Distributed Shuffling for Preserving Access Confidentiality

Sabrina De Capitani di Vimercati[1], Sara Foresti[1], Stefano Paraboschi[2], Gerardo Pelosi[3], and Pierangela Samarati[1]

[1] Università degli Studi di Milano, 26013 Crema, Italy
firstname.lastname@unimi.it
[2] Università degli Studi di Bergamo, 24044 Dalmine, Italy
parabosc@unibg.it
[3] Politecnico di Milano, 20133 Milan, Italy
gerardo.pelosi@polimi.it

Abstract. The shuffle index has been recently proposed for organizing and accessing data in outsourcing scenarios while protecting the confidentiality of the data as well as of the accesses to them. In this paper, we extend the shuffle index to the use of multiple servers for storing data, introducing a new protection technique (shadow) and enriching the original ones by operating in a distributed scenario. Our distributed shuffle index produces a significant increase in the protection of the system, with no additional costs.

1 Introduction

Recent years have witnessed an over increasing reliance on external services for data storage and management, towards the emerging cloud scenario, characterized by a rich and diverse availability of providers offering storage and computational functionalities. Together with data management functionality, the research and industrial communities have been investigating different solutions to ensure confidentiality of data whose management is outsourced to the cloud [8]. Complementing data confidentiality, more recent approaches have also considered protection of *access* and *pattern confidentiality*, which require to maintain confidential to the server storing the data the fact that a given access aims at a specific target or that two accesses aim at the same target. Among these approaches, the *shuffle index* [5] organizes data in a hierarchical encrypted data structure and provides access and pattern confidentiality by obfuscating accesses and dynamically changing the allocation of data to physical blocks, so to break the correspondence between data and locations where they are stored. Such a dynamic allocation prevents the server observing sequences of accesses from withdrawing inferences which could compromise pattern confidentiality and even break data confidentiality. The advantages of the shuffle index are the ability to support equality and range predicates in data retrieval, and the limited performance overhead compared with other access protection solutions [7].

The availability of different providers for data outsourcing can help in providing protection in cloud scenarios and has been investigated in some proposals, adopting, for example, data fragmentation and slicing at different servers (e.g., [3, 16]). The intuition is that relying on multiple servers (in contrast to a single one) for managing data or providing services naturally increases protection, since it diminishes the knowledge and visibility that each server has on the data and on accesses to them, and enjoys diversity of risks.

In this paper, we extend the shuffle index to operate with multiple servers for storing and accessing data. Every data access entails accessing the servers and shuffling data dynamically changing their allocation even across servers. Since retrieval of the targeted data may entail traversing the hierarchical structure across servers (i.e., a parent might be stored at one server and a children at another), we introduce a *shadowing* technique that ensures protection of this path information by making observations by each server as if the server was the only one involved in the access. The distribution of the shuffle index increases protection for data and accesses, quickly destroying knowledge that servers might have and effectively preventing the servers from acquiring knowledge by observing sequences of accesses. Such increased protection comes without impact on the system performance.

The remainder of the paper is organized as follows. Section 2 recalls the basic concepts of the shuffle index on which we build. Section 3 extends the index organization to the adoption of multiple servers. Section 4 introduces shadows and extends the original protection techniques (covers, cache, and shuffling) to operate with them. Section 5 describes access execution. Section 6 discusses the protection offered by our approach illustrating how distributing the shuffle index provides greater confidentiality guarantees while not impacting performance. Section 7 illustrates related work. Section 8 concludes the paper.

2 Basic Concepts

A shuffle index [5] organizes the outsourced data as an abstract *unchained* B+-*tree* $\mathcal{T}^a(\mathcal{N}^a)$ (i.e., leaves are not connected in a linked list) with fan out F defined over a candidate key K, with actual data stored in the leaves of the tree. Each internal node of the index is a pair $n^a = \langle \mathit{values}, \mathit{children} \rangle \in \mathcal{N}^a$, where *values* is a list of q values with $\lceil \frac{F}{2} \rceil - 1 \leq q \leq F - \mathit{1}$ (the lower-bound does not apply to the root) ordered from the smallest to the greatest, and *children* is a list of $q + 1$ children. The first child of a node is the root of the subtree with all values $v < \mathit{values}[\mathit{1}]$; the i-th child is the root of the subtree storing the values v such that $\mathit{values}[i - \mathit{1}] \leq v < \mathit{values}[i]$, $i = 2, \ldots, q$; the last child is the root of the subtree with all values $v \geq \mathit{values}[q]$. Leaf nodes are pairs $n^a = \langle \mathit{values}, \mathit{tuples} \rangle \in \mathcal{N}^a$, where *tuples* represents the tuples with index value in *values*. Figure 1(a) illustrates an example of unchained B+-tree with fan out 3. For simplicity, we refer to the content of a node with a label (e.g., a), instead of explicitly reporting the values it represents.

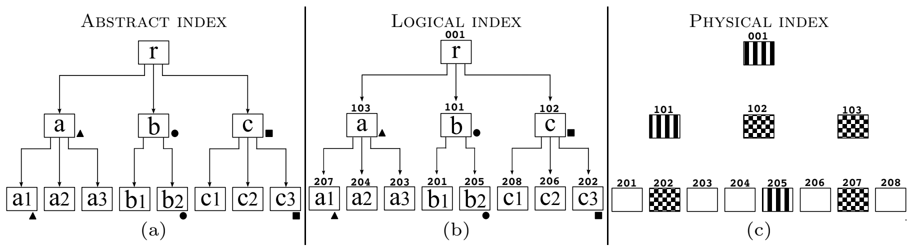


Fig. 1. An example of abstract (a), logical (b), and physical (c) shuffle index
Legend: ■ target, • node in cache, ▲ cover; blocks read and written: chessboard filling, blocks written: lines filling

At the *logical level*, nodes are allocated to logical addresses that work as logical *identifiers*. Given an abstract unchained B+-tree $\mathcal{T}^a(\mathcal{N}^a)$, its logical representation $\mathcal{T}$ is a triple $(\mathcal{N},\mathcal{ID},\phi)$, with $\mathcal{N}$ a set of logical nodes, $\mathcal{ID}$ a set of logical identifiers, and $\phi : \mathcal{N}^a \rightarrow \mathcal{ID}$ a bijective function that associates a logical identifier with each abstract node. Note that the possible order among identifiers does not necessarily correspond to the order in which nodes appear in the value-ordered abstract representation. Each non-leaf abstract node $n^a = \langle values, children\rangle$ in $\mathcal{N}^a$ is represented by a logical node $n{=}\langle id, vals, ptrs\rangle$ in $\mathcal{N}$, with $id{=}\phi(n^a)$, $vals{=}values$, and $ptrs[j]{=}\phi(children[j])$, $j = 0,\dots,q$. In fact, pointers to the children of the nodes in the abstract unchained B+-tree are translated, at the logical level, into the identifiers of the corresponding child nodes. Analogously, each abstract leaf node $n^a = \langle values, tuples\rangle$ in $\mathcal{N}^a$ translates into a logical node $n{=}\langle id, vals, t\rangle$ that includes tuples $t{=}tuples$ instead of pointers to children. Figure 1(b) illustrates an example of logical representation of the abstract index in Figure 1(a). Logical identifiers are reported on the top of each node and, for easy reference, their first digit denotes its level in the tree.

At the *physical level*, logical identifiers are mapped to physical addresses and the shuffle index is represented by a set of disk *blocks* storing the nodes in the tree. Every node is encrypted by first prefixing it with a random salt and then applying symmetric encryption in CBC mode. Formally, each non-leaf node $\langle id, vals, ptrs\rangle \in \mathcal{N}$ (leaf node $\langle id, vals, t\rangle \in \mathcal{N}$, resp.) is stored at block $\langle id,b\rangle$, where $b{=}E_k(salt||id||vals||ptrs)$ ($b{=}E_k(salt||id||vals||t)$, resp.), with E a symmetric encryption function, k the encryption key, and *salt* a nonce generated for each encryption. Figure 1(c) illustrates the physical representation of the logical index in Figure 1(b), which corresponds to the view of the server.

The retrieval of the leaf block containing the tuple corresponding to a given index value (target value) requires an iterative process. Starting from the root of the tree and ending at a leaf, the client reads from the server the block in the path to the target, and decrypts the block for retrieving the address of the child to be read at the next step. To protect the fact that different accesses may aim at the same content, this iterative process is extended by:

– performing, in addition to the target search, other fake *cover searches*, guaranteeing indistinguishability of target and cover searches and operating on disjoint paths of the tree (retrieving, at every level of the tree, *num_cover*+1 blocks at the same time);
– maintaining a set of *num_cache* nodes in a local *cache* for each level of the tree, but level 0;
– mixing (*shuffling*) the content of all retrieved blocks as well as those maintained in cache, and overwriting them accordingly on the server.

Cover searches protect the confidentiality of accesses by introducing uncertainty on the leaf block target of the access (any of the accessed leaves could store the searched value). The cache makes searches repeated within a short time interval not recognizable as such. In fact, if the target of an access is in cache, the corresponding block is not read from the server (the target is substituted by an additional cover). Shuffling destroys the correspondence between nodes and the physical blocks where they are stored. (Note that at every reallocation, a node is encrypted with a different random salt.) Repeated accesses to the same block do not then imply repeated accesses to the same node. As an example of access to the shuffle index in Figure 1, consider a search for *c3* that adopts *a1* as cover, and assume that the cache contains the path to *b2*. The access visits the tree level by level. The client has the root r in cache, downloads and decrypts blocks 102 and 103 from the server, shuffles and encrypts nodes a, b, and c (e.g., allocating a to 102, b to 101, and c to 103), and overwrites blocks 101, 102, and 103 at the server. At the leaf level, the client downloads and decrypts blocks 202 and 207, shuffles and encrypts nodes *a1*, *b2*, and *c3*, and overwrites blocks 202, 205, and 207 at the server. Figure 1(c) illustrates the observations on the access at the server in terms of blocks read and/or written. Note that the root (being in cache) is only written. The server cannot detect which among the accessed leaves is the target of the access and how the content of blocks has been shuffled.

3 Distributed Shuffle Index

In a *distributed shuffle index*, the data owner exploits more than one server for storing and managing data, enjoying then increased protection of data, access, and pattern confidentiality by dynamically changing the allocation of the nodes also across the servers. For simplicity, we illustrate our distributed shuffle index assuming the use of *two* servers, with the note that the approach can be easily extended to the consideration of an arbitrary number of servers. For simplicity of notation and clarity of the figures, we denote our servers by $\mathcal{S}_G$ and $\mathcal{S}_Y$, coloring nodes stored at them with *Green* and *Yellow*, respectively (in b/w printouts, Green is the darker color).

The consideration of more than one server for the allocation of an abstract index and for accesses to it requires revising the shuffle index structure discussed in Section 2 with the following extensions.

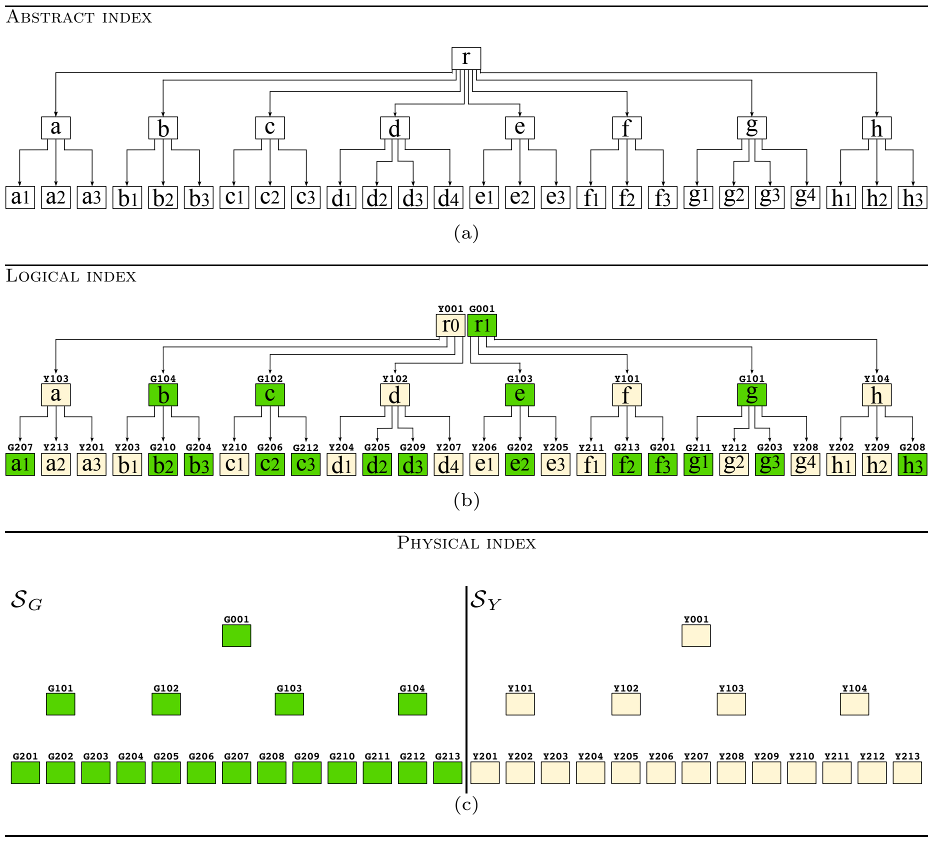
– *Abstract level.* The root r^a of a distributed shuffle index is extended to have twice the capacity as the other nodes. Hence, for an index with fan out F, the

root can contain up to $2F-1$ values (in contrast to the original $F-1$). In the translation to the logical level, the abstract root $r^a = \langle values, children\rangle$ will be interpreted as two abstract root nodes, r^a_0 and r^a_1, each storing around half of the values and children in r^a. Formally, $r^a_0 = \langle values_0, children_0\rangle$ and $r^a_1 = \langle values_1, children_1\rangle$ with $values_0{=}values[1,\ldots,\lceil q/2\rceil]$, $values_1{=}values[\lceil q/2\rceil + 2,\ldots,q]$, $children_0{=}children[0,\ldots,\lceil q/2\rceil]$, and $children_1{=}children[\lceil q/2\rceil + 1,\ldots,q]$, where q is the number of index values in the abstract root. (Note that $values[\lceil q/2\rceil + 1]$ disappears since it is no more needed for the index.) The set $\mathcal{N}^a$ of abstract nodes therefore becomes $\mathcal{N}^a{=}\mathcal{N}^a\setminus\{r^a\}\cup\{r^a_0,r^a_1\}$.
- *Logical level.* The logical identifiers of a distributed shuffle index must take into consideration logical identifiers (which translate to physical addresses) at the two servers. We then distinguish, in the set $\mathcal{ID}$ of logical identifiers, two different subsets: $\mathcal{ID}_G$, corresponding to addresses at server $\mathcal{S}_G$, and $\mathcal{ID}_Y$, corresponding to addresses at server $\mathcal{S}_Y$, with $\mathcal{ID}_G \cup \mathcal{ID}_Y = \mathcal{ID}$. The result of function ϕ over an abstract node, determining the logical identifier of the node, therefore determines also the server at which the abstract node is stored. Function ϕ guarantees the natural requirement to store r^a_0 and r^a_1 at different servers. Formally, $\phi(r^a_0){\in}\mathcal{ID}_X$ and $\phi(r^a_1){\in}\mathcal{ID}_Z$, with $X, Z \in \{Y, G\}$ and $X \neq Z$. In the following, given a node n in the set $\mathcal{N}$ of logical nodes, we will use $\sigma(n.id)$ to denote the server at which the node is stored. Formally, given $n = \langle id, vals, ptrs\rangle$, with $id{=}\phi(n^a)$, $id{\in}\mathcal{ID}_G \Longrightarrow \sigma(n.id){=}\mathcal{S}_G$; $id{\in}\mathcal{ID}_Y \Longrightarrow \sigma(n.id){=}\mathcal{S}_Y$.
- *Physical level.* It works like in the original shuffle index, storing (according to allocation function ϕ defined at the logical level) nodes at each server in encrypted form as described in Section 2.

Figure 2 illustrates an example of abstract, logical, and physical distributed shuffle index. For simplicity and easy reference, logical identifiers start with a letter denoting the server where the corresponding block is stored (G for $\mathcal{S}_G$ and Y for $\mathcal{S}_Y$) and nodes stored at server $\mathcal{S}_G$ and $\mathcal{S}_Y$ are color-coded (*Green* and *Yellow*). In the following, without loss of generality, we assume that the physical address of a block corresponds to the logical identifier of the node it stores. Also, we use the term *node* to refer to an abstract content and *block* to refer to a specific memory slot in the logical/physical structure. When either terms can be used, we will use node/block interchangeably.

4 Shadows, Covers, Cache, and Shuffling

The shuffle index entails two types of protections. The first involves obfuscating the fact that the access aims at a specific block. The second is the shuffling, which changes the allocation of nodes so to dynamically modify the node/block correspondence. Both these types of protection, provided by cover searches, caching, and shuffling in the original proposal, are complemented in the distributed shuffle index with the consideration of *shadows*. In this section, we introduce shadows and extend cover searches, cache, and shuffling to operate with them.



Abstract index
r
a b c d e f g h
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3 d4 e1 e2 e3 f1 f2 f3 g1 g2 g3 g4 h1 h2 h3
(a)
Logical index
Y001 r0 G001 r1
Y103 a, G104 b, G102 c, Y102 d, G103 e, Y101 f, G101 g, Y104 h
G207 a1, Y213 a2, Y201 a3, Y203 b1, G210 b2, G204 b3, Y210 c1, G206 c2, G212 c3, Y204 d1, G205 d2, G209 d3, Y207 d4, Y206 e1, G202 e2, Y205 e3, Y211 f1, G213 f2, G201 f3, G211 g1, Y212 g2, G203 g3, Y208 g4, Y202 h1, Y209 h2, G208 h3
(b)
Physical index
$\mathcal{S}_G$
G001
G101 G102 G103 G104
G201 G202 G203 G204 G205 G206 G207 G208 G209 G210 G211 G212 G213
$\mathcal{S}_Y$
Y001
Y101 Y102 Y103 Y104
Y201 Y202 Y203 Y204 Y205 Y206 Y207 Y208 Y209 Y210 Y211 Y212 Y213
(c)

Fig. 2. An example of abstract (a), logical (b), and physical (c) shuffle index, distributed over two servers

4.1 Shadows

Retrieval of a target key value in a distributed shuffle index entails traversing the index starting from the root ancestor of the target and following, at every node, the pointer to the child in the path to the leaf possibly containing the target value. Such a path can naturally involve nodes stored at any of the servers. For instance, with reference to the shuffle index in Figure 2 a search for target value *e3* involves access to blocks (G001/Y001;G103;Y205), with G001 and G103 stored at $\mathcal{S}_G$, and Y001 and Y205 stored at $\mathcal{S}_Y$. (For simplicity, we assume both roots to be always accessed.) Like in the original shuffle index proposal, we assume each server, which initially knows only the number of blocks it stores, to potentially have knowledge of the height of the shuffle index and of the level of the node stored at each of its blocks (which can be acquired by observing the iterations in the accesses). Combined with such knowledge, discontinuity of accesses with respect to levels (e.g., for $\mathcal{S}_Y$) could leak information to the servers. To avoid such a leakage, in our approach, every time we need to access a block at one of the servers, we will also access another block, which we call *shadow*, at the other server. With shadows, each server will observe accesses to blocks as if it was the only one storing the data structure and involved in the access. With respect to this aspect, any block at the other server at the same level as the block for which it works as shadow would do. The choice of the shadow for a block during a given access, however, needs to take into account the fact that the shuffle index changes allocation of nodes at every access. Re-allocating a node n requires changing in its parent the pointer to n, to refer to the block where it has been moved. Therefore, the nodes involved in an access should always form a sub-tree. In other words, a shadow at a given level should be child of a node that is available for the access (i.e., read in the path to the target or to a cover, or available in cache).

A convenient way to ensure this requirement is to use, as shadow of a node for a given access, one of its siblings stored at the other server. For each node n, we call *far siblings* of n the children of the same parent (i.e., n's siblings) stored at a server different from the one where n is stored. We denote with $FS(n)$ the nodes having such properties at a given time with respect to a node n, as formally defined in the following.

Definition 1 (Far siblings and shadow). *Let* $\mathcal{T}(\mathcal{N},\mathcal{ID}_G,\mathcal{ID}_Y,\phi)$ *be a logical index, and* n *be a non-root node in* $\mathcal{N}$*. The* far siblings *of node* n*, denoted* $FS(n)$*, are defined as* $FS(n) = \{n_i \in \mathcal{N} : \text{ parent}(n_i) = \text{parent}(n) \text{ and } \sigma(n_i.id) \neq \sigma(n.id)\}$*. As particular cases,* $FS(r_0) = \{r_1\}$ *and* $FS(r_1) = \{r_0\}$*.*
The shadow *of a node for a given access is one of its far siblings selected for the specific access.*

For instance, with reference to the shuffle index in Figure 2(b), $FS(e) = \{f, h\}$, and $FS(e3)$=$\{e2\}$. Assuming to choose h and $e2$ as shadows for e and $e3$, respectively, the search for key value $e3$ will translate into accessing blocks (G001;G103;G202) at $\mathcal{S}_G$ and blocks (Y001;Y104;Y205) at $\mathcal{S}_Y$.

Note that the far siblings relationship is dynamic as every re-allocation of nodes, which will operate across servers, changes it. Also, the shadow relationship is dynamic as, at any given access to a node, any of its far siblings can be dynamically selected to serve as shadow. Although in principle nodes can be randomly allocated to servers, distributing allocation uniformly provides better protection and has the advantage that the set $FS(n)$ of a node n can never be empty. We then require uniform distribution between the two servers among the children of each node (storing half of the node's children at one server and half at the other). The allocation function ϕ must enforce a balanced allocation of nodes' children to the servers, satisfying the following property.

Property 1 (Balanced allocation). Let $\mathcal{T}(\mathcal{N},\mathcal{ID}_G,\mathcal{ID}_Y,\phi)$ be a logical index. $\mathcal{T}$ satisfies the *balanced allocation* property iff:

1. $\sigma(r_0.id) \neq \sigma(r_1.id)$ (i.e., the two roots are stored at a different server);
2. $\forall\langle id, vals, ptrs\rangle\in\mathcal{N}$: $|\text{card}(ptrs_G) - \text{card}(ptrs_Y)| \leq 1$,
   with $ptrs_G$=$\{id_i\in ptrs\text{: } id_i\in\mathcal{ID}_G\}$ and $ptrs_Y$=$\{id_i\in ptrs\text{: } id_i\in\mathcal{ID}_Y\}$.

The first condition in the property states that the two roots are stored at different servers. The second condition states that, for each node, half of its children are stored at server $\mathcal{S}_G$ and the other half at server $\mathcal{S}_Y$. Property 1 guarantees that every node n, child of a node with k children, has at least $\lfloor \frac{k}{2} \rfloor$ far siblings. It is easy to see that the distributed shuffle index in Figure 2 obeys to the balanced allocation property.

4.2 Covers

As in the original proposal, cover searches are fake searches executed in parallel with the search for the target value to the aim of hiding the target request within a group of other requests. The fact that the shuffle index is distributed has two effects with respect to covers: one is the extension of the definition of cover search, the other is the application of shadows to covers.

As in [5], the only constraint on covers chosen for an access is that they actually act as such, that is, they should be indistinguishable from actual searches, and their paths should not intersect or intersect the path to the target. The first aspect is already guaranteed from [5], the latter aspect simply requires extending the definition of cover searches to the consideration of the fact that the root is split, and therefore the constraint that paths of the covers have only the root in common translates into requiring that their paths have nothing in common but - possibly - any of the roots, as formally stated by the following definition.

Definition 2 (Cover searches). *Let $\mathcal{T}(\mathcal{N},\mathcal{ID}_G,\mathcal{ID}_Y,\phi)$ be a logical index built on candidate key K with domain $\mathcal{D}$, and v_0 be a value in $\mathcal{D}$. A set $\{v_1,\ldots,v_{num\_cover}\} \subseteq \mathcal{D}$ is a set of* cover searches *for v_0 iff $\forall path_i, path_j \in \{path_0,\ldots,path_{num\_cover}\}$, $i \neq j$: $(path_i \cap path_j)\setminus\{r_0,r_1\} = \emptyset$, where $path_i$ is the set of nodes in the path from r_0 or r_1 to the leaf where v_i is possibly stored.*

Note that the nodes in the paths to covers can be indifferently stored at one of the two servers. This does not create any problem in our approach. In fact, just like the target, covers will also be shadowed and for every node to be accessed in the path to a cover at a server, one of its far siblings will be accessed at the other server. In particular, for each level in the shuffle index, if a node n_c in the path to a cover is actually stored at a different server from the node n_t in the path to the target, n_c will act as a protection of n_t's shadow and n_c's shadow will act as a protection for n_t, respectively, at the two servers. The application of shadows to nodes in the path to covers nicely provides a symmetric behavior at the two servers, regardless of where these nodes are stored. In fact, a server will observe access to *num_cover + 1* different blocks for each level, but level 0.

As an example, consider the distributed shuffle index in Figure 2(b), and a search for *e3* (path (r_1;*e*;*e3*)), using *a2* as cover (path (r_0;*a*;*a2*)). Assuming to choose, in the set of its far siblings, *h* as shadow for *e*, *c* for *a*, *e2* for *e3*, and *a1* for *a2*, the accessed blocks are (G001;G102,G103;G202,G207) at $\mathcal{S}_G$ and (Y001;Y103,Y104;Y205,Y213) at $\mathcal{S}_Y$.

4.3 Cache

Caching works essentially like in the original proposal, maintaining a copy of the last *num_cache* target searches (where for each target search all nodes in the path to the target leaf are maintained). In addition to the actual targets, in our distributed scenario, we also store, in association with every node n in the path to the target, the node n' that acted as n's shadow last time n was accessed.

Formally, the cache of a distributed shuffle index is defined as follows.

Definition 3 (Cache). *Let $\mathcal{T}(\mathcal{N},\mathcal{ID}_G,\mathcal{ID}_Y,\phi)$ be a logical index with height h. A* cache *of size num_cache for $\mathcal{T}$ is a layered structure of h+1 sets $Cache_0,\ldots,Cache_h$ of pairs of nodes where:*

1. *$Cache_0$ contains pair $\langle r_i,r_j\rangle$ with $i,j \in \{0,1\}$ and $i \neq j$;*
2. *$Cache_l$, $l = 1,\ldots,h$, contains num_cache pairs of nodes $\langle n_i,n_j\rangle$ s.t. n_i and n_j belong to the l-th level of $\mathcal{T}$, with n_i and n_j far siblings one of the other (cache balancing);*
3. *$\forall\langle n_i,n_j\rangle\in Cache_l$, $l = 1,\ldots,h$, the node parent of n_i and n_j in the shuffle index belongs to $Cache_{l-1}$ (path continuity).*

Note how the path continuity requirement (Condition 3 in the definition), requesting that the parent of a cached node be also in cache and here extended to the consideration of shadows, does not impose any complication to the approach. As a matter of fact, the choice of the shadows among the far siblings of target nodes included in the cache nicely guarantees that their parent (being a target) is already in the cache by construction.

A nice advantage of including shadows in cache is that $Cache_l$, $l = 1,\ldots,h$, contains 2*num_cache* nodes, half of which are stored at $\mathcal{S}_G$ and the others are stored at $\mathcal{S}_Y$. This will provide a symmetric behavior of the access at the two servers, with each of them operating with a view as if it was the only one involved in the access (see Section 5). After the search illustrated in Section 4.2 for value *e3* over the distributed shuffle index in Figure 2(b), the cache includes the nodes in the path to *e3* and their shadows (i.e., $\langle r_1,r_0\rangle$, $\langle e,h\rangle$, and $\langle e3,e2\rangle$).

4.4 Shuffling

Shuffling aims at destroying the one-to-one correspondence between blocks and nodes stored in them. The idea is to randomly re-allocate all nodes available in an access (i.e., accessed as targets, covers, shadows or in cache) so to break the otherwise static relationship between nodes and blocks where they are stored. Shuffling is formally defined as follows.

Definition 4 (Shuffling). *Given a set $ID \subseteq \mathcal{ID}_G \cup \mathcal{ID}_Y$ of logical identifiers, a* shuffling, *denoted by π, over ID is a random permutation π: $ID \rightarrow ID$.*

The effect of a shuffling π: $ID \rightarrow ID$ over shuffle index $\mathcal{T}(\mathcal{N},\mathcal{ID}_G,\mathcal{ID}_Y,\phi)$ is that the corresponding abstract index remains unchanged while the allocation of some nodes (and the pointers to them in their parents) is changed. More precisely, each node $\langle id, vals, ptrs\rangle$ is updated as follows: $id{=}\pi(id)$ if $id{\in}ID$, it remains unchanged otherwise; and $\forall i = 0, \ldots, q$ with q the number of values in $vals$, $ptrs[i]{=}\pi(ptrs[i])$ if $ptrs[i]{\in}ID$, it remains unchanged otherwise.

Like in the original (non-distributed) shuffle index, shuffling is performed only within levels and not cross-levels, due to complications that would otherwise arise for updating pointers to children.

Also, in our distributed shuffle index, where nodes (accessed because in the paths to the target or to a cover, or present in cache) are always accompanied by a shadow, we need to ensure that shuffling does not compromise the balanced allocation of the index. We then require the shuffling to ensure balancing, as captured by the following property.

Property 2 (Balanced shuffling). Let $\mathcal{T}(\mathcal{N}, \mathcal{ID}_G, \mathcal{ID}_Y, \phi)$ be a logical index, and $\mathcal{P} = \{\langle n_1, n'_1\rangle, \ldots, \langle n_m, n'_m\rangle\}$ be a set of pairs of nodes in $\mathcal{N}$ s.t. $\forall\langle n_i, n'_i\rangle \in \mathcal{P}$, $\sigma(n_i.id){\neq}\sigma(n'_i.id)$. A shuffling π over $ID_{\mathcal{P}}{=}\{id : \exists\langle n_i, n'_i\rangle \in \mathcal{P}$ with $id{=}n_i.id$ or $id{=}n'_i.id\}$ is *balanced* iff $\forall\langle n_i, n'_i\rangle \in \mathcal{P}$, $\sigma(\pi(n_i.id)){\neq}\sigma(\pi(n'_i.id))$.

Balanced shuffling essentially guarantees that pairs of nodes provided as input and stored at different servers before the shuffling remain stored at different servers after the shuffling. Since we operate shuffling on pairs of nodes that are far siblings one of the other, balanced shuffling ensures that these pairs of nodes will remain as such after the shuffling (indeed, shuffling does not change the 'being child of' relationship over the abstract index). Note that this does not mean that the two nodes in a pair can only be swapped one with the other as shuffling can actually change the blocks to which they are allocated; the only constraint is that the two nodes do not end up being stored at the same server. It is then easy to see that a balanced shuffling guarantees that the shuffling does not compromise the balanced allocation of the shuffle index (Property 1).

We realize a balanced shuffling by: *i)* randomly shuffling the nodes allocated at each of the two servers separately; and *ii)* possibly swapping the allocation of a node and its shadow. The random shuffling (step *i)*) does not move nodes from $\mathcal{S}_G$ to $\mathcal{S}_Y$ or vice versa. The controlled swapping (step *ii)*) operates between pairs of nodes stored at the two servers: whenever a node allocated at $\mathcal{S}_G$ is moved to $\mathcal{S}_Y$, its shadow (which by definition is at $\mathcal{S}_Y$) is moved from $\mathcal{S}_Y$ to $\mathcal{S}_G$, and vice versa. More precisely, our shuffling works as follows.

- Consider a logical index $\mathcal{T}(\mathcal{N}, \mathcal{ID}_G, \mathcal{ID}_Y, \phi)$, a set $\mathcal{P} = \{\langle n_1, n'_1\rangle, \ldots, \langle n_m, n'_m\rangle\}$ of pairs of nodes in $\mathcal{N}$, and the set $ID_{\mathcal{P}}{=}\{id : \exists\langle n_i, n'_i\rangle \in \mathcal{P}$ with $id{=}n_i.id$ or $id{=}n'_i.id\}$ of their identifiers.
- Define an *intra-server shuffling* over $ID_{\mathcal{P}}$, $\pi_1 : ID_{\mathcal{P}} \rightarrow ID_{\mathcal{P}}$, such that $\forall id \in ID_{\mathcal{P}}$, $\sigma(\pi_1(id)) = \sigma(id)$.
- Randomly select a subset S of pairs of nodes in $\mathcal{P}$ for *inter-server swapping*.
- Return π over $ID_{\mathcal{P}}$ such that $\forall\langle n_i, n'_i\rangle \in \mathcal{P}$:
  - if $\langle n_i, n'_i\rangle \in S \Longrightarrow \pi(n_i.id) = \pi_1(n'_i.id)$ and $\pi(n'_i.id) = \pi_1(n_i.id)$;
  - if $\langle n_i, n'_i\rangle \notin S \Longrightarrow \pi(n_i.id) = \pi_1(n_i.id)$ and $\pi(n'_i.id) = \pi_1(n'_i.id)$.

Note that, while guaranteeing balancing, our shuffling can move a node to any block on which the shuffling is operating (either at the same or at a different server). This provides for a fast degradation of the correspondences between nodes and blocks, ensuring the protection of access and pattern confidentiality (see Section 6).

As an example, consider the shuffle index in Figure 2(b), reported in Figure 3(a) for the reader's convenience, and the set $\mathcal{P} = \{\langle r_0,r_1\rangle;\ \langle e,h\rangle,\ \langle a,c\rangle,\ \langle b,d\rangle;\ \langle e3,e2\rangle,\ \langle a2,a1\rangle,\ \langle b1,b2\rangle\}$ of pairs of nodes accessed by the search for value *e3* illustrated above. Figures 3(b) and (c) illustrate an example of intra-server shuffling π_1 over $ID_{\mathcal{P}}$ and of inter-server swapping, with $S=\{\langle r_0,r_1\rangle;\ \langle a,c\rangle,\langle e,h\rangle;\langle a1,a2\rangle\}$, respectively. It is easy to see that the resulting shuffling π satisfies the balancing property (e.g., $\sigma(r_0.id)=\mathcal{S}_G$ and $\sigma(r_1.id)=\mathcal{S}_Y$). Figure 3(d) illustrates the shuffle index after the shuffling.

5 Access Execution

The application of shadows, covers, cache, and shuffling when performing an access works in combination to ensure two kinds of protection: *i)* obfuscating the fact that the access aims at a specific block (shadows, covers, and cache); and *ii)* changing the allocation of nodes so to dynamically modify the node/block correspondence and therefore provide protection for future accesses. Access execution with our protection techniques works as follows.

Given a search for a target value v, we first choose a set of num_cover+1 cover searches for v (Definition 2), where the additional one is to be used if a node in the path to v is in cache. For each level l of the distributed shuffle index, we identify the blocks in the paths to covers and target and choose a shadow (Definition 1) for each of them. Like covers, shadows are chosen in such a way to ensure block diversity, meaning that they should not appear in the paths to the target and to covers, and should not be stored in $Cache_l$. Intuitively, block diversity guarantees that all the techniques play a role in providing protection as they will not end up clashing over the same blocks.

Enforcement of block diversity also on shadows, and application of shadows to both the target and cover searches, as well as availability of shadows in cache, provide a nice symmetric behavior of the access at the two servers, with each of them observing $num\_cover\ +1$ reads and $num\_cover\ + num\_cache\ +1$ writes for each level of the shuffle index (but level 0). In other words, each server will observe a pattern of (read/write) accesses to blocks as if it was the only server storing the data and managing the access. Note that this does not cause any performance overhead with respect to the single server solution while enjoying significant higher protection (see Section 6).

For instance, consider the shuffle index in Figure 3(a). Figure 4 illustrates, step by step, a search for value *e3* that adopts *a2* as cover and that assumes that the local cache has size one and contains the path to *b1* (e.g., $\langle r_0,r_1\rangle$, $\langle b,d\rangle$ and $\langle b1,b2\rangle$, with *d* and *b2* the shadows for *b* and *b1* chosen in a previous search). Among its far siblings, *h* is chosen as shadow for *e*, *c* for *a*, *e2* for *e3*, and *a1*

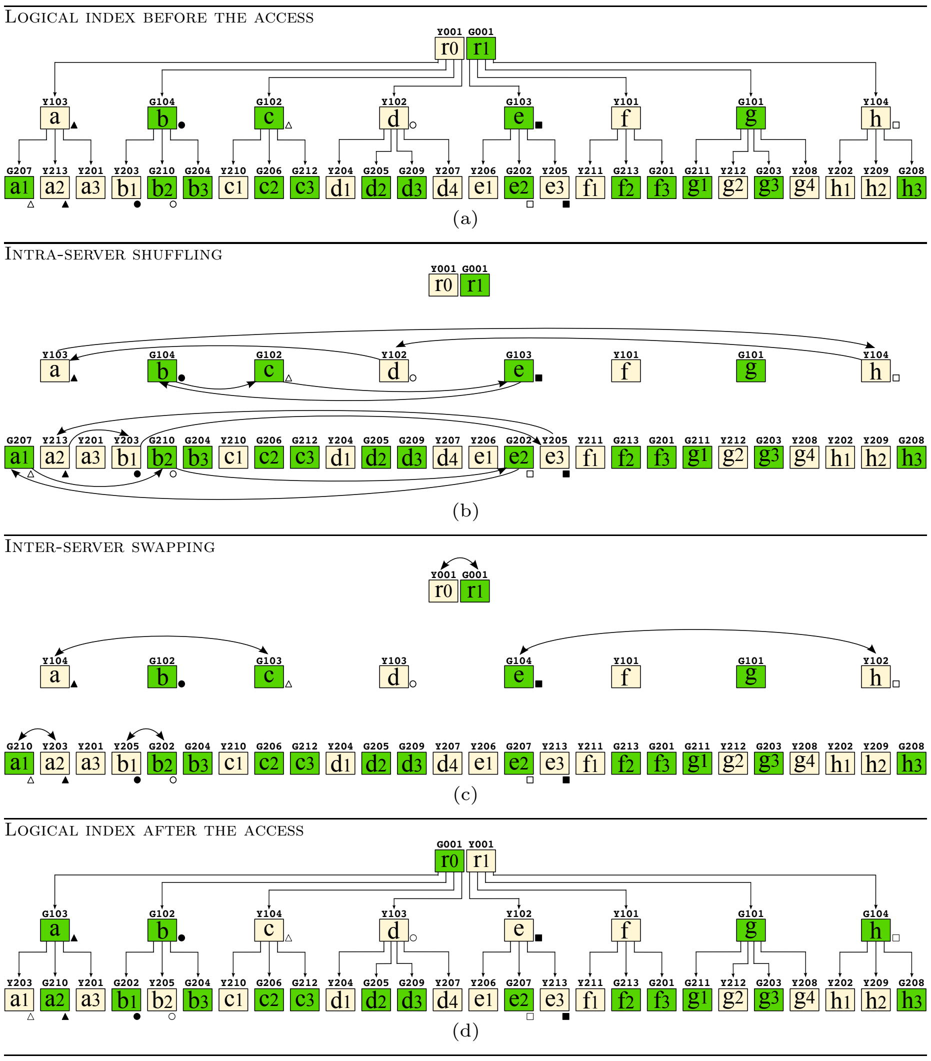

Logical index before the access
(a)
Intra-server shuffling
(b)
Inter-server swapping
(c)
Logical index after the access
(d)

Fig. 3. Evolution of the shuffle index for our running example
Legend: ■ target and □ its shadow; • node in cache and ∘ its shadow; ▲ cover and △ its shadow

for *a2*. The columns of the table represent: the visited level of the shuffle index (l); the content of the cache (*$Cache_l$* in *Retrieved nodes*) and the nodes read from the servers (*Read* in *Retrieved nodes*); the balanced shuffling (*shuffle*); the nodes in the cache (*$Cache_l$* in *Shuffled nodes*) and read (*Non_Cached* in *Shuffled nodes*) after the shuffling; the nodes written on the server that are also kept in cache (*$Cache_{l-1}$* in *Written nodes*) or that are only stored at the server (*Non_Cached_P* in *Shuffled nodes*). Note that column *$Cache_{l-1}$* in *Written nodes*

l	**Retrieved nodes**		**Shuffle**	**Shuffled nodes**		**Written nodes**	
	Cache$_l$	*Read*		*Cache*$_l$	*Non_Cached*	*Cache*$_{l-1}$	*Non_Cached_P*
0	Y001 **r0** [Y103,G104,G102,Y102]		Y001→G001	G001 **r0** [Y103,G104,G102,Y102]			
	G001 **r1** [G103,Y101,G101,Y104]		G001→Y001	Y001 **r1** [G103,Y101,G101,Y104]			
1	G104 **b** ● [Y203,G210,G204,-]		G104→G102	G102 **b** ● [Y203,G210,G204,-]		Y001 **r1** [Y102,Y101,G101,G104]	
	Y102 **d** ○ [Y204,G205,G209,Y207]		Y102→Y103	Y103 **d** ○ [Y204,G205,G209,Y207]		G001 **r0** [G103,G102,Y104,Y103]	
		G103 **e** ■ [Y206,G202,Y205,-]	G103→Y102		Y102 **e** ■ [Y206,G202,Y205,-]		
		Y104 **h** □ [Y202,Y209,G208,-]	Y104→G104		G104 **h** □ [Y202,Y209,G208,-]		
		Y103 **a** ▲ [G207,Y213,Y201,-]	Y103→G103		G103 **a** ▲ [G207,Y213,Y201,-]		
		G102 **c** △ [Y210,G206,G212,-]	G102→Y104		Y104 **c** △ [Y210,G206,G212,-]		
2	Y203 **b1** ●		Y203→G202	G202 **b1** ●		Y102 **e** ■ [Y206,G207,Y213,-]	
	G210 **b2** ○		G210→Y205	Y205 **b2** ○		G104 **h** □ [Y202,Y209,G208,-]	
		Y205 **e3** ■	Y205→Y213		Y213 **e3** ■		G102 **b** ● [G202,Y205,G204,-]
		G202 **e2** □	G202→G207		G207 **e2** □		Y103 **d** ○ [Y204,G205,G209,Y207]
		Y213 **a2** ▲	Y213→G210		G210 **a2** ▲		G103 **a** ▲ [Y203,G210,Y201,-]
		G207 **a1** △	G207→Y203		Y203 **a1** △		Y104 **c** △ [Y210,G206,G212,-]
						Y213 **e3** ■	G202 **b1** ●
						G207 **e2** □	Y205 **b2** ○
							G210 **a2** ▲
							Y203 **a1** △

Fig. 4. An example of access to the distributed shuffle index in Figure 2 searching for *e3*, with *a2* as cover
Legend: ■ target and □ its shadow; ● node in cache and ○ its shadow; ▲ cover and △ its shadow

represents the status of the local cache at the end of the access. The evolution of the shuffle index for the search in Figure 4 is illustrated in Figure 3.

Figure 5 shows the observations of the servers in terms of blocks read and written by the access in Figure 4. The different blocks read provide confusion to each of the server with respect to which is the target of the access (as a matter of fact, the observations of a server might even not include the target but its shadow); the different blocks written provide confusion over what is stored in the blocks after the access (as a matter of fact, even the set of nodes stored at each server might have changed), thus practically destroying any possibility for the servers to correlate observations over different access requests (see Section 6).

6 Discussion and Evaluation of the Approach

We discuss the protection guarantees and the performance of our distributed shuffle index, in particular comparing it with the original proposal [5] adopting a single server.

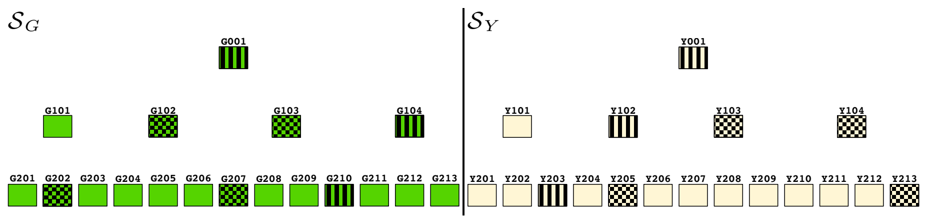
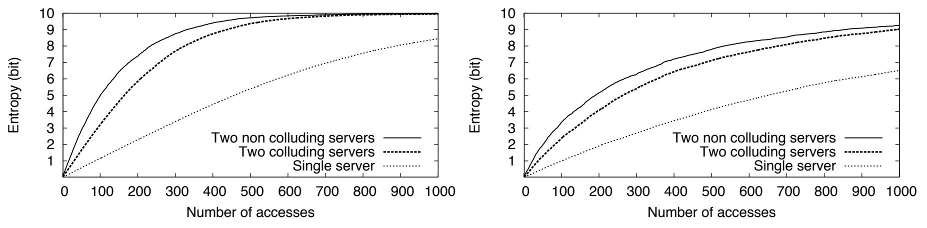


Fig. 5. Observations by each server of read/written blocks in our running example Legend: blocks read and written: chessboard filling, blocks written: lines filling

Protection. Like in [5], we focus the analysis on leaf nodes, which are more exposed than the internal ones. Indeed, internal nodes are clearly involved in shuffling operations more often than leaf nodes. Also, while in the analysis we assume the servers not to collude, we note that a possible collusion can cause a slight decrease of protection but does not cause critical breaches because the distributed shuffle index would remain protected as in the case of a shuffle index employing 2*num_cover* + *1* covers and a double cache size. We start by noting that access confidentiality naturally increases with the use of two servers. In fact, even if no cover was to be applied, the server could have just a 50% confidence that its observations refer to blocks in the path to a target as they could just refer to their shadows. Shadows provide then a natural increase to the protection when covers are applied. The fact that shuffling operates across servers also provide a natural protection since, again, every node has 50% probability of remaining on the same server after a shuffling is applied. (Note that encryption with a different salt at every re-allocation prevents servers from making any inference on the shuffling performed.)

To study the protection offered by shuffling, we model the knowledge of a server on the fact that a node n is stored at a given block id as a probability value $\mathcal{P}(n, id)$, expressing the confidence in such a knowledge, with $\mathcal{P}(n, id) = 1$ corresponding to certainty and $\mathcal{P}(n, id) = \frac{1}{|\mathcal{N}'|}$, with $\mathcal{N}'$ the set of leaf nodes in $\mathcal{N}$, corresponding to complete absence of knowledge. We assume the worst starting case where a server knows the exact correspondence between nodes and blocks (i.e., $\mathcal{P}(n, id) = 1$ when n is allocated at block id, $\mathcal{P}(n, id) = 0$ otherwise) and evaluate the knowledge degradation of the server due to the shuffling performed at every access.

Let $ID'_G \subseteq \mathcal{ID}_G$ and $ID'_Y \subseteq \mathcal{ID}_Y$ be the sets of identifiers of the m leaf blocks accessed at servers $\mathcal{S}_G$ and $\mathcal{S}_Y$, respectively. Consider also a leaf node $n \in \mathcal{N}$ and suppose that server $\mathcal{S}_G$ knows that n is stored at one of the m accessed blocks (the same discussion applies to server $\mathcal{S}_Y$). After the access, two cases can occur: *i)* n is still stored at server $\mathcal{S}_G$, or *ii)* n has been moved to server $\mathcal{S}_Y$. In the first case, for all $id_G \in ID'_G$, we have that $\mathcal{P}(n, id_G) = \sum_{id_G \in ID'_G} \frac{\mathcal{P}(n, id_G)}{2m}$ since there is a 50% chance for node n to remain at server $\mathcal{S}_G$ and there are m possible blocks where the node can be stored. In the second case, node n can



(a) (b)

Fig. 6. Evolution of the entropy for values of γ equal to 0.5 (a) and 0.25 (b). Every access request directed to a server has *num_cover*=3 and *num_cache*=1

be moved to any of the m blocks accessed at server $\mathcal{S}_Y$. However, server $\mathcal{S}_G$ does not know which are the m leaf blocks accessed at server $\mathcal{S}_Y$. Then, for all $id_Y \in \mathcal{ID}_Y$ we have that $P(n, id_Y) = \frac{1-\sum_{id_G \in \mathcal{ID}_G} \mathcal{P}(n, id_G)}{|\mathcal{ID}_Y|}$.

We performed a set of experiments for studying the degradation of the knowledge of a server at the aggregate level. These experiments evaluate the entropy of the server knowledge under different configurations (i.e., varying the number of covers and the size of the cache) with different access profiles, where access profiles have been simulated by randomly generating sequences of accesses following a self-similar[4] probability distribution with skewness γ in the range [0.25, 0.5]. We then evaluated the increase of entropy at the increase of the number of accesses for three scenarios where: *1)* the shuffle index is distributed over two servers; *2)* the shuffle index is distributed over two servers but the two servers collude; *3)* the shuffle index is stored at a single server. Note that the second scenario has a double role, representing two different cases: *2.1)* when the two servers collude (exchanging all the knowledge they have on the initial allocation as well as the knowledge on every subsequent observation); *2.2)* when a single server is applied but with the use of 2*num_cover* + *1* covers and with a cache of size 2*num_cache*. Figure 6 illustrates the experimental results using 3 covers and a cache with size 1 for every access request directed to a server, considering a logical shuffle index with 1000 leaves, skewness γ equal to 0.5 and 0.25, and varying the number of accesses. (Experiments with different configurations presented a similar behavior.) As it is visible in the figures, in the scenario where the shuffle index is distributed over two servers (scenario 1, solid line), the entropy increases much faster than in the scenario of a single server subject to a similar workload (scenario 3, dotted line). The dashed line, reporting the entropy evolution in case of collusion (scenario 2), with respect to the other two lines tells us that: *i)* collusion among servers implies a slower knowledge degradation (as the servers combine their knowledge), but does not cause confidentiality breaches (since entropy remains high); *ii)* the use of two servers, even when such servers

[4] Given a domain of cardinality d, a self-similar distribution with skewness γ provides a probability equal to $1-\gamma$ of choosing one of the first γd domain values.

collude, enjoys a faster entropy increase and hence, protection guarantees, over the case when a single server is used but with the application of $2num\_cover + 1$ covers and with a double size of the cache.

System Performance. The performance of the distributed shuffle index is based on the response time experienced by the client when submitting an access request. Among the different factors contributing to the response time, in our experimental evaluation, we observed that the latency of the network is the factor with the greatest impact in a large-bandwidth WAN scenario (which is the most interesting and natural environment for data outsourcing applications [5, 6]).

To assess the system performance, we considered a data set of 2 GiB stored in the leaves of a shuffle index with 3 levels with nodes of 8 KiB. To properly configure the network environment, we adopted a professional-grade tool suite (i.e., Traffic Control and Network Emulation, for Linux systems) and we chose a representative WAN configuration suitable for interactive traffic, with LAN-like bandwidth and round-trip time modeled as a normal distribution with mean of 100 *ms* and standard deviation of 2.5 *ms*. Then, we compared the average response time in two different scenarios: *i)* our distributed shuffle index where each request accesses m leaf blocks at each of the two servers; and *ii)* the original (non-distributed) shuffle index where each request accesses $2m$ leaf blocks. The experiments considered a variety of configurations, with different values for m. The average response time in the distributed scenario is approximately 5% lower than the one obtained in the original scenario. As an example, fixing $m = 3$, the average response time is 380 *ms* in the distributed scenario and 405 *ms* in the original one. Our experiments also show that, in both the original and distributed scenario, the costs of adopting one additional cover search (cache element, respectively) is 1.18% (0.6%, respectively) of the average response time.

7 Related Work

Previous related works proposed different indexing techniques for the evaluation of queries over encrypted data (e.g., [4, 13, 14, 18–20]). These solutions however aim at protecting data confidentiality only. Traditional approaches for protecting access and pattern confidentiality are based on PIR protocols (e.g., [2,10]), which however suffer from high computation and communication costs and do not provide content confidentiality. More efficient PIR solutions rely on the presence of different copies of the data stored at different servers (e.g., [1]), and are based on the assumption that servers do not communicate with each-other.

The first approach that protects data, access, and pattern confidentiality has been illustrated in [22] and combines the pyramid-shaped hierarchy layout of the Oblivious RAM (ORAM) data structure [11] with Bloom filters. Even if this proposal adopts an enhanced reordering technique between adjacent levels of the ORAM to provide a limited amortized cost of accesses, the response time of queries submitted during the reordering of the bottom level of the structure remains linear in the database size. Different approaches try to mitigate the cost of these accesses, for instance by limiting shuffling to fetched records (e.g., [9]); guaranteeing a constant number of interactions between the data owner and the server, independently from the number of levels in the ORAM (e.g., [21]); introducing the support for concurrent accesses by multiple clients (e.g., [12]). ORAM has been recently extended to the distributed scenario [16], but its privacy guarantees rely on the presence of non-communicating servers.

The line of works most related to our is represented by solutions that provide data, access, and pattern confidentiality by exploiting dynamic data allocation, which destroys the otherwise static relationship between disk blocks and the information they store (e.g., [5, 6, 15, 23]). The first approach adopting dynamic data allocation has been introduced in [15] and is based on a B-tree index structure. This proposal however does not guarantee pattern confidentiality. Similarly to the shuffle index [5], the proposal in [23] adopts cover searches, repeated searches, and shuffling protection techniques to provide access and pattern confidentiality. This solution is less flexible than the shuffle index, as it does not have an underlying index structure and the number of cover searches is fixed. Our solution provides higher protection guarantees than the proposals above, since we operate in a distributed scenario. Also, with respect to distributed PIR and distributed ORAM approaches, we remove the limiting assumption that the storing servers cannot communicate.

8 Conclusions

We extended the shuffle index to the consideration of multiple servers. Our approach is based on distributing the index structure over two servers, and on the use of shadows for providing to each server a view as if it was the only server storing the data. The distributed index enjoys an increased protection with respect to the use of a single server while not impacting performance.

Acknowledgements. This work was supported in part by the EC within the 7FP under grant agreement 257129 (PoSecCo), by the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH), and by Google, under the Google Research Award program.

References

1. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Proc. of EUROCRYPT 1999. Prague, Czech Republic (May 1999)
2. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM 45(6), 965–981 (1998)
3. Ciriani, V., De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Combining fragmentation and encryption to protect privacy in data storage. ACM TISSEC 13(3), 22:1–22:33 (2010)

4. Damiani, E., De Capitani Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. In: Proc. of CCS 2003. Washington, DC (October 2003)
5. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Efficient and private access to outsourced data. In: Proc. of ICDCS 2011. Minneapolis, MN (June 2011)
6. De Capitani di Vimercati, S., Foresti, S., Paraboschi, S., Pelosi, G., Samarati, P.: Supporting concurrency in private data outsourcing. In: Proc. of ESORICS 2011. Leuven, Belgium (September 2011)
7. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Managing and accessing data in the cloud: Privacy risks and approaches. In: Proc. of CRiSIS 2012. Cork, Ireland (October 2012)
8. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Selective and fine-grained access to data in the cloud. In: Jajodia, S., Kant, K., Samarati, P., Swarup, V., Wang, C. (eds.) Secure Cloud Computing. Springer (2013)
9. Ding, X., Yang, Y., Deng, R.: Database access pattern protection without full-shuffles. IEEE TIFS 6(1), 189–201 (March 2011)
10. Gasarch, W.: A survey on private information retrieval. Bulletin of the EATCS 82, 72–107 (2004)
11. Goldreich, O., Ostrovsky, R.: Software protection and simulation on Oblivious RAMs. Journal of the ACM 43(3), 431–473 (1996)
12. Goodrich, M., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless Oblivious RAM simulation. In: Proc. of SODA 2012. Kyoto, Japan (January 2012)
13. Hacigümüs, H., Iyer, B., Mehrotra, S., Li, C.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. of SIGMOD 2002. Madison, WI (June 2002)
14. Hore, B., Mehrotra, S., Canim, M., Kantarcioglu, M.: Secure multidimensional range queries over outsourced data. The VLDB Journal 21, 333–358 (2012)
15. Lin, P., Candan, K.S.: Hiding traversal of tree structured data from untrusted data stores. In: Proc. of WOSIS 2004. Porto, Portugal (April 2004)
16. Lu, S., Ostrovsky, R.: Distributed Oblivious RAM for secure two-party computation. In: Proc. of TCC 2013. Tokyo, Japan (March 2013)
17. Murugesan, M., Jiang, W., Clifton, C., Si, L., Vaidya, J.: Efficient privacy-preserving similar document detection. VLDBJ 19(4), 457–475 (2010)
18. Samarati, P., De Capitani di Vimercati, S.: Data protection in outsourcing scenarios: Issues and directions. In: Proc. of ASIACCS 2010. Beijing, China (April 2010)
19. Wang, C., Cao, N., Ren, K., Lou, W.: Enabling secure and efficient ranked keyword search over outsourced cloud data. IEEE TPDS 23(8), 1467–1479 (August 2012)
20. Wang, H., Lakshmanan, L.V.: Efficient secure query evaluation over encrypted XML databases. In: Proc. of VLDB 2006. Seoul, Korea (September 2006)
21. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: Proc. of CCS 2012. Raleigh, NC (October 2012)
22. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In: Proc. of CCS 2008. Alexandria, VA (October 2008)
23. Yang, K., Zhang, J., Zhang, W., Qiao, D.: A light-weight solution to preservation of access pattern privacy in un-trusted clouds. In: Proc. of ESORICS 2011. Leuven, Belgium (September 2011)