# Shuffle Index: Efficient and Private Access to Outsourced Data

SABRINA DE CAPITANI DI VIMERCATI, Università degli Studi di Milano
SARA FORESTI, Università degli Studi di Milano
STEFANO PARABOSCHI, Università degli Studi di Bergamo
GERARDO PELOSI, Politecnico di Milano
PIERANGELA SAMARATI, Università degli Studi di Milano

Author's address: Sabrina De Capitani di Vimercati, Sara Foresti, Pierangela Samarati, Dipartimento di Informatica - Università degli Studi di Milano, 26013 Crema, Italy; Gerardo Pelosi, Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano, 20133 Milano, Italy; Stefano Paraboschi, Dipartimento di Ingegneria - Università degli Studi di Bergamo, 24044 Dalmine, Italy. This paper extends the previous work by the authors appeared under the title "Efficient and Private Access to Outsourced Data," in Proc. of the 31st International Conference on Distributed Computing Systems (ICDCS 2011), June 2011, Minneapolis, Minnesota, USA [De Capitani di Vimercati et al. 2011a].

## 1. INTRODUCTION

The research and industrial communities have been recently showing considerable interest in the outsourcing of data and computation. The motivations for this trend come from the economics of system administration, which present large scale economies, and by the evolution of ICT (Information and Communications Technology), which offers universal network connectivity that makes it convenient for users owning multiple devices to store personal data at an external server. A major obstacle toward the large adoption of outsourcing, otherwise particularly attractive to individuals and to small/medium organizations, is the perception of insecurity and potential loss of control on sensitive data and the exposure to privacy breaches. Guaranteeing privacy in a context where data are externally outsourced entails protecting the confidentiality of the data as well as of the accesses to them. In particular, it requires to maintain confidentiality on: the data being outsourced (*content confidentiality*); the fact that an access aims at a specific piece of information (*access confidentiality*); the fact that two accesses aim at the same target (*pattern confidentiality*).

Several solutions have been proposed in the past few years, both in the theoretical and in the system communities, for guaranteeing data availability (e.g., [Bowers et al. 2009; Bessani et al. 2011]) and for protecting the confidentiality of the outsourced data. Typically, solutions focusing on data confidentiality (e.g., [Damiani et al. 2003; Hacigümüs et al. 2002b]) consider an honest-but-curious server (i.e., a server trusted to provide the required storage and management services but not authorized to read the actual data content) and resort to encryption to protect the outsourced data. Since the server is not allowed to decrypt the data for access execution, these solutions provide different techniques for elaborating queries on encrypted data. Furthermore, they aim at content confidentiality but do not address the problem of access and pattern confidentiality.

Access and pattern confidentiality have been traditionally addressed within a different line of work by *Private Information Retrieval* (PIR) proposals (e.g., [Ostrovsky and Skeith, III 2007; Sion and Carbunar 2007]), which provide protocols for querying a database that prevent the storage server from inferring which tuples are being accessed. PIR approaches typically work on a different problem setting. As a matter of fact, in most proposals, the external database being accessed is in plaintext (i.e., content confidentiality is not an issue). Regardless of whether the external database is plaintext or encrypted, PIR solutions have high computational complexity and are therefore not applicable to real systems. It has been proved [Sion and Carbunar 2007] that the execution of information-theoretic PIR protocols requires more resources than those required for a complete transfer of the database from the server to the client. Recent solutions, based on a careful adaptation of the Oblivious RAM data structure (e.g., [Stefanov and Shi 2013; Stefanov et al. 2013; Williams et al. 2012]), protect access and pattern confidentiality at a reduced cost with respect to PIR. These solutions however imply a still relevant computational overhead, compared to the adoption of non-privacy preserving access structures.

In this paper, we aim at providing a novel efficient approach addressing the different aspects of the privacy problem. We consider a reference scenario where a *data owner* outsources data to an external honest-but-curious server, and accesses her data by submitting requests to a *client* that directly interacts with the server. Our goal is to enable the owner to efficiently access the outsourced data, either to search for a value (or a set thereof) or to update the outsourced data collection. The access protocol should not reveal to any observer, including the server itself, which kind of access is being executed (i.e., read or update) and should guarantee content, access, and pattern confidentiality. In [De Capitani di Vimercati et al. 2011a] we presented an early ver-

sion of our proposal, which here is extended with the support of range queries and of updates to the outsourced data collection as well as with more extensive analysis and experimental results (see Section 11 for more details).

We propose a novel data structure, called *shuffle index*, with which the data to be outsourced are organized (Section 3). The design of the shuffle index data structure uses as a foundation the well-known and carefully investigated $B+$-tree, arguably today the structure most commonly used to efficiently store and manage large amounts of data. $B+$-trees are preferred to hash structures thanks to the support of a total order relationship among the values of the index key. Our shuffle index assumes data to be organized in a $B+$-tree with no link between leaves and applies node-level encryption to protect actual data from the external storage server. In the working of the system, the client can hide the actual request within cover (fake) requests, cache nodes, and shuffle the content among blocks stored at the server. In this way, no observer, including the server itself, can reconstruct the association between blocks read and actually accessed data (Section 4). Our solution combines cover, caching, and shuffling techniques in an effective way and efficiently supports both equality (Section 5) and range queries (Section 6). Indeed, the ordering of the data in the unchained $B+$-tree structure leads to a clear benefit when executing range queries, which instead are not supported by hash indexes. To guarantee that content, access, and pattern confidentiality are preserved also when the outsourced data collection is updated (Section 7) the client adopts a probabilistic split approach that possibly splits a node in the shuffle index every time it is visited (although it could still accommodate the insertion of new values). In this way, an observer cannot infer whether an access to the shuffle index is searching for a value or is updating the data collection. Our proposal provides confidentiality (Sections 9) while maintaining a limited performance overhead (Section 10). Considering reasonable system configurations and a set of requests by users, the shuffle index causes a performance overhead of about 20% with respect to a plain encrypted index. Compared with the most efficient recently proposed approach based on the Oblivious RAM data structure [Stefanov and Shi 2013], the shuffle index is more efficient as it requires less access operations to retrieve outsourced data. Also, it directly supports range queries and updates. We also note that the shuffle index technique simply relies on a soft state stored at the client side. This makes our solution resilient against failures at the client side and accessible by different clients. In fact, after each access, the shuffle index stored at the server is in a consistent state. The performance advantage and extended features of our solution derive from the use of a simpler structure and a limited reduction in the level of protection; for many applications the performance advantages of the shuffle index make it preferable to the alternatives.

## 2. MOTIVATION

The motivation for the shuffle index is to provide complete confidentiality protection to the data as well as to the accesses to them. Such protection is provided even to the eyes of the server storing and managing access to the data. In this section, we illustrate how, in absence of the access and pattern confidentiality provided by our shuffle index, sensitive information about users' activities or data content can be improperly leaked (even assuming encryption of the stored data). For concreteness, in the discussion below, we consider two representative scenarios, with the notes that observations on them can apply to other application scenarios. Both scenarios see a cloud provider (our honest-but-curious server) storing and managing access to data on behalf of the data owner. In the first scenario (also described in [Williams et al. 2008; Stefanov et al. 2013]) data are stocks stored on behalf of a financial organization and accessed by the organization's customers. Access to a given item corresponds to the search, in the data structure, of a specific stock. In the second scenario, data are encrypted authentica-

tors (e.g., passwords or biometric traits) against which physical access to locations by employees must be controlled (with control outsourced to the cloud provider). Access to a given item corresponds to the search, in the data structure, of the authentication proof being sought (e.g., the encrypted biometric traits of the employee entering the gate). While in both scenarios basic encryption can protect confidentiality of data stored at the cloud provider, we note below how observations on the accesses to such data can, without our shuffle index, breach confidentiality of the data themselves or leak sensitive information on users' activities.

For simplicity and concreteness, in the discussion below, we assume that each atomic unit of access (i.e., data block) corresponds to a single data item. This simplification is not limiting; for trees supporting range queries, while every data block will store multiple items, such items will all be close within a given range. Also, the fact that multiple data items can be associated with a single data block introduces noise in the reconstruction, but the noise can be progressively reduced with the observation of a larger number of accesses.

The external server storing the (encrypted) data and managing accesses to them has knowledge of the encrypted physical blocks where the data are stored and can observe every access to such blocks. We now illustrate how, even if data are encrypted, the server, can, based on such observations and on possible limited other knowledge, breach confidentiality of the stored data or actions on them. We identify four main cases.

*No knowledge.* Even without any additional knowledge, the static nature of the (encrypted) data structure exposes information. In fact, by observing accesses, the server can establish correlation of accesses aimed at the same item. For the financial application, the server would be able to infer when different transactions operate on the same stock. In the authentication scenario, the server would be able to keep track of all accesses by the same employee.

*Single-access knowledge.* Suppose the server has knowledge of one specific access instance (e.g., it knows a given access is aimed at a given stock or that a given employee is entering a given building). This limited punctual knowledge can, thanks to the tracking noted above, expose confidentiality of many accesses. Knowledge of a specific access discloses in fact to the server that a given item (stock or authenticator) is stored at a given physical block: all accesses operating on the same block would then be for the same item. For the financial application, the server would be able to know when different transactions will be operating on the same (known) stock. In the authentication scenario, the server would be able to keep track of all the accesses by the same (known) employee.

*Data or access frequency knowledge.* Suppose the server has (some) knowledge on the frequency of accesses to the stored items. Such a knowledge does not necessarily require availability of confidential information. For instance, there are public data describing the distribution of values for many categories of information. Since the server observes every access, it can also build a frequency histogram of the accesses served. Comparing these observations with the frequency knowledge above, the server can establish the physical blocks of certain data items (which are outliers, for example, most frequently or less frequently accessed). For instance, in the financial scenario the server can infer that certain blocks contain information of given stocks. In the authentication scenario, the server can infer that certain blocks contain information of given employees. Each of such inference introduces a 'single access knowledge' that subsequently triggers further inference.

*No knowledge, with range queries.* When the query protocol supports the efficient execution of range queries, the server can observe the fact that accesses hit specific

sets of blocks. By combining several of such observations, the server can infer the plaintext order on the encrypted content of the blocks. A limited amount of additional knowledge about the content of the protected information can then allow the server to quickly reconstruct the plaintext content. In particular, in [Islam et al. 2014] it is shown that, thanks to a limited additional knowledge on the distribution of domain values, an attacker is able to identify up to 90% of the plaintext content after a limited number of range queries have been executed, even under the assumption that the leaves of the tree are not visited in order and are simply accessed as a set. In [Pang et al. 2013] it is shown that, after a sequence of range queries that cover the whole domain have been executed, the analysis of correlation is able to determine the plaintext ordering of the $n$ data blocks with an uncertainty equal to $1/2 - 1/n!$, where the $1/2$ term is due to the uncertainty about the direction of the ordering. In both, the financial and the authentication scenario, such inference would allow the server to establish the plaintext order of the stored values (stocks or authenticator tokens, assuming the ordering key of the latter to be the last name). A limited set of range queries quickly discloses the ordering of blocks and can be used to reconstruct the content.

All the scenarios of inference above are made possible, despite the application of encryption, by the static nature of the data structure, which exposes the fact that accesses targeted a same item and, when the item is known, targeted that specific item. The shuffle index illustrated in this paper breaks such a static correspondence by reorganizing the data structure at every access, so that accesses to the same data item cannot be related anymore since they will target different physical blocks. Analogously, accesses to the same physical blocks will not correspond to accesses to the same data items. As we will illustrate, cover (fake) searches provide further uncertainty in the observations and, together with some caching, enables the execution of such shuffling.

## 3. SHUFFLE INDEX DATA STRUCTURE

For outsourcing, we assume data to be indexed over a candidate key $K$, defined over actual domain $\mathcal{D} \subseteq D_k$, for the data collection and organized as a $B+$-tree, with data stored in the leaves in association with their index values, and where there are no links from a leaf to the next, representing a chain. In the following, we will refer to such a structure as an *unchained $B+$-tree*. Accesses to the data are based on the value of the index. The reason for not representing the links between the leaves is that following such links, when accessing data, would leak to the server (to which the content of the nodes is not known) *i)* the fact that the query being executed is a range query, and *ii)* the order relationship among index values in different nodes. Our data structure is therefore characterized by a fan out $F$, meaning that each node (except the root) has $q \geq \lceil F/2 \rceil$ children and stores $q-1$ values $v_1, \ldots, v_{q-1}$, ordered from the smallest to the greatest. The first child of any internal node in the unchained $B+$-tree is the root of the subtree containing all the values lower than the first value in the node ($v < v_1$). The last child is the root of the subtree containing all the values greater than the last value in the node ($v \geq v_{q-1}$). The second child is the the root of the subtree containing all the values greater than or equal to the first value in the node, but lower than the second one ($v_1 \leq v < v_2$). In general, the $i$-th child, $1 < i < q-1$, is the root of the subtree containing all the values greaten than or equal to the $(i-1)$-th value in the node, but lower than the $i$-th one ($v_{i-1} \leq v < v_i$). Figure 1(a) illustrates a graphical representation of an example of our data structure characterized by fan out $F$ equal to 5. In the figure, nodes appear ordered (left to right) according to the values they store and pointers are represented by arrows. Pointers between nodes of the abstract data structure correspond, at the logical level, to *node identifiers*, which can then be

easily translated at the physical level into physical addresses. At the logical level, our data structure can be seen as a set of nodes, where each node is a pair $\langle id, n\rangle$, with *id* the node identifier and $n$ the node content. Note that the possible order between identifiers does not necessarily correspond to the order in which nodes appear in the value-ordered abstract representation. Figure 1(b) illustrates a possible logical representation of the data structure in Figure 1(a), where nodes appear ordered (left to right) according to their identifiers, which are reported on the top of each node. For simplicity and easy reference, in our example, the first digit of the node identifier denotes the level of the node in the tree. The reason why we distinguish between node identifier and node content is that, as we will see later on, our approach is based on shuffling content among nodes. In other words, a given content may be associated with different identifiers at different times. In the following, when clear from the context, we will use the term *node* to refer either to the content of a node or to the content together with the identifier.

As typical in emerging outsourcing solutions, we use *encryption* to preserve *content* confidentiality. We assume encryption to be applied at the node level (i.e., each node is individually encrypted). To destroy plaintext distinguishability, the encryption function adopts a random salt. Also, the encrypted node is concatenated with the result of a MAC (Message Authentication Code) function applied to the encrypted node and its identifier. In this way, the client can assess the authenticity of the node returned by the server and check whether the server properly performed the write operations requested by the data owner during previous interactions. Note that, since nodes contain pointers to children, the ability to establish authenticity of a node (starting from the root) implies the ability to establish authenticity, and therefore integrity, of the whole data structure.

In the realization of physical accesses, for efficiency reasons, the size of the node to be stored (i.e., its encrypted version together with the result of the MAC function) should be a multiple of the size of the disk block. For simplicity, we assume the size of each encrypted node to be equal to the size of one disk block of the server, and the identifier of the block to be the same as the identifier of the node. Without loss of generality, we also assume that each leaf node can store $F-1$ key values and tuples. We refer to an encrypted node as a *block*. Blocks are formally defined as follows (see Appendix A for more details).

*Definition* 3.1 (*Block*). Let $\langle id, n\rangle$ be a node of an unchained $B+$-tree. The encrypted version of $\langle id, n\rangle$, called *block*, is a pair $\langle id, b\rangle$, with $b{=}\mathcal{E}||\mathcal{T}$, $\mathcal{E}{=}E_{k_e}(\mathit{salt}||n)$, $\mathcal{T}{=}\mathit{MAC}_{k_m}(id||\mathcal{E})$, with $E$ a symmetric encryption function, $k_e$ the encryption key, *salt* a value chosen at random during each encryption, and *MAC* a strongly unforgeable keyed cryptographic hash function with key $k_m$.

We refer to the encrypted version of the logical data structure, outsourced to the server and on which accesses are executed, as *shuffle index*. The term *shuffle* derives from the fact that the structure is dynamically reorganized at every access (see Section 4). Our shuffle index is defined as follows.

*Definition* 3.2 (*Shuffle index*). Let $\{\langle id_0, n_0\rangle, \ldots, \langle id_m, n_m\rangle\}$ be a set of nodes of an unchained $B+$-tree. The *shuffle index* is the set $\{\langle id_0, b_0\rangle, \ldots, \langle id_m, b_m\rangle\}$ of corresponding blocks (Definition 3.1).

Figure 1(c) illustrates the physical representation of the logical structure in Figure 1(b). According to the definition of shuffle index, the server just sees a collection of blocks, each with a given identifier but whose content is encrypted. Access to the data requires an iterative process between the client and the server [Damiani et al. 2003]. The client performs an iteration for each level of the shuffle index starting from the

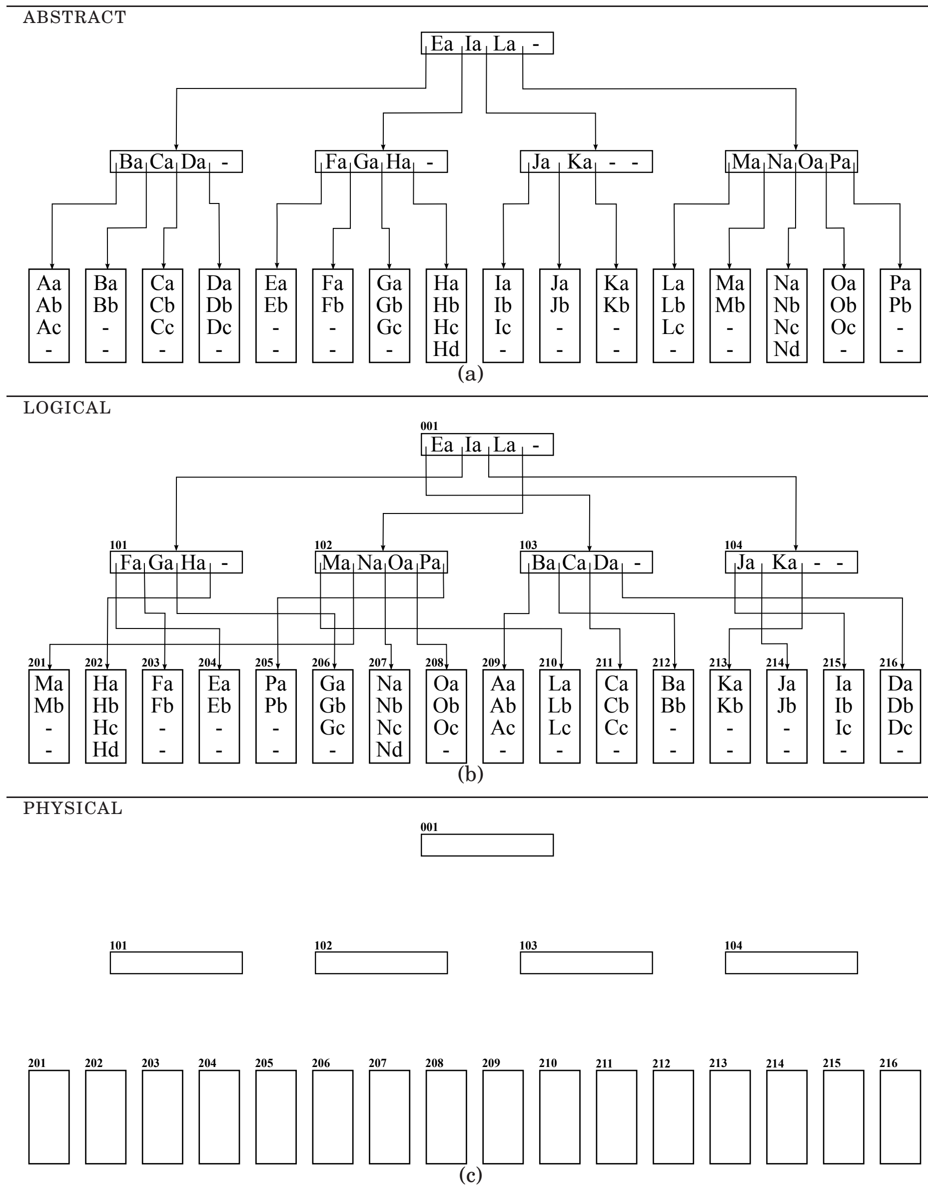



Fig. 1. An example of abstract (a) and logical (b) representation of an unchained $B+$-tree, and of the corresponding view of the server (c)

root. At each iteration it determines the node to be read (i.e., the block to be retrieved from the server) at the next level. The process ends when a leaf block is retrieved, which is the block that contains the index value target of the search (or where it would have appeared, if the index value does not belong to the database).

## 4. PROTECTION TECHNIQUES

We first describe the different aspects of confidentiality we want to guarantee against non authorized observers. We then illustrate our protection techniques complementing encryption for ensuring confidentiality.

### 4.1. Problem Statement and Observer Knowledge

We consider an *observer* that is not authorized to access outsourced data, but that can observe both outsourced data and accesses to them. Such an observer may exploit this information to partially infer confidential data and/or the target of private accesses. Our goal is to protect the confidentiality of the outsourced data and accesses against any possible observer. Since, among all possible observers, the server is the party that has the highest potential for observations (all accesses are executed by it), without loss of generality, in the following we assume the server as our observer. Recall that the server is assumed honest-but-curious, meaning that it is trusted to provide the required storage and management service, but it is not authorized to read the actual data content. It will then try to exploit its knowledge to possibly partially infer confidential data and/or the target of private accesses. The assumption that the server does not corrupt the outsourced data is supported by the use of MAC structures that are verified at every access and are able to immediately detect violations to the integrity.

*Observer's knowledge.* We assume the server to have, or be able to infer from its interactions with the client, the following information:

— the number $m$ of blocks (nodes) in the shuffle index;
— the height $h$ of the shuffle index;
— for each physical block $b_i$, its identifier $id_i$ and the level in the tree of the node it stores;
— the sequence $\{b_{i0}, \ldots, b_{ih}\}$ of blocks read and written for each access operation;
— the frequency with which values in the actual domain of the key $K$ of the shuffle index are typically accessed;
— the target of some of the accesses and the time at which these accesses will be (or have been) performed.

In fact, the server receives from the data owner a set of blocks to store as described in Section 3 and has therefore knowledge of the number $m$ of blocks (nodes) and their identifiers. Since the iterative process adopted to access the shuffle index (summarized in Section 3 and illustrated in the details in Section 5) requires the retrieval of a block for each level of the shuffle index, the server also knows the height $h$ of the shuffle index and, by observing a long enough history of accesses, it can easily establish the level associated with each block. Note that, assuming the adoption of our shuffle index approach, the server does not know and cannot infer the topology of the shuffle index (i.e., the pointers between parent and children). Figure 1(c) illustrates the view of the server on the shuffle index in Figure 1(b). In the working of the system, every access request is performed through a set of read and/or write operations affecting a sequence $\{b_{i0}, \ldots, b_{ih}\}$ of blocks at the server. It then translates into an *observation* $o_i$ of the server for blocks $\{b_{i0}, \ldots, b_{ih}\}$. Adopting our shuffle index, the knowledge of the frequency of accesses to values cannot be inferred from the working of the system. However, we assume that the server might exploit additional external knowledge to gain information about these frequencies (e.g., the frequencies of accesses to the values in the index domain may be publicly known). The server may also have additional knowledge (e.g., thanks to its knowledge of the application domain) on a limited subset of the accesses that the client is expected to perform. As an example, the server may have knowledge of accesses that are scheduled to be performed periodically by the

client (e.g., that every day at noon, the client accesses the remote dataset checking the warehouse status).

*Confidentiality goals.* Before defining the confidentiality we want to guarantee, we note that the server can only monitor accesses at the granularity of a block (node). The basic protection granted by encryption ensures uncertainty on the actual index value (and therefore on the specific data) requested by an access, since any of the index values stored in the returned node could potentially be the target. Such a basic protection cannot be considered sufficient, also because index values stored in the same node will all be close within a given range. Given this observation, in the following, we consider confidentiality breaches at the granularity of nodes.

At any point in time, given a sequence of observations $o_1, \ldots, o_z$ corresponding to all the accesses performed, the server should not be able to infer: *i)* the data stored in the shuffle index (*content confidentiality*); *ii)* the data to which access requests are aimed, that is, $\forall i = 1, \ldots, z$, the server should not infer that $o_i$ aims at a specific node (*access confidentiality*); and *iii)* that $o_i$ aims at accessing the same node as $o_j$, $\forall i, j \in \{1, \ldots, z\}, i \neq j$ (*pattern confidentiality*). Intuitively content confidentiality refers to the data stored in the leaves of the unchained $B+$-tree, access confidentiality to the data targeted by a request, and pattern confidentiality to the relationship among the data targeted by different requests. It is easy to see that encryption provides content confidentiality of data at rest (i.e., data stored on persistent storage that are not accessed) and access confidentiality of individual requests. It is however not sufficient for providing pattern confidentiality of a set of observations. To illustrate this, suppose that a shuffle index never changes. By observing that two accesses retrieve the same blocks, an observer could easily determine that the accesses refer to the same node, thus breaching pattern confidentiality. An observer can then exploit the possible information on the frequencies with which different values can be accessed and a set of observations to reconstruct the correspondence between plaintext values and blocks and infer (or restrict her uncertainty on) the specific node to which a specific access refers, thus breaching access confidentiality.

Since data are encrypted in storage, the information that the server can exploit in the working of the system is the comparison between the frequencies with which blocks are accessed and the frequencies of accesses to different values. The key aspect for guaranteeing all forms of confidentiality above is then to destroy such a correspondence through the combination of three basic strategies: *1) cover searches*, *2) cached searches*, and *3) shuffling*. In a nutshell, cover searches aim at hiding the actual target of an access among a set of fake searches; cached searches are used to protect repeated accesses by keeping recent results in a cache at the client side; and shuffling dynamically changes the allocation of nodes to blocks to decouple physical accesses from logical ones. In the following subsections, we will describe each technique singularly taken. We will then illustrate their combined adoption in Section 5.

### 4.2. Cover Searches

As noted above, the execution of an access over the shuffle index can trivially leak information on the fact that two accesses aim, or do not aim, at the same node. Also, combined with the possible knowledge of the server on frequencies of accesses to node contents, it can help the server to establish the correspondence between node contents and blocks where they are stored (frequently accessed data will correspond to frequently accessed blocks) [Ceselli et al. 2005]. For instance, consider the logical representation of the shuffle index in Figure 1(b), and two consecutive requests for index value 'Fb' translating into accesses to blocks {(001); (101); (203)} and {(001); (101); (203)}, respectively. By observing these sequences of accessed blocks, the server can

infer that the two requests refer to the same data (i.e., the content of block 203). Our first protection technique aims at introducing confusion on the target of an access request by hiding it within a group of other requests that work as covers.

*Cover searches* are fake searches that the client executes in conjunction with the actual *target* search of the index value it aims to access. The number of cover searches is a protection parameter of our approach.

Since, as noted in Section 4.1, the granularity of protection is the block (node), cover searches must provide block diversity, that is, must translate into accesses to different blocks at each level of the shuffle index, but the root. As a matter of fact, covers translating to the same block would not provide any additional protection than that offered by encryption. For instance, 'Fa' cannot be chosen as a cover for 'Fb' as both would translate into accesses to block 203, thus disclosing that the access requests refer to the content of block 203. Given a shuffle index built over a candidate key with actual domain $\mathcal{D}$ and a value $v \in \mathcal{D}$, *path*($v$) denotes the set of blocks in the unique path of the shuffle index that starts at the root and ends in the leaf block where $v$ is possibly stored, if $v$ is in the database. Cover searches are formally defined as follows.

*Definition* 4.1 (*Cover searches*). Let $\{\langle id_0, b_0\rangle, \ldots, \langle id_m, b_m\rangle\}$ be a set of blocks forming a shuffle index built over a candidate key with domain $\mathcal{D}$, and let $v_0$ be a value in $\mathcal{D}$. A set $\{v_1, \ldots, v_n\}$ of values in $\mathcal{D}$ is a set of *cover searches* for $v_0$ if $\forall v_i, v_j \in \{v_0, v_1, \ldots, v_n\} : \Longrightarrow$ *path*($v_i$)$\cap$*path*($v_j$)=$\langle id_0, b_0\rangle$, that is, contains only the root of the shuffle index.

Basically, assuming $num_cover$ cover searches are adopted in the execution of an access, instead of asking the server to retrieve, for each level in the shuffle index, the block in the path from the root to the target, the client asks the server to retrieve $num_cover + 1$ blocks: one corresponds to the block in the path to the target, and each of the others corresponds to the block in the path to one cover.

Intuitively, cover searches hide the actual search within a set of searches. When cover searches are indistinguishable from actual searches, any of the $num_cover + 1$ leaf blocks have the same probability of containing the actual target. We guarantee this cover/target indistinguishability property by ensuring that the frequency distribution with which values in the candidate key domain $\mathcal{D}$ are used as cover searches is the same as the frequency distribution with which values are searched upon client's request (see Section 5). For instance, consider again the two searches above for index value 'Fb' (block 203), and assume the first uses cover 'Ic' while the second one uses cover 'Ma'. The sequences of accesses to blocks observed by the server would now be {(001); (101,104); (203,215)} and {(001); (101,102); (201,203)}, respectively. While without cover the server was able to detect that the two requests aimed at the same block (node), with one cover the server can assess this only with probability $0.5 \cdot 0.5 = 0.25$.

The fact that searches are all executed in parallel (i.e., all the $num_cover + 1$ blocks at each level of the shuffle index are retrieved before proceeding to the next level), confuses the parent-child relationship of the different blocks. In fact, at each level any of the $num_cover + 1$ parents could be associated with any of the $num_cover + 1$ children, producing therefore $(num_cover + 1)^h$ potential paths. For instance, with reference to the example above, 215 could be child of either 101 or 104. Of course, parent-child information (like actual targets) can be disclosed by intersection attacks, observing the same set of blocks in different accesses (101 and 203 in the example above). Intersection attacks are counteracted by caching and shuffling, as explained in the remainder of this section.

### 4.3. Cached Searches

Our second protection technique aims at counteracting *intersection* attacks in the short term and consists in maintaining at the trusted client side a local copy, called *cache*, of nodes in the path to the target. Being client side, we maintain the cache in plaintext (i.e., the cache stores plaintext nodes and not their encrypted version).

*Definition* 4.2 (*Cache*). Let $\{\langle id_0, n_0\rangle, \ldots, \langle id_m, n_m\rangle\}$ be a set of nodes forming an unchained $B+$-tree of height $h$. A cache $\mathcal{C}$ of size *num_cache* for the unchained $B+$-tree is a layered structure of $h+1$ sets $Cache_0, \ldots, Cache_h$, where:

— $Cache_0$ contains the root node $\langle id_0, n_0\rangle$;
— $Cache_l$, $l = 1, \ldots, h$, contains *num_cache* nodes belonging to the $l$-th level of the unchained $B+$-tree;
— $\forall n \in Cache_l$, $l = 1, \ldots, h$, the parent of $n$ in the unchained $B+$-tree belongs to $Cache_{l-1}$ (path continuity property).

Path continuity guarantees that the parent of any node in the cache belongs to the cache. As a consequence, the path connecting the root of the unchained $B+$-tree to every node in the cache completely belongs to the cache itself. We assume the cache to be properly initialized by the data owner at the time of outsourcing, by locally storing nodes in $num_cache$ disjoint paths (i.e., with only the root in common) of the unchained $B+$-tree.

In the working of the system, the cache will be updated and will keep track only of actual (and not of cover) searches, since it is intended to work as an actual cache. We assume the cache at each level to be managed according to the LRU policy, that is, when a new node is added to $Cache_l$, the node least recently used is pushed out from $Cache_l$. The application of the LRU policy guarantees the satisfaction of the path continuity property (Section 8).

The cache helps in counteracting short term intersection attacks since it avoids the client to search for a repeated target of two close access requests. In fact, with the cache, two close searches, regardless of whether they aim or not at the same target, will always have some blocks in common. Therefore, two close searches aimed at the same target will not be observable as such. For instance, with reference to the two consecutive requests for index value 'Fb' in Section 4.2, the second request would find 'Fb' in cache. Since the number of blocks requested to the server has always to be the same (i.e., $num_cover + 1$), the client would generate, for the second request, two cover searches (e.g., 'Cb' and 'Ma'). Consequently, the observations of the server on the two requests would be {(001); (101,104); (203,215)} and {(001); (102,103); (201,211)}, respectively. The server would not be able to determine whether the two requests aim at the same target. The reader may wonder why we perform $num_cover + 1$ fake cover searches when the target node is already in cache. First, as illustrated in Section 4.1, if the observer knows that an access was to be executed, not performing it would leak information on the fact that the target node is in the cache. The server may then exploit such information and try to guess the target of the access. As an example, if every day at noon the client checks the warehouse balance, not accessing the dataset at that time would reveal to the server that this information is in cache. If there is no concern about this, the client can avoid to repeat accesses that can be directly managed with the cache content (i.e., when the leaf node target of the access is in cache). Second, the protection given by the cache does not work only as an independent technique, but plays a role together with the other protection techniques (for more details, see Section 9).

### 4.4. Shuffling

Caching does not prevent intersection attacks on observations that go beyond the size of the cache. As an example, suppose that no cache is used (i.e., $num_cache$=0), and with reference to Figure 1(b) consider three consecutive requests all for index value 'Fb', using one cover search for each request (e.g., 'Ma', 'Cb', and 'Ic', respectively). These access requests will translate into the following sequences of accesses to blocks {(001); (101,102); (201,203)}, {(001); (101,103); (203,211)}, and {(001); (101,104); (203,215)}, respectively. Assuming the indistinguishability of targets and covers, by the observation of these sequences of accesses the server can infer with probability $0.5 \cdot 0.5 \cdot 0.5 = 0.125$ that the three access requests refer to the same data (i.e., the content of block 203). Also, accesses leak to the server the parent-child relationship between blocks. While the information on the parent-child relationship by itself might seem to not compromise confidentiality, it can easily open the door to privacy breaches and should then remain confidential. Given a long enough history of observations, the server will be able to reconstruct the topology of the shuffle index and therefore gain knowledge on the similarity between values stored in the blocks.

Our third protection technique starts from the observation that inferences such as the one mentioned above are possible to the server by exploiting the one-to-one correspondence between a block and the node stored in it: accesses to the same block trivially correspond to accesses to the same node. *Node shuffling* breaks this one-to-one correspondence by exchanging the content among nodes (and therefore blocks). Since a block depends on the content of the corresponding node and on the node identifier (Definition 3.1), shuffling clearly requires the re-computation of the blocks associated with shuffled nodes and then requires node decryption and re-encryption. Note how the re-encryption of a node, applied to the node content concatenated with a possibly different node identifier and a different random salt, produces a different encrypted text (block). This aspect is particularly important since encrypted text corresponding to a given node automatically changes at each access, making it impossible for the server to track the shuffling executed and to determine if the node content stored in a block has been changed or has remained the same. Node shuffling is formally defined as follows.

*Definition* 4.3 (*Shuffling*). Let $\mathcal{N}$=$\{\langle id_1, n_1\rangle, \ldots, \langle id_m, n_m\rangle\}$ be a set of nodes at the same level of an unchained $B$+-tree and $\pi$ be a permutation of $id_1, \ldots, id_m$. The *node shuffling* of $\mathcal{N}$ with respect to $\pi$ is the set $\{\langle id_1, n'_1\rangle, \ldots, \langle id_m, n'_m\rangle\}$ of nodes, where $id_i = \pi(id_j)$ and $n'_i = n_j$, with $i, j = 1, \ldots, m$.

Intuitively, our approach exploits shuffling by exchanging the contents of all blocks read in the execution of an access and the nodes in cache (so that their contents are shuffled), and rewriting all of them back to the server. In this way, the correspondence existing between block identifiers and the content of the nodes they store is destroyed. For instance, assume that shuffling is used and that the server observes the following sequences of accesses to blocks (which are the same sequences discussed above): {(001); (101,102); (201,203)}, {(001); (101,103); (203,211)}, and {(001); (101,104); (203,215)}. The server can observe that the three sequences have a leaf block in common (i.e., 203). The three requests aim at accessing the same node only if: the second and third requests are for the content of block 203 (the probability is $0.5 \cdot 0.5 = 0.25$); the data target of the first request coincides with the content of block 203 after the first shuffling operation (the probability is $0.5$); and the content of block 203 is not moved by the second shuffling operation (the probability is $0.5$). As a consequence, $0.0625$ is the probability that the three requests aim at the same node.

```
/* S : shuffle index on a candidate key with domain D, height h, fan out F */
/* Cache_l, l=0, ..., h : cache */
/* num_cache : number of nodes in Cache_l, l=1, ..., h */
/* num_cover : number of cover searches */
INPUT    target_value : value to be searched in the shuffle index
OUTPUT   n : leaf node that contains target_value
MAIN
 1: randomly choose a set cover_value[1...num_cover+1] cover searches for target_value (Definition 4.1)
 2: for l:=1...h do
 3:    /* identify the blocks to read from the server */
 4:    target_id := identifier of the node at level l along the path to target_value
 5:    cover_id[i] := identifier of the node at level l along the path to cover_value[i], i=1...num_cover+1
 6:    if target_id is the identifier of a node in Cache_l then
 7:       ToRead_ids := cover_id[1...num_cover+1]
 8:    else ToRead_ids := {target_id} ∪ cover_id[1...num_cover]
 9:    /* read blocks */
10:    Read := download from the server the blocks with identifier in ToRead_ids and decrypt their content
11:    /* shuffle nodes */
12:    let π be a permutation of the identifiers of nodes in Read and Cache_l
13:    shuffle nodes in Read and Cache_l according to π
14:    update pointers to children of the parents of nodes in Read and Cache_l according to π
15:    encrypt and write at the server nodes accessed and in Cache_{l−1} at iteration l − 1
16:    target_id := π(target_id)
17:    cover_id[i] := π(cover_id[i]), i=1...num_cover+1
18:    /* update cache at level l */
19:    update Cache_l according to LRU policy and access to node with identifier target_id
20: encrypt and write at the server nodes accessed and in Cache_h at iteration h
21: /* return the target leaf node */
22: return the node in Cache_h with identifier equal to target_id
```

Fig. 2. Shuffle index access algorithm

Shuffling among nodes at a given level requires to update the parents of the nodes so that the pointers in them properly reflect the shuffling. For instance, consider Figure 1(b) and assume nodes (101,102) are shuffled so that $\pi(101)=102$ and $\pi(102)=101$, (i.e., their contents are swapped). As a consequence, root node [103 Ea 101 Ia 104 La 102 - -] must be updated to be [103 Ea 102 Ia 104 La 101 - -].

Note that cached nodes are shuffled together with accessed nodes and rewritten back to the server. This participation in the shuffling of the cached blocks is needed since: *i)* cached blocks do need to be rewritten, as their content (pointers to non-cached children involved in the access, in particular) might have changed; and *ii)* rewriting the blocks in the cache back to the server without shuffling their content would not enforce on them the protection of shuffling (as they would be known by the server to have the same content as when they were read). The participation of the cache in the shuffling and the rewriting of the cache on the server do not diminish the protection given by caching since again, two close searches aimed at the same target will not be observable as such (their access profile is the same as the one of searches for different targets).

## 5. ACCESS EXECUTION AND SHUFFLE INDEX MANAGEMENT

We illustrate how the protection techniques described in Section 4 (cover, cache, and shuffling) are applied in a joint way in the execution of an access and how the shuffle index is managed. Figure 2 presents the algorithm, executed at client-side, enforcing the search process and the updates to the blocks composing the shuffle index. Here, we assume that each internal node $n$ is associated with a unique identifier, $n.id$, and two arrays, $n.values$ and $n.pointers$, storing the key values and pointers to the child nodes, respectively. Leaf nodes differ from internal nodes by the array $pointers$ that is replaced by the array $data$ containing the data associated with the key values stored in the leaf nodes.

Given a request for searching $target_value$ in shuffle index $\mathcal{S}$, the algorithm first determines $num_cover+1$ values, $cover_value[1],\ldots,cover_value[num_cover+1]$ to be used

as cover searches (Definition 4.1) for $target_value$ (line 1). Note that the number of cover searches is $num_cover + 1$, because for each level of the shuffle index, $num_cover + 1$ blocks have to be downloaded from the server and therefore, if the block in the path to the target value belongs to the cache, an additional cover search becomes necessary. For each level $l = 1, \ldots, h$, the algorithm then executes the following process. The algorithm first determines the identifiers (i.e., $ToRead_ids$) of the blocks at level $l$ in the path to the target value (i.e, $target_id$, line 4) and to the cover searches (i.e., $cover_id[1], \ldots, cover_id[num_cover + 1]$, line 5). If the node in the path to the target value does not belong to $Cache_l$ (i.e., a cache miss occurs), one of the values initially chosen as a cover is discarded and only $num_cover$ out of the $num_cover + 1$ cover searches are performed (lines 6-8). It sends to the server a request for the blocks with identifiers in $ToRead_ids$ and decrypts their content, obtaining a set $Read$ of nodes (line 10). The nodes in $Read$ and $Cache_l$ are then shuffled according to a random permutation $\pi$ (Definition 4.3) (lines 12-13). As a consequence, the pointers stored in the nodes that are parents of the nodes in $Read$ and $Cache_l$, which are the nodes accessed by the algorithm during the previous iteration (at level $l - 1$), are updated according to permutation $\pi$, encrypted, and sent back to the server for storage (lines 15). To reflect the effects of the shuffling on all the variables of interest, $target_id$ and $cover_id[i]$, $i = 1, \ldots, num_cover+1$, are updated according to $\pi$ (lines 16-17). The algorithm finally updates $Cache_l$ by possibly inserting, if a cache miss occurred, the most recently accessed node in the path to the target value (line 19). When the visit of the shuffle index terminates, the node identified by $target_id$, which is the leaf node where $target_value$ is stored (if present in the database), is returned (line 22).

To properly access and manage the shuffle index structure, it is necessary to keep track of the nodes and blocks accessed during the current and previous iteration, as well as of the first occurrence of a cache miss during the traversal of the tree, which are not explicitly considered by the algorithm in Figure 2. A detailed pseudocode of the algorithm accessing and managing a shuffle index structure is illustrated in Appendix B.

We note that the choice of cover searches (line 1) has to satisfy the target/cover *indistinguishability property*. Intuitively, indistinguishability is guaranteed if cover searches and target searches follow the same frequency distribution. However, the frequency distribution with which the target values are accessed may not be known in advance. If this is the case, the client can build a simple statistical model [Silverman 1986] that: *i)* estimates the probability density function bound to the occurrence of target values; and *ii)* chooses cover values by sampling from the estimated distribution. Our implementation of the shuffle index concretely implements indistinguishability (see Section 9.3). To empirically demonstrate this property, we considered recurrences within 100 accesses of the same physical blocks, and analyzed, for every recurrence, if this was due to a target or a cover access. The average value of the absolute difference in probability between targets and covers was equal to 0.0001 (i.e., it is hard for the server to distinguish targets from covers).

*Example* 5.1. Consider the index in Figure 1(b) (reported for convenience at the top of Figure 3) and assume $num_cover$=1, $num_cache$=2, $target_value$='Fb', and that the cache initially stores the nodes with identifier $\{001\}$ at level 0, $\{101, 103\}$ at level 1, and $\{206, 209\}$ at level 2. Initially, two values, for example, 'Ma' and 'Ic', are randomly chosen as covers for 'Fb'.

For $l = 1$, the identifier of the node along the path to 'Fb' is 101, which is in $Cache_1$. Therefore, the two nodes in the paths to the cover searches (i.e., 102 and 104, respectively) are read from the server. The nodes in $Cache_1$ and in $Read$ (i.e., downloaded from the server) are shuffled according to the following permutation $\pi$: 101's content

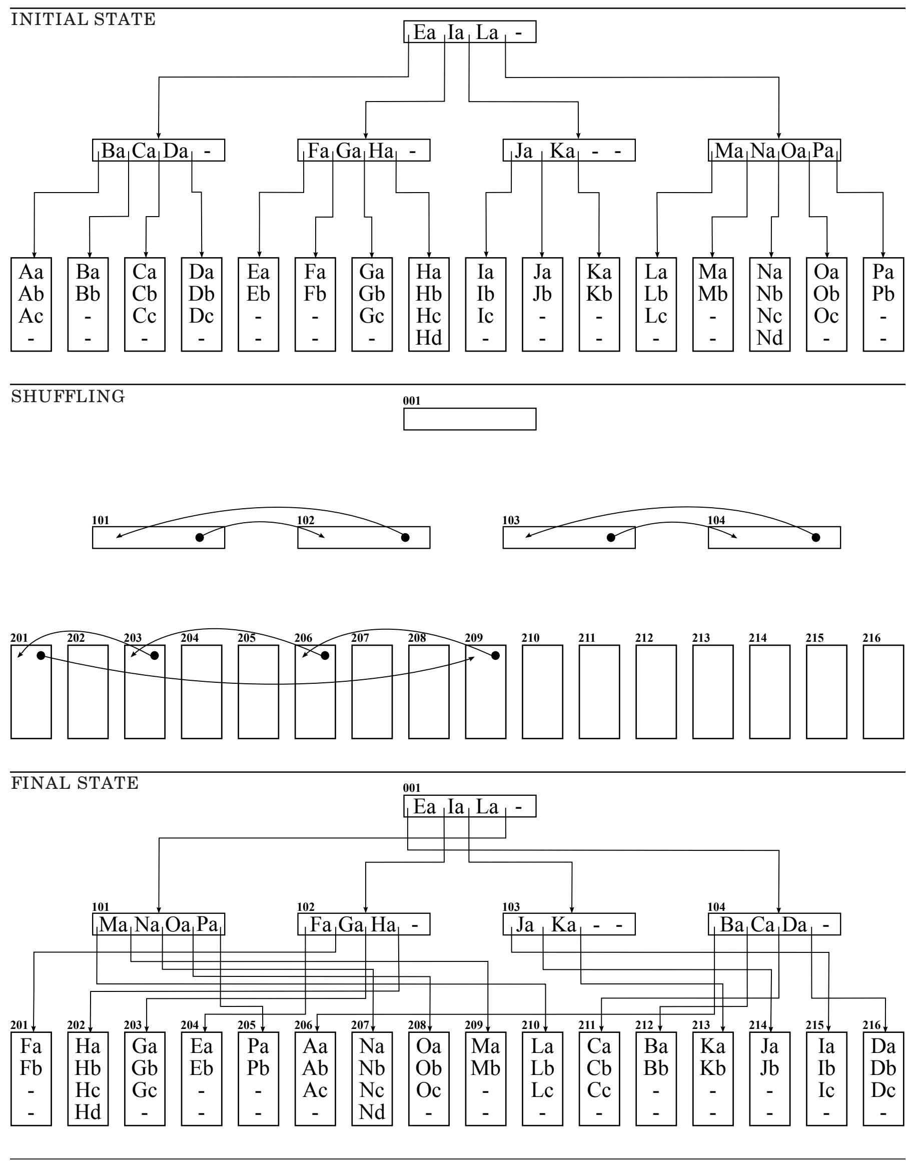



Fig. 3. Evolution of the shuffle index for Example 5.1

moves to block 102; 102's to 101; 103's to 104; and 104's to 103. $Cache_1$ is updated by refreshing the timestamp of node 102 (i.e., $target_id$). Finally, the pointers in $n_0$ (i.e., the root node) are updated according to $\pi$, and node 001 is encrypted and stored at the server.

For $l = 2$, the identifier of the node along the path to 'Fb' is 203, which does not belong to $Cache_2$, and hence the second cover is dropped. Nodes 201 and 203 are read. The nodes in $Cache_2$ and in $Read$ are shuffled according to the following permutation $\pi$: 201's content moves to block 209; 203's to 201; 206's to 203; and 209's to 206. Node 201 (i.e., $target_id$) is inserted into $Cache_2$ and node 206 (which we suppose the least recently used) is pushed out. The pointers in the nodes accessed during the previous

iteration (i.e., 101, 102, 103, 104) are updated according to $\pi$, encrypted, and sent to the server.

Finally, accessed leaf nodes (i.e., 201, 203, 206, 209) are encrypted and sent to the server. Node 201 (i.e., *target_id*) is returned. Figure 3 shows the evolution of the shuffle index.

## 6. EXECUTION OF RANGE QUERIES

Our shuffle index can support the execution of *range queries* defined on the candidate key $K$ on which the shuffle index is built. A range query aims at retrieving all data whose keys are in a range [*lower_bound*,*upper_bound*]. With a traditional $B$+-tree, a range query is executed by first performing an equality query for $K$=*lower_bound* to retrieve the first leaf node $n$ of interest. The data in $n$ are examined and then the sibling link to the next leaf is possibly followed. This process terminates when a leaf node with at least one key greater than or equal to *upper_bound* is reached. As already noted in Section 3, this process would leak to the server both the fact that the query being executed is a range query and the relative order of leaf nodes in the shuffle index. To prevent such a leakage, the leaf nodes of the shuffle index are not linked, and the evaluation of range queries requires a different approach. Our solution consists in translating a range query into an *equivalent set of equality queries*, meaning that the leaf nodes returned by these equality queries are all and only the leaf nodes that store data whose keys are in the range [*lower_bound*,*upper_bound*]. Note that our protection techniques guarantee that an observer cannot infer whether these equality queries are related to each other. In other words, an observer is not able to distinguish the evaluation of a range query from an arbitrary sequence of accesses to the shuffle index (see Section 9). Also, the equivalent set of equality queries cannot be computed a priori because neither the actual key values stored in the leaves nor their organization is known to the client. We then dynamically determine these queries by exploiting the organization of the key values in the unchained $B$+-tree. Recall that the $i$-th child of any internal node of a $B$+-tree, with $i = 1, \ldots, q-1$, is the root of the subtree that contains the values $v$ with: $v < v_1$; $v_{i-1} \leq v < v_i$, $i = 2, \ldots, q-2$; $v \geq v_{q-1}$ (see Section 3). It is then easy to see that the first key value stored in a leaf node (the only exception is for the first leaf node that contains the minimum key value) is also stored in one of the internal nodes. Suppose now to consider two contiguous leaf nodes of the shuffle index, say $n_i$ and $n_j$ (i.e., we assume that there is not another node, say $n_l$, such that $n_i.values[x] < n_l.values[y] < n_j.values[z]$, $x = 1, \ldots, \mathbf{Length}(n_i.values)$, $y = 1, \ldots, \mathbf{Length}(n_l.values)$, $z = 1, \ldots, \mathbf{Length}(n_j.values)$, with $\mathbf{Length}(n.values)$ the number of values actually stored by node $n$). The smallest value in $n_j$ (i.e., $n_j.values[1]$) is represented within the deepest common ancestor of $n_i$ and $n_j$. As an example, consider leaf nodes [Da Db Dc -] and [Ea Eb - -] in Figure 1. Their common ancestor is the root node that contains 'Ea'. Therefore, when we execute the first equality query for $K$=*lower_bound*, we have to identify the common ancestor of the leaf node possibly storing *lower_bound* and the next leaf node. To this purpose, we adapt the algorithm described in Section 5 to keep track of the next target to be searched (see the pseudocode in Appendix B). Intuitively, the process starts with the execution of an equality query looking for *target_value*=*lower_bound*, which identifies, while visiting the shuffle index, the first key value stored in the leaf node (if any) following the one resulting from the search for *target_value*. A new equality query is then executed looking for this value as a target. The process terminates when the interval has been completely covered.

As an example, consider the shuffle index in Figure 1 and a range query with range [Cb,Db]. We first evaluate an equality query for $K$='Cb'. The algorithm starts from the root [103 Ea 101 Ia 104 La 102 - -], visits node 103 [209 Ba 212 Ca 211 Da 216 - -], and returns leaf node 211 [Ca Cb Cc -]. Since the interval is not completely covered by the leaf, we execute

another equality query for $K$='Da', which is the first value in the next leaf node. Indeed, it is the value in the parent of the returned leaf following the pointer to node 211. When the search for value 'Da' terminates, returning leaf node $_{216}$[Da Db Dc -], the interval has been covered as the first value in the next leaf is 'Ea'>'Db'. The set of leaf nodes returned by the sequence of searches is then {$_{211}$ [Ca Cb Cc -], $_{216}$[Da Db Dc -]}, which is equivalent to the result of the original range query.

Note that the overhead introduced in the evaluation of range queries with our shuffle index is due to the fact that, for accessing the next leaf node, we need to restart the process from the root. The number of additional nodes visited is $h(w-1)$, where $h$ is the height of the tree and $w$ is the number of leaves in the result of the range query. In practical scenarios, since $h$ is low and the number of values stored in each leaf is high, the number $w$ of leaves in the result of a range query is small. The overhead is then limited, as also shown by the experimental results (see Section 10).

## 7. UPDATE MANAGEMENT

Since the outsourced data are likely to change over time, the shuffle index storing them may need to be updated accordingly. The possible update operations are related to the change of an existing tuple, the removal of a tuple, or the insertion of a new tuple. These operations are recognizable from read-only accesses whenever their management implies a change in the structure of the shuffle index. Note that the update of a tuple may impact the shuffle index only when it requires a change in the value of attribute $K$ of the tuple. In this case, the update operation can be seen as the removal of a tuple followed by the insertion of the same tuple with a new value for attribute $K$, and can then be treated as described in the following.

The removal of a tuple within a node $n$ can bring the number of data stored in $n$ below the minimum allowed (which is usually equal to $\lceil F/2 \rceil$ for traditional $B+$-trees). In this case, a traditional $B+$-tree is updated by redistributing the data stored in $n$ and in one of its siblings or by merging node $n$ with one of its siblings. These operations cannot be executed on our shuffle index without revealing that the access removed a tuple since they require the download of an additional node (i.e., the sibling node) followed by a possible change in the structure of the shuffle index. We then propose to remove tuples by marking them as 'not-valid'. The advantages of such a solution are that the shuffle index remains unchanged and the indistinguishability of removal operations from all the other operations is preserved.

The insertion of a new tuple in a leaf node that can accommodate it, either because it stores less than $F-1$ tuples or because it stores tuples marked as 'not-valid' (which is then overwritten), is clearly indistinguishable from a search or removal operation. A problem arises when we need to insert a new tuple in a leaf node that is already full because, according to traditional $B+$-tree management operations, the node has to be split in two: half the keys (the ones representing higher values) and the corresponding tuples are moved into a new node $n'$, and the smallest key moved into $n'$ is also inserted into the parent of $n$. Note that the insertion of a key value in the parent node may in turn cause a split of the parent. In the worst case, all nodes in the insertion path are split, thus possibly increasing by one the height of the shuffle index, Since the splitting of a node is recognizable by an observer, also the insertion of a new tuple may be recognizable. Our idea for counteracting this problem is to split nodes in a probabilistic way when they are accessed, independently from the purpose of the access (i.e., insertion, removal, or read). In other words, whenever we access a node of the shuffle index, the node is split with a given probability that increases as the number of values stored within it increases; the probability is equal to one when the node is full. This strategy has a twofold advantage: *1)* we have the guarantee that whenever a node is split, its parent is not full (otherwise, we would have split its parent before

accessing the node) and therefore it can accommodate an additional key value without propagating the split up in the shuffle index; *2)* an observer cannot know whether a split operation is due to the insertion of a new tuple or whether it is the consequence of our probabilistic approach (a node can be split when we visit it during any kind of operation).

In the following, we describe more in details the split operation and its impact on the shuffle index, differentiating between the split of the root and the split of a leaf or internal node.

### 7.1. Split of the root node

The split of the root is performed only when it is full (i.e., the probability of splitting a non-full root is zero) because it causes the increase by one of the height of the shuffle index, thus impacting the performance of all subsequent accesses. Differently from traditional $B+$-trees, when the root of a shuffle index is split, it cannot generate a new root with two children, but we need to guarantee the presence of at least $num_cover$ + $num_cache$ + 1 children of the new root. This constraint arises from the application of our protection techniques that require $num_cover$ + $num_cache$ + 1 distinct paths to introduce uncertainty on the target of the access to the shuffle index. The split of the root is formally defined as follows.

*Definition* 7.1 (*Root splitting*). Let $\langle id_0, n_0\rangle$ be the root of an unchained $B+$-tree with $F - 1$ key values, $nc$ be the number of children required for the new root, and $nk$=$\lfloor(F - 1 - (nc - 1))/nc\rfloor$ be the minimum number of key values that each child of the new root will store. The splitting of the root produces $nc$ new nodes $\langle id'_0, n'_0\rangle$, $\langle id_1, n_1\rangle, \ldots, \langle id_{nc-1}, n_{nc-1}\rangle$, with $\langle id'_0, n'_0\rangle$ the new root and $\langle id_i, n_i\rangle$, $i = 1, \ldots, nc - 1$, the children of $\langle id'_0, n'_0\rangle$, such that:

R1. $i = 1, \ldots, (nc - 1)$, $n_0.values[(1 + nk) \cdot i]$ is moved to $n'_0.values[i]$;
R2. $i = 0, \ldots, (nc - 1)$, $id_i$ is inserted into $n'_0.pointers[i]$;
R3. $i = 1, \ldots, (nc - 2)$, $n_0.values[(1 + nk) \cdot i + 1], \ldots, n_0.values[(1 + nk) \cdot i + nk]$ are moved to $n_i.values[1], \ldots, n_i.values[nk]$;
R4. $i = 1, \ldots, (nc - 2)$, $n_0.pointers[(1 + nk) \cdot i + 0], \ldots, n_0.pointers[(1 + nk) \cdot i + nk]$ are moved to $n_i.pointers[0], \ldots, n_i.pointers[nk]$;
R5. $i = (1 + nk)(nc - 1) + 1, \ldots, \mathbf{Length}(n_0.values)$, $n_0.values[i]$ is moved to $n_{nc-1}.values[i - (1 + nk)(nc - 1)]$;
R6. $i = (1 + nk)(nc - 1), \ldots, \mathbf{Length}(n_0.pointers)$, $n_0.pointers[i]$ is moved to $n_{nc-1}.pointers[i - (1 + nk)(nc - 1)]$;

Conditions R1-R2 determine the key values and pointers that have to be moved from the old root note to the new root. Conditions R3-R4 determine the key values and pointers that have to be moved from $n_0$ to nodes $n_1, \ldots, n_{nc-2}$. Conditions R5-R6 determine the key values and pointers that have to be moved from $n_0$ to the last children $n_{nc-1}$ of the new root. Note that node $n_{nc-1}$ has to be considered separately from the other children of the new root because it may contain a number of key values (and pointers) greater than $nk$. According to Definition 7.1, node $\langle id_0, n_0\rangle$ becomes the left-hand side child of the new root, and the key values and pointers originally stored in $n_0$ are re-distributed within nodes $\langle id'_0, n'_0\rangle$, $\langle id_1, n_1\rangle, \ldots,$ $\langle id_{nc-1}, n_{nc-1}\rangle$ to preserve the correctness of the shuffle index. As an example of root split, consider the root node in Figure 4(a), and assume that $num_cover$=1, $num_cache$=0. In this case, the split of the root has to generate a new root with at least two children, as illustrated in Figure 4(a). Note that both the new root node and the second of its children are allocated to free blocks.

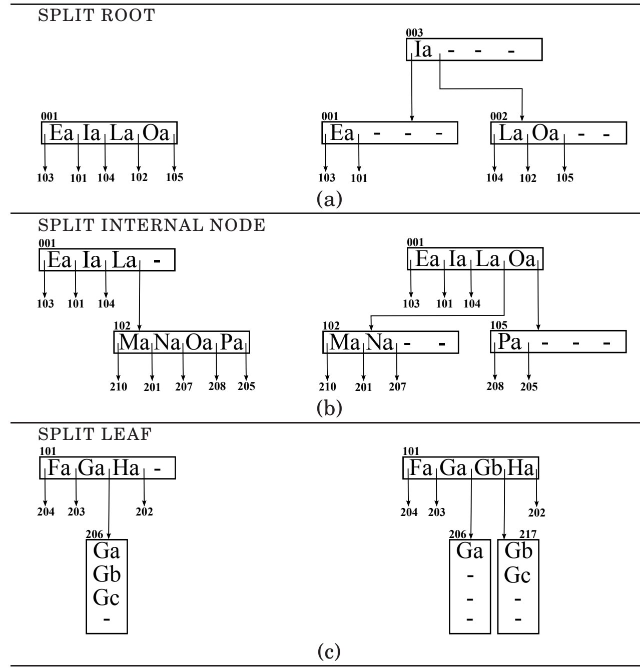



Fig. 4. An example of split of the root node, of an internal node, and of a leaf node

To support insertion, removal, and update operations, the algorithm described in Section 5 should check, at each access to the shuffle index, whether the root node is full and possibly split it according to the strategy illustrated above. The pseudocode of the function splitting the root node and its detailed description are reported in Appendix C.

### 7.2. Split of non-root nodes

The split of an internal or leaf node is performed as in traditional $B+$-trees, with the difference that it can be executed also when the node is not full. It then creates a new node in the shuffle index and requires the update of the parent of the split node, as formally defined in the following definition.

*Definition* 7.2 (*Internal and leaf node splitting*). Let $\langle id,n \rangle$ be an internal or leaf node of an unchained $B+$-tree, $\langle id_p,n_p \rangle$ be the parent node of $\langle id,n \rangle$, and $promoted_value$=$n.values[i]$, with $i = \lceil(\textbf{Length}(n.values))/2\rceil$, be a value stored in $n$. The splitting of node $n$ with respect to $promoted_value$ produces a new node $\langle id', n' \rangle$ such that:

— if $\langle id,n \rangle$ is an internal node:
  I1. $promoted_value$ is moved to $n_p.values$ and $id'$ is inserted into $n_p.pointers$;
  I2. $j = i+1, \ldots, \textbf{Length}(n.values)$, $n.values[j]$ is moved to $n'.values[j-i]$;
  I3. $j = i, \ldots, \textbf{Length}(n.pointers)$, $n.pointers[j]$ is moved to $n'.pointers[j-i]$;
— if $\langle id,n \rangle$ is a leaf node:
  L1. $promoted_value$ is copied into $n_p.values$ and $id'$ is inserted into $n_p.pointers$;
  L2. $j = i, \ldots, \textbf{Length}(n.values)$, $n.values[j]$ is moved to $n'.values[j-i+1]$;
  L3. $j = i, \ldots, \textbf{Length}(n.data)$, $n.data[j]$ is moved to $n'.data[j-i+1]$.

Condition I1 (L1, resp.) determines the value and the pointers that have to be added to the parent of the split node for preserving the correctness of the index structure.

Note that the insertion of $promoted_value$ and $id'$ into $n_p$ is performed so to preserve the ascending order among the values and pointers already stored in the node. Also, this insertion may cause the parent node to overflow, which then requires the recursive application of the split operation to the parent node $n_p$. Condition I2 (L2, resp.) determines the key values that have to be moved from $n$ to the new node $n'$. Condition I3 (L3, resp.) determines the pointers (data, resp.) that have to be moved from $n$ to the new node $n'$.

According to Definition 7.2, the split of a leaf node differs from the split of an internal node not only because the leaf node stores the actual data while the internal node stores the pointers to its children, but also because for leaf nodes the key value $promoted_value$ is inserted both into the new node $n'$ and into the parent node; it is inserted only into the parent node, otherwise. As an example of node splitting, consider the internal node allocated to block 102 in the shuffle index in Figure 1 and reported in Figure 4(b) for the reader's convenience together with its parent node, and suppose that $promoted_value$='Oa'. The nodes resulting from the split are illustrated in Figure 4(b), which also highlights the update to the parent of the split node. Figure 4(c) illustrates the split of the leaf node allocated to block 206 in the shuffle index in Figure 1. It is immediate to see that the promoted value $promoted_value$='Gb' appears both in the new leaf and in its parent.

To support the probabilistic split of visited nodes, the access algorithm illustrated in Section 5 is modified as discussed in detail in Appendix C. In fact, the algorithm needs to invoke, for each visited node, a procedure in charge of evaluating its possible split according to the strategy illustrated above. This procedure then possibly splits the node and updates the variables necessary to guarantee the correctness of the access process accordingly (e.g., cached nodes, variables representing the target and cover searches). A detailed description and the pseudocode of this procedure are presented in Appendix C.

The split operation of an internal or leaf node $n$ is regulated by a probability function $\wp(n)$ that is equal to: 1, if $n$ is full; 0, if $n$ stores less than $t$ values, where $t$ is a threshold fixed by the client to prevent the presence in the shuffle index of almost empty nodes; a value in (0,1) that increases with the number of values stored in the node, otherwise. Formally, $\wp(n)$ is defined as follows.

*Definition* 7.3 (***Probabilistic Split***). Let $\langle id, n\rangle$ be a node of an unchained $B$+-tree with fan out $F$, and $2 \leq t < F-1$ be a threshold value representing the minimum number of values in each node. The event $E(n)$ of splitting $n$ when visiting it follows a Bernoulli distribution and takes value $E(n)$=***true*** with success probability $\wp(n)$ and value $E(n)$=***false*** with failure probability $1-\wp(n)$, where:

$$\wp(n)=\begin{cases} 0, & \textbf{Length}(n.values)\leq t \\ \frac{\textbf{Length}(n.values)-t}{(F-1)-t} & \text{otherwise} \end{cases}$$

The value of parameter $t$ has clearly an impact on the cost of accessing a tuple since it determines the height of the shuffle index. Note however that the maximum growth in the height of the shuffle index caused by the adoption of a probabilistic split is constant, as it depends on constant values, that is: the fan out $F$ of the tree, the chosen threshold $t$, and the number $|\mathcal{D}|$ of tuples (actual valid tuples or old tuples marked as 'no-valid' and not yet overwritten) in the tree. The height of a traditional $B$+-tree with fan out $F$ and $N$ leaf nodes is $\lceil \log_F(N) \rceil$. In our case, since the number of tuples stored in the leaf nodes is $|\mathcal{D}|$ and each node contains at least $t$ key values, the maximum height of the shuffle index is $\left\lceil \log_{t+1} \frac{|\mathcal{D}|}{t} \right\rceil$, which corresponds to a shuffle index where

each node (except the root) contains exactly $t$ key values. The minimum height of the shuffle index is instead obtained when all nodes are full and is equal to $\left\lceil \log_F \frac{|\mathcal{D}|}{F-1} \right\rceil$. It is easy to see that the performance overhead can be reduced by properly tuning parameter $t$ choosing a value close to $F-1$ to limit the height of the tree, while protecting access confidentiality. We note that the value of parameter $t$ does not impact the indistinguishability between search and insert operations (see Section 9.2).

*Example* 7.4. Consider the index in Figure 1(b) (reported for convenience at the top of Figure 5) and assume that *num_cover*=1, *num_cache*=2, and that the client is interested in inserting a new tuple, with key attribute value *target_value*='Fc'. We also assume that the initial configuration of the cache is the same considered in Example 5.1 (i.e., it stores nodes {001} at level 0, {101, 103} at level 1, and {206, 209} at level 2) and that the covers for 'Fc' are still 'Ma' and 'Ic'. Initially, the algorithm visits the root node, which is not split as it is not full.

For $l = 1$, the identifier of the node along the path to 'Fc' is 101, which is in $Cache_1$. Therefore, the two nodes in the paths to the cover searches (i.e., 102 and 104, respectively) are read from the server. The algorithm evaluates whether to split each of these nodes and splits node 102 because it is full. To this purpose, it gets a free block identifier from the server, 105 in our example, splits node [210 Ma 201 Na 207 Oa 208 Pa 205] into nodes [210 Ma 201 Na 207 - - - - ] and [208 Pa 205 - - - - - - ] allocated to blocks 102 and 105, respectively, and updates the root node (i.e., the parent of node 102) that becomes [103 Ea 101 Ia 104 La 102 Oa 105] (as illustrated in Figure 4(b)). The nodes in $Cache_1$ and in *Read* (i.e., downloaded from the server) are shuffled according to the following permutation $\pi$: 101's content moves to block 102; 102's to 104; 103's to 105; 104's to 101; and 105's to 103. $Cache_1$ is updated by refreshing the timestamp of node 102 (i.e., *target_id*). Finally, the pointers in $n_0$ (i.e., the root node) are updated according to $\pi$, and node 001 is encrypted and stored at the server.

For $l = 2$, the identifier of the node along the path to 'Fc' is 203, which does not belong to $Cache_2$, and hence the second cover is dropped. Nodes 201 and 203 are read. The algorithm evaluates whether to split visited nodes and decides to split node 206 although it is not full, thanks to the probabilistic approach adopted. The procedure gets a free block identifier from the server, 217 in our example, splits leaf node [Ga Gb Gc -] into nodes [Ga - - - ] and [Gb Gc - - ] allocated to blocks 206 and 217, respectively, and updates the parent of node 206 (i.e., node 102) as [204 Fa 203 Ga 206 Gb 217 Ha 202] (as illustrated in Figure 4(c)). The nodes in $Cache_2$ and in *Read* are shuffled according to the following permutation $\pi$: 201's content moves to block 209; 203's to 206; 206's to 201; 209's to 217; and 217's to 203. Node 206 (i.e., *target_id*) is updated inserting the new tuple with key value 'Fc' and inserted into $Cache_2$ and node 217 (which we suppose the least recently used) is pushed out. The pointers in the nodes accessed during the previous iteration (i.e., 101, 102, 103, 104, 105) are updated according to $\pi$, encrypted, and sent to the server.

Finally, accessed leaf nodes (i.e., 201, 203, 206, 209, 217) are encrypted and sent to the server. Figure 5 shows the evolution of the shuffle index. In the figure, gray boxes represent blocks that have been added by the split operations performed during the algorithm execution.

## 8. CORRECTNESS AND COMPLEXITY OF SHUFFLE INDEX MANAGEMENT

We analyze the correctness and computational complexity of the algorithm in Figure 2 extended to support the probabilistic split of visited nodes (for the detailed pseudocode, see Appendix B and Appendix C). To prove the correctness of the algorithm, we need to prove that it correctly performs the search of the target value and that

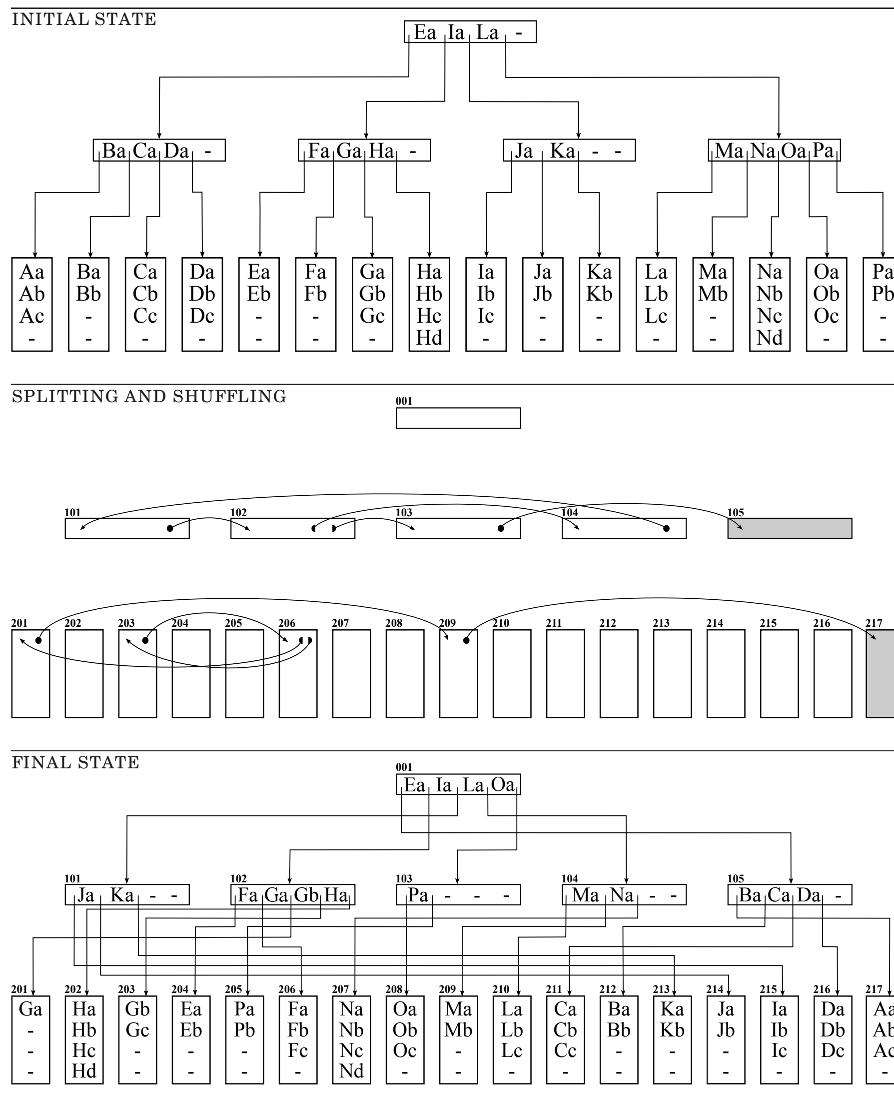



Fig. 5. Evolution of the shuffle index for Example 7.4

it preserves the correctness of the shuffle index and cache structures, as stated by Theorems 8.1–8.3 below (the proofs of the theorems are in Appendix D).

We start by proving correctness of the search operation (Theorem 8.1).

THEOREM 8.1 (RETRIEVAL CORRECTNESS). *Let $\mathcal{S}$ be a shuffle index built on candidate key $K$ with domain $\mathcal{D}$ and $target_value$ be a value in $\mathcal{D}$. The algorithm in Figure 2 returns the unique leaf node where $target_value$ is (or should be) stored.*

Theorem 8.2 proves that after the execution of a search, removal, or insert operation, the shuffle index still represents a correct unchained $B+$-tree defined on the same set of data, possibly with the addition of the target of the executed operation if it corresponds to an insert.

THEOREM 8.2 (SHUFFLE INDEX CORRECTNESS). *Let $\mathcal{S}$ be a shuffle index representing an unchained $B$+-tree built on candidate key $K$ defined over domain $D_K$, $\mathcal{D}\subset D_K$ be the set of key values stored in the unchained $B$+-tree, and $target_value$ be a value in $D_K$. After the execution of an access operation on $\mathcal{S}$ by the algorithm in Figure 2 with target value $target_value$, $\mathcal{S}$ is a shuffle index representing an unchained $B$+-tree defined on $\mathcal{D}$ possibly extended with $target_value$ if the access is inserting $target_value$.*

Theorem 8.3 proves that the algorithm correctly manages the cache stored at the client-side, that is, the cache satisfies Definition 4.2.

THEOREM 8.3 (CACHE CORRECTNESS). *Let $\mathcal{S}$ be a shuffle index and $Cache_0, \ldots, Cache_h$ be a cache (Definition 4.2). After the execution of an access operation on $\mathcal{S}$ by the algorithm in Figure 2, $Cache_0, \ldots, Cache_h$ satisfies Definition 4.2.*

We conclude this section by noting that the algorithm in Figure 2 works in polynomial time in the height of the shuffle index, and in the number of cover and cached searches. Intuitively, each access to the shuffle index implies the visit of $num_cover$ + $num_cache$ + 1 disjoint paths in parallel: one for the target of the access, $num_cover$ for cover searches, and $num_cache$ for cached paths. If no node is split, an access to a shuffle index then costs $O((num_cover + num_cache)\log_F(m))$ as $h = \log_F(m)$. Split operations impact access time as the split of a node requires to update its parent. Since nodes in the shuffle index do not have a direct reference to their parent, the update of the parents of shuffled nodes costs, in the worst case, $O(num_cover + num_cache + 1)$. In fact, it is necessary to search for the parent of the split node among the nodes in cache or read for the previous level in the shuffle index. As a consequence, in the worst case (i.e., if each visited node is split) the cost of the algorithm in Figure 2 becomes $O((num_cover + num_cache)^2 \log_F(m))$, as formally stated by the following theorem, whose proof is in Appendix D.

THEOREM 8.4 (COMPUTATIONAL COMPLEXITY). *The algorithm in Figure 2 operates in $O((num_cover + num_cache)^2 \log_F(m))$ time, where $m$ is the number of blocks in the shuffle index and $F$ is its fan out.*

Our experiments in a representative configuration scenario (see Section 10) show that the overall performance overhead caused by the adoption of our protection techniques remains limited (≈20% with respect to a plain encrypted index assuming $num_cover$=1 and $num_cache$ between 1 and 2). Also, as discussed in Section 10, configurations with $num_cover$=1 and $num_cache$=2 provide sufficient guarantees of both access and pattern confidentiality.

## 9. PROTECTION ANALYSIS

In this section, we analyze the confidentiality guarantees offered by our shuffle index.

Before diving into the discussion of confidentiality issues, we would like to make a note on a major guarantee of our proposal (which is not satisfied instead by alternatives like Practical ORAM [Stefanov et al. 2013] and Path ORAM [Stefanov et al. 2013]), which is to always leave, at the end of each access, a consistent representation of the data stored at the server. We believe this to be a critical requirement that needs to be considered to permit access to the shuffle index by different clients as well as guarantee reliability and recoverability of the system.

In the remainder of this section, we first formally prove that our approach guarantees both access and pattern confidentiality (Section 9.1), and then that the storage server is not able to distinguish the kind of operation executed on the shuffle index (Section 9.2). Intuitively, these guarantees are a consequence of the shuffling, which degrades the possible knowledge that an observer has on the correspondence between

nodes and blocks where they are stored, and of the probabilistic split, which has the effect of making the different operations indistinguishable. Finally, we present an experimental evaluation showing that the theoretical results illustrated actually hold in practice (Section 9.3).

### 9.1. Access and Pattern Confidentiality

We first analyze how shuffling degrades the information on the correlation between nodes and blocks. We then describe how the shuffle index supports access and pattern confidentiality (content confidentiality as well as integrity of the outsourced data are guaranteed by encryption). For simplicity and without loss of generality, our analysis will consider only the leaf blocks accessed at each request, since leaves are more exposed than internal nodes, which, representing only summary information on the descendants, are more protected. We will also consider the worst case assumption that no accessed node is split since the splitting increases the number of shuffled nodes, providing higher protection.

*Degradation due to shuffling.* The continuous shuffling, which occurs at every access, is able to degrade any information the server may possess on the correspondence between nodes and blocks, reaching, after a sufficient number of accesses, a complete loss of information. This result shows an interesting feature of the shuffle index behavior and the absence of long term accumulation of information. Let $\mathcal{N} = \{n_1, \ldots, n_m\}$ and $\mathcal{B} = \{b_1, \ldots, b_m\}$ be the set of leaf nodes in the unchained $B$+-tree and the set of leaf blocks in the shuffle index, respectively. The knowledge that the server has on the correspondence between nodes and blocks storing their encrypted content can be modeled as the probability $\mathcal{P}(n_i, b_j)$ that the server can establish an association between a node $n_i \in \mathcal{N}$ and a block $b_j \in \mathcal{B}$, with $i, j = 1, \ldots, m$.

We now analyze how the application of cover searches and shuffling techniques at each access progressively destroys any information the server may have acquired on the correspondence between nodes and blocks, thus providing access confidentiality. In this analysis, we consider the worst case assumption that no cache is maintained (only to simplify the analysis) and that the server, at initialization time, knows the block where each node is stored. We first prove that the maximum value of probability $\mathcal{P}(n_i, b_j)$ for each node $n_i$ and each accessed block $b_j$ decreases at each access, thanks to shuffling.

THEOREM 9.1 (KNOWLEDGE DEGRADATION). *Let $\mathcal{S}$ be a shuffle index representing an unchained $B$+-tree, $\mathcal{N}$ be the set of leaf nodes in the unchained $B$+-tree, $\mathcal{B}$ be the set of leaf blocks in $\mathcal{S}$, $\mathcal{P}(n_i, b_j)$ be the probability estimated by the server that node $n_i$ is stored in block $b_j$, and $\mathcal{B}_a = \{b_{q_1}, \ldots, b_{q_{num_cover+1}}\} \subset \mathcal{B}$ be the set of leaf blocks involved in an access. After the search operation, for each $n_i \in \mathcal{N}$, the maximum value of probability $\mathcal{P}(n_i, b_j)$, with $b_j \in \mathcal{B}_a$:* i) *does not change, if $\mathcal{P}(n_i, b_j)$=$\mathcal{P}(n_i, b_k)$ for all $b_j$,$b_k \in \mathcal{B}_a$;* ii) *decreases, otherwise.*

PROOF. After the access is executed, for all nodes $n_i \in \mathcal{N}$, and all blocks $b_{q_j} \in \mathcal{B}_a$, the probability $\mathcal{P}(n_i, b_{q_j})$ becomes equal to $(\sum_{j=1}^{num_cover+1} \mathcal{P}(n_i, b_{q_j}))/(num_cover + 1)$ due to the shuffling. In fact, thanks to re-encryption with a different salt, the encrypted representation of shuffled nodes always changes at each access. Then, an observer is not able to determine to which block $b_{q_j} \in \mathcal{B}_a$ each node $n_i \in \mathcal{N}$ has been moved. The maximum value among $\mathcal{P}(n_i, b_{q_j})$, with $b_{q_j} \in \mathcal{B}_a$, then decreases. It does not change only if $\forall b_{q_j}, b_{q_k} \in \mathcal{B}_a$, $\mathcal{P}(n_i, b_{q_j})$=$\mathcal{P}(n_i, b_{q_k})$. □

Note that $\mathcal{P}(n_i, b_j)$ remains unaltered for all the blocks that are not involved in the access (i.e., blocks in $\mathcal{B} \setminus \mathcal{B}_a$). For instance, assume that before the ac-

cess: $num_cover$=3; $\mathcal{B}_a$={301, 302, 303, 304}; and $\mathcal{P}(n_i, 301)$=0.1, $\mathcal{P}(n_i, 302)$=0.4, $\mathcal{P}(n_i, 303)$=0.2, and $\mathcal{P}(n_i, 304)$=0.1. After the access, each probability would be equal to ($\mathcal{P}(n_i, 301)$+$\mathcal{P}(n_i, 302)$+$\mathcal{P}(n_i, 303)$+$\mathcal{P}(n_i, 304)$)/4=0.2, due to the random shuffling. If $\exists b_k \notin \mathcal{B}_a$ s.t. $\mathcal{P}(n_i, b_k) \geq \mathcal{P}(n_i, b_j)$, $\forall\ b_j \in \mathcal{B}$, then the overall maximum probability does not change and is equal to $\mathcal{P}(n_i, b_k)$. If the previous condition is not satisfied and, on the contrary, $\exists b_{q_k} \in \mathcal{B}_a$ s.t. $\mathcal{P}(n_i, b_{q_k}) \geq \mathcal{P}(n_i, b_j)$, $\forall\ b_j \in \mathcal{B}$, then $\mathcal{P}(n_i, b_{q_k})$ decreases and therefore also the maximum probability decreases. Note that, as already noted, this probability does not decrease only if $\mathcal{P}(n_i, b_{q_j})$=$\mathcal{P}(n_i, b_{q_k})$ for all $b_{q_j}$,$b_{q_k} \in \mathcal{B}_a$, but due to the random choice of covers, this will typically happen only when $\mathcal{P}(n_i, b_j)$=$\mathcal{P}(n_i, b_k)$ for all $b_j$,$b_k \in \mathcal{B}$, that is, when the server has no knowledge on the node/block association.

We apply classical concepts of information theory to model the overall degree of uncertainty of the server about the block containing a node. For each node $n_i \in \mathcal{N}$, such uncertainty is measured through the *entropy*, denoted $\mathcal{H}_{n_i}$, applied on the probabilities $\mathcal{P}(n_i, b_j)$, for all $b_j \in \mathcal{B}$. Formally, $\mathcal{H}_{n_i}$= $-\sum_{j=1}^{m} \mathcal{P}(n_i, b_j) \log_2 \mathcal{P}(n_i, b_j)$. Note that, $\mathcal{H}_{n_i}$=0 means that the server knows exactly the block storing node $n_i$; on the other hand, $\mathcal{H}_{n_i}$=$\log_2 m$ means that the server has complete uncertainty about the correspondence, since $\mathcal{P}(n_i, b_j) = 1/m$, $j = 1, \ldots, m$. Indeed, the knowledge of the server about the block containing node $n_i$ (corresponding to the block $b_x$ s.t. $\mathcal{P}(n_i, b_x) \geq \mathcal{P}(n_i, b_j)$, $\forall b_j \in \mathcal{B}$) decreases progressively at each access, due to shuffling. The entropy remains constant only when all the accessed blocks have equal probability. The following corollary proves this property.

COROLLARY 9.2 (ENTROPY INCREASE). *Let $\mathcal{S}$ be a shuffle index representing an unchained B+-tree, $\mathcal{N}$ be the set of leaf nodes in the unchained B+-tree, $\mathcal{B}$ be the set of leaf blocks in $\mathcal{S}$, and $\mathcal{B}_a = \{b_{q_1}, \ldots, b_{q_{num_cover+1}}\} \subset \mathcal{B}$ be the set of leaf blocks involved in an access. After the search operation, $\forall n_i \in \mathcal{N}$, $\mathcal{H}_{n_i}$ does not change, if $\mathcal{P}(n_i, b_j)$=$\mathcal{P}(n_i, b_k)$ for all $b_j$,$b_k \in \mathcal{B}_a$; increases, otherwise.*

As an example, suppose that the server knows, at a given time $t_0$, the exact correspondence between a node $n_i \in \mathcal{N}$ and the block $b_j \in \mathcal{B}$ storing it, meaning that $\mathcal{P}(n_i, b_j)$=1; $\mathcal{P}(n_i, b_y)$=0, for all $y = 1, \ldots, m$, with $y \neq j$, and $\mathcal{H}_{n_i}$=0. Suppose now that there is an access and that among the $num_cover + 1$ blocks accessed there is block $b_j$. Because of the shuffling, when the access is terminated, the observer does not know which of the $num_cover + 1$ written blocks contains node $n_i$. The probability that $n_i$ is stored in one of the $num_cover + 1$ written blocks becomes $1/(num_cover + 1)$ while it remains equal to zero for all the other blocks. As a consequence, the value of $\mathcal{H}_{n_i}$ increases for each node $n_i \in \mathcal{N}$. We note that the increase in $\mathcal{H}_{n_i}$ depends on the set of blocks accessed by each search operation. As shown by the experimental results in Section 9.3, after a limited number of accesses the value of entropy starts to increase and quickly reaches its maximum, even for scenarios where initially the choice of blocks tends to produce lower increases.

*Access confidentiality.* Access confidentiality is characterized as the protection against the server ability to associate a specific access request with a specific node/data. Static encrypted indexing structures do not exhibit access confidentiality, because the server may exploit information on the frequency of accesses (e.g., the server may know that people last names are used as key and "Smith" is the most frequently accessed value) and may thus identify the content associated with a specific block.

The shuffle index offers a natural protection against this attack. Even disregarding the caching and considering only the contribution offered by covers, every time an access is performed any information on the specific access has to be divided among all the $num_cover + 1$ nodes involved in the access request. After the nodes are shuffled, the

information on the correspondence between nodes and blocks is further destroyed. In general, we observe here a reenforcing mechanism: access confidentiality is typically at risk when there are values that are characterized by high access frequency, but the higher the access frequency, the greater the destruction of information realized by shuffling. An experimental verification of this aspect is provided by our experiment that shows how extremely different target distributions produce almost identical block access profiles (Section 9.3).

*Pattern confidentiality.* Pattern confidentiality is characterized as the protection against the server ability to recognize that two separate accesses refer to the same node. We first consider a generic scenario, for which we quantify the minimum level of protection offered by the shuffle index. We then extend the analysis to the consideration of patterns separated by a number of steps smaller than the size of the cache. We can observe that the degradation of information that derives from shuffling guarantees that accesses separated by a significant number of steps will not be recognizable.
*1) Protection by covers and shuffling*. To simplify the analysis, here we suppose that the cache is not used. The server observes two consecutive requests that translate into accesses to the following two sets of leaf blocks: $\{b_{i_1}, \ldots, b_{i_{num_cover+1}}\}$ and $\{b_{j_1}, \ldots, b_{j_{num_cover+1}}\}$, respectively (non-consecutive requests are characterized by better protection). Two cases may occur: *i)* the two sets do not have any block in common, or *ii)* there is (at least) one block that appears in their intersection. In the first case, there is no repeated access and therefore no pattern to protect. In the second case, there is the possibility that the two accesses represent a repeated access to the same target node content. By the cover/target indistinguishability, the probability that the intersection identifies a repeated access is $1/(num_cover + 1)^2$. We observe that the consideration of patterns presenting a greater number of accesses (i.e., the identification of $z$ accesses to the same node) will be characterized by a probability decreasing at a geometric rate (i.e., a sequence of $z$ accesses presenting a non-empty intersection will be due with probability $1/(num_cover + 1)^z$ to the execution of $z$ accesses to the same node). The server then cannot use the information on the accessed blocks to recognize accesses to the same nodes.
*2) Protection by caching*. Considering a worst case scenario where the server knows which are the nodes in the cache before the $i$-th access (with $i \leq num_cache$), pattern confidentiality is violated if the server can determine whether the $i$-th target access refers to the same node $n_1$ as the first access or not. The following theorem formally proves that, under the cover/target indistinguishability hypothesis (see Section 9.3), pattern confidentiality is fully protected.

THEOREM 9.3 (SAME ACCESS UNRECOGNIZABILITY). *Let $\mathcal{S}$ be a shuffle index representing an unchained B+-tree, $\mathcal{N} = \{n_1, \ldots, n_m\}$ be the set of leaf nodes in the unchained B+-tree, $Cache_0, \ldots, Cache_h$ be the cache, and $\{n_{j_1}, \ldots, n_{j_i}\}$ be a sequence of accessed (target) nodes, with $i \leq num_cache$. Even if the server knows the content of the cache before the $i$-th access, the server cannot infer whether $n_{j_i} = n_{j_1}$.*

PROOF. The proof must consider two cases.

**Case 1:** $n_{j_1}$ and $n_{j_i}$ are both in cache. Both accesses would correspond to a cache hit; then, the $num_cover + 1$ blocks accessed at the $i$-th access $\{b_{i_1}, \ldots, b_{i_{num_cover+1}}\}$ are all covers and there is no information gained from knowledge of the cache and of the accessed blocks that can help in identifying if the access has been to $n_{j_1}$ or to $n_{j_i}$.

**Case 2:** $n_{j_1}$ is in cache and $n_{j_i}$ is not in cache. If it is $n_{j_1}$ that is accessed, a cache hit is generated and the $num_cover + 1$ blocks $\{b_{i_1}, \ldots, b_{i_{num_cover+1}}\}$ accesses at time $i$ are all covers. If it is $n_{j_i}$ that is accessed, a cache miss is produced and the accessed blocks will represent $num_cover$ covers and one target access to the physical node containing

$n_{j_i}$. Since the server ignores where $n_{j_i}$ is stored, the server cannot recognize if $n_{j_1}$ or $n_{j_i}$ is accessed since, by hypothesis, the server is not able to distinguish cover searches from targets. □

Proving protection in the scenario considered by the above theorem, where uncertainty is reduced to a single bit, implies that pattern confidentiality holds in any scenario, where the server has access to less knowledge. We can then conclude that the shuffle index fully protects pattern confidentiality when the distance between the observations is within the size of the cache. As a consequence, we can also conclude that the evaluation of range queries adopting the approach discussed in Section 6 guarantees their indistinguishability from a sequence of arbitrary accesses to the shuffle index. Indeed, the distance between the accesses in the sequence is within the size of the cache, for any $num_cache \geq 1$, and presents repeated accesses to internal nodes.

### 9.2. Indistinguishability of Accesses

In this section, we prove that our approach to insert new tuples in the shuffle index does not reveal to the server whether an access is searching for a value or inserting a tuple. To this purpose, we prove that the server cannot infer whether a value has been inserted into a node by simply observing whether the node has been subject to a split during the access to the shuffle index. This is formally proved by the following theorem.

THEOREM 9.4 (INSERT UNRECOGNIZABILITY). *Let $\mathcal{S}$ be a shuffle index representing an unchained $B$+-tree, and $n$ be a node in the unchained $B$+-tree. Read and Insert operations are indistinguishable since:*

(*1*) *even if the server can recognize the split of a node $n$, generating nodes $n_i$ and $n_j$, and a subsequent split operating on $n_i$, the server cannot infer whether a value has been inserted into $n_i$ or not (the case for $n_j$ is symmetric);*
(*2*) *even if the server can recognize that an access did not split node $n$, the server cannot infer whether a value has been inserted into $n$.*

PROOF. We now prove each of the conditions in the theorem.

*(1)* Let us assume that the server can recognize the split of a node $n$, generating nodes $n_i$ and $n_j$; and a subsequent split operating on $n_i$, generating nodes $n_k$ and $n_l$. (Note that this is a worst case scenario as the server does not know which are the blocks storing $n_i$ and $n_j$ after the split of $n$, thanks to shuffling.) We now prove that this scenario can happen also if no value has been inserted into $n_i$. To this purpose, let us assume that $n$ is full, that is, it stores $F-1$ values. Its split generates two nodes, $n_i$ and $n_j$, storing each either $\lfloor \frac{F-1}{2} \rfloor$ or $\lfloor \frac{F-1}{2} \rfloor - 1$ values. Let us now assume that the threshold $t$ fixed by the client is lower than or equal to $\lfloor \frac{F-1}{2} \rfloor$. As a consequence, the probability $\wp(n_i)$ ($\wp(n_j)$, resp.) of splitting node $n_i$ ($n_j$, resp.) when visiting it is not null, even if it has been obtained splitting another node $n$. Therefore, it may be split by the next access to the index, although no value has been inserted into the node.

*(2)* The second property immediately follows from the observation that not all the insert operations cause a node split. Indeed, if the nodes along the path to the leaf where the new tuple should be stored are not full, they can accommodate the new value without splits. On the other hand, in our approach, search operations may cause the split of visited nodes. □

This theorem proves that the split of a node and the insertion of a tuple into the same node are independent events. In fact, the number of accesses occurring between the insertion of a tuple into a node and the split of the same is arbitrary.

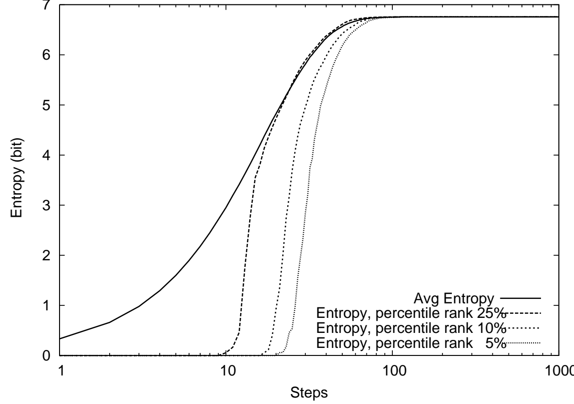



Fig. 6. Uncertainty $\mathcal{H}_n$ on the block where a given node $n$ is stored

We note that the theorem above considers a worst case scenario that cannot happen adopting a shuffle index for the management of a data collection. Indeed, as already proved in Section 9.1, the server is not able to recognize two subsequent accesses to the same node, thanks to shuffling. Analogously, it is not able to infer whether an access splits one of the nodes resulting from a previous split. Furthermore, our approach possibly splits nodes along the path to the target value, but also nodes along the path to cover searches and nodes in cache. Since the server is not able to distinguish cover searches from the target search, it cannot precisely determine which of the nodes accessed during an access has been split, and therefore it cannot determine whether an access to the shuffle index performs an insert or a search operation.

To strengthen the fact that the server is not able to infer which operation is being executed by an access to the shuffle index, we formally analyze the probability of splitting a node, considering a sequence of accesses visiting it.

THEOREM 9.5 (PROBABILITY OF SPLITTING A NODE). *Let $\mathcal{S}$ be a shuffle index representing an unchained $B$+-tree, and $n$ be a node in the unchained $B$+-tree. The probability that the $k$-th access to $n$ splits the node is* $\wp(n) \cdot (1 - \wp(n))^{k-1}$.

PROOF. The probability that an access to a node splits it is $\wp(n)$. Since the accesses in the considered sequence are independent, the probability that the $k$-th access splits the node is obtained multiplying the probability that the first $k-1$ accesses do not split the node by the probability that the $k$-th access splits it, that is, $\wp(n) \cdot (1 - \wp(n))^{k-1}$. □

Recall that the probability $\wp(n)$ of splitting node $n$ when visiting it changes (increases) every time a new value is inserted into the node (Definition 7.3). Also, the nodes resulting from a split have probability of being split that is lower than the original node, since they store at most half the values in the original node.

### 9.3. Experimental Results

To assess the protection guaranteed by our algorithm, we implemented it in Java and evaluated: *1)* the rate of decrease of the server's knowledge about the correspondence between nodes and blocks due to cover searches and shuffling (supporting access confidentiality), and *2)* the degree of similarity of block access profiles, showing that different accesses exhibit the same behavior at the block level, and therefore are not distinguishable by the server (supporting pattern confidentiality).

*Entropy evolution.* In the experiment for evaluating knowledge degradation by the server, we considered a shuffle index with 100 leaf blocks. We started from a configuration where the server knows the block $b_i$ where a leaf node $n_i$ is stored (i.e., $\mathcal{P}(n_i, b_i) = 1$). We applied a sequence of random accesses with $num\text{-}cover$=10 and $num_cache$=0 and measured entropy $\mathcal{H}_{n_i}$. The idea is to simulate random accesses to the blocks. The entropy $\mathcal{H}_{n_i}$ increases when the probability $\mathcal{P}(n_i, b_j)$ of a read block $b_j$ is greater than zero. From the starting configuration, this entropy increase will happen only when $b_i$ is read. Intuitively, as soon as $b_i$ is read, the probability is distributed over a larger number of blocks (those accessed together with $b_i$), increasing the probability of a further entropy increase. Figure 6 illustrates the average entropy (over 5000 experiments) and the value of the entropy for the 25th, 10th, and 5th bottom percentile (the $p$-th bottom percentile is the case that exhibits a value of $\mathcal{H}_{n_i}$ worse than 100(1$-p$)% of the simulations), as a function of the number of accesses. As the graph shows, $\mathcal{H}_{n_i}$ progressively increases (with a rate increasing with $num_cover/m$, where $m$ is the number of blocks) and after at most $m$ accesses it reaches its maximum value (i.e., $\log_2 m$), meaning that the server's knowledge about the correspondence between nodes and blocks has been completely destroyed.

*Block access profiles.* In the experiment to evaluate the indistinguishability of access profiles, we simulated different access profiles by synthetically generating index values that follow a *self-similar* probability distribution with *skewness* $\gamma$ in the range $[0, 0.5]$ [Gray et al. 1994]. Given a domain of cardinality $d$, a self-similar distribution with skewness $\gamma$ provides a probability equal to $1-\gamma$ of choosing one of the first $\gamma d$ domain values. The same proportion holds when considering also any sub-range of the domain values. For instance, a value $\gamma = 0.5$ generates a sequence of domain values that follows a uniform probability density function. Figure 7 illustrates the rank/frequency distribution of block identifiers corresponding to three access profiles, generated extracting 1000 target values from three self-similar distributions with $\gamma = 0.5$ (50-50 rule), $\gamma = 0.25$ (75-25 rule), and $\gamma = 0.10$ (90-10 rule), respectively. On the x-axis, we report, in decreasing order of frequency, the rank number assigned to the blocks accessed with the same frequency (i.e., the first value on the x-axis corresponds to the block accessed more frequently). The figure shows that our approach makes the frequency distributions of accesses to the blocks so close to become statistically indistinguishable, even if the considered logical access profiles are extremely different. Since the statistical profile of accesses to index values is always near to that produced by the uniform distribution, the use of simple statistical models to support the choice of covers easily satisfies the indistinguishability property.

## 10. PERFORMANCE ANALYSIS

To assess the performance of our algorithm, we used a data set of 2 GiB stored in the leaves of a shuffle index with 3 levels, built on a numerical candidate key $K$ of fixed-length, with fan out 512, and representing $2^{18}$ different index values. The size of the nodes of the shuffle index was 8 KiB. The hardware used in the experiments included a server machine with an Intel Core i7-920 CPU at 2.6 GHz, L3–8 MiB, 12 GiB RAM DDR3 1066, 120 GiB SSD disk SATA III with read throughput 240 MiB/s, write throughput 220 MiB/s, running an Ubuntu operating system with the ext4 file system. The client machine was running an Intel Core i5-2520M CPU at 2.5 GHz, L3–3 MiB, 8 GiB RAM DDR3 1066, running an Arch Linux operating system with the ext4 file system. We considered both a scenario where the client and the server operate in a wide area network, by properly simulating adequate network configurations using professional-grade tools, and a scenario where they operate in a local area network. The implementation of the algorithm in Figure 2 that accesses the shuffle index has

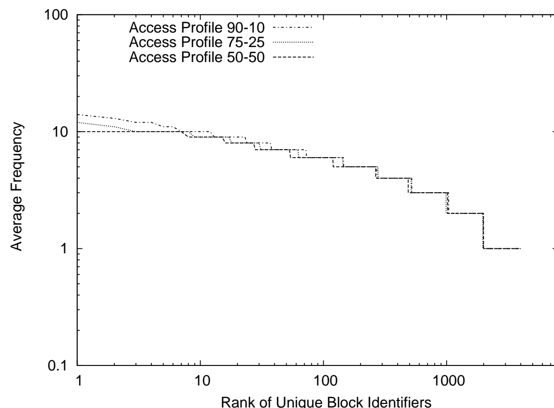



Fig. 7. Rank / Frequency distribution of leaf block identifiers ($num_cover$=4, $num_cache$=4)

been optimized to minimize the number of interactions between the client and the server. The performance analysis started after the system had processed a significant number of accesses, to be in a steady state. We generated access requests according to a uniformly random profile of the target values. Also, we considered the management of both read accesses and of updates to the dataset stored in the leaves of the shuffle index. As a consequence, each access to a node in the shuffle index (either to search for a value or to modify the dataset) may cause its split (see Section 7). We will therefore describe the effects, in terms of performance overhead, of our probabilistic split approach with respect to the base scenario where the outsourced data collection never changes. In the following, we first analyze the performance of the algorithm in Figure 2, extended with probabilistic node split, and we then evaluate the computational overhead caused by the evaluation of range queries (Section 10.2).

### 10.1. Single Access Request

To evaluate the performance of the shuffle index we took into consideration the cost of: CPU, disk, and network.

*CPU.* The computational load required for the management of the shuffle index is quite limited. The algorithm uses only symmetric encryption and a MAC function; the execution times we measured on a 8 KiB block for both cryptographic functions are under 100 $\mu$s, a negligible fraction of the time required by network and disk accesses, which then drive the performance of the shuffle index.

*Disk.* We analyzed the performance of the shuffle index when the client and the server operate in a local area network (we used a 100 Mbps switched Fast Ethernet network with average Round Trip Time - RTT equal to 0.67 ms). In this configuration, disk performance becomes the limiting factor. Figure 8 reports observed times in milliseconds. The values are grouped by the same value of $num_cover$, between $1$ to $10$, varying the value of $num_cache$ from $1$ to $10$. The access time grows linearly with the number of cover searches, since every additional cover requires to traverse an additional path in the shuffle index; the depth of the tree remains constant and the quadratic element in the complexity analysis in Theorem 8.4 does not apply. Although an increase in $num_cache$ causes a growth in the number of blocks written for each level of the shuffle index, the number of cached nodes has a smaller impact on the access time. This is justified by the fact that the disk operations caused by the increase in $num_cache$ greatly benefit from buffering and cache mechanisms at the

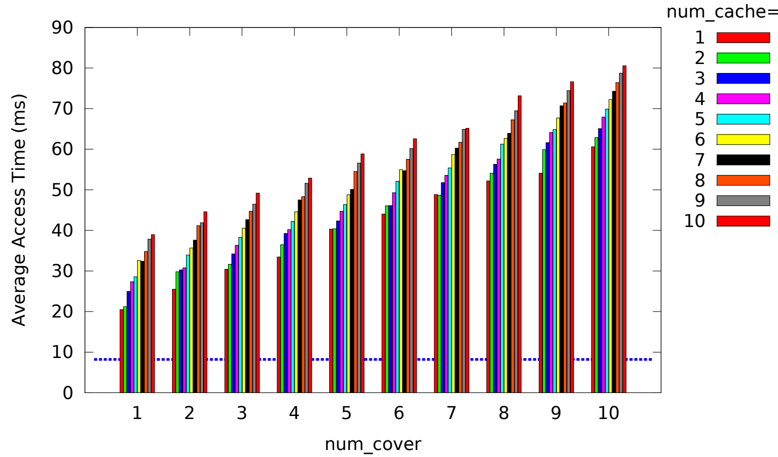
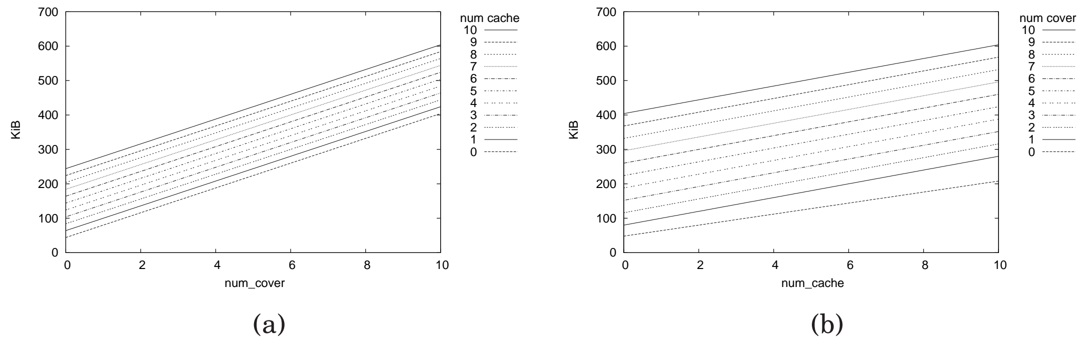



Fig. 8. Access time of the shuffle index in a switched Fast Ethernet LAN as a function of the number of covers. The dashed line (baseline) represents the service time observed using a plain encrypted index with the same height.

operating system and disk controller level, which currently provide significant performance improvements on high capacity storage. In particular, the cost of the repeated write operations on cached blocks will be lower than that associated with cover and target blocks. We claim that, as it is typical for database index structures, the bottleneck in the performance of the shuffle index in a Local Area Network (LAN) scenario is the number of read and write operations on the local disk. The best performance is obtained when using a single cover and a single cache; increasing the number of covers there is an impact on performance, but in every tested configuration the average access time was below 80 ms.

The dashed line in Figure 8 represents our baseline. It has been obtained measuring the access time of a plain encrypted index with the same static structure of the shuffle index (this is essentially the tree structure that was proposed in [Damiani et al. 2003]). The adoption of a plain encrypted index still requires the client to visit the nodes in the tree level-by-level, but it does not use covers, caching, and shuffling to provide access and pattern confidentiality. The comparison with a plain encrypted index, which provides content confidentiality, permits to evaluate the specific overhead caused by the adoption of the techniques to protect access and pattern confidentiality. The performance overhead introduced by the adoption of our protection techniques ranges from $\times 2.5$ to $\times 10$ of the baseline, depending on the number of covers and on the size of the cache. This increase in the access time is mainly due to the disk cost caused by a higher number of (random) read/write operations.

To assess the performance overhead caused by probabilistic node split, we compared the access time when visited nodes are not split and when a subset of the visited nodes are split. The split of a subset of the nodes visited while accessing the shuffle index implies the transmission of additional nodes (i.e., one for each node that has been split) to the server. This causes an increase in the measured access time of $\approx 1.3(num_cache + num_cover)$ ms, depending on the number of nodes that have been split. It is however interesting to note that access times of requests that are evaluated after a node split largely benefit from the split since, once a node split is performed, the probability of a further split of the same node (in particular, if it is an internal one) quickly decreases. Hence, the slowdown of the measured access time due to a split operation is distributed on the subsequent requests.




(a) (b)

Fig. 9. Number of KiB exchanged as a function of the number of covers (a), and of the size of the cache (b)

The approach described in [Stefanov and Shi 2013] (which is discussed in the related work) shows an access response time comparable with the one provided by our technique. However, with respect to network bandwidth overhead, the performance analysis in [Stefanov and Shi 2013] reports a 40x–50x penalty with respect to the baseline represented by performing an unprotected access (i.e., for accessing one data block, on average, 40–50 data blocks need to be accessed). By contrast, the shuffle index incurs a lower I/O overhead compared to the unprotected version, which is equal to $(2{+}2{\cdot}num_cover{+}num_cache)$.

*Network*. To better analyze how the network impacts on the time necessary to access the shuffle index, we first study the number of messages and the number of nodes (which corresponds to the number of bytes) exchanged between the client and the server during each access. Given a shuffle index of height $h$, the client and the server exchange $2h+1$ messages for the evaluation of each access. In fact, for each level in the shuffle index but the root level, the client sends a request for a set of blocks, and the server replies with their content. The client also sends to the server, together with her request for blocks at level $l$, also the blocks to be rewritten at level $l-1$. The visit of each level then implies the exchange of two messages. The root level, on the contrary, implies no message exchange as the root node is in cache. An additional message is finally required to write the leaves of the shuffle index.

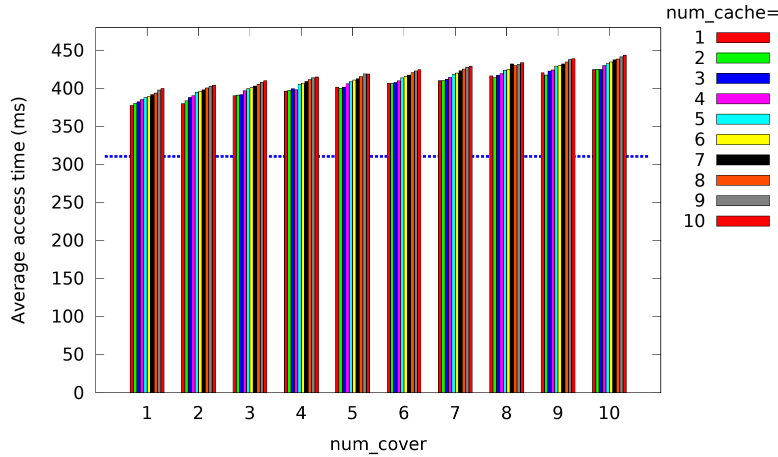
The number of blocks downloaded from the server for each level of the shuffle index (except for the root level) is always equal to $1 + num_cover$. The number of blocks sent to the server (i.e., written back) instead possibly varies as a consequence of split operations. If no accessed node is split, the number of written blocks is $1 + num_cover + num_cache$ for each level, but the root level that requires to write one block only. The split of one among the accessed nodes causes the need to write one additional block. Hence, the number of written blocks can be computed as $(1 + num_cover + num_cache) \cdot (1 + \bar{\wp}(n))$, with $\bar{\wp}(n)$ the expected value of $\wp(n)$ (which follows a Bernoulli distribution as the split of different nodes during a same access are independent events). Considering a single access to the shuffle index, the local expected value of $\wp(n)$ at the time of the access is $\bar{\wp} = \frac{F-1-t}{2(F-1)}$, therefore the overall number of blocks exchanged during each access is, on average, equal to $1+h(2+2num_cover+num_cache+\bar{\wp}(1+num_cover+num_cache))$. Figure 9 illustrates the average number of bytes exchanged during an access, considering a shuffle index with $F$=512, $h$=2, blocks of 8KiB, and $t$=256 and varying the number of cover searches and of nodes in cache for each level of the shuffle index between 0 and 10.

We analyzed the performance of the shuffle index when the client and the server operate in a Wide Area Network (WAN). This scenario, where a client uses a remote

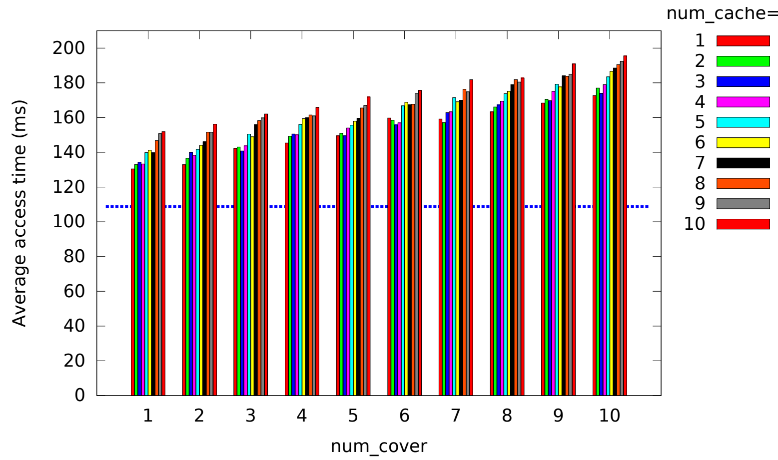
untrusted party for the private access to data, is the most interesting and natural for the shuffle index. We adopted a network configuration suitable for interactive traffic between the client and the server, in contrast to configurations where network connections are sized to better support database replication or database backup operations. To properly configure the network used for our experiments, we adopted a professional-grade tool suite (i.e., *Traffic Control* and *Network Emulation*, for Linux systems), which permits to tune the networking configuration through a granular control over the queuing systems and mechanisms by which packets are received, transmitted, and reordered. We chose two representative WAN configurations with LAN-like bandwidth. The first configuration is characterized by round trip time typical of US east-coast to Europe connections (modeled as a normal distribution with mean of 100 ms and standard deviation of 2.5 ms). The second configuration is characterized by round trip time typical of continental connections (modeled as a normal distribution with mean of 30 ms and standard deviation of 2.5 ms). In both configurations, network performance becomes the limiting factor. Figures 10(a) and 10(b) report observed times in milliseconds for the US-to-Europe connection and for the continental connection, respectively. The values are grouped by the same value of $num_cover$, between 1 and 10, varying the value of $num_cache$ from $1$ to $10$. As it is visible from the figure, for every tested pair of $num_cover$ and $num_cache$ parameters, the average access time was below 450 ms for the US-to-Europe connection, and below 200 ms for the continental connection. In both scenarios, the average access time to the shuffle index mostly depends on the number of send and receive operations on the network channel between the client and the server. Therefore, the performance overhead caused by an additional cover search is greater than the overhead caused by an additional cached search. Indeed, cached nodes do not need to be downloaded from the server at each access (see Figure 2) as they are locally stored at the client side.

Like for the LAN scenario, we compared the access time of our shuffle index with the access time of a plain encrypted index with the same static structure of the shuffle index, which is represented by the dashed lines in Figures 10(a) and 10(b). We note that the performance of the shuffle index scales much better in a WAN configuration, especially in the US-to-Europe connection configuration, than in a LAN scenario. Indeed, the performance overhead caused by the adoption of our protection techniques ranges from $\times 1.2$ up to $\times 1.8$, depending on the round trip time of the WAN connection, on the number covers, and on the size of the cache. More precisely, from the results obtained in the US-to-Europe configuration, we can conclude that each increase in the number of covers and cache searches adds 1.2% and 0.6% (4.2% and 2.1% in the continental scenario), respectively, of the plain encrypted access time to the overall performance. We note that configurations with $num_cover$=1 and $num_cache$ between 1 and 2 already provide a strong degree of access and pattern confidentiality and cause a limited performance overhead. The choice of $num_cover$=1 permits at each access to shuffle the position of the target node with a randomly chosen node; larger numbers of covers would offer a faster degradation of the information the server may obtain from the access, but each cover also requires an additional node involved at every step of the protocol. With respect to the cache, the choice of $num_cache$ between 1 and 2 guarantees the protection of accesses that are repeated after a short period, and this is particularly important for range queries, which would otherwise be recognizable by the repetition of the accesses to the nodes above the leaf; the increase in $num_cache$ produces a limited performance impact and the chosen value appears a good compromise between security and the extreme attention to performance that characterizes most scenarios. Configurations with $num_cover$=1 and $num_cache$ between 1 and 2 have a performance overhead factor of ≈20% with respect to a plain encrypted index (the measured values were 310.58 ms for the plain encrypted index and less than 380 ms for the shuffle in-

.




(a)




(b)

Fig. 10. Access time of the shuffle index as a function of the number of covers in the US-Europe WAN configuration (a), and in the continental WAN configuration (b). The dashed line (baseline) represents the service time observed using a plain encrypted index with the same height.

dex in the US-to-Europe configuration, and 108.78 ms and 130 ms, respectively, in the continental configuration).

The performance overhead caused by our approach to support updates to the data collection is: $\approx 1.42(num_cache + num_cover)$ ms in the US-to-Europe configuration, and $\approx 1.44(num_cache + num_cover)$ ms in the continental configuration, depending on the number of nodes that have been split. The overhead caused by the support of updates to the data collection is almost the same in the LAN and in the WAN scenarios, because the split of a node causes a limited increase in the transmission time of the set of blocks exchanged between the client and the remote server, which is not influenced by the network delay.

We note that, even in a broadband WAN configuration where the network latency is the dominant factor, our solution enjoys a limited communication and computational cost. From the observations above, we believe our approach to be particularly appealing in

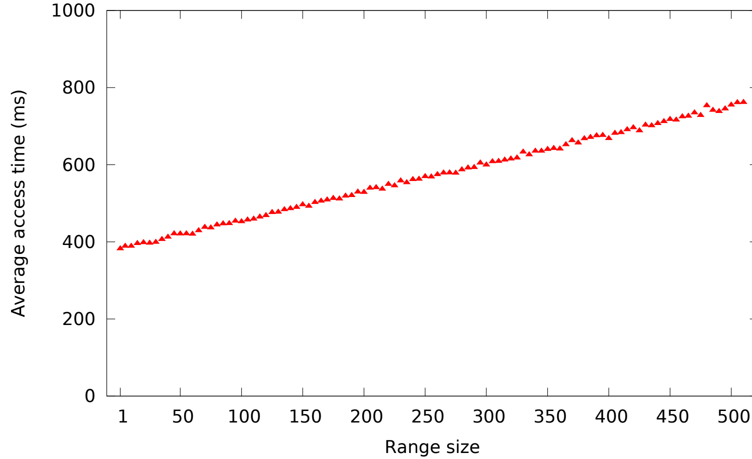



Fig. 11. Range queries performance

many application scenarios, since it provides adequate access and pattern confidentiality at an affordable overhead.

### 10.2. Range Queries

The approach discussed in Section 6 for supporting range queries is based on the execution of a sequence of accesses to the leaf nodes that contain keys that fall in the requested range. The shuffle index is expected to have high fan out and then also to have relatively large nodes, which leads to a tree with a low height. The number of leaves needed to cover the result of a range query is then expected to be limited, as the access to every leaf node is amortized over all the other values belonging to the queried range that appear in the same leaf. In fact, each leaf in the shuffle index stores up to $F - 1$ tuples with contiguous key values and then a range condition including, for example, $F$ tuples will require to access no more than two (contiguous) leaves. To demonstrate the validity of this observation, we run a series of queries with a range width varying between 1 and $512$ index keys, which is the fan out of our shuffle index. For each considered range width, we generated 100 queries by randomly selecting the lower bound of the query range from the numerical domain of the index, following a uniform probability distribution. Figure 11 illustrates the average access times obtained in the US-to-Europe WAN configuration described in the previous section. The figure confirms that the average access time slowly increases with the increase of the range width, with an average cost for the access to an element of the range that continuously decreases with the increase in the range width.

## 11. RELATED WORK

The problem of defining efficient and effective indexing techniques supporting the execution of queries on encrypted data has been extensively studied in the data outsourcing scenario (e.g., [Agrawal et al. 2013; Agrawal et al. 2004; Ceselli et al. 2005; Damiani et al. 2003; Hacigümüs et al. 2002a; Hacigümüs et al. 2002b; Hore et al. 2012; Ren et al. 2012; Shmueli et al. 2005; Wang and Lakshmanan 2006]). The first approach in this direction supported equality conditions only and is based on partitioning the plaintext attribute domain in non-overlapping subsets of values, each mapped to a different index value [Hacigümüs et al. 2002a; Hacigümüs et al. 2002b]. An alternative approach is based on the adoption of hash-based index functions [Ceselli et al. 2005]. To support

also range queries, the outsourced dataset can be complemented with an encrypted $B+$-tree structure, which is iteratively accessed to retrieve the tuples of interest [Ceselli et al. 2005]. Other proposals [Agrawal et al. 2004; Hacigümüs et al. 2004; Hore et al. 2012; Wang et al. 2005] aim at better supporting SQL clauses or at reducing the burden for the requesting client in the query evaluation process. All these proposals, as demonstrated in [Ceselli et al. 2005], suffer from inference attacks when even a limited number of indexes is published. To address inference exposure, in [Wang and Lakshmanan 2006] the authors propose an indexing method that exploits $B$-trees for supporting both equality and range queries, while limiting inference exposure thanks to an almost flat distribution of the frequencies of index values. Other solutions support keyword-based search operations or select/range queries on encrypted and outsourced data without revealing to the server any information about the outsourced data and the target values (e.g., [Chang and Mitzenmacher 2005; Curtmola et al. 2006; Song et al. 2000; Sun et al. 2013; Wang et al. 2010; Wang et al. 2012; Wang et al. 2011]). Both traditional indexing techniques and keyword search approaches focus on protecting confidentiality of data at rest, but do not consider the privacy risks caused by the exposure of the target of accesses. In [Pang et al. 2013] the authors propose a privacy enhancing $B+$-tree index structure that protects both content and access confidentiality. While interesting, also this proposal does not address the pattern confidentiality problem.

Access and pattern confidentiality have been typically addressed by the Private Information Retrieval (PIR) proposals, which however do not protect content confidentiality and suffer from high computation costs that limit their applicability (e.g., [Ostrovsky and Skeith, III 2007]). Recent efforts trying to make PIR more practical have investigated the application of the Oblivious RAM (ORAM) structure [Goldreich and Ostrovsky 1996] and of dynamic data allocation techniques. The proposal in [Williams et al. 2008] exploits the pyramid-shaped database layout of the ORAM, associating with each level a Bloom filter and a hash function for data retrieval. Confidentiality is provided by caching searches and reorganizing the ORAM every time the cache becomes full. Such a reorganization entails a significant performance overhead. The cost of reorganizing the bottom level of the pyramid is $O(N)$, where $N$ is the number of index values in the dataset. The response time of access requests submitted during the reordering of lower levels of the database is high and not acceptable in many real-world scenarios. Also, the architecture requires a secure coprocessor trusted by the client on the server. The proposal in [Ding et al. 2011] mitigates the cost of query evaluation when a reordering of low levels in the ORAM structure becomes necessary. The idea is to limit the shuffling to fetched records only. All these proposals entail a significant performance overhead remaining impractical for many real-life applications. Our shuffle index does not rely on any trust assumption on server components and enjoys a (non amortized) computational cost for query evaluation of $O(\log N)$, with a low constant, maintaining a stable and practical computational and communication overhead (as confirmed by the experimental evaluation).

Recently, novel solutions have been proposed to make ORAM more practical. ObliviStore [Stefanov and Shi 2013] splits the data collection into $P$=$\sqrt{N}$ partitions, each organized as an ORAM data structure. The partitioning framework is designed to reduce the worst-case shuffling cost of the original ORAM model. The client stores a local cache used for background eviction and a position map, keeping track of the partition where each block is stored. Every time a user needs to access a block, she either directly accesses it (if in cache) or downloads it from the server. In both cases, the block is re-assigned to a randomly chosen partition and stored in the local cache. The position map is updated according to the new partition assignment. Blocks in cache are

periodically evicted and written back to the server in the partition to which they are assigned. Read and write of blocks in partitions work according to the ORAM access protocol. Path ORAM technique [Stefanov et al. 2013] relies on a similar approach. It is a tree-shaped data structure whose nodes are buckets storing a fixed number of (dummy and data) blocks. Each block is mapped to a randomly chosen leaf and it is stored either at the client (in a local stash) or in one of the buckets along the path to the leaf. Every read operation downloads (and stores in the stash) all the buckets in the path from the root to the leaf to which the block is mapped. The mapping of the target block is then randomly updated and the path downloaded from the server is written back, possibly inserting into the buckets along the path a subset of blocks in the local stash. The advantage of our shuffle index over ObliviStore and Path ORAM is three-fold. First, thanks to the unchained $B+$-tree logical organization of the data, the shuffle index directly supports the evaluation of range queries. A second advantage (as noted in Section 9) is that our proposal leaves the shuffle index structure stored at the server in a consistent state after each access. Hence, our approach can easily recover from failures at the client side and easily supports the transition from one client to another one. The third advantage is in terms of resource consumption. In fact, while presenting comparable access response time, our approach enjoys a lower network bandwidth overhead, which directly influences the economic cost of the communication. For each target block accessed, a shuffle index with $h$ levels, requires to transfer $(2{+}2{\cdot}num_cover{+}num_cache){\cdot}h$ blocks, while the proposal in [Stefanov and Shi 2013] requires to transfer $40$–$50$ blocks.

Alternative approaches are based on the definition of a dynamically allocated index structure that guarantees efficient and private access to the data by swapping the content of (a subset of) the accessed blocks. The approach in [Lin and Candan 2004a] preserves content and access confidentiality by using a tree-shaped structure and combining access redundancy, node swapping, and node re-encryption as follows. Each access request includes $m$ blocks (the block storing the target node and $m-1$ additional blocks, at least one of which must be empty) and swaps the content of the block storing the target node with the content of one of the blocks downloaded from the server and whose content is empty. The client re-encrypts all the blocks downloaded from the server to hide the swap. This proposal does not provide pattern confidentiality since frequently accessed blocks can be easily identified by the server and exploited for inference purposes. In [Lin and Candan 2004b], the authors propose a solution to this drawback that is based on the preliminary definition of fixed query plans to be adopted in query evaluation, which is difficult to apply in a real-world scenario.

In [Yang et al. 2011], the authors present a solution where the accesses to data items are based on a hierarchical structure with which the data items are organized. Two dummy data items (cover searches, in our terminology) are added to the target data item, and then the target data is possibly swapped with one of these dummy items to protect access confidentiality. To protect repeated accesses, one of the three accessed data items is among the items visited by the previous search. Although similar, the solution in [Yang et al. 2011] also presents differences with respect to our approach, which provides higher flexibility thanks to the support of an arbitrary number of covers and to shuffling among all the accessed/cached nodes. Also, the solution in [Yang et al. 2011] does not rely on a tree-shaped index structure, thus providing lower access performance. The limited fixed number of nodes involved in each access operation and the adoption of a swap, in contrast to shuffling, operation make the protection guarantees provided by the solution in [Yang et al. 2011] weaker than the guarantees offered by the shuffle index approach.

The shuffle index approach presented in this paper (Sections 3–5) has been first introduced in [De Capitani di Vimercati et al. 2011a]. The current submission extends

the original shuffle index proposal in several directions. This paper clearly introduces the motivation of the problem and real world application scenarios (Section 2). It then introduces support for range queries and for updates to the outsourced data collection (i.e., tuple insertion, removal, and update). These enhancements are formally defined, introducing a novel secure access protocol, and analyzed to prove that they do not affect the correctness and security of the original shuffle index proposal (Sections 6–7, and Sections 8–9, respectively). A wide experimental evaluation confirms that the support for range queries and updates to the outsourced data collection does not affect the efficiency of the shuffle index (Section 10). We note that the shuffle index structure proposed in this paper can be integrated with the approach illustrated in [De Capitani di Vimercati et al. 2013b], which supports concurrent read accesses by different clients. This solution consists in dynamically creating versions of the accessed shuffle index nodes, called delta versions, on which each transaction accessing the outsourced data collection has exclusive lock. Periodically, delta versions are reconciled and applied to the shuffle index, to preserve the effects of the different operations performed by different transactions. This approach naturally supports concurrent range queries, while it needs to be adapted to support concurrent write operations. To this purpose, both traditional locking techniques and novel eventual serializability approaches can be adapted to operate in the considered scenario. Although the shuffle index structure proposed in this paper assumes data to be stored at one server only, it can easily be extended to operate in a distributed scenario, according to the proposal in [De Capitani di Vimercati et al. 2013a]. This approach further enhances access and pattern confidentiality guarantees as shuffling operates across different servers. Intuitively, the nodes of the shuffle index are stored at different servers, which are not aware of each other. Every access to the shuffle index entails accessing blocks stored at each of the servers, in such a way to make it believe that it is the only server storing data.

## 12. CONCLUSIONS

We presented an indexing technique for data outsourcing that proves to be efficient while ensuring content, access, and pattern confidentiality. Our solution is based on dynamic data allocation for destroying the otherwise static relationship between blocks and the data they store. We provided a description of the protection techniques adopted and of their combined use for accessing the data collection, while preserving confidentiality. We also described how the shuffle index handles range queries and how to manage insert, delete, and update operations without revealing to the server which kind of operation is being executed. The indistinguishability among the different kinds of accesses is provided by a probabilistic split approach that, at each visit, possibly splits the nodes in the shuffle index even if the access is not inserting a new index value. The experimental results illustrated in this paper show that the proposed solution can manage large data collections, with a limited overhead in the response time to users requesting access to outsourced data. To our knowledge, this is the first work providing content, access, and pattern confidentiality with such a limited overhead. Another advantage is related to the fact that the underlying structure is a $B+$-tree, which is commonly used in relational DBMSs to support the efficient execution of queries. This similarity can facilitate the integration between shuffle indexes and traditional query processing.

## APPENDIX

## A. NODE ENCRYPTION

The definition of block (Definition 3.1) guarantees *content confidentiality* and *integrity*.

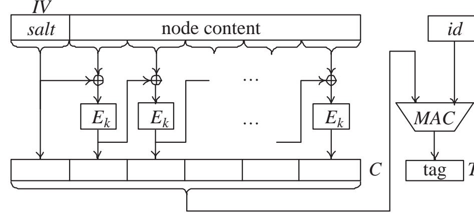



Fig. 12. CBC encryption of a node

**Confidentiality**. To guarantee that a block does not leak information on the node it stores, the symmetric encryption function $E$ adopted must ensure *indistinguishability* under: *i) chosen-plaintext attack* (IND-CPA), meaning that an observer can choose a number of messages and obtain the corresponding ciphertexts accessing an encryption oracle; and *ii) (adaptive) chosen-ciphertext attack* (IND-CCA), meaning that an observer may send a number of ciphertexts to a decryption oracle and then choose other ciphertexts depending on the previous decryption operations. IND-CPA is guaranteed since each node is encrypted with a symmetric function $E$ using the Cipher Block Chaining (CBC) mode [Bellare et al. 1997]. In the CBC mode (Figure 12), the content of a node is partitioned into fragments of equal length. The first fragment is `xor`ed with an Initialization Vector (IV) and then encrypted. Each subsequent fragment is encrypted, after being `xor`ed with the previously encrypted fragment. IND-CCA is guaranteed by the use of a strongly unforgeable keyed cryptographic hash function *MAC*, which is composed with the encryption function $E$ according to the *encrypt-then-mac composition scheme* (i.e., the ciphertext is obtained by first encrypting the plaintext and then appending a MAC of the encrypted plaintext).

**Integrity**. To guarantee integrity, we need to provide: *i) the integrity of plaintext* (INT-PTXT), meaning that it is computationally infeasible to produce a ciphertext that decrypts to a message that the data owner had never encrypted; and *ii) the integrity of ciphertext* (INT-CTXT), meaning that it is computationally infeasible to produce a ciphertext not previously produced by the data owner, regardless of whether or not the underlying plaintext is known. These properties are guaranteed by the use of a strongly unforgeable keyed cryptographic hash function *MAC*. Function *MAC* takes a secret key and an arbitrary-length message (i.e., the node identifier concatenated with the encryption of the node) to be authenticated as input and returns a tag. The tag value guarantees both integrity and authenticity of the considered message by allowing a client, who also possesses the secret key, to detect any change to the message content. Furthermore, it is computationally unfeasible for an observer to find another message-tag pair, even under the assumption of a CPA [Bellare and Namprempre 2008]. As a consequence, the inclusion of the node identifier *id* as input to *MAC* enables the client to assess the authenticity of the returned node (i.e., possible misbehavior of the server returning an incorrect node can be immediately detected). Note that Bellare and Namprempre also demonstrate that the encrypt-then-mac composition scheme used in our definition of block is the only one that is secure in IND-CCA+INT-CTXT and IND-CPA+INT-PTXT.

## B. ACCESS EXECUTION - DETAILED ALGORITHM

Figure 13 reports the detailed pseudocode of the algorithm in Figure 2. In the following, we provide an example of step-by-step execution of the algorithm introduced in Section 5 for accessing a shuffle index structure.

```
INPUT   target_value : value to be searched in the shuffle index
                  /* for range queries: [lower_bound, upper_bound]: query range, with target_value = lower_bound */

OUTPUT  n : leaf node that contains target_value

MAIN
1: /* Initialize variables*/
2: Non_Cached := ∅ /* nodes at level l in cache before or read by the access, but not in cache after the access */
3: Non_Cached_P := ∅ /* nodes at level l − 1 in cache before or read by the access, but not in cache after the access */
4: let n_0 be the unique node in Cache_0
5: target_id := n_0.id
6: cache_hit := TRUE /* the root always belongs to Cache_0 */
7: num_cover := num_cover + 1 /* additional cover necessary to support possible cache hits */
8: for i:=1...num_cover do cover_id[i] := target_id
9: /* Choose cover searches (line 1, Figure 2) */
10: for i:=1...num_cover do
11:    randomly choose cover_value[j] in D s.t. ∀j=1,...,i−1,
12:    ChildToFollow(n_0,cover_value[i]) ≠ ChildToFollow(n_0,cover_value[j])
13:    ChildToFollow(n_0,cover_value[i]) ∉ {n.id|n∈Cache_1} and
14:    ChildToFollow(n_0,cover_value[i]) ≠ ChildToFollow(n_0,target_value)
                                        /* for range queries: next_target := target_value */
                                        /* for updates: start := EvaluateRootSplit(num_cover+num_cache+1) */
15: /* Search, shuffle, and update cache and index structure */
16: for l:=1...h do                      /* for updates for l:=start...h do */
17:    let n∈Cache_{l−1} such that n.id=target_id
18:    target_id := ChildToFollow(n,target_value) /* (line 4, Figure 2) */
                      /* for range queries: if target_value < n.values[Length(n.values)] then */
                      /* next_target := {v∈n.values|(v>target_value) ∧ (∄v'∈n.values, v'>target_value ∧ v>v')}*/
19:    /* identify the blocks to read from the server (lines 5–8, Figure 2) */
20:    if target_id∉{n.id|n∈Cache_l} then
21:       ToRead_ids := {target_id}
22:       if cache_hit then
23:          cache_hit := FALSE
24:          num_cover := num_cover − 1 /* reduce the number of covers by one at the first cache miss */
25:    else ToRead_ids := ∅
26:    for i:=1...num_cover do
27:       let n∈Cache_{l−1}∪Non_Cached_P such that n.id=cover_id[i]
28:       cover_id[i] := ChildToFollow(n,cover_value[i])
29:       ToRead_ids := ToRead_ids ∪ {cover_id[i]}
30:    /* read blocks (line 10, Figure 2) */
31:    Read := Decrypt(ReadBlocks(ToRead_ids))
                                        /* for updates: EvaluateNodesSplit */
32:    /* shuffle nodes (lines 12–13, Figure 2) */
33:    let π be a permutation of ToRead_ids∪{n.id|n∈Cache_l}
34:    for each n∈Read∪Cache_l do n.id := π(n.id)
35:    /* determine effects on parents and store nodes at level l−1 (lines 14–15, Figure 2) */
36:    for each n∈Cache_{l−1}∪Non_Cached_P do
37:       for i:=0...Length(n.values) do n.pointers[i] := π(n.pointers[i])
38:       WriteBlock(n.id, Encrypt(n))
39:    target_id := π(target_id) /* (line 16, Figure 2) */
40:    for i:=1...num_cover do cover_id[i] := π(cover_id[i]) /* (line 17, Figure 2) */
41:    /* update cache at level l (line 19, Figure 2) */
42:    Non_Cached := Read
43:    if cache_hit then refresh the timestamp of n∈Cache_l s.t. n.id=target_id
44:    else let deleted be the least recently used node in Cache_l
45:       let n∈Read s.t. n.id=target_id
46:       insert n into Cache_l
47:       Non_Cached := Non_Cached ∪ {deleted} \ {n}
48:    Non_Cached_P := Non_Cached
49: /* Write nodes at level h (line 20, Figure 2) */
50: for each n∈Cache_h∪Non_Cached_P do WriteBlock(n.id, Encrypt(n))
51: /* Return the target leaf node (line 22, Figure 2) */
52: let n∈Cache_h such that n.id=target_id
                      /* for range queries: if ((target_value≠next_target) AND (next_target≤upper_bound))
                                              then search for next_target */
53: return(n)
```

Fig. 13. Detailed pseudocode of the shuffle index access algorithm

| $l$ | Retrieved nodes | | $\pi$ | Shuffled nodes | | Written nodes | |
|---|---|---|---|---|---|---|---|
| | $Cache_l$ | $Read$ | | $Cache_l$ | $Non_Cached$ | $Cache_{l-1}$ | $Non_Cached_P$ |
| 0 | 001 [103 Ea 101 Ia 104 La 102 - -] * | | | | | | |
| 1 | 101 [204 Fa 203 Ga 206 Ha 202 - -] * | | 101→102 | 102 [204 Fa 203 Ga 206 Ha 202 - -] | | 001 [104 Ea 102 Ia 103 La 101 - -] | |
| | 103 [209 Ba 212 Ca 211 Da 216 - -] | | 103→104 | 104 [209 Ba 212 Ca 211 Da 216 - -] | | | |
| | | 102 [210 Ma 201 Na 207 Oa 208 Pa 205] | 102→101 | | 101 [210 Ma 201 Na 207 Oa 208 Pa 205] | | |
| | | 104 [215 Ja 214 Ka 213 - - - - ] | 104→103 | | 103 [215 Ja 214 Ka 213 - - - - ] | | |
| 2 | 206 [Ga Gb Gc -] | | 206→203 | 203 [Ga Gb Gc -] | | 102 [204 Fa 201 Ga 203 Ha 202 - -] | |
| | 209 [Aa Ab Ac -] | | 209→206 | | 206 [Aa Ab Ac -] | 104 [206 Ba 212 Ca 211 Da 216 - -] | |
| | | 203 [Fa Fb - -] * | 203→201 | 201 [Fa Fb - -] | | | 101 [210 Ma 209 Na 207 Oa 208 Pa 205] |
| | | 201 [Ma Mb - -] | 201→209 | | 209 [Ma Mb - -] | | 103 [215 Ja 214 Ka 213 - - - - ] |
| | | | | | | 203 [Ga Gb Gc -] | 206 [Aa Ab Ac -] |
| | | | | | | 201 [Fa Fb - -] | 209 [Ma Mb - -] |

Fig. 14. An example of access to the shuffle index in Figure 3 with $target_value$=‘Fb’, $cover_value$[1]=‘Ma’, $cover_value$[2]=‘Ic’

*Example* B.1. Figure 14 illustrates an example of execution of the algorithm in Figure 13 for the access to the shuffle index described in Example 5.1. The columns of the table represent: the level of the shuffle index ($l$); the content of the cache ($Cache_l$ in *Retrieved nodes*) and the nodes read from the server (*Read* in *Retrieved nodes*); the permutation ($\pi$); the nodes in the cache ($Cache_l$ in *Shuffled nodes*) and read after the shuffling ($Non_Cached$ in *Shuffled nodes*); the nodes written on the server (*Written nodes*). In column *Retrieved nodes*, a $*$ denotes the node in the path to $target_value$. We note that each row in columns *Retrieved nodes* and *Shuffled nodes* represents the evolution of the same node in the shuffle index.

## C. ACCESS EXECUTION WITH A DYNAMIC INDEX STRUCTURE

In this section, we introduce function **EvaluateRootSplit** and procedure **EvaluateNodesSplit** invoked by the algorithm in Figure 13 and provide a step-by-step example of their execution.

The algorithm for processing a request to search, remove, or insert $target_value$ in shuffle index $\mathcal{S}$ is based on the algorithm described in Section 5 and better detailed in Appendix B. The main differences are related to how the visited nodes are managed. More precisely, the algorithm has to call function **EvaluateRootSplit** that verifies whether the root is full and, in this case, splits it. Also, for each level of the shuffle index, the node in the path to the target value as well as the cover nodes and those in the cache are subject to a probabilistic split, which works as described in Section 7 and is implemented through procedure **EvaluateNodesSplit**. If one or more of these nodes are split, the procedure properly updates the variables used by the algorithm and the cache to preserve the correctness of the algorithm. We now illustrate more in details how the split of the root and of the internal/leaf nodes are managed.

**EvaluateRootSplit**. The split of the root generates a new root with $num_shares \geq num_cover + num_cache + 1$ children. The key values stored in the old root are therefore re-distributed into these nodes according to Definition 7.1. Since the number of levels in the shuffle index increases by one, the cache structure is updated accordingly. Figure 15 illustrates the pseudocode of function **EvaluateRootSplit**, which is called after the cover searches have been determined by the algorithm in Figure 13 (i.e., after line 14 in Figure 13). It receives as input the number $num_shares$ of children that the new root node must have, and returns the level in the tree from which the search process should start (i.e., the starting value of variable $l$ used by the **for** loop at line 16 in Figure 13), which is equal either to 2 or to 1 depending on whether the root node has been split or not, respectively.

The function retrieves the root of the shuffle index from $Cache_0$ (line 2). If the root is not full, the function returns 1 and terminates (line 3). Otherwise, the function creates

1: **EVALUATE_ROOT_SPLIT**($num_shares$)
2: let $n_0$ be the unique node in $Cache_0$ /* $n_0$ is the current root of the shuffle index */
3: **if** **Length**($n_0.values$)$<$($F - 1$) **then** **return**(1) /* the root is not full and it is not split */
4: $new_root.id$ := **GetFreeIdentifier**($\mathcal{S}$) /* returns a logic identifier not used by the nodes of the shuffle index */
5: $val_per_child$ := $\left\lfloor \frac{\mathbf{Length}(n_0.values) - (num_shares - 1)}{num_shares} \right\rfloor$ /* number of values in each new child of the root */
6: $new_root.pointers[0]$ := $n_0.id$
7: move $n_0.values[val_per_child + 1]$ to $new_root.values[1]$
8: $pos$ := $val_per_child + 1$ /* position of the last value in $n_0$ that has been copied into one of its new children */
9: **for** $i$:=1, ..., $num_shares - 1$ **do** /* create the $i$-th child of the new root; the first child is the old root */
10: $n_i.id$ := **GetFreeIdentifier**($\mathcal{S}$)
11: **if** $i \neq num_shares - 1$ **then** /* all the new children of the root but the last one store the same number of values */
12: $num_vals$ := $val_per_child$
13: move $n_0.values[pos + num_vals + 1]$ to $new_root.values[i{+}1]$ /* promote the value to the root node */
14: **else** $num_vals$ := **Length**($n_0.values$)$-pos$ /* the last child might store more values than others */
15: $new_root.pointers[i]$ := $n_i.id$ /* insert into the root the pointer to the new child */
16: move $n_0.values[pos + 1, \ldots, pos + num_vals]$ to $n_i.values[1, \ldots, num_vals]$ /* move values to the new child */
17: move $n_0.pointers[pos, \ldots, pos + num_vals]$ to $n_i.pointers[0, \ldots, num_vals]$ /* move pointers to the new child */
18: $pos$ := $pos + num_vals + 1$
19: /* update the cache structure */
20: $h$ := $h$+1 /* increase by 1 the height of the shuffle index */
21: **for** $l$:=$h$...1 **do** $Cache_l$ := $Cache_{l-1}$ /* update the cache structure inserting a new level */
22: $Cache_0$ := $\emptyset$; $Cache_1$ := $\emptyset$
23: insert $new_root$ into $Cache_0$ with the timestamp of $n_0$
24: **for** $i$=0, ..., $num_shares - 1$ **do**
25: **if** ($\exists n_c \in Cache_2$ s.t. $n_c.id \in n_i.pointers$) **then** /* $n_i$ must be kept in cache to provide path continuity */
26: insert $n_i$ into $Cache_1$ with the timestamp of the MRU node in $Cache_2$ and child of $n_i$
27: **else** $Non_Cached_P$ := $Non_Cached_P \cup \{n_i\}$ /* move the new children that are not in cache to $Non_Cached_P$ */
28: **if** $|Cache_1| < num_cache$ **then** move $num_cache - |Cache_1|$ nodes from $Non_Cached_P$ to $Cache_1$ with $n_0$'s timestamp
29: **WriteBlock**($new_root.id$, **Encrypt**($new_root$))
30: $target_id$ := **ChildToFollow**($new_root$,$target_value$) /* update the identifier of the node in the path to $target_value$ */
31: let $n \in \{n_0, \ldots, n_{num_shares-1}\}$ s.t. $n.id$=$target_id$
32: let $deleted$ be the least recently used node in $Cache_1$
33: **if** $n \in Cache_1$ **then** refresh the timestamp of $n$ /* update $Cache_1$ */
34: **else**
35: insert $n$ into $Cache_1$
36: $Non_Cached_P$ := $Non_Cached_P \cup \{deleted\} \setminus \{n\}$
37: **return**(2)

Fig. 15. Function that evaluates whether to split the root node

a new root node $new_root$ whose identifier corresponds to a block that is not in use and is obtained from the server through function **GetFreeIdentifier** (line 4). Function **EvaluateRootSplit** then determines the minimum number of values ($val_per_child$) that will be stored in each of the children of the new root. This value is computed by taking into account the fact that $num_shares - 1$ values in $n_0$ will be promoted to the new root (line 5). The function then inserts the identifier of the first child ($n_0.id$) of the new root into $new_root.pointers[0]$ and moves value $n_0.values[val_per_child + 1]$ to $new_root.values[1]$ (lines 6-7). The function continues by creating the remaining children of the new root and by re-distributing the values and pointers originally stored in $n_0$ according to Definition 7.1 (lines 8-18).

Function **EvaluateRootSplit** then updates the cache structure by adding a new layer and inserting the new root into $Cache_0$ with the same timestamp of the old root $n_0$ (lines 20-23). To guarantee the path continuity property, the function inserts into $Cache_1$ the children of the new root that have at least one child in $Cache_2$ (lines 24-27); the other children of the new root are inserted into $Non_Cached_P$ (line 27). Note that the timestamp associated with each node $n_i$ inserted into $Cache_1$ is the highest among the ones of its descendants in $Cache_2$. In this way, a node is not pushed out from the cache when its children are still in cache. Also, if $Cache_1$ does not contain $num_cache$ nodes, then $num_cache - |Cache_1|$ nodes are moved from $Non_Cached_P$ to $Cache_1$ (line 28). The new root is then encrypted, and sent to the server for storage (lines 29). The function updates variable $target_id$, which represents the identifier of

the node in the path to $target_value$, and sets it to the identifier of the node resulting from the split that is in the path to the target (line 30). It then updates $Cache_1$ accordingly, that is, it updates the timestamp of the node $n$ in the path to $target_value$ if $n$ is in cache; it inserts $n$ into $Cache_1$ otherwise (lines 31-36). Function **EvaluateRootSplit** finally returns value 2 as the access to the shuffle index starts from level 2 of the index (line 37).

**EvaluateNodesSplit**. The split of an internal or leaf node generates a new node and requires to update the parent of the split node. Figure 16 illustrates the pseudocode of procedure **EvaluateNodesSplit**, which is called at each level $l$ of the shuffle index after the blocks representing nodes in the paths to the target and cover values have been downloaded from the server and decrypted (i.e., after line 31 in Figure 13). The procedure is called before shuffling, since also the nodes resulting from the split must be shuffled together with the nodes in $Read$ (i.e., downloaded from the server) and in $Cache_l$.

The procedure first creates a copy of $Read$ (variable $Copy_Read$) and of $Cache_l$ (variable $Copy_Cache$) as their content will be possibly modified as a result of the split of the nodes they include (lines 2-3). Then, for each node $n$ in $Read$ and in $Cache_l$, the procedure verifies whether $n$ should be split. The procedure randomly extracts a number $rnd$ from a uniform distribution $U(0,1)$ and compares it with $\wp(n)$. If $rnd$ is lower than $\wp(n)$, node $n$ is split (lines 4-6). In such a case, the procedure gets the identifier of a free block from the server and assigns it to the new node $n'$ (line 8). Procedure **EvaluateNodesSplit** determines the value $promoted_value$ in $n.values$ that will be promoted to the parent $parent$ of node $n$ (lines 9-10), and then proceeds with the split of node $n$ according to Definition 7.2 (lines 11-16). Note that value $promoted_value$ is inserted into $parent.values[i]$ so that $parent.values[i-1]<promoted_value<parent.values[i+1]$, and consequently $n'.id$ is inserted into $parent.pointers[i]$ (lines 18-25). The procedure then updates the variables used to follow the paths to the target and cover values, and the cache structure. To this purpose, it first checks whether $n$ is the node in the path to $target_value$. If this is the case and $target_value$ follows $promoted_value$, then the procedure sets $target_id$ to $n'.id$ (line 27). Subsequently, the procedure verifies whether $n$ is in $Copy_Cache$. The following two cases may then occur.

*Case 1: node $n$ is in cache.* If node $n$ does not have any child in $Cache_{l+1}$ and the new node $n'$ has a child in $Cache_{l+1}$ or $n'$ is the node in the path to $target_value$, node $n$ is removed from the cache and is inserted into $Copy_Read$ while its identifier is inserted in $ToRead_ids$ (lines 32-37). Otherwise, node $n$ remains in the cache and its timestamp is updated to the timestamp of the most recently used among its children in cache (if any). Then, if node $n'$ has at least one child in $Cache_{l+1}$ or $n'$ is the node in the path to $target_value$, node $n'$ is inserted in the cache (lines 38-39). Otherwise, node $n'$ is inserted into $Copy_Read$, and its identifier into $ToRead_ids$ (lines 40-42). Note that both $n$ and $n'$ can be in $Copy_Cache$ only if they have at least one child each in $Cache_{l+1}$. This implies that the node least recently used (i.e., node $deleted$) can be removed and added to $Copy_Read$, and its identifier inserted into $ToRead_ids$ (lines 43-45). The removal of $deleted$ does not violate the path continuity property since $deleted$ cannot have any child in $Cache_{l+1}$ (see Theorem 8.3).

*Case 2: node $n$ is not in cache.* Node $n$ can be a node in the path to a cover, say $cover_value[i]$. If $cover_value[i]$ follows $promoted_value$, the node in the path to the cover is $n'$ and the procedure then sets $cover_id[i]$ to $n'.id$ (lines 48-49), adds $n'$ to $Copy_Read$, and inserts $n'.id$ into $ToRead_ids$ (lines 50-51).

```
1: EVALUATE_NODES_SPLIT
2: Copy_Read := Read
3: Copy_Cache := Cache_l
4: for each n∈Read∪Cache_l do
5:    randomly extract a value rnd from uniform distribution U(0, 1)
6:    if rnd<℘(n) then /* decide whether to split n using function ℘(n) */
7:       /* split n adding a new node n' */
8:       n'.id := GetFreeIdentifier(S) /* returns a logic identifier not used by the nodes of the shuffle index */
9:       split := ⌊Length(n.values)/2⌋ /* split position */
10:      promoted_value := n.values[split + 1] /* promote the split value to the parent of n */
11:      if l = h then /* n is a leaf node and all the values are reported in the two leaves */
12:         move n.values[split + 1, . . . , Length(n.values)] to n'.values[1, . . . , Length(n.values) − split]
13:         move n.data[split + 1, . . . , Length(n.values)] to n'.data[1, . . . , Length(n.values) − split]
14:      else /* n is an internal node */
15:          move n.values[split + 2, . . . , Length(n.values)] to n'.values[1, . . . , Length(n.values) − split − 1]
16:          move n.pointers[split + 1, . . . , Length(n.values)] to n'.pointers[0, . . . , Length(n.values) − split − 1]
17:      /* update the parent of node n to point to n' */
18:      let parent be the node in Non_Cached_P∪Cache_{l−1} s.t. n.id∈parent.pointers
19:      i := Length(parent.values) + 1
20:      while parent.values[i − 1]>promoted_value do
21:            parent.values[i] := parent.values[i − 1]
22:            parent.pointers[i] := parent.pointers[i − 1]
23:            i := i − 1
24:      parent.values[i] := promoted_value
25:      parent.pointers[i] := n'.id
26:      /* update the pointer to the node in the path to target_value */
27:      if n.id = target_id AND target_value≥promoted_value then target_id := n'.id
28:      /* update the cache at level l */
29:      if n∈Copy_Cache then /* Case 1: n is in Copy_Cache */
30:         let deleted be the least recently used node in Copy_Cache
31:         N := {n_i∈Cache_{l+1} s.t. n_i.id∈n.pointers OR n_i.id∈n'.pointers} /* N=∅ when l+1> h */
32:         if ((n.pointers ∩ N = ∅) AND (N ≠ ∅ OR n'.id=target_id)) then
33:            remove n from Copy_Cache
34:            Copy_Read := Copy_Read ∪ {n} /* necessary to include the node in the shuffling */
35:            ToRead_ids := ToRead_ids ∪ {n.id}
36:            else
37:                 set the timestamp of n to the timestamp of the MRU node n_i in N∩n.pointers, if it exists
38:         if ((n'.pointers ∩ N ≠ ∅) OR (N = ∅ AND n'.id = target_id)) then
39:            insert n' into Copy_Cache with the timestamp of the MRU node in N∩n'.pointers, if it exists; n otherwise
40:         else
41:             Copy_Read := Copy_Read ∪ {n'} /* necessary to include n' in the shuffling */
42:             ToRead_ids := ToRead_ids ∪ {n'.id}
43:         if n,n'∈Copy_Cache then
44:            Copy_Read := Copy_Read ∪ {deleted} /* necessary to include deleted in the shuffling */
45:            ToRead_ids := ToRead_ids ∪ {deleted.id}
46:      /* update the nodes in the paths to cover searches */
47:      else /* Case 2: n is not in Copy_Cache and could be a cover */
48:          for i=1, . . . , num_cover do
49:              if n.id = cover_id[i] AND cover_value[i]≥promoted_value then cover_id[i] := n'.id
50:          Copy_Read := Copy_Read ∪ {n'}
51:          ToRead_ids := ToRead_ids ∪ {n'.id}
52: Read := Copy_Read
53: Cache_l := Copy_Cache
```

Fig. 16. Procedure that possibly splits nodes visited at level $l$>0 of the shuffle index

When all the nodes in *Read* and in $Cache_l$ have been checked for split, procedure **EvaluateNodesSplit** sets *Read* to *Copy_Read* and $Cache_l$ to *Copy_Cache* and terminates (lines 52-53).

*Example* C.1. Figure 17 illustrates an example of execution of the algorithm in Figure 13, extended to manage possible updates to the data collection for the insert operation described in Example 7.4. The table has the same structure as the one in Figure 14 and its columns represent: the level of the shuffle index ($l$); the content of the cache ($Cache_l$ in *Retrieved nodes*) and the nodes read from the server (*Read* in *Retrieved nodes*); the permutation ($\pi$); the nodes in the cache and read after the shuffling

| $l$ | **Retrieved nodes** | | $\pi$ | **Shuffled nodes** | | **Written nodes** | |
|---|---|---|---|---|---|---|---|
| | *Cache*$_l$ | *Read* | | *Cache*$_l$ | *Non_Cached* | *Cache*$_{l-1}$ | *Non_Cached_P* |
| 0 | 001 [103 Ea 101 Ia 104 La 102 - - ] * | | | | | | |
| 1 | 101 [204 Fa 203 Ga 206 Ha 202 - - ] * | | | | | | |
| | 103 [209 Ba 212 Ca 211 Da 216 - - ] | | | | | | |
| | | 102 [210 Ma 201 Na 207 Oa 208 Pa 205] | | | | | |
| | | 104 [215 Ja 214 Ka 213 - - - - ] | | | | | |
| | SPLIT NODE 102 | | | | | | |
| | 101 [204 Fa 203 Ga 206 Ha 202 - - ] * | | 101→102 | 102 [204 Fa 203 Ga 206 Ha 202 - - ] | | 001 [105 Ea 102 Ia 101 La 104 Oa 103] | |
| | 103 [209 Ba 212 Ca 211 Da 216 - - ] | | 103→105 | 105 [209 Ba 212 Ca 211 Da 216 - - ] | | | |
| | | 102 [210 Ma 201 Na 207 - - - - ] | 102→104 | | 104 [210 Ma 201 Na 207 - - - - ] | | |
| | | 105 [208 Pa 205 - - - - - - ] | 105→103 | | 103 [208 Pa 205 - - - - - - ] | | |
| | | 104 [215 Ja 214 Ka 213 - - - - ] | 104→101 | | 101 [215 Ja 214 Ka 213 - - - - ] | | |
| 2 | 206 [Ga Gb Gc -] | | | | | | |
| | 209 [Aa Ab Ac -] | | | | | | |
| | | 203 [Fa Fb - - ] * | | | | | |
| | | 201 [Ma Mb - - ] | | | | | |
| | SPLIT NODE 206 | | | | | | |
| | 206 [Ga - - - ] | | 206→201 | 201 [Ga - - - ] | | 102 [204 Fa 206 Ga 201 Gb 203 Ha 202] | |
| | | 217 [Gb Gc - - ] | 217→203 | | 203 [Gb Gc - - ] | 105 [217 Ba 212 Ca 211 Da 216 - - ] | |
| | 209 [Aa Ab Ac -] | | 209→217 | 217 [Aa Ab Ac -] | | | 104 [210 Ma 209 Na 207 - - - - ] |
| | | 203 [Fa Fb - - ] * | 203→206 | | 206 [Fa Fb - - ] | | 103 [208 Pa 205 - - - - - - ] |
| | | 201 [Ma Mb - - ] | 201→209 | | 209 [Ma Mb - - ] | | 101 [215 Ja 214 Ka 213 - - - - ] |
| | | | | | | 206 [Fa Fb Fc -] | 203 [Gb Gc - - ] |
| | | | | | | 201 [Ga - - - ] | 209 [Ma Mb - - ] |
| | | | | | | | 217 [Aa Ab Ac -] |

Fig. 17. An example of access to the shuffle index in Figure 3 to insert value *target-value*='Fc', with *cover-value*[1]='Ma', *cover-value*[2]='Ic'

(*Shuffled nodes*); the nodes written on the server (*Written nodes*). In column *Retrieved nodes*, a * denotes the node in the path to *target-value*. We note that column *Retrieved nodes* may include less nodes than columns *Shuffled nodes* and *Written nodes*, since the first column includes original nodes downloaded from the server, while the second and the third columns include the nodes possibly resulting from a split operation.

.

## D. CORRECTNESS AND COMPUTATIONAL COMPLEXITY

THEOREM 8.1 (RETRIEVAL CORRECTNESS). Let $\mathcal{S}$ be a shuffle index built on candidate key $K$ with domain $\mathcal{D}$ and *target_value* be a value in $\mathcal{D}$. The algorithm in Figure 2 returns the unique leaf node where *target_value* is (or should be) stored.

PROOF. To prove the theorem, we need to show that at each iteration $l$ of the **for** loop traversing the shuffle index (line 16, Figure 13), variable *target_id* always contains the identifier of the node in the path to *target_value*. To this purpose, we first show that function **EvaluateRootSplit** and procedure **EvaluateNodesSplit** preserve this property, and we then show that also the algorithm in Figure 13 preserves this property.

*Function* **EvaluateRootSplit**. Before calling function **EvaluateRootSplit** (after line 14, Figure 13), variable *target_id* is initialized to the identifier of the root node of the shuffle index $\mathcal{S}$, and therefore it correctly stores the identifier of the node in the path to *target_value*. When function **EvaluateRootSplit** is called, two cases may occur. In the first case, the root node is not full and the function immediately terminates without making any change. Variable *target_id* therefore stores the identifier of the root node when the function returns. In the second case, the root node is full and the function splits it by creating a new root node with *num_shares* children $\{n_0, \ldots, n_{num_shares-1}\}$. It is easy to see that the identifiers of all the children of the new root are stored in *new_root.pointers* (line 6 and line 15, Figure 15), and that the re-distribution of the values in $n_0$.*values* among the new root node and the other *num_shares* − *1* children as well as the re-distribution of the pointers in $n_0$.*pointers* among the other *num_shares*−*1* children of the new root satisfy Definition 7.1. The split of node $n_0$ results therefore in *num_shares* + *1* nodes that contain the same information stored in $n_0$. When function **EvaluateRootSplit** updates variable *target_id* (line 30, Figure 15), the new value of *target_id* will correspond to the identifier of a node $n_i$ at level 1 that is the node that contains the values and pointers originally stored in $n_0$ such that *target_value*∈ [$n_i$.*values*[1],$n_i$.*values*[**Length**($n_i$.*values*)]]. It follows that when function **EvaluateRootSplit** terminates, *target_id* contains the identifier of the node in the path to *target_value*.

*Procedure* **EvaluateNodesSplit**. At each iteration $l$ of the **for** loop (line 16, Figure 13), before calling procedure **EvaluateNodesSplit**, variable *target_id* is set to **ChildToFollow**($n$,*target_value*), where $n$ is the node in the path to *target_value* at level $l-1$ (lines 17-18, Figure 13). As a result of function **ChildToFollow**, *target_id* is the identifier of a child of $n$ and therefore belongs to level $l$ of the shuffle index. Since *target_id* is determined through function **ChildToFollow**($n$,$v$), which identifies the only subtree rooted at $n$ where $v$ can appear, when procedure **EvaluateNodesSplit** is called (after line 31, Figure 13) variable *target_id* represents the identifier of the node at level $l$ in the path to *target_value*. When procedure **EvaluateNodesSplit** is called, two cases can occur. First, the procedure splits nodes that do not correspond to *target_id*. In this case, variable *target_id* does not change (the condition of the **if** instruction on line 27 in Figure 16 evaluates to false) and therefore when the procedure terminates it still corresponds to a node at level $l$ in the path to *target_value*. Second, the procedure splits a node $n$ that corresponds to *target_id*. In this case, a new node $n'$ is created according to Definition 7.2. Therefore, if node $n$ is the node in the path to *target_value*, after its split either $n$ or $n'$ is still the node in the path to *target_value*. The procedure then updates variable *target_id* accordingly (line 27, Figure 16). We can then conclude that after the call to procedure **EvaluateNodesSplit**, variable *target_id* corresponds to the node in the path to *target_value*.

*Search Algorithm.* Before entering the **for** loop, variable $target_id$ corresponds to the node on the path to $target_value$ at level 0 (if **EvaluateRootSplit** does not split the root) or at level 1 (if **EvaluateRootSplit** splits the root), as we showed above. Consequently, the **for** loop starts at level 1 or level 2. As already shown above, at each iteration $l$ of the **for** loop (line 16, Figure 13), both before and after calling procedure **EvaluateNodesSplit**, variable $target_id$ correctly represents the identifier of the node at level $l$ in the path to $target_value$. The block identified by $target_id$ is then downloaded from the server, if the corresponding node (identified by $target_id$) is not in $Cache_l$ (lines 20-21, Figure 13). Since all the leaves of an unchained $B+$-tree are at level $h$, at the end of the **for** loop, $target_id$ is the identifier of the only leaf node where $target_value$ is stored, if it belongs to the data collection. Therefore, if node $target_id$ does not contain $v$=$target_value$, then $target_value$ does not belong to the dataset. Note also that the integrity of the blocks in the path to $target_value$ is guaranteed by the use of a MAC function in the definition of block (see Appendix A). □

THEOREM 8.2 (SHUFFLE INDEX CORRECTNESS). Let $\mathcal{S}$ be a shuffle index representing an unchained $B+$-tree built on candidate key $K$ defined over domain $D_K$, $\mathcal{D}\subset D_K$ be the set of key values stored in the unchained $B+$-tree, and $target_value$ be a value in $D_K$. After the execution of an access operation on $\mathcal{S}$ by the algorithm in Figure 2 with target value $target_value$, $\mathcal{S}$ is a shuffle index representing an unchained $B+$-tree defined on $\mathcal{D}$ possibly extended with $target_value$ if the access is inserting $target_value$.

PROOF. We first prove that the direct ancestor of every node in $Read$ and $Cache_l$, $l$=$1,\ldots,h$, always belongs to either $Cache_{l-1}$ or $Non_Cached_P$ (i.e., the path continuity property is satisfied considering the cache and $Non_Cached_P$). When the algorithm is called for the first time, $Cache_{l-1}$ contains the direct ancestors of all the nodes in $Cache_l$, $l$=$1,\ldots,h$, since the data owner is assumed to correctly initialize the cache at the time of outsourcing. We now prove that the path continuity property between $Read\cup Cache_l$ and $Non_Cached_P\cup Cache_{l-1}$ is preserved by function **EvaluateRootSplit** and procedure **EvaluateNodesSplit**, and also by the search algorithm in Figure 13.

*Function* **EvaluateRootSplit**. When the function is called, both $Non_Cached_P$ and $Read$ are empty, therefore the path continuity property holds between nodes in cache. Two cases may then occur. In the first case, the root node is not full and the function immediately terminates without making any change. The path continuity property remains satisfied. In the second case, the root is full, it is split, and the cache structure is changed accordingly. More precisely, the function first moves $Cache_{l-1}$ to $Cache_l$, $l = h,\ldots,1$, without compromising the path continuity between nodes in contiguous levels of the cache structure (line 21, Figure 15). Then, it inserts into $Cache_0$ the new root (line 23, Figure 15) and into $Cache_1$ all the children of the new root that have at least one child in $Cache_2$; all the other children of the new root are instead inserted into $Non_Cached_P$ (lines 24-27, Figure 15). Note that if the resulting nodes in $Cache_1$ are less than $num_cache$, the procedure moves some nodes from $Non_Cached_P$ to $Cache_1$ (line 28, Figure 15). Also, the procedure possibly inserts into $Cache_1$ the node in the path to $target_value$ and consequently moves the least recently used node from $Cache_1$ to $Non_Cached_P$ (lines 32-36, Figure 15) . At this point, the path continuity property of the cache has been restored since the new root in $Cache_0$ is the parent of the nodes in $Cache_1$, and nodes in $Cache_1\cup Non_Cached_P$ are the parents of the nodes in $Cache_2\cup Read$ ($Read$ is empty and node $deleted$ removed from $Cache_1$ has been inserted into $Non_Cached_P$).

*Procedure* **EvaluateNodesSplit**. We assume that the path continuity property is satisfied before calling procedure **EvaluateNodesSplit**. In particular, we as-

sume that when the procedure is called, the path continuity property between $Cache_{l-1} \cup Non_Cached_P$ and $Cache_l \cup Read$, and between $Cache_l$ and $Cache_{l+1}$ hold. Clearly, we are interested in showing that the split of a node does not violate such a property. We therefore consider a node $n \in Cache_l \cup Read$ that is split. In this case, the procedure updates the parent node $parent$ of $n$ (lines 18-25, Figure 16), which is a node in $Cache_{l-1} \cup Non_Cached_P$, so that it correctly refers to the split node $n$ and the new node $n'$ (i.e., $parent$ is extended to include value $promoted_value$ and the pointer to $n'$). The value of variable $target_id$ is then possibly updated to guarantee that it always corresponds to the node at level $l$ that is in the path to $target_value$ (line 27, Figure 16). Then, two cases may occur.

The first case happens when node $n$ belongs to $Cache_l$. In this case, since the original node $n$ has been split in two, we need to verify whether $n$ and/or $n'$ have at least a child in $Cache_{l+1}$ or whether one of them is the node in the path to $target_value$. More precisely, if neither node $n$ nor $n'$ have a child in $Cache_{l+1}$ and $n'$ is not the node in the path to $target_value$, the procedure keeps node $n$ in $Cache_l$. Otherwise, if node $n'$ has at least a child in $Cache_{l+1}$ or it is the node in the path to $target_value$, the procedure replaces $n$ with $n'$. Note that both $n$ and $n'$ are in $Cache_l$ only if both of them have at least one child in $Cache_{l+1}$. In this case, since each level of the cache contains the same number of nodes, we have the guarantee that there is at least one node in $Cache_l$ that does not have a child in $Cache_{l+1}$. This node must be the least recently used in $Cache_l$ as the timestamp of each node is refreshed any time it is visited. This implies that when both $n$ and $n'$ are in $Cache_l$, the removal of the least recently used node $deleted$ does not violate the path continuity property between $Cache_l$ and $Cache_{l+1}$. The node between $n$ and $n'$ that is not inserted into $Cache_l$ is inserted into $Read$ (lines 33-35 and lines 41-42, Figure 16). Therefore, the procedure preserves the path continuity property between $Cache_{l-1} \cup Non_Cached_P$ and $Cache_l \cup Read$.

The second case happens when node $n$ does not belong to $Cache_l$. In this case, $Cache_l$ is not updated and therefore the path continuity property between $Cache_l$ and $Cache_{l+1}$ remains valid. We also observe that node $n$ already belongs to $Read$ and the procedure then adds node $n'$ in $Read$ (line 50, Figure 16), thus maintaining the path continuity property between $Cache_{l-1} \cup Non_Cached_P$ and $Cache_l \cup Read$. Note also that if node $n$ is a cover, the procedure verifies whether the corresponding identifier has to be fixed (lines 48-49, Figure 16).

*Search Algorithm.* When the algorithm is called for the first time, $Cache_{l-1}$ contains the direct ancestors of all the nodes in $Cache_l$, $l$=$1, \ldots, h$. We now prove, by induction, that at the beginning of each iteration of the **for** loop traversing $\mathcal{S}$ (line 22, Figure 13), $Cache_{l-1} \cup Non_Cached_P$ contains the direct ancestors of all the nodes in $Cache_l \cup Read$. At the beginning of the first iteration, $Cache_l \cup Non_Cached_P$ contains the direct ancestors of all the nodes in $Cache_{l+1} \cup Read$, with $l$=0 or $l$=1, since function **EvaluateRootSplit** guarantees the path continuity between $Cache_l$ and $Cache_{l+1}$ (both when the root node is not split and when the root node is split). We now assume that $Cache_{l-1} \cup Non_Cached_P$ contains the direct ancestors of all the nodes in $Cache_l \cup Read$ at the beginning of the $l$-th iteration. We prove that this property holds also at the beginning of the ($l$+1)-th iteration.

Procedure **EvaluateNodesSplit**, as shown above, preserves the path continuity property between $Cache_l$ and $Cache_{l+1}$. After procedure **EvaluateNodesSplit** terminates, the content of $Cache_l$ is updated by possibly inserting a new node and removing the least recently used node. The node inserted into $Cache_l$ (lines 45-46, Figure 13) belongs to $Read$ that contains nodes whose identifiers have been determined by calling function **ChildToFollow** on a node in $Cache_{l-1} \cup Non_Cached_P$ (lines 18 and 28, Figure 13) and by possibly splitting them through procedure **EvaluateNodesSplit**. The

node removed from $Cache_l$ (line 45, Figure 13), *deleted*, is inserted into $Non_Cached$ (line 47, Figure 13) and $Non_Cached$ is used to initialize $Non_Cached_P$ for the following iteration. If *deleted* is a direct ancestor of a node in $Cache_{l+1}$, say $n'$, at the next iteration $(l+1)$ the direct ancestor of $n'$ in $Cache_{l+1}$ belongs to $Non_Cached_P$. We can then conclude that the direct ancestor of every node in $Cache_l$ belongs to either $Cache_{l-1}$ or $Non_Cached_P$.

To prove that $\mathcal{S}$ represents an unchained $B+$-tree equivalent to the original one, we then need to prove that: for each node $n$ that has not been split, the pointers $n.pointers[i], i = 0,\dots,$ **Length**$(n.values)$, represent the pointers to the children of $n$ in the original unchained $B+$-tree; for each node $n$ that has been split, the pointers in $n.pointers \cup n'.pointers$ ($n.pointers \cup \dots \cup n_{num_shares-1}.pointers$, if $n$ is the root node) represent the pointers to the children of the original node $n$ in the initial unchained $B+$-tree.

First, we note that if function **EvaluateRootSplit** splits the root node $n_0$ (Definition 7.1), the pointers in $n_0.pointers$ are redistributed among nodes $n_1,\dots,n_{num_shares-1}$ so that the pointers remaining in $n_0$ along with $n_1.pointers \cup \dots \cup n_{num_shares-1}.pointers$ correspond to those stored in the original root node (see the proof of Theorem 8.1). Analogously, if procedure **EvaluateNodesSplit** splits node $n$ (Definition 7.2), the pointers in the new node $n'$ (i.e., $n'.pointers$) along with those still stored in $n$ correspond to the pointers stored in the original node $n$ (see the proof of Theorem 8.1). At iteration $l$ of the **for** loop scanning $\mathcal{S}$ (line 16, Figure 13), the identifier of each node $n \in Read \cup Cache_l$ is substituted with its permuted value, $\pi(n.id)$ (line 34, Figure 13). Since all the nodes resulting from the split of $n$ belong to either $Read$ or $Cache_l$ as shown above, the split of node $n$ does not affect the shuffling operation. For each node $n' \in Cache_{l-1} \cup Non_Cached_P$, the algorithm then substitutes the value of $n'.pointers[i]$, $i$=0,...,**Length**($n.values$), with its permutation $\pi(n'.pointers[i])$ (lines 36-37, Figure 13). Since all the direct ancestors of the nodes in $Read \cup Cache_l$ belong to $Cache_{l-1} \cup Non_Cached_P$ as proved above, all the pointers to children are correctly represented. The nodes in $Cache_{l-1}$ and $Non_Cached_P$ are then encrypted and permanently stored at the server (line 38, Figure 13). At the end of the **for** loop scanning $\mathcal{S}$ (line 16, Figure 13), the nodes in $Cache_h$ and $Non_Cached_P$ include the node where $target_value$ is (or should be) stored (see the proof of Theorem 8.1). If the access performs an insert operation, $target_value$ is written in the correct leaf node. The nodes in $Cache_h$ and $Non_Cached_P$ are then encrypted and stored at the server (line 50, Figure 13). □

THEOREM 8.3 (CACHE CORRECTNESS). Let $\mathcal{S}$ be a shuffle index and $Cache_0,\dots,Cache_h$ be a cache (Definition 4.2). After the execution of an access operation on $\mathcal{S}$ by the algorithm in Figure 2, $Cache_0,\dots,Cache_h$ satisfies Definition 4.2.

PROOF. To prove the theorem, we need to prove that none of the properties in Definition 4.2 is violated by the operations of the algorithm, of function **EvaluateRootSplit**, and of procedure **EvaluateNodesSplit**.

*1)* $Cache_0$ ***contains the root node*** $\langle id_0, n_0\rangle$. The algorithm in Figure 13 updates $Cache_0$ only if the root is split by function **EvaluateRootSplit**, which happens when the root is full. In this case, function **EvaluateRootSplit** moves $Cache_{l-1}$ to $Cache_l$, $l$=1,...,$h$, and empties $Cache_0$ and $Cache_1$ (line 22, Figure 15). Then, it inserts node $new_root$, which is the new node created by the function and representing the parent of the $num_shares$ nodes resulting from the split of the old root, into $Cache_0$ (line 23, Figure 15). Since neither the search algorithm in Figure 13 nor procedure **EvaluateNodesSplit** in Figure 16 insert/remove nodes into/from $Cache_0$, the first property in Definition 4.2 is satisfied.

*2) $Cache_l$, $l = 1, \ldots, h$, contains* num_cache *nodes belonging to the $l$-th level of the unchained $B+$-tree.* The update of the timestamp of one of the nodes in cache does not violate this property. We then need to prove that split operation and the insertion (removal, respectively) of a node into the cache do not violate this property.
First, we prove that the split of the root of the shuffle index by function **EvaluateRootSplit** does not violate the property. If the root node is full, function **EvaluateRootSplit** generates $num_shares$ nodes (a new root node and $num_shares - 1$ additional nodes that, along with the old root, become the children of the new root). The function moves $Cache_{l-1}$ to $Cache_l$ for each $l$=$h, \ldots, 1$ (lines 20-21, Figure 15). Since $Cache_l$, $l = 2, \ldots, h$, is no more updated by the function, it includes $num_cache$ nodes that belong to level $l$ in the tree. The function then inserts a subset of the $num_shares$ nodes children of the new root into $Cache_1$, which are the nodes with at least one child in $Cache_2$. Since $Cache_2$ stores $num_cache$ nodes and each node in the shuffle index (but the root) has exactly one parent, the function will insert into $Cache_1$ at most $num_cache$ nodes (lines 24-27, Figure 15). Also, if $Cache_1$ includes less than $num_cache$ nodes, the function inserts the nodes necessary to fill in the cache level among the nodes resulting from the split that do not have a child in $Cache_2$ (line 28, Figure 15). Therefore, we can conclude that also $Cache_1$ includes $num_cache$ nodes at level 1 in the shuffle index.

Second, we prove that also the split of a non-root node $n$ by procedure **EvaluateNodesSplit** does not violate this property. If $n$ is not in cache, procedure **EvaluateNodesSplit** does not update the cache. Consequently, if before calling procedure **EvaluateNodesSplit** the property is satisfies, then it holds also when the procedure terminates. Otherwise, if $n$ is in $Cache_l$ the procedure either keeps $n$ in $Cache_l$, replaces $n$ with $n'$, or keeps both $n$ and $n'$ in $Cache_l$ (lines 29-45, Figure 16). Nodes $n$ and $n'$ are clearly nodes that belong to level $l$, and therefore $Cache_l$ contains only nodes at level $l$ also after the split. If only one of the two nodes $n$ and $n'$ is kept in $Cache_l$, then $Cache_l$ clearly includes $num_cache$ nodes when procedure **EvaluateNodesSplit** terminates. If both $n$ and $n'$ are in $Cache_l$, then the node least recently used in $Cache_l$ is removed, and therefore again $Cache_l$ contains $num_cache$ nodes.

Finally, we prove by induction that the main search algorithm in Figure 13 satisfies this property at each iteration of the **for** loop (line 16, Figure 13). At the first iteration of the **for** loop, if $l$=1, $Read$ contains only nodes that are children of the root of the $B+$-tree, since their identifiers are obtained by calling function **ChildToFollow** on the root node $n_0$. Since the node inserted into $Cache_1$ belongs to $Read$, after the first iteration $Cache_1$ contains exactly $num_cache$ nodes at level $1$. If $l$=2 because the root node has been split, both $Read$ and $Cache_2$ contain only nodes that are children of the nodes resulting from the split of the root node. Indeed, when function **EvaluateRootSplit** terminates, $Cache_1$ stores the nodes resulting from the split that have a direct descendant in $Cache_2$, and $Non_Cached_P$ stores all the other nodes resulting from the split. Since $Read$ is defined by calling function **ChildToFollow** on a node that belongs either to $Cache_1$ or to $Non_Cached_P$, the nodes in $Read$ are all at level 2. Assume now by induction that, at iteration $l-1$, $Cache_{l-1}$ contains $num_cache$ nodes at level $l-1$ and that $Read$ contains only nodes at level $l-1$. We prove that at iteration $l$, $Cache_l$ contains $num_cache$ nodes at level $l$ and that $Read$ contains only nodes at level $l$. The nodes in $Read$ are obtained by calling function **ChildToFollow** on a node that belongs either to $Cache_{l-1}$ or to $Non_Cached_P$. Since $Non_Cached_P$ contains the nodes that were in $Read$ at iteration $l-1$, or the result of their split, the nodes in $Read$ are direct descendants of the nodes in $Cache_{l-1}$ and in $Non_Cached_P$. Therefore, they belong to level $l$ of the unchained $B+$-tree. Since the node inserted into $Cache_l$ belongs to $Read$, $Cache_l$ contains only nodes at level $l$. As already noted, the insertion of a node into $Cache_l$ causes the removal of another node in cache. Therefore, the number of nodes in

cache is always $num_cache$ for each level $l>0$. We conclude that the second property in Definition 4.2 is satisfied.

*3)* $\forall n \in Cache_l$, $l = 1,\dots,h$, *the parent of* $n$ *in the unchained* $B+$*-tree belongs to* $Cache_{l-1}$ (*path continuity property*). We now prove, by induction, that if the cache satisfies this property, any update performed by the main search algorithm, by function **EvaluateRootSplit**, and by procedure **EvaluateNodesSplit** to the cache does not violate it. The path continuity property between $Cache_0$ and $Cache_1$ is always satisfied since $Cache_1$ includes only nodes at level 1 (by the second property in Definition 4.2 proved above), and $Cache_0$ stores the only node in the shuffle index at level 0 (i.e., the root node), which is the direct ancestor of any node at level 1. Assume now that the path continuity property is satisfied for any pair of sets in $Cache_0,\dots,Cache_{l-1}$. The split of a node $n$ in $Cache_l$ implies that the node after the split, the resulting new node $n'$ or both of them are kept in $Cache_l$. Their presence in $Cache_l$ does not violate the path continuity property between $Cache_{l-1}$ and $Cache_l$. In fact, the parent of node $n$, which belongs to $Cache_{l-1}$, is modified by adding the pointer to $n'$. Therefore, the ancestor of $n$ and $n'$, possibly appearing in $Cache_l$, belongs to $Cache_{l-1}$. The split of node $n$ also preserves, as already noted (see the Proof of Theorem 8.2), the path continuity between $Cache_l$ and $Cache_{l+1}$. In fact, we have the guarantee that if node $n$, $n'$, or both of them have at least one child in $Cache_{l+1}$, $n$, $n'$, or both are kept in $Cache_l$. Note also that if both $n$ and $n'$ are in $Cache_l$, node $deleted$ is removed from $Cache_l$. However, $deleted$ cannot have a descendant in $Cache_{l+1}$ since $deleted$ is the least recently used node in $Cache_l$ and $Cache_{l+1}$ includes at least one node descendant of node $n$ and one node descendant of node $n'$ (meaning that at least a node in $Cache_l$ does not have a child in $Cache_{l+1}$). If $deleted$ has a child in $Cache_{l+1}$, it cannot be the least recently used since otherwise there would be another node $n''$ in $Cache_l$ with a child at level $l$+1 that has been pushed out from the cache to make space for the descendants of $n$ and $n'$. The timestamp of $n''$ and its child are the same and would be older than the one associated with $deleted$. When procedure **EvaluateNodesSplit** terminates the path continuity property then holds. At iteration $l$, if $cache_hit$ is true, the timestamp associated with a node in $Cache_l$ is updated. Then, the path continuity property is satisfied. Otherwise, if $cache_hit$ is false, the oldest node in $Cache_l$ is removed from the cache and the node with identifier $target_id$ is inserted into $Cache_l$. It is easy to see that the removal of the oldest node from $Cache_l$ does not affect the path continuity property between $Cache_{l-1}$ and $Cache_l$. We need therefore to prove that, if the node $n$ with identifier $target_id$ is inserted into $Cache_l$ (line 46, Figure 13), its direct ancestor, say $n_a$, belongs to $Cache_{l-1}$. As shown by the proof of Theorem 8.2, $n_a$ belongs to either $Cache_{l-1}$ or $Non_Cached_P$. Since at iteration $l$-1 the node with identifier $target_id$ is, by definition, $n_a$ either the timestamp of $n_a$ is updated or $n_a$ is inserted into $Cache_{l-1}$. However, if $n_a$ is inserted into $Cache_{l-1}$, the oldest node in $Cache_{l-1}$, say $n_o$, is removed from the cache. There may exist a node in $Cache_l$ that does not have a direct ancestor in $Cache_{l-1}$. However, $n_a$ is inserted into $Cache_{l-1}$ only if $cache_hit$ is false. Since $cache_hit$ cannot become true during the **for** loop (line 16, Figure 13), at iteration $l$ $cache_hit$ is still false. Consequently, $n$ is inserted into $Cache_l$ while the oldest element is removed from the cache. Since the timestamp of the nodes in cache is not affected by node split (the nodes resulting from the split inherit the same timestamp of their most recently accessed descendant when inserted into the cache, lines 38 and 40, Figure 16) and is updated at each iteration, the node removed from $Cache_l$ is a descendant of $n_o$ and the path continuity property is satisfied. □

THEOREM 8.4 (COMPUTATIONAL COMPLEXITY). The algorithm in Figure 2 operates in $O(F(num_cover + num_cache)^2 \log_F(m))$ time, where $m$ is the number of blocks in the shuffle index and $F$ is its fan out.

PROOF. The **for** loop initializing array $cover_id$ (line 8, Figure 13) and the **for** loop choosing cover searches (lines 10-14, Figure 13) cost $O(num_cover)$. The call to function **EvaluateRootSplit** in Figure 15 costs $O(F \cdot num_cache + h)$, as illustrated in the following.

The split of node $n$ into $num_shares$ nodes (lines 6-18, Figure 15) costs $O(F)$ since it is necessary to scan the values and pointers to children in $n$. The update of the cache structure to insert a new layer costs instead $O(h)$, since it is necessary to scan all the levels in the cache (lines 20-21, Figure 15). The update of $Cache_1$ (lines 24-28, Figure 15) costs $O(F \cdot num_cache)$, since it is necessary to determine which among the $num_shares$ nodes generated from the split operation have a descendant in $Cache_2$. In fact, $Cache_2$ includes $num_cache$ elements and the children of the original root node, which are the same as the children of the nodes resulting from the split, are exactly $F$. The update of variable $target_id$ costs $O(F)$, which is the cost of calling function **ChildToFollow** that scans all the pointers and values in a node (line 30, Figure 15). The cost of identifying the node in the path to $target_value$ among the $num_shares$ nodes resulting from the split operation is $O(num_shares)$ (line 31, Figure 15). Finally, the cost of updating the $Cache_1$ to include the target node of the search operation is $O(num_cache)$ (lines 32-36, Figure 15). The cost of function **EvaluateRootSplit** is therefore $O(F \cdot num_cache) + O(h) + O(num_shares) = O(F \cdot num_cache + h)$, since $num_shares \leq F$ and therefore $O(F \cdot num_cache)$ dominates $O(num_shares)$.

The **for** loop scanning the shuffle index level by level performs $1{+}num_cover{+}num_cache$ parallel searches (lines 26-48, Figure 13). Among the operations executed within this **for** loop, the call to procedure **EvaluateNodesSplit** in Figure 16 and the two **for each** loops scanning the nodes in $Read \cup Cache_l$ (line 34, Figure 13) and $Cache_{l-1} \cup Non_Cached_P$ (lines 36-38, Figure 13), respectively, dominate the cost of each iteration. The call to procedure **EvaluateNodesSplit** in Figure 16 costs $O((num_cover + num_cache) \cdot (num_cover + F \cdot num_cache))$, as illustrated in the following. The **for each** loop is executed $(1 + num_cover + num_cache)$ times. The split of node $n$ (lines 5-16, Figure 16) costs $O(F)$, since it is necessary to scan the values and pointers to children in $n$. The update of the parent $parent$ of node $n$ (lines 17-25, Figure 16) costs $O(num_cover + num_cache + F)$, since it is necessary first to search for $parent$ in $Cache_{l-1} \cup Non_Cached$ (which includes $(1 + num_cover + num_cache)$ nodes) and then to scan the values and pointers to children in $parent$. The update of variable $target_id$ costs $O(1)$ (line 27, Figure 16). The update of $Cache_l$ if $n$ is in cache (lines 29-45, Figure 16) costs $O(F \cdot num_cache)$, due to the scan of the nodes in $Cache_{l+1}$ against the pointers in $n.pointers$ and $n'.pointers$. The update of variables $cover_id[i = 1, \ldots, num_cover]$ (lines 48-51, Figure 16) costs $O(num_cover)$, due to the scan of $cover_id[i = 1, \ldots, num_cover]$. The cost of procedure **EvaluateNodesSplit** is then $O((num_cover + num_cache) \cdot ((num_cover + num_cache) + F \cdot num_cache)) = O((num_cover + num_cache) \cdot (num_cover + F \cdot num_cache))$.

The computational complexity of the two **for each** loops scanning the nodes in $Read \cup Cache_l$ (line 34, Figure 13) and $Cache_{l-1} \cup Non_Cached_P$ (lines 36-38, Figure 13) is $O(num_cover + num_cache)$ and $O(F \cdot (num_cover + num_cache))$. In fact, since both $Read \cup Cache_l$ and $Cache_{l-1} \cup Non_Cached_P$ contain at most $1{+}num_cover{+}num_cache$ nodes for every level $l$ and the second loop scans all the pointers in each visited node. The overall time complexity of the **for** loop scanning the shuffle index level by level is therefore $O(F \cdot (num_cover + num_cache)^2 \cdot h)$. The **for each** loop writing on the server the nodes at level $h$ (line 50, Figure 13) costs $O(num_cover + num_cache)$, since it scans the nodes in $Cache_h \cup Non_Cached$.

Overall, the time complexity of the algorithm is dominated by the **for** loop scanning the shuffle index and is then $O(F \cdot (num_cover + num_cache)^2 \cdot h)$ = $O(F \cdot (num_cover +$

$num_cache)^2 \cdot \log_F(m))$, since the shuffle index represents an unchained $B+$-tree with $m$ nodes and fan out $F$. □

## REFERENCES


D. Agrawal, A. El Abbadi, and S. Wang. 2013. Secure and Privacy-Preserving Database Services in the Cloud. In *Proc. of the 29th International Conference on Data Engineering (ICDE 2013)*. Brisbane, Australia.

R. Agrawal, J. Kierman, R. Srikant, and Y. Xu. 2004. Order Preserving Encryption for Numeric Data. In *Proc. of the 30th ACM International Conference on Management of Data (SIGMOD 2004)*. Paris, France.

V. Atluri, B. Shafiq, S. Ae Chun, G. Nabi, and J. Vaidya. 2011. UICDS-based information sharing among emergency response application systems. In *Proc. of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times (DG.O 2011)*. College Park, MD, USA.

M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. 1997. A Concrete Security Treatment of Symmetric Encryption. In *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*. Miami Beach, Florida, USA.

M. Bellare and C. Namprempre. 2008. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology* 21, 4 (2008), 469–491.

A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. 2011. DepSky: Dependable and Secure Storage in a Cloud-of-clouds. In *Proc. of the 6th Conference on Computer Systems (EuroSys 2011)*. Salzburg, Austria.

K.D. Bowers, A. Juels, and A. Oprea. 2009. HAIL: A High-availability and Integrity Layer for Cloud Storage. In *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*. Chicago, Illinois, USA.

A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. 2005. Modeling and Assessing Inference Exposure in Encrypted Databases. *ACM Transactions on Information and System Security* 8, 1 (2005), 119–152.

Y.C. Chang and M. Mitzenmacher. 2005. Privacy Preserving Keyword Searches on Remote Encrypted Data. In *Proc. of the 3rd International Conference on Applied Cryptography and Network Security (ACNS 2005)*. New York, NY, USA.

R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. 2006. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*. Alexandria, VA, USA.

E. Damiani, S. De Capitani Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. 2003. Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. In *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*. Washington, DC, USA.

S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. 2011a. Efficient and Private Access to Outsourced Data. In *Proc. of the 31st International Conference on Distributed Computing Systems (ICDCS 2011)*. Minneapolis, Minnesota, USA.

S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. 2011b. Supporting Concurrency in Private Data Outsourcing. In *Proc. of the 16th European Symposium on Research in Computer Security (ESORICS 2011)*. Leuven, Belgium.

S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. 2013a. Distributed Shuffling for Preserving Access Confidentiality. In *Proc. of the 18th European Symposium on Research in Computer Security (ESORICS 2013)*. Egham, U.K.

S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. 2013b. Supporting Concurrency and Multiple Indexes in Private Access to Outsourced Data. *Journal of Computer Security (JCS)* 21, 3 (2013), 425–461.

X. Ding, Y. Yang, and R.H. Deng. 2011. Database Access Pattern Protection Without Full-Shuffles. *IEEE Transactions on Information Forensics and Security* 6, 1 (2011), 189–201.

O. Goldreich and R. Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P.J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proc. of the 20th ACM International Conference on Management of Data (SIGMOD 1994)*. Minneapolis, MN, USA.

H. Hacigümüs, B. Iyer, and S. Mehrotra. 2002a. Providing Database as a Service. In *Proc. of the 18th International Conference on Data Engineering (ICDE 2002)*. San Jose, CA, USA.

H. Hacigümüs, B. Iyer, and S. Mehrotra. 2004. Efficient Execution of Aggregation Queries over Encrypted Relational Databases. In *Proc. of the 9th International Conference on Database Systems for Advances Applications (DASFAA 2004)*. Jeju Island, Korea.

H. Hacigümüs, B. Iyer, S. Mehrotra, and C. Li. 2002b. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proc. of the ACM International Conference on Management of Data (SIGMOD 2002)*. Madison, WI, USA.

B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. 2012. Secure Multidimensional Range Queries over Outsourced Data. *The VLDB Journal* 21, 3 (2012), 333–358.

M.S. Islam, M. Kuzu, and M. Kantarcioglu. 2014. Inference Attack Against Encrypted Range Queries on Outsourced Databases. In *Proc. of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY 2014)*. San Antonio, TX, USA.

M. Kandias, N. Virvilis, and D. Gritzalis. 2011. The Insider Threat in Cloud Computing. In *Proc. of the 6th Conference on Critical Information Infrastructures Security (CRITIS 2011)*. Lucerne, Switzerland.

P. Lin and K.S. Candan. 2004a. Hiding Traversal of Tree Structured Data from Untrusted Data Stores. In *Proc. of the 2nd International Workshop On Security in Information Systems (WOSIS 2004)*. Porto, Portugal.

P. Lin and K.S. Candan. 2004b. Secure and Privacy Preserving Outsourcing of Tree Structured Data. In *Proc. of the 1st International Conference on Secure Data Management (SDM 2004)*. Toronto, Canada.

R. Ostrovsky and W. E. Skeith, III. 2007. A Survey of Single-Database Private Information Retrieval: Techniques and Applications. In *Proc. of the 10th International Conference on Practice and Theory in Public-Key Cryptography (PKC 2007)*. Beijing, China.

H. Pang, J. Zhang, and K. Mouratidis. 2013. Enhancing Access Privacy of Range Retrievals over $B+$-Trees. *IEEE Transactions on Knowledge and Data Engineering* 25, 7 (2013), 1533–1547.

K. Ren, C. Wang, and Q. Wang. 2012. Security Challenges for the Public Cloud. *IEEE Internet Computing* 16, 1 (2012), 69–73.

E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes. 2005. Designing Secure Indexes for Encrypted Databases. In *Proc. of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2005)*. Storrs, CT, USA.

B. W. Silverman. 1986. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall Monographs on Statistics & Applied Probability. 1st edition.

R. Sion and B. Carbunar. 2007. On the Computational Practicality of Private Information Retrieval. In *Proc. of the 14th Annual Network & Distributed System Security Conference (NDSS 2007)*. San Diego, CA, USA.

D.X. Song, D. Wagner, and A. Perrig. 2000. Practical Techniques for Searches on Encrypted Data. In *Proc. of the 21st IEEE Symposium on Security and Privacy (S&P 2000)*. Berkeley, CA, USA.

E. Stefanov and E. Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *Proc. of the 34th IEEE Symposium on Security and Privacy (S&P 2013)*. San Francisco, CA, USA.

E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proc. of the 20th ACM Conference on Computer and Communications Security (CCS 2013)*. Berlin, Germany.

W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y.T. Hou, and H. Li. 2013. Privacy-Preserving Multi-Keyword Text Search in the Cloud Supporting Similarity-Based Ranking. In *Proc. of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2013)*. Hangzhou, China.

C. Wang, N. Cao, J. Li, K. Ren, and W. Lou. 2010. Secure Ranked Keyword Search over Encrypted Cloud Data. In *Proc. of the 30th International Conference on Distributed Computing Systems (ICDCS 2010)*. Genoa, Italy.

C. Wang, N. Cao, K. Ren, and W. Lou. 2012. Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data. *IEEE Transactions on Parallel and Distributed Systems* 23, 8 (2012), 1467–1479.

H. Wang and L.V.S. Lakshmanan. 2006. Efficient Secure Query Evaluation over Encrypted XML Databases. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB 2006)*. Seoul, Korea.

S. Wang, D. Agrawal, and A. El Abbadi. 2011. A Comprehensive Framework for Secure Query Processing on Relational Data in the Cloud. In *Proc. of the 8th International Conference on Secure Data Management (SDM 2011)*. Seattle, WA, USA.

Z.F. Wang, W. Wang, and B.L. Shi. 2005. Storage and Query over Encrypted Character and Numerical Data in Database. In *Proc. of the 5th International Conference on Computer and Information Technology (CIT 2005)*. Shanghai, China.

P. Williams, R. Sion, and B. Carbunar. 2008. Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *Proc of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*. Alexandria, VA, USA.

P. Williams, R. Sion, and A. Tomescu. 2012. PrivateFS: A Parallel Oblivious File System. In *Proc. of the ACM Conference on Computer and Communications Security (CCS 2012)*. Raleigh, NC, USA.

K. Yang, J. Zhang, W. Zhang, and D. Qiao. 2011. A Light-Weight Solution to Preservation of Access Pattern Privacy in Un-Trusted Clouds. In *Proc. of the 16th European Symposium on Research in Computer Security (ESORICS 2011)*. Leuven, Belgium.