SN Computer Science (2023) 4:440
https://doi.org/10.1007/s42979-023-01862-6

ORIGINAL RESEARCH

Protecting Data and Queries in Cloud-Based Scenarios

Sabrina De Capitani di Vimercati[1] · Sara Foresti[1] · Pierangela Samarati[1]

Received: 26 January 2023 / Accepted: 26 April 2023

Abstract

The availability of cloud services offered by different providers brings several advantages to users and companies, facilitating the storage, sharing, and processing of data. At the same time, the adoption of cloud services brings new security and privacy risks and challenges. As a matter of fact when leveraging cloud-based services for data storage and processing, data owners loose direct control on their data. Data and queries over them could then be at risk for both potentially improper exposure, compromising their confidentiality, or tampering, compromising their integrity. In this paper, we discuss the main issues to be addressed for guaranteeing data security and privacy in cloud-based storage and processing. We illustrate the different challenges to be considered and the research directions toward their solutions.

Keywords Cloud-based scenario · Data protection · Selective data sharing · Access confidentiality · Querying encrypted data · Query integrity · Distributed query execution

Introduction

The adoption of cloud services has seen a significant increase in the last years as it has created new market opportunities for companies. There is no doubt that there are many benefits in using cloud services such as better business continuity, scalability, and economical savings. However, there are also new security and privacy risks that need to be carefully considered (e.g., [1–4]), when migrating applications and data from a local on-premises system to the cloud. In fact, data are stored and processed at external cloud providers on which the owners of the data have no control. Such lack of control raises several concerns: data should be safeguarded against unauthorized accesses (confidentiality) and modifications (integrity), should be accessible to authorized users when needed (availability), and should be adopted for performing secure computations to support, for example, decision making and business analytics.

The security and privacy issues to be addressed when data and computations are outsourced to the cloud vary depending on the considered scenario (e.g., a scenario where the cloud is used mainly for data storage or a scenario where the cloud is also used for fine-grained retrieval and query processing). In this paper, we consider a scenario characterized by a multiplicity of cloud providers offering storage and computational services, which can then be used by data owners and users for storing data and for performing (distributed) computations. The complexity of this scenario comes not only from the security and privacy guarantees that must be offered on data and accesses on them but also from the need of supporting computational services on protected data. For instance, if the data outsourced to the cloud are stored in encrypted form for confidentiality reasons, the execution of queries and, more in general, of computations over these data requires the adoption of novel approaches that should be directly applicable to the encrypted data.

The goal of this paper is to present an overview of the main issues that must be considered for properly protecting outsourced data and computations. These issues and

Sabrina De Capitani di Vimercati, Sara Foresti and Pierangela Samarati have contributed equally to this work.

This article is part of the topical collection “Advances on Cloud Computing and Services Science” guest edited by Donald F. Ferguson, Claus Pahl and Maarten van Steen.

✉ Pierangela Samarati
pierangela.samarati@unimi.it

Sabrina De Capitani di Vimercati
sabrina.decapitani@unimi.it

Sara Foresti
sara.foresti@unimi.it

[1] Computer Science Department, Università degli Studi di Milano, Via Celoria 18, 20133 Milan, Italy

Published online: 10 June 2023

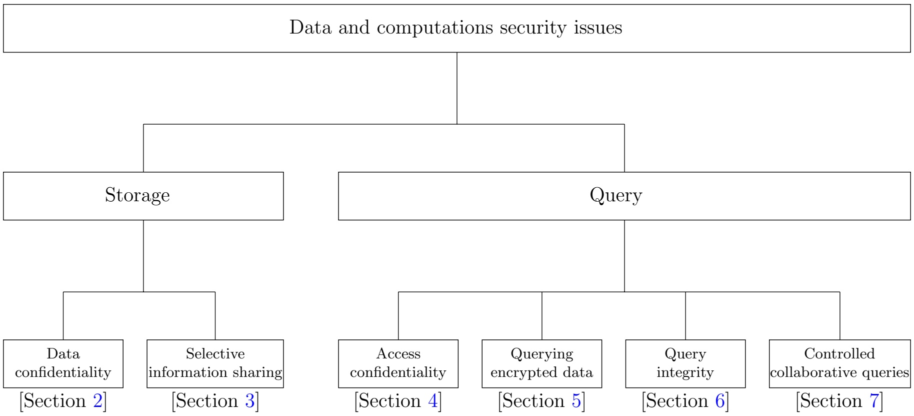


Fig. 1 Data and computations security issues in cloud-based storage and processing

challenges can be characterized according to different dimensions. In particular, our analysis is guided by the level of service requested from the cloud, which we identify as: (1) *storage*, and (2) *query*. These different services raise different security and privacy issues. The first level (storage) refers to scenarios where data owners use the cloud for storing their data, and access to data simply corresponds to upload and/or download operations. In this case, security refers to proper protection of data stored on the cloud platform, and to empower data owners with control over their data in the cloud, meaning that data owners should be able to administer their data and selectively share them with others. The second level (query) refers to scenarios where access to data requires fine-grained retrieval and execution of queries, also using the presence of multiple providers for conveniently supporting collaborative queries. In this case, security refers to protection of dynamically retrieved data and query results. For both the storage level and query level, the paper will address the problem of guaranteeing *confidentiality* and *integrity*. Data confidentiality is needed whenever the outsourced data are sensitive or confidential, and therefore should not be known to the cloud providers themselves. In many scenarios, cloud providers are also not trustworthy, and data owners, as well as cloud users, should then be able to verify the correctness of the responses (retrieved data or computations) received from the cloud. Clearly, the precise meaning of confidentiality and integrity depends on the level of service (i.e., refer to stored data and queries). Figure 1 summarizes the issues that will be discussed in the remainder of this paper. For each issue, we will describe existing approaches that have been adopted for addressing it.

Running example. For concreteness, in the following, we frame the discussion in the context of relational database systems. We then consider data as relational tables and computations as queries of the general form “SELECT FROM WHERE”. Our examples will be based on a scenario with two data owners $\mathbb{A}$ and $\mathbb{C}$, a subject $\mathbb{S}$ interested in performing an analysis involving the data managed by $\mathbb{A}$ and $\mathbb{C}$, and three external providers $\mathbb{X}$, $\mathbb{Y}$, and $\mathbb{Z}$ offering computational services only (see Fig. 2). Owner $\mathbb{A}$ manages the information about the access points of a cell-free network. Such information is stored in relation `Access Point` (`AP`) with attributes (`Apid`, `Numant`, `Platitude`, `Mlongitude`) reporting information about the identifier of an access point (`Apid`), the number of antennas of the access point (`Numant`), and the coordinates (latitude and longitude, respectively) of the access point (`Platitude`, and `Mlongitude`, respectively). Owner $\mathbb{C}$ manages the information about the activities of user devices. Such information is stored in relation `CallDetailRecord` (`CDR`) with attributes (`Cid`, `Idap`, `Hashid`, `Time`, `Duration`) reporting the information about the identifier of the relation (`Cid`), the identifier of an access point (`Idap`), the hash of the identification number of a user device (`Hashid`), the activity time (`Time`), and the duration of the activity (`Duration`). Figure 3 shows an example of the two relations `AP` and `CDR`. Note that, in the following, attributes will be denoted using their initials.

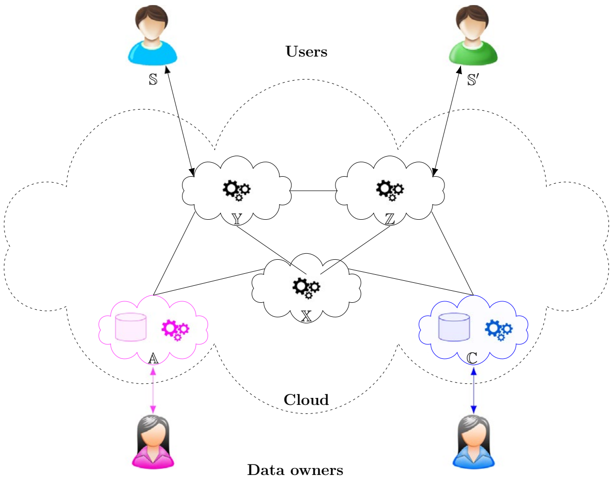


Fig. 2 Reference scenario for the running example

Outline. The remainder of the paper is organized as follows. Section “Data Protection” provides an overview of the approaches that ensure data confidentiality and integrity.

AccessPoint (AP)

	Apid	Numant	Platitude	Mlongitude
t_1	A1	4	38.846224	-77.306373
t_2	A2	2	13.923404	2.158925
t_3	A3	1	45.529661	9.269336
t_4	A4	4	46.127087	10.353258
t_5	A5	7	41.902782	12.496365
t_6	A6	6	43.923555	19.408342
t_7	A7	6	25.482951	75.508649
t_8	A8	7	13.923404	79.508649

CallDetailRecord (CDR)

	Cid	Idap	Hashid	Time	Duration
t_1	C1	A1	jk78	11:12:00	5
t_2	C2	A2	jk78	11:20:00	10
t_3	C3	A6	k07g	23:59:10	12
t_4	C4	A7	uog5	15:15:10	7

Fig. 3 Relations of the running example

Section “Selective Information Sharing” discusses the solutions addressing the problem of supporting selective sharing of data. Section “Access Confidentiality” describes solutions for ensuring access confidentiality. Section “Indexes for Queries over Encrypted Data” illustrates techniques that can be adopted for fine-grained access to encrypted data, focusing on index-based solutions. Section “Query Integrity” discusses how to assess the integrity of data returned as a result of queries executed by cloud providers. Section “Controlled Execution of Collaborative Queries” focuses on solutions supporting controlled collaborative computations with the involvement of multiple providers. Finally, Section “Conclusions” gives our conclusions.

Data Protection

When data are moved to the cloud, data owners have to first select the providers that offer the storage and/or computational services more suitable for their needs, in terms of quality of the service, performance, and security offered (e.g., [5–11]). Whenever the outsourced data are sensitive or confidential, there is the problem of ensuring proper protection (integrity and confidentiality) of such data. We now discuss some of the solutions proposed for protecting the integrity and confidentiality of data.

Integrity and Confidentiality

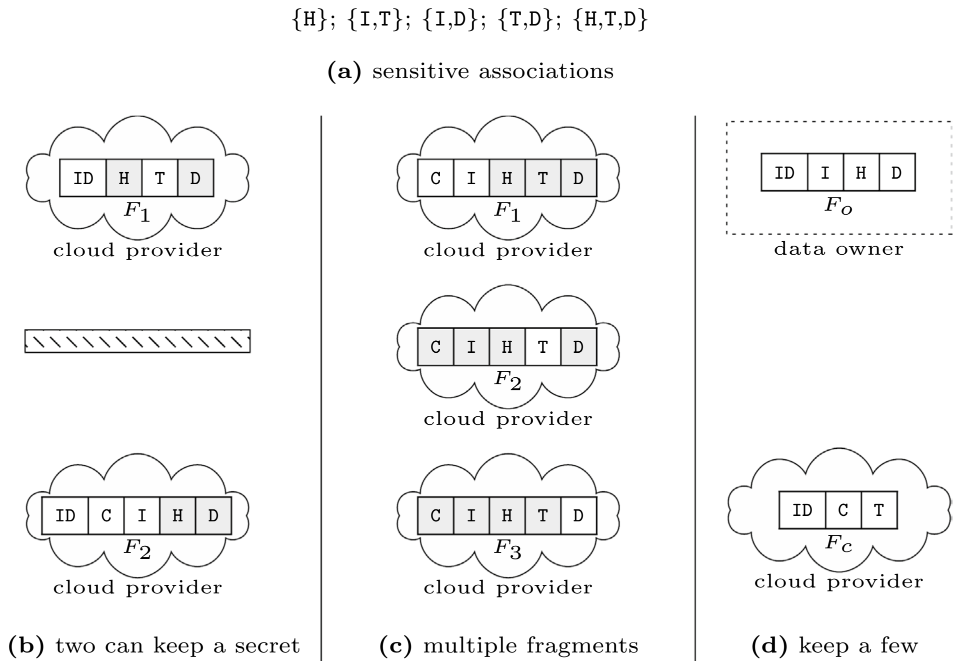
Ensuring data integrity means that the data owner and users should be able to verify whether data have been improperly modified or tampered with. Existing solutions (e.g., [12]) are based on the use of hashing and digital signatures as building blocks. With these solutions, the verification of the integrity of the data requires data owners to access their data in the cloud. Other solutions (e.g., proof of retrievability (POR) and/or provable data possession (PDP) schemes [13, 14]) are based on the idea of inserting *sentinels* in the encrypted outsourced data (POR) or pre-compute tokens over encrypted or plaintext data (PDP) to provide the owner with a probabilistic guarantee that the data have not been modified by non-authorized users.

With respect to confidentiality, existing solutions (e.g., [15, 16]) are typically based on the assumption that the outsourced data are managed by a *honest-but-curious* provider. Honest-but-curious means that the provider is trusted to manage the data but it is not trusted with respect to their confidentiality. In this case, data owners apply an encryption layer to their data before storing them to the external providers. Although encryption is a powerful mechanism that protects data confidentiality, there are also approaches that limit or depart from encryption whenever possible. These approaches are based on the observation that dealing with encrypted data is a burden since encryption makes it not always possible to efficiently execute queries and evaluate conditions over the data. Furthermore, encrypting the whole data collection may not be needed when it is the association among different pieces of information that is considered confidential and not a single piece of information. As an example, consider relation CDR in Fig. 3(b), and suppose that the association (H,T,D) is considered sensitive because it permits to infer that a specific user device (H) was connected in a specific timeframe (T and D). While the association is sensitive, the list of hash user identifiers, the list of times, and the list of durations singularly taken are not sensitive. Therefore, there is no need to encrypt these attributes if there are alternative solutions that protect their association. A possible solution for protecting sensitive associations is *data fragmentation*. Data fragmentation consists in vertically splitting the set of attributes of a relation in different fragments in such a way that the fragments are not linkable (e.g., [17, 18]). Intuitively, fragmentation protects sensitive associations among different attributes when the attributes involved in a sensitive association are not visible in the same (publicly available) fragment, and fragments cannot be joined by non-authorized users. For instance, considering the sensitive association (H,T,D) above-mentioned, relation CDR could be split in two fragments, denoted F_1 and F_2, with F_1={C,I,H} and F_2={T,D}. In this way, the three attributes H, T, and D are not visible together in a single fragment. Note that when a single attribute is sensitive, fragmentation cannot provide protection. In this case, the attribute is protected only when it does not appear in plaintext in any fragment stored at an external provider. For instance, suppose that attribute H is sensitive. Fragment F_1 of the previous example should then store the values of this attribute in encrypted form. Fragmentation strategies differ in how (and whether) fragmentation is coupled with encryption [15]. In particular, three different fragmentation paradigms have been proposed. In the following discussion, examples will refer to the sensitive associations of relation CDR illustrated in Fig. 4a.

The first paradigm, called *two can keep a secret*, is characterized by the presence of two independent, non-communicating, cloud providers, each of which stores a fragment of the relation. Sensitive attributes are always encoded (e.g., encrypted) across both cloud providers so that they cannot deduce the attribute values (e.g., a cloud provider can store the encrypted attribute values and the other cloud provider can store the keys used for encrypting these values). All the other attributes must be encoded whenever storing them in plaintext at any of the two providers would make at least one sensitive association visible in the fragment. Note that the two fragments must have a common attribute (i.e., a unique tuple ID) to allow authorized users to correctly reconstruct the original relation by joining the two fragments. Figure 4b illustrates a fragmentation for relation CDR such that neither F_1 nor F_2 store all the attributes of the sensitive associations in Fig. 4a in plaintext. Note that attribute D is stored in encrypted form (which is represented with a gray background) since its plaintext storage in F_1 would expose sensitive association {T,D}, and its plaintext storage in F_2 would expose sensitive association {I,D}.

The second paradigm, called *multiple fragments*, supports the splitting of the original relation in any number of fragments. Encryption is adopted to protect sensitive attributes, and fragmentation to protect sensitive associations. Fragments contain all attributes of the original relation in either encrypted form or plaintext form (i.e., every fragment is complete) and are not linkable since they do not have common attributes. Since fragments are not linkable, there is no need to assume that the providers storing them do not communicate and they can also be stored on the same provider. Figure 4c illustrates a fragmentation for relation CDR in three fragments F_1, F_2, and F_3, where encrypted attributes have a gray background. Note that in this case, attribute D does not need to be encrypted but can be stored in plaintext in a fragment different from fragments F_1 and F_2 that already contain the plaintext attributes I and T, respectively. The query involving D can then be efficiently executed on its plaintext representation in F_3.

The third paradigm, called *keep a few*, see the involvement of a trusted party (e.g., the owner) for storing a small amount of data. This paradigm does not apply encryption since sensitive attributes are always stored at the trusted-party side. Sensitive associations are protected by storing at least one of their attributes at the trusted-party side. Like for the two can keep a secret paradigm, the two fragments must have a common key attribute to permit authorized users to correctly reconstruct the content of the original relation. Figure 4d illustrates a fragmentation for relation CDR in two



(a) sensitive associations

(b) two can keep a secret **(c)** multiple fragments **(d)** keep a few

Fig. 4 An example of sensitive associations **a** for relation CDR(C,I,H,T,D) in Fig. 3b and of its fragmentation with respect to the *two can keep a secret* (**b**), *multiple fragments* (**c**), and *keep a few* (**d**) paradigm

fragments F_o and F_c, where F_o is the fragment stored at the trusted party and F_c is the fragment stored at an external cloud provider. Note that at least one attribute of each sensitive association in Fig. 4a is in F_o.

Fragmentation solutions can be further extended to consider, in the allocation of fragments to providers, the different characteristics of the providers themselves such as economic cost and performance.

Fragmentation in Decentralized Cloud Storage

The idea of splitting data in fragments (i.e., "data chunks") for confidentiality reasons is also applied in decentralized cloud storage (DCS) systems. In DCS systems, a resource (e.g., a file) is split in *shards* each of which is allocated (with replication to provide availability guarantees) to different nodes. The resource can be accessed by retrieving and recomposing its shards. Since the nodes in the DCS are usually of different providers that are outside the control of data owners, owner-side encryption is typically used to protect the confidentiality of the resources. While effective, owner-side encryption leaves resources exposed to threats. Resources are still vulnerable when, for example, the encryption key is exposed or when malicious nodes not deleting their shards upon owner's requests try to reconstruct a resource in its entirety. The approach in [19] addresses these issues by making the (even partial) reconstruction of data impossible if even a single shard is missing. This is obtained through the application of an All-Or-Nothing-Transform (AONT) encryption mode [20] that transforms a plaintext resource into a ciphertext so that the whole encrypted resource is required to obtain back the original plaintext resource. The encrypted resource is then properly partitioned into several slices that are distributed at the different nodes in the system in such a way that at least $k + 1$ nodes should collaborate to collect the slices composing a resource. This means that a resource is protected against collusion of up to k malicious nodes. Data availability is instead provided through the storage of a number of replicas of each slice of a resource.

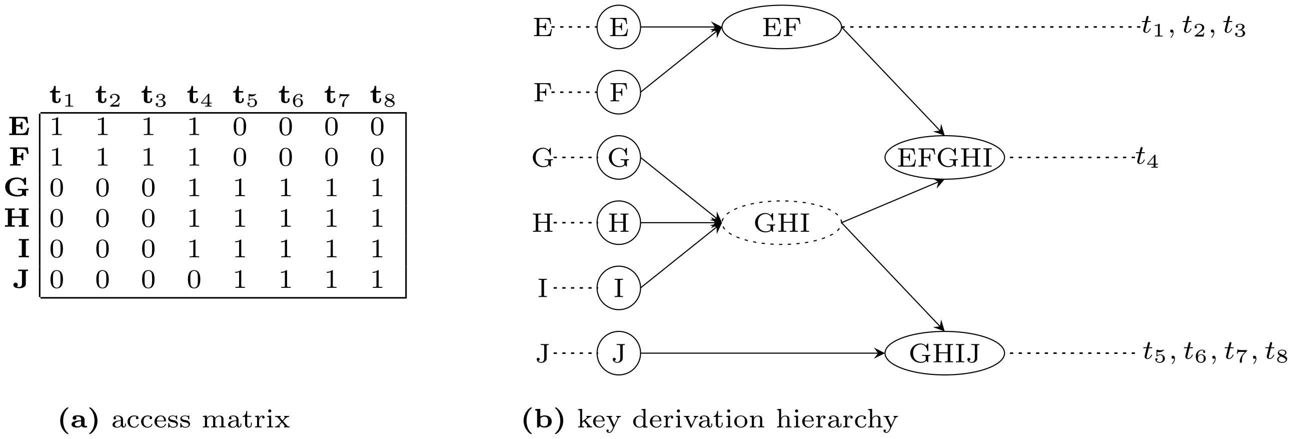
An interesting problem to be address in the DCS context is related to the need of balancing data availability and security guarantees while limiting the data owner's intervention in case of failure of a node (and hence unavailability of a subset of shards) [19, 21].

Selective Information Sharing

The consideration of a large community of users introduces the problem of how to grant access to the data outsourced to the cloud in a selective way. Selective sharing means that the access to data by other users should obey possible authorizations that the data owner wishes to apply. The enforcement of such authorizations cannot be performed by the data owners themselves since it is impractical to assume that the owners intercept each and every access request. Similarly, the enforcement of the authorizations cannot be delegated to the cloud provider because it is not trusted to access the data content, and because the data owner should remain in control. Existing solutions to this problem (e.g., [22]) combine selective encryption and key derivation strategies (e.g., [23]), or rely on attribute-based encryption (e.g., [24]).

Selective Encryption

Selective encryption means that different pieces of information are encrypted with different encryption keys. These keys are distributed to users in such a way that they can decrypt all and only the data authorized to access. The authorization policy is then translated into an equivalent encryption policy,



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
E	1	1	1	1	0	0	0	0
F	1	1	1	1	0	0	0	0
G	0	0	0	1	1	1	1	1
H	0	0	0	1	1	1	1	1
I	0	0	0	1	1	1	1	1
J	0	0	0	0	1	1	1	1

(a) access matrix

(b) key derivation hierarchy

Fig. 5 An example of access matrix defined by $\mathbb{A}$ for relation $\mathtt{AP}$ (**a**), and of corresponding key derivation hierarchy (**b**)

regulating key distribution to users and the keys used to encrypt the resources. As an example, consider relation AP in Fig. 3a, and suppose that before its outsourcing the relation is encrypted at tuple level, meaning that for each tuple, the attributes of the relation are encrypted all together. Suppose that there are six users, Ellen (E), Frank (F), Gary (G), Hope (H), Ily (I), and Joe (J), that are authorized to access the information stored in this relation. Figure 5a illustrates an example of access matrix defined by the data owner $\mathbb{A}$ of relation AP regulating access to the tuples in the relation. The access matrix represents the authorization state, that is, the authorizations defined by the data owner at a given time. According to this access matrix, Ellen and Frank can access the first four tuples t_1, t_2, t_3, and t_4, Gary, Hope, and Ily can access the last five tuples $t_4, t_5, \ldots, t_8$, and Joe can access the last four tuples $t_5, \ldots, t_8$. A simple way for enforcing such access restrictions consists in first encrypting all tuples characterized by the same access profile (i.e., the same access control list—ACL) using the same encryption key, and then communicating the encryption keys to the users authorized to access the corresponding tuples. For instance, with respect to the access matrix in Fig. 5a, we can see that there are three different access control lists for the tuples of relation AP: ACL(t_1) = ACL(t_2) = ACL(t_3) = {Ellen, Frank}, ACL(t_4) = {Ellen, Frank, Gary, Hope, Ily}, and ACL(t_5) = ACL(t_6) = ACL(t_7) = ACL(t_8) = {Gary, Hope, Ily, Joe}. We can then use three encryption keys: one key, say k_1, for encrypting tuples t_1, t_2, t_3, a second key, say k_2, for encrypting t_4, and a third key, say k_3, for encrypting tuples $t_5, \ldots t_8$. Key k_1 is then communicated to Ellen, Frank, and Gary only, k_2 to Ellen, Frank, Gary, Hope, and Ily, and k_3 to Gary, Hope, Ily, and Joe only. In this way, the users can only decrypt the tuples that they are authorized to access. Note that each user has to manage as many keys as the number of access control lists to which the user belongs.

Although effective, selective encryption requires data re-encryption every time a user is granted or revoked access to a piece of information. This implies a download/upload overhead of possibly huge resources at the data owner side for data re-encryption. For instance, suppose that the owner $\mathbb{A}$ of relation AP grants access for a new user Kal to tuples t_5, t_6, t_7, and t_8. These tuples are download, decrypted, re-encrypted with a new key, say k_4, and then re-uploaded. The encryption key k_4 is then communicated to Kal and to all other users who can still access the tuples (i.e., Gary, Hope, Ily, and Joe). To avoid users having to store and manage a huge number of keys, the approach in [22] is based on *a key derivation method* and on *two layers of encryption*.

A key derivation method allows the derivation of a key starting from another key and some public information, and requires the definition of a *key derivation hierarchy* that establishes which key can be derived from which other key. A key derivation hierarchy can be seen as a directed graph with a vertex for each key in the system, and an edge from key k_i to key k_j iff k_j can be derived from k_i. The key derivation method used in [22] associates each key k with a *public label l*, and each edge in the key derivation hierarchy with a *public token* [23]. Given two keys k_i and k_j, with public label l_i and l_j, respectively, the public token $t_{i,j}$ that permits to derive k_j from k_i and l_j is computed as $t_{i,j} = k_j \oplus h(k_i, l_j)$, where $\oplus$ is the bitwise XOR operator, and h is a hash function. The set containment relationship over the set of users is then used to define the key derivation hierarchy that initially contains one vertex for each user and each access control list. Then for each vertex corresponding to an access control list, the direct ancestors are the vertices that form a non-redundant set covering for it. For instance, with respect to the access matrix in Fig. 5a, the direct ancestors of the vertex representing access control list {E,F} are vertex {E} and vertex {F}. For the vertices that have more than two common ancestors, a factorization process can be applied. Such a process corresponds to the insertion of an intermediate vertex representing the common ancestors. This allows a saving in the number of edges (and therefore tokens) that needs to be defined. Figure 5b illustrates the key derivation hierarchy corresponding to the access matrix in Fig. 5a, where the dotted lines from a vertex to tuples indicate that the tuples are encrypted using the key represented by the vertex (e.g., tuple t_4 is encrypted with the key represented by vertex {EFGHI}), and the dotted lines from users to vertices represent the keys assigned to users. This hierarchy has a vertex ({GHI}, denoted with a dotted line) that does not correspond to any access control list and that represents the set of common ancestors for vertices {EFGHI} and {GHIJ}. The presence of this vertex saves one token because instead of connecting vertices {G}, {H}, and {I} to both {EFGHI} and {GHIJ}, they are connected to {GHI}, and then vertex {GHI} is connected to {EFGHI} and {GHIJ}. The keys associated with vertices {E}, {F}, … ,{J} are those communicated to the corresponding user and are used for (directly or indirectly) deriving the keys associated with the vertices representing the access control list (which are used to encrypt the tuples of relation AP). For instance, suppose that user Ellen wishes to access tuple t_4. User Ellen uses key k_{E} (i.e., the key represented by vertex {E} in the key derivation hierarchy) to derive key $k_{\mathrm{EF}} = t_{\mathrm{E,EF}} \oplus h(k_{\mathrm{E}}, l_{\mathrm{EF}})$ that in turn is used to derive key $k_{\mathrm{EFGHI}} = t_{\mathrm{EF,EFGHI}} \oplus h(k_{\mathrm{EF}}, l_{\mathrm{EFGHI}})$, which is the key adopted for encrypting tuple t_4.

With respect to the use of the two layers of encryption for policy update enforcement, the proposal in [22] uses one (static) layer applied by the data owner before storing the data in the cloud, and a second (dynamic) layer applied by the storage provider. The idea is that the second layer applied by the storage provider enforces policy updates, according to the data owner's requests, encrypting the data in such a way that only the authorized users can remove the two layers of encryption. This approach requires support by the cloud provider in managing policy updates. For instance, suppose that the authorization of Ellen to access tuple t_1 of relation AP is revoked. Instead of downloading this tuple from the cloud provider, decrypting it, re-encrypting the tuple with a different key, and then re-uploading the encrypted tuple, the data owner sends to the cloud provider a request of protecting the tuple with a second layer of encryption, using a key that only Frank (who can still access t_1) knows or can derive. In this way, Ellen cannot remove this second encryption layer enforced by the cloud provider, and therefore cannot access the tuple anymore. An alternative solution for efficiently supporting access revocation without relying on the cloud provider consists in using an encryption that performs a complete mixing of the resource. This mixing guarantees that the unavailability of a small portion of the encrypted resource prevents its (even partial) reconstruction [20]. The encrypted resource is then sliced into fragments and every time a user is revoked access to the resource, the owner can re-encrypt a randomly selected (small) fragment with a key that the revoked user neither knows nor can derive. Few works have extended the selective encryption technique to also enforce selective write privileges (e.g., [25]) and to support the presence of multiple data owners that selectively share their data (e.g., [26]).

Attribute-Based Encryption

Attribute-based encryption (ABE) is a public-key encryption that regulates access to data according to access policies that are defined over attributes associated with the data and/or with the users (e.g., [1]). These approaches can then be distinguished in two main classes: Ciphertext-Policy ABE (CP-ABE), and Key-Policy ABE (KP-ABE). In CP-ABE, the secret key of a user is associated with a list of attributes, and data are associated with an access policy defined over an attribute universe of the system. A user can access (decrypt) a data item if and only if the attribute values associated with the user satisfy the access policy associated with the encrypted data. For instance, suppose that the attribute universe includes attributes {Company, Role}, where Company can assume values 'Ghost', 'Microservice', or 'Wise' and Role can assume values 'Technician' or 'Administrative'. Suppose also that the tuples of relation AP can be accessed by technicians working for the Ghost company or from users working for the Microservice company. The tuples of relation AP are then encrypted under the access policy ((Company:Ghost AND Role:Technician) OR Company :Microservice). In this case, user Alice with attribute list {Company:Ghost, Role:Administrative} cannot decrypt the tuples of relation AP.

In KP-ABE, the access policy is encoded into the user's secret key and data are encrypted according to an attribute list. A user can access (decrypt) a data item if and only if the attribute values associated with the data item satisfy the access policy associated with the user's attribute secret key. For instance, if the access policy ((Company:Ghost AND Role:Technician) OR Company:Microservice) is encoded into the Bob's attribute secret key, then Bob cannot decrypt the tuples of relation CDR, which are encrypted based on the attribute list {Company:Wise, Role:Technician}. Between the CP-ABE and KP-ABE, the first one has received much more attention than the second one mainly because the definition of the access policy is on the hand of the data owners. We conclude by noting that also the ABE-based approaches have been extended to support write privileges through, for example, the application of attribute-based signature techniques (e.g., [27]).

Access Confidentiality

Although encryption and/or fragmentation protect the confidentiality of data stored at external cloud providers (Section"Data Protection"), these solutions are not enough when data are involved in the execution of queries. The observation of the accesses to an outsourced data collection may reveal sensitive information about the user performing the query as well as about the data collection itself. As an example, suppose that Alice is looking for the information about a specific disease, say arthritis. By knowing that Alice is looking for information about arthritis, it is possible to infer that Alice (or a person close to her) is suffering from it. Analogously, disclosing the fact that two accesses aim at the same encrypted piece of information allows an observer to maintain the information about the frequency of accesses to data and, exploiting external knowledge on the frequency of accesses to the corresponding plaintext data, reveals to the observer the plaintext data behind the accessed encrypted data. It is, therefore, necessary to protect, beside data confidentiality, also *access confidentiality* (i.e., confidentiality of the target of each access) and *patter confidentiality* (i.e., confidentiality of the fact that two accesses aim at the same piece of information). Different solutions have been proposed to protect access and pattern confidentiality (e.g., PIR, ORAM-based approaches, shuffle index) [16]. In particular, the shuffle index [28] organizes the outsourced relation as an *unchained* B+-tree (i.e., a B+-tree where contiguous leaf nodes are not connected) built over a candidate key of the relation. Figure 6a illustrates a logical representation of a shuffle index with fan-out 3 (i.e., the number of pointers to child nodes in a node is at most 3), where each node is associated with a unique logical identifier. These identifiers do not reflect the order relationship among the values in the nodes. Each node of the shuffle index is encrypted (with a random salt to destroy plaintext distinguishability), and

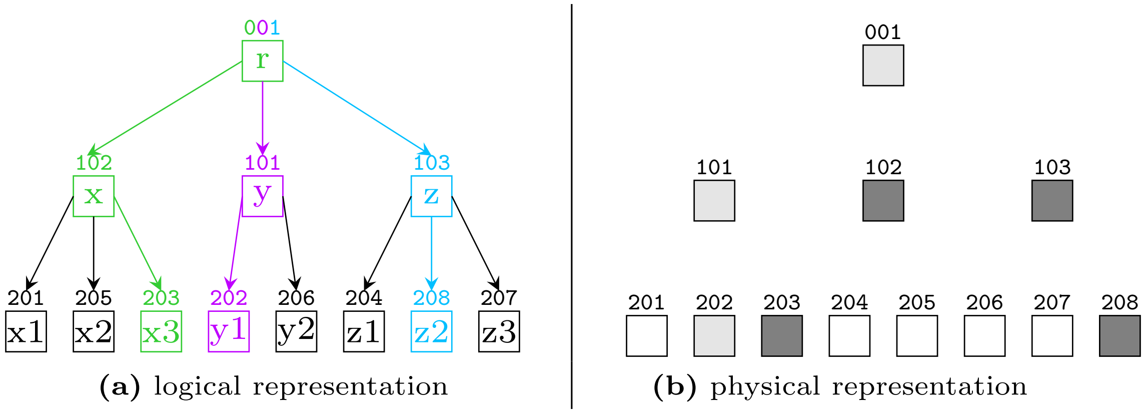


Fig. 6 An example of logical and physical representation of a shuffle index and of a search for value x3

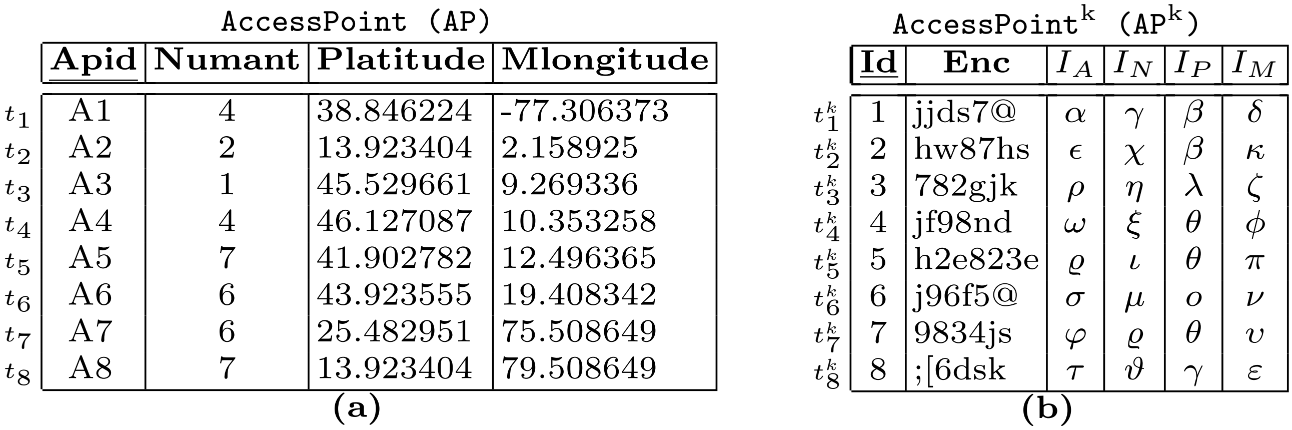
stored in a physical block. Figure 6b illustrates the physical representation of the shuffle index in Fig. 6a.

Access and pattern confidentiality are guaranteed through the combined application of three protection techniques: *cover searches*, *cached searches*, and *shuffling*. Cover searches are fake searches that are executed together (i.e., in parallel) with the search operation for a target value. To be effective, cover searches should be indistinguishable from real searches, and visit disjoint paths along the tree, also with respect to the target. More precisely, for each level of the shuffle index (but the root level), the user downloads from the cloud provider a fixed number of nodes (physical blocks), say $num\_cover + 1$: one is the block along the path to the target value, and *num_cover* are the blocks along the (disjoint) paths to the cover searches. For the cloud provider, each of the $num\_cover + 1$ searches, and hence accessed leaf blocks, can be the target. Cover searches then hide the target search within a group of other fake searches.

Cached searches consists in maintaining at the user side a local layered cache (i.e., a cache with one layer for each level of the shuffle index) that is used to store the nodes along the paths to the target values of the *num_cache* most recent accesses to the shuffle index. Whenever the target value is in cache, the corresponding block is not read from the cloud provider and an additional cover search is used during the access, to guarantee that the cloud provider always observes the visit of $num\_cover + 1$ disjoint paths. Intuitively, the cache avoids short-time intersection attacks, which could be exploited by the cloud provider to identify repeated subsequent accesses downloading non-disjoint sets of blocks.

Shuffling destroys the otherwise static node-block correspondence by storing each visited node in a different block. Shuffling then assigns a different block to each accessed node, choosing among the downloaded blocks. This guarantees that repeated accesses to the same block do not imply repeated accesses to the same node. To prevent the cloud provider from inferring where a node has been moved, every time a node is stored into a different block, it is re-encrypted using a different random salt. Note that the parent of a shuffled node is updated to preserve the consistency of the shuffle index (i.e, the pointers to the child nodes must be updated according to the shuffling performed). The shuffle index then guarantees that accesses downloading the same block(s) might have a different target, and that accesses aimed at the same target may download disjoint sets of blocks. The cloud provider is then not able to reconstruct the frequency of accesses to data based on the observed accesses to physical blocks.

Figure 6b illustrates an example of shuffle index where we perform a search for value x3, the cache contains the path to y1, and value z2 is chosen as cover. The search operation starts with an access to the root node (001) in the local cache, and with the identification of the nodes along the path to the target (block 102), cover (block 103), and in cache (block 101). Blocks 102 and 103 are downloaded from the cloud provider and decrypted, and node x along the target path is inserted in the local cache. The client then shuffles nodes 101, 102, and 103 (e.g., x is assigned to 103, z to 101, and y to 102), updates the root node to adjust the pointers to the children according to the shuffling performed, re-encrypts its content, and stores it at the cloud provider. Analogously, the blocks at the second level of the shuffle index along the path to the target (block 203) and to the cover (208) are downloaded and decrypted, and the local cache is updated by inserting node x3 and removing y1. The client then shuffles blocks 203, 208, and 202 (e.g., x3 is assigned to 208, z2 to 202, and y1 to 203), updates the parents of leaf nodes (i.e., x, y, and z) with the new values for the pointers to the children, re-encrypts them, and re-writes them back at the cloud provider. Finally, the client re-encrypts the accessed leaf nodes and sends the corresponding blocks to the cloud provider for storage. The gray blocks in Fig. 6b are the blocks read and written during the search operation (i.e., the blocks corresponding to nodes in the target and cover paths), and the



AccessPoint (AP)

	Apid	Numant	Platitude	Mlongitude
t_1	A1	4	38.846224	-77.306373
t_2	A2	2	13.923404	2.158925
t_3	A3	1	45.529661	9.269336
t_4	A4	4	46.127087	10.353258
t_5	A5	7	41.902782	12.496365
t_6	A6	6	43.923555	19.408342
t_7	A7	6	25.482951	75.508649
t_8	A8	7	13.923404	79.508649

(a)

AccessPoint$^{\text{k}}$ (AP$^{\text{k}}$)

	Id	Enc	I_A	I_N	I_P	I_M
t_1^k	1	jjds7@	α	γ	β	δ
t_2^k	2	hw87hs	ϵ	χ	β	κ
t_3^k	3	782gjk	ρ	η	λ	ζ
t_4^k	4	jf98nd	ω	ξ	θ	ϕ
t_5^k	5	h2e823e	ϱ	ι	θ	π
t_6^k	6	j96f5@	σ	μ	o	ν
t_7^k	7	9834js	φ	ϱ	θ	υ
t_8^k	8	;[6dsk	τ	ϑ	γ	ε

(b)

Fig. 7 An example of plaintext (**a**) and corresponding encrypted and indexed relation (**b**)

light gray blocks are those only written (i.e., the blocks corresponding to nodes in the local cache).

Note that the presence of multiple providers can be beneficial for efficiently performing anonymized version (e.g., *k*-anonymity [29]) of very large datasets (e.g., [30, 31]), and for providing access and pattern confidentiality. For instance, the shuffle index has been extended to work in a distributed scenario characterized by three independent providers storing a portion of the shuffle index (and that can then observe only a portion of the accesses over it) [32]. Each access to the shuffle index is designed to involve all the providers and to dynamically move accessed data among the providers. The developed solution provides stronger protection compared to the use of a single provider and has been designed to resist to collusion even among all the three providers.

Indexes for Queries over Encrypted Data

In this section, we consider the problem of supporting query execution over data stored in the cloud in encrypted form while preserving data confidentiality. There are two lines of approaches that address this problem. The first line consists in applying, at the owner side, cryptographic algorithms that support the evaluation of conditions directly over the data encrypted with these algorithms (e.g., property-preserving encryption, searchable symmetric encryption, and homomorphic encryption [33–35]). The second line consists in complementing the encrypted data with *indexes* that are then used for query execution (e.g., [36]). Indexes are metadata associated with the encrypted relation stored at an external provider that allow the provider to retrieve the data of interest. In the following, we focus on index-based solutions, and we first discuss how a relation can be encrypted and indexed (Section "Encrypted and Indexed Relation"), and then present some indexing solutions (Sections "Single-Dimensional Indexes" and "Multi-dimensional Indexes").

Encrypted and Indexed Relation

Before storing a relation $R(a_1, \ldots, a_n)$ in the cloud, it is important to decide the granularity level at which the relation must be encrypted. A relation R can be encrypted at relation, attribute, tuple, or cell level. Clearly, the granularity level has an impact on the overhead implied by encryption/decryption operations and on the number of spurious tuples (i.e., tuples that do not belong to the query result but that the cloud provider cannot filter out) returned by the provider in response to a query. For instance, encryption at the relation level causes low overhead in terms of encryption/decryption operations but it requires to always download the whole relation independently from the query. Cell level encryption, instead, causes a high overhead for encryption/decryption operations, but allows the provider to be more precise in finding the data of interest. Tuple level encryption represents a reasonable trade-off between encryption/decryption overhead and precision in query execution.

The encrypted and indexed relation R^k corresponding to the tuple-level encryption of relation $R(a_1, \ldots, a_n)$ has schema $R^k(\texttt{id}, \texttt{Enc}, I_l, \ldots, I_m)$, with id a randomly chosen tuple identifier, Enc the encrypted tuple, and I_i the index computed over attribute a_i, $i = l, \ldots, m$. In the following, for simplicity, we assume that all attributes in R are associated with an index in R^k while noting that only attributes expected to be involved in queries need to be indexed. Figure 7b illustrates an example of encrypted relation for the plaintext relation in Fig. 3a, which is reported in Fig. 7a. In this example, indexes have been defined for all the attributes of the plaintext relation (e.g., I_A is the index over attribute Apid), and Greek letters denote index values.

To limit the number of spurious tuples in query results, indexes should reflect the properties of the plaintext values they represent but, at the same time, they should not reveal the underlying plaintext values. The analysis in [37] shows that indexes are sensitive to frequency-based attacks and that an index function, to be robust against frequency attacks, should flatten the distribution of index values and generate collisions. This implies that all index values should have the same number of occurrences and that a same index value could correspond to different plaintext values. By destroying the frequency distribution of index values, an observer cannot exploit the frequency distributions of plaintext and index values for reconstructing the index function.

We now describe some indexing solutions, distinguishing between indexes defined over each single attribute (Section "Single-Dimensional Indexes"), and indexes defined over multiple attributes (Section "Multi-dimensional Indexes") of the relation to be outsourced.

Single-Dimensional Indexes

Single-dimensional indexes can support the execution of equality and/or range conditions over a single attribute of the plaintext relation. There are different kinds of indexes that differ in how the mapping between plaintext values and index values is computed (and hence queries are translated). Such techniques can be classified as follows.

- *Direct index* (1:1) Each plaintext value is mapped to one index value and vice versa. This mapping can be computed, for example, by applying a deterministic encryption function over the plaintext values. This index preserves the frequency of the original plaintext values since all the occurrences of the same plaintext value are mapped to the same index value. As an example, index I_A in Fig. 7b is a direct index computed over the plaintext values of attribute Apid of the relation in Fig. 7a.
- *Bucket index* (n:1) Each plaintext value is mapped to one index value, with collisions. This means that different occurrences of the same plaintext value are all mapped to the same index value, and that different plaintext values can be mapped to the same index value. This mapping can be computed, for example, by applying a hash function with collisions [38] or by partitioning the plaintext attribute domain and mapping all values in a partition to a same index value. As an example, index I_P in Fig. 7b is a bucket index computed over the plaintext values of attribute Platitude of the relation in Fig. 7a. Here, plaintext values 38.846224 and 13.923404 are both mapped to the same index value β, and plaintext values 46.127087, 41.902782, and 25.482951 are mapped to θ.
- *Flattened index* (1:n) Each plaintext value is mapped to one or more index values so that all index values have the same number of occurrences (flattening). Each index value, however, represents one plaintext value only. This mapping can be computed by applying, for example, an encryption function over plaintext values that ensures such properties. As an example, index I_N in Fig. 7b is a flattened index computed over the plaintext values of attribute Numant of the relation in Fig. 7a. Here, the two occurrences of plaintext value 4 are mapped to different index values, that is, γ and ξ, and the two occurrences of plaintext value 6 are mapped to μ and ϱ. In this way, all index values have a flat frequency equal to 1.

Given a query of the form "SELECT *Attributes* FROM R WHERE $a_j = v$" submitted by a user, it can be easily translated into a query over the corresponding encrypted and indexed relation: "SELECT Enc FROM R^k WHERE I_{a_j} IN $map_{a_j}(v)$" where map_{a_j} is the index function defined over attribute a_j that maps a value v in the domain of the attribute to the corresponding index value(s). The transformed query is executed by the provider storing the encrypted and indexed relation and the result is returned to the user. The user decrypts the query result and possibly executes the original query over the decrypted result to filter spurious tuples. For instance, suppose that a user submits query "SELECT Apid FROM AP WHERE Platitude=38.846224". This query is transformed into the following query operating on the encrypted and indexed relation AP$^{\mathrm{k}}$: "SELECT Enc FROM AP$^{\mathrm{k}}$ WHERE I_P IN $\{\beta\}$". The provider storing relation AP$^{\mathrm{k}}$ executes the transformed query and returns the set $\{t_1^k,t_2^k\}$ of tuples of the encrypted and indexed relation in Fig. 7b. Upon receiving the encrypted tuples, the user decrypts them and executes query "SELECT Apid FROM Res WHERE Platitude=38.846224", with Res the relation containing the two decrypted tuples. This query returns the first tuple only since the second tuple is spurious.

Alternative indexing solutions (not based on the definition of an indexing function) designed for supporting range queries leverage traditional indexing approaches usually adopted in the database context. For instance, the proposal in [38] uses a B+tree for supporting range queries. The B+-tree is constructed over an attribute of the original plaintext relation and is stored at the cloud provider in encrypted form. More precisely, the B+-tree is stored as a relational table with two attributes: a node ID, automatically assigned by the system on insertion, and an encrypted value, representing the node content. The execution of a range query over the attribute used for constructing the B+-tree is performed by the client through the execution of a sequence of queries that retrieve tree nodes at progressively deeper levels. The execution starts by retrieving from the cloud provider the encrypted root of the B+-tree, which is decrypted by the

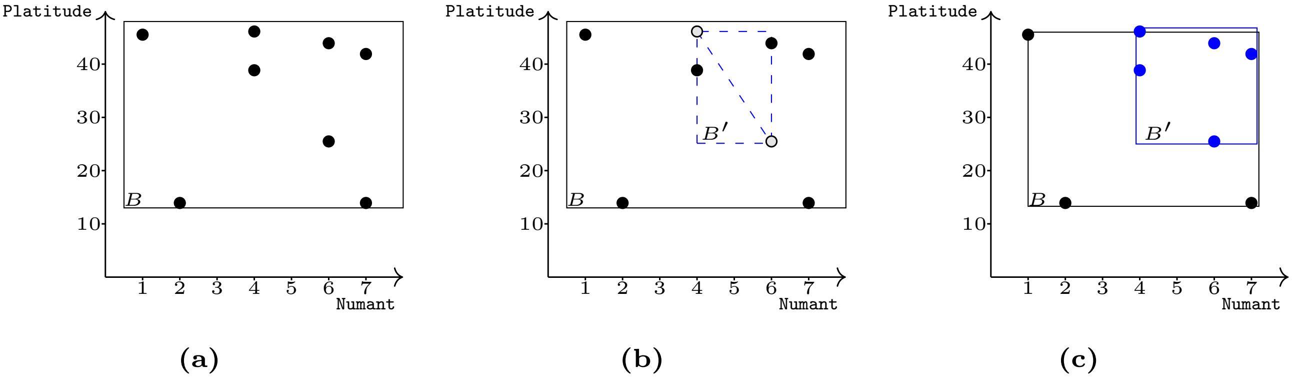


Fig. 8 An example of clustering for the proposal in [42]

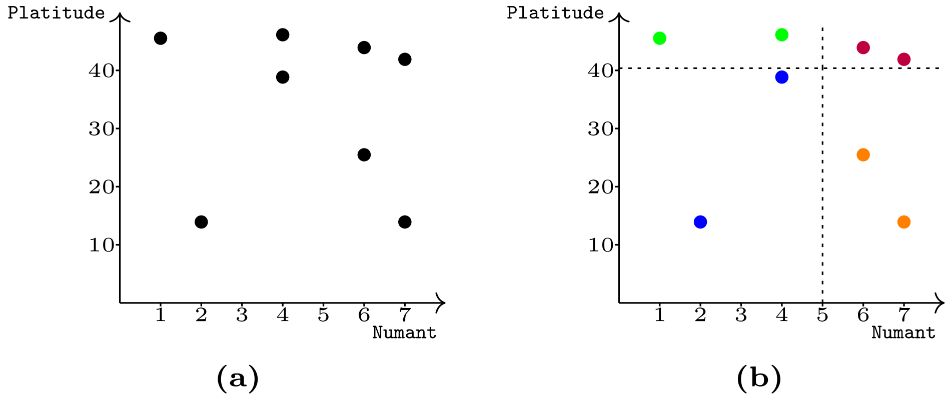
client to determine the node at the next level of the tree that has to be retrieved from the provider. The process terminates when a leaf is reached: the node ID of the leaf can be used to retrieve the tuples satisfying the range condition. For instance, suppose that the execution of a range query terminates over a leaf node with ID 10. Since all the leaf nodes of a B+-tree are linked, the search process continues from node 10 by following the sibling link to the next leaf node, say the node with ID 11, to see whether it contains tuples that satisfy the range conditon, and so on. The search process terminates when the examined leaf node contains a value that falls outside the range condition.

Multi-dimensional Indexes

Multi-dimensional indexes are build for supporting multi-dimensional queries, that is, queries with conditions involving different attributes. Some proposals use tree-based structures (e.g., encrypted R-tree [39] and virtual binary tree [40]) for indexing data with the goal of defining techniques to efficiently visit the tree-based structure, while protecting data confidentiality.

Other multi-dimensional indexing techniques are based on an idea similar to the bucket-based index described in the previous section (e.g., [41, 42]). The tuples in the original plaintext relation are represented as points in a multi-dimensional space, with one dimension for each attribute that can be involved in a query. Then the points in the space are partitioned into different regions (buckets), and each region is assigned with an index or a set of indexes. As an example, consider the plaintext relation in Fig. 7a, and suppose that the attributes possibly involved together in queries are attributes `Numant` and `Platitude`. Figure 8a illustrates the two-dimensional representation of the tuples over the two attributes `Numant` and `Platitude`. The indexing techniques in [41, 42] then differ in how the points in the space are partitioned and in how indexes are assigned to each partition.

The proposal in [42] partitions the tuples with the goal of minimizing a cost measure that depends on the number of spurious tuples retrieved by a query (for details on the cost measure see [42]) while keeping the disclosure risk (i.e., the ability of estimating the value of one or more attributes of the tuples in a bucket) low. More precisely, the partitioning operates in two phases. In the first phase, the tuples are partitioned having as input the dataset and the number M of buckets that the partitioning should produce. The process starts with all data points in a bucket B (i.e., one rectangle) to which corresponds a cost computed with the selected cost measure, and then among all possible pairs of points, the process selects the pair such that the corresponding rectangle (i.e., the rectangle where the two selected points are the end points of the longest diagonal of the rectangle) reduces the actual cost by the maximum amount. As an example, consider the `AP` relation and its spatial representation in Fig. 8a, where initially all points belong to the drawn rectangular B. Figure 8b shows the two data points chosen, corresponding to the two light gray end points of the dashed line in the figure, which determine the "best rectangle" B' to consider for the creation of a new bucket (i.e., the rectangle that determines the maximum reduction in the cost). After this selection, the data points are partitioned into two buckets: one corresponds to the new rectangle, B', (i.e., the dashed rectangle in Fig. 8b) and the other one corresponds to the initial rectangular B modified so to cover only the data points that are not already covered by B'. Then, the minimum bounding rectangle (i.e., the smallest rectangle that includes all points in a given d-dimensional rectangle) of all rectangular areas affected by the creation of B' (i.e., the rectangular area that lost some data points that are now included in B') is recomputed. After this step, all data points are checked and are possibly reassigned to the actually existing buckets (possibly readjusting their minimum bounding



Enc	I_N	I_P
hjhsd785	α	ν
uiw7d87	α	ν
jfu!&u	β	ξ
nskdf8	β	ξ
jo%$uj1	γ	o
908j8	γ	o
hsd71	δ	π
jkj-9	δ	π

(c)

Fig. 9 Spatial representation (**a**) of the relation in Fig. 7a, its partitioning (**b**), and corresponding encrypted and indexed relation computed as described in [41] (**c**)

rectangles) if the cost can be further reduced. Fig. 8c illustrates the two buckets B and B' resulting after a reassignment has been applied on the two existing buckets. The process continues until M buckets $\{B_1, \ldots, B_M\}$ have been created. In the second phase, the data points in the generated buckets are re-distributed, among the existing buckets, in a way that the average entropy and variance of the distribution of values in the buckets are increased. The rationale is to add "confusion" in the buckets so to make difficult to infer something about the values of the attributes of the tuples in the buckets. The encrypted and indexed relation is then obtained by encrypting the plaintext relation at the tuple level. Each tuple is associated with the identifier of the bucket to which the corresponding plaintext tuple belongs.

The proposal in [41] partitions the tuples in buckets with the goal of constructing a *multi-dimensional flattened index*. The idea is to create buckets with almost the same number of tuples (denoted b). Each bucket is then associated with a set of indexes, one for each attribute of interest of the plaintext relation. In other words, all (and only) tuples in a bucket are associated with the same combination of index values. In this way, the multi-dimensional index is robust against static inferences because the distribution of the combination of index values is almost flat. The partitioning process starts with all points in a bucket, and then selects an attribute/dimension and a threshold value in its domain. The selection of the attribute can be performed considering different metrics (e.g., the attribute with the highest number of distinct values). The multi-dimensional space is partitioned into two subspaces based on the selected threshold, each containing the points falling on its side of the cut (i.e., lower or higher than the threshold). The threshold used for the cut is computed according to the type of the selected attribute. If the attribute is continuous, meaning that the attribute is characterized by a total order relationship on its domain, the threshold corresponds to the median. If the attribute is nominal, meaning that there is not any natural order relationship defined over its domain, the threshold corresponds to the value that splits the space in two sub-spaces with nearly 50% of the tuples each. When the spaces cannot be further divided without generating a sub-space with less than b tuples, the process terminates. As an example, consider the plaintext relation in Fig. 7(a) and suppose that a multi-dimensional index must be supported on attributes `Numant` and `Platitude`, and that the partitioning process must create buckets of at least two tuples. The partitioning process starts by selecting one attribute between `Numant` and `Platitude`. In this example, `Numant` has five distinct values and `Platitude` has 7 distinct values. The selected attribute is then attribute `Platitude`. The median value is 40.374503 and a first cut splits the overall space in Fig. 9a in two subspaces: one contains all points where `Platitude` is greater than or equal to the median and the other one contains all the other points. Since the two sub-spaces contain four points, a further cut is performed on the two sub-spaces. In both cases, the cuts are performed over attribute `Numant` whose median value is 5. Figure 9b illustrates the resulting buckets and Fig. 9c illustrates the encrypted and indexed relation, which contains four groups of tuples corresponding to the four sub-spaces in Fig. 9b. All tuples in a group are associated with the same pair of index values. Note that for readability, the tuples in the encrypted and indexed relation appear in the same order as the tuples in the corresponding plaintext relation in Fig. 7a.

Query Integrity

As discussed in Section "Data Protection", integrity, together with confidentiality, is a critical aspect that must be ensured when data owners outsource their data to the cloud. However, verifying the integrity of outsourced data

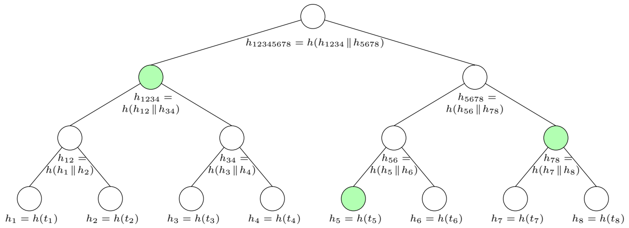

$h_{12345678} = h(h_{1234} \,\|\, h_{5678})$
$h_{1234} = h(h_{12} \,\|\, h_{34})$
$h_{5678} = h(h_{56} \,\|\, h_{78})$
$h_{12} = h(h_1 \,\|\, h_2)$
$h_{34} = h(h_3 \,\|\, h_4)$
$h_{56} = h(h_5 \,\|\, h_6)$
$h_{78} = h(h_7 \,\|\, h_8)$
$h_1 = h(t_1)$
$h_2 = h(t_2)$
$h_3 = h(t_3)$
$h_4 = h(t_4)$
$h_5 = h(t_5)$
$h_6 = h(t_6)$
$h_7 = h(t_7)$
$h_8 = h(t_8)$

Fig. 10 An example of Merkle hash tree for attribute `Apid` of relation `AP` in Fig. 7a

by their owners is only one of the aspects of integrity. In particular, the processing of data (possibly distributed among multiple collaborative providers) requires solutions for verifying the integrity of the data processing result. The integrity aspect is particularly important when the parties involved in a data processing are not trustworthy, meaning that they are not reliable for properly responding to queries. Intuitively, providing integrity of query execution requires guaranteeing that the query result includes all data satisfying the query (completeness) and has been computed correctly (correctness) over the most recent version of truthful data (freshness). Existing solutions can be classified as *deterministic* techniques and *probabilistic* techniques (e.g., [1, 43]). Deterministic techniques are based on the use of *authenticated data structures* that are associated with the outsourced relation and are computed over an attribute of the relation. These techniques permit to detect integrity violations with certainty. Examples of deterministic approaches for correctness/completeness are signature chaining schemas, Merkle hash trees, and skip lists. These authenticated data structures provide deterministic integrity guarantees but only for queries that include conditions over the attribute on which the structure has been built.

As an example, we consider the technique based on the definition of a Merkle Hash Tree (MHT). A Merkle Hash Tree over a relation R is a binary tree that stores, in each leaf, the result of a one-way hash function h applied over a tuple of the relation. The internal nodes store the result of the hash function applied over the concatenation of the values stored at their children. The tuples in the leaves of the MHT are ordered according to the values of an attribute `a`, and the root of the MHT is signed by the data owner, and communicated to users authorized to access the outsourced relation. Figure 10 illustrates an example of a MHT defined over attribute `Apid` of relation `AP` in Fig. 7a. To verify the correctness of a range query over attribute `a` of a relation R, the cloud provider returns to the user a set of tuples in R with contiguous values for `a` (i.e., the tuples resulting from the evaluation of the range query) together with a Verification Object (VO) that includes the values of the nodes needed by the user to compute the hash value of the root. The user then computes the hash value of the root using the VO and the tuples received from the provider, and checks whether such a value corresponds to the root value initially computed by the data owner [44]. If there is a match of the computed value of the root with the known signed root, the query result is complete. Note that the computation of the VO depends on the set of tuples returned. For instance, in case of a point query that returns a specific tuple, the VO contains the values of all the nodes being sibling of those in the path from the root to the leaf corresponding to the returned tuple. As an example, consider the `AP` relation in Fig. 7a and the MHT in Fig. 10 built over attribute `Apid`. Suppose that a user submits query "Select * from `AP` where `Apid`=A6" that returns tuple t_6. To verify the correctness of the query result, tuple t_6 is returned together with a VO that contains the green nodes in Fig. 10 (i.e., t_5, h_{78}, and h_{1234}). The user can then compute the hash of tuple t_6 and combine it with the VO, as illustrated in the figure, to compute the root of the tree, which is then compared to the one computed and signed by the data owner. More precisely, the user computes $h'_5 = h(t_5)$, $h'_6 = h(t_6)$, $h'_{56} = h(h'_5 \,\|\, h'_6)$, $h'_{5678} = h(h'_{56} \,\|\, h_{78})$, and $h'_{12345678} = h(h_{1234} \,\|\, h'_{5678})$, and then verifies whether $h'_{12345678}$ corresponds to the signed value received from the data owner.

Probabilistic techniques [1] enable the assessment of query integrity by injecting control tuples in the stored data or in the computation. The advantage of these solutions with respect to the deterministic approaches is a larger applicability, as they are not limited to operate on a specific attribute. However, not operating on an authenticated data structure, the offered guarantee is only probabilistic, as integrity compromises that affect the completeness of a query result can be detected only if the missed information in a query result corresponds to the control tuples. Control tuples injected into the input dataset are of two kinds: non-genuine tuples (called *sentinels* or *markers*), and controlled replicas of tuples (called *twins*). Absence of a sentinel or of one of the twins (in the presence of the other) from a query result signals its incompleteness. As an example of working of the probabilistic techniques, we consider the approach in [43], where queries are performed by an untrusted computational provider. Suppose that a client wishes to perform a join query over relations `AP(A,N)` and `CDR(I,H)` (see Fig. 11). The cloud providers storing the two relations first inject both sentinels and twins (which are the tuples that satisfy a replication condition communicated to the storage providers by the client). The resulting extended relations (`AP`* and `CDR`*) are then encrypted on-the-fly and sent to the computational provider (in the figure, encrypted values are represented as Greek letters). The encrypted relations $\mathtt{AP}^*_k$ and $\mathtt{CDR}^*_k$ have two attributes: J_k, the encrypted join attribute; and T_k, the encryption of all attributes (including the join attribute). Note that encryption protects both data confidentiality from the computational provider and makes control tuples

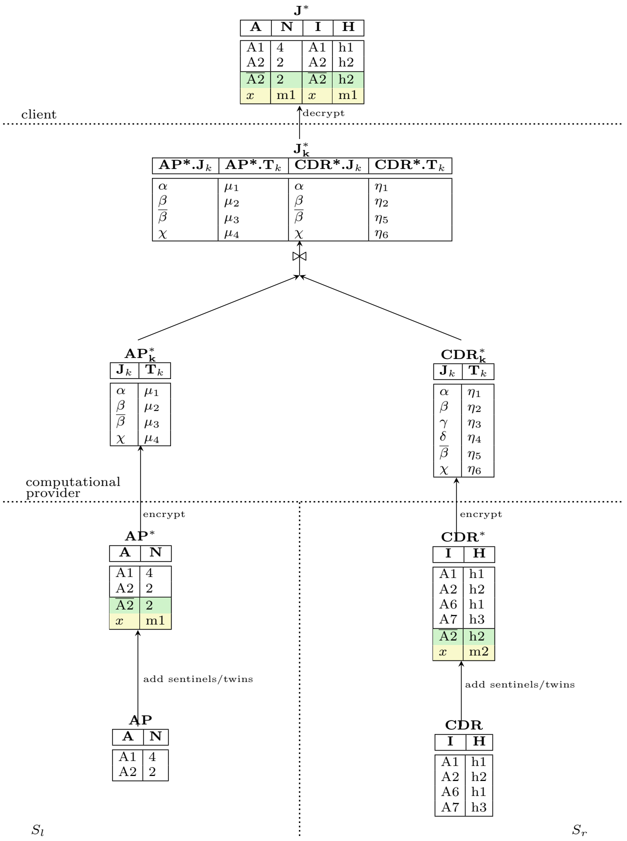


Fig. 11 An example of the evaluation of a join query with twins (green tuples) on ‘A2’ and one marker (yellow tuples)

indistinguishable. The computational provider computes the natural join between the received encrypted relations and sends the result (J_k^*) to the subject. The subject decrypts J_k^*, verifies its (J^*) completeness (i.e., if all the expected sentinels and twins are in J^*) and correctness, and, if no omission is detected, projects over attributes A, N, and H and removes twins and sentinels to obtain the final join result.

Related to the use of sentinels and twins, there is the problem of regulating the injection and distribution of such control data in distributed cloud scenarios where a computation is distributed among different workers [45]. Attention is also needed to the problem of accountability in case of integrity violation.

Controlled Execution of Collaborative Queries

Emerging scenarios require cooperation among providers storing independent data collections for performing distributed collaborative computations, while ensuring that data security and privacy are properly protected. In the following, we first describe existing solutions that focus on the problem of processing distributed queries under protection requirements regulating the selective visibility of data (Section “Distributed Queries Under Protection Requirements”). Then, we describe an approach that can be used in scenarios where multiple external providers offering computational services can be conveniently involved in query execution (Section “Collaborative Queries with Multiple Providers”).

Distributed Queries Under Protection Requirements

In the relational database context, the problem of supporting query execution considering constraints on the visibility of data is addressed through the specification of views and the definition and enforcement of access restrictions over such views (e.g., [46, 47]). Views provide fine-grained content-dependent access control. Basically, a query submitted by a subject over a given set of (basic) relations is rewritten using the views available to the requesting subject. This query rewriting can be performed according to two different models [47]. With the Truman model, the query is transparently rewritten, and the result is returned to the subject. The main problem with this model is that the subject can obtain misleading results as they have been computed over a portion of the dataset (i.e., the one visible to the user). As an example, consider relation CDR in Fig. 3 and suppose a user is permitted to access only the tuples related to her device. In this case, a view is created for the user having the form “CREATE AUTHORIZATION VIEW MyCDR AS SELECT * FROM CDR WHERE Hash-id = $user-hashid”, with $user-hashid a parameter of the view. Suppose now that the user submits the query “SELECT avg(Duration) FROM CDR”. The system modifies this query as “SELECT avg(Duration) FROM MyCDR” and returns the average of the duration of the user’s connections, giving her an impression that her average connection duration is the same as the overall average connection duration. With the non-Truman model, only if the rewritten query is equivalent to the original query (i.e., the rewritten query returns the same result as the original query), the query is considered valid and the result is returned to the requesting subject; otherwise the query is rejected. For instance, the previous query would be rejected.

Other solutions are based on the concept of access pattern (e.g., [48, 49]). An access pattern for a relation specifies which attribute values should be given as input to gain access to the values of (a subset of) the other attributes for the same tuples. For instance, with respect to the Access-Point relation in Fig. 12a, an access pattern over this relation can specify that the identifier Apid of an access point must be provided as input to access attributes Numant, Platitude, and Mlongitude. Relations can then be accessed only according to their corresponding access patterns. The query evaluation process must be revised to take into account the restrictions modeled as access patterns.

Sovereign join [50] is a solution for performing a join operation between two relations in a way that the provider in charge of performing the join operation (which must be different from the owners of the involved relations) cannot infer anything about the operands. Similarly, the owners of the operands do not learn anything about the join result or about the other operand relation. To this purpose, the join operation is performed through a secure coprocessor located at the provider in charge of join evaluation. The query execution then starts with a subject sending a join operation to the provider. Note that the subject submitting a join operation must be different from the owners of the two relations involved in the join. The provider is not trusted for confidentiality, and therefore has an encrypted version of the relations on which the join operation must be computed. The provider sends the join operation and the encrypted relations to the coprocessor. The coprocessor decrypts the relations, performs the join operation, encrypts the query result with a key shared with the requesting subject, and then returns the encrypted results to the provider. Finally, the provider sends the encrypted result to the requesting subject.

Collaborative Queries with Multiple Providers

A recent approach for supporting the collaborative execution of queries over distributed data collections introduces an authorization model regulating the data on which different providers have explicit visibility [51]. This solution supports queries of the general form "SELECT FROM WHERE GROUP BY HAVING" and their execution is performed according to a *query plan* that is represented as a tree T(N), with N the set of nodes in the tree, whose leaf nodes are base relations and whose internal nodes are the relational operations to be executed to perform the query. The query plan is produced with classical optimization criteria, and, in particular, we assume that projections and selections are pushed down to avoid the retrieval of data that are not of interest for the query. The working of this approach will be described with reference to the running example (Section "Introduction"), considering a subject $\mathbb{S}$ interested in performing the query in Fig. 12b over the relations in Fig. 12a (which are the relations of our running example) whose query plan is shown in Fig. 12c. In the query, ts1 and ts2 correspond to two timestamps (e.g., '09:00:00' and '17:00:00', respectively).

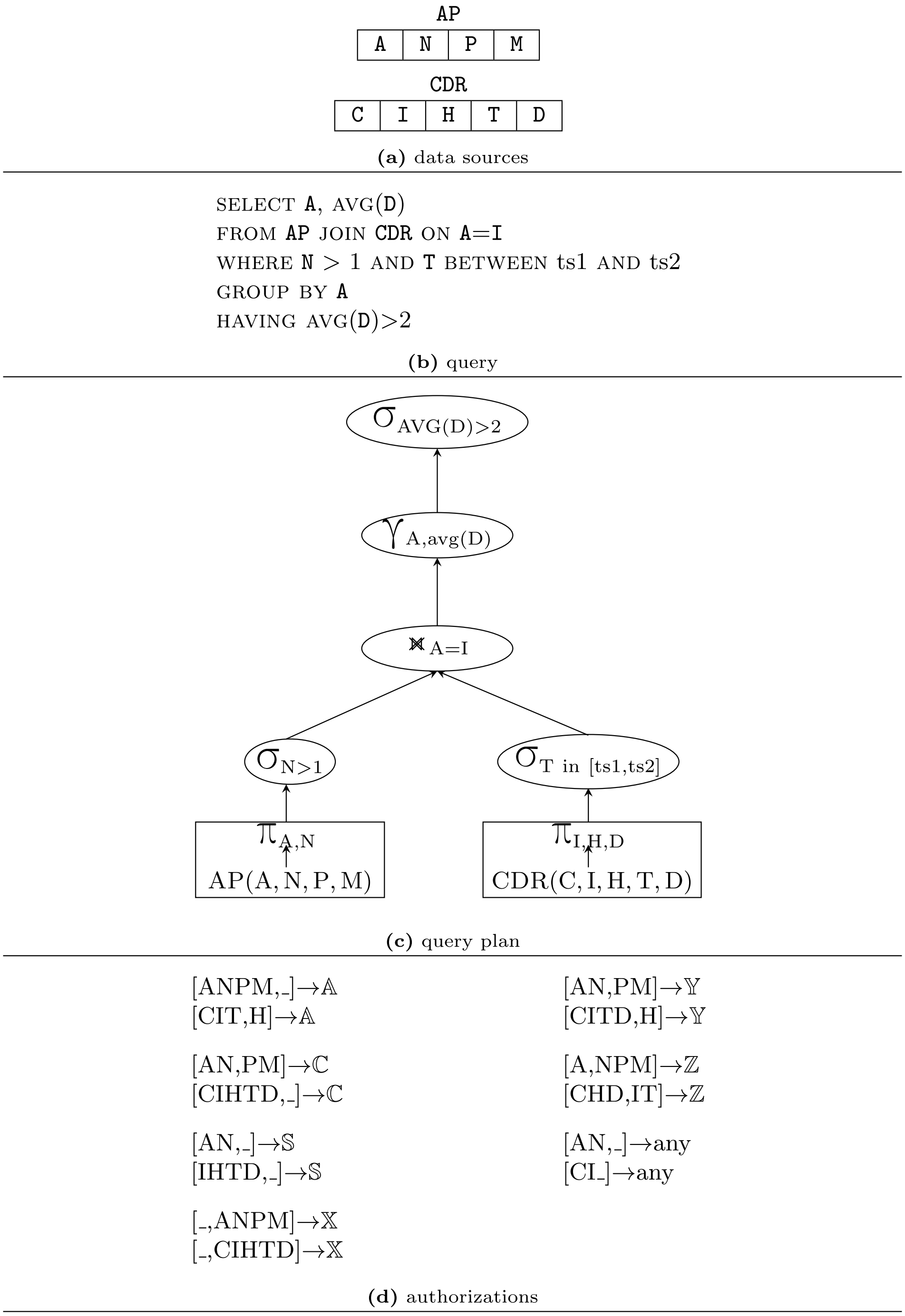
We now illustrate: (i) the model and specification language that the data owners can use to express access restrictions on their data together with the concept of *relation profile* (Section "Policy Specification and Relation Profile"), and (ii) the mechanism enforcing such access restrictions during the execution of queries (Section "Policy Enforcement").

Policy Specification and Relation Profile

Data owners keep control on their data through the definition of authorizations that state which subject can access which data in which form (plaintext or encrypted). Formally, given a relation R and a subject S, an *authorization* over R for S is a rule of the form $[R_p,R_e] \rightarrow S$, with $R_p,R_e \subseteq R$ and $R_p \cap R_e = \emptyset$, stating that subject S can access in plaintext the set $R_p \subseteq R$ of attributes and in encrypted form the set $R_e \subseteq R$ of attributes (S cannot access all the other attributes in R). Clearly, the plaintext visibility over the set R_p of attributes also implies the encrypted visibility over them. Also, since the subjects available in the system, which can be involved in the execution of a query, may not be all known a priori, the proposed approach supports the specification of authorizations that apply to all subjects when no explicit authorization already exists. Such authorizations have "any" as subject of the authorization. Figure 12d shows the set of authorizations defined for the running example. Each data owner can access all attributes of its relation in plaintext (i.e., $\mathbb{A}$ can access in plaintext all attributes of relation AP and $\mathbb{C}$ can access in plaintext all attributes of relation CDR), and can possibly access the attributes of other relations in plaintext or encrypted form (e.g., $\mathbb{A}$ can access attribute H of relation CDR in encrypted form and in plaintext attributes C, I, and T). External subjects offering computational resources only can access a subset of the attributes of the relations managed by the data owners in plaintext or encrypted form. Finally, there are two authorizations with value "any" as subject stating that attributes A and N of relation AP and attributes C and I of relation CDR can be accessed in plaintext by any other subject.

To determine when a subject can access a base or a derived (i.e., resulting from the execution of a query) relation, it is necessary to capture the informative content of the relation itself. Each relation is then associated with a *relation profile*. Intuitively, a relation profile is characterized by three components: the set R^v of visible attributes, that is, the attributes appearing in the schema of the relation; the set R^i of implicit attributes used in the computation of the relation (e.g., attributes involved in a selection condition that, even if removed from the relation schema, might have left a trace of their values in the query result); and the set $R^{\simeq}$ of equivalent attributes, that is, the attributes that have been compared in the computation of the relation. Visible and implicit attributes are distinguished between plaintext (i.e., R^{vp} and R^{ip}) and encrypted (i.e., R^{ve} and R^{ie}). Note that the profile of a relation can be extended to take into consideration the fact that the attributes used in a query can be renamed and that the relations managed by their data owners can also be stored in encrypted form [51, 52].

The profile of a base relation has all the elements but R^{vp} empty, since the relation is assumed accessible in plaintext



AP

A	N	P	M

CDR

C	I	H	T	D

(**a**) data sources

```
SELECT A, AVG(D)
FROM AP JOIN CDR ON A=I
WHERE N > 1 AND T BETWEEN ts1 AND ts2
GROUP BY A
HAVING AVG(D)>2
```

(**b**) query

(**c**) query plan

[ANPM,_]→$\mathbb{A}$
[CIT,H]→$\mathbb{A}$

[AN,PM]→$\mathbb{C}$
[CIHTD,_]→$\mathbb{C}$

[AN,_]→$\mathbb{S}$
[IHTD,_]→$\mathbb{S}$

[_,ANPM]→$\mathbb{X}$
[_,CIHTD]→$\mathbb{X}$

[AN,PM]→$\mathbb{Y}$
[CITD,H]→$\mathbb{Y}$

[A,NPM]→$\mathbb{Z}$
[CHD,IT]→$\mathbb{Z}$

[AN,_]→any
[CI_]→any

(**d**) authorizations

Fig. 12 An example of data sources (**a**), query (**b**), corresponding query plan (**c**), and authorizations on relations AP and CDR (**d**)

by the subject storing it (which, however, does not imply that the relation is stored in plaintext but only that it is accessible in plaintext), and does not carry any implicit content or equivalence relationship.

The profile of a derived relation depends on the profile of the operand relations and on the operators involved in its computation. Every operator operates on visible attributes only (i.e., attributes in R^{vp} and R^{ve}, which belong to the schema of the operand relation R), but it may affect also implicit attributes and the equivalence relationship in the profile of the resulting relation. For instance, the selection of tuples having N > 1 operates on visible attribute N, which is inserted into the implicit component after the operation has been evaluated. Figure 13 illustrates the graphical representation of the relation profiles resulting from relational operations, encryption operation, and decryption operation, together with an example for each operator on our running example. The relation profile of a relation obtained through the execution of the operation represented by a node in a query plan is shown as a dotted flag attached to the node itself with three components: v (visible attributes R^{vp} and R^{ve}), i (implicit attributes R^{ip} and R^{ie}), and $\simeq$ (sets of equivalent attributes $R^{\simeq}$). Within visible and implicit attributes, we distinguish the encrypted ones (i.e., R^{ve} and R^{ie}) by representing them on a gray background. As we will discuss in the following section, query plans can be extended with the insertion of: encryption operations, represented as gray boxes, containing the attributes to be encrypted on top of the operand relation; and decryption operations, represented as white boxes, containing the attributes to be decrypted, below the node representing the operator.

We now discuss the profile resulting from the application of each operator, focusing on profile components affected by the operator evaluation, while not discussing components that remain unchanged.

- **Projection (π)** The profile of the resulting relation contains, in the visible attributes, only the attributes that have been projected.
- **Selection (σ)** For conditions of the form 'a *op* x', with x a value, attribute a is added to the implicit component (either encrypted or plaintext, consistently with the visibility of a in the operand). For conditions of the form 'a_i *op* a_j', equivalence $\{a_i, a_j\}$ is added to the equivalence set.
- **Cartesian product** ($\times$) The profile of the resulting relation is obtained by taking the union of the corresponding components in the profiles of the operands.
- **Join** ($\bowtie$) It is equivalent to a selection with a (conjunction of) conditions of the form 'a_i *op* a_j', which is applied to the Cartesian product of the operands (i.e., $\sigma_C(R_l \times R_r)$). The profile of the result reflects then the information conveyed by both these operators.
- **Group by (γ)** The profile of the resulting relation contains, in the visible attributes, only those attributes on which the grouping (A) and aggregate function (a) operate (when $f(a)$ is COUNT(*), only attributes in A are maintained). Attributes appearing in the grouping function (A) are added to the implicit attributes (to capture the possible information leakage from their grouping).
- **Encryption** Attributes on which encryption is applied are moved from visible plaintext to visible encrypted component.
- **Decryption** Attributes on which decryption is applied are moved from visible encrypted to visible plaintext component.

Figure 14a illustrates the profiles of the relations resulting from the operations in the query in Fig. 12c.

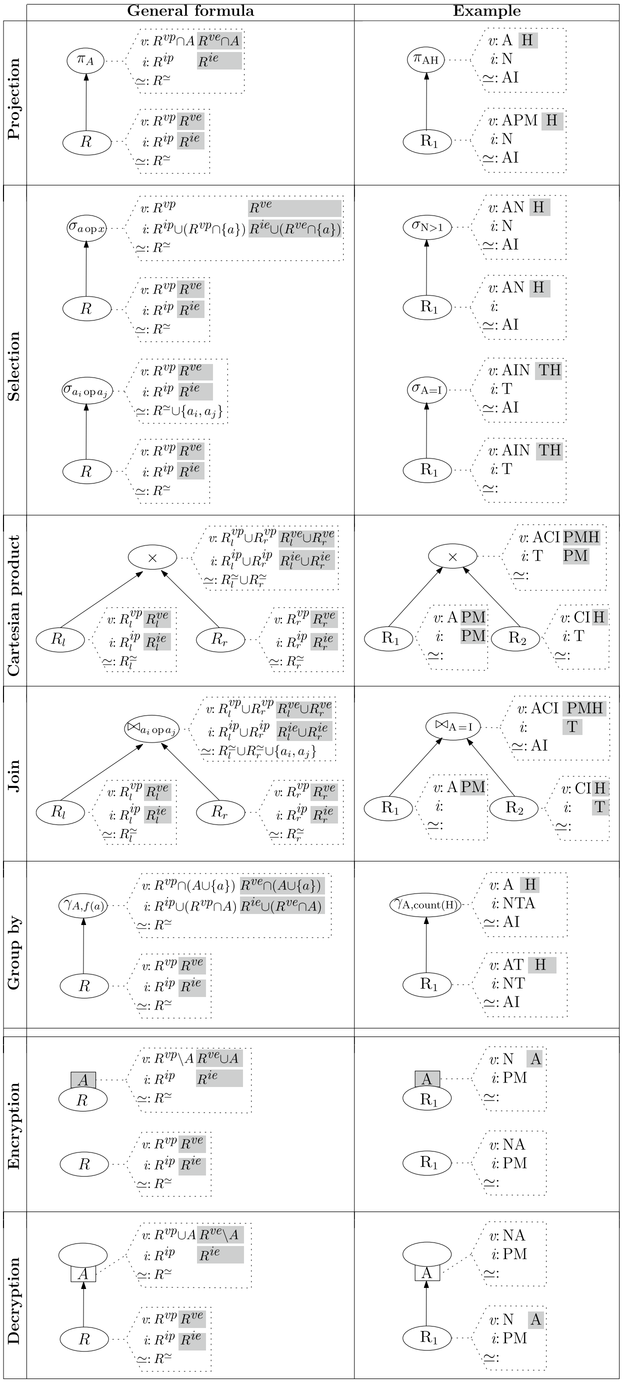


Fig. 13 Graphical representation of the profiles resulting from relational, encryption, and decryption operations

Policy Enforcement

Given a (base or derived) relation, we can check whether a subject can access the relation comparing the relation profile with the authorizations of the subject. A subject can access a relation (authorized visibility) if the subject is authorized to access: *1)* in plaintext, all plaintext attributes appearing in the visible and implicit component of the relation profile; *2)* in plaintext or encrypted form, all encrypted attributes appearing in the visible and implicit component of the relation profile; *3)* in the same form (plaintext or encrypted) all attributes appearing in the same set of the equivalent component of the relation profile. The first two conditions correspond to the classical enforcement of policy rules considering both the visible and implicit components of a relation profile. The third condition prevents unintended information leakage of attribute values due to comparison in query evaluation. As an example, consider the authorizations in Fig. 12(d) and a relation *R* with profile [H,AIPM,_,_,{AI}]:

- 𝔸, 𝕏, and 𝕐 are not authorized for *R* (condition 1, attribute H);
- ℂ is authorized for *R*;
- 𝕊 is not authorized for *R* (condition 2, attribute P and attribute M);
- ℤ is not authorized for *R* (condition 3, attribute A and attribute I).

The definition of authorized visibility is then used to regulate the assignment of operations in a query plan to a subject in respect of the authorization policy. Intuitively, an operation in a query plan operates on one or two operand relations, and produces a relation as output. Since the relation profile of a relation resulting from the evaluation of an operation captures all the information necessary to compute the operator, a subject is authorized for the execution of the operation if and only if it is authorized for all the relations involved: the operand(s) and the operation result. Given a query plan T(N), the goal is to produce an *authorized assignment* of operations to subjects. Note that any assignment obtained selecting, for each operation in T(N), a subject authorized to execute the operation can be made authorized by injecting encryption and dencryption operations. An authorized assignment can be determined by applying a three-phase process: (1) for each operation in the query plan, compute the set of *candidates* (i.e., authorized subjects); (2) for each operation, select an assignee according to a parameter of interest (e.g., economic cost); (3) inject encryption/decryption operations in the query plan to make the assignment authorized and to satisfy operation visibility requirements (i.e., some operations cannot be evaluated over encrypted attribute values, and therefore such attributes must be available in plaintext). We refer to a query plan enriched with encryption/decryption operations as an *extended query plan*. Note that the encryption operations necessary to make an assignment authorized depend on the subject to which an operation is assigned. For instance, for the query in Fig. 12c, attributes A and I would need to be encrypted for assigning the execution of the join to 𝕏 but could remain in plaintext if the join is assigned to ℂ.

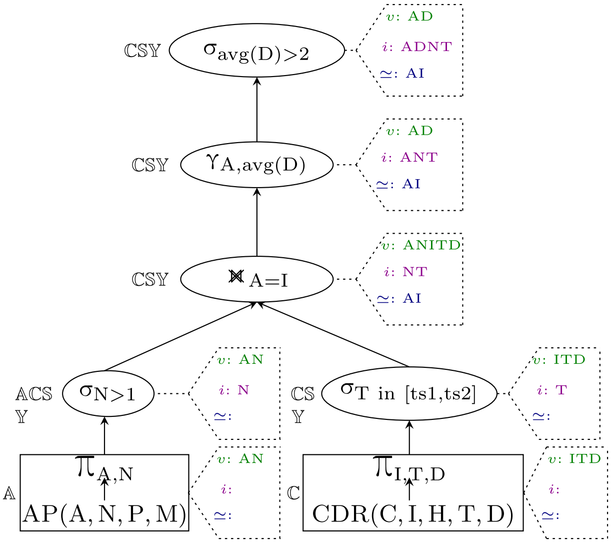


Fig. 14 Query plan with relation profiles and candidates for the query in Fig. 12c assuming all attributes in plaintext

Decryption operations are instead applied when the execution of an operation over an attribute cannot be performed over its encrypted representation.

Candidate computation Different strategies can be adopted for finding the set of candidates. A first possible strategy consists in leaving all attributes in plaintext in the query plan. The main advantage of this strategy is that operations are always performed over plaintext data. The disadvantage is the limited number of candidates to which operations can be assigned. Figure 14a illustrates for the query of the running example, the candidates for each operation assuming all attributes in plaintext. Here, for example, only subjects ℂ and 𝕊 can perform the selection over attribute T (as well as all the following operations in the query plan).

An alternative strategy consists in encrypting all attributes. This increases the number of subjects to which each operation can be assigned but it would prevent the evaluation of operations requiring plaintext visibility over attributes. For instance, considering the query of our running example suppose that the last selection cannot be performed over encrypted values. Encrypting attribute D would prevent the possibility of executing such a selection at an external server. To avoid this problem and to have the possibility of selecting an assignment considering a larger number of candidates, we assume that all (visible) attributes of a basic (leaf nodes of a query plan) or derived (internal nodes of a query plan) relation are encrypted but those that need to be in plaintext for the execution of the operation. All subjects that are authorized to access these relations can be considered candidates for executing the operation.

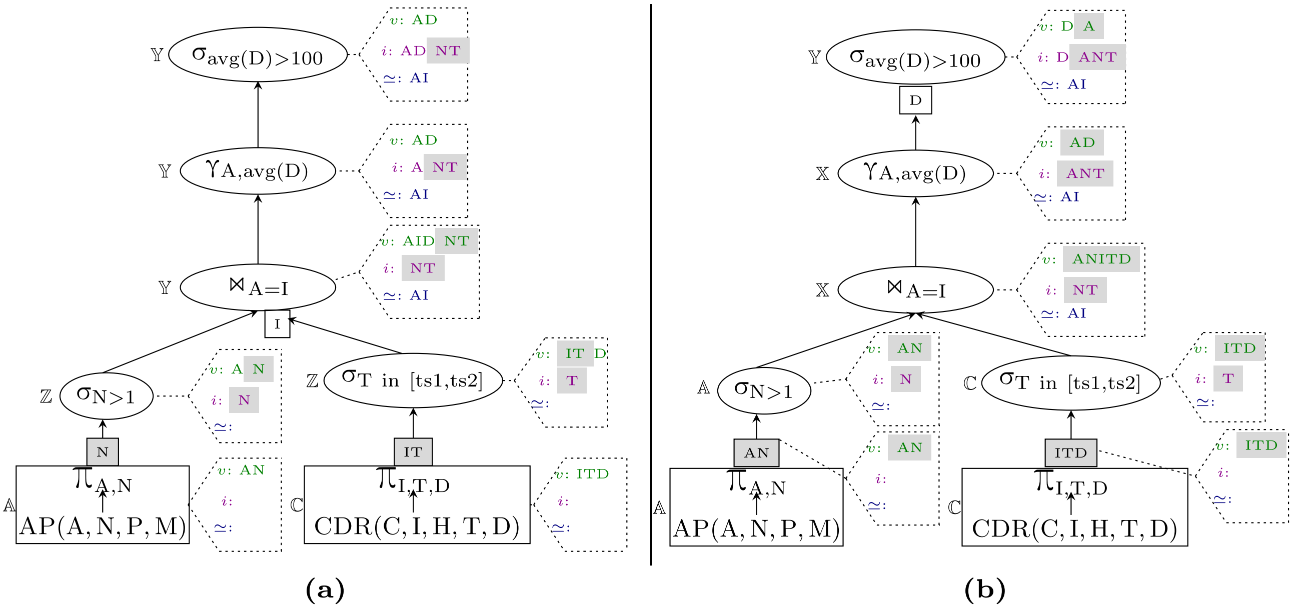


Fig. 15 An example of two extended authorized query plans for the query plan in Fig. 12c

Assignee selection After each operation in the query plan has been associated with the set of candidates, the assignees are selected from the corresponding candidate sets. We note that any candidate in a candidate set can be selected independently from the candidate selected for other nodes since any assignment can be made authorized by injecting encryption and decryption operations as needed. Among all the possible assignments, we can select the one that optimizes a parameter of interest, such as cost or performance.

The computation of a minimum cost assignment needs to consider the cost of executing a computation, the cost of transferring data between different subjects involved in the computation, and also the cost of possible encryption/decryption operations that would be required for the selected candidates to be authorized for the operations assigned to them [51].

Encryption/decryption injection After the selection of the assignee of each operation in the plan, encryption and decryption operations are injected in the query plan as needed to guarantee authorization enforcement and operation execution. In particular, for each operation (node) in the query plan, a decryption operation is inserted for those attributes that must be in plaintext for the evaluation of the operation but that are encrypted in its operand(s). Encryption operations are instead injected after the execution of an operation in the query plan for all attributes appearing in plaintext and that the subject receiving the operation result can access only in encrypted form. Figure 15 illustrates two extended query plans for the running example, assuming operations allocated to the subject indicated on the left of each node. In the plan in Fig. 15a attribute N of relation AP and attributes I and T of relation CDR are encrypted before being transmitted to ℤ since ℤ cannot access them in plaintext. However, attribute I has to be decrypted before the execution of the join operation by 𝕐. In the plan in Fig. 15b, all attributes of the two relations AP and CDR are encrypted before being transmitted to 𝕏 since 𝕏 cannot access them in plaintext. Note also that attributes N and T must be encrypted before the execution of the selection operations because, otherwise, the operations would leave an implicit plaintext trace in the computation that 𝕏, executing the subsequent operations, cannot access. Note also that AVG(D) is decrypted before the execution of the final selection operation because we assume that the execution of this operation can be evaluated on plaintext values only.

Conclusions

The adoption of cloud-based solutions provides several advantages to data owners and users, who can enjoy efficient and scalable services for the storage, management, and processing of their data. At the same time, data externally stored may be sensitive or company confidential, and hence their confidentiality and integrity—and in some cases also confidentiality of accesses executed on them as well as the integrity of data resulting from outsourced computations,—should be guaranteed, even with respect to the storage and processing provider(s). There are several challenges to tackle, which require the investigation of new issues and the design of novel technological solutions to address them. This paper has discussed problems to be addressed and illustrated some directions introducing emerging approaches to protect data and computations in cloud-based scenarios.

Acknowledgements This work was supported in part by the EC under projects MARSAL (101017171), GLACIATION (101070141), and by project SERICS (PE00000014) under the MUR NRRP funded by the EU - NextGenerationEU.

Funding Open access funding provided by Università degli Studi di Milano within the CRUI-CARE Agreement.

Data availability No data have been generated or analyzed.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

References

1. De Capitani di Vimercati S, Foresti S, Livraga G, Samarati P. Practical techniques building on encryption for protecting and managing data in the cloud. In: Ryan P, Naccache D, Quisquater J-J, editors. The new codebreakers. Springer; 2016.
2. Jhawar R, Piuri V. Fault tolerance and resilience in cloud computing environments. In: Vacca J, editor. Computer and information security handbook. 2nd ed. Morgan Kaufmann; 2013. p. 125–41 (**978-0-1239-4397-2**).
3. Tang J, Cui Y, Li Q, Ren K, Liu J, Buyya R. Ensuring security and privacy preservation for cloud data services. ACM CSUR. 2016;49(1):1–39.
4. Jhawar R, Piuri V. Fault tolerance management in IaaS clouds. In: Proc. of ESTEL, Rome, Italy, 2012; pp. 1–6.
5. De Capitani di Vimercati S, Foresti S, Livraga G, Piuri V, Samarati P. A fuzzy-based brokering service for cloud plan selection. IEEE SJ. 2019;13(4):4101–9.
6. De Capitani S, Foresti S, Livraga G, Piuri V, Samarati P. Supporting user requirements and preferences in cloud plan selection. IEEE TSC. 2021;14(1):274–85.
7. De Capitani S, Foresti S, Livraga G, Piuri V, Samarati P. Security-aware data allocation in multicloud scenarios. IEEE TDSC. 2021;18(5):2456–68.
8. Garg S, Versteeg S, Buyya R. A framework for ranking of cloud computing services. FGCS. 2013;29(4):1012–23.
9. Li A, Yang X, Kandula S, Zhang M. CloudCmp: comparing public cloud providers. In: Proc. of ACM IMC, Melbourne, Australia 2010.
10. Jhawar R, Piuri V, Santambrogio M. A comprehensive conceptual system-level approach to fault tolerance in cloud computing. In: Proc. of SysCon, Vancouver, BC, Canada 2012.
11. Jhawar R, Piuri V, Santambrogio M. Fault tolerance management in cloud computing: a system-level perspective. IEEE SJ. 2013;7(2):288–97.
12. De Capitani di Vimercati S, Foresti S, Livraga G, Samarati P. Supporting users in data outsourcing and protection in the cloud. In: Helfert M, Ferguson D, Munoz V, Cardoso J, editors. International Conference on Cloud Computing and Services Science. USA: Springer; 2017.
13. Ateniese G, Burns R, Curtmola R, Herring J, Kissner L, Peterson Z, Song D. Provable data possession at untrusted stores. In: Proc. of ACM CCS, Alexandria, VA, USA; 2007.
14. Juels A, Kaliski B. PORs: Proofs of retrievability for large files. In: Proc. of ACM CCS, Alexandria, VA, USA; 2007.
15. De Capitani di Vimercati S, Erbacher R, Foresti S, Jajodia S, Livraga G, Samarati P. Encryption and fragmentation for data confidentiality in the cloud. In: Aldini A, Lopez J, Martinelli F, editors. Foundations of security analysis and design VII. Springer; 2014.
16. De Capitani di Vimercati S, Foresti S, Paraboschi S, Pelosi G, Samarati P. Access privacy in the cloud. In: Ray I, Ray I, Samarati P, editors. From database to cyber security. Springer; 2018.
17. Aggarwal G, Bawa M, Ganesan P, Garcia-Molina H, Kenthapadi K, Motwani R, Srivastava U, Thomas D, Xu Y. Two can keep a secret: A distributed architecture for secure database services. In: Proc. of CIDR, Asilomar, CA, USA. 2005.
18. Ciriani V, De Capitani S, Foresti S, Jajodia S, Paraboschi S, Samarati P. Combining fragmentation and encryption to protect privacy in data storage. ACM TISSEC. 2010;13(3):22–12233.
19. Bacis E, De Capitani S, Foresti S, Paraboschi S, Rosa M, Samarati P. Securing resources in decentralized cloud storage. IEEE TIFS. 2020;15(1):286–98.
20. Bacis E, De Capitani di Vimercati S, Foresti S, Paraboschi S, Rosa M, Samarati P. Mix &slice: efficient access revocation in the cloud. In: Proc. of CCS, Vienna, Austria; 2016.
21. Bacis E, De Capitani di Vimercati S, Foresti S, Paraboschi S, Rosa M, Samarati P. Dynamic allocation for resource protection in decentralized cloud storage. In: Proc. of GLOBECOM, Waikoloa, Hawaii, USA; 2019.
22. De Capitani S, Foresti S, Jajodia S, Paraboschi S, Samarati P. Encryption policies for regulating access to outsourced data. ACM TODS. 2010;35(2):1–46.
23. Atallah M, Blanton M, Fazio N, Frikken K. Dynamic and efficient key management for access hierarchies. ACM TISSEC. 2009;12(3):18–11843.
24. Zhang Y, Deng R, Xu S, Sun J, Li Q, Zheng D. Attribute-based encryption for cloud computing access control: a survey. ACM CSUR. 2020;53(4):1–41.
25. De Capitani S, Foresti S, Jajodia S, Livraga G, Paraboschi S, Samarati P. Distributed query execution under access restrictions. COSE. 2023;127:1–18.
26. De Capitani di Vimercati S, Foresti S, Jajodia S, Paraboschi S, Pelosi G, Samarati P. Encryption-based policy enforcement for cloud storage. In: Proc. of SPCC, Genova, Italy; 2010.
27. Zhao F, Nishide T, Sakurai K. Realizing fine-grained and flexible access control to outsourced data with attribute-based cryptosystems. In: Proc. of ISPEC, Guangzhou, China; 2011.
28. De Capitani di Vimercati S, Foresti S, Paraboschi S, Pelosi G, Samarati P. Shuffle index: efficient and private access to outsourced data. ACM TOS. 2015;11(4):1–55 (**Article 19**).
29. Samarati P. Protecting respondents' identities in microdata release. IEEE TKDE. 2001;13(6):1010–27.
30. De Capitani di Vimercati S, Facchinetti D, Foresti S, Oldani G, Paraboschi S, Rossi M, Samarati P. Scalable distributed data

anonymization. In: Proc. of PerCom, Kassel, Germany (virtual); 2021.

31. De Capitani di Vimercati S, Facchinetti D, Foresti S, Oldani G, Paraboschi S, Rossi M, Samarati P. Artifact: Scalable distributed data anonymization. In: Proc. of PerCom, Kassel, Germany (virtual); 2021.
32. De Capitani di Vimercati S, Foresti S, Paraboschi S, Pelosi G, Samarati P. Three-server swapping for access confidentiality. IEEE TCC. 2018;6(2):492–505.
33. Gentry C. Fully homomorphic encryption using ideal lattices. In: Proc. of STOC, Bethesda, MA, USA; 2009.
34. Li D, Lv S, Huang Y, Liu Y, Li T, Liu Z, Guo L. Frequency-hiding order-preserving encryption with small client storage. PVLDB. 2021;14(14):3295–307.
35. Poh G, Chin J, Yau W, Choo K, Mohamad M. Searchable symmetric encryption: designs and challenges. ACM CSUR. 2017;50(3):1–37.
36. De Capitani di Vimercati S, Foresti S, Samarati P. Selective and fine-grained access to data in the cloud. In: Jajodia S, Kant K, Samarati P, Swarup V, Wang C, editors. Secure cloud computing. Springer; 2014.
37. Ceselli A, Damiani E, Capitani De, di Vimercati S, Jajodia S, Paraboschi S, Samarati P. Modeling and assessing inference exposure in encrypted databases. ACM TISSEC. 2005;8(1):119–52.
38. Damiani E, De Capitani di Vimercati S, Jajodia S, Paraboschi S, Samarati P. Balancing confidentiality and efficiency in untrusted relational DBMSs. In: Proc. of CCS, Washington, DC, USA; 2003.
39. Wang P, Ravishankar C. Secure and efficient range queries on outsourced databases using r-trees. In: Proc. of IEEE ICDE, Brisbane, Australia; 2013.
40. Wu Z, Li K. VBTree: forward secure conjunctive queries over encrypted data for cloud computing. VLDB J. 2019;28:25–46.
41. De Capitani di Vimercati S, Facchinetti D, Foresti S, Oldani G, Paraboschi S, Rossi M, Samarati P. Multi-dimensional indexes for point and range queries on outsourced encrypted data. In: Proc. of GLOBECOMM, Madrid, Spain; 2021.
42. Hore B, Mehrotra S, Canim M, Kantarcioglu M. Secure multidimensional range queries over outsourced data. VLDB J. 2012;21(3):333–58.
43. De Capitani di Vimercati S, Foresti S, Jajodia S, Livraga G, Paraboschi S, Samarati P. Integrity for distributed queries. In: Proc. of CNS, San Francisco, CA, USA; 2014.
44. Devanbu P, Gertz M, Martel C, Stubblebine S. Authentic third-party data publication. In: Proc. of DBSec, Schoorl, The Netherlands; 2000.
45. De Capitani di Vimercati S, Foresti S, Jajodia S, Paraboschi S, Sassi R, Samarati P. Sentinels and twins: effective integrity assessment for distributed computation. IEEE TPDS. 2023;34(1):108–22.
46. Guarnieri M, Basin D. Optimal security-aware query processing. PVLDB. 2014;7(12):1307–18.
47. Rizvi S, Mendelzon A, Sudarshan S, Roy P. Extending query rewriting techniques for fine-grained access control. In: Proc. of SIGMOD, Paris, France; 2004.
48. Amarilli A, Benedikt M. When can we answer queries using result-bounded data interfaces? In: Proc. of PODS, Houston, TX, USA; 2018.
49. Benedikt M, Leblay J, Tsamoura E. Querying with access patterns and integrity constraints. PVLDB. 2015;8(6):690–701.
50. Agrawal R, Asonov D, Kantarcioglu M, Li Y. Sovereign joins. In: Proc. of ICDE, Atlanta, GA, USA; 2006.
51. De Capitani di Vimercati S, Foresti S, Jajodia S, Livraga G, Paraboschi S, Samarati P. An authorization model for query execution in the cloud. The VLDB J. 2022;31(3):555–79.
52. De Capitani di Vimercati S, Foresti S, Jajodia S, Livraga G, Paraboschi S, Samarati P. Distributed query evaluation over encrypted data. In: Proc. of DBSec, Calgary, Canada (virtual); 2021.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.