

Chapter 2

Overview of the State of the Art

This chapter discusses the state of the art in the area of data outsourcing, which is mainly focused on efficient methods for querying encrypted data. We also present some approaches for evaluating the inference exposure due to data publication, and solutions for granting data integrity. A few research efforts have instead addressed the problem of developing access control systems for outsourced data and for securely querying distributed databases.

2.1 Introduction

The amount of information held by organizations' databases is increasing very quickly. To respond to this demand, organizations can:

- add data storage and skilled administrative personnel (at a high rate);
- delegate database management to an external service provider (*database outsourcing*), a solution becoming increasingly popular.

In the database outsourcing scenario, usually referred to as *Database As a Service* (DAS), the external service provider provides mechanisms for clients to access the outsourced databases. A major advantage of database outsourcing is related to the high costs of in-house versus outsourced hosting. Outsourcing provides significant cost savings and promises higher availability and more effective disaster protection than in-house operations. On the other hand, database outsourcing poses a major security problem, due to the fact that the external service provider, which is relied upon for ensuring high availability of the outsourced database (i.e., it is trustworthy), cannot always be trusted with respect to the confidentiality of the database content.

Besides well-known risks of confidentiality and privacy breaks, threats to outsourced data include improper use of database information: the server could extract, resell, or commercially use parts of a collection of data gathered and organized by the data owner, potentially harming the data owner's market for any product or service that incorporates that collection of information. Traditional database access

control techniques cannot prevent the server itself from making unauthorized access to the data stored in the database. Alternatively, to protect against “honest-but-curious” servers, a protective layer of encryption can be wrapped around sensitive data, preventing outside attacks as well as infiltration from the server itself [38]. This scenario raises many interesting research challenges. First, data encryption introduces the problem of efficiently querying outsourced encrypted data. Since confidentiality demands that data decryption must be possible only at the client-side, techniques have then been proposed, enabling external servers to directly execute queries on encrypted data. Typically, these solutions consist mainly in adding a piece of information, called *index*, to the encrypted data. Indexes are computed based on the plaintext data and preserve some of the original characteristics of the data to allow (partial) query evaluation. However, since indexes carry some information about the original data, they may be exploited as inference channels by malicious users or by the service provider itself. Second, since data are not under the owner’s direct control, unauthorized modifications must be prevented to the aim of granting data integrity. For this purpose, different solutions based on different signature mechanisms have been proposed, with the main goal of improving verification efficiency. Third, although index-based solutions represent an effective approach for querying encrypted data, they introduce an overhead in query execution, due to both query formulation through indexes and data decryption and filtering of query results. However, since often what is sensitive in a data collection is the association among attributes more than the values assumed by each attribute per se, new solutions based on the combination of fragmentation and encryption have been proposed to reduce the usage of encryption and to therefore increase query execution efficiency. Fourth, an interesting issue that has not been deeply studied in the data outsourcing scenario is represented by the access control enforcement, which cannot be delegated to the service provider. Finally, when the outsourced data are stored at different servers, new safe data integration mechanisms are needed that should take into consideration the different data protection needs of the cooperating servers.

2.1.1 Chapter Outline

In this chapter, we survey the main proposals addressing the data access and security issues arising in the data outsourcing scenario. The remainder of the chapter is organized as follows. Section 2.2 gives an overview of the entities involved in the data outsourcing scenario and of their typical interactions. Section 2.3 describes the main indexing methods proposed in the literature for supporting queries over encrypted data. Section 2.4 addresses inference exposure due to different indexing techniques. Section 2.5 focuses on techniques granting data integrity. Section 2.6 describes solutions efficiently combining fragmentation and encryption for granting privacy protection. Section 2.7 presents the main proposals for access control enforcement on outsourced encrypted data. Section 2.8 illustrates problems and so-

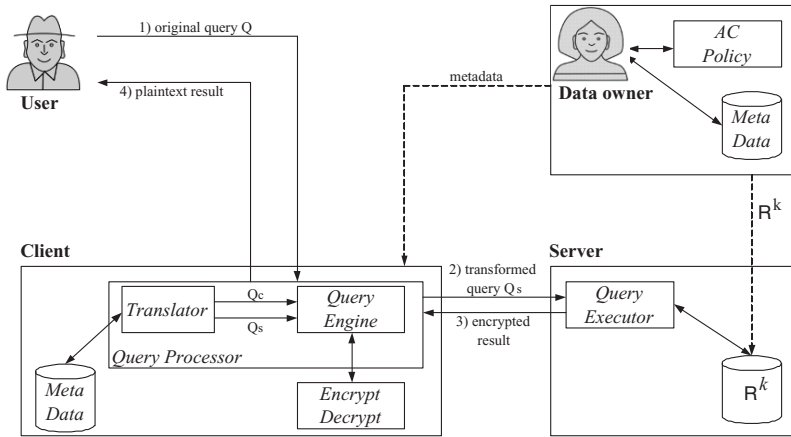


Fig. 2.1 DAS scenario

lutions for safe data integration in a distributed system. Finally, Sect. 2.9 concludes the chapter.

2.2 Basic Scenario and Data Organization

In this section, we describe the entities involved in the DAS scenario, how data are organized in the outsourced database context, and the interactions among the entities in the system for query evaluation.

2.2.1 Parties Involved

There are four distinct entities interacting in the DAS scenario (Fig. 2.1):

- a *data owner* (person or organization) produces and outsources resources to make them available for controlled external release;
- a *user* (human entity) presents requests (queries) to the system;
- a *client* front-end transforms the queries posed by users into equivalent queries operating on the encrypted data stored on the server;
- a *server* receives the encrypted data from one or more data owners and makes them available for distribution to clients.

Clients and data owners, when outsourcing data, are assumed to trust the server to faithfully maintain outsourced data. The server is then relied upon for the availability of outsourced data, so the data owner and clients can access data whenever requested. However, the server (which can be “honest-but-curious”) is not trusted

with the confidentiality of the actual database content, as outsourced data may contain sensitive information that the data owner wants to release only to authorized users. Consequently, it is necessary to prevent the server from making unauthorized accesses to the database. To this purpose, the data owner encrypts her data with a key known only to trusted clients, and sends the encrypted database to the server for storage.

2.2.2 Data Organization

A database can be encrypted according to different strategies. In principle, both symmetric and asymmetric encryption can be used at different granularity levels. Symmetric encryption, being cheaper than asymmetric encryption, is usually adopted. The granularity level at which database encryption is performed can depend on the data that need to be accessed. Encryption can then be at the finer grain of [55, 63]:

- *relation*: each relation in the plaintext database is represented through a single encrypted value in the encrypted database; consequently, tuples and attributes are indistinguishable in the released data, and cannot be specified in a query on the encrypted database;
- *attribute*: each column (attribute) in the plaintext relation is represented by a single encrypted value in the encrypted relation;
- *tuple*: each tuple in the plaintext relation is represented by a single encrypted value in the encrypted relation;
- *element*: each cell in the plaintext relation is represented by a single encrypted value in the encrypted relation.

Both relation level and attribute level encryption imply the communication to the requesting client of the whole relation involved in a query, as it is not possible to extract any subset of the tuples in the encrypted representation of the relation. On the other hand, encrypting at element level would require an excessive workload for data owners and clients in encrypting/decrypting data. For balancing client workload and query execution efficiency, most proposals assume that the database is encrypted at tuple level.

While database encryption provides an adequate level of protection for data, it makes impossible for the server to directly execute the users' queries on the encrypted database. Upon receiving a query, the server can only send to the requestor the encrypted relations involved in the query; the client needs then to decrypt such relations and execute the query on them. To allow the server to select a set of tuples to be returned in response to a query, a set of indexes can be associated with the encrypted relation. In this case, the server stores an encrypted relation with an index for each attribute on which conditions may need to be evaluated. For simplicity, we assume the existence of an index for each attribute in each relation of the database. Different kinds of indexes can be defined for the attributes in a relation, depending on the clauses and conditions that need to be remotely evaluated. Given a plaintext

EMPLOYEE					EMPLOYEE ^k						
Emp-Id	Name	YoB	Dept	Salary	Counter	Etuple	I ₁	I ₂	I ₃	I ₄	I ₅
P01	Ann	1980	Production	10	1	ite6Az*+8wc	π	α	γ	ε	λ
R01	Bob	1975	R&D	15	2	8(Xznfeua4!=	ϕ	β	δ	θ	λ
F01	Bob	1985	Financial	10	3	Q73gnew321*/	ϕ	β	γ	μ	λ
P02	Carol	1980	Production	20	4	-1vs9e892s	π	α	γ	ε	ρ
F02	Ann	1980	Financial	15	5	e32rfs4aS+@	π	α	γ	μ	λ
R02	David	1978	R&D	15	6	r43arg*5[]	ϕ	β	δ	θ	λ

(a)

(b)

Fig. 2.2 An example of plaintext (a) and encrypted (b) relation

database \mathcal{R} , each relation r_i over schema $R_i(a_{i1}, a_{i2}, \dots, a_{in})$ in \mathcal{R} is mapped onto a relation r_i^k over schema $R_i^k(\text{Counter}, \text{Etuple}, I_{i1}, I_{i2}, \dots, I_{in})$ in the corresponding encrypted database \mathcal{R}^k . Here, *Counter* is a numerical attribute added as primary key of the encrypted relation; *Etuple* is the attribute containing the encrypted tuple, whose value is obtained applying an encryption function E_k to the plaintext tuple, where k is the secret key; and I_{ij} is the index associated with the j -th attribute a_{ij} in R_i . While we assume encrypted tuples and indexes to be in the same relation, we note that indexes can be stored in a separate relation [35].

To illustrate, consider relation `Employee` in Fig. 2.2(a). The corresponding encrypted relation is shown in Fig. 2.2(b), where index values are conventionally represented with Greek letters. The encrypted relation has exactly the same number of tuples as the original relation. For the sake of readability, the tuples in the encrypted relation are listed in the same order with which they appear in the corresponding plaintext relation. The same happens for the order of indexes, which are listed in the same order as the corresponding attributes are listed in the plaintext relation schema. For security reasons, real-world systems do not preserve the order of attributes and tuples and the correspondence between attributes and indexes is maintained by metadata relations that only authorized parties can access [32].

2.2.3 Interactions

The introduction of indexes allows the partial evaluation of any query Q at the server-side, provided it is previously translated in an equivalent query operating on the encrypted database. Figure 2.1 summarizes the most important steps necessary for the evaluation of a query submitted by a user.

1. The user submits her query Q referring to the schema of the plaintext database \mathcal{R} , and passes it to the client front-end. The user needs not to be aware that data have been outsourced to a third party.
2. The client maps the user's query onto: *i*) an equivalent query Q_s , working on the encrypted relations through indexes, and *ii*) an additional query Q_c working on

the results of Q_s . Query Q_s is then passed on to the remote server. Note that the client is the unique entity in the system that knows the structure of both \mathcal{R} and \mathcal{R}^k and that can translate the queries the user may submit.

3. The remote server executes the received query Q_s on the encrypted database and returns the result (i.e., a set of encrypted tuples) to the client.
4. The client decrypts the tuples received and eventually discards spurious tuples (i.e., tuples that do not satisfy the query submitted by the user). These spurious tuples are removed by executing query Q_c . The final plaintext result is then returned to the user.

Since a client may have limited storage and reduced computation capacity, one of the primary goals of the query execution process is to minimize the workload at the client side, while maximizing the operations that can be computed at the server side [36, 55, 57, 63].

Iyer et al. [55, 63] present a solution for minimizing the client workload that is based on a graphical representation of queries as trees. Since the authors limit their analysis to *select-from-where* queries, each query $Q = \text{"SELECT } A \text{ FROM } R_1, \dots, R_n \text{ WHERE } C\text{"}$ can be reformulated as an algebra expression of the form $\pi_A(\sigma_C(R_1 \bowtie \dots \bowtie R_n))$. Each query can then be represented as a binary tree, where leaves correspond to relations R_1, \dots, R_n and internal nodes represent relational operations, receiving as input the result produced by their children. The tree representing a query is split in two parts: the lower part includes all operations that can be executed by the server, while the upper part contains all operations that cannot be delegated to the server and that therefore need to be executed by the client. In particular, since a query can be represented with different, but equivalent, trees by simply pushing down selections and postponing projections, the basic idea of the proposed solution is to determine a tree representation of the query, where the operations that only the client can execute are in the highest levels of the tree. For instance, if there are two ANDed conditions in the query and only one can be evaluated at the server-side, the selection operation is split in such a way that one condition is evaluated server-side and the other client-side.

Hacıgümüş et al. [57] show a method for splitting the query Q_s to be executed on the encrypted data into two sub-queries, Q_{s1} and Q_{s2} , where Q_{s1} returns only tuples that belongs to the final result, and query Q_{s2} may contain also spurious tuples. This distinction allows the execution of Q_c over the result of Q_{s2} only, while tuples returned by Q_{s1} can be immediately decrypted. To further reduce the client's workload, Damiani et al. [36] propose an architecture that minimizes storage at the client and introduce the idea of selective decryption of Q_s . With selective decryption, the client decrypts the portion of the tuples needed for evaluating Q_c , while complete decryption is executed only for tuples that belong to the final result and that will be returned to the final user. The approach is based on a block-cipher encryption algorithm, operating at tuple level, that allows the detection of the blocks containing the attributes necessary to evaluate the conditions in Q_c , which are the only ones that need decryption.

It is important to note that the process of transforming Q in Q_s and Q_c greatly depends both on the indexing method adopted and on the clauses and conditions

composing query Q . There are operations that need to be executed by the client, since the indexing method adopted does not support the specific operations (e.g., range queries are not supported by all types of indexes) and the server is not allowed to decrypt data. Also, there are operations that the server could execute over the index, but that require a pre-computation that only the client can perform and therefore must be postponed in Q_c (e.g., the evaluation of a condition in the HAVING clause, which needs a grouping over an attribute, whose corresponding index has been created by using a method that does not support the GROUP BY clause).

2.3 Querying Encrypted Data

When designing a solution for querying encrypted data, one of the most important goals is to minimize the computation at the client-side and to reduce communication overhead. The server therefore should be responsible for the majority of the work. Different indexing approaches allow the execution of different types of queries at the server side.

We now describe in more detail the methods initially proposed to efficiently execute simple queries at the server side, and we give an overview of more recent methods that improve the server's ability to query encrypted data.

2.3.1 Bucket-Based Approach

Hacigümüs et al. [58] propose the first method to query encrypted data, which is based on the definition of a number of *buckets* on the attribute domain. Let r_i be a plaintext relation over schema $R_i(a_{i1}, a_{i2}, \dots, a_{in})$ and r_i^k be the corresponding encrypted relation over schema $R_i^k(\underline{Counter}, Etuple, I_{i1}, \dots, I_{in})$. Considering an arbitrary plaintext attribute a_{ij} in R_i , with domain D_{ij} , bucket-based indexing methods partition D_{ij} in a number of non-overlapping subsets of values, called *buckets*, containing contiguous values. This process, called *bucketization*, usually generates buckets that are all of the same size.

Each bucket is then associated with a unique value and the set of these values is the domain for index I_{ij} associated with a_{ij} . Given a plaintext tuple t in r_i , the value of attribute a_{ij} for t (i.e., $t[a_{ij}]$) belongs to only one bucket defined on D_{ij} . The corresponding index value is then the unique value associated with the bucket to which the plaintext value $t[a_{ij}]$ belongs. It is important to note that, for better preserving data secrecy, the domain of index I_{ij} may not follow the same order as the one of the plaintext attribute a_{ij} . Attributes I_3 and I_5 in Fig. 2.2(b) are the indexes obtained by applying the bucketization method defined in Fig. 2.3 for attributes Y_{OB} and $Salary$ in Fig. 2.2(a). Note that I_3 values do not reflect the order of the domain values it represents, since $1975 < 1985$, while δ follows γ in lexicographic order.

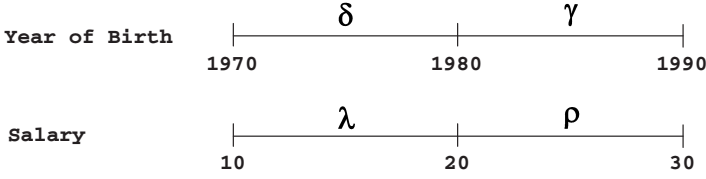


Fig. 2.3 An example of bucketization

Bucket-based indexing methods allow the server-side evaluation of equality conditions appearing in the WHERE clause, since these conditions can be mapped into equivalent conditions operating on indexes. Given a plaintext condition of the form $a_{ij}=v$, where v is a constant value, the corresponding condition operating on index I_{ij} is $I_{ij}=\beta$, where β is the value associated with the bucket containing v . As an example, with reference to Fig. 2.3, condition $Y_{OB}=1985$ is transformed into $I_3=\gamma$. Also, equality conditions involving attributes defined on the same domain can be evaluated by the server, provided that attributes characterized by the same domain are indexed using the same bucketization. In this case, a plaintext condition of the form $a_{ij}=a_{ik}$ is translated into condition $I_{ij}=I_{ik}$ operating on indexes.

Bucket-based methods do not easily support range queries. Since the index domain does not necessarily preserve the plaintext domain ordering, a range condition of the form $a_{ij}\geq v$, where v is a constant value, must be mapped into a series of equality conditions operating on index I_{ij} of the form $I_{ij}=\beta_1$ OR $I_{ij}=\beta_2$ OR ... OR $I_{ij}=\beta_k$, where β_1, \dots, β_k are the values associated with buckets that correspond to plaintext values greater than or equal to v . For instance, with reference to Fig. 2.3, condition $Y_{OB}\geq 1977$ must be translated into $I_3=\gamma$ OR $I_3=\delta$, since both values represent years greater than 1977.

Note that, since the same index value is associated with more than one plaintext value, queries exploiting bucket-based indexes usually produce spurious tuples that need to be filtered out by the client front-end. Spurious tuples are tuples that satisfy the condition over the indexes, but that do not satisfy the original plaintext condition. For instance, with reference to the relations in Fig. 2.2, query “SELECT * FROM Employee WHERE $Y_{OB}=1985$ ” is translated into “SELECT Etuple FROM Employee^k WHERE $I_3=\gamma$ ”. The result of the query executed by the server contains tuples 1, 3, 4, and 5; however, only tuple 3 satisfies the original condition as written by the user. Tuples 1, 4, and 5 are spurious and must be discarded by the client during the postprocessing of the Q_s result.

Hore et al. [61] propose an improvement to bucket-based indexing methods by introducing an efficient way for partitioning the domain of attributes. Given an attribute and a query profile on it, the authors present a method for building an efficient index, which tries to minimize the number of spurious tuples in the result of both range and equality queries.

As we will see in Sect. 2.4, one of the main disadvantages of bucket-based indexing methods is that they expose data to inference attacks.

2.3.2 Hash-Based Approach

Hash-based index methods are similar to bucket-based methods and are based on the concept of *one-way hash function* [35].

Let r_i be a plaintext relation over schema $R_i(a_{i1}, a_{i2}, \dots, a_{in})$ and r_i^k be the corresponding encrypted relation over schema $R_i^k(\text{Counter}, \text{Etuple}, I_{i1}, \dots, I_{in})$. For each attribute a_{ij} in R_i to be indexed, a one-way hash function $h : D_{ij} \rightarrow B_{ij}$ is defined, where D_{ij} is the domain of a_{ij} and B_{ij} is the domain of index I_{ij} associated with a_{ij} . Given a plaintext tuple t in r_i , the index value corresponding to attribute a_{ij} for t is computed by applying function h to the plaintext value $t[a_{ij}]$.

An important property of any hash function h is its *determinism*; formally, $\forall x, y \in D_{ij} : x = y \Rightarrow h(x) = h(y)$. Another interesting property of hash functions is that the codomain of h is smaller than its domain, so there is the possibility of *collisions*; a collision happens when given two values $x, y \in D_{ij}$ with $x \neq y$, we have that $h(x) = h(y)$. A further property is that h must produce a strong mixing, that is, given two distinct but near values x, y ($|x - y| < \varepsilon$) chosen randomly in D_{ij} , the discrete probability distribution of the difference $h(x) - h(y)$ is uniform (the results of the hash function can be arbitrarily different, even for very similar input values). A consequence of strong mixing is that the hash function does not preserve the domain order of the attribute on which it is applied. As an example, consider the relations in Fig. 2.2. Here, the indexes corresponding to attributes Emp-Id, Name, and Dept in relation `Employee` are computed by applying a hash-based method. The values of attribute Name have been mapped onto two distinct values, namely α and β ; the values of attribute Emp-Id have been mapped onto two distinct values, namely π and ϕ ; and the values of attribute Dept have been mapped onto three distinct values, namely ε , θ , and μ . Like for bucket-based methods, hash-based methods allow an efficient evaluation of equality conditions of the form $a_{ij}=v$, where v is a constant value. Each condition $a_{ij}=v$ is transformed into a condition $I_{ij}=h(v)$, where I_{ij} is the index corresponding to a_{ij} in the encrypted relation. For instance, condition `Name="Alice"` is transformed into $I_2=\alpha$. Also, equality conditions involving attributes defined on the same domain can be evaluated by the server, provided that these attributes are indexed using the same hash function. The main drawback of hash-based methods is that they do not support range queries, for which a solution similar to the one adopted for bucket-based methods is not viable: colliding values are in general not contiguous in the plaintext domain.

If the hash function used for index definition is not collision free, then queries exploiting the index produce spurious tuples that need to be filtered out by the client front-end. A collision-free hash function guarantees absence of spurious tuples, but may expose data to inference (see Sect. 2.4). For instance, assuming that the hash function adopted for attribute Dept in Fig. 2.2(a) is collision-free, condition `Dept="Financial"` is translated into $I_4=\mu$, that will return only the tuples (in our example, tuples with Counter equal to 3 and 5) that belong to the result of the query that contains the corresponding plaintext condition.

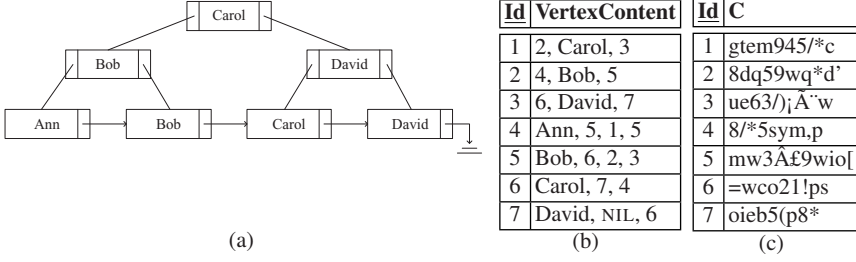


Fig. 2.4 An example of B+ tree indexing structure

2.3.3 B+ Tree Approach

Both bucket-based and hash-based indexing methods do not easily support range queries, since both these solutions are not order preserving. Damiani et al. [35] propose an indexing method that, while granting data privacy, preserves the order relationship characterizing the domain of attribute a_{ij} . This indexing method exploits the traditional B+ tree data structure used by relational DBMSs for physically indexing data. A B+ tree with fan out n is a tree where every vertex can store up to $n - 1$ search key values and n pointers and, except for the root and leaf vertices, has at least $\lceil n/2 \rceil$ children. Given an internal vertex storing f key values k_1, \dots, k_f with $f \leq n - 1$, each key value k_i is followed by a pointer p_i and k_1 is preceded by a pointer p_0 . Pointer p_0 points to the subtree that contains keys with values lower than k_1 , p_f points to the subtree that contains keys with values greater than or equal to k_f , and each p_i points to the subtree that contains keys with values included in the interval $[k_i, k_{i+1})$. Internal vertices do not directly refer to tuples in the database, but just point to other vertices in the structure; on the contrary, leaf vertices do not contain pointers, but directly refer to the tuples in the database having a specific value for the indexed attribute. Leaf vertices are linked in a chain that allows the efficient execution of range queries. As an example, Fig. 2.4(a) represents the B+ tree index built for attribute Name of relation Employee in Fig. 2.2(a). To access a tuple with key value k , value k is first searched in the root vertex of the B+ tree. The tree is then traversed by using the following scheme: if $k < k_1$, pointer p_0 is chosen; if $k \geq k_f$, pointer p_f is chosen, otherwise if $k_i \leq k < k_{i+1}$, pointer p_i is chosen. The process continues until a leaf vertex has been examined. If k is not found in any leaf vertex, the relation does not contain any tuple having, for the indexed attribute, value k .

A B+ tree index can be usefully adopted for each attribute a_{ij} in the schema of relation R_i , provided a_{ij} is defined over a partially ordered domain. The index is built by the data owner over the plaintext values of the attribute, and then stored on the remote server, together with the encrypted database. To this purpose, the B+ tree structure is translated into a specific relation with the two attributes: *Id*, represents the vertex identifier; and *VertexContent*, represents the actual vertex content. The relation has a row for each vertex in the tree and pointers are represented through

cross references from the vertex content to other vertex identifiers in the relation. For instance, the B+ tree structure depicted in Fig. 2.4(a) is represented in the encrypted database by the relation in Fig. 2.4(b). Since the relation representing the B+ tree contains sensitive information (i.e., the plaintext values of the attribute on which the B+ tree is built) this relation has to be protected by encrypting its content. To this purpose, encryption is applied at the level of vertex (i.e., of tuple in the relation), to protect the order relationship among plaintext and index values and the mapping between the two domains. The corresponding encrypted relation has therefore two attributes: *Id* that represents, as before, the identifier of the vertex; and *C* that contains the encrypted vertex. Figure 2.4(c) illustrates the encrypted B+ tree relation that corresponds to the plaintext B+ tree relation in Fig. 2.4(b).

The B+ tree based indexing method allows the evaluation of both equality and range conditions appearing in the WHERE clause. Moreover, being order preserving, it also allows the evaluation of ORDER BY and GROUP BY clauses of SQL queries, and of most of the aggregate operators, directly on the encrypted database. Given the plaintext condition $a_{ij} \geq v$, where v is a constant value, it is necessary to traverse the B+ tree stored on the server to find out the leaf vertex representing v for correctly evaluating the considered condition. To this purpose, the client queries the B+ tree relation to retrieve the root, which conventionally is the tuple t with $t[Id]=1$. It then decrypts $t[C]$, evaluates its content and, according to the search process above-mentioned, queries again the remote server to retrieve the next vertex along the path to v . The search process continues until a leaf vertex containing v is found (if any). The client then follows the chain of leaf vertices starting from the retrieved leaf to extract all the tuples satisfying condition $a_{ij} \geq v$. For instance, consider the B+ tree in Fig. 2.4(a) defined for attribute Name in relation Employee in Fig. 2.2(a). A query asking for tuples where the value of attribute Name follows “Bob” in the lexicographic order is evaluated as follows. First, the root is retrieved and evaluated: since “Bob” precedes “Carol”, the first pointer is chosen and vertex 2 is evaluated. Since “Bob” is equal to the value in the vertex, the second pointer is chosen and vertex 5 is evaluated. Vertex 5 is a leaf, and all tuples in vertices 5, 6, and 7 are returned to the final user.

It is important to note that B+ tree indexes do not produce spurious tuples when executing a query, but the evaluation of conditions is much more expensive for the client with respect to bucket and hash-based methods. For this reason, it may be advisable to combine the B+ tree method with either hash-based or bucket-based indexing, and use the B+ tree index only for evaluating conditions based on intervals. Compared with traditional B+ tree structures used in DBMSs, the vertices in the indexing structure presented here do not have to be of the same size as a disk block; a cost model can then be used to optimize the number of children of a vertex, potentially producing vertices with a large number of children and trees with limited depth. Finally, we note that since the B+ tree content is encrypted, the method is secure against inference attacks (see Sect. 2.4).

2.3.4 Order Preserving Encryption Approaches

To support equality and range queries over encrypted data without adopting B+ tree data structures, Agrawal et al. [4] present an *Order Preserving Encryption Schema* (OPES). An OPES function has the advantage of flattening the frequency spectrum of index values, thanks to the introduction of new buckets when needed. It is important to note here that queries executed over this kind of indexes do not return spurious tuples. Also, OPES provides data secrecy only if the intruder does not know the plaintext database or the domain of original attributes.

Order Preserving Encryption with Splitting and Scaling (OPESS) [96] is an evolution of OPES that both supports range queries and does not suffer from inference problems. This indexing method exploits the traditional B-tree data structure used by relational DBMSs for physically indexing data. B-tree data structure is similar to B+ tree data structure, but internal vertices directly refer to tuples in the database and leaves of the tree are not linked in a unique list.

An OPESS index can be usefully adopted for each attribute a_{ij} in the relation schema R_i , provided a_{ij} is defined over a partially ordered domain. The index is built by the data owner over the plaintext values of the attribute, and then stored on the remote server, together with the encrypted database. Differently from B+ tree indexing structure, the B-tree data structure exploited by OPESS is built on index values, and not on plaintext values. Therefore, before building the B-tree structure to be remotely stored on the server, OPESS applies two techniques on the original values of a_{ij} , called *splitting* and *scaling*, aimed at obtaining a flat frequency distribution of index values.

Consider attribute a_{ij} defined on domain D_{ij} and assume that the values $\{v_1, \dots, v_n\}$ in the considered relation r_i have occurrences, in the order, equal to $\{f_1, \dots, f_n\}$. First, a splitting process is performed on a_{ij} , producing a number of index values having almost a flat frequency distribution. The splitting process applies to each value v_h assumed by a_{ij} in r_i . It determines three consecutive positive integers, $m-1$, m , and $m+1$, such that the frequency f_h of value v_h can be expressed as a linear combination of the computed values: $f_h = c_1(m-1) + c_2(m) + c_3(m+1)$, where c_1 , c_2 , and c_3 are non negative integer values. The plaintext value v_h can therefore be mapped into c_1 index values each with $m+1$ occurrences, c_2 index values each with m occurrences, and c_3 index values each with $m-1$ occurrences. To preserve the order of index values with respect to the original domain of attribute a_{ij} , for any two values $v_h < v_l$ and for any index values i_h and i_l associated with v_h and v_l respectively, we need to guarantee that $i_h < i_l$. To this purpose, the authors in [96] propose to exploit an order preserving encryption function. Specifically, for each plaintext value v_h , its index values are obtained by adding a randomly chosen string of low order bits to a common string of high order bits computed as follows: $v_h^e = E_k(v_h)$, where E is an order preserving encryption function with key k .

Since splitting technique grants the sum of frequencies of indexes representing value v to be exactly the same as the original frequency of v , an attacker who knows the frequency distribution of plaintext domain values could exploit this property to break the indexing method adopted. Indeed, the index values mapping a given

plaintext value are, by definition, contiguous values. Therefore, the authors in [96] propose to adopt a *scaling* technique together with splitting. Each plaintext value v_h is associated with a scaling factor s_h . When v_h is split into n index values, namely i_1, \dots, i_n , each index entry in the B-tree corresponding to i_h is replicated s_h times. Note that all s_h replicas of the index point to the same block of tuples in the encrypted database. After scaling has been applied, the index frequency distribution is not uniform any more. Without knowing the scaling factor used, it is not possible for the attacker to reconstruct the correspondence between plaintext and index values.

The OPESS indexing method allows the evaluation of both equality and range conditions appearing in the WHERE clause. Moreover, being order preserving, it also allows the evaluation of ORDER BY and GROUP BY clauses of SQL queries, and of most of the aggregate operators, directly on the encrypted database. It is important to note that query execution becomes expensive, even if it does not produce spurious tuples, due to the fact that the same plaintext value is mapped into different index values and both splitting and scaling methods need to be inverted for query evaluation.

2.3.5 Other Approaches

In addition to the three main indexing methods previously presented, many other solutions have been proposed to support queries on encrypted data. These methods try to better support SQL clauses or to reduce the amount of spurious tuples in the result produced by the remote server.

Wang et al. [97, 98] propose a new indexing method, specific for attributes whose domain is the set of all possible strings over a well defined set of characters, which adapts the hash-based indexing methods to permit direct evaluation of LIKE conditions. The index value associated with any string s , composed of n characters $c_1 c_2 \dots c_n$, is obtained by applying a secure hash function to each pair of subsequent characters in s . Given a string $s = c_1 c_2 \dots c_n = s_1 s_2 \dots s_{n/2}$, where $s_i = c_{2i} c_{2i+1}$, the corresponding index is computed as $i = h(s_1) h(s_2) \dots h(s_{n/2})$.

Hacıgümüş et al. [57] study a method to remotely support aggregation operators, such as COUNT, SUM, AVG, MIN, and MAX. The method is based on the concept of *privacy homomorphism* [19], which exploits properties of modular algebra to allow the execution over index values of sum, subtraction, and product operations, while not preserving the order relationship characterizing the original domain. Evdokimov et al. [47] formally analyze the security of the method based on privacy homomorphism, with respect to the degree of confidentiality assigned to the remote server. The authors formally introduce a definition of *intrinsic security* for encrypted databases, and it is proved that almost all indexing methods are not intrinsically secure. In particular, methods that do not cause spurious tuples to belong to the result of a query inevitably are exposed to attacks coming from a malicious third party or from the service provider itself.

Index	Query		
	Equality	Range	Aggregation
Bucket-based [58]	●	○	—
Hash-based [35]	●	—	○
B+ Tree [35]	●	●	●
OPEs [4]	●	●	○
OPESS [96]	●	●	●
Character oriented [97, 98]	●	○	—
Privacy homomorphism [57]	●	—	●
PPC [63]	●	●	●
Secure index data structures [16, 20, 51, 93, 99]	●	○	—

● fully supported; ○ partially supported; — not supported

Fig. 2.5 Indexing methods supporting queries

The *Partition Plaintext and Ciphertext* (PPC) is a new model for storing server-side outsourced data [63]. This model proposes to outsource both plaintext and encrypted information that need to be stored on the remote server. In this model, only sensitive attributes are encrypted and indexed, while the other attributes are released in plaintext form. The authors propose an efficient architecture for the DBMS to store together, and specifically in the same page of memory, both plaintext and encrypted data.

Different working groups [16, 20, 51, 93, 99] introduce other approaches for searching keywords in encrypted documents. These methods are based on the definition of a *secure index data structure*. The secure index data structure allows the server to retrieve all documents containing a particular keyword without the need to know any other information. This is possible because a trapdoor is introduced when encrypting data, and such a trapdoor is then exploited by the client when querying data. Other similar proposals are based on *Identity Based Encryption* techniques for the definition of secure indexing methods. Boneh and Franklin [17] present an encryption method allowing searches over ciphertext data, while not revealing anything about the original data. This method is shown to be secure through rigorous proofs. Although these methods for searching keywords over encrypted data have been originally proposed for searching over audit logs or email repositories, they are also well suited for indexing data in the outsourced database scenario.

Figure 2.5 summarizes the discussion by showing, for each indexing method discussed, what type of query it (partially) supports. Here, an hyphen means that the query is not supported, a black circle means that the query is fully supported, and a white circle means that the query is partially supported.

2.4 Evaluation of Inference Exposure

Given a plaintext relation r over schema $R(a_1, a_2, \dots, a_n)$, it is necessary to decide which attributes need to be indexed, and how the corresponding indexes can be de-

fined. In particular, when defining the indexing method for an attribute, it is important to consider two conflicting requirements: on one side, the indexing information should be related to the data well enough to provide for an effective query execution mechanism; on the other side, the relationship between indexes and data should not open the door to *inference and linking attacks* that can compromise the protection granted by encryption. Different indexing methods can provide different trade-offs between query execution efficiency and data protection from inference. It is therefore necessary to define a measure for the risk of exposure due to the publication of indexes on the remote server.

Although many techniques supporting different kinds of queries in the DAS scenario have been developed, a deep analysis of the level of protection provided by all these methods against inference and linking attacks is missing. In particular, exposure has been evaluated for a few indexing methods only [24, 35, 37, 61].

Hore et al. [61] analyze the security issues related to the use of bucket-based indexing methods. The authors consider data exposure problems in two situations: *i*) the release of a single attribute, and *ii*) the publication of all the indexes associated with a relation. To measure the protection degree granted to the original data by the specific indexing method, the authors propose to exploit two different measures. The first measure is the *variance* of the distribution of values within a bucket b . The second measure is the *entropy* of the distribution of values within a bucket b . The higher is the variance, the higher is the protection level granted to the data. Therefore, the data owner should maximize, for each bucket in the relation, the corresponding variance. Analogously, the higher is the entropy of a bucket, the higher is the protection level granted to the data. The optimization problem that the data owner has to solve, while planning the bucketization process on a relation, is the *maximization of minimum variance and minimum entropy*, while maximizing query efficiency. Since such an optimization problem is *NP-hard*, Hore et al. [61] propose an approximation method, which fixes a maximum allowed performance degradation. The objective of the algorithm is then to maximize both minimum variance and entropy, while guaranteeing performances not to fall under an imposed threshold.

To the aim of taking into consideration also the risk of exposure due to associations, Hore et al. [61] propose to adopt, as a measure of the privacy granted by indexes when posing a multi-attribute range query, the well known *k-anonymity* concept [83]. Indeed, the result of a range query operating on multiple attributes is exposed to data linkage with publicly available datasets. *k*-Anonymity is widely recognized as a measure of the privacy level granted by a collection of released data, where respondents can be re-identified (or the uncertainty about their identity lower under a predefined threshold k) by linking private data with public data collections.

Damiani et al. [24, 35, 37] evaluate the exposure to inference due to the adoption of hash-based indexing methods. Inference exposure is measured by taking into account the prior knowledge of the attacker, thus introducing two different scenarios. In the first scenario, called *Freq+DB^k*, the attacker is supposed to know, in addition to the encrypted database (DB^k), the domains of the plaintext attributes and the distribution of plaintext values (*Freq*) in the original database. In the second scenario, called *DB+DB^k*, the attacker is supposed to know both the encrypted (DB^k) and the

plaintext database (*DB*). In both scenarios, the exposure measure is computed as the probability for the attacker to correctly map index values onto plaintext attribute values. The authors show that, to guarantee a higher degree of protection against inference, it is convenient to use a hash-based method that generates collisions. In case of a hash-based method where the collision factor is equal to 1, meaning that there is no collision, inference exposure measure depends only on the number of attributes used for indexing. In the $DB+DB^k$ scenario, the exposure grows as the number of attributes used for indexing grows. In the $Freq+DB^k$ scenario, the attacker can discover the correspondences between plaintext and indexing values by comparing their occurrence profiles. Intuitively, the exposure grows as the number of attributes with a different occurrence profile grows. For instance, considering relation *Employee* in Fig. 2.2(a), we can notice that both *Salary* and the corresponding index I_5 have a unique value with one occurrence only, that is, 20 and ρ , respectively. We can therefore conclude that the index value corresponding to 20 is ρ , and that no other salary value is mapped into ρ as well.

Damiani et al. [37] extend the inference exposure measures presented in [24, 35] to produce an inference measure that can be associated with the whole relation instead of with single attributes. The authors propose two methods for aggregating the exposure risk measures computed at attribute level. The first method exploits the weighted mean operator and weights each attribute a_i proportionally with the risk connected with the disclosure of the values of a_i . The second one exploits the OWA (Ordered Weighted Averaging) operator, which allows the assignment of different importance values to different sets of attributes, depending on the degree of protection guaranteed by the indexing method adopted for the specific subset of attributes.

Agrawal et al. [4] evaluate the exposure to inference due to the adoption of OPESS as an indexing method, under the $Freq+DB^k$ scenario. They prove that the solution they propose is intrinsically secure, due to the flat frequency distribution of index values and to the additional guarantee given by scaling method, which avoids the combination of the attackers frequency knowledge with the knowledge of the indexing method adopted.

2.5 Integrity of Outsourced Data

The database outsourcing scenario usually assumes the server to be “honest-but-curious”, and that clients and data owners trust it to faithfully maintain outsourced data. However, this assumption is not always applicable and it is also important to protect the database content from improper modifications (*data integrity*). The approaches proposed in the literature have the main goal of detecting unauthorized updates of remotely stored data [56, 73, 74, 92]. Hacigümüs et al. [56] propose to add a signature to each tuple in the database. The signature is computed by digitally signing, with the private key of the owner, a hash value obtained through the application of a hash function to the tuple content. The signature is then added to the tuple before encryption. When a client receives a tuple, as a result of its query, it

can verify if the tuple has been modified by an entity different from the data owner. The verification process consists in recomputing the hash over the tuple content and checking whether there is a match with the value stored in the tuple itself. In addition to tuple level integrity, also relation level integrity (i.e., absence of non authorized insertions and deletions of tuples) needs to be preserved. Therefore, for each relation, a signature computed on the basis of the tuples in the relation is added. An advantage of the proposed method is that relation level signature does not need to be recomputed any time a tuple is inserted or deleted because the old signature can be adapted to the new content, thus saving computation time at the data owner side.

Since an integrity check performed on each tuple in the result set of a query can be quite expensive, Mykletun et al. [73] propose methods for checking the signature of a set of tuples in a single operation. The first method, called *condensed RSA*, works only if the tuples in the set have been signed by the same user; the second method, which is based on bilinear mappings and is less efficient than condensed RSA, is called *BGLS* (from the name of the authors who first proposed this signature method [18]) and works even if the tuples in the set have been signed by different users. A major drawback of these solutions is that they do not guarantee the *immutability property*. Immutability means that it is difficult to obtain a valid aggregated signature from a set of other aggregated signatures. To solve this problem, Mykletun et al. [72] propose alternative solutions based on zero knowledge protocols.

Narasimha and Tsudik [74] present another method, called *Digital Signature Aggregation and Chaining (DSAC)*, that is again based on hash functions and signature. Here, the main goal is to evaluate whether the result of a query is *complete* and *correct* with respect to the database content. This solution builds over each relation chains of tuples, one for each attribute that may appear in a query, that are ordered according to the attribute value. The signed hash associated with a tuple is then computed by composing the hash value associated with the immediate predecessors of the considered tuple in all the chains. This solution is quite expensive when there are different chains associated with a relation.

Sion [92] proposes a method to ensure result accuracy and guarantee that the server correctly executes the query on the remote data. The method works for batch queries and is based on the pre-computation of *tokens*. Basically, before outsourcing the database, the data owner pre-computes a set of queries on plaintext data and associates, with each query, a token computed by using a one-way cryptographic hash function on the query results, concatenated with a nonce. Any set of batch queries submitted to the server contains then a subset of pre-computed queries, along with the corresponding tokens, and fake tokens. The server, when answering, has to indicate which are the queries in the batch set that correspond to the given tokens. If the server correctly individuates which tokens are fake, the client is guaranteed that the server has executed all the queries in the set.

2.6 Privacy Protection of Databases

Often encryption of the whole database containing sensitive data is an overdo, since not all the data are sensitive per se but only their association needs protection. To reduce the usage of encryption in data outsourcing, thus improving query execution efficiency, it is convenient to combine fragmentation and encryption techniques [2]. In [2] the authors propose an approach where privacy requirements are modeled simply through confidentiality constraints (i.e., sets of attributes whose joint visibility must be prevented) and are enforced by splitting information over two independent database servers (so to break associations of sensitive information) and by encrypting information only when strictly necessary. By assuming that only trusted clients know the two service providers (each of which is not aware of the existence of the other server), sensitive associations among data can be broken by fragmenting the original data. When fragmentation is not sufficient for solving all confidentiality constraints characterizing the data collection, data encryption can be exploited. In this case, the key used for encrypting the data is stored on one server and the encrypted result on the other one. Alternatively, other data obfuscation methods can be exploited; the parameter value is stored on one server and the obfuscated data on the other one. Since the original data collection is divided on two non-communicating servers, the evaluation of queries formulated by trusted users requires the presence of a trusted client for possibly combining the results coming from the two servers. The original query is split in two subqueries operating at each server, which results are then joined and refined by the client. The process of query evaluation becomes therefore expensive, especially if fragmentation does not take into account the query workload characterizing the system (i.e., when attributes frequently appearing in the same query are not stored on the same server). After proving that identifying a fragmentation that minimizes query execution costs at the client side is NP-hard (this problem can be reduced to the hypergraph coloring problem), the authors propose a heuristic algorithm producing good results.

While presenting an interesting idea, the approach in [2] suffers from several limitations. The main limitation is that privacy relies on the complete absence of communication between the two servers, which have to be completely unaware of each other. This assumption is clearly too strong and difficult to enforce in real environments. A collusion among the servers (or the users accessing them) easily breaches privacy. Also, the assumption of two servers limits the number of associations that can be solved by fragmenting data, often forcing the use of encryption. The solution presented in Chap. 4 overcomes the above limitations: it allows storing data even on a single server and minimizes the amount of data represented in encrypted format, therefore allowing for efficient query execution.

A related line of work is represented by [13, 14], where the authors exploit functional dependencies to the aim of correctly enforcing access control policies. In [14] the authors propose a policy based classification of databases that, combined with restriction of the query language, preserves the confidentiality of sensitive information. The classification of a database is based on the concept of classification instance, which is a set of tuples representing the combinations of values that need

to be protected. On the basis of the classification instance, it is always possible to identify the set of allowed queries, that is, the queries whose evaluation return tuples that do not correspond to the combinations represented in the classification instance. In [13] the authors define a mechanism for defining constraints that reduce the problem of protecting the data from inferences to the enforcement of access control in relational databases.

2.7 Access Control Enforcement in the Outsourcing Scenario

Traditional works on data outsourcing assume all users to have complete access to the whole database by simply knowing the (unique) encryption key adopted for data protection. However, this simplifying assumption does not fit current scenarios where different users may need to see different portions of the data, that is, where *selective access* needs to be enforced, also because the server cannot be delegated such a task. Adding a traditional authorization layer to the current outsourcing scenarios requires that when a client poses a query, both the query and its result have to be filtered by the data owner (who is in charge of enforcing the access control policy), a solution that however is not applicable in a real life scenario. More recent researches [15, 33, 70, 102] have addressed the problem of enforcing selective access on outsourced encrypted data by combining cryptography with authorizations, thus enforcing access control via *selective encryption*. Basically, the idea is to use different keys for encrypting different portions of the database. These keys are then distributed to users according to their access rights.

The naive solution for enforcing access control through selective encryption consists in using a different key for each resource in the system, and in communicating to each user the set of keys associated with the resources she can access. This solution correctly enforces the policy, but it is very expensive since each user needs to keep a number of keys that depends on her privileges. That is, users having many privileges and, probably, often accessing the system, will have a greater number of keys than users having a few privileges and, probably, accessing only rarely the system. To reduce the number of keys a user has to manage, access control mechanisms based on selective encryption exploit *key derivation methods*. A key derivation method is basically a function that, given a key and a piece of publicly available information, allows the computation of another key. The basic idea is that each user is given a small number of keys from which she can derive all the keys needed to access the resources she is authorized to access.

To the aim of using a key derivation method, it is necessary to define which keys can be derived from another key and how. Key derivation methods proposed in the literature are based on the definition of a *key derivation hierarchy*. Given a set of keys \mathcal{K} in the system and a partial order relation \preceq defined on it, the corresponding key derivation hierarchy is usually represented as a pair (\mathcal{K}, \preceq) , where $\forall k_i, k_j \in \mathcal{K}$, $k_j \preceq k_i$ iff k_j is derivable from k_i . Any key derivation hierarchy can be graphically represented through a directed acyclic graph, having a vertex for each key in \mathcal{K} ,

and a path from k_i to k_j only if k_j can be derived from k_i . Depending on the partial order relationship defined on \mathcal{K} , the key derivation hierarchy can be: a *chain* (i.e., \preceq defines a total order relation); a *tree*; or a *directed acyclic graph* (DAG). The different key derivation methods can be classified on the basis of the kind of hierarchy they are able to support, as follows.

- The hierarchy is a *chain of vertices* [85]. Key k_j of a vertex is computed on the basis of key k_i of its (unique) direct ancestor (i.e., $k_j = f(k_i)$) and no public information is needed.
- The hierarchy is a *tree* [54, 85, 86]. Key k_j of a vertex is computed on the basis of key k_i of its (unique) parent and on the publicly available label l_j associated with k_j (i.e., $k_j = f(k_i, l_j)$).
- The hierarchy is a *DAG* [6, 8, 31, 59, 62, 67, 69, 87, 91]. Since each vertex in a DAG can have more than one direct ancestor, key derivation methods are in general more complex than the methods used for chains or trees. There are many proposals that work on DAGs; typically they exploit a piece of public information associated with each vertex of the key derivation hierarchy. In [8], Atallah et al. introduce a new class of methods that maintain a piece of public information, called *token*, associated with each edge in the hierarchy. Given two keys, k_i and k_j arbitrarily assigned to two vertices, and a public label l_j associated with k_j , a token from k_i to k_j is defined as $t_{i,j} = k_j \oplus h(k_i, l_j)$, where \oplus is the n -ary xor operator and h is a secure hash function. Given $t_{i,j}$, any user knowing k_i and with access to public label l_j , can compute (derive) k_j . All tokens $t_{i,j}$ in the system are stored in a *public catalog*.

It is important to note that key derivation methods operating on trees can be used for chains of vertices, even if the contrary is not true. Analogously, key derivation methods operating on DAGs can be used for trees and chains, while the converse is not true.

Key derivation hierarchies have also been adopted for access control enforcement in contexts different from data outsourcing. For instance, pay-tv systems usually adopt selective encryption for selective access enforcement and key hierarchies to easily distribute encryption keys [12, 79, 94, 95, 100]. Although these applications have some similarities with the DAS scenario, there are important differences that do not make them applicable for data outsourcing. First, in the DAS scenario we need to protect stored data, while in the pay-tv scenario streams of data are the resources that need to be protected. Second, in the DAS scenario key derivation hierarchies are used to reduce the number of keys each user has to keep secret, while in the pay-tv scenario a key derivation hierarchy is exploited for session key distribution.

The main problem any solution adopting selective encryption suffers from is that they require data re-encryption for policy updates, thus causing the data owner's intervention any time the policy is modified. The selective encryption solution proposed in Chap. 3 is organized to both reduce the client burden in data access and the data owner intervention in policy updates.

2.8 Safe Data Integration

Data outsourcing scenarios typically assume data to be managed by a unique external server, managing sensitive information. As already noted for solutions combining fragmentation and encryption for privacy purposes, data may also be stored at different servers. Furthermore, emerging scenarios often require different parties to cooperate with other parties to the aim of sharing information and perform distributed computations. Cooperation for query execution implies data to flow among parties. Therefore, it is necessary to provide the system with solutions able to enforce access control restrictions in data exchange for distributed query evaluation. Indeed, classical works on the management of queries in centralized and distributed systems [11, 23, 26, 64, 68, 90, 101] cannot be exploited in such a scenario. These approaches in fact describe how efficient query plans can be obtained, but do not take into consideration constraints on attribute visibility for servers. However, in light of the crucial role that security has in the construction of future large-scale distributed applications, a significant amount of research has recently focused on the problem of processing distributed queries under protection requirements. Most of these works [21, 46, 48, 52, 66, 75] are based on the concept of *access pattern*, a profile associated with each relation/view where each attribute has a value that may either be *i* or *o* (i.e., input or output). When accessing a relation, the values for all *i* attributes must be supplied, to obtain the corresponding values of *o* attributes. Also, queries are represented in terms of Datalog, a query language based on the logic programming paradigm. The main goal of all these works is that of identifying the classes of queries that a given set of access patterns can support; a secondary goal is the definition of query plans that match the profiles of the involved relations, while minimizing some cost parameter (e.g., the number of accesses to data sources [21]). In Chap. 5, we propose a complementary approach to access patterns that can be considered a natural extension of the approach normally used to describe database privileges in a relational schema; our approach introduces a mechanism to define access privileges on join paths; while access patterns describe authorizations as special formulas in a logic programming language for data access. Also, the model presented in Chap. 5 explicitly manages a scenario with different independent subjects who may cooperate in the execution of a query, whereas the work done on access patterns only considers two actors, the owner of the data and a single user accessing the data.

In [80], the authors propose a model based on the definition of authorization views that implicitly define the set of queries that a user can view. A query is allowed if it can be answered using only the information in the authorization views regulating the system. An interesting advantage of this model is the exploitation of referential integrity constraints for the automatic identification of security compliance of queries with respect to views. It is interesting to note that the approach in [80] operates at a low level since it analyzes the integration with a relational DBMS optimizer and focuses on the consideration of “instantiated” queries (i.e., queries that present predicates that force attributes to assume specific values) aiming at evaluate compatibility of the instantiated queries with the authorized views.

The approach proposed in Chap. 5 operates at a higher level, proposing an overall data-model characterizing views and focusing on the data integration scenario at a more abstract level.

Sovereign joins [3] represent an interesting alternative solution for secure information sharing. This method is based on a secure coprocessor, which is involved in query execution, and exploits cryptography to grant privacy. The advantage of sovereign joins is that they extend the plans that allow an execution in the scenario we present; the main obstacle is represented by their high computational cost, due to the use of specific asymmetric cryptography primitives, that make them currently not applicable when large collections of sensitive information must be combined.

2.9 Chapter Summary

Database outsourcing is becoming an emerging data management paradigm that introduces many research challenges. In this chapter, we focused on the solutions known in the literature for solving problems related to query execution and access control enforcement. For query execution, different indexing methods have been discussed. These methods mainly focus on supporting specific kind of queries and on minimizing the client burden in query execution. Fragmentation has also been proposed as a method for reducing encryption and improving query execution performance. Access control enforcement is instead a relative new issue for the DAS scenario and has not been deeply studied. The most important proposal for enforcing access control on outsourced encrypted data is based on selective encryption and key derivation strategies. Finally, the evaluation of queries when outsourced data are distributed at different servers requires a deeper collaboration among servers as well as mechanisms regulating the exchange of data among the collaborating parties. This problem has been addressed in some proposals that are based on the access pattern concept.

In the following of this book, we will analyze more in depth the access control, proposing a new mechanism based on selective encryption, and we will study a solution to the well known problem of dynamically manage access control updates. We will also focus on the usage of fragmentation for reducing encryption, trying to overcome the limitations of the proposal in [2]. Furthermore, we will address the problem related to the execution of queries on distributed data, modeling authorized data flows among involved parties in a simple while powerful manner.



<http://www.springer.com/978-1-4419-7658-1>

Preserving Privacy in Data Outsourcing

Foresti, S.

2011, XV, 180 p., Hardcover

ISBN: 978-1-4419-7658-1