Query Integrity in Smart Environments

Sabrina De Capitani di Vimercati[0000−0003−0793−3551], Sara Foresti[0000−0002−1658−6734], and Pierangela Samarati[0000−0001−7395−4620]

Computer Science Department, Università degli Studi di Milano
{sabrina.decapitani,sara.foresti,pierangela.samarati}@unimi.it

Abstract. Our smart society strongly relies on data, which are continuously generated, collected, stored, and processed by millions of connected IoT devices and smart sensors. Such data are at the basis of typically complex decision-making processes that require advanced analytics.
Due to the vast and increasing amount of data, their storage and processing are often outsourced to third parties (e.g., service providers and decentralized computational services) that might be not fully trustworthy in their operating. In this chapter, we focus on the problem of assessing integrity of query computations involving external service providers, and illustrate possible approaches for enabling the verification of the integrity of query results. We will cover both deterministic approaches, based on the definition of authenticated data structures over the data and giving full integrity guarantees, and probabilistic approaches, based on the insertion of control information in the data and providing probabilistic integrity guarantees.

Keywords: Query integrity, deterministic techniques, probabilistic techniques

1 Introduction

The advancements of digital and smart technologies (e.g., Internet of Things, big data analytics, and 5G/6G connectivity) are at the basis of today's smart society that supports new applications in a variety of sectors, also thank you to the availability of a powerful hyperconnected infrastructure offering unprecedented network capacity and speed. Distributed sensors, mobile and pervasive devices, cloud/edge/fog computational and storage nodes, can all be involved in providing advanced storage and computational services and applications. At the center of such novel scenarios are *data*, gathered, generated, shared, processed, and communicated among the different components of the infrastructure at an incredible pace. The possibility of efficiently performing analysis on such data for making data-informed decisions becomes then extremely important. Often data storage, as well as data analytics, are outsourced to external providers or rely on the involvement of a distributed framework for storing and processing large datasets (e.g., peer-to-peer networks [6], Apache Hadoop [20], and Spark [41]). Outsourcing data and data analytics brings several advantages, including cost savings, increased efficiency, and flexibility. However, such advantages come at

the price of the data owners losing control over their own data and processing, and introducing therefore the problem of their proper protection [12].

The problem of protecting data and computations managed by an external service provider, allowing data owners to keep the control over them, has many facets. The service provider can be *honest-but-curious* (i.e., the service provider is trusted for the management of data but, at the same time, it is not trusted with respect to the data content, which should remain confidential), or can be *lazy* (i.e., the service provider might not be considered fully trustworthy and, for example, might delete data that are accessed rarely or omit some computations to save resources) or *malicious* (i.e., the service provider may intentionally behave improperly in the storage and processing of data) and its behavior should be controlled. Depending on the trust assumption on the service provider, there are different security problems that need to be addressed, including the confidentiality and integrity of data and computations, the management and specification of policies, the exposure to different cyber-attacks, and the reliability and availability of services (e.g., [2, 12, 14, 17, 21]). For instance, natural solutions for protecting data confidentiality are based on *encryption* [30] (i.e., data are protected by encrypting them before their storage at external service providers), and *fragmentation* [1, 5] (i.e., data are split in different non-linkable fragments to protect sensitive associations). Data integrity is usually ensured through the application of solutions that rely on *encryption* such as digital signatures, Proof Of Retrievability, and Provable Data Possession (e.g., [3, 22]).

When the trustworthiness of the service provider cannot be taken for granted, data owners may be concerned about the correctness of the results (retrieved data or queries) received from the service provider. As a matter of fact, the lack of control of data owners and the open nature of the adopted storage and processing platforms may open the door to possible misbehavior by the providers involved in data storage and computation. It is therefore important to design efficient and practical techniques that enable data owners and, in general, clients asking the service provider to perform a query over the data, to assess the integrity of every computation performed by the service provider.

The goal of this chapter is to provide an overview of the main techniques for verifying the *integrity* of query results. The remainder of the chapter is organized as follows. Section 2 presents the reference scenario and discusses the integrity verification objectives (i.e., correctness, completeness, and freshness) together with the main characteristics of the two categories (i.e., deterministic and probabilistic) of integrity verification techniques. Section 3 describes the main deterministic techniques, that is, solutions that provide deterministic integrity guarantees, relying on authenticated data structures. Such techniques include those based on signatures, tree-based data structures, and list-based data structures. Section 4 presents the main techniques (sentinels and twins) that provide probabilistic integrity guarantees, and illustrates how to assess the completeness of join queries. Section 5 discusses some open research directions. Finally, Section 6 gives our conclusions.

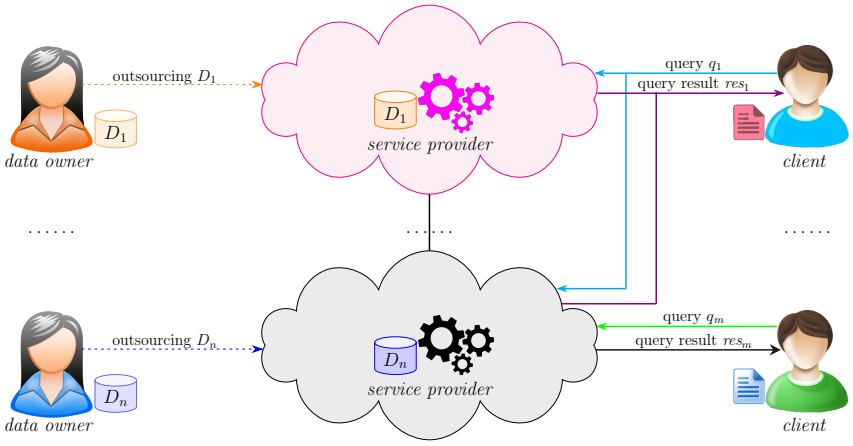


Fig. 1. Reference outsourcing scenario

2 Query Integrity Verification

We consider a data outsourcing scenario that is usually characterized by three interacting parties: data owners, service providers, and clients (see Figure 1). The *data owner* is a company or an individual who outsources their data to an external *service provider*. The service provider offers storage and/or computation resources for the management of data. The *client* is a company or an individual who can require the execution of a computation over the outsourced data. For concreteness, we will consider the evaluation of queries over an (outsourced) dataset composed of one or more relations. Each query can involve the data managed by a service provider or can also involve data stored and managed by different service providers (e.g., a join query), which have to collaborate for query execution. If the service providers are not trustworthy, they can return incorrect query results. There can be multiple reasons for a query result to be incorrect, including a temporary misconfiguration, or a malicious action by the service provider (which may want to either sabotage the execution of the query or get the reward for computing queries while omitting to do so). Note that there is an integrity issue regardless of whether the incorrect result has been caused by failure, malfunctioning, sloppiness, or intentional opportunistic behavior since they all have the effect of the service provider not correctly computing the queries submitted to it. The problem then arises of providing clients with the ability to assess the integrity of query results, that is, to assess whether, given a query q, the result res obtained by the execution of q over the outsourced dataset D, denoted $q(D)$, (i.e., $res{=}q(D)$) is *correct*, *complete*, and *fresh*.

– *Correct*. The query result res is correct if the data items in res have been obtained from the execution of q on dataset D, and have not been tampered with.

- *Complete*. The query result *res* is complete if no valid data is missing from the result.
- *Fresh*. The query result *res* is fresh if it has been obtained by executing the query q over the most recent version of dataset D.

Most of the current approaches provide guarantees of completeness and correctness, with a few proposals complementing them with timestamps or periodical refreshing to provide freshness guarantees. In the following, we present some of the most well-known solutions proposed for ensuring completeness and correctness of query results.

Integrity verification techniques can be classified in two main categories: *deterministic* approaches, which provide integrity guarantees with full confidence, and *probabilistic* approaches, which provide integrity guarantees with a certain degree of confidence [13].

Deterministic approaches (Section 3). Deterministic approaches are typically based on the generation of a proof, called *Verification Object* (VO), which provides deterministic integrity guarantees. Intuitively, given a query q, the service provider executes the query over the outsourced dataset D and constructs a VO for the query result $res{=}q(D)$. The service provider then returns to the client the query result *res* together with the verification object. The construction of the verification object is based on an *authenticated data structure* that is built over D by the owner of the data and is stored together with the data at the external service provider. In addition to the authenticated data structure, other pieces of information can be distributed to the client. The client then uses the VO and the information possibly received from the owner of the data to verify whether *res* satisfies certain properties (i.e., correctness, completeness, and freshness). Deterministic techniques can assess with full confidence the integrity of the result of queries defined over the attribute(s) on which the authenticated data structure has been defined; no guarantee is instead provided for queries using other attributes.

Other deterministic solutions are based on advanced cryptographic protocols (e.g., [24]) and trusted hardware (e.g., [43]). This chapter, however, focuses on those based on the definition of an authenticated data structure mainly because they are largely used and practical.

Probabilistic approaches (Section 4). Probabilistic approaches provide probabilistic integrity guarantees, meaning that a query result that violates the integrity property passes the integrity check with a given probability. Such techniques enable the assessment of query integrity by injecting *control information* in the dataset. The advantage of the probabilistic approaches is a larger applicability than deterministic approaches, as they are not limited to operate considering queries defined over a specific attribute. The offered integrity guarantee is probabilistic because an integrity compromise can be detected only if it affects the control information. Control information is of two types: non-genuine data (called *sentinels* or *markers*) injected in the original dataset, or controlled replication of data (*twinning*). Absence of a sentinel or of one replica (in the presence of the other) from a query result signals that the query result is not

HospitalPatient (HP)

	PId	PName	YoB	Doc
t_1	12	Amos	1961	d50
t_2	14	Bea	2000	d52
t_3	16	Cal	1985	d52
t_4	20	Dennis	1933	d54
t_5	22	Ethan	1973	d56
t_6	24	Frank	1965	d58
t_7	26	Grady	1953	d60
t_8	28	Helen	1987	d62
t_9	30	Ian	1987	d60
t_{10}	32	Loretta	1961	d60

stored at $\mathbb{H}$

FamilyDoctor (FD)

	DId	Name	Phone	Specialty
f_1	d50	Ann	123876	Cardiology
f_2	d52	Bart	784309	Allergy
f_3	d54	Carl	619345	Dermatology
f_4	d56	Dexter	914382	Nephrology
f_5	d58	Elen	903658	Cardiology
f_6	d60	Frank	814309	Urology
f_7	d62	George	357823	Psychiatry
f_8	d64	Hal	813456	Neurology

stored at $\mathbb{M}$

Fig. 2. Relations of the running example

complete. The incorrect value for a sentinel or different results over twins signal that the query result is not correct. The guarantee is probabilistic because omissions or incorrect values of data items that are neither sentinels nor twins cannot be detected. The probability of detecting an integrity violation depends on the amount of sentinels and twins injected, as we elaborate next.

In the remainder of this chapter, we will describe the main deterministic and probabilistic integrity verification techniques. To fix ideas and make the discussion clear, the examples will refer to queries operating over the relations in Figure 2. Relation `HospitalPatient` (`HP`) contains the identifier (attribute `PId`), name (attribute `PName`), year of birth (attribute `YoB`), and the identifier of the family doctor (attribute `Doc`) of the patients of a hospital $\mathbb{H}$, which is the owner of the relation. Relation `FamilyDoctor` (`FD`) contains the identifier (attribute `DId`), name (attribute `Name`), phone number (attribute `Phone`), and specialty (attribute `Specialty`) of family doctors who practice in Milan (Italy), and the municipality $\mathbb{M}$ of Milan is the owner of the relation.

3 Deterministic Approaches

Deterministic integrity verification techniques are based on the definition, and storage at the service provider, of an authenticated data structure built on one of the attributes (or set thereof) in the outsourced dataset. Deterministic approaches can be classified based on the kind of authenticated data structure used for integrity verification, which can be a chain of *signatures*, a *tree-based* structure, or a *list-based* structure, as illustrated in the following.

3.1 Signatures-based Approaches

Signature-based techniques rely on the *digital signature* of tuples for generating a verification object (e.g., [4, 26, 42]). The data owner signs (e.g., using an RSA

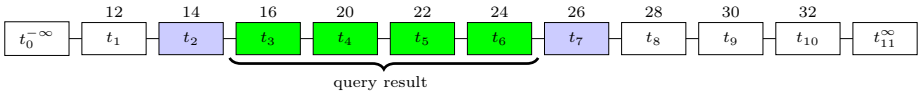


Fig. 3. An example of a chain of tuples of relation `HospitalPatient` in Figure 2 ordered on attribute `PId` (for convenience of the reader the attribute value is reported above each tuple)

signature) each tuple of a relation, and outsources the relation where each tuple is complemented with its signature (e.g., [26]). When the service provider performs a query over the outsourced relation, the service provider returns to the client the query result together with a signature obtained by aggregating the signatures of the tuples in the query result. This aggregate signature forms the verification object of the query result. The aggregate signature can be computed according to different approaches (e.g., condensed RSA [13]). The client can then verify the correctness of the query result by recalculating the single aggregate signature combining the signatures associated with the returned tuples. The computation by the client of the aggregate signature requires a number of operations that is linear in the number of tuples in the query result. This technique permits to assess the correctness of the query result and the non-tampering of each tuple singularly taken but not the completeness of the query result. To provide completeness guarantees to query results, this technique can be extended by constructing an *authenticated chain* over the signatures associated with the tuples in the relation. Intuitively, tuples in the outsourced relation are ordered according to the values of an attribute `a` that is defined over a totally ordered domain. Let $\{t_0^{-\infty}, t_1, \ldots, t_n, t_{n+1}^{\infty}\}$ be the ordered list of tuples in the outsourced relation, with $t_0^{-\infty}$ and t_{n+1}^{∞} two fictitious tuples: $t_0^{-\infty}$ is a left delimiter and t_{n+1}^{∞} is a right delimiter. The owner signs each pair (t_i,t_{i+1}) of tuples, $i = 0, \ldots, n$, and stores the signature of each pair (t_i,t_{i+1}) together with t_{i+1}. The signature, denoted $s(t_{i+1})$, associated with tuple t_{i+1}, $i = 1, \ldots, n+1$, is computed as: $s(t_{i+1})$=$sign(h(t_i) \,||\, h(t_{i+1}), sk)$, where h is a cryptographic hash function, $||$ is the concatenation operator, *sign* is a signature algorithm, and *sk* is the private key of the data owner.

Figure 3 illustrates an example of the chain of tuples built over attribute `PId` of relation `HospitalPatient` in Figure 2. Tuples are first ordered according to the values of attribute `PId`, the left and right delimiters ($t_0^{-\infty}$ and t_{n+1}^{∞}) are added to the chain, and then the data owner computes the signatures as described above. Suppose now that a client submits query "SELECT * FROM `HospitalPatient` WHERE `PId` BETWEEN 15 AND 25". The result of this range query includes all the tuples in the relation with a value for attribute `PId` that falls in the query range [15,25] (i.e., tuples $\{t_3,t_4,t_5,t_6\}$ with a green - dark gray on a b/w printout - background in Figure 3) together with two left and right tuples, that is, the tuple (left) preceding the first tuple satisfying the range query and the tuple (right) following the last tuple satisfying the range query (i.e.,

tuples t_2 and t_7 with a light blue - light gray on a b/w printout - background in Figure 3). The VO includes instead the signature of the tuples between t_3 and t_7 (the signature of tuple t_2 is not needed since this tuple is included in the signature of t_3), and an aggregate signature resulting from the combination of all the signatures of tuples between t_3 and t_7. The client can then verify the correctness and completeness of the query result by: *i)* checking if the set of returned tuples, together with their signatures, form a valid chain; *ii)* if the values for attribute `PId` of the returned tuples are within the query range (in our example, `PId` is between 15 and 25), and *iii)* if the boundary tuples are outside the query range (in our example, 14<15 and 26>25). If the signatures do not form a valid chain, the client can conclude that the query result is not correct or not complete. As an example, suppose that tuple t_4 has been omitted from the query result, meaning that the client receives the sequence t_3, t_5, t_6, t_7. By checking the signature associated with the returned tuple t_5, the client would discover that its value obtained removing the encryption layer of the signature (i.e., $h(t_4)\,||\,h(t_5)$) is different from the one that the client can compute with the returned tuples (i.e., $h(t_3)\,||\,h(t_5)$).

3.2 Tree-based Authenticated Data Structures

Most authenticated tree-based structures are based on variants of the Merkle Hash Tree (Merkle Tree, for short) structure. A Merkle 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 original relation (e.g., h can be a collision-resistant hash function such as SHA-1). The tuples in the leaves of the Merkle Tree are ordered according to the values of an attribute `a`, defined over a totally ordered domain. The internal nodes store the result of the hash function applied over the concatenation of the values stored at their children. In other words, given an internal node n with children n_x and n_y, its hash value h_n is $h(h_{n_x}||h_{n_y})$, where h_{n_x} and h_{n_y} are the hash values of n_x and n_y, respectively, and $||$ is the concatenation operator. The root of the Merkle Tree is signed by the data owner. Figure 4 illustrates an example of a Merkle Tree defined over attribute `DId` of relation `FamilyDoctor` in Figure 2. The hash of the root (i.e., $h_{12345678}$) is signed by the data owner with their private key (i.e., sk).

To verify the correctness and completeness of a point or range query over attribute `a`, the service provider returns to the client (together with the tuples resulting from the evaluation of the query) a VO that includes the values of the nodes in the Merkle Tree needed by the client to compute the hash value associated with the root of the tree. The client then computes the hash value of the root using the VO and the tuples in the query result, and checks whether such a value corresponds to the hash value of the root initially computed (and signed) by the data owner [15]. Note that, while being stored at the service provider, the Merkle Tree cannot be modified by the provider itself, since any update to the structure would imply a change in the hash of the root and hence would invalidate its signature.

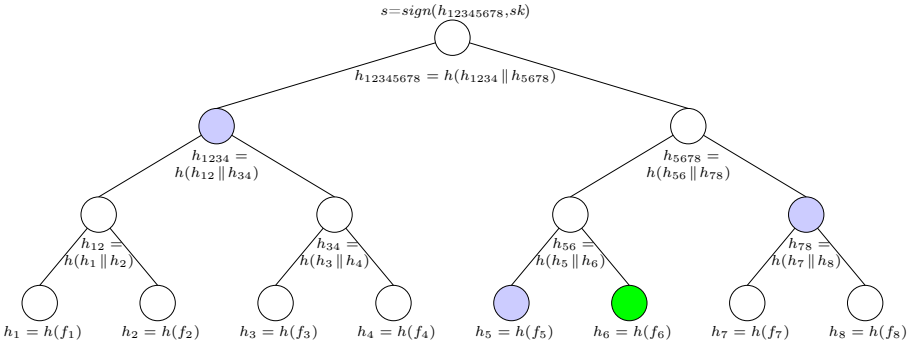


Fig. 4. An example of Merkle Tree built over the tuples of relation `FamilyDoctor` in Figure 2 ordered according to attribute `DId`

The computation of the VO depends on the kind of query evaluated (and hence on the set of tuples returned) and on the integrity guarantee (correctness and/or completeness) to be verified. In case of a point query (i.e., a query with a select condition of the form `a`='v', with v a value in the domain of attribute `a`), the result includes one tuple only, and 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. The use of a collision-resistant hash function for computing the hash values guarantees that it is computationally infeasible for a malicious service provider to return a fake tuple since this would imply the ability of finding a collision in the tree to pass integrity verification. In fact, the root of the Merkle Tree, being signed, should remain unchanged. The cost for verifying the correctness of a point query then corresponds to the computation of $log\ n$ hash values, where n is the number of nodes in the Merkle Tree. As an example of point query, consider relation `FamilyDoctor` in Figure 2 and the Merkle Tree in Figure 4 built over attribute `DId`. Suppose that a client submits query "SELECT * FROM `FamilyDoctor` WHERE `DId`=d60" that returns tuple f_6. To verify the correctness of the query result, tuple f_6 is returned together with a VO that contains the light blue nodes (light gray in b/w printout) in Figure 4. The client can then compute the hash of tuple f_6 (green - dark gray in b/w printout - node) and combine it with the VO, as illustrated in the figure, to compute the hash of the root of the tree, which is then compared to the one computed (and signed) by the data owner. To verify the completeness of a query result, the VO must include the tuples corresponding to the two left and right nodes of the range of the returned tuple(s), together with the values of the nodes needed to compute the hash value of the root. When relying on a Merkle Tree, the size of the VO depends then on the size of the outsourced relation and is independent from both the number of tuples in the query result and from the width of the range covered by the query. Therefore, the approach based on the construction of a signature chain returns a smaller VO than a Merkle Tree for point queries, while

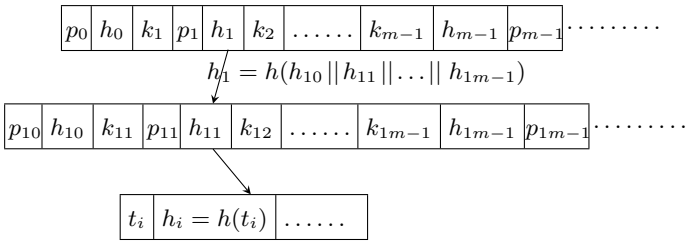


Fig. 5. An example of nodes in a MB-tree with order m

the Merkle Tree returns a smaller VO for range queries covering more than *log* n tuples.

Variations of this basic tree-based integrity verification technique have been proposed to improve the efficiency of the verification process (e.g., [25, 27]) and to support integrity verification of more complex (e.g., join) queries (e.g., [23, 40]). In [23] the authors have proposed the *Merkle B-tree* (MB-tree) structure for supporting the efficient execution and verification of range queries over a single attribute. A MB-tree combines a Merkle Tree with a B+-tree, meaning that the nodes of the B+-tree are extended with a hash value associated with every pointer entry in the node. More precisely, in the leaf nodes, each tuple t is associated with a hash value $h(t)$ computed on the tuple itself. In the internal nodes, each pointer p_i to a child node is associated with hash value $h_i = h(h_{i1} || h_{i2} || \ldots || h_{im-1})$, with $h_{i1}, \ldots, h_{im-1}$ the hash values in the child node pointed by p_i, assuming a B+-tree of order m (i.e., a tree where the number of children of an internal node is at most m). Figure 5 illustrates an example of internal and leaf nodes of a MB-tree with order m. Similarly to Merkle Tree, the data owner signs the concatenation of the hash values stored in the root.

When a client submits a range query, the service provider executes the query over the dataset and returns the set of tuples that satisfy the query together with a VO. The VO is computed by visiting the MB-tree twice, to find the tuples at the left and at the right of the range, respectively. The VO then includes all the information needed to the client to reconstruct all the hash values that appear in the sub-tree whose leaf nodes contain the tuples that satisfy the range query.

Figure 6 illustrates an example of MB-tree that has been built over attribute `PId` of the tuples of relation `HospitalPatient` in Figure 2. Suppose that the client submits query "SELECT * FROM `HospitalPatient` WHERE `PId` BETWEEN 15 AND 25". The service provider returns the query result *res*=$\{t_3, t_4, t_5, t_6\}$ together with the VO that the client uses to verify the completeness and correctness of the query result. In particular, the service provider includes in the VO the tuple on the left (t_2 in our example) and the tuple on the right (t_7 in our example) of those belonging to the query result, respectively. These left and right tuples allow the client to verify the completeness of the query result. The VO must also include all the hash values needed to the client for recomputing the hash values

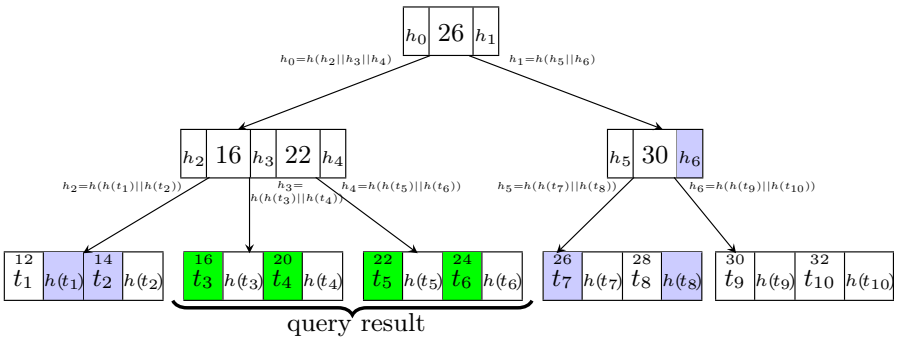


Fig. 6. An example of MB-tree with order $m = 4$ built over attribute `PId` of the tuples in relation `HospitalPatient` of Figure 2

in the root. For the leaf nodes, the service provider includes in the VO the hash values of all the residual tuples (i.e., tuples that are not returned to the client) in the left and right leaves (i.e., the leaves that contain the left and right tuples). In the example, the VO includes $h(t_1)$ and $h(t_8)$. For the internal nodes, the service provider includes in the VO the hash values associated with all the pointers that appear in the nodes visited when searching for the left and right tuples except the hash value of the pointers that are on the right (left) of the pointer traversed when searching for the left (right) tuple. In the example, the service provider includes in the VO hash value h_6. Figure 6 illustrates the elements forming the VO with a light blue (light gray in b/w printout) background.

In [23] the authors compared a MB-tree structure defined over attribute `a`, with the adoption of a B+-tree structure over relation R for attribute `a` combined with a signature chain over `a` for R (Section 3.1). Indeed, both solutions efficiently support the evaluation of range queries and their integrity verification. While the adoption of the MB-tree structure implies that the size of the VO depends on the order of the MB-tree (and can therefore be larger than the VO defined when using a signature chain), the construction of the authenticated data structure is expected to be lower. To combine the advantages of the MB-tree structure with the ones of using a B+-tree with a signature chain over the tuples in the relation, in [23] the authors propose an alternative authenticated data structure (the embedded MB-tree).

3.3 List-based Authenticated Data Structures

Besides tree-based structures, integrity verification approaches can rely on list-based authenticated data structures such as *skip lists* (e.g., [29]). A skip list defined for a set $\mathcal{K}$ of distinct key values includes a set of lists $L_0, L_1, \ldots, L_n$ such that: *i)* L_0 contains all the keys in $\mathcal{K}$ in non-decreasing order, together with special values $-\infty$ and $+\infty$ as first and last element in the list, respectively;

ii) each list $L_i, i = 1, \ldots, n$, contains an arbitrary subset of the keys in L_{i-1}; *iii)* all lists $L_0, L_1, \ldots, L_n$ include values $-\infty$ and $+\infty$. Figure 7(a) illustrates an example of skip list defined over the set $\mathcal{K}$=$\{12, 14, 16, 22, 26\}$ of values for attribute `PId` (i.e., patient identifier) of relation `HospitalPatient` in Figure 2. This skip list includes three lists L_0, L_1, and L_2.

Skip lists have been designed in such a way to efficiently support search operations. The search for a key value v starts from $-\infty$ in the top list (i.e., L_n), and proceeds through the application of two operations: *hop forward*, and *drop down*. Hop forward means that the search proceeds right along the current list until the visited key value v_i is the largest value lower than or equal to v. Then, the search moves down of one step (i.e., from the current list L_j to L_{j-1}). The search iteratively applies the hops forward and drops down operations until it reaches the bottom list L_0. Figure 7(b) illustrates an example of the search process for value 22 in the skip list in Figure 7(a). In the figure, visited nodes are denoted with bold lines, while bold arrows denote hop forward (horizontal) and drop down (vertical) operations.

Skip lists can be efficiently used to verify the integrity of point queries defined over an attribute with actual domain corresponding to $\mathcal{K}$. In this case, each node of the skip list defined over $\mathcal{K}$ is enriched with a *label*, computed through a *commutative* and *collision-resistant* hash function (i.e., a hash function h such that $h(x, y) = h(y, x)$ and such that its application to different inputs always returns different values). Given a node v of the skip list and the node w at its right, the label for v is computed as follows for L_0:

$$\ell(v, L_0) = \begin{cases} h(v, w) & \text{if } w \in L_1 \\ h(v, \ell(w, L_0)) & \text{otherwise} \end{cases} \quad (1)$$

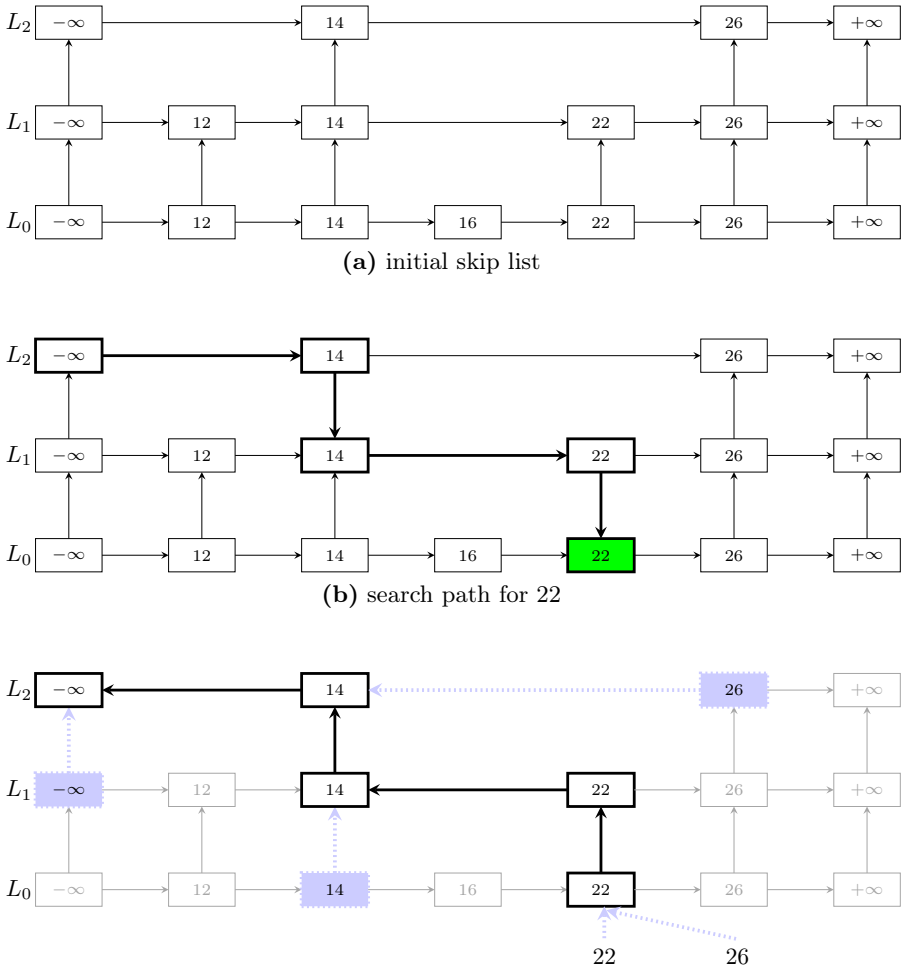
and as follows for L_i with $i = 1 \ldots n$:

$$\ell(v, L_i) = \begin{cases} \ell(v, L_{i-1}) & \text{if } w \in L_{i+1} \\ h(\ell(v, L_{i-1}), \ell(w, L_i)) & \text{otherwise} \end{cases} \quad (2)$$

For instance, with respect to the skip list in Figure 7(a), the label of node 22 on L_0 is computed as $\ell(22, L_0) = h(22, 26)$ and the label of node 14 on L_0 is computed as $\ell(14, L_0) = h(14, \ell(16, L_0))$. The label of node 12 on L_1 is instead computed as $\ell(12, L_1) = \ell(12, L_0)$ and the label of node 14 on L_1 is computed as $\ell(14, L_1) = h(\ell(14, L_0), \ell(22, L_1))$.

The label of the first node in the top list (i.e., L_n) is signed by the data owner and used for integrity check.

The verification of the integrity of a point query targeting value v consists in checking whether v is included in the skip list. In particular, if v belongs to the skip list, the integrity verification process verifies its presence. Otherwise, it verifies the existence of two values v' and v'', consecutive in list L_0, such that $v' < v < v''$. The verification process uses the information in the VO, which includes: the signed label of the starting node of the skip list and the labels of the nodes on the right and below the nodes in the path visited to reach v,



(a) initial skip list

(b) search path for 22

(c) verification

Fig. 7. A skip list for set $\mathcal{K}=\{12, 14, 16, 22, 26\}$ of keys with three lists (a), search process for value 22 (b), and verification object for a point query searching value 22 (c)

which are necessary to recompute the label of the starting node of the skip list. If such recomputed value corresponds to the value signed by the data owner, the verification process succeeds. For instance, consider the skip list in Figure 7(a) and a query targeting value 22. Figure 7(c) highlights the visited nodes with bold lines and the nodes included in the verification object with a light blue (light gray in b/w printout) background and with dashed lines. The verification object then corresponds to the list $\langle 22,\ 26,\ \ell(14, L_0),\ \ell(-\infty, L_1),\ \ell(26, L_2)\rangle$. The client formulating the query verifies the query result by hashing the values in

the verification object and comparing the result with the label of the starting node of the skip list.

The main advantage of skip lists over tree-based structures is that they can be efficiently managed by a relational database with a limited overhead at the client-side for integrity verification [16]. Such technique, however, permits the verification of the correctness and completeness of point queries only.

4 Probabilistic Approaches

Probabilistic techniques offer a probabilistic guarantee on the integrity of query results, that is, there is a (typically low) probability that an integrity violation (e.g., omission of tuples in the query result) goes undetected. Probabilistic integrity verification approaches mainly focus on providing correctness and completeness guarantees and most of them are based on the injection in the original dataset of non genuine data (called *sentinels* or *markers*) [19, 34–36], or the controlled replication of data (*twins*) [10, 32, 33]. The offered integrity guarantees are probabilistic because, while the omission of an expected sentinel or replica signals an integrity violation, its presence in the computation of the query result does not imply the integrity of the result. As a matter of fact, the service provider might have just been lucky in not missing any of the sentinels and/or twins inserted by the data owner. We now provide a more detailed description of the use of sentinels and twins for integrity verification (Section 4.1), and then describe their use for the verification of join queries (Section 4.2).

4.1 Probabilistic Integrity Controls

We describe two probabilistic integrity verification approaches (i.e., sentinels and twins) and their adoption for controlling the correctness and completeness of query results.

Sentinels. Sentinels are fake tuples inserted in a dataset before using it in a computation, and are built in such way to be indistinguishable from original data to the providers involved in the computation (e.g., [8, 37]). The insertion of a set S of sentinels is driven by the client requesting the execution of a query. Given a dataset D and a query q, the service provider executes the query over $D \cup S$ and returns to the client the query result *res*. The correctness of the query result is verified by the client by checking whether, for each sentinel in the query result *res*, the result of the evaluation of query q over the sentinel is the expected one. Absence in the query result *res* of one or more of the expected tuples (i.e., the tuples obtained by the execution of q over S, denoted $q(S)$) signals instead the fact that the query result is not complete. Otherwise, the query result is considered complete with a certain probability. As proved in [36], even a limited number of sentinels ensures high probabilistic guarantees of completeness of the query result. The main drawback of the use of sentinels is that the client should store the set S of sentinels injected in the dataset to compare the query result received from the service provider with the evaluation of the query q

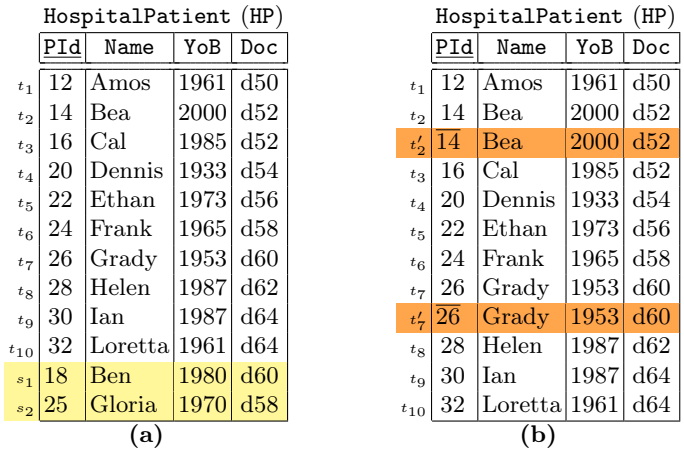


Fig. 8. Relation `HospitalPatient` in Figure 2 enriched with sentinels (a) and twins (b)

over S. To avoid this drawback, some approaches (e.g., [36]) are based on the use of deterministic functions for sentinels generation. A client can then have knowledge and check sentinels without storing them. Another problem is related to the generation of sentinels, which should be: *i)* indistinguishable from the original tuples; and *ii)* generated to cover in a uniform way the domains of the attributes in the dataset, to maximize the probability that any query hits at least a sentinel. The approaches in [8, 37] use encryption to prevent the service provider from distinguishing between sentinels and real tuples. Other approaches use specific functions that generate uniformly distributed sentinels (e.g., [19]). As an alternative, if the provider storing the data is assumed to be trusted, sentinels can be generated and injected on the fly, before sending the dataset to the provider in charge of query evaluation. In this case, sentinels can be generated in such a way that they belong to the query result. As an example of the use of sentinels, consider relation `HospitalPatient` in Figure 2 and suppose to inject two sentinels (s_1 and s_2) in the relation as illustrated in Figure 8(a), where sentinels have a yellow (light gray in b/w printout) background. Assume that a client submits query "SELECT $*$ FROM `HospitalPatient` WHERE `YoB`$\leq$1970", asking for all patients born in 1970 or before. The client would expect sentinel s_2 to belong to the query result, that is, res=$\{t_1, t_4, t_6, t_7, t_{10}, s_2\}$. Absence of s_2 from the query result signals the incompleteness of the result. We note, however, that a result including tuples t_4, t_6, t_7, t_{10}, and s_2, while not complete (it misses t_1) would not violate the integrity check over sentinels.

Controlled replication (twins). Controlled replication consists in replicating the tuples in the relation(s) involved in a query that satisfy a *replication condition* C_r. The service provider involved in the computation should not be able to identify pairs of replicated tuples. To verify the completeness and correctness

of the query result, the client checks the presence of two identical copies for each tuple in the query result that satisfies the replication condition C_r. The presence of one copy only signals the incompleteness of the query result. Also receiving, for twin tuples, inconsistent results signals an incorrect result (i.e., at least one of the two results is incorrect). As an example, consider relation `HospitalPatient` in Figure 2 and assume to replicate tuples with `PId` equal to 14 or 26 (i.e., tuples t_2 and t_7), as illustrated in Figure 8(b) where twins have a orange (dark gray in b/w printout) background. Note that, for simplicity, in the example we did not change twin tuples t'_2 and t'_7 to make them indistinguishable from the corresponding twins (t_2 and t_7, respectively). Suppose that a client submits query "SELECT $*$ FROM `HospitalPatient` WHERE `Name`='Bea' ". The query result should include both t_2 and t'_2. Absence of one of these tuples from the query result signals its incompleteness. We note, however, that an empty result would pass integrity verification.

Complementary adoption of sentinels and twins. Sentinels and twins have been proposed independently, and can be applied in isolation (i.e., either sentinel or twins can be adopted to verify the integrity of a query result) or can be applied in combination (e.g., [7–9]). Indeed, the complementary nature of sentinels and twins ensures that their combined adoption provides stronger integrity guarantees. As observed in [8, 11] twins are twice as effective as sentinels in detecting omissions since the absence of any of the tuples in a twin pair signals an integrity violation. However, twins lose effectiveness when the service provider omits a large fraction of the tuples in the query result: the greater the number of omitted tuples in the query result the more likely it is for the service provider to omit twins in pairs and therefore to have the omission undetected (e.g., the omission of all the tuples in the join result would pass the integrity check based on twins only). This is then where sentinels come into help. In fact, when the number of tuples omitted from the query result increases, the probability for the service provider to be undetected with respect to the sentinel control decreases. This is due to the fact that the greater the number of omissions, the greater the probability of omitting a sentinel. The combined use of a limited number of sentinels together with a limited number of twins ensures then complementarity of controls and stronger integrity guarantees, as formally illustrated by the analysis in [11].

4.2 Probabilistic Guarantees of Join Queries

In the discussion so far we have described how existing techniques can be adopted for assessing the integrity of point and range queries. Often, however, data coming from multiple data owners need to be combined (joined) to extract useful information. The verification of join queries is more complex than the verification of point/range queries. In the following, we illustrate how sentinels and twins can be also adopted for assessing the integrity of the result of join queries. In the discussion, we consider a client that wants to execute a join query over two relations, denoted R_l and R_r, stored at two independent trusted service providers, S_l and

S_r, respectively. We distinguish service providers that offer storage resources and that manage the relations on which the queries are performed, from *computational providers* that offer computational resources. We assume that the service providers storing the relations are trustworthy while computational providers are not. The execution of joins can be delegated to computational providers because the adoption of the service providers for performing the joins (which are expensive operations) might not be the most economically viable solution or because they might not want to use their network and computational resources for performing queries on behalf of the client. In the following, we describe how to perform one-to-one or one-to-many join operations with sentinels and twins and with the help of a computational provider.

One-to-one join. Correctness and completeness of the result of join queries performed over R_l and R_r is provided through the coordinated insertion of sentinels and twins in the relations before transmitting them to the computational provider. To have the guarantee that sentinels and twins also belong to the join result, the insertion of such checks is driven and coordinated by the client submitting the query. More precisely, the client determines the number of sentinels to be inserted in the two relations and their values for the join attribute, as well as the replication condition regulating the percentage of twins to be inserted in R_l and R_r. To avoid spurious tuples in the join result, the values of the join attribute for sentinels are chosen outside the domain of the original join attribute values. The twinning condition is defined over the join attribute, because it is the only attribute common between the two relations. The values of the join attribute for twinned tuples are combined with a random nonce. In this way, twinned tuples do not join with the original ones. The relations to be joined are encrypted before sending them to the untrusted computational provider. Note that to allow the computational provider to perform the join operation, the join attribute in the two relations R_l and R_r is encrypted separately using the same deterministic encryption algorithm and the same key. The encryption of the relations guarantees the confidentiality of the data and ensures the indistinguishability of sentinels and twins from the original tuples. The computational provider performs the join operation over the encrypted relations and returns the query result to the client. The client checks the integrity of the query result by analyzing sentinels and twins: an integrity violation is detected if a sentinel is missing or a tuple satisfying the twinning condition appears solo.

Figure 9 illustrates an example of one-to-one join query "SELECT `DId`,`Name` FROM `FamilyDoctor` JOIN `HospitalPatient` ON `DId`=`Doc`" retrieving the identifier and name of the family doctors with a patient in the hospital. For simplicity, the figure shows a simplified version of the schema of the relations in Figure 2, and a simplified version of the instances that produces a one-to-one join. Also, we use the abbreviations `FD` and `HP` as name of the relations. Service providers $\mathbb{M}$ and $\mathbb{H}$, storing the two relations, first inject both sentinels and twins (which are the tuples that satisfy replication condition `DId`=d52 or `Doc`=d52, respectively). The resulting extended relations (`FD`* and `HP`*) are then encrypted on-the-fly and sent to the computational provider (in the figure, encrypted values are represented

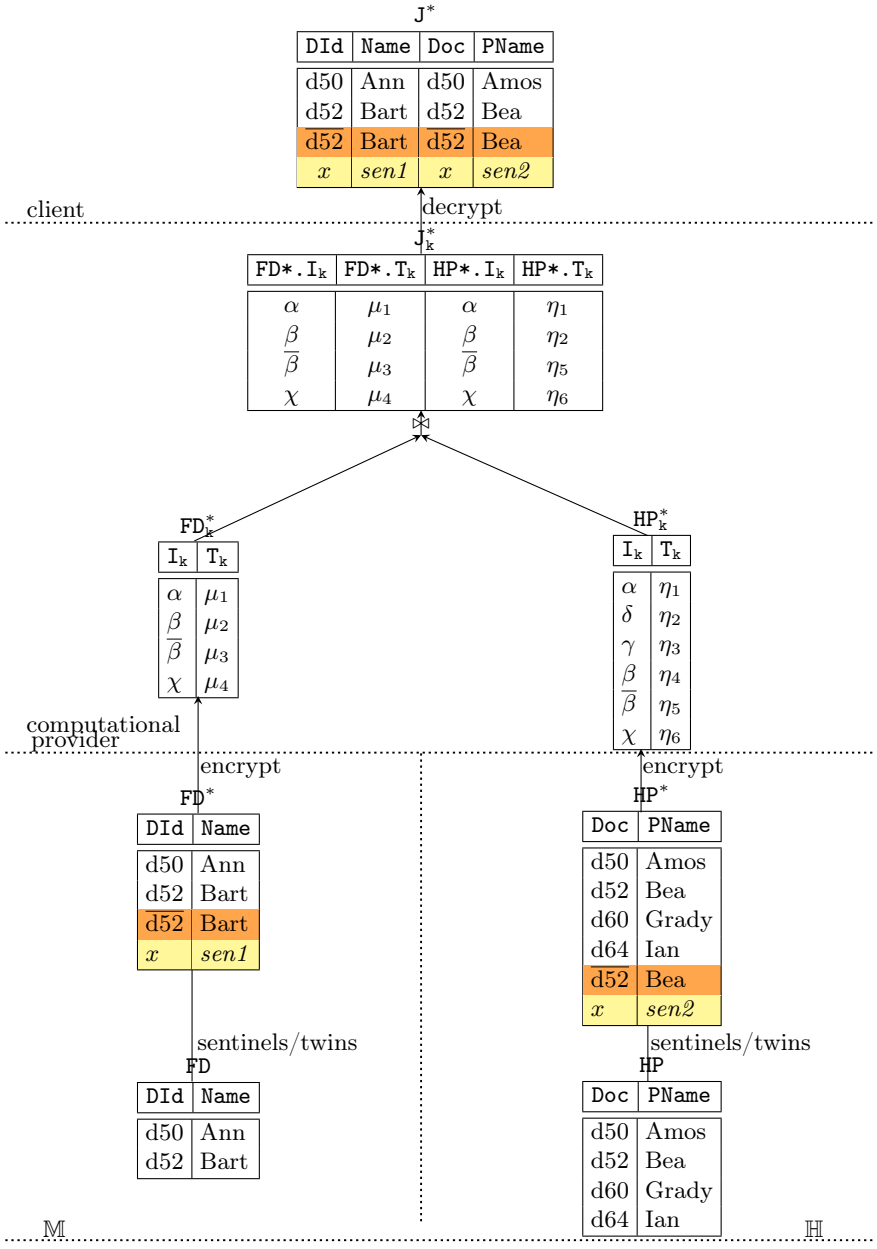


Fig. 9. An example of evaluation of a one-to-one join query with twins (orange - dark gray on a b/w printout - tuples) on 'd52' and one sentinel (yellow - light gray on a b/w printout - tuple)

as Greek letters). The encrypted relations $\texttt{FD}_k^*$ and $\texttt{HP}_k^*$ have two attributes: $\texttt{I}_\texttt{k}$, the encrypted join attribute; and $\texttt{T}_\texttt{k}$, the encryption of all the attributes in the original relation (including the join attribute). The computational provider computes the natural join between the received encrypted relations and sends the result ($\texttt{J}_\texttt{k}^*$) to the client. The client decrypts $\texttt{J}_\texttt{k}^*$ ($\texttt{J}^*$), verifies its completeness (i.e., if all the expected sentinels and twins are in $\texttt{J}^*$) and correctness, and, if no omission is detected, projects over attributes `DId` and `Name`, and removes twins and sentinels to obtain the final join result `J` to be returned to the client.

One-to-many join. The correctness and completeness of one-to-many joins can be verified combining sentinels and twins as illustrated for one-to-one join operations. However, when the join between relation R_l and relation R_r is a one-to-many join (we assume R_l to be the relation on side "one" and R_r to be the relation on side "many"), the frequencies of the values of the join attribute remain visible to the computational provider, thus possibly making twins and sentinels recognizable. Indeed, sentinels are all distinct and therefore multiple tuples with the same value for the join attribute cannot be sentinels. Also, the uncertainty on twin tuples can be reduced since twin pairs are characterized by the same number of occurrences for the join attribute. As a simple example, suppose that the computational provider receives a relation R_r where there are 8 tuples with 4 distinct values for the join attribute: one value occurs 3 times (3 tuples), two values occur twice (4 tuples), and one value occurs once (1 tuple). The computational provider can immediately infer that the three tuples with the same value for the join attribute cannot be sentinels, since sentinels have distinct values for the join attribute, and cannot be twins since twins are always in pairs and no other value in R_r has three occurrences. The two pairs of tuples with two occurrences each can be twins or can be genuine tuples. The only tuple with one occurrence can be a sentinel or a genuine tuple. The frequency distribution of the values of the join attribute can then compromise the indistinguishability property of twins and sentinels, and therefore it should not be revealed to the computational provider.

The approach in [8] flattens the frequency distribution of values of the join attribute of the tuples participating in a one-to-many join by using *salts*, *buckets*, or a combination of salts and buckets. Intuitively, salts aim at transforming a one-to-many join into an equivalent one-to-one join. The different occurrences of a same value for the join attribute in relation R_r are made different by combining each occurrence with a different random salt. To enable the correct evaluation of the join operation, each value of the join attribute in relation R_l is replicated as many times as the maximum expected number of occurrences of a value in R_r, and each replica is combined with a different random salt. Clearly, the values for salts used by the two service providers S_l and S_r must be the same, therefore their generation is coordinated. Buckets aim instead at guaranteeing a flat frequency distribution of the join attribute values in R_r. The idea is to make the number of occurrences of all values of the join attribute in R_r equal to the number of occurrences of the join value that appears more frequently in the relation. For the join attribute values with a number of occurrences smaller

than the number of occurrences of the most frequent value, dummy tuples (i.e., tuples with the same value for the join attribute and a dummy content) are then inserted in the relation. Salts and buckets can also be used in combination to limit the increase of the size of relation R_l when using salts due to the replication of salted tuples, and the increase of the size of relation R_r and of the join result when using buckets due to the addition of dummy tuples in R_r. Note that salts and/or buckets operate on original as well as on control (i.e., sentinels and twins) tuples, to guarantee their indistinguishability. The number of salts and the size of buckets need to be coordinated among the client and the service providers, and the number of salts in particular need to be known to both S_l and S_r. The client is therefore in charge of choosing and communicating the number of salts to be used. The size of buckets can be autonomously computed by S_r based on the maximum frequency in relation R_r (i.e., by dividing such a frequency by the number of available salts).

Consider, as an example, the evaluation of the one-to-many join query "SELECT `DId`,`Name` FROM `FamilyDoctor` JOIN `HospitalPatient` ON `DId`=`Doc`" in Figure 10, where each provider $\mathbb{H}$ and $\mathbb{M}$ injects in its relation one sentinel and twins tuples that satisfy the replication condition `DId`=d52 or `Doc`=d52. Like for the one-to-one join, the figure reports a simplified version of the schema of the relations in Figure 2. We assume that the client sets the number of salts to 2. Buckets will have size 2, since the most frequent value in `HospitalPatient` (i.e., value d60) has 3 occurrences ($\lceil \frac{3}{2} \rceil = 2$). The 3 tuples with value d60 are then split in two buckets, one of which includes a dummy tuple. Similarly, sentinel x is included in a bucket with a dummy tuple, since by definition sentinels have one occurrence only. Figure 10 illustrates the evaluation of the join, which proceeds as already discussed for the one-to-one join in Figure 9, with the addition of salts and buckets.

5 Open Issues

Although the problem of query integrity verification has been widely studied, there are still several interesting research directions that need to be further investigated, as summarized in the following.

- *Type of outsourced computation and data.* Existing integrity verification techniques mainly consider SQL queries (e.g., point, range, aggregate, and join queries), location-based range queries, and top-k queries as outsourced computations, and work on relational databases. An interesting research direction would then be the investigation of integrity verification techniques for other kinds of computations, (e.g., data mining, machine learning, data classification, and clustering) as well as for other kinds of data. Very few approaches have addressed the problem of verifying the integrity of queries formulated, for example, over spatial data or graph data (e.g., [18, 31]). These solutions are based on a variation of the Merkle hash tree.
- *Distributed platforms.* The growing interest toward the use of distributed platforms (e.g., distributed cloud storage platforms) for processing large vol-

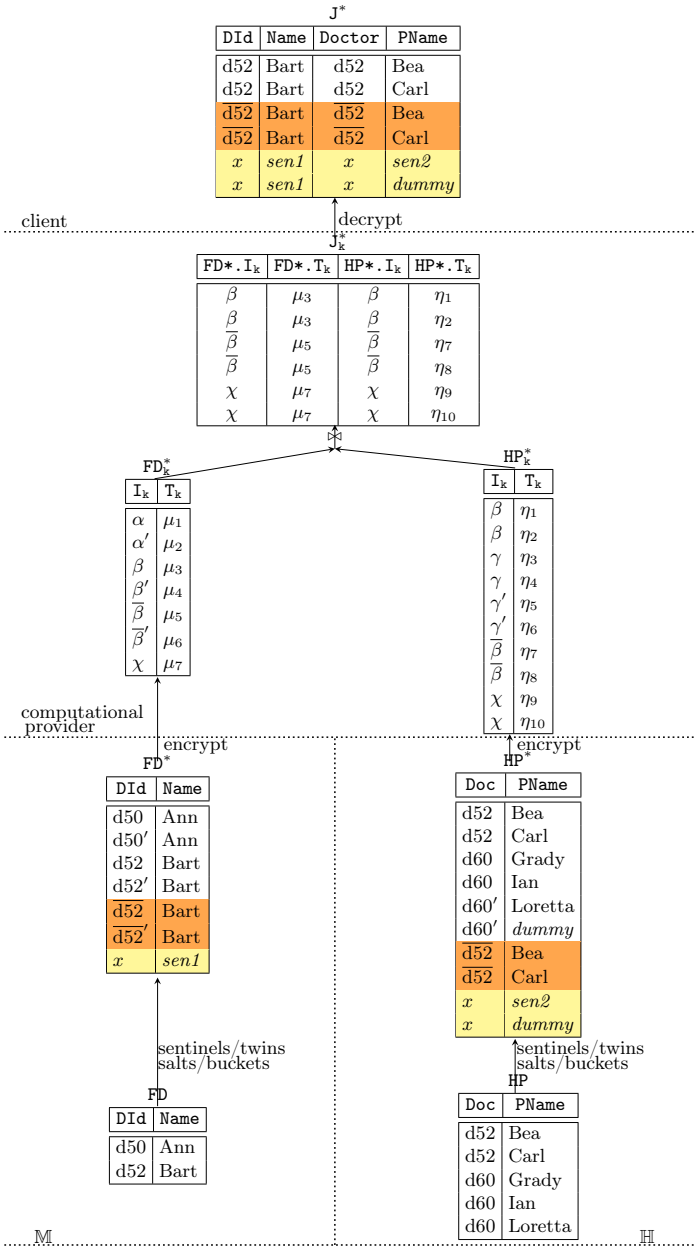


Fig. 10. An example of evaluation of a one-to-many join query with twins (orange - dark gray on a b/w printout - tuples) on 'd52', one sentinel (yellow - light gray on a b/w printout - tuple), two salts, and bucket of size two

umes of data goes along with the growing interest for designing integrity verification techniques that are able to verify the behavior of all the parties involved in the computation (e.g., [7]). These platforms are typically characterized by the presence of independent *workers* that collaboratively perform a computation. There are several aspects that need to be studied to efficiently and effectively apply the integrity verification techniques in such distributed context, including: *i)* the need to verify the behavior of all the workers; *ii)* the consideration of different trust assumptions on the workers involved in the computation; and *iii)* the need to limit the overhead due to coordination of integrity verification. The presence of different workers also introduces the problem of collusion. As a matter of fact, the working of distributed platforms is typically based on the assumption that workers are independent and do not communicate. It would then be interesting to investigate what can happen when such independency cannot be assumed and some workers, under the control of a same subject, can communicate and collude to go undetected in their omissions. The design of probabilistic integrity verification techniques for distributed scenarios also requires the definition of a model for capturing the distribution among workers of sentinels and twins, their combined use, and their generation so to provide best effectiveness for integrity guarantees [11].

- *Freshness.* Most of the existing integrity verification techniques mainly focus on the correctness and completeness of query results. Only few proposals also address the problem of verifying the freshness of query results (e.g., [28, 38]). An interesting research direction would then be the design of efficient integrity verification techniques (especially probabilistic) able to assess at the same time the correctness, completeness, and freshness of query results.
- *Combined application of integrity verification and other security approaches.* While integrity verification techniques might work well in isolation, their combined application with other approaches for protecting data/computations may open the door to new vulnerabilities that need to be addressed. As an example, the problem of combining integrity and query privacy in its different aspects (e.g., protection of the data, protection of single queries, and protection of query patterns) requires a careful analysis, investigating approaches that can balance the trade-off between protection enjoyed and performance overhead paid, allowing clients to tune the protection guarantees and overhead in different contexts and scenarios. A further example is represented by the combination of integrity and access control. In fact, existing query integrity verification techniques are typically based on the assumption that the client is authorized to access the whole outsourced datasets. Such an assumption, however, does not fit real world applications, which demand for selective access by different users (e.g., [39]).

6 Conclusions

The outsourcing of query computations brings several advantages but, at the same time, requires solutions that should consider not only the protection of

data confidentiality, but also the need of verifying the integrity of query results. Integrity is particularly important when the service providers involved in a query evaluation are not trustworthy, that is, they are not reliable for properly performing queries. In this chapter, we presented an overview of the main integrity verification techniques. We described both deterministic and probabilistic techniques, which differ on the kind of guarantees provided. For each category, we illustrated the main techniques, highlighting the verification cost. We concluded the chapter with a discussion on possible open research directions.

Acknowledgements This work was supported in part by the EC under projects EdgeAI (101097300) and GLACIATION (101070141), by the Italian MUR under PRIN project POLAR (2022LA8XBH), and by project SERICS (PE00000014) under the MUR NRRP funded by the EU - NGEU.

References

1. 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 (January 2005)
2. Albanese, M., Jajodia, S., Jhawar, R., Piuri, V.: Securing mission-centric operations in the cloud. In: Jajodia, S., Kant, K., Samarati, P., Swarup, V., Wang, C. (eds.) Secure Cloud Computing, pp. 239–260. Springer, New York (2014)
3. 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 (October/November 2007)
4. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. In: Proc. of EUROCRYPT. Warsaw, Poland (May 2003)
5. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Combining fragmentation and encryption to protect privacy in data storage. ACM TISSEC **13**(3), 22:1–22:33 (July 2010)
6. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: Managing and sharing servents' reputations in P2P systems. IEEE TKDE **15**(4), 840–854 (July/August 2003)
7. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G., Paraboschi, S., Samarati, P.: Integrity for distributed queries. In: Proc. of IEEE CNS. San Francisco, CA, USA (October 2014)
8. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Integrity for join queries in the cloud. IEEE TCC **1**(2), 187–200 (July-December 2013)
9. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Optimizing integrity checks for join queries in the cloud. In: Proc. of DBSec. Vienna, Austria (July 2014)
10. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Efficient integrity checks for join queries in the cloud. JCS **24**(3), 347–378 (2016)

11. 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 **34**(1), 108–122 (January 2023)
12. De Capitani di Vimercati, S., Foresti, S., Livraga, G., Paraboschi, S., Samarati, P.: Privacy in pervasive systems: Social and legal aspects and technical solutions. In: Colace, F., Santo, M.D., Moscato, V., Picariello, A., Schreiber, F., Tanca, L. (eds.) Data Management in Pervasive Systems, pp. 43–65. Springer, USA (2015)
13. 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. (eds.) The New Codebreakers. Springer, USA (2016)
14. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Protecting data and queries in cloud-based scenarios. SN Computer Science **4**(5) (September 2023)
15. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.: Authentic third-party data publication. In: Proc. of DBSec. Schoorl, The Netherlands (August 2000)
16. Di Battista, G., Palazzi, B.: Authenticated relational tables and authenticated skip lists. In: Proc. of DBSec. Redondo Beach, CA, USA (July 2007)
17. Donida Labati, R., Genovese, A., Piuri, V., Scotti, F., Vishwakarma, S.: Computational intelligence in cloud computing. In: Kovács, L., Haidegger, T., Szakál, A. (eds.) Recent Advances in Intelligent Engineering, Topics in Intelligent Engineering and Informatics, vol. 14, pp. 111–127. Springer, Cham, USA (2020)
18. Fan, Z., Peng, Y., Choi, B., Xu, J., Bhowmick, S.: Towards efficient authenticated subgraph query service in outsourced graph database. IEEE TSC **7**(4), 696—-713 (October-December 2014)
19. Ghazizadeh, P., Mukkamala, R., Olariu, S.: Data integrity evaluation in cloud database-as-a-service. In: Proc. of IEEE SERVICES. Santa Clara, CA, USA (June-July 2013)
20. Apache hadoop, http://hadoop.apache.org/
21. Jhawar, R., Piuri, V.: Fault tolerance and resilience in cloud computing environments. In: Vacca, J. (ed.) Computer and Information Security Handbook, 2nd Edition, pp. 125–141. Morgan Kaufmann, USA (2013), 978-0-1239-4397-2
22. Juels, A., Kaliski, B.: PORs: Proofs of retrievability for large files. In: Proc. of ACM CCS. Alexandria, VA, USA (October–November 2007)
23. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: Proc. of SIGMOD. Chicago, IL, USA (June 2006)
24. Li, X., Weng, C., Xu, Y., Wang, X., Rogers, J.: ZKSQL: Verifiable and efficient query evaluation with zero-knowledge proofs. PVLDB **16**(8), 1804–1816 (2023)
25. Mouratidis, K., Sacharidis, D., Pang, H.: Partially materialized digest scheme: An efficient verification method for outsourced databases. The VLDB Journal **18**, 363–381 (2009)
26. Pang, H., Jain, A., Ramamritham, K., Tan, K.: Verifying completeness of relational query results in data publishing. In: Proc. of ACM SIGMOD 2005. Baltimore, MD, USA (June 2005)
27. Pang, H., Tan, K.: Authenticating query results in edge computing. In: Proc. of ICDE. Boston, MA, USA (April 2004)
28. Pang, H., Zhang, J., Mouratidis, K.: Scalable verification for outsourced dynamic databases. PVLDB **2**(1), 802–813 (2009)
29. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of ACM **33**(6), 668—-676 (1990)

30. Samarati, P., De Capitani di Vimercati, S.: Cloud security: Issues and concerns. In: Murugesan, S., Bojanova, I. (eds.) Encyclopedia on Cloud Computing. Wiley, USA (2016)
31. Tian, F., Wu, Z., Gui, X., Ni, J., Shen, X.: Fine-grained query authorization with integrity verification over encrypted spatial data in cloud storage. IEEE TCC **10**(3), 1831–1847 (July-September 2022)
32. Ulusoy, H., Kantarcioglu, M., Pattuk, E.: TrustMR: Computation integrity assurance system for MapReduce. In: Proc. of BigData. Santa Clara, CA, USA (Oct-Nov 2015)
33. Wand, H., Yin, J., Perng, C.S., Yu, P.: Dual encryption for query integrity assurance. In: Proc of ACM CIKM. Napa Valley, CA, USA (October 2008)
34. Wang, C., Chow, S., Wang, Q., Ren, K., Lou, W.: Privacy-preserving public auditing for secure cloud storage. IEEE TC **62**(2), 362–375 (Feb 2013)
35. Wong, W., Cheung, D., Kao, B., Hung, E., Mamoulis, N.: An audit environment for outsourcing of frequent itemset mining. PVLDB **2**(1), 1162–1172 (2009)
36. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity auditing of outsourced data. In: Proc. of VLDB. Vienna, Austria (September 2007)
37. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity auditing of outsourced data. In: Proc. of VLDB. Vienna, Austria (September 2007)
38. Xie, M., Wang, H., Yin, J., Meng, X.: Providing freshness guarantees for outsourced databases. In: Proc. of EDBT. Nantes, France (March 2008)
39. Xu, C., Xu, J., Hu, H., Au, M.: When query authentication meets fine-grained access control: A zero-knowledge approach. In: Proc. of SIGMOD. Houston, TX, USA (June 2008)
40. Yang, Y., Papadias, D., Papadopoulos, S., Kalnis, P.: Authenticated join processing in outsourced databases. In: Proc. of SIGMOD. Providence, RI, USA (June-July 2009)
41. Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proc. of HotCloud. Boston, MA, USA (June 2010)
42. Zheng, Q., Xu, S., Ateniese, G.: Efficient query integrity for outsourced dynamic databases. In: Proc. of ACM CCSW. Raleigh, NC, USA (October 2012)
43. Zhou, W., Cai, Y., Peng, Y., Wang, S., Ma, K., Li, F.: VeriDB: An SGX-based verifiable database. In: Proc. of SIGMOD (June 2021)