each step, selects the attribute involved in the highest number of unsolved constraints. The rationale behind this selection criterion is to bring all constraints to satisfaction in a few steps. The selected attribute is then inserted into a fragment that is determined in such a way that there is no violation of the constraints involving the attribute. If

```
FRAGMENT(𝒜_f,𝒞_f)
A_ToPlace := 𝒜_f
C_ToSolve := 𝒞_f
Min := ∅
for each a∈A_ToPlace do /* initialize arrays Con[] and N_con[] */
   Con[a] := {c ∈ C_ToSolve: a ∈ c}
   N_con[a] := |Con[a]|
repeat
   if C_ToSolve ≠ ∅ then
      let attr be an attribute with the maximum value of N_con[]
      for each c ∈ (Con[attr] ∩ C_ToSolve) do
         C_ToSolve := C_ToSolve − {c} /* adjust the constraints */
         for each a ∈ c do N_con[a] := N_con[a]−1 /* adjust array N_con[] */
   else /* since all the constrains are satisfied, choose any attribute in A_ToPlace */
      let attr be an attribute in A_ToPlace
   A_ToPlace := A_ToPlace − {attr}
   inserted := false /* try to insert attr into the existing fragments */
   for each F ∈ Min do /* evaluate if F ∪ {attr} satisfies the constraints */
      satisfies := true
      for each c ∈ Con[attr] do
         if c ⊆ (F ∪ {attr}) then
            satisfies := false /* choose the next fragment */
            break
      if satisfies then
         F := F ∪ {attr} /* attr has been inserted into F */
         inserted := true
         break
   if NOT inserted then /* insert attr into a new fragment */
      add {attr} to Min
until A_ToPlace = ∅
return(Min)
```

Fig. 4.8 Function that finds a vector-minimal fragmentation

such a fragment does not exist, a new fragment for the selected attribute is created. The process terminates when all attributes have been inserted into a fragment. Figure 4.8 illustrates function **Fragment** that implements this heuristic method. The function takes as input the set $\mathscr{A}_f$ of attributes to be fragmented, and the set $\mathscr{C}_f$ of well defined non singleton constraints, used to initialize variables *A_ToPlace* and *C_ToSolve*, respectively. It computes a vector-minimal fragmentation *Min* of $\mathscr{A}_f$ as follows.

First, the function initializes *Min* to the empty set and creates two arrays *Con*[] and *N_con*[] that contain an element for each attribute a in *A_ToPlace*. Element *Con*[a] contains the set of constraints on a, and element *N_con*[a] is the number of non solved constraints involving a (note that, at the beginning, *N_con*[a] coincides with the cardinality of *Con*[a]). The function then executes a **repeat until** loop that, at each iteration, places an attribute *attr* into a fragment as follows. If there are constraints still to be solved (*C_ToSolve*$\neq \emptyset$) *attr* is selected as an attribute appearing in the highest number of unsolved constraints. Then, for each constraint c in *Con*[*attr*]$\cap$*C_ToSolve*, the function removes c from *C_ToSolve* and, for each attribute a in c, decreases *N_con*[a] by one. Otherwise, that is, if all constraints are solved (*C_ToSolve*$= \emptyset$), the function chooses *attr* by randomly extracting an attribute from *A_ToPlace* and removes it from *A_ToPlace*. Then, the function looks for a fragment F in *Min* in which *attr* can be inserted without violating any constraint

Min=∅
C_ToSolve={c_1,c_2,c_3}
A_ToPlace={Name,Occupation,Sickness,ZIP}

	c_1	c_2	c_3	$N\_con[a_i]$
Name	×	×		2
Occupation	×		×	2
Sickness		×	×	2
ZIP			×	1
ToSolve	yes	yes	yes	

attr = Name
Con[Name]={c_1,c_2}

Min = {{Name}}
C_ToSolve = {c_3}
A_ToPlace = {Occupation,Sickness,ZIP}

	c_1	c_2	c_3	$N\_con[a_i]$
Name	✓	✓		0
Occupation	✓		×	1
Sickness		✓	×	1
ZIP			×	1
ToSolve	✓	✓	yes	

attr = Occupation
Con[Occupation]={c_1,c_3}

Min = {{Name},{Occupation}}
C_ToSolve = ∅
A_ToPlace = {Sickness,ZIP}

	c_1	c_2	c_3	$N\_con[a_i]$
Name	✓	✓		0
Occupation	✓		✓	0
Sickness		✓	✓	0
ZIP			✓	0
ToSolve	✓	✓	✓	

attr = Sickness
Con[Sickness]={c_2,c_3}

Min = {{Name},{Occupation,Sickness}}
C_ToSolve = ∅
A_ToPlace = {ZIP}

	c_1	c_2	c_3	$N\_con[a_i]$
Name	✓	✓		0
Occupation	✓		✓	0
Sickness		✓	✓	0
ZIP			✓	0
ToSolve	✓	✓	✓	

attr = Z
Con[Z]={c_3}

Min = {{Name,ZIP},{Occupation,Sickness}}}
C_ToSolve = ∅
A_ToPlace = ∅

	c_1	c_2	c_3	$N\_con[a_i]$
Name	✓	✓		0
Occupation	✓		✓	0
Sickness		✓	✓	0
ZIP			✓	0
ToSolve	✓	✓	✓	

Fig. 4.9 An example of the execution of function **Fragment** in Fig. 4.8

including *attr*. If such a fragment F is found, *attr* is inserted into F, otherwise a new fragment {*attr*} is added to *Min*. Note that the search for a fragment terminates as soon as a fragment is found (*inserted=true*). Also, the control on constraint satisfaction terminates as soon as a violation to constraints is found (*satisfies=false*).

Example 4.6. Figure 4.9 presents the execution, step by step, of function **Fragment** in Fig. 4.8 applied to the example in Fig. 4.1. The left hand side of Fig. 4.9 illustrates the evolution of variables *attr*, *Min*, *C_ToSolve*, and *A_ToPlace*, while the right hand side graphically illustrates the same information through a matrix with a row for each attribute and a column for each constraint. If an attribute belongs to an unsolved constraint c_i, the corresponding cell is set to ×; otherwise, if c_i is solved, the cell is set to ✓. At the beginning, *Min* is empty, all constraints are unsolved, and all attributes need to be placed. In the first iteration, function **Fragment** chooses

attribute Name, since it is one of the attributes involved in the highest number of unsolved constraints. The constraints in $Con[\texttt{Name}]$ become now solved, $N\_con[a_i]$ is updated accordingly (for all the attributes in the relation), and fragment {Name} is added to *Min*. Function **Fragment** proceeds in an analogous way by choosing attributes Occupation, Sickness, and Zip. The final solution is represented by fragmentation *Min*={{Name,ZIP}, {Occupation,Sickness}}, which corresponds to the one computed by the complete search function in Fig. 4.6.

4.6.2 Correctness and Complexity

The correctness and complexity of function **Fragment** in Fig. 4.8 are stated by the following theorems.

Theorem 4.6 (Correctness). *Function* **Fragment** *in Fig. 4.8 terminates and finds a vector-minimal fragmentation* (*Definition 4.12*).

Proof. Function **Fragment** in Fig. 4.8 terminates since each attribute is considered only once, and the **repeat until** loop is performed till all the attributes are extracted from *A_ToPlace* (which is initialized to $\mathscr{A}_f$).

We now prove that a solution $\mathscr{F}$ computed by this function over $\mathscr{A}_f$ and $\mathscr{C}_f$ is a vector-minimal fragmentation. According to Definition 4.12, a fragmentation $\mathscr{F}$ is *vector-minimal* if and only if (1) it is correct, (2) it maximizes visibility, and (3) $\nexists \mathscr{F}' : \mathscr{F} \prec \mathscr{F}'$ that satisfies the two conditions above. A fragmentation $\mathscr{F}$ computed by function **Fragment** in Fig. 4.8 satisfies these three properties.

1. Function **Fragment** inserts *attr* into a fragment F if and only if $F \cup \{attr\}$ satisfies the constraints in $Con[attr]$. By induction, we prove that if $F \cup \{attr\}$ satisfies constraints in $Con[attr]$, it satisfies all constraints in $\mathscr{C}$.
   If $\{attr\}$ is the first attribute inserted into F, $F \cup \{attr\} = \{attr\}$. Since $attr \in \mathscr{A}_f$, then the set $\{attr\}$ satisfies all constraints in $\mathscr{C}$. Otherwise, if we suppose that F already contains at least one attribute and that it satisfies all constraints in $\mathscr{C}$, then, by adding *attr* to F the constraints that may be violated are only the constraints in $Con[attr]$. Consequently, if $F \cup \{attr\}$ satisfies all these constraints, it satisfies all constraints in $\mathscr{C}$.
   We can therefore conclude that $\mathscr{F}$ is a correct fragmentation.
2. Since each attribute a in $\mathscr{A}_f$ is inserted exactly into one fragment, function **Fragment** produces a fragmentation $\mathscr{F}$ such that the condition of maximal visibility is satisfied.
3. By contradiction, let $\mathscr{F}'$ be a fragmentation satisfying the constraints in $\mathscr{C}_f$, maximizes visibility, and such that $\mathscr{F} \prec \mathscr{F}'$. Let $V_{\mathscr{F}}$ and $V_{\mathscr{F}'}$ be the fragment vectors associated with $\mathscr{F}$ and $\mathscr{F}'$, respectively.
   First, we prove that $\mathscr{F}'$ contains a fragment $V_{\mathscr{F}'}[a_i]$ that is the union of two different fragments, $V_{\mathscr{F}}[a_i]$ and $V_{\mathscr{F}}[a_j]$, of $\mathscr{F}$. Second, we prove that function **Fragment** cannot generate two different fragments whose union does not violate

any constraint. These two results generate a contradiction since $V_{\mathcal{F}'}[a_i]$, which contains $V_{\mathcal{F}}[a_i] \cup V_{\mathcal{F}}[a_j]$, is a fragment of $\mathcal{F}'$, and thus it does not violate the constraints.

a. Since $\mathcal{F} \prec \mathcal{F}'$, there exists a fragment such that $V_{\mathcal{F}}[a_i] \subset V_{\mathcal{F}'}[a_i]$, and then there exists an attribute a_j (with $j \neq i$) such that $a_j \in V_{\mathcal{F}'}[a_i]$ and $a_j \notin V_{\mathcal{F}}[a_i]$. Note that $a_j \neq a_i$ because, by definition, $a_i \in V_{\mathcal{F}}[a_i]$ and $a_i \in V_{\mathcal{F}'}[a_i]$.
$V_{\mathcal{F}}[a_j]$ and $V_{\mathcal{F}'}[a_j]$ are the fragments that contain a_j. We now show that, not only $a_j \in V_{\mathcal{F}'}[a_i]$, but also the whole fragment $V_{\mathcal{F}}[a_j] \subset V_{\mathcal{F}'}[a_i]$. Since, $a_j \in V_{\mathcal{F}'}[a_j]$ and $a_j \in V_{\mathcal{F}'}[a_i]$ we have that $V_{\mathcal{F}'}[a_j] = V_{\mathcal{F}'}[a_i]$, but since $V_{\mathcal{F}}[a_j] \subset V_{\mathcal{F}'}[a_j]$ we have that $V_{\mathcal{F}}[a_j] \subset V_{\mathcal{F}'}[a_i]$ and therefore $(V_{\mathcal{F}}[a_i] \cup V_{\mathcal{F}}[a_j]) \subseteq V_{\mathcal{F}'}[a_i]$.
b. Let F_h and F_k be the two fragments computed by function **Fragment**, corresponding to $V_{\mathcal{F}}[a_i]$ and $V_{\mathcal{F}}[a_j]$, respectively. Assume, without loss of generality, that $h < k$ (since the proof in the case $h > k$ immediately follows by symmetry). Let a_{k_1} be the first attribute inserted into F_k by the function. Recall that the function inserts an attribute into a new fragment if and only if the attribute cannot be inserted into the already-existing fragments (e.g., F_h) without violating constraints. Therefore, the set of attributes $F_h \cup \{a_{k_1}\}$ violates a constraint as well as the set $V_{\mathcal{F}}[a_i] \cup V_{\mathcal{F}}[a_j]$ that contains $F_h \cup \{a_{k_1}\}$.

This generates a contradiction.

Therefore the solution $\mathcal{F}$ computed by function **Fragment** in Fig. 4.8 is a vector-minimal fragmentation.

Theorem 4.7 (Complexity). *Given a set $\mathcal{C} = \{c_1, \ldots, c_m\}$ of constraints and a set $\mathcal{A} = \{a_1, \ldots a_n\}$ of attributes the complexity of function* **Fragment**$(\mathcal{A}, \mathcal{C})$ *in Fig. 4.8 is $O(n^2 m)$ in time.*

Proof. To choose attribute *attr* from *A_ToPlace*, in the worst case function **Fragment** in Fig. 4.8 scans array *N_con*[], and adjusts array *N_con*[] for each attribute involved in at least one constraint with *attr*. This operation costs $O(nm)$ for each chosen attribute. After the choosing phase, each attribute is inserted into a fragment. Note that the number of fragments is $O(n)$ in the worst case. To choose the right fragment that will contain *attr*, in the worst case the function tries to insert it into all fragments $F \in \mathcal{F}$, and compares $F \cup \{attr\}$ with the constraints in *Con*[*attr*]. Since the sum of the number of attributes in all the fragments is $O(n)$, then $O(n)$ attributes will be compared with the $O(m)$ constraints containing *attr*, giving, in the worst case, a $O(nm)$ complexity for each *attr*. Thus, the complexity of choosing the right fragment is $O(n^2 m)$. We can then conclude that the overall time complexity is $O(n^2 m)$.

4.7 Taking Attribute Affinity into Account

The computation of a minimal fragmentation exploits the basic principle according to which the presence of a high number of plaintext attributes permits an efficient execution of queries. Although this principle may be considered acceptable in many situations, other criteria can also be applied for computing a fragmentation. Indeed, depending of the use of the data, it may be useful to preserve the associations among some attributes. As an example, consider the fragmentation in Fig. 4.2 and suppose that the data need to be used for statistical purposes. In particular, suppose that physicians should be able to explore the link between a specific `Sickness` and the `Occupation` of patients. The computed fragmentation however does not make visible the association between `Sickness` and `Occupation`, thus making the required analysis not possible (as it would violate the constraints). In this case, a fragmentation where these two attributes are stored in clear form in the same fragment is preferable to the computed fragmentation. The need for keeping together some specific attributes in the same fragment may not only depend on the use of the data but also on the queries that need to be frequently executed on the data. Indeed, given a query Q and a fragmentation $\mathscr{F}$, the execution cost of Q varies according to the specific fragment used for computing the query. This implies that, with respect to a specific query workload, different fragmentations may be more convenient than others in terms of query performance.

To take into consideration both the use of the data and the query workload in the fragmentation process, we exploit the concept of *attribute affinity* traditionally applied to express the advantage of having pairs of attributes in the same fragment in distributed DBMSs [76] and that is therefore adopted by schema design algorithms using the knowledge of a representative *workload* for computing a suitable partition. In our context, attribute affinity is also a measure of how strong the need of keeping the attributes in the same fragment is. By considering the total order relationship $<_A$ among attributes in $\mathscr{A}_f$ and assuming a_i to denote the i-th attribute in the ordered sequence, the affinity between attributes is represented through an *affinity matrix*. The matrix, denoted M, has a row and a column for each attribute appearing in non singleton constraints, and each cell $M[a_i,a_j]$ represents the benefit obtained by having attributes a_i and a_j in the same fragment. Clearly, the affinity matrix contains only positive values and is symmetric with respect to its main diagonal. Also, for all attributes a_i, $M[a_i,a_i]$ is not defined. The affinity matrix can then be represented as a triangular matrix, where only cells $M[a_i,a_j]$, with $i < j$ (i.e., $a_i <_A a_j$), are represented. Figure 4.10 illustrates an example of affinity matrix for relation `Patient` in Fig. 4.1, where $<_A$ is the lexicographic order.

The consideration of attribute affinity naturally applies to fragments and fragmentations. Fragmentations that maintain together attributes with high affinity are to be preferred. To reason about this, we define the concept of *fragmentation affinity*. Intuitively, the affinity of a fragment is the sum of the affinities of the different pairs of attributes in the fragment; the affinity of a fragmentation is the sum of the affinities of the fragments in it. This is formalized by the following definition.

	N	O	S	Z
N		10	15	5
O			5	10
S				20
Z				

Fig. 4.10 An example of affinity matrix

Definition 4.13 (Fragmentation affinity). Let R be a relation schema, M be an affinity matrix for R, $\mathscr{C}$ be a set of well defined constraints over R, and $\mathscr{F}=\{F_1,\ldots,F_n\}$ be a correct fragmentation of R. The affinity of $\mathscr{F}$, denoted *affinity*($\mathscr{F}$), is computed as:

$affinity(\mathscr{F}) = \sum_{k=1}^{n} aff(F_k)$, where $aff(F_k) = \sum_{a_i,a_j \in F_k, i<j} M[a_i,a_j]$ is the affinity of fragment F_k, $k = 1 \ldots n$.

As an example, consider the affinity matrix in Fig. 4.10 and fragmentation $\mathscr{F}$={{`Name`,`ZIP`}, {`Occupation`,`Sickness`}}. Then, *affinity*($\mathscr{F}$) = *aff*({`Name`,`ZIP`}) + *aff*({`Occupation`,`Sickness`}) = M[N,Z] + M[O,S] = 5+5 = 10. With the consideration of affinity, the problem becomes therefore to determine a correct fragmentation that has maximum affinity. This is formally defined as follows.

Problem 4.2 (Maximum affinity). Given a relation schema R, a set $\mathscr{C}$ of well defined constraints over R, and an affinity matrix M, find a *fragmentation* $\mathscr{F}$ of R such that all the following conditions hold:

1. $\mathscr{F}$ correctly enforces $\mathscr{C}$ (Definition 4.5);
2. $\mathscr{F}$ maximizes visibility (Definition 4.6);
3. $\nexists \mathscr{F}'$ satisfying the conditions above such that $affinity(\mathscr{F}') > affinity(\mathscr{F})$.

Like Problem 4.1, the maximum affinity problem is *NP-hard*, as formally stated by the following theorem.

Theorem 4.8. *The maximum affinity problem is NP-hard.*

Proof. The proof is a reduction from the NP-hard minimum hitting set problem [50], which can be formulated as follows: *given a collection C of subsets of a set S, find the smallest subset S' of S such that S' contains at least one element from each subset in C*.

The reduction of the hitting set problem to the maximum affinity problem can be defined as follows. Let S' be the solution of the minimum hitting set problem, and let $R = S \cup \{a_c\}$ be a relation, where a_c is an attribute different from any other element in S.

We consider only the sets in C with cardinality greater than 1, since any singleton set s in C corresponds to an element that must be inserted into the solution S', and we can directly put it in. Moreover, if $s_i, s_j \in C$ and $s_i \subset s_j$, s_j is redundant and

can be removed from C, since if S' contains an element of s_i, then it also contains an element of s_j. Thus, let $\mathscr{C}_f = \{s \in C\colon |s| > 1 \text{ and } \forall s' \in C,\ s' \not\subset s\}$ be the set of association constraints, and let $\mathscr{A}_f = \{a \in R\colon \{a\} \notin C\}$ be the set of attributes to be fragmented. We note that the construction of the set of constraints $\mathscr{C}_f$ is polynomial in C, and that, by construction, $\mathscr{C}_f$ is a set of well defined association constraints. Also, a_c is not contained in any constraint in $\mathscr{C}_f$. Consider now an affinity matrix that contains the value 0 in every cell but the cells corresponding to a_c, which are set to 1 (i.e., $M[a_i, a_j] = 1$ iff $a_i = a_c$ or $a_j = a_c$; $M[a_i, a_j] = 0$, otherwise).

Since only the affinity between attribute a_c and any other attribute is greater than 0, a fragmentation algorithm with the goal of maximizing the affinity computes a fragmentation where fragment F_c containing a_c includes the maximum number of attributes that can be inserted into a single fragment without violating the constraints. The affinity of the computed fragmentation corresponds to the cardinality of F_c. Since a constraint is violated only if all its attributes belong to the same fragment, a fragment may include all attributes composing a constraint but one. Therefore, maximizing the number of attributes composing F_c is equivalent to minimizing the size of the set S' of attributes that contains at least one attribute from each constraint. S' is the solution of the minimum hitting set problem. Consequently, a maximal affinity fragmentation $\mathscr{F}$ of R, with respect to M, satisfying all constraints in $\mathscr{C}_f$, corresponds to a solution for the minimum hitting set problem. In particular, given fragment F_c that contains attribute a_c, the solution for the minimum hitting set problem is $S' = R - F_c$.

In the following, we describe a heuristic approach for Problem 4.2.

4.8 A Heuristic Approach to Maximize Affinity

Our heuristic approach to determine a fragmentation that maximizes affinity exploits a greedy approach that, at each step, combines fragments that have the highest affinity. The heuristic starts by putting each attribute to be fragmented into a different fragment. The affinity between pairs of fragments is the affinity between the attributes contained in their union (as dictated by the affinity matrix). Then, the two fragments with the highest affinity, let call them F_i and F_j, are merged together (if this does not violate constraints) and F_i is updated by adding the attributes of F_j, while F_j is removed. The affinity of the new version of F_i with respect to any other fragment F_k is the sum of the affinities that F_k had with the old version of F_i and F_j. The heuristic proceeds in a greedy way iteratively merging, at each step, the fragments with highest affinity until no more fragments can be merged without violating the constraints. Figure 4.11 gives a graphical representation of our heuristic approach; at each step, light grey boxes denote the pair of fragments with highest affinity. The correctness of the heuristics lies in the fact that, at each step, the affinity of the resulting fragmentation can only increase. As a matter of fact, it is easy to see that affinity is monotonic with respect to the dominance relationship (see Lemma 4.2 in Sect. 4.8.2).

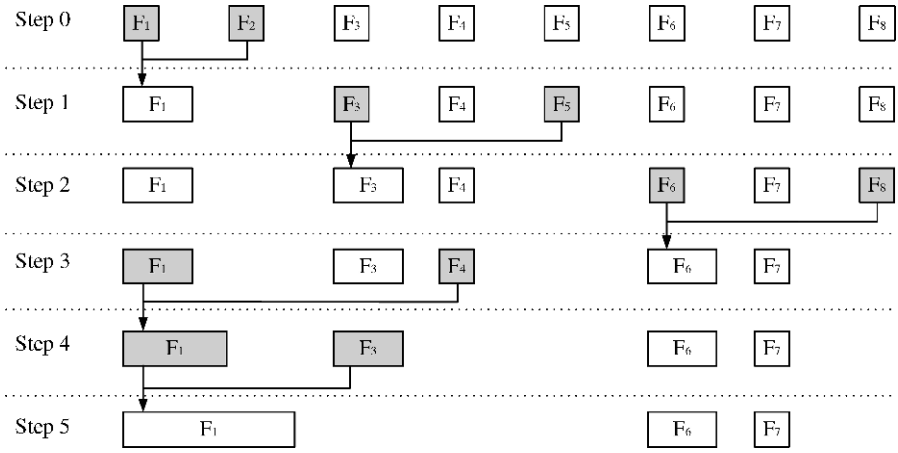


Fig. 4.11 Graphical representation of the working of the function in Fig. 4.12

The following subsection describes the function implementing this heuristic approach. In the function, instead of controlling constraints to determine whether two fragments can be merged, we exploit the affinity matrix and set to -1 the affinity of fragments whose merging would violate the constraints (thus ignoring them in the evaluation of fragments to be merged).

4.8.1 Computing a Vector-minimal Fragmentation with the Affinity Matrix

Function **Fragment** in Fig. 4.12 takes as input the set $\mathscr{A}_f$ of attributes to be fragmented and a set $\mathscr{C}_f$ of well defined non singleton constraints. It computes a vector-minimal fragmentation *Max* of $\mathscr{A}_f$, where at each step the fragments to be merged are chosen according to their affinity. In the following, with a slight abuse of notation, we use $M[F_i,F_j]$ to denote the cell in the affinity matrix identified by the smallest attribute in F_i and F_j (i.e., $F_i.first$ and $F_j.first$), according to the order relationship $<_A$ on attributes in $\mathscr{A}_f$.

First, the function initializes the set of constraints *C_ToSolve* to be solved with $\mathscr{C}_f$, *Max* to a fragmentation having a fragment F_i for each of the attributes a_i in $\mathscr{A}_f$, and creates a set *FragmentIndex* that contains the index i of each fragment $F_i \in Max$. The function also checks all constraints in *C_ToSolve* composed of two attributes only, and sets to -1 the corresponding cells in the affinity matrix. These constraints are removed from *C_ToSolve*. In general, at each iteration of the algorithm, for each $i < j$, $M[F_i,F_j]$ is equal to -1 if the fragment obtained as $F_i \cup F_j$ violates some constraints.

FRAGMENT($\mathscr{A}_f$,$\mathscr{C}_f$)
/* initial solution with a fragment for each attribute */
C_ToSolve := $\mathscr{C}_f$
Max := $\emptyset$
FragmentIndex := $\emptyset$
for i=1...$|\mathscr{A}_f|$ **do**
  $F_i := \{a_i\}$
  $Max := Max \cup \{F_i\}$
  $FragmentIndex := FragmentIndex \cup \{i\}$
/* cells in M corresponding to constraints are invalidated */
for each $\{a_x,a_y\} \in C\_ToSolve$ **do**
  $M[F_{min(x,y)},F_{max(x,y)}] := -1$
  $C\_ToSolve := C\_ToSolve - \{\{a_x,a_y\}\}$
/* extract the pair of fragments with maximum affinity */
Let $[F_i,F_j]$, $i<j$ and $i,j \in FragmentIndex$, be the pair of fragments with maximum affinity
while $|FragmentIndex| > 1 \wedge M[F_i,F_j] \neq -1$ **do** /* merge the two fragments */
  $F_i := F_i \cup F_j$
  $Max := Max - \{F_j\}$
  $FragmentIndex := FragmentIndex - \{j\}$
  /* update the affinity matrix */
  for each $k \in FragmentIndex : k \neq i$ **do**
    if $M[F_{min(i,k)},F_{max(i,k)}]=-1 \vee M[F_{min(j,k)},F_{max(j,k)}]=-1$ **then**
      $M[F_{min(i,k)},F_{max(i,k)}] := -1$
    else
      for each $c \in C\_ToSolve$ **do**
        if $c \subseteq (F_i \cup F_k)$ **then**
          $M[F_{min(i,k)},F_{max(i,k)}] := -1$
          $C\_ToSolve := C\_ToSolve - \{c\}$
      if $M[F_{min(i,k)},F_{max(i,k)}] \neq -1$ **then**
        $M[F_{min(i,k)},F_{max(i,k)}] := M[F_{min(i,k)},F_{max(i,k)}] + M[F_{min(j,k)},F_{max(j,k)}]$
  Let $[F_i,F_j]$, $i<j$ and $i,j \in FragmentIndex$, be the pair of fragments with maximum affinity
return(*Max*)

Fig. 4.12 Function that finds a vector-minimal fragmentation with maximal affinity

Function **Fragment** in Fig. 4.12 then executes a **while** loop that, at each iteration, merges two fragments in *Max* as follows. If there are still pairs of fragments that can be merged, that is, there are still cells in M different from -1, the function identifies the cell $[F_i,F_j]$ (with $i<j$) with the maximum value in M. Then, F_i is updated to the union of the two fragments and F_j is removed from *Max*. Also, j is removed from *FragmentIndex*, since the corresponding fragment is no more part of the solution. The function, in the end, updates M. In particular, for each fragment F_k, $k \in (FragmentIndex - \{i\})$, cell $M[F_i,F_k]$ is set to -1 if either cell $M[F_i,F_k]$ or cell $M[F_j,F_k]$ is -1, or if $F_i \cup F_k$ violates at least one constraint still in *C_ToSolve*. In this latter case, the violated constraints $\{c_x,\ldots,c_y\}$ are removed from *C_ToSolve*. Otherwise, cell $M[F_i,F_k]$ is summed with the value in cell $M[F_j,F_k]$.

Example 4.7. Figure 4.13 presents the execution, step by step, of function **Fragment** in Fig. 4.12, applied to the example in Fig. 4.1 and considering the affinity matrix in Fig. 4.10. The left hand side of Fig. 4.13 illustrates the evolution of fragments and of the chosen pair F_i, F_j. The central part of Fig. 4.13 illustrates the evolution of matrix M, where dark grey columns represent fragments merged with other fragments, and thus removed from the set of fragments. The right hand side of Fig. 4.13 illustrates the set *C_ToSolve* of constraints to be solved: if an attribute belongs to constraint c_i in *C_ToSolve*, the corresponding cell is set to $\times$; if c_i is removed from *C_ToSolve*,

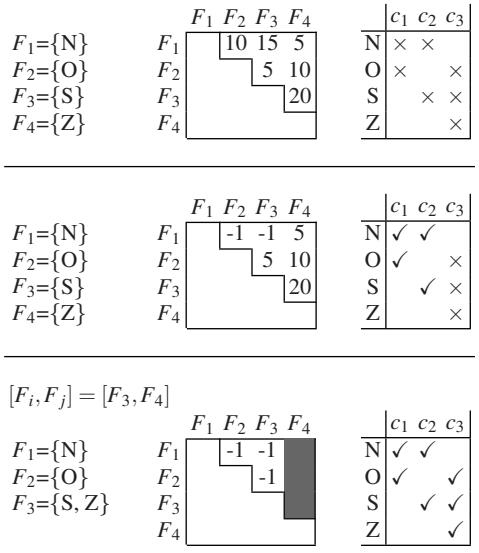


Fig. 4.13 An example of the execution of function **Fragment** in Fig. 4.12

the cell is set to ✓. At the beginning, all constraints are not solved and there is a fragment F for each attribute in $\mathscr{A}_f$. First, M is updated by setting to -1 the cells representing constraints involving only two attributes, that is, constraints c_1 and c_2, which are then removed from *C_ToSolve*. Function **Fragment** chooses the cell in M with the highest affinity, that is, $M[F_3,F_4] = 20$. Consequently, F_4 is merged with F_3 (the 4th column becomes dark grey to denote that fragment F_4 does not exist anymore). Then, values in the affinity matrix are updated: cell $M[F_1,F_3]$ is set to -1, since $M[F_1,F_3]$ were -1 before the merge operation; $M[F_2,F_3]$ should be set to $M[F_2,F_3]+M[F_2,F_4] = 5+10 = 15$, but it represents fragment $\{O,S,Z\}$ that violates constraint c_3, therefore the cell is set to -1 and c_3 is removed from *C_ToSolve*. The final solution is *Max*={{`Name`}, {`Occupation`}, {`Sickness`,`ZIP`}}, with affinity equal to 20. (Note that the solution computed by function **Fragment** in Fig. 4.8, and represented in Fig. 4.9, has 2 fragments only, but its affinity is 10.)

We note that function **Fragment** in Fig. 4.12 can be used to simulate function **Fragment** in Fig. 4.8 by sorting the attributes in the order with which they are considered by the function in Fig. 4.12 and considering an initial affinity matrix containing 0 as affinity value between each pair of attributes. The ordering of attributes can be simply computed by iteratively calculating the number of unsolved constraints $N\_con[a]$ involving each attribute a, and inserting, as next element of the ordered list, the attribute that maximizes $N\_con[a]$. Since the affinity matrix contains values 0 and -1 only, the order for choosing pair of fragments as the next maximum affinity pair is the same of function **Fragment** in Fig. 4.8.

4.8.2 Correctness and Complexity

Before proving the correctness and complexity of our heuristic, we introduce two lemmas proving the monotonicity property of fragmentation affinity with respect to the dominance relationship $\preceq$ and the correctness of the matrix computation, respectively.

Lemma 4.2 (Monotonicity). *Let R be a relation, M be an affinity matrix for R, $\mathscr{C}$ be a set of well defined constraints over R, and $\mathscr{F}$ and $\mathscr{F}'$ be two correct fragmentations for R. If $\mathscr{F} \preceq \mathscr{F}' \Longrightarrow$ affinity($\mathscr{F}$)$\leq$affinity($\mathscr{F}'$).*

Proof. By definition, given two fragmentations $\mathscr{F}=\{F_1,\ldots,F_n\}$ and $\mathscr{F}' = \{F_1',\ldots,F_m'\}$ such that $\mathscr{F} \prec \mathscr{F}'$, then $V_{\mathscr{F}}[a] \subseteq V_{\mathscr{F}'}[a]$, $\forall a \in \mathscr{A}_f$. Therefore, for each a such that $V_{\mathscr{F}}[a]=V_{\mathscr{F}'}[a]$, the affinity of the two fragments F and F' containing a in $\mathscr{F}$ and $\mathscr{F}'$ respectively, is the same. On the contrary, for all attributes a such that $V_{\mathscr{F}}[a] \subset V_{\mathscr{F}'}[a]$, the affinity of the two fragments F and F' containing a in $\mathscr{F}$ and $\mathscr{F}'$ respectively, is such that $aff(F) \leq aff(F')$. In fact, $aff(F')=aff(F)+\sum M[a_i,a_j]$, $\forall a_i \in F', a_j \in (F'-F)$ with $i<j$. Since $M[a_i,a_j]$ is always a non negative value, it holds that if $\mathscr{F} \prec \mathscr{F}'$, then $affinity(\mathscr{F}) \leq affinity(\mathscr{F}')$.
If $\mathscr{F} = \mathscr{F}'$ it is straightforward to see that $affinity(\mathscr{F})=affinity(\mathscr{F}')$.

Lemma 4.3. *At the beginning of each iteration of the* **while** *loop in function* **Fragment** *in Fig. 4.12, $M[F_i,F_j] = -1 \Longleftrightarrow \exists c \in \mathscr{C} : c \subseteq (F_i \cup F_j)$.*

Proof. At initialization, function **Fragment** checks constrains involving exactly two attributes $\{a_x,a_y\}$ and sets to -1 the cell in M corresponding to the pair of fragments $F_x=\{a_x\}$ and $F_y=\{a_y\}$. Also, these constraints are removed from *C_ToSolve*.

When function **Fragment** merges two fragments F_i and F_j ($i<j$), j is removed from *FragmentIndex*. For each k in *FragmentIndex* but i, cell $M[F_{min(i,k)},F_{max(i,k)}]$ is set to -1 if either $M[F_{min(i,k)},F_{max(i,k)}]$ or $M[F_{min(j,k)},F_{max(j,k)}]$ were -1 before the update. Indeed, if either $F_i \cup F_k$ or $F_j \cup F_k$ violated a constraint before merging F_i with F_j, also $F_i \cup F_k$ (i.e., $\exists c \in \mathscr{C}$ such that $c \subseteq F_i$ or $c \subseteq F_j$) since F_i is set to $F_i \cup F_j$ after the update. Note that constraints removed from *C_ToSolve* are represented by -1 being always kept in M. Also, when $F_i \cup F_k$ is checked against constraints, the algorithm looks for constraints representing a subset of $F_i \cup F_k$ in *C_ToSolve*, and the corresponding constraints are removed from *C_ToSolve*, since there is a -1 in M representing it.

Theorem 4.9 (Correctness). *Function* **Fragment** *in Fig. 4.12 terminates and finds a vector-minimal fragmentation* (*Definition 4.12*).

Proof. Function **Fragment** always terminates. In fact, the **while** loop terminates because at each iteration the number of indexes in *FragmentIndex* decreases by one, and the iterations are performed only if *FragmentIndex* contains at least two indexes.

We now prove that a solution $\mathscr{F}$ computed by this function over $\mathscr{A}_f$ and $\mathscr{C}_f$ is a vector-minimal fragmentation. According to Definition 4.12 of minimality, a fragmentation $\mathscr{F}$ is *vector-minimal* if and only if (1) it is correct, (2) it maximizes visibility, and (3) $\nexists \mathscr{F}' : \mathscr{F} \prec \mathscr{F}'$ that satisfies the two conditions above. A fragmentation $\mathscr{F}$ computed by function **Fragment** in Fig. 4.12 satisfies these three properties.

1. Function **Fragment** starts with a simple correct fragmentation ($F_i := \{a_i\}$, for all $a_i \in \mathscr{A}_f$), and it iteratively merges only fragments that form a correct fragment, since the pair of fragments to be merged is extracted as the pair with maximum affinity and the fragments are merged only if their affinity is a positive value. By Lemma 4.3, only fragments whose union does not violate constraints are merged. We can therefore conclude that $\mathscr{F}$ correctly enforces $\mathscr{C}$.
2. Since each attribute in $\mathscr{A}_f$ is initially inserted exactly into one fragment, and when two fragments are merged only the result of their union is kept in $\mathscr{F}$, function **Fragment** produces a fragmentation $\mathscr{F}$ such that the condition of maximal visibility is satisfied.
3. By contradiction, let $\mathscr{F}'$ be a fragmentation satisfying the constraints in $\mathscr{C}_f$ and maximizing visibility, such that $\mathscr{F} \prec \mathscr{F}'$. Let $V_{\mathscr{F}}$ and $V_{\mathscr{F}'}$ be the fragment vectors associated with $\mathscr{F}$ and $\mathscr{F}'$, respectively.
   As already proved for Theorem 4.6, $\mathscr{F}'$ contains a fragment $V_{\mathscr{F}'}[a_i]$ that is the union of two different fragments, $V_{\mathscr{F}}[a_i]$ and $V_{\mathscr{F}}[a_j]$, of $\mathscr{F}$. We need then to prove that function **Fragment** cannot terminate with two different fragments whose union does not violate any constraint.
   Let F_h and F_k be the two fragments computed by function **Fragment**, corresponding to $V_{\mathscr{F}}[a_i]$ and $V_{\mathscr{F}}[a_j]$, respectively. Assume, without loss of generality, that $h < k$ (since the proof in the case $h > k$ immediately follows by symmetry). By Lemma 4.3, M contains non-negative values only for pairs of fragments whose union generates a correct fragment, and therefore function **Fragment** cannot terminate with fragmentation $\mathscr{F}$ since M still contains a non negative value to be considered ($M[F_h, F_k]$). This generates a contradiction.

Therefore the solution $\mathscr{F}$ computed by **Fragment** in Fig. 4.12 is a vector-minimal fragmentation.

Theorem 4.10 (Complexity). *Given a set of constraints $\mathscr{C} = \{c_1, \ldots, c_m\}$ and a set of attributes $\mathscr{A} = \{a_1, \ldots a_n\}$ the complexity of function* **Fragment**$(\mathscr{A}, \mathscr{C})$ *in Fig. 4.12 is $O(n^3 m)$ in time.*

Proof. The first **for** and **for each** loops of function **Fragment** cost $O(n+m)$. The **while** loop is performed $O(n)$ times, since at each iteration an element from *FragmentIndex* is extracted. The **for each** loop nested into the **while** loop updates the cells corresponding to fragments F_i and F_j in the affinity matrix. While j is simply removed from *FragmentIndex*, and the column F_j in the matrix is simply ignored, the update of the cells corresponding to F_i, which are $O(n)$ in number, costs $O(n^2 m)$ because all the constraints in *C_ToSolve* containing $F_i \cup F_j$ are considered. Each extraction of the pair of fragments with maximum affinity from M simply scans (in the worst case) the affinity matrix, and its computational cost is $O(n^2)$ in time. The overall time complexity is therefore $O(n^3 m)$.

4.9 Query Cost Model

The standard approach to physical database design considers a representative set of queries as the starting point for the concrete identification of a satisfying solution. The same approach can also be applied for fragmenting data by taking into consideration the gain due to sets of attributes with more than two plaintext attributes appearing in the same fragment. To this purpose, we first introduce the following query cost function.

Given a fragmentation $\mathscr{F}$ for R, any query Q can be evaluated on each of the fragments composing $\mathscr{F}$ because the corresponding physical fragment contains all the attributes of R, either in encrypted or in clear form. However, the execution cost of a query varies depending on the schema of the fragment used for query computation. Overall, with respect to a given query workload, some fragmentations can exhibit a lower cost than others. We are then interested in identifying a correct fragmentation with maximal visibility characterized by the minimum cost. To this purpose, we introduce a query cost model for query execution on a fragmented schema.

We describe a query workload $\mathscr{Q}$ as a set $\{Q_1,\ldots,Q_m\}$ of queries, where each query Q_i, $i = 1,\ldots,m$, is characterized by an execution frequency $freq(Q_i)$ and is of the form:

SELECT $a_{i_1},\ldots,a_{i_n}$
FROM R
WHERE $\bigwedge_{j=1}^{k}$ (a_j IN V_j)

where V_j is a set of values in the domain of attribute a_j. Given a fragment $F_l \in \mathscr{F}$ and a query $Q_i \in \mathscr{Q}$, the cost of executing query Q_i over F_l depends on the set of attributes appearing in clear form in F_l and on their selectivity; the availability of more attributes in clear form in a fragment permits a more efficient execution of queries on the fragment. We therefore estimate the *selectivity* of query Q_i on F_l in terms of the percentage of tuples in F_l that are returned by the execution of query Q_i on F_l. First, we evaluate the selectivity of each single condition in query Q_i as follows. The selectivity of the j-th condition is computed as the ratio of the number of tuples in the fragment such that the value of attribute a_j is a value in V_j, over the number of tuples in F_l, which corresponds to the number of tuples in the original relation R: $\frac{\sum_{v \in V_j} num\_tuples(a_j,v)}{|R|}$, where $num\_tuples(a_j,v)$ denotes the number of tuples whose value for attribute a_j is v. Since we assume that the values of different attributes are distributed independently of each other, the selectivity of $\bigwedge_{j=1}^{k}$ (a_j IN V_j) in query Q_i on fragment F_l, denoted $S(Q_i,F_l)$, is the product of the selectivity of each single condition. In particular, the j-th condition contributes to the computation of the selectivity if and only if the corresponding attribute a_j appears in clear form in F_l; otherwise the condition cannot be evaluated on the fragment and it is therefore not useful to select the tuples to be returned in response to the query (this restriction will be relaxed when we will consider in Sect. 4.12 the introduction of indexes on encrypted attributes).

The cost of evaluating query Q_i on fragment F_l, denoted $Cost(Q_i,F_l)$, is then estimated by the size of the information returned, which is computed by multiplying $S(Q_i,F_l)$ (i.e., the selectivity of Q_i on F_l) by the number of tuples in the considered fragment, and by the size in bytes, denoted $size(t_l)$, of the result tuples:

$$Cost(Q_i,F_l) = S(Q_i,F_l) \cdot |R| \cdot size(t_l)$$

This is a common assumption in cost models for query optimizers, particularly in systems where information has to be exchanged among different components, where the computational cost of queries is considered less important. We note that in the architecture only symmetric encryption is used, which current processors are typically able to apply even on high-rate transfers. It is reasonable then to build a cost model that does not consider this aspect.

Note that both the set of attributes in the SELECT clause and the set of attributes in the WHERE clause of query Q_i determine the size in bytes of each result tuple. Indeed, $size(t_l)$ is obtained by summing the size in bytes of each attribute in the SELECT clause that appears in clear form in F_l and the size in bytes of the *enc* attribute of the fragment, if there exists at least one attribute in the SELECT or WHERE clauses that does not appear in clear form in F_l. The rationale is that the encrypted portion of the fragment is needed to subsequently retrieve the desired attribute by decrypting it. The final cost of evaluating query Q_i on $\mathscr{F}$ is therefore the minimum among the costs of evaluating the query on each of the fragments in $\mathscr{F}$. In other words, given $\mathscr{F} = \{F_1,\ldots,F_r\}$, the cost of evaluating query Q_i on $\mathscr{F}$ is:

$$Cost(Q_i,\mathscr{F}) = Min(Cost(Q_i,F_1),\ldots,Cost(Q_i,F_r))$$

The cost of fragmentation $\mathscr{F}$ with respect to $\mathscr{Q}$ is the sum of the costs $Cost(Q_i,\mathscr{F})$ of each single query Q_i weighted by its frequency, as formally stated in the following definition.

Definition 4.14 (Fragmentation cost). Let R be a relation schema, $\mathscr{C}$ be a set of well defined constraints over R, $\mathscr{F}$ be a fragmentation of R maximizing visibility, and $\mathscr{Q}=\{Q_1,\ldots,Q_m\}$ be a query workload for R. The *fragmentation cost* of $\mathscr{F}$ with respect to $\mathscr{Q}$, denoted $Cost(\mathscr{Q},\mathscr{F})$, is computed as:

$$Cost(\mathscr{Q},\mathscr{F}) = \sum_{i=1}^{m} (freq(Q_i) \cdot Cost(Q_i,\mathscr{F}))$$

Example 4.8. Consider the fragmentation of the `Patient` relation in Fig. 4.2. Given query Q:

```
SELECT *
FROM   Patient
WHERE  Sickness='Latex al.' AND Occupation='Nurse'
```

the selectivity of the fragments is: $S(Q,F_1)$=1, since neither `Sickness` nor `Occupation` are plaintext represented in F_1; $S(Q,F_2)$=2/6, since `Occupation` belongs to F_2 and there are 2 nurses out of 6 patients; $S(Q,F_3)$=3/6, since `Sickness` belongs to F_3 and there are 3 patients suffering from Latex al-

lergy. Supposing, for simplicity, that $size(t_1)$=$size(t_2)$=$size(t_3)$=1, we have that $Cost(Q,\mathscr{F})$=Min$(6,2,3)$. $Cost(Q,F_2)$=2.

The cost function here defined enjoys a nice property. Indeed, it is monotonic with respect to the dominance relationship $\preceq$, as proved by the following lemma.

Lemma 4.4 (Monotonicity). *Given a relation schema R, a set $\mathscr{C}$ of well defined constraints over R, the set $\mathscr{A}_f \subseteq R$ of attributes to be fragmented, and a query workload $\mathscr{Q}$ for R, $\forall \mathscr{F}_i, \mathscr{F}_j \in \mathfrak{F}$: $\mathscr{F}_i \preceq \mathscr{F}_j \Longrightarrow Cost(\mathscr{Q},\mathscr{F}_j) \leq Cost(\mathscr{Q},\mathscr{F}_i)$.*

Proof. Consider two fragmentations $\mathscr{F}_i$ and $\mathscr{F}_j$ such that $\mathscr{F}_i \preceq \mathscr{F}_j$, $\mathscr{F}_i=\{F_1^i,\ldots,F_n^i\}$, and $\mathscr{F}_j=\{F_1^j,\ldots,F_{n-1}^j\}$. By Definition 4.8, $\mathscr{F}_j$ is obtained by merging two fragments in $\mathscr{F}_i$, say F_a^i and F_b^i, into F_c^j. Therefore, $\forall F_x^j$, $x \neq c$ there exists a fragment F_y^i=F_x^j, and then $\forall Q_k \in \mathscr{Q}$, $S(Q_k,F_x^j)$=$S(Q_k,F_y^i)$. Considering now fragment F_c^j, we conclude that $\forall Q_k \in \mathscr{Q}$, $S(Q_k,F_c^j) \leq S(Q_k,F_a^i)$ and $S(Q_k,F_c^j) \leq S(Q_k,F_b^i)$, since F_c^j=$F_a^i \cup F_b^i$ and the selectivity of any condition (a IN V) is between 0 and 1. Also, since F_c^j has more attributes in clear from than F_a^i (and F_b^i), the evaluation of any query Q_k can be more precise in projecting attributes. Therefore, $size(t_a) \geq size(t_c)$ and $size(t_b) \geq size(t_c)$. As a consequence, $\forall Q_k \in \mathscr{Q}$, $Cost(Q,F_c^j) \leq Cost(Q_k,F_a^i)$ and $Cost(Q,F_c^j) \leq Cost(Q_k,F_b^i)$.

Since $\forall Q_k \in \mathscr{Q}$, $Cost(Q_k,\mathscr{F})$ is computed as the minimum among $Cost(Q_k,F)$, all the queries assigned to F_a^i and F_b^i by $\mathscr{F}_i$ are assigned to F_c^j by $\mathscr{F}_j$, thus $Cost(Q_k,\mathscr{F}_j) \leq Cost(Q_k,\mathscr{F}_i)$ for these queries. Queries not assigned by $\mathscr{F}_i$ to F_a^i and F_b^i may be assigned by $\mathscr{F}_j$ to F_c^j. This happens only if $Cost(Q_k,F_c^j)$ is lower than $Cost(Q_k,F_x^i)$ for the previously chosen fragment F_x^i. Consequently, $\forall Q_k \in \mathscr{Q}$, $Cost(Q_k,\mathscr{F}_j) \leq Cost(Q_k,\mathscr{F}_i)$. Since the frequency of queries is the same for both $\mathscr{F}_i$ and $\mathscr{F}_j$, we conclude that $Cost(\mathscr{Q},\mathscr{F}_j) \leq Cost(\mathscr{Q},\mathscr{F}_i)$.

This property is easily extended to any pair of fragmentations $\mathscr{F}_i$ and $\mathscr{F}_j$, $\mathscr{F}_i \preceq \mathscr{F}_j$. Considering $(\mathfrak{F},\preceq)$, there is a path from $\mathscr{F}_i$ to $\mathscr{F}_j$. Each solution $\mathscr{F}_a$ in the path dominates the solution $\mathscr{F}_b$ preceding it in the path. Therefore, $Cost(\mathscr{Q},\mathscr{F}_a) \leq Cost(\mathscr{Q},\mathscr{F}_b)$. By inductively applying this observation along all the path from $\mathscr{F}_i$ to $\mathscr{F}_j$, we obtain that $Cost(\mathscr{Q},\mathscr{F}_j) \leq Cost(\mathscr{Q},\mathscr{F}_i)$.

We are now interested in finding a correct fragmentation $\mathscr{F}$ with maximal visibility that minimizes the cost associated with a specific query workload, meaning that there does not exist another fragmentation satisfying constraints, maximizing visibility, and such that its cost is less than the cost associated with $\mathscr{F}$. This problem can be formalized as follows.

Problem 4.3 (Minimum cost). Given a relation schema R, a set $\mathscr{C}$ of well defined constraints over R, and a query workload $\mathscr{Q}$=$\{Q_1,\ldots,Q_m\}$ for R, find a fragmentation $\mathscr{F}$ of R such that all the following conditions hold:

1. $\mathscr{F}$ correctly enforces $\mathscr{C}$ (Definition 4.5);
2. $\mathscr{F}$ maximizes visibility (Definition 4.6);
3. $\nexists \mathscr{F}'$ satisfying the conditions above and such that $Cost(\mathscr{Q},\mathscr{F}') < Cost(\mathscr{Q},\mathscr{F})$.

Like Problems 4.1 and 4.2, the minimum cost problem is *NP-hard*, as formally stated by the following theorem

Theorem 4.11. *The minimum cost problem is NP-hard.*

Proof. The proof is a reduction from the NP-hard minimum hitting set problem [50], which can be formulated as follows: *given a collection C of subsets of a set S, find the smallest subset S' of S such that S' contains at least one element from each subset in C.*

The reduction of the hitting set problem to the minimum cost problem can be defined as follows. Let S' be the solution of the minimum hitting set problem, let $R = S$ be a relation composed of only binary attributes where 0 and 1 values are equally distributed, and let $\mathcal{Q}$ be the query workload of the system.

As for the proof of Theorem 4.8, we consider only the sets s_i in C with cardinality greater than 1 and such that there does not exists $s_j \in C$, $s_j \subset s_i$. Let $\mathscr{C}_f = \{s \in C: |s| > 1 \text{ and } \forall s' \in C, s' \not\subset s\}$ be the set of association constraints, and let $\mathscr{A}_f = \{a \in R: \{a\} \notin C\}$ be the set of attributes to be fragmented. We note that the construction of the set of constraints $\mathscr{C}_f$ is polynomial in C. Also, by construction, $\mathscr{C}_f$ is well defined and does not contain singleton constraints.

Let us now suppose that $\mathcal{Q}=\{Q\}$, with Q="SELECT * FROM R WHERE $\bigwedge_{a_i \in \mathscr{A}_f}$ (a_i=0)" and $freq(Q)= 1$. Since the attribute values are equally distributed, the selectivity of all the conditions in Q is the same. As a consequence, the cost of Q with respect to an arbitrary fragment F is proportional to the number of attributes in the fragment itself. The fragment F in a fragmentation $\mathscr{F}$ that minimizes the cost with respect to the given query is therefore the one containing the maximum number of attributes. As described in the proof of Theorem 4.8, computing the fragment with the maximum cardinality corresponds to solve the minimum hitting set problem, since $S' = R - F$.

4.10 A Heuristic Approach to Minimize Query Cost Execution

The two heuristic algorithms proposed in previous sections are not suited for solving Problem 4.3, since they do not take into account the advantage that arises in having sets of plaintext attributes appearing in the same fragment. Due to the monotonicity of the cost function introduced in the previous section with respect to the dominance relationship (see Lemma 4.4), the complete search algorithm proposed in Sect. 4.5 could also be used to compute a solution for Problem 4.3. In this case, function **Evaluate** should implement the $Cost(\mathcal{Q},\mathscr{F})$ function. The complete search algorithm remains however exponential in the number of attributes. While this may not be an issue for small schemas, it may make the algorithm not applicable for complex schemas. We then propose a heuristic algorithm working in polynomial time.

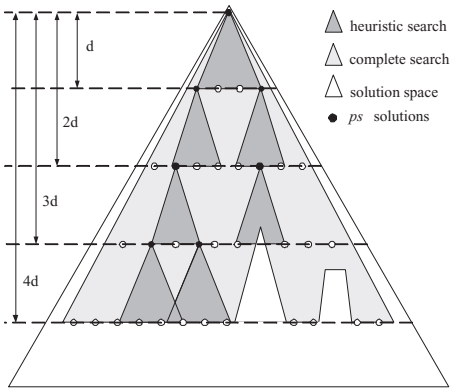


Fig. 4.14 Depiction of the search spaces

4.10.1 Computing a Vector-minimal Fragmentation with the Cost Function

Our heuristic is based on a variant of the depth-first search algorithm proposed for the complete search, where a selected number of subtrees composing the fragmentation tree are visited following the same strategy proposed for the complete search algorithm. As shown in Fig. 4.14, the fragmentation lattice is logically divided into $\lceil \frac{n}{d} \rceil$ bands, where:

- n is the cardinality of $\mathscr{A}_f$;
- d is a parameter indicating the number of levels in the tree completely visited at each step;[3]
- ps is a parameter indicating number of promising fragmentations explored at each step.

The first subtree of depth d is built considering as a root vertex the top element $\mathscr{F}_\top$ of the lattice. At level $x \cdot d$, ps subtrees are visited (where ps is another parameter of the heuristic), taking as a root one of the fragmentations computed at level $x \cdot d$. These visits artificially stop at level $(x+1) \cdot d$, where the best ps solutions are chosen as the root for the next in-depth visits of the solution space.

The function in Fig. 4.15 takes as input the set $\mathscr{A}_f$ of attributes to be fragmented, the set $\mathscr{C}_f$ of well defined non singleton constraints, and d and ps additional parameters. It computes a vector-minimal fragmentation *Min* of $\mathscr{A}_f$, by visiting a subset of the fragmentations in $\mathfrak{F}$.

The algorithm uses variables: $marker[\mathscr{F}]$, representing the position of the marker within fragmentation $\mathscr{F}$; *Min*, representing the current minimal fragmentation; *MinCost*, representing the number of fragments composing *Min*; *currentqueue*, containing the best ps fragmentations at level $x \cdot d$ that represent the roots of the subtrees

[3] If d is equal to $|\mathscr{A}_f|$ the heuristic approach degenerates in a complete search.

```
FRAGMENT(A_f,C_f,d,ps)
nextqueue:= NULL /* priority queue of promising solutions */
currentqueue:= NULL /* queue containing the best ps solutions */
for each a_i∈A_f do F_i^⊤ := {a_i} /* root of the search tree F_⊤ */
marker[F_⊤] := 1 /* next fragment to be merged */
Min := F_⊤ /* current minimal fragmentation */
MinCost := Cost(Q,Min)
/* compute the best ps solution within d levels from F_⊤ */
insert(nextqueue,Min,MinCost)
while nextqueue≠NULL do
    i := 1
    while (i≤ps)∧(nextqueue≠NULL) do
        i := i+1
        enqueue(currentqueue,extractmin(nextqueue))
    nextqueue := NULL
    while currentqueue≠NULL do
        F := dequeue(currentqueue)
        marker[F] := 1
        BoundedSearchMin(F,d)
return(Min)

BOUNDEDSEARCHMIN(F_p,dist)
localmin := true /* minimal correct fragmentation */
for i=marker[F_p]...(|F_p|-1) do
  for j:=(i+1)...|F_p| do
    if F_i^p.last<_A F_j^p.first then /* F_i^p fully precedes F_j^p */
      for l=1...|F_p| do
        case:
            (l<j ∧ l≠i): F_l^c := F_l^p
            (l>j):       F_{l-1}^c := F_l^p
            (l=i):       F_l^c := F_i^p F_j^p
      marker[F_c] := i
      if SatCon(F_i^c) then
        localmin := false
        if dist= 1 then
          insert(nextqueue,F_c,Cost(Q,F_c))
        else
          BoundedSearchMin(F_c,dist−1) /* recursive call */
if localmin then
  cost := Cost(Q,F_p)
  if cost<MinCost then
    MinCost := cost
    Min := F_p
```

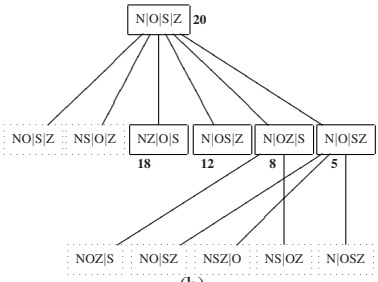
Fig. 4.15 Function that finds a vector-minimal fragmentation with minimal cost

to be visited; and *nextqueue*, containing, in increasing cost order, the correct fragmentations at level $(x+1)\cdot d$ computed by the visits of the subtrees rooted at the solutions in *currentqueue*. At start, the algorithm initializes variable *Min* to $\mathscr{F}_\top$ and variable *MinCost* to the cost of $\mathscr{F}_\top$. Then, the algorithm calls function **BoundedSearchMin** on $\mathscr{F}_\top$ that iteratively builds the children of $\mathscr{F}_\top$ according to Definition 4.11. Function **BoundedSearchMin**($\mathscr{F}_p$) is then recursively called on each $\mathscr{F}_c$, child of $\mathscr{F}_p$, only if $\mathscr{F}_c$ satisfies all the constraints (i.e., if function **SatCon** returns *true*) and level d has not been reached. In this latter case, if $\mathscr{F}_c$ is correct, it is inserted in *nextqueue*. Note that the function exploits the monotonicity of the cost function adopted and compares the cost of $\mathscr{F}_p$ with *Min* only if $\mathscr{F}_p$ is locally minimal (i.e., it does not have correct children).

When the subtree rooted at $\mathscr{F}_\top$ has been visited, the first *ps* fragmentations in *nextqueue* become the content of *currentqueue* and *nextqueue* is re-initialized to

Bounded($\mathscr{F}_p$,*dist*)	F_i^p	F_j^p	$\mathscr{F}_c$	**SatCon**(F_i^c)	**Bounded**($\mathscr{F}_c$,*dist*)	*Cost*($\mathscr{Q}$,$\mathscr{F}_c$)	*Min*	*nextqueue*
N|O|S|Z,1	N	O	NO|S|Z	*false*	–			
		S	NS|O|Z	*false*	–			
		Z	NZ|O|S	*true*	–	18		NZ|O|S,18
	O	S	N|OS|Z	*true*	–	12		N|OS|Z,12
		Z	N|OZ|S	*true*	–	8		N|OZ|S,8
	S	Z	N|O|SZ	*true*	–	5		N|O|SZ,5
N|O|SZ,1	N	O	NO|SZ	*false*	–			
		SZ	NSZ|O	*false*	–			
	O	SZ	N|OSZ	*false*	–	5	N|O|SZ	
N|OZ|S,1	N	OZ	NOZ|S	*false*	–			
		S	NS|OZ	*false*	–			
	OZ	S	–	–	–	8		

(a)



(b)

Fig. 4.16 An example of the execution of function **Fragment** in Fig. 4.15

NULL. Function **BoundedSearchMin** is then called for each $\mathscr{F} \in$*currentqueue*, but moving back the marker of $\mathscr{F}$ to its first fragment. The re-initialization of the marker implies that, for the root fragmentation $\mathscr{F}$ of each subtree, all the fragmentations that represent a child of $\mathscr{F}$ in the lattice are re-evaluated, but possibly not in the order-based cover exploited by the complete search. We note that this strategy could visit more than once the same vertex in the lattice. However, the maximum number of times that a fragmentation can be generated is *ps*. When *currentqueue* becomes empty, it is replaced with the first *ps* fragmentations in *nextqueue*, until the last layer in the tree is reached.

Example 4.9. Figure 4.16 illustrates the execution, step by step, of function **BoundedSearchMin** (**Bounded** for short) applied to Example 4.1, assuming $d = 1$ and $ps = 2$. The table in Fig. 4.16(a) describes, for each (recursive) call to **BoundedSearchMin**, the updates to the variables as well as to *nextqueue*. Therefore, the table in Fig. 4.16(a) has the same structure as the table in Fig. 4.7(a), except for the last column, which is dedicated to *nextqueue*, and for the column dedicated to the number of fragments in the solution, which is substituted here by the cost of the same. Fig. 4.16(b) illustrates the portion of the lattice visited by the algorithm. At the beginning variable *Min* is initialized to [N|O|S|Z], which is the fragmentation representing the root of the tree, the cost *MinCost* is initialized to 20, and *nextqueue* is initially empty. First, function **BoundedSearchMin** is called on [N|O|S|Z], with

dist = 1. Since *dist*−1 is 0, the fragmentations generated from [N|O|S|Z] and satisfying constraints do not cause a recursive call to **BoundedSearchMin**, but they are inserted in *nextqueue* after the evaluation of their cost. Then, **BoundedSearchMin** is called on the first two fragmentations extracted from *nextqueue*, that is, [N|O|SZ] and [N|OZ|S]. The final fragmentation computed by the heuristic algorithm is the same computed by **SearchMin**.

4.10.2 Correctness and Complexity

We now evaluate the correctness and the complexity of function **Fragment** in Fig. 4.15.

Theorem 4.12 (Correctness). *Function* **Fragment** *in Fig. 4.15 terminates and finds a vector-minimal fragmentation* (*Definition 4.12*).

Proof. Function **Fragment** terminates if all the **while** loops composing it terminate. The external **while** loop terminates when *nextqueue* is empty, provided the two internal loops terminate. The first internal loop terminates since variable *i* is increased by one at each step. It terminates when $i > ps$. The second internal **while** loop terminates since, at each iteration, an element is extracted from *currentqueue* and function **BoundedSearchMin** terminates. Indeed, function **BoundedSearchMin** at each recursive call, combines two of the fragments in the parent to compute its children and the recursion terminates, since at each call *dist* is decreased by one. Since **BoundedSearchMin** terminates, the number of items inserted in *nextqueue* is finite. Also, the number of layers in the fragmentation tree is finite. Therefore, *nextqueue* becomes empty and **Fragment** terminates.

We now prove that a solution $\mathscr{F}$ computed by this function over $\mathscr{A}_f$ and $\mathscr{C}_f$ is a vector-minimal fragmentation. According to Definition 4.12 of minimality, a fragmentation $\mathscr{F}$ is *vector-minimal* if and only if (1) it is correct, (2) it maximizes visibility, and (3) $\nexists \mathscr{F}' : \mathscr{F} \prec \mathscr{F}'$ that satisfies the two conditions above. The first two properties come directly from the proof of Theorem 4.4, since function **BoundedSearchMin** works exactly as **SearchMin** when generating candidate solutions. We need only to prove the third property.

By contradiction, let $\mathscr{F}'$ be a fragmentation satisfying the constraints in $\mathscr{C}_f$ and maximizing visibility, such that $\mathscr{F} \prec \mathscr{F}'$. Let $V_{\mathscr{F}}$ and $V_{\mathscr{F}'}$ be the fragment vectors associated with $\mathscr{F}$ and $\mathscr{F}'$, respectively. As already proved in the proof of Theorem 4.6, $\mathscr{F}'$ contains a fragment $V_{\mathscr{F}'}[a_i]$ that is the union of two different fragments, $V_{\mathscr{F}}[a_i]$ and $V_{\mathscr{F}}[a_j]$, of $\mathscr{F}$. We need then to prove that function **Fragment** cannot terminate with two different fragments whose union does not violate any constraint.

There are two different situations when invoking **BoundedSearchMin**($\mathscr{F}$,*dist*), that is, *dist*$>$ 1 or *dist*$=$ 1. In the first case, $\mathscr{F}'$ is generated and **BoundedSearchMin**($\mathscr{F}'$,*dist* − 1) called. In the second case, $\mathscr{F}'$ is generated and inserted in *nextqueue*. Since *nextqueue* is an ordered queue, **BoundedSearchMin**($\mathscr{F}'$,*dist*) is called only if there are no more than *ps* solution

with cost lower than *nextqueue*. But if $\mathscr{F}$ is returned as a solution of **Fragment**, no solution in *nextqueue* has lower cost than $\mathscr{F}$, since **BoundedSearchMin**($\mathscr{F}''$) is called for each $\mathscr{F}'' \in$*nextqueue*. This generates a contradiction since, from Lemma 4.4, $Cost(\mathscr{Q},\mathscr{F}') \leq Cost(\mathscr{Q},\mathscr{F})$.

Therefore the solution $\mathscr{F}$ computed by **Fragment** in Fig. 4.15 is a vector-minimal fragmentation.

Theorem 4.13 (Complexity). *Given a set of constraints* $\mathscr{C}=\{c_1,\ldots,c_m\}$*, a set of attributes* $\mathscr{A}=\{a_1,\ldots a_n\}$*, and the two parameters d and ps, the complexity of function* **Fragment**$(\mathscr{A},\mathscr{C},d,ps)$ *in Fig. 4.15 is* $O(\frac{ps}{d}n^{2d+2}m)$ *in time.*

Proof. The maximum number of iterations for the external **while** loop in function **Fragment** is $O(\frac{n}{d})$, since the fragmentation tree is composed of n layers and, at each iteration, solutions inserted in *nextqueue* are d layers under the solutions currently in *nextqueue*. Function **BoundedSearchMin**($\mathscr{F}_p$,d) is recursively called for each $\mathscr{F}_p \in$*currentqueue*, which contains at most *ps* solutions, since it is filled in during the preceding **while** loop. Function **BoundedSearchMin**, which behavior is similar to function **SearchMin**, visits the solutions in the subtree rooted at $\mathscr{F}_p$ within d layers. Therefore, the number of solutions built at each recursion of **BoundedSearchMin**($\mathscr{F}_p$,d) is $O(n^{2d})$ and each generated solution is compared with constraints in $\mathscr{C}$. The overall time complexity is therefore $O(\frac{ps}{d}n^{2d+2}m)$.

4.11 Query Execution

Fragmentation of a relation R implies that only fragments, which are stored in place of the original relation to satisfy confidentiality constraints, are used for query execution. The fragments can be stored on a single server or on multiple servers. The server (or servers) storing the fragments while needs not to be trusted with respect to the confidentiality, since accessing single fragments or encrypted information does not expose to any privacy breach, it is trusted for correctly evaluating queries on fragments (honest-but-curious).

Users who are not authorized to access the content of the original relation R have only a *partial view* on the data, meaning that they can only access the fragments. A query submitted by a user with a partial view can be presented directly to the server(s) storing the desired fragment. Users who are authorized to access the content of the original relation have a *full view* on the data and can present queries referring to the schema of the original relation. The queries issued by users with full view are then translated into equivalent queries operating on the encrypted and fragmented data stored on the server(s). The translation process is executed by a trusted component, called *query mapping component*, invoked every time there is the need to access sensitive information (see Fig. 4.17). In particular, the query mapping component receives a query Q submitted by a user with full view along with the key k possibly needed for decrypting the query result computed by the server, and returns the result of query Q to the user. Since every physical fragment of R

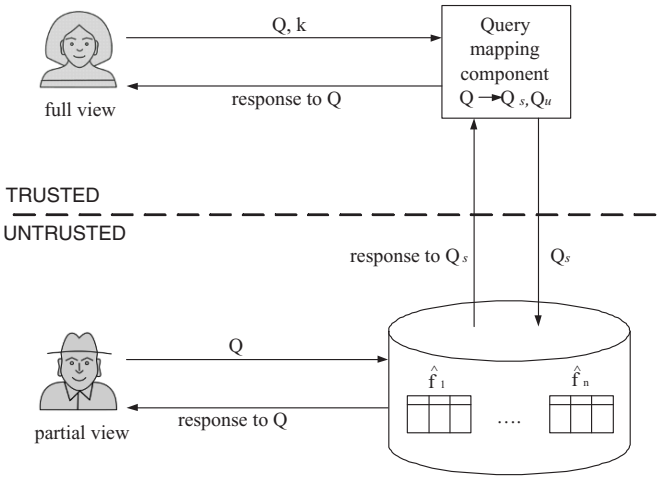


Fig. 4.17 Interactions among users and server storing the fragments

contains all the attributes of R, either in encrypted or in clear form, no more than one fragment needs to be accessed to respond to Q. The query mapping component therefore maps the user's query Q onto an equivalent query Q_s, working on a specific fragment. The server executes the received query Q_s on the required fragment and returns the result to the query mapping component. Note that, whenever query Q may involve attributes that do not appear in the clear form in the selected fragment, the query mapping component may need to execute an additional query Q_u on the decrypted results of Q_s, which is in charge of enforcing all conditions that cannot be evaluated on the physical fragment or of projecting the attributes reported in the SELECT clause of query Q. In this case, the query mapping component decrypts the result received, executes query Q_u on it, and returns the result of Q_u to the user. We now describe the query translation process in more details.

We consider *select-from-where* SQL queries of the form $Q =$"SELECT A_Q FROM R WHERE C", where A_Q is a subset of the attributes of R, and C is a conjunction of basic conditions $c_1 \ldots c_n$ of the form $(a\ op\ v)$ or $(a_j\ op\ a_k)$, with a, a_j, and a_k attributes of R, v constant value, and op comparison operator in $\{=, \neq, >, <, \leq, \geq\}$. Let us then consider the evaluation of query Q on physical fragment $\hat{F}_i(\underline{salt}, enc, a_{i_1}, \ldots, a_{i_n})$, where *salt* is the primary key, *enc* contains the encrypted attributes, and $a_{i_1}, \ldots, a_{i_n}$ are the plaintext attributes (see Sect. 4.3). Suppose, for generality, that C contains some conditions that involve attributes stored in the clear form in $\hat{F}_i$ and some others that cannot instead be evaluated on $\hat{F}_i$. The query mapping component translates the original query Q into a query Q_s operating on the physical fragment and defined as:

SELECT $A_Q \cap \{a_{i_1}, \ldots, a_{i_n}\}$, *salt*, *enc*
FROM $\hat{F}_i$
WHERE $\bigwedge_{c_j \in C_i^e} c_j$

Original query on R	Translation over encrypted fragments
Q := SELECT SSN, Name FROM Patient WHERE Sickness='Latex al.' AND ZIP='94140'	$Q_{s.3}$:= SELECT salt, enc FROM $\hat{F}_3$ WHERE Sickness='Latex al.' AND ZIP='94140' Q_u := SELECT SSN, Name FROM *Decrypt*($Q_{s.3}$, *Key*)
Q′ := SELECT SSN, Name FROM Patient WHERE Sickness='Latex al.' AND ZIP='94140' AND Occupation='Nurse'	$Q'_{s.3}$:= SELECT salt, enc FROM $\hat{F}_3$ WHERE Sickness='Latex al.' AND ZIP='94140' Q'_u := SELECT SSN, Name FROM *Decrypt*($Q'_{s.3}$, *Key*) WHERE Occupation='Nurse'

Fig. 4.18 An example of query translation over a fragment

where C_i^e is the set of basic conditions in C that can be evaluated on physical fragment $\hat{F}_i$, that is, $C_i^e = \{c_j : c_j \in C \wedge attributes(c_j) \in \hat{F}_i\}$, with $attributes(c_j)$ representing the attributes appearing in c_j. Note that the *salt* and *enc* attributes in the SELECT clause of Q_s are specified only if the SELECT or WHERE clauses of the original query Q involve attributes not appearing in clear form in the fragment. The query mapping component then decrypts the tuples received and executes on them a query Q_u defined as:

SELECT A_Q
FROM *Decrypt*(Q_s, k)
WHERE $\bigwedge_{c_j \in \{C - C_i^e\}} c_j$

where *Decrypt*(Q_s, k) denotes a temporary relation including the tuples returned by Q_s and where attribute *enc* has been decrypted through key k. The WHERE clause of Q_u includes all conditions defined on attributes that do not appear in clear form in the physical fragment and that can be only evaluated on the decrypted result. The final result of query Q_u is then returned to the user.

Note that since we are interested in minimizing the query evaluation cost, a query optimizer can be used to select the fragment that allows the execution of more selective queries by the server, thus decreasing the workload of the application and maximizing the efficiency of the execution [25]. For instance, the physical fragment $\hat{F}_i$ exploited by Q_s can be conveniently chosen as the fragment minimizing *Cost*(Q,F_i) as defined in Sect. 4.9.

Example 4.10. Consider the relation in Fig. 4.1(a) and its fragments in Fig. 4.2.

- Consider a query Q retrieving the Social Security Number and the name of the patients whose Sickness is *Latex al.* and whose ZIP is 94140. Since fragment $\hat{F}_3$ contains both Sickness and ZIP, it can evaluate both the conditions in the WHERE clause and is chosen for query evaluation. Figure 4.18 illustrates the

translation of Q to queries $Q_{s.3}$ executed by the server on the fragment (notation $Q_{s.x}$ indicates a query executed by the server on fragment x), and Q_u executed by the application. Query $Q_{s.3}$ returns to the application only the tuples belonging to the final result. The application just needs to decrypt them for projecting attributes `SSN` and `Name`.

- Consider a query Q′ retrieving the Social Security Number and the name of the patients whose `Sickness` is *Latex al.*, whose `ZIP` is 94140, and whose occupation is *Nurse*. Fragment $\hat{F}_3$ contains both `Sickness` and `ZIP` and $S(Q',F_3)=1/6$. Fragment F_2 contains only `Occupation` and $S(Q',F_3)=1/3$. The query mapping component therefore translates query Q′ into queries $Q'_{s.3}$ executed by the server on the fragment, and Q'_u executed by the application (see Fig. 4.18). Since `ZIP` does not appear in clear form in fragment $\hat{F}_3$, the condition on it needs to be evaluated by the application, which also performs the projection of the `SSN` and `Name` attributes after decrypting the result computed by $Q_{s.3}$.

Note that queries whose WHERE clause contains negated conditions can be easily managed by the query mapping component since whenever a basic condition c can be evaluated on a physical fragment, also its negation (i.e., NOT(c)) can be evaluated on the same fragment. Queries whose WHERE clause contains disjunctions need special consideration. As a matter of fact, according to the semantics of the OR operator, any condition that cannot be evaluated over a fragment but that is in disjunction with other conditions that can be evaluated on the fragment cannot be simply evaluated on the result returned by the server (like done in the case of conjunction). Three scenarios are then possible. *1)* The query conditional part can be reduced to a conjunctive normal form; then the query mapping and evaluation can proceed as illustrated in the conjunctive case above. *2)* The query conditional part can be reduced to a disjunctive normal form where all components can be evaluated over different fragments; in this case the query mapping component will ask the server for the execution of as many queries as the components of the disjunction and will then merge (union) their results. *3)* The query conditional part contains a basic condition (to be evaluated in disjunction with others) that cannot be evaluated on any fragment (as it involves a sensitive attribute or attributes that appear in two different fragments); in this case the query mapping component will need to retrieve the entire fragment (any fragment will do) and evaluate the query condition at its site.

4.12 Indexes

As discussed in Sect. 4.3, each physical fragment reports in the clear only some of the attributes (as dictated by the fragmentation) while reporting the remaining attributes as a single encrypted tuple. This clearly has an impact on the performance of queries that need to evaluate selection predicates on both data appearing in clear and on data appearing in encrypted form (see Sect. 4.11). In the encrypted database proposals, queries on encrypted data are typically evaluated by means of indexes

built on encrypted attributes: each cleartext query is translated into a query on the indexes and the result (complete but maybe including spurious tuples) is then decrypted and filtered by a trusted client (see Fig. 4.17). As discussed in Chap. 2, different kinds of indexes have been proposed, each providing a different balance between efficiency and confidentiality. We distinguish here these methods in three main classes.

- **Direct index.** The index is obtained by applying an encryption (unsalted) function on the cleartext values of the attribute [58].
- **Hash index.** The index is obtained by applying a keyed hash function to the cleartext values and restricting the result to produce collisions [24].
- **Flattened hash index.** The index is obtained by applying a keyed hash function with collision as in the case of hash index while applying a post processing that flattens the distribution of index values (so to avoid exposures of outliers) [45, 96].

In the encrypted database scenario, direct indexes are the most efficient, as conditions on cleartext values have a one to one correspondence with conditions on indexed values; at the same time they exhibit a major vulnerability making them applicable only in restricted situations. Hash indexes may create exposure problems only in the presence of outliers or in the case of use of multiple indexes in the same table, but otherwise guarantee confidentiality. Flattened hash indexes provide better protection. While one may think that the same properties could hold for fragmentations, unfortunately the application of indexes to fragments (which, unlike encrypted databases, report some cleartext values) introduces new vulnerabilities. In this section we briefly discuss the vulnerabilities to the aim of identifying a safe use of indexes, which we apply to our scenario. For simplicity, in the discussion we refer to a simple fragmentation problem characterized by a relation $R(a_1, a_2)$ and by a single confidentiality constraint $\{a_1, a_2\}$. We then examine the exposure risk of a fragment where a_1 appears in the clear jointly with an index of a_2, for each of the above classes of indexes. An instance of such a configuration, to which we refer for concreteness in the examples, is table `Patient` in Fig. 4.1(a) restricted to attributes `Name` and `Sickness`, together with the confidentiality constraint on them (c_2). We then evaluate the protection of the fragment reporting `Name` (Fig. 4.2(a)) in the clear when indexes on attribute `Sickness` are added. Fig. 4.19(c–e) reports the indexed fragments under the different indexing assumptions.

To examine the vulnerability of the indexed fragments, we first need to identify the knowledge available to the adversary, whose aim is to reconstruct the protected association (`Name, Sickness`). We can identity two kinds of knowledge: *vertical knowledge* and *horizontal knowledge*, characterized as follows.

- **Vertical knowledge.** Vertical knowledge is due to the fact that the values not appearing in the clear in one fragment (for a confidentiality constraint forbidding their association with other values) may appear in the clear in other fragments. Vertical knowledge does not require any additional external information for the adversary since, apart from the case where the attribute appears in a singleton constraint, it refers to information immediately present in other accessible

Knowledge | Indexed fragment $\hat{f}_1$

Sickness
Latex al.
Latex al.
Latex al.
Celiac
Pollen al.
Nickel al.

(a) **vk**

Name	Sickness
A. Smith	Latex al.

(b) **hk**

salt	enc	Name	i_{s_1}
s_1	α	A. Smith	λ
s_2	β	B. Jones	λ
s_3	γ	C. Taylor	λ
s_4	δ	D. Brown	ϕ
s_5	ε	E. Cooper	π
s_6	ζ	F. White	ψ

(c) **di**

salt	enc	Name	i_{s_2}
s_1	α	A. Smith	σ
s_2	β	B. Jones	σ
s_3	γ	C. Taylor	σ
s_4	δ	D. Brown	ρ
s_5	ε	E. Cooper	σ
s_6	ζ	F. White	ρ

(d) **hi**

salt	enc	Name	i_{s_3}
s_1	α	A. Smith	η
s_2	β	B. Jones	η
s_3	γ	C. Taylor	η
s_4	δ	D. Brown	μ
s_5	ε	E. Cooper	μ
s_6	ζ	F. White	μ

(e) **fhi**

Fig. 4.19 Adversary knowledge (a,b) and choices for indexed fragments (c,d,e)

fragments (Fig. 4.2(c)). Figure 4.19(a) reports the vertical knowledge for our example, illustrating the projection of the `Sickness` attribute of Fig. 4.1(a). An adversary observing the fragments can then have complete knowledge of the distribution (cleartext values and their number of occurrences) of the indexed attributes. In the example, the observer knows that there are three patients with latex allergy.

- **Horizontal knowledge.** Horizontal knowledge is due to possible external knowledge that the adversary has with respect to the presence of specific tuples (corresponding to sensitive associations) in the table. In its simplest form, horizontal knowledge is then represented by knowledge of a single tuple (v_1, v_2). In the example, the adversary may know that A. Smith suffers from latex allergy, that is, (A. Smith, latex al.) belongs to the original table R. Figure 4.19(b) reports this example of horizontal knowledge.

Let us now examine the exposure risk of indexed fragments under the assumption of horizontal and vertical knowledge.[4]

Direct index, vertical knowledge (di-vk). Sensitive associations are exposed depending on their distinguishability with respect to the number of occurrences of the indexed values. In our example, the index corresponding to latex allergy is completely recognizable being the only one with three occurrences. Consequently, the adversary infers that A. Smith, B. Jones, and C. Taylor suffer from latex allergy. As for the other three patients, the adversary can estimate they suffer from one of the three other sicknesses, each with equal probability.

Direct index, horizontal knowledge (di-hk). By joining this knowledge on the attribute appearing in the clear in the indexed fragment (`Name`), the adversary can retrieve the index value λ corresponding to the specific cleartext value of the indexed attribute (`Sickness`). This exposes the associations having the same index value as the one the adversary knows. In our example, knowledge of the association (A. Smith, latex al.) allows the adversary to know that λ is the index for latex allergy and therefore to infer that also B. Jones, and C. Taylor suffer from latex allergy.

[4] We note that the treatment of vertical knowledge strictly resembles threat models, proposed for encrypted databases, that assume that the adversary had complete knowledge of the cleartext database and aimed at reconstructing the correspondence between cleartext and index values (scenario **Freq+DB$^{\mathbf{K}}$** in [24]).

Hash index, vertical knowledge (hi-vk). The use of the hash index diminishes the exposure of association since different cleartext values may be represented by the same index value. However, values with a high number of occurrences (outliers), typically remain recognizable. In the example, the adversary can infer that index σ refers to latex allergy, since it is the only one with at least 3 occurrences. She can then infer that 3 out of the 4 patients have latex allergy (i.e., each one has latex allergy with 0.75 probability).

Hash index, horizontal knowledge (hi-hk). Like in the direct index case, the adversary can recognize the index value representing the known cleartext value, with the only difference that the index value can correspond also to other cleartext values. The adversary can then infer that some associations are not present in the database (tuples with a different index value will certainly not have the known cleartext value). Together with vertical knowledge, it allows the adversary to infer the probability that some sensitive associations (with the known cleartext value) belong to the database. In the example, knowledge of the association (A. Smith, latex al.) allows the adversary to know that σ is the index for latex allergy. Since there are 3 occurrences of latex allergy and 4 occurrences of σ, by removing the known one, the adversary can infer that B. Jones, C. Taylor, and E. Cooper have a 0.66 probability of suffering from latex allergy.

Flattened hash index, vertical knowledge (fhi-vk). Flattening the occurrences of the index values makes impossible to establish correspondences between cleartext values and index values on the basis of the number of occurrences. Flattened hash indexes are not vulnerable to vertical knowledge.

Flattened hash index, horizontal knowledge (fhi-hk). Like in the hashed case, the adversary can recognize the index value representing the known cleartext value. Together with vertical knowledge, it allows the adversary to identify the subset of tuples that may be associated with the cleartext value for which the index is known, with an estimate of the probability of their association. In the example, knowledge of the association (A. Smith, latex al.) allows the adversary to know that η is the index for latex allergy and therefore to infer that B. Jones and C. Taylor have a 1.0 probability of suffering from latex allergy (since there are only three occurrences of latex allergy).

In summary, vertical and horizontal knowledge create inference risks on the basis of the number of occurrences of cleartext (and corresponding index) values. Even when values are equally distributed, all indexes above remain vulnerable to horizontal knowledge, allowing the adversary to infer associations with the known cleartext value. It is easy to see that such vulnerabilities are blocked when values are equally distributed and horizontal knowledge refers to association with indexed values that have only one occurrence. Both conditions are certainly satisfied when indexes refer to key attributes. Without compromising confidentiality of fragments, we can therefore apply indexes on attributes corresponding to candidate keys of the original relations.

Indexes can be easily integrated in our cost model, by simply refining $Cost(\mathcal{Q},\mathcal{F})$ function. This can easily be done by considering the selectivity of indexes for con-

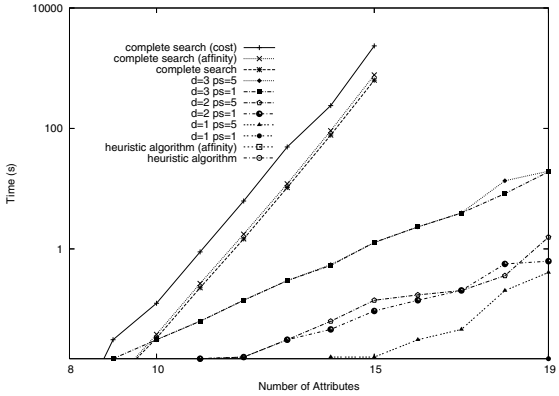


Fig. 4.20 Computational time of the algorithms

ditions on indexed values. Indexes do not have any impact on the monotonicity property of the cost function on fragments (Lemma 4.4) and therefore on the applicability of our solutions. With reference to our example we can then consider direct indexes on `SSN` and `Name` (assuming `Name` is a candidate key) in any fragment where they appear encrypted (all fragments for `SSN` and those in Fig. 4.2(b) and Fig. 4.2(c) for `Name`).

4.13 Experimental Results

The heuristic algorithms presented in Sects. 4.6, 4.8, and 4.10 have been implemented as C programs to obtain experimental data and assess their behavior in terms of execution time and quality of the returned solution. Aiming to a comparison of the results computed by our heuristic algorithms to the optimal solutions, we also implemented three versions of the algorithm presented in Sect. 4.5, analyzing the complete solution space computing the fragmentation with the minimal number of fragments, the one with maximum affinity, and the one with minimum cost, since all these three functions are monotonic with respect to $\preceq$. The relation schema we considered in the experiments is composed of 19 attributes and is inspired by a database of medical information. Taking into account possible confidentiality requirements we expressed up to 18 confidentiality constraints. These constraints are well defined (see Definition 4.2) and composed of a number of attributes varying from 2 to 4 (we did not consider singleton constraints as they cannot be solved via fragmentation). The content of the affinity matrix has been produced using a pseudo-random generation function. We considered 14 queries, each characterized by a frequency value. The experiments have considered configurations with an increasing number of attributes, from 3 to 19, taking into account, for every configuration, only the constraints completely fitting in the selected attributes. The number of constraints

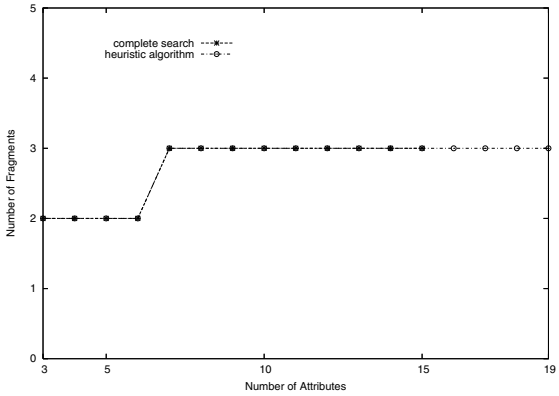


Fig. 4.21 Number of fragments of the solution produced by the algorithms

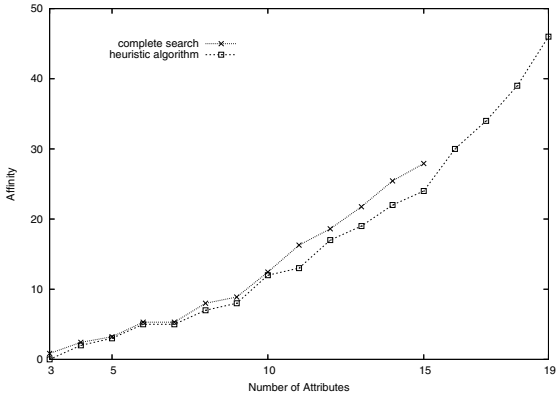
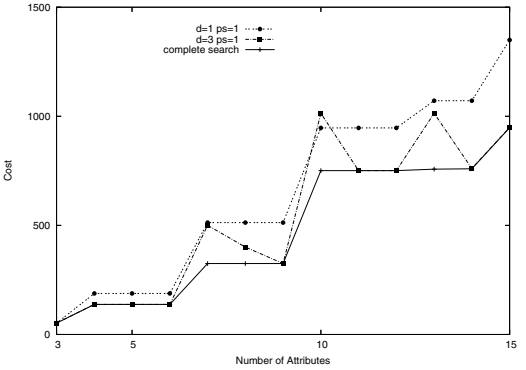


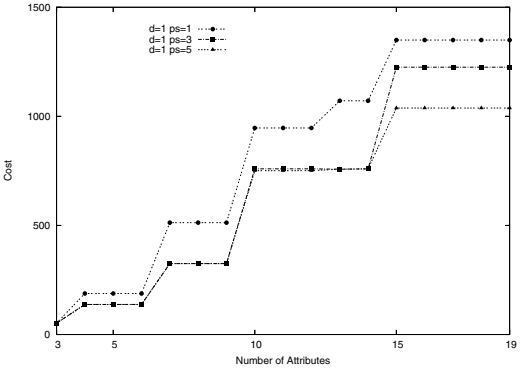
Fig. 4.22 Affinity of the solution produced by the algorithms

for a configuration with n attributes ranges between $n-3$ to $n+1$. The system implemented presents as an option the use of indexes, according to the analysis of Sect. 4.12.

Figure 4.20 compares the time required for the execution of the complete search algorithms with the heuristic algorithms presented in this chapter. Consistently with the fact that the problem of minimizing the number of fragments, the problem of maximizing affinity, and the problem of minimizing cost while satisfying confidentiality constraints are NP-hard, the three complete search strategies require exponential time in the number of attributes. The complete search then becomes unfeasible even for a relatively small number of attributes; with the availability of large computational resources it would still not be possible to consider large configurations (in our experiments we were able only to run the complete search for schemas with less than 15 attributes). By contrast, the time required for the execution of the heuristic analysis always remains low. The heuristic functions computing the vector minimal



(a)



(b)

Fig. 4.23 Cost of the solution produced by the algorithms

fragmentation and the vector minimal fragmentation maximizing affinity have computational time near 0. On the other hand, the time required by the heuristic for the minimum cost fragmentation problem increases exponentially with the increase in the look-ahead depth and linearly with the increase in the number of parallel steps, always showing a limited time for the simplest search (d=1,ps=1). It is therefore important to have available a family of heuristics, so to apply in real systems a dynamic approach where initially a search is executed with the most efficient heuristic, increasing the depth according to the amount of available resources. The number of parallel steps is a parameter that should become particularly interesting for the implementation of the heuristics on a multi-core architecture, where each core can manage the exploration of one of the alternatives.

Obviously, a successful heuristics presents a good behavior if it combines time efficiency with a demonstrated ability to produce good solutions. We therefore compared the solutions computed by the execution of each of the heuristic algorithms with those returned by the corresponding complete search algorithms.

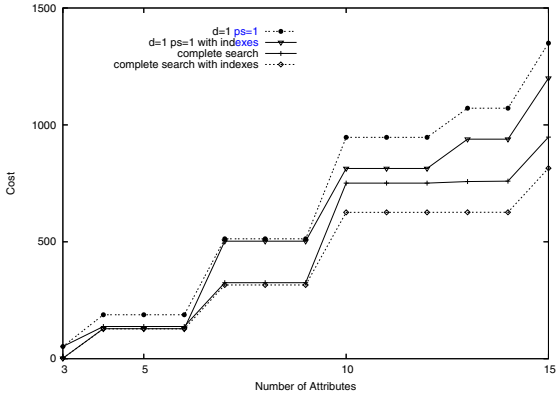


Fig. 4.24 Cost of the solution with indexes

Figure 4.21 presents the number of fragments obtained by the execution of the heuristic algorithm computing a vector-minimal fragmentation (Sect. 4.6) compared with in a solution computed by the complete search function. As the graph shows, in all the cases that allow the comparison, our heuristic has always identified an optimal solution.

Figure 4.22 instead compares the affinity of the fragmentation computed through our heuristic (Sect. 4.8) with the optimal affinity produced by the complete search strategy. As the graph shows, for all the cases that allow the comparison, the affinity of the solution computed by the heuristic algorithm is close to the optimal value: the average of the difference is 4.2% and the maximum percentage difference is around 14.1%.

Figure 4.23(a) compares the cost of the solution obtained by our heuristic algorithm (Sect. 4.10) in two configurations: ($d = 1, ps = 1$) and ($d = 3, ps = 1$) with the optimal cost produced by the complete search strategy. The graph shows that even the simplest configuration ($d = 1, ps = 1$) guarantees good-quality fragmentations. Figure 4.23(b) shows the cost of the solutions produced by the heuristic with different values for parameter ps (i.e., 1, 3, and 5) and with the fixed value $d = 1$. It is sufficient to use $ps = 5$ to obtain near-optimum fragmentations.

Finally, experiments have been run to evaluate the benefit of indexes and they have proved (see Fig. 4.24) that the use of indexes on encrypted attributes can produce a significant benefit. The amount of the benefit is highly dependent on specific features of the relation schema and query profile.

4.14 Chapter Summary

We presented an approach combining fragmentation and encryption to efficiently enforce privacy constraints over data collections, with particular attention to query

execution efficiency. The algorithms proposed for fragmentation take into account the information available about the system, to the aim of efficiently executing queries on the fragmented data.

Besides the technical contribution, the ideas illustrated in this chapter can represent a step towards the effective enforcement, as well as the establishment, of privacy regulations. Technical limitations are in fact claimed as one of the main reasons why privacy cannot be achieved and, consequently, regulations not be put into enforcement. Research along the line presented here can then help in providing the building blocks for a more precise specification of privacy needs and regulations, as well as their actual enforcement, together with the benefit of a clearer and more direct integration of privacy requirements within existing ICT infrastructures.