Journal of Computer Security 21 (2013) 425–461
DOI 10.3233/JCS-130468
IOS Press

Supporting concurrency and multiple indexes in private access to outsourced data [1]

Sabrina De Capitani di Vimercati [a], Sara Foresti [a], Stefano Paraboschi [b], Gerardo Pelosi [c] and Pierangela Samarati [a,*]

[a] *DI, Università degli Studi di Milano, Crema, Italy*
E-mails: {sabrina.decapitani, sara.foresti, pierangela.samarati}@unimi.it
[b] *Università degli Studi di Bergamo, Dalmine, Italy*
E-mail: parabosc@unibg.it
[c] *DEIB, Politecnico di Milano, Milano, Italy*
E-mail: pelosi@elet.polimi.it

Data outsourcing has recently emerged as a successful solution allowing individuals and organizations to delegate data and service management to external third parties. A major challenge in the data outsourcing scenario is how to guarantee proper privacy protection against the external server. Recent promising approaches rely on the organization of data in indexing structures that use encryption and the dynamic allocation of encrypted data to physical blocks for destroying the otherwise static relationship between data and the blocks in which they are stored. However, dynamic data allocation implies the need to rewrite blocks at every read access, thus requesting exclusive locks that can affect concurrency. Also, these solutions only support search conditions on the values of the attribute used for building the indexing structure.

In this paper, we present an approach that overcomes such limitations by extending the recently proposed shuffle index structure with support for concurrency and multiple indexes. Support for concurrency relies on the use of several differential versions of the data index that are periodically reconciled and applied to the main data structure. Support for multiple indexes relies on the definition of secondary shuffle indexes that are then combined with the primary index in a single data structure whose content and allocation is unintelligible to the server. We show how using such differential versions and combined index structure guarantees privacy, provides support for concurrent accesses and multiple search conditions, and considerably increases the performance of the system and the applicability of the proposed solution.

Keywords: Data outsourcing, access privacy, concurrency, shuffle index

[1] A preliminary version of this paper appeared under the title “Supporting concurrency in private data outsourcing”, in *Proc. of the 16th European Symposium on Research in Computer Security* (ESORICS 2011), Leuven, Belgium, September 2011 [10].

[*] Corresponding author: Pierangela Samarati, DI, Università degli Studi di Milano, Via Bramante 65, 26013 Crema, Italy. Tel.: +39 0373 898061; Fax: +39 0373 898074; E-mail: pierangela.samarati@unimi.it.

1. Introduction

The evolution of information and communication technology is leading to information system architectures that rely more and more on third parties for the management of IT functions. A major motivation for such a trend is economical: with outsourcing an organization can simplify its structure and benefit from the large scale economies of rented IT services, with low costs and high availability. However, a significant obstacle to a greater adoption of outsourcing is today represented by possible concerns over improper exposure of confidential or sensitive information. As a matter of fact, while the external service provider can be relied upon for guaranteeing protection of managed data, it is of utmost importance to protect possible sensitive information from the eyes of the service provider itself.

The research and development communities have devoted significant attention to the problem of protecting data confidentiality in outsourcing scenarios, producing several solutions addressing different aspects of the problem. All proposals apply encryption to make data not intelligible to the server, providing support for query execution either by associating additional indexes with the encrypted data (e.g., [1,4, 8,16,17,24,26]) or extending the tree-based indexing structures typically adopted in DBMSs (e.g., [8]). Tree-based approaches, unlike additional indexes, are not vulnerable to privacy breaches exploiting the possible correlation between frequencies of the index values and of the actual data behind them [4]. However, even tree-based indexes remain vulnerable to attacks based on the observation of sequences of accesses and on the analysis of the frequency distribution of access requests. Such vulnerability can be counteracted by adopting approaches that change the location of the encrypted data at every access, so to break the otherwise static relationship between data and their physical location [9,19,28]. Dynamically allocated data structures represent the best defense against frequency attacks by the server. Among them, the shuffle index [9] extends the classical $B+$-tree structure used in databases with encryption, cover searches (to "hide" the actual target search in a set of additional fake searches, thus providing uncertainty over the block actually targeted), and shuffling to enforce dynamic allocation.

Although the shuffle index enjoys limited overhead with respect to the protection guarantees it offers [9], like other dynamically allocated data structures, it could potentially affect performance in scenarios where accesses need to operate concurrently and need to be performed according to different search attributes. In fact, reallocating data at the server requires write (hence exclusive) locks on the block involved in an access even in the execution of read-only operations. Also, searches based on attributes different from the index with which the shuffling structure has been organized require downloading the whole database for evaluating the search condition locally.

In this paper, we extend the shuffle index in [9] to efficiently support concurrent accesses and multiple indexes while guaranteeing privacy in data accesses. Our solution to support concurrent accesses to the indexed data consists in having transactions operating on dynamically created portions of the primary shuffle index, which we call *delta versions*. Delta versions are maintained in the server main memory, are managed – and shuffled at each access – independently one from the other, and are periodically reconciled and applied to the primary data structure on disk.

Our solution to support multiple indexes consists in the definition of secondary index structures which are then merged – at the logical and physical level – together with the primary index so to form a single structure whose organization and allocation is unintelligible to the server. This combined shuffle index guarantees protection against the long-term accumulation of information by the server [9].

The approach presented in this paper supports concurrency and multi-index searches while offering a guarantee that the server monitoring the sequence of accesses will not be able to use the information about the frequency or the order of accesses to infer the content of the database and the target values of accesses performed by users. The experimental results show that our solution produces up to a fourfold increase in system throughput in case of concurrent accesses. In [10] we presented an early version of our proposal that introduced delta versions for supporting concurrency. Here we extend the proposal with support for multiple indexes. Also, we provide the algorithm for reconciling delta versions and formally analyze its correctness and security. The experimental results are extended by measuring the average service time of a system using the proposed indexing technique, and the average response time as seen by the client.

The remainder of this paper is organized as follows. Section 2 introduces the shuffle index and describes how it protects access confidentiality by means of cover searches and dynamic data allocation. Section 3 formally defines a shuffle index at the abstract, logical, and physical levels, and introduces the use of delta versions for supporting concurrent accesses. Section 4 describes the execution of concurrent accesses to the shuffle index using a set of delta versions. Section 5 discusses an approach for reconciling delta versions with the main index. Section 6 presents the reconciliation algorithm and formally shows its correctness. Section 7 presents secondary shuffle indexes as a solution for supporting search conditions on different attributes and describes how they can be combined with the primary shuffle index at the logical and physical levels. Section 8 analyzes the security guarantees offered by the proposed indexing technique. Section 9 presents the experimental results assessing the throughput, average service time, and average response time provided by our shuffle index. Section 10 discusses related work. Finally, Section 11 reports our conclusions. The proofs of the theorems are reported in Appendix.

2. Preliminary concepts

Before introducing our approach, we illustrate the shuffle index with which outsourced data are organized [9]. We assume that the outsourced data collection is stored in a relation r defined over schema $R(A_1,\dots,A_n)$. The tuples in r are indexed over a candidate key $K \subseteq \{A_1,\dots,A_n\}$. For simplicity, but without loss of generality, in this paper we assume that indexes are defined over candidate keys composed of one attribute only, even if they can be defined on arbitrary subsets of attributes that represent a candidate key for the relation. The outsourced data are organized as an *unchained* B*+-tree*, with actual data stored in the leaves of the tree in association with their index values (i.e., the shuffle index is a primary index for the outsourced data collection). The fact that the tree is unchained means that there are no links connecting the leaves. The fan-out F of the tree regulates the number of index values stored in the nodes. Each node stores a list $V[1,\dots,q]$ of q values, with $\lceil \frac{F}{2} \rceil - 1 \leqslant q \leqslant F - 1$ (the lower-bound does not apply to the root) ordered from the smallest to the greatest, and has $q + 1$ children. The first child of a node is the root of the subtree with all values $v < V[1]$; the ith child is the root of the subtree containing the values v with $V[i-1] \leqslant v < V[i]$; and the last child is the root of the subtree with all values $v \geqslant V[q]$. Figure 1(a) illustrates a graphical representation of

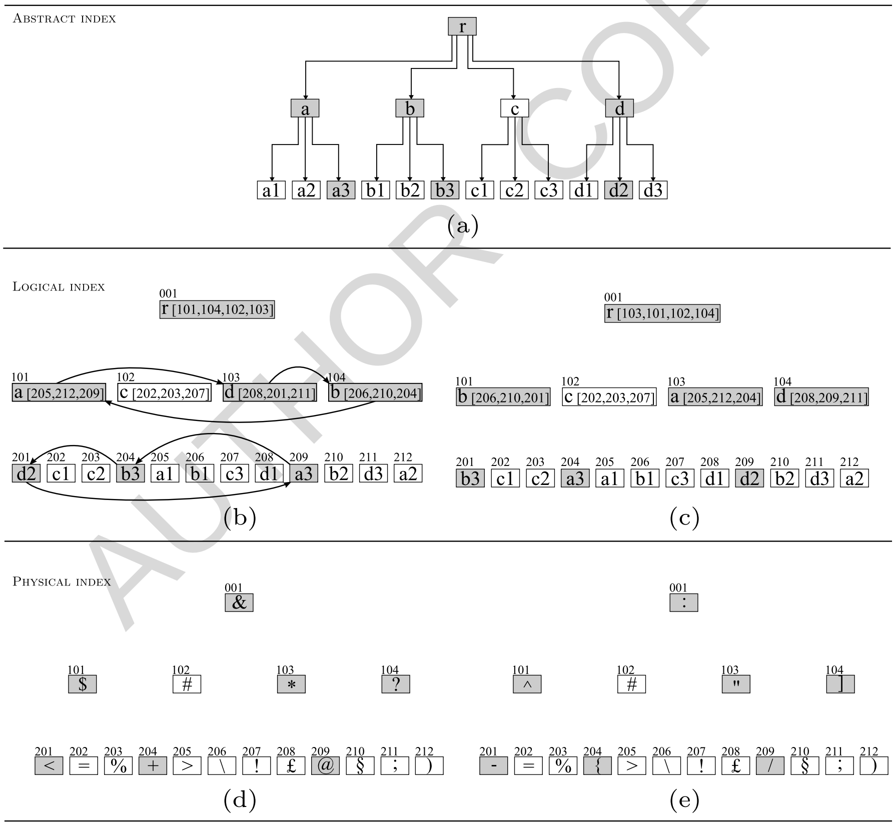


Fig. 1. An example of abstract (a), logical (b)–(c), and physical (d)–(e) index before (b)–(d) and after (c)–(e) the execution of a search operation.

an example of our abstract data structure. For simplicity, in our examples we refer to every node with a label (not explicitly reporting values in the node).

At the logical level, nodes are allocated to logical addresses that work as logical *identifiers*. 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 storing server. In the following, we assume that the physical address corresponds to the logical identifier of the node stored in it. Note that the possible order among identifiers does not necessarily correspond to the order in which nodes appear in the value-ordered abstract representation. Figure 1(b) illustrates a possible representation at the logical level of the abstract data structure in Fig. 1(a). In the figure, nodes appear ordered (left to right) according to their identifiers, which are reported on the top of each node. Pointers to children are represented by reporting in each node the ordered list of the identifiers of its children. For simplicity and easy reference, in our example, the first digit of the node identifier denotes the level of the node in the tree. Before sending to the server the index for storing it, the content of each node is prefixed with a random salt and then encrypted in CBC mode with a symmetric encryption function producing an encrypted block. All nodes in the index are encrypted with the same encryption key, which is shared between the data owner and users authorized to access the outsourced data collection. Figure 1(d) illustrates the physical representation of the logical data structure in Fig. 1(b). Since each block is encrypted, the server does not have any information on the content of the node stored in the block or on the parent-child relationship between nodes stored in blocks. Retrieval of the leaf block containing the tuple corresponding to an index value requires an iterative process. Starting from the root of the tree and ending at a leaf, the read block is decrypted retrieving the address of the child block to be read at the next step. Each access to a leaf block then requires $h + 1$ rounds of communication between the client and the server, where h is the height of the tree. To avoid leaking to the server the fact that (i) some blocks represent a path in the tree and (ii) different accesses aim at the same content, the shuffle index extends the search operation 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 blocks in a local *cache*;
- mixing (*shuffling*) the content of all retrieved blocks as well as those maintained in cache and rewriting them accordingly on the server.

Intuitively, cover searches introduce uncertainty over the leaf block actually belonging to the target search and do not allow the server to establish the parent-child relationship between blocks (since every access entails reading multiple blocks at every level). The cache is used to make searches repeated within a short time interval not recognizable as being the same search (if the nodes in the target path are already in cache, an additional cover search will be executed instead). Shuffling moves content among blocks, thus breaking the correspondence between nodes (contents) and blocks (addresses). Note that shuffling requires to re-encrypt the involved nodes with a different random salt, so to produce a different encrypted text, and changing the pointers to them in their parents (which will have to point to the new blocks to which nodes have been allocated). Changing the allocation of nodes to blocks provides confidentiality: (i) subsequent searches looking for the same content would aim at different blocks, and (ii) subsequent searches hitting the same block would involve a different content.

As an example, consider a search for value $b3$ over the abstract index in Fig. 1(a), and assume that the search adopts $a3$ as cover and that the local cache contains the path to $d2$ (i.e., $(001, 103, 201)$). Figure 1(b) illustrates the logical representation of the abstract index before the execution of the search operation and how accessed blocks are shuffled, level by level, to obtain the structure in Fig. 1(c). The nodes involved in the search operation are denoted in gray in the figure. Note that although the server knows which blocks have been accessed (gray blocks in Fig. 1(d)–(e)) it cannot detect which of those blocks is the actual target of the search and how the content of blocks has been shuffled, since blocks are re-encrypted using a different salt at each write operation.

3. Main index and delta versions

Before introducing our solution for supporting concurrent accesses and the evaluation of conditions over different attributes of the outsourced data collection, we need to formalize the components of the shuffle index data structure and of the shuffling (which were only procedurally managed in the original proposal [9]). Data can be seen at the abstract, logical, and physical levels, which we formally capture as follows.

- *Abstract* ($\mathcal{T}^a$): set $\{n_1^a, \dots, n_m^a\}$ of abstract nodes forming an unchained $B+$-tree. Each internal node in $\mathcal{T}^a$ is a pair $n^a = \langle \mathit{Values}, \mathit{Children} \rangle$ with *Values* a list of q index values and *Children* a list of $q + 1$ child nodes. Leaf nodes are represented with nodes of the form $\langle \mathit{Values}, \mathit{Tuples} \rangle$ that include, instead of the list of child nodes, the set *Tuples* of tuples in r with index values in *Values*.
- *Logical* ($\mathcal{T}$): triple $(\mathcal{T}^a, \mathcal{ID}, \phi)$, where $\mathcal{T}^a$ is an abstract data structure, $\mathcal{ID}$ is a set of logical identifiers, and $\phi : \mathcal{T}^a \rightarrow \mathcal{ID}$ is a bijective function associating each abstract node n^a in $\mathcal{T}^a$ with a logical identifier *id* in $\mathcal{ID}$. Triple $(\mathcal{T}^a, \mathcal{ID}, \phi)$ determines how the abstract nodes in $\mathcal{T}^a$ are allocated to logical identifiers in $\mathcal{ID}$. Each internal node $n^a = \langle \mathit{Values}, \mathit{Children} \rangle \in \mathcal{T}^a$ is then represented by a (logical) node of the form $\langle \mathit{id}, V, P \rangle$, where $\mathit{id} = \phi(n^a)$, $V = \mathit{Values}$, and $P[j] = \phi(\mathit{Children}[j])$, $j = 1, \dots, q + 1$. Leaf nodes are represented with logical nodes of the form $\langle \mathit{id}, V, T \rangle$ that include tuples T instead of pointers to children.
- *Physical* ($\mathcal{T}^e$): set of (disk) blocks storing $\mathcal{T}$. Each logical node $\langle \mathit{id}, V, P \rangle \in \mathcal{T}$ ($\langle \mathit{id}, V, T \rangle \in \mathcal{T}$ for leaves) is stored in a block that can be seen as a pair of the

form $\langle id,b\rangle$, where $b = E_k(salt \| id \| V \| P)$ ($b = E_k(salt \| id \| V \| T)$ for leaves) with E a symmetric encryption function, k the encryption key (which is the same for all nodes in $\mathcal{T}$ and is known to the data owner and authorized users), and *salt* a value chosen at random for each encryption.

In the following, 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 the term node or the term block can be used, we will use them interchangeably.

Shuffling executed at every access randomly exchanges the content among blocks. A shuffling of logical index $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ is equivalent to reallocating nodes to potentially different blocks (the corresponding abstract index structure remains unaltered), as formally defined in the following.

Definition 3.1 (Shuffling). Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index and $\pi : \mathcal{ID} \to \mathcal{ID}$ be a random permutation of $\mathcal{ID}$. The *shuffling* of $\mathcal{T}$ with respect to π is a logical index $\mathcal{T}' = (\mathcal{T}^a, \mathcal{ID}, \phi')$, where ϕ' is a composite function defined as $\phi' = \pi \circ \phi$.

Note that a change in the allocation of nodes to blocks implies that the pointers to children must be updated to reflect their new allocation, thus preserving the correct parent-child relationship. After shuffling, each abstract node $n^a = \langle \mathit{Values}, \mathit{Children} \rangle$ in $\mathcal{T}^a$ is represented by logical node $\langle id, V, P\rangle$ in $\mathcal{T}'$, where $id = \phi'(n^a) = \pi(\phi(n^a))$, $V = \mathit{Values}$, and $P[j] = \phi'(\mathit{Children}[j]) = \pi(\phi(\mathit{Children}[j]))$, $j = 1, \ldots, q+1$. In the following, for simplicity and without loss of generality, we assume shuffling to operate within the boundary of the tree level (i.e., permutations are always performed among nodes of the same level of the tree).

A delta version is essentially a – potentially shuffled – portion of the main index, as captured by the following definition.

Definition 3.2 (Delta version). Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index. A *delta version* $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ of $\mathcal{T}$ is a shuffling of $(\Delta_i^a, \mathcal{ID}_i, \phi)$ with respect to a permutation π, where $\Delta_i^a \subseteq \mathcal{T}^a$ such that $\forall n^a \in \Delta_i^a$, the parent of n^a belongs to Δ_i^a; $\mathcal{ID}_i = \bigcup_{n^a \in \Delta_i^a} \phi(n^a)$; and $\phi_i : \mathcal{T}^a \to \mathcal{ID}$ such that $\phi_i(n^a) = \phi(n^a)$ if $n^a \notin \Delta_i^a$.

At the physical level, delta versions are stored in blocks obtained by encrypting the nodes in Δ_i^a with the same encryption key k used for the main shuffle index. These blocks are maintained by the server in main memory. Figure 2(b) illustrates an example of delta version of the logical index in Fig. 2(a). Note that, since a delta version is composed of nodes along paths that are traversed when executing search operations, the parent of each node in the delta version also belongs to the delta version. Therefore, every delta version always includes the root of $\mathcal{T}^a$. In the following, we use $\mathcal{T}$ and Δ_i to denote the set of logical nodes forming a shuffle index and a delta version, respectively.

Merging a delta version with a main index implies enforcing on the main index the allocation of nodes to blocks prescribed by the delta version, as captured by the following definition.

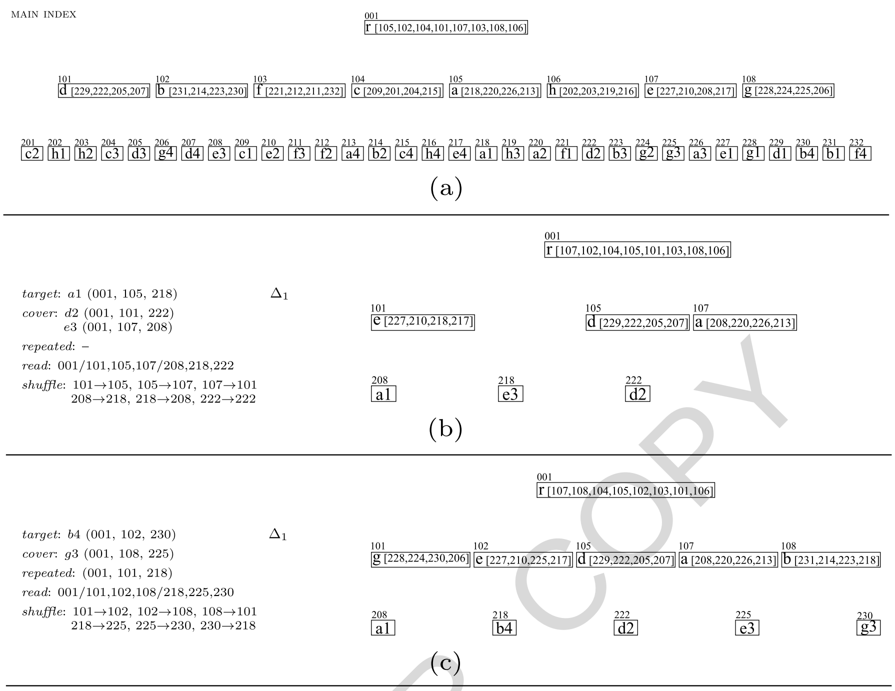


Fig. 2. An example of main index (a) and of execution of two subsequent searches (b)–(c) over it using delta version Δ_1.

Definition 3.3 (Merge). Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index and $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ be a delta version of $\mathcal{T}$. The *merge* of $\mathcal{T}$ and Δ_i, denoted $\mathcal{T} \oplus \Delta_i$, is logical index $\mathcal{T}' = (\mathcal{T}^a, \mathcal{ID}, \phi_i)$.

In terms of actual enforcement, $\mathcal{T}'$ can be simply obtained by flushing the blocks of the delta version to the main index (i.e., by overwriting the blocks on disk with the blocks in main memory associated with the same physical address), while leaving the other blocks unaltered. Such an operation – which can be performed without any need to download the involved blocks or performing computation by the client – produces an index that correctly represents the original data structure and includes the shuffling operated on the delta version (see Theorem 5.1 in Section 5).

4. Operating on delta versions

The basic idea of our approach is that transactions operate on delta versions (dynamically created and maintained in main memory at the server) rather than on the main shuffle index. Concurrent transactions execute Algorithm 1, operating at

Algorithm 1. Access to a shuffle index through delta versions

```
$\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$: logical index on a candidate key $K$ with domain $D_K$, height $h$, fan-out $F$
$\Delta_1,\ldots,\Delta_n$: delta versions of $\mathcal{T}$
$Allocation_1[0,\ldots,h],\ldots,Allocation_n[0,\ldots,h]$: layered structure that keeps track of node-block associations
                  and of parent-child relationships by storing, for each node in $\Delta_i$, its logical identifier id,
                  its label label, and the label of its parent parent (see Section 6)
num_cover: number of cover searches
target_value: value to be searched in the shuffle index

MAIN
/* Delta version assignment */
v_id := GetAvailableDeltaVersion /* require the allocation of a delta version to the server */
let $t \in$ VERSION s.t. $t[\Delta id]$=v_id
if $\exists t' \in$ VERSION s.t. $t'$[status]='free' $\wedge$ $t'[ts] > t[ts]$ then exit /* server's misbehavior: $\Delta_{v\_id}$ is not the MRU delta version */
t[status] := 'busy'
Root := Decrypt(ReadBlocks(root_id, $\Delta_{v\_id}$)) /* retrieve the root node of the chosen delta version */
if Root.ts$<$t[ts] then exit /* server's misbehavior: the access at time t[ts] has been discarded */
t[ts] := current_time
Root.ts := current_time
/* Access execution */
target_id := repeated_id := Root.id /* identifier of the node along the path to the target and of the repeated search */
repeated := TRUE /* the root is always a repeated search */
num_cover := num_cover + 1
Parents := {Root}
for i:=1...num_cover do cover_id[i] := target_id
for i:=1...num_cover do /* choose cover searches */
  randomly choose cover_value[j] in $D_K$ s.t. /* $D_K$ is the domain of attribute $K$ */
    $\forall$j=1,...,i$-$1, ChildToFollow(Root,cover_value[i])$\neq$ChildToFollow(Root,cover_value[j]),
    ChildToFollow(Root,cover_value[i])$\notin\Delta_{v\_id}$.Repeated[1], and
    ChildToFollow(Root,cover_value[i])$\neq$ChildToFollow(Root,target_value)
/* search, shuffle, and update repeated searches and the delta version */
for l:= 1...h do
  let $n \in$ Parents such that $n$.id=target_id /* node along the path to the target at level $l-1$ */
  target_id := ChildToFollow($n$,target_value) /* identifier of the node along the path to the target at level l */
  /* identify the blocks to read from the server */
  ToRead_ids := {target_id}
  if target_id$\notin\Delta_{v\_id}$.Repeated[l] then /* the target node is not along the path to a repeated search */
    if repeated then
      num_cover := num_cover $-$ 1
      repeated := FALSE
    else
      let $n \in$ Parents s.t. $n$.id = repeated_id /* repeated_id is the identifier of the node along the path to the repeated search */
    repeated_id := $n$.P $\cap$ $\Delta_{v\_id}$.Repeated[l] /* identifier of the node along the path to the repeated search */
    ToRead_ids := ToRead_ids $\cup$ {repeated_id}
  for i:=1...num_cover do
    let $n \in$ Parents s.t. $n$.id=cover_id[i]
    cover_id[i] := ChildToFollow($n$,cover_value[i])
    ToRead_ids := ToRead_ids $\cup$ {cover_id[i]}
  /* read blocks */
  Read := Decrypt(ReadBlocks({id$\in$ToRead_ids: id$\in$Allocation$_{v\_id}$[l]}, $\Delta_{v\_id}$)) /* blocks read from the delta version */
  Read := Read $\cup$ Decrypt(ReadBlocks({id$\in$ToRead_ids: id$\notin$Allocation$_{v\_id}$[l]}, $\mathcal{T}$)) /* blocks read from the main index */
  for each $n \in$ Read s.t. $n$.id$\notin$Allocation$_{v\_id}$[l] do /* insert a tuple in Allocation$_{v\_id}$[l] for each new node in the delta version */
    $t_a$.id := $n$.id
    $t_a$.label := label($n$.V)
    let $n_{par} \in$ Parents s.t. t.id$\in n_{par}$.P
    $t_a$.parent := label($n_{par}$.V)
    Insert($t_a$,Allocation$_{v\_id}$[l])
  /* shuffle nodes */
  let $\pi$ be a permutation of ToRead_ids
  for each $n \in$ Read do
    let $t_a \in$ Allocation$_{v\_id}$[l] s.t. $t_a$.id=$n$.id /* update Allocation$_{v\_id}$[l] with the new node-block association */
    $n$.id := $\pi$($n$.id)
    $t_a$.id := $n$.id
  for each $n \in$ Parents do /* determine effects on parents and store nodes at level $l-1$ */
    for i:=0...F do $n$.P[i] := $\pi$($n$.P[i])
      WriteBlock($n$.id, Encrypt($n$), $\Delta_{v\_id}$) /* write blocks in the delta version */
  target_id := $\pi$(target_id) /* update the identifier of the node along the path to the target */
  repeated_id := $\pi$(repeated_id) /* update the identifier of the node along the path to the repeated search */
  for i:=1...num_cover do cover_id[i] := $\pi$(cover_id[i]) /* update the identifier of the nodes along the paths to covers */
  $\Delta_{v\_id}$.Repeated[l] := ToRead_ids /* update repeated searches at level l */
  Parents := Read
for each $n \in$ Read do WriteBlock($n$.id, Encrypt($n$), $\Delta_{v\_id}$) /* write nodes at level h */
/* return the target leaf node */
let $n \in$ Read such that $n$.id=target_id
t[status] := 'free'
return($n$)
```

a trusted party, to access the outsourced relation r to search for a *target_value* of the attribute on which the shuffle index has been defined (i.e., attribute K). The algorithm is composed of two steps: (i) the choice of the delta version on which the search should operate and (ii) the execution of the access to the shuffle index through the chosen delta version. We now briefly describe the two steps.

4.1. Delta version assignment

Every access operation is executed over a delta version. To avoid imposing synchronization constraints, we assume the allocation of delta versions to each transaction to be determined by the server (line 1). However, we need to provide a means to control the proper behavior of the server in the allocation of the versions. It is important to ensure that the server: (i) does not discard the shuffling requests, (ii) does not create a new delta version at each access to have transactions always operating on the main index (and therefore on a static data structure), and (iii) does not selectively allocate versions to monitor specific activities. Therefore, we assume that a trusted client acts as a coordinator for concurrent accesses by different transactions. The presence of the client allows detecting possible misbehaviors of the server in the management of delta versions and in the execution of shuffling operations. The client initially sets the number of delta versions (which corresponds to the maximum amount of concurrent accesses to be supported). The client maintains a table VERSION($\underline{\Delta id}$, *ts*, *status*), reporting for each delta version Δ*id* the time *ts* of last access and the *status* (busy or free) of the version. We require the transaction at the client side to update the entry for the version allocated to it, by setting *ts* to the current time and *status* to busy (line 4) before accessing the data. We instead account for a lazy process for the transactions in setting that the version allocated to each of them has been released (*status* free, line 56). Hence, while a version appearing as free in table VERSION is certainly free, a version appearing as busy could actually have been released (but the transaction be late in reporting the status change). We require the server to manage the allocation of delta versions according to a Most Recently Used (MRU) policy, that is, an access should always be enforced on the version most recently used. The client can then check that the server has performed proper allocation by verifying whether the time of last access to the delta version allocated to a transaction is greater than the greatest *ts* associated with a free version in the table (the "greater than" condition is to accommodate for possible delays of transactions operating at the client side to set version status free, lines 2–3). We note that, to perform such a control, the client requires an exclusive lock on table VERSION when it sends to the server a request for the allocation of a delta version to a transaction (line 1). This exclusive lock is necessary to prevent changes to the status of the delta versions in the time window between the assignment by the server of a delta version to the transaction and the verification by the client of the correctness of the assignment. The exclusive lock is released when the transaction has set to busy the status of the delta version assigned to it (line 4). We assume the root of every delta version to be timestamped at each access. This allows the client to check that the root is actually the result of the access executed at the time *ts* recorded in the table for the delta version (lines 5–6) and, therefore (since the root points to the other blocks in the tree) the freshness of the whole version.

4.2. Access execution

Access execution works essentially like in the original shuffle index proposal requesting $h + 1$ communication steps between a transaction and the server such that at each level of the shuffle index *num_cover* $+ 2$ blocks (lines 9–56) are read from the server. If the operation needs to read a block that does not belong to the delta version, such a block is taken from the main index and included in the delta version. Apart from the need to include new blocks in the delta version, the only notable difference with respect to the original shuffle index proposal is that we depart from the local cache maintained for hiding that subsequent searches were aiming at the same node. The reason for departing from the cache is that its maintenance would impose a strong synchronization overhead among the different transactions operating at the client side. To prevent the server from recognizing that two subsequent accesses aim at the same block, we take a dual approach with respect to using the cache, and adopt *repeated searches* instead. Intuitively, while the cache ensures consequent searches never access the same block (if a value just retrieved is needed, a fake value is searched instead, so to ensure no intersection between the two searches and that the same number of blocks is accessed at each level), repeated searches always ensure intersection between subsequent searches (regardless of whether the two searches are looking for the same value or not). For enforcing repeated searches, we store, in conjunction with each delta version, a layered structure that keeps track of the identifiers of the blocks accessed during the last search (*Repeated*$[0, \ldots, l]$). Execution of an access on a delta version will also request at least one block per level among those appearing in the last search. Each search then accesses *num_cover* $+ 2$ blocks at every level of the index, since, besides the blocks of the target and cover searches, an additional block is necessary for the repeated search. At the beginning, when the delta version is empty, there is no search to repeat and an additional cover is requested instead.

As an example, consider the index in Fig. 2(a) and assume that the data owner decides to adopt six delta versions that are, at initialization time, all empty. Let us now suppose that the data owner searches for value $a1$ and decides to adopt one cover and operates on delta version Δ_1. In this case, two covers (e.g., $d2$ and $e3$) are needed. The blocks on the paths to $a1$, $d2$, and $e3$ are all read from the main index, shuffled, and written back in Δ_1 as illustrated in Fig. 2(b). Suppose now that, after a number of concurrent accesses to the shuffle index, the status of the six available delta versions is the one illustrated in Table 1 and that the data owner needs to execute another search for value $b4$. The server assigns to the transaction delta version Δ_1, as it is the most recently used available delta version. Let us also assume that the

Table 1

Status of the delta versions available for the shuffle index

Δ*id*	*ts*	*status*
1	15	free
2	14	busy
3	9	free
4	12	busy
5	18	busy
6	10	free

transaction adopts one cover search (e.g., $g3$), and one repeated access (e.g., 001, 101, 218). Since the nodes along the paths to $b4$ and $g3$ (except the root) do not belong to Δ_1 they are read from the main index, and after shuffling their content with all accessed blocks, they are copied in the delta version. Figure 2(c) illustrates Δ_1 after the execution of the second search operation.

5. Reconciling delta versions and main index

A delta version grows at every access by including new requested blocks that were not previously contained in the delta version. In the long run, a delta version could potentially include all the blocks of the main index saturating the server main memory. Hence, we periodically synchronize the main index with the delta versions, reporting shuffling operations on the main index and resetting the delta versions. Note that we cannot simply destroy the delta versions without changing the main index. In fact, although all operations are read-only (i.e., the abstract data structure remains unaltered), the principle of the shuffle index is that the allocation of nodes to blocks is dynamic. It is therefore important to apply the shuffle performed on the delta versions to the main index, so to enjoy the protection of shuffling for subsequent accesses.

If there were a single delta version, applying the performed shuffling on the main index would be simple. Indeed, it would be sufficient to flush the blocks included in the delta version to the main index on disk, as formally proved by the following theorem.

Theorem 5.1. *Given a logical index* $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ *and a delta version* $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ *of* $\mathcal{T}$*, the set* $(\mathcal{T} \cup \Delta_i) \setminus (\{\langle id, V, P\rangle \in \mathcal{T}\colon id \in \mathcal{ID}_i\} \cup \{\langle id, V, T\rangle \in \mathcal{T}\colon id \in \mathcal{ID}_i\})$ *of logical nodes represents logical index* $\mathcal{T}' = \mathcal{T} \oplus \Delta_i$.

Proof. See the Appendix. □

The situation may however be complicated by the presence of several delta versions, which can have operated independently on the same nodes/blocks. In this case, a reconciliation is needed to ensure correctness of the index and, in particular, to ensure that no content is lost and pointers to child blocks are properly set. We first note that, while it is important that shuffling is enforced on the main index, the specific way in which nodes are shuffled (i.e., which node goes to which block) does not have any impact, provided it represents a random permutation. As long as allocation is dynamic, any rearrangement would do. Hence, a straightforward approach to enforce shuffling on the main index would be to download all the blocks contained in the delta versions at the client side, retrieve (by decrypting) the corresponding nodes, allocate them to blocks, and re-upload them at the server by rewriting the involved blocks on the main index. Such a naive approach, requiring to download all the blocks and to re-encrypt all the nodes, is clearly too expensive and not needed. Our approach aims at minimizing the blocks to be downloaded and re-uploaded by limiting these blocks to the ones strictly needed to guarantee correctness or to avoid leakage on the node allocation, while flushing as many blocks as possible directly to disk (without downloading them at the client side).

To determine which blocks need to be downloaded and re-encrypted, we have to identify the blocks for which the presence of multiple delta versions represents a problem. In principle, it is sufficient for two delta versions to have a block (and hence the corresponding node) in common to require checking all the blocks in them, since the node (which should be reported in only one block to the main index) may have been re-allocated to any of the blocks within each delta version. In practice, only the block where the node was originally allocated in the main index and the new block where it has been allocated in each of the delta versions need to be strictly involved in some re-encryption, since the delta versions have conflicting node/block allocations.

We then start by characterizing conflicting node/block allocations among a set of delta versions as follows.

Definition 5.2 (Conflicting allocations). Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index and $\{\Delta_1, \dots, \Delta_n\}$ be a set of delta versions of $\mathcal{T}$. The *conflicting allocations* of Δ_i with respect to $\{\Delta_1, \dots, \Delta_n\} \setminus \{\Delta_i\}$ is a set $\mathcal{C}_i$ of pairs $\langle n_i^a, id_i \rangle$, where $n_i^a \in \Delta_i^a$, $id_i = \phi_i(n_i^a)$, and $\exists n_j^a \in \Delta_j^a$, $i \neq j$, such that either: (1) $n_i^a = n_j^a$ (same node); or (2) $\phi_i(n_i^a) = \phi_j(n_j^a)$ (same block).

It is easy to see that, with respect to nodes, the nodes that are in conflict for a given delta version Δ_i are all those nodes that are also present in another version (i.e., belong to $\Delta_i^a \cap \Delta_j^a$, for some $j \neq i$) or that are contained in blocks which are also present in another version (i.e., are allocated to a block in $\mathcal{ID}_i \cap \mathcal{ID}_j$, for some $j \neq i$). Analogously, with respect to blocks, the blocks that are in conflict for a given delta version Δ_i are all those blocks that are also present in another version (i.e., belong to $\mathcal{ID}_i \cap \mathcal{ID}_j$, for some $j \neq i$) or that contain nodes that are also present in another version (i.e., belong to $\Delta_i^a \cap \Delta_j^a$, for some $j \neq i$). For completeness, Definition 5.2 captures both components representing conflicts, in terms of pairs $\langle$*node*, *block*$\rangle$ since the conflict requires to revisit the allocation of the *node* contained

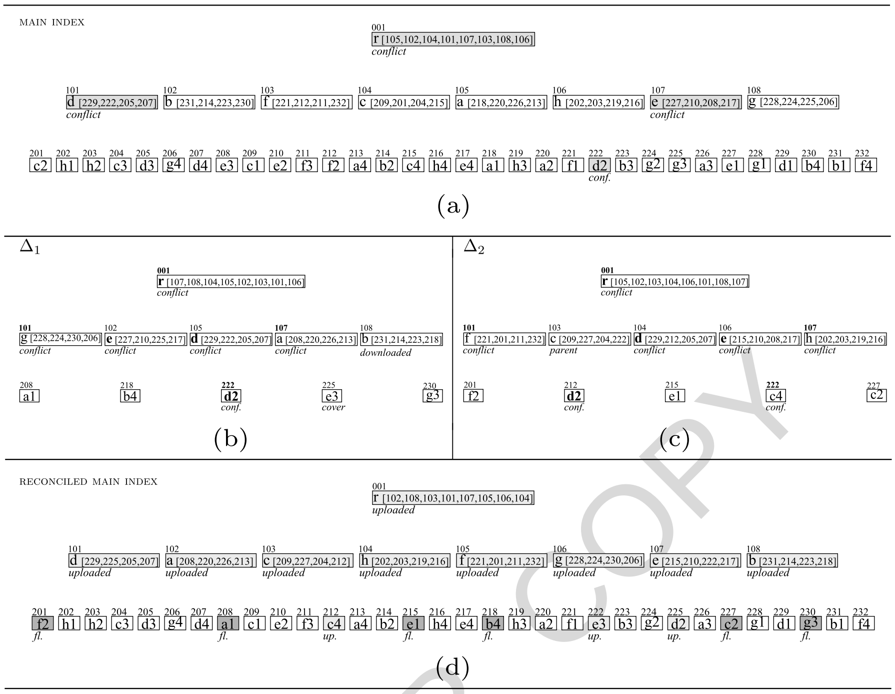

MAIN INDEX
001 r [105,102,104,101,107,103,108,106] conflict
101 d [229,222,205,207] conflict
102 b [231,214,223,230]
103 f [221,212,211,232]
104 c [209,201,204,215]
105 a [218,220,226,213]
106 h [202,203,219,216]
107 e [227,210,208,217] conflict
108 g [228,224,225,206]
201 c2
202 h1
203 h2
204 c3
205 d3
206 g4
207 d4
208 e3
209 c1
210 e2
211 f3
212 f2
213 a4
214 b2
215 c4
216 h4
217 e4
218 a1
219 h3
220 a2
221 f1
222 d2 conf.
223 b3
224 g2
225 g3
226 a3
227 e1
228 g1
229 d1
230 b4
231 b1
232 f4
(a)
Δ_1
001 r [107,108,104,105,102,103,101,106] conflict
101 g [228,224,230,206] conflict
102 e [227,210,225,217] conflict
105 d [229,222,205,207] conflict
107 a [208,220,226,213] conflict
108 b [231,214,223,218] downloaded
208 a1
218 b4
222 d2 conf.
225 e3 cover
230 g3
(b)
Δ_2
001 r [105,102,103,104,106,101,108,107] conflict
101 f [221,201,211,232] conflict
103 c [209,227,204,222] parent
104 d [229,212,205,207] conflict
106 e [215,210,208,217] conflict
107 h [202,203,219,216] conflict
201 f2
212 d2 conf.
215 e1
222 c4 conf.
227 c2
(c)
RECONCILED MAIN INDEX
001 r [102,108,103,101,107,105,106,104] uploaded
101 d [229,225,205,207] uploaded
102 a [208,220,226,213] uploaded
103 c [209,227,204,212] uploaded
104 h [202,203,219,216] uploaded
105 f [221,201,211,232] uploaded
106 g [228,224,230,206] uploaded
107 e [215,210,222,217] uploaded
108 b [231,214,223,218] uploaded
201 f2 fl.
202 h1
203 h2
204 c3
205 d3
206 g4
207 d4
208 a1 fl.
209 c1
210 e2
211 f3
212 c4 up.
213 a4
214 b2
215 e1 fl.
216 h4
217 e4
218 b4 fl.
219 h3
220 a2
221 f1
222 e3 up.
223 b3
224 g2
225 d2 up.
226 a3
227 c2 fl.
228 g1
229 d1
230 g3 fl.
231 b1
232 f4
(d)

Fig. 3. An example of main index (a), two delta versions Δ_1 (b) and Δ_2 (c), and the result of their reconciliation (d).

in the *block*. To illustrate, consider the two delta versions Δ_1 and Δ_2 in Fig. 3(b)–(c). The nodes/blocks representing a conflicting allocation in each version are marked with the word *conflict* (*conf.* for leaves) below each block.

All blocks involved in a conflict for some delta version cannot be simply written to disk as the resulting index would not be correct (some nodes would be lost and others would appear replicated). To ensure consistency of the content, it is important to reconcile the delta versions so that there is agreement – with respect to common nodes or common blocks – on which node is allocated to which block. We capture this by formalizing the definition of reconciled delta version, resulting from a reconciliation of different delta versions, as follows.

Definition 5.3 (Reconciled delta version). Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index, $\{\Delta_1, \ldots, \Delta_n\}$ be a set of delta versions of $\mathcal{T}$, and $\mathcal{C}_i$ be the conflicting allocations of Δ_i with respect to $\{\Delta_1, \ldots, \Delta_n\} \setminus \{\Delta_i\}$, $i = 1, \ldots, n$. A *reconciled delta version* of $\{\Delta_1, \ldots, \Delta_n\}$ is a delta version $\Delta_r = (\Delta_r^a, \mathcal{ID}_r, \phi_r)$ where $\Delta_r^a = \Delta_1^a \cup \cdots \cup \Delta_n^a$, $\mathcal{ID}_r = \mathcal{ID}_1 \cup \cdots \cup \mathcal{ID}_n$, and $\phi_r(n^a) = \phi_i(n^a)$ if $n^a \in \Delta_i^a$ and $\langle n^a, \phi_i(n^a)\rangle \notin \mathcal{C}_i$.

Since Δ_r is a delta version, by Definition 3.2 $\phi_r : \mathcal{T}^a \to \mathcal{ID}$ is a bijective function that associates each abstract node $n^a \in \mathcal{T}^a$ with a logical identifier *id* $\in \mathcal{ID}$.

Function ϕ_r is such that: $\phi_r(n^a) = \phi(n^a)$ if n^a does not belong to any delta version; $\phi_r(n^a) = \phi_i(n^a)$ if n^a belongs to one delta version only (i.e., Δ_i); and $\phi_r(n^a)$ needs to be properly defined as the result of a reconciliation among $\Delta_1, \ldots, \Delta_n$, otherwise. Therefore ϕ_r does not generate any conflicting allocation and correctly associates nodes with physical addresses reserved by the server to the shuffle index.

The reconciled delta version can then be enforced on the shuffle index as in the case of a single delta version, by merging $\mathcal{T}$ and Δ_r producing logical index $\mathcal{T}_r = \mathcal{T} \oplus \Delta_r$ that represents the same abstract index as $\mathcal{T}$.

For producing the reconciled version, in addition to blocks in conflict also the blocks containing a pointer to a block in conflict (e.g., block 103 in Δ_2 in Fig. 3(c)) need to be re-written, as the pointer should be changed to refer to the new block where the child node (e.g., $c4$) has been allocated.

While the blocks in conflict and their parents are the only ones that should be downloaded by the client and re-uploaded (after shuffling the nodes in conflict) to produce a correct reconciled version (all other blocks in the delta versions could simply be flushed to disk directly by the server), we may need to download (and either include in the shuffling or simply re-write) other blocks. The reason is to ensure that the server cannot infer node/block allocations by observing that only few blocks have been involved in a reconciliation. As an example, for Δ_1 in Fig. 3(b), the only leaf block to download and re-upload would be conflicting block 222, therefore the server can infer that it stores the value accessed (as target or cover) by two searches performed with different delta versions. To avoid leakages like this, and provide the same uncertainty over the block allocation enjoyed by the original shuffle index proposal, we require each version, for each level of the index, to: (i) perform shuffling of either 0 or at least *num_cover* $+ 1$ blocks and (ii) flush directly either 0 or not less than *num_cover* $+ 1$ blocks. If for a given level there are less than *num_cover* $+ 1$ blocks to flush, additional cover blocks are also downloaded and re-uploaded after re-encrypting them with a new salt (to make them not recognizable). Like parents, these latter nodes are not involved in the shuffling to avoid propagating the need for changes to higher levels of the index. For instance, with reference to Δ_1 in Fig. 3(b): (i) 225 is added as cover to perform shuffling among at least two nodes at leaf level, and (ii) 108 is also downloaded since it would have been the only one flushed at level one. Figure 3(d) illustrates the merging of the index in Fig. 3(a) after reconciliation of delta versions Δ_1 and Δ_2 in Fig. 3(b)–(c). In the figure, blocks flushed by the server from main memory to disk are represented in dark gray and are marked with the word *fl.* below each of them; blocks that have been downloaded at the client side, re-encrypted, and uploaded back on the server are represented in light gray and are marked with the word *upload* (*up.,* for leaves) below each of them; and blocks that have not been involved in the reconciliation process are represented in white.

6. Algorithm for reconciling delta versions and main index

Algorithm 2, operating at the client side, performs the reconciliation of a set of delta versions and the main index. It works as follows. Given a logical index $\mathcal{T}$ and

Algorithm 2. Reconciliation of delta versions and the main index

```
$\mathcal{T}$=($\mathcal{T}^a$, $\mathcal{ID}$, $\phi$): logical index on a candidate key $K$ with domain $D_K$, height $h$, fan-out $F$
$\Delta_1$,...,$\Delta_n$: delta versions of $\mathcal{T}$
$Allocation_1[0,\ldots,h]$,...,$Allocation_n[0,\ldots,h]$: layered structure that keeps track of node-block associations
                    and of parent-child relationships by storing, for each node in $\Delta_i$, its logical identifier id,
                    its label label, and the label of its parent parent
$num\_cover + 1$: minimum number of nodes to shuffle (to flush, respectively) at each level of each delta version

MAIN
for $i$:=1,...,$n$ do $Allocation_i[0,\ldots,h]$ := Decrypt(Download($Allocation[0,\ldots,h]$,$\Delta_i$))
/* reconcile $\Delta_1$,...,$\Delta_n$ with $\mathcal{T}$ starting from the leaves */
for $l$:=$h$,...,0 do
  $toWrite := \emptyset$, $Replicas := \emptyset$
  for $i$:=1,...,$n$ do
    /* classify the nodes at level $l$ in $\Delta_i$ */
    $ToReconcile_i[l] := \{tr \in Allocation_i[l]: \exists tr' \in Allocation_j[l], i \neq j, tr.label=tr'.label\}$ /* conflicting nodes */
    $ToShuffle_i[l] := ToReconcile_i[l] \cup \{ts \in Allocation_i[l]: \exists ts' \in Allocation_j[l], i \neq j, ts.id=ts'.id\}$ /* conflicting blocks */
    if $l$=$h$ then $ToAdjust_i[l] := \emptyset$
    else $ToAdjust_i[l] := \{ta \in Allocation_i[l]: \exists ta' \in ToShuffle_i[l+1], ta.label=ta'.parent\}$ /* parents of conflicting allocations */
    $Unchanged_i[l] := Allocation_i[l] \setminus (ToShuffle_i[l] \cup ToAdjust_i[l])$ /* nodes to flush */
    /* extend the set of nodes to shuffle to be $num\_cover + 1$ */
    if $0 < |ToShuffle_i[l]| \leqslant num\_cover$ then
      if $|ToShuffle_i[l] \cup ToAdjust_i[l]| > num\_cover$ then /* extend the set of nodes to shuffle with parents of conflicting allocations */
        let $cover \subseteq ToAdjust_i[l]$ s.t. $|ToShuffle_i[l] \cup cover| = num\_cover + 1$
        $ToShuffle_i[l] := ToShuffle_i[l] \cup cover$
        $ToAdjust_i[l] := ToAdjust_i[l] \setminus cover$
      else
        if $|Allocation_i[l]| > num\_cover$ then
          let $cover \subseteq Unchanged_i[l]$ s.t. $|ToShuffle_i[l] \cup ToAdjust_i[l] \cup cover| = num\_cover + 1$
        else $cover := Unchanged_i[l]$
        $ToShuffle_i[l] := ToShuffle_i[l] \cup ToAdjust_i[l] \cup cover$
        $ToAdjust_i[l] := \emptyset$
        $Unchanged_i[l] := Unchanged_i[l] \setminus cover$
    /* extend the set of nodes to flush to be $num\_cover$ */
    if $0 < |Unchanged_i[l]| \leqslant num\_cover$ then
      $ToAdjust_i[l] := ToAdjust_i[l] \cup Unchanged_i[l]$
      $Unchanged_i[l] := \emptyset$
    Blind-Write($Unchanged_i[l]$,$\Delta_i$) /* flush unchanged blocks */
    $Read$ := Decrypt(Download($ToShuffle_i[l] \cup ToAdjust_i[l]$,$\Delta_i$) /* read blocks */
    for each $ta \in (ToAdjust_i[l] \cup ToShuffle_i[l] \setminus ToReconcile_i[l])$ do
      let $block \in Read$ such that $block.id=ta.id$
      if $l<h$ then /* adjust pointer to children in nodes that do not have multiple copies */
        for $j$=1,...,$F$ do
          if $\exists ts' \in ToShuffle_i[l+1]$ s.t. $ts'.id=block.P[j]$ then $block.P[j] := \rho_{l+1}(ts'.label)$
      $toWrite := toWrite \cup \{block\}$
    $Replicas := Replicas \cup \{block \in Read: \exists tr \in ToReconcile_i[l], block.V=tr.V\}$
  /* define an assignment function, with shuffling, for nodes to shuffle */
  $\rho_l$: $\{ts.label{:}ts \in (ToShuffle_1[l] \cup \ldots \cup ToShuffle_n[l])\} \rightarrow \{ts.id{:}ts \in (ToShuffle_1[l] \cup \ldots \cup ToShuffle_n[l])\}$ /* identity function otherwise */
  for each $block \in toWrite$ do $block.id := \rho_l(label(V))$ /* update the block identifier */
  for each $block \in Replicas$ do
    $Copies := \{block \in Replicas{:}block.V=val\}$ /* set of blocks storing the same abstract node */
    if $l<h$ then
      create $block'$ with $block'.id=\rho_l(label(V))$, $block'.V=V$ /* create a new block representing the abstract node */
      for $j$=1,...,$F$ do /* adjust pointers to children */
        for $i$:=1,...,$n$ do
          if $\exists tr \in ToReconcile_i[l]$ s.t. $tr.label=label(V)$ and
            $\exists ta' \in Allocation_i[l+1]$ s.t. $block.P[j]=t'.id$, $block \in Copies$ then
            $block'.P[j] := \rho_{l+1}(ta'.label)$
          else $block'.P[j] := block.P[j]$
    else $block' := \langle \rho_l(label(V)), V, block.T \rangle$ with $block \in Copies$
    $toWrite := toWrite \cup \{block'\}$
    /* write reconciled blocks in the main index */
    for each $block \in toWrite$ do Write($block.id$,Encrypt($salt||block$))
for $i$:=1,...,$d$ do /* remove all delta versions */
  remove $\Delta_i$ from main memory /* server-side */
  VERSION[$i$] := $\langle i, 0, 0 \rangle$ /* client-side */
```

a set $\{\Delta_1, \ldots, \Delta_n\}$ of delta versions of $\mathcal{T}$, the reconciliation algorithm visits all the blocks in the delta versions level by level, following a bottom-up strategy that starts from the leaves. For each level $l = h, \ldots, 0$, the algorithm first partitions the blocks in each delta version in the following three classes (lines 5–24).

- *ToShuffle*$_i[l]$: set of blocks at level l of delta version Δ_i that must be shuffled before writing them in the main index. The blocks that must be shuffled include both conflicting allocations (Definition 5.2) and cover nodes and are at least *num_cover* $+ 1$. In the choice of covers, the algorithm first includes in *ToShuffle*$_i[l]$ the blocks in *ToAdjust*$_i[l]$.
- *ToAdjust*$_i[l]$: set of blocks at level l, with $l < h$, of delta version Δ_i that do not belong to *ToShuffle*$_i[l]$ and that have at least a child in *ToShuffle*$_i[l + 1]$.
- *Unchanged*$_i[l]$: set of blocks at level l of delta version Δ_i that belong neither to *ToShuffle*$_i[l]$ nor to *ToAdjust*$_i[l]$. These nodes must be at least *num_cover* $+ 1$, otherwise they are moved to *ToAdjust*$_i[l]$.

To easily classify the blocks composing a delta version without the need for the client to download all of them, each delta version is associated with a layered structure, called *Allocation*$_i[0, \ldots, h]$, stored at the server side and updated at each shuffling (see Algorithm 1, lines 36–40, 43–45). This structure summarizes the node/block associations and the parent-child relationships between (abstract) nodes in the delta version. For each block at level l in Δ_i, *Allocation*$_i[l]$ stores a triple of the form $\langle \mathit{id}, \mathit{label}, \mathit{parent} \rangle$, where *id* is the block identifier, *label* is the identifier of the values *V* stored in the block (i.e., it is the identifier of the abstract node), and *parent* is the label of the parent of the block. Before starting the reconciliation process, the client downloads from the server these metadata (line 1) and classifies the blocks in each delta version in the server's main memory without the need to download them (lines 2–24). The algorithm then flushes to the main index on disk all the blocks in *Unchanged*$_i[l]$ (line 25) and downloads the blocks in *ToShuffle*$_i[l]$ and *ToAdjust*$_i[l]$, $i = 1, \ldots, n$, whose content or allocation need to be updated (line 26). The algorithm first adjusts the pointers to children of the internal nodes in *ToAdjust*$_i[l]$, $i = 1, \ldots, n$, according to the node/block allocation resulting from the reconciliation of the nodes/blocks at level $l + 1$ (lines 27–31). The algorithm then shuffles and reconciles the blocks in *ToShuffle*$_1[l] \cup \cdots \cup$ *ToShuffle*$_n[l]$, such that no two nodes are allocated to the same block and, vice versa, no two blocks store the same node (line 34). The algorithm also updates the pointers to children in shuffled blocks representing internal nodes, according to the node/block allocation resulting from the reconciliation of the nodes/blocks at level $l + 1$ (lines 35–46). The algorithm encrypts and writes to the main index the resulting blocks (line 47). Finally, the algorithm empties all the delta versions on the server and updates relation VERSION at the client side accordingly (lines 48–50). Note that the reconciliation algorithm, similarly to the search algorithm illustrated in Section 4, requires an interaction between the client and the server that consists of $h + 1$ communication steps. During these steps, the client however does not download all the nodes of the delta versions, but only the ones in *ToShuffle*$_i[l]$ and *ToAdjust*$_i[l]$, with $i = 1, \ldots, n$ and $l = 0, \ldots, h$.

Consider the main shuffle index in Fig. 3(a) and its delta versions in Fig. 3(b)–(c) and assume that *num_cover* $= 1$. Table 2 illustrates an example of execution

Table 2

Reconciliation of the shuffle index in Fig. 3(a) with the two delta versions in Fig. 3(b)–(c)

l	Δ_1			Δ_2				
	ToShuffle	*ToAdjust*	*Unchanged*	*ToShuffle*	*ToAdjust*	*Unchanged*	ρ	*ToWrite*
2			208 $a1$			201 $f2$		
			218 $b4$			215 $e1$		
			230 $g3$			227 $c2$		
	222 $d2$			212 $d2$			d2 → 225	225 $d2$
	225 $e3^*$						e3 → 222	222 $e3$
				222 $c4$			c4 → 212	212 $c4$
1	102 e[227, 210, 225, 217]			106 e[215, 210, 208 ,217]			e → 107	107 e[215, 210, 222, 217]
	105 d[229, 222, 205, 207]			104 d[229, 212, 205 ,207]			d → 101	101 d[229, 225, 205, 207]
	101 g[228, 224, 230, 206]						g → 106	106 g[228, 224, 230, 206]
	107 a[208, 220, 226, 213]						a → 102	102 a[208, 220, 226, 213]
				101 f[221, 201, 211, 232]			f → 105	105 f[221, 201, 211, 232]
				107 h[202, 203, 219, 216]			h → 104	104 h[202, 203, 219, 216]
		108 b[231, 214, 223, 218]*						108 b[231, 214, 223, 218]
					103 c[209, 227, 204, 222]			103 c[209, 227, 204, 212]
0	001 r[107, 108, 104, 105, 102, 103, 101, 106]	001 r[107, 108, 104, 105, 102, 103, 101, 106]		001 r[105, 102, 103, 104, 106, 101, 108, 107]	001 r[105, 102, 103, 104, 106, 101, 108, 107]		r → 001	001 r[102, 108, 103, 101, 107, 105, 106, 104]

of the reconciliation algorithm. The columns of the table represent: the level of the shuffle index (l); the nodes that must be downloaded, reconciled, and shuffled by the data owner (*ToShuffle*); the nodes that must be downloaded and updated but that should not be involved in shuffling operations (*ToAdjust*); the nodes that are flushed directly to disk (*Unchanged*); the assignment of nodes in *ToShuffle* to blocks (ρ); and the blocks written on the server (*ToWrite*). In columns *ToShuffle* and *ToAdjust*, a^* denotes covers. We note that each row in the table illustrates the evolution of an abstract node during the reconciliation of the delta versions with the main index.

It is interesting to note that, for $l = 2$, *ToAdjust* is empty for both the delta versions since the visited nodes are leaves. Also, node 225 storing value $e3$ is moved from *Unchanged* to *ToShuffle*, since *ToShuffle* would otherwise include less than *num_cover* $+ 1$ nodes. For $l = 1$, *Unchanged* is empty for both the delta versions, since the only node in *Unchanged* for Δ_1 is moved to *ToAdjust* to guarantee that *Unchanged* includes either 0 or at least *num_cover* $+ 1$ nodes in each delta version. For $l = 0$, the root note clearly belongs to *ToAdjust* since its children has been shuffled and therefore its content must be updated accordingly. It is easy to see from the figure that each abstract node is written back to the server only once, only the nodes in *ToShuffle* are possibly associated with a different block, while nodes in *ToAdjust* are stored in the same block after updating pointers to children.

The main index $\mathcal{T}'$ computed by Algorithm 2 is the result of the merge of the original index $\mathcal{T}$ with a delta version Δ_ρ that includes all the blocks/nodes in all delta versions and that shuffles (at least) all the nodes/blocks with conflicting allocations. That is, Δ_ρ guarantees that the node/block allocation of blocks that do not belong to any delta version remains unchanged, while any other allocation may change. Hence, Δ_ρ can be seen as an *extended reconciled delta version* of $\{\Delta_1, \ldots, \Delta_n\}$ and $\mathcal{T}$ (i.e., it represents a reconciled delta version possibly including additional cover nodes). Formally, the correctness of Algorithm 2 is proven by the following theorem.

Theorem 6.1. *Given a logical index* $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$*, a set* $\{\Delta_1, \ldots, \Delta_n\}$ *of delta versions of* $\mathcal{T}$*, and* $\Delta_r = (\Delta_r^a, \mathcal{ID}_r, \phi_r)$ *a reconciled delta version of* $\{\Delta_1, \ldots, \Delta_n\}$ *with* $\mathcal{T}$*, the logical index* $\mathcal{T}' = (\mathcal{T}^a, \mathcal{ID}, \phi_\rho)$ *computed by Algorithm* 2 *represents* $\mathcal{T} \oplus \Delta_\rho$*, where* $\Delta_\rho = (\Delta_\rho^a, \mathcal{ID}_\rho, \phi_\rho)$ *with* $\Delta_\rho^a = \Delta_r^a$*,* $\mathcal{ID}_\rho = \mathcal{ID}_r$*, and* $\phi_\rho(n^a) = \phi(n^a)$ *if* $n^a \notin \Delta_\rho^a$*.*

Proof. See the Appendix. □

7. Supporting multiple indexes

The adoption of a shuffle index built on a candidate key K permits to efficiently evaluate conditions of the form $K = v$, with v a value in the domain of K, without revealing to the server any information about the target v of the access. The evaluation of conditions on attributes different from K requires instead the data owner to download all the tuples in r and to locally evaluate the search condition on them. The computational cost of these queries could potentially be quite high. To efficiently support the evaluation of these queries at the server side, even when their frequency is low, the data owner has to define additional indexes on different attributes of the outsourced relation. If the data owner adopts index structures that statically allocate data to disk blocks, the definition of more than one index on the same relation may compromise both data and access confidentiality, since the storage server can exploit static indexes for drawing inferences. In fact, the adoption of an index structure that does not change data allocation at each access permits the server to keep track of the frequency with which the content of each block is accessed by users' queries. If the server knows the frequency distribution of accesses to values in the domain of the indexed attribute, it can exploit the collected information to infer both the target of each access and the value of the indexed attribute stored in each (encrypted) block. Also, the presence of multiple indexes defined on the same relation may enable the server to reconstruct the associations among the values of indexed attributes in each tuple of the relation. As an example, suppose that the outsourced relation stores the medical data of the patients in a hospital and that the data owner defines two indexes, on attribute *Name* and on attribute *Disease*, respectively. By monitoring the frequency of accesses to blocks, the server might possibly reconstruct, not only the names of the hospitalized patients, but also the diseases of some of them (which should be kept secret). An evaluation of the risk can be found in [4], where the exposure deriving from access to index structures over encrypted data has been investigated. In fact, as shown in [4], knowledge of the frequency distribution of values can enable the server to reconstruct, for a significant fraction of the data, the correspondence between the plaintext data and the values of the index.

In this section, we extend our shuffle index data structure to support multiple indexes.

7.1. Secondary indexes and combined shuffle index

We define a set of *secondary shuffle indexes* over the attributes frequently involved in accesses to the outsourced relation. Consistently with usual practice in commercial systems, we assume that both the primary and secondary shuffle indexes are defined over candidate keys, that is, each value in the domain of the indexed attributes appears at most in one tuple of the outsourced relation.

At the *abstract level*, primary and secondary shuffle indexes represent independent abstract data structures. Given candidate key A_i, the abstract secondary index $\mathcal{T}_i^a$ on A_i is a set $\{n_1^a, \ldots, n_m^a\}$ of abstract nodes forming an unchained $B+$-tree. The internal nodes of the secondary index are pairs of the form $n^a = \langle \mathit{Values}, \mathit{Children} \rangle$, with *Values* a list of index values and *Children* a list of $q + 1$ child nodes. The leaf nodes are pairs of the form $n^a = \langle \mathit{Values}, \mathit{Key\text{-}values} \rangle$, where *Key-values* represents the set of values of attribute K (on which the primary shuffle index has been defined) for the tuples such that $t[A_i] \in$ *Values* (the primary shuffle index has instead the tuples in

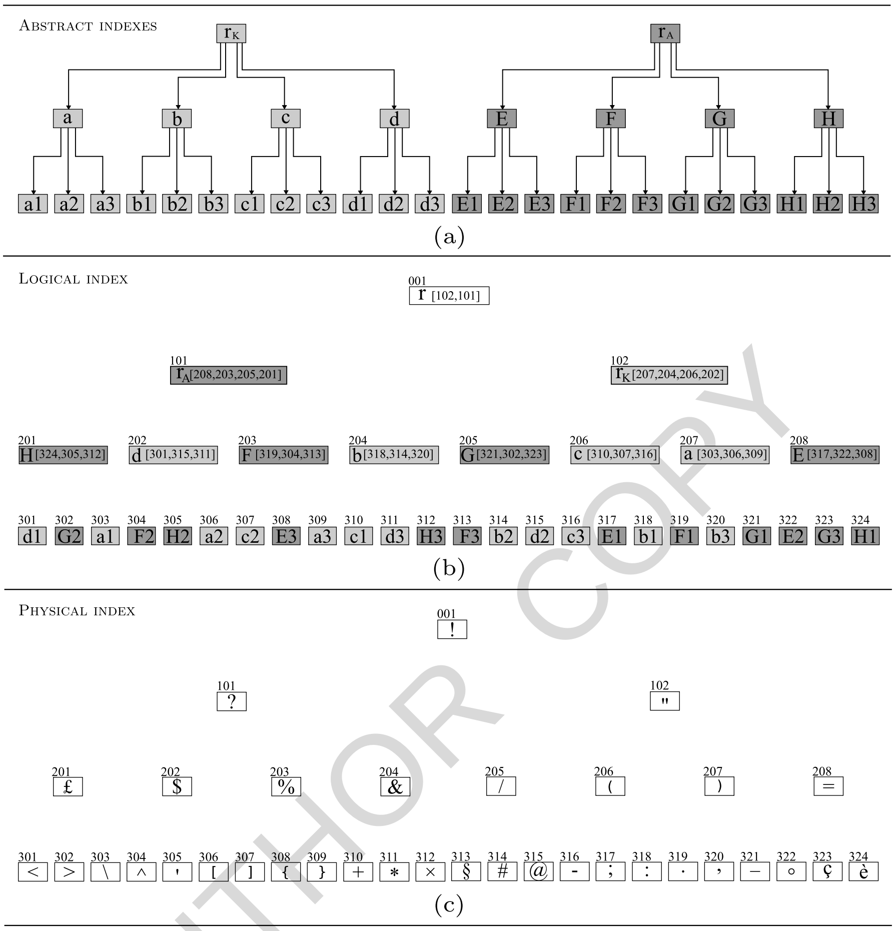

Abstract indexes
r_K
r_A
a
b
c
d
E
F
G
H
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3 E1 E2 E3 F1 F2 F3 G1 G2 G3 H1 H2 H3
(a)
Logical index
001 r [102,101]
101 r_A[208,203,205,201]
102 r_K[207,204,206,202]
201 H[324,305,312]
202 d [301,315,311]
203 F [319,304,313]
204 b[318,314,320]
205 G[321,302,323]
206 c [310,307,316]
207 a [303,306,309]
208 E [317,322,308]
301 d1 302 G2 303 a1 304 F2 305 H2 306 a2 307 c2 308 E3 309 a3 310 c1 311 d3 312 H3 313 F3 314 b2 315 d2 316 c3 317 E1 318 b1 319 F1 320 b3 321 G1 322 E2 323 G3 324 H1
(b)
Physical index
001 !
101 ?
102 "
201 £
202 $
203 %
204 &
205 /
206 (
207)
208 =
301 < 302 > 303 \ 304 ^ 305 ' 306 [307] 308 { 309 } 310 + 311 * 312 × 313 § 314 # 315 @ 316 - 317 ; 318 : 319 · 320 , 321 – 322 ◦ 323 ç 324 è
(c)

Fig. 4. An example of an abstract primary and secondary index (a) and the corresponding logical (b) and physical (c) combined shuffle index.

the leaves). Figure 4(a) illustrates, on the right-hand side, an abstract secondary index defined on the same outsourced relation as the primary index in Fig. 1(a), which has been reported on the left-hand side of Fig. 4(a) for the reader's convenience.

At the *logical level*, the primary and secondary indexes can either be kept separate or combined in a single shuffle index representing all the abstract structures defined on the outsourced relation. The main advantage of keeping indexes separate at the logical level is that each index can be managed independently from the others. The drawback is that an observer (e.g., the storage server) can monitor the attribute involved in each query evaluation (i.e., the attribute on which the search condition is defined). To prevent this leakage of information, we combine all the abstract indexes defined for the outsourced relation in a single *combined shuffle index* at the logical level. This permits to shuffle blocks that store nodes of different abstract structures, breaking the correspondence between blocks and the abstract index to which the nodes they store belong (i.e., the attribute on which the index has been defined). All the abstract indexes defined on the outsourced relation must have the same height (i.e., the leaves of all the indexes must be at the same level) to be combined in one logical structure. In fact, if the abstract indexes had different heights, the paths visited while accessing the logical index would be composed of a different number of nodes depending on the abstract index to which the path belongs, potentially disclosing to the server the attributes involved (either as target, cover, or repeated searches) in each access to the index. Note however that, since the fan-out of $B+$-trees is usually relatively high, forcing the leaves of all abstract indexes to be at the same level can only modestly increase (at most one level) the height of each index.

To combine multiple abstract indexes in one logical structure, we add a prefix to all the values in each abstract node. This prefix represents the attribute on which the index has been defined. We then create an auxiliary root node for the structure, whose children are the roots of the abstract primary and secondary indexes defined for the outsourced relation. At the logical level, a combined shuffle index is defined as follows.

Definition 7.1 (Combined shuffle index – Logical level). Let $R(A_1,\ldots,A_n)$ be the outsourced relation, and $\mathcal{I} = \{K, A_i, \ldots, A_j\} \subseteq \{A_1, \ldots, A_n\}$ be the set of attributes on which either the primary or a secondary index has been defined. A *logical index* $\mathcal{T}$ for R over $\mathcal{I}$ is a triple $(\mathcal{T}^a, \mathcal{ID}, \phi)$ where:

- $\mathcal{T}^a = \{A_i \cdot n^a \mid A_i \in \mathcal{I} \wedge n^a \in \mathcal{T}_i^a\} \cup \{root\}$ is the set of nodes in the abstract indexes $\mathcal{T}_i^a$, $A_i \in \mathcal{I}$, defined on the outsourced relation and the auxiliary root node $root = \langle \mathcal{I}, Children \rangle$, with *Children* the root nodes of the abstract indexes;
- $\mathcal{ID}$ is a set of logical identifiers;
- $\phi : \mathcal{T}^a \rightarrow \mathcal{ID}$ is a bijective function associating each abstract node $A_i \cdot n^a$ in $\mathcal{T}^a$ with a logical identifier *id* in $\mathcal{ID}$.

The logical index determines how the nodes in the abstract structures $\mathcal{T}_i^a$, with $A_i \in \mathcal{I}$, are allocated to logical identifiers in $\mathcal{ID}$. Note that the allocation of abstract nodes to logical identifiers is independent from the abstract index to which nodes belong (i.e., nodes of the same abstract index may be allocated to non-contiguous identifiers). As discussed in Section 3, each internal node $n^a = \langle Values, Children \rangle$ of an abstract index is represented by a (logical) node of the form $\langle id, V, P \rangle$, where $id = \phi(n^a)$, $V = Values$, and $P[j] = \phi(Children[j])$, $j = 1, \ldots, q+1$. The leaf nodes of the logical index may have a different structure, depending on whether they represent the leaves of the primary index or the leaves of a secondary index. Leaves of the primary index are of the form $\langle id, V, T \rangle$, where T is a set of tuples. Leaves of the secondary indexes are of the form $\langle id, V, K\text{-}v \rangle$, where *K-v* is a set of values in the domain of attribute K on which the primary index has been defined. Figure 4(b) illustrates the combined shuffle index representing the two abstract structures in Fig. 4(a). For simplicity, in the figure we do not report the prefix of each value stored in logical nodes, but we distinguish the nodes of the primary index from the nodes of the secondary index by denoting them with a different color.

At the *physical level*, a combined shuffle index $\mathcal{T}$ is stored in a set of (disk) blocks $\mathcal{T}^e$ that contain the encrypted representation of the nodes in $\mathcal{T}$. Figure 4(c) illustrates the physical representation of the logical structure in Fig. 4(b).

7.2. Access to data via secondary indexes

The evaluation of a search condition $t[A_i] = v_i$, with A_i an attribute used for building a secondary shuffle index and v_i a value in the domain of attribute A_i, operates in two steps:

(1) search for v_i to retrieve the value v_k in the domain of attribute K such that $\exists t \in r$: $t[A_i] = v_i \wedge t[K] = v_k$;
(2) search for v_k to retrieve the tuple t with $t[K] = v_k$ to be returned in response to the query.

If the first access to the combined shuffle index returns an empty result, the data owner does not need to evaluate the second search since no tuple in the outsourced relation satisfies the search condition. Otherwise, if the first access returns a value, the second search will certainly return a tuple. Note that the server cannot infer, by observing two subsequent accesses to the combined shuffle index, whether they are related to the evaluation of the same search condition. In fact, two subsequent accesses related to the evaluation of the same condition adopt different target, cover, and repeated searches. Thanks to the combination of all abstract indexes in one logical structure, the data owner can choose cover and repeated searches from the domain of any attribute on which an index (either primary or secondary) has been defined. In the choice of cover searches, the data owner must guarantee [9]: (i) the indistinguishability of target and covers to the server's eyes, and (ii) that the paths that correspond to target, cover, and repeated searches are disjoint, with only the auxiliary root and possibly the nodes at level 1 in common (i.e., the access visits disjoint paths on the abstract indexes).

To illustrate, consider a search for A's value $H1$ over the index in Fig. 4(b) and assume *num_cover* $= 1$. The data owner first calls Algorithm 1 searching for value $H1$, using $F2$ as cover and $d2$ as repeated search. The algorithm download blocks: (001) at level 0, $(101, 102)$ at lever 1, $(201, 202, 203)$ at level 2 and $(304, 315, 324)$ at level 3. The leaves accessed by the search algorithm are therefore: $\langle 304, F2, d2\rangle$, $\langle 315, d2, T_{d2}\rangle$ and $\langle 324, H1, b3\rangle$, where T_{d2} is the tuple in the outsourced relation with value $d2$ for attribute K. The data owner invokes again Algorithm 1 searching for value $b3$, using $a2$ as cover and $F2$ as repeated search. The algorithm, in this second access to the combined shuffle index, downloads blocks: (001) at level 0,

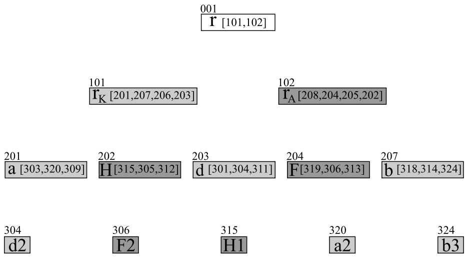


Fig. 5. An example of delta version of the combined shuffle index in Fig. 4.

$(101, 102)$ at lever 1, $(201, 204, 207)$ at level 2 and $(306, 315, 320)$ at level 3. The leaves accessed by the search algorithm are then: $\langle 306, a2, T_{a2}\rangle$, $\langle 315, F2, d2\rangle$ and $\langle 320, b3, T_{b3}\rangle$, where T_{a2} and T_{b3} are the tuples in the outsourced relation with value $a2$ and $b3$ for attribute K, respectively. Tuple T_{b3} is the only tuple in the outsourced relation satisfying the search condition (i.e., composing the query result).

The support of multiple indexes nicely complements the support of concurrency without requiring any special reconsideration. In fact, having merged primary and secondary indexes in a single data structure essentially makes the presence of multiple indexes transparent to the concurrency manager. The only note to make is that delta versions will be defined on the combined shuffle index (in contrast to the single primary index). Figure 5 illustrates the delta version resulting from the search for value $H1$ over the combined index in Fig. 4 described above.

8. Security analysis

We analyze the protection offered by our proposal for the new aspects introduced with respect to the serial version operating only with the main index built on one candidate key. When a secondary index is used, it is first necessary to retrieve the node in the secondary index that satisfies the search condition and then execute an independent access to the primary shuffle index, searching for the value retrieved by the first search.

Like in the original proposal, we focus the analysis on leaves of the index (nodes at a higher level are subject to a greater number of accesses, due to the multiple paths that pass through them, and are then involved in a larger number of shuffling operations, which increase their protection). Since a combined shuffle index is, at the logical and physical levels, indistinguishable from a (primary) shuffle index and our search operations execute essentially like in the original proposal (with repeated searches instead of cache), our solution enjoys the protection guarantees given by cover searches like in [9]. The only potential exposure in our solution is when two different delta versions require access to the same block in the main index for the first time. Since the main index changes only upon reconciliation, the server can infer that the two requests actually refer to the same node. However, since every access execution entails reading at least $\mathit{num\_cover}+1$ blocks (in addition to the repeated search) at every level, and covers are chosen guaranteeing indistinguishability (with respect to access profiles) between targets and covers, the server cannot determine whether the transactions operating on the two different delta versions are actually aiming at the same target, or either or both of them are accessing the block as a cover. The probability that the two transactions aimed at the same target is then $\frac{1}{(\mathit{num\_cover}+1)^2}$. When m delta versions request access to the same block from the main index, the probability that all the transactions aimed at the same target is $\frac{1}{(\mathit{num\_cover}+1)^m}$, as formally stated by the following theorem.

Theorem 8.1. *When the server detects m conflicts over a single physical block in m versions, the probability that any pair of them is due to accesses to the same logical node as a target is $\frac{1}{(\mathit{num\_cover}+1)^2}$. The probability that all of them are due to accesses to the same logical node as a target is $\frac{1}{(\mathit{num\_cover}+1)^m}$.*

Proof. See the Appendix. □

The crucial property we are interested in evaluating is the protection against the inferences the server may make on the data content by exploiting information on the frequency of accesses to the blocks. Applying classical concepts of information theory, we can model the information available to the server on the association between a node n_i^a and the block id_j storing it as probability $\mathcal{P}(n_i^a, id_j)$. A value equal to 1 for this probability means that the server will be able to correctly identify the node-block correspondence, whereas a value equal to $\frac{1}{|\mathcal{T}^a|}$ will correspond to the absence of any knowledge. If the block is replicated in delta versions, each instance will be associated with the analogous probability. Let ID' be the set of blocks involved in an access in a version (excluding the repeated search). For all $n_i^a \in \mathcal{T}^a$, and for all $id_j \in \mathit{ID}'$, after the shuffling $\mathcal{P}(n_i^a, id_j)$ becomes $\sum_{id_j \in \mathit{ID}'} \frac{\mathcal{P}(n_i^a, id_j)}{\mathit{num\_cover}+1}$, because the shuffling can associate each node with any of the blocks involved in the access with equal probability, thus flattening the probability distribution. After the reconciliation, all the blocks that have been accessed by a single version will be transferred to the main index, where they will be associated with the probabilities computed in the version. Blocks accessed by multiple versions will be shuffled together, with a further averaging of probabilities among the blocks. In fact, since the set of blocks that are shuffled in each level of each delta version is at least $\mathit{num\_cover}+1$, Algorithm 2 guarantees that the server cannot infer with probability higher than $\frac{1}{\mathit{num\_cover}+1}$ the block that stores, after reconciliation, each of the nodes that have been accessed by more than one delta version, as formally proved by the following theorem.

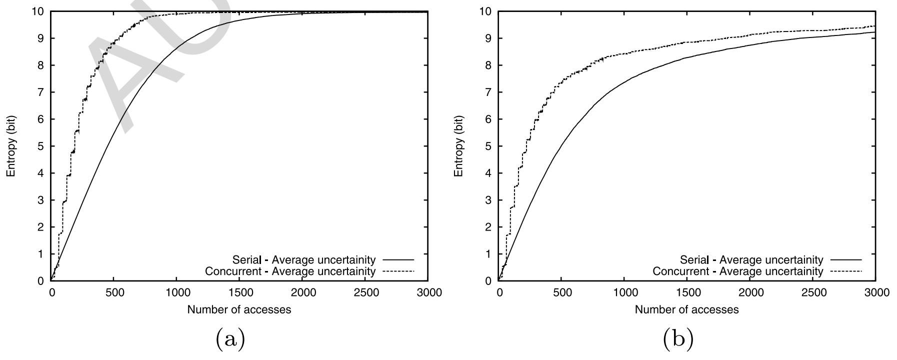
Theorem 8.2. *Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index, $\{\Delta_1, \ldots, \Delta_n\}$ be a set of delta versions of $\mathcal{T}$, $\mathcal{T}'$ be the logical index resulting from reconciling $\{\Delta_1, \ldots, \Delta_n\}$*

and $\mathcal{T}$ through Algorithm 2, and $n^a \in (\Delta_i \cap \Delta_j)$, $i, j = 1, \ldots, n$, $i \neq j$. The server has probability lower than $\frac{1}{num\_cover+1}$ to identify the block where n^a is allocated.

Proof. See the Appendix. □

As a consequence of this theorem and of the above observations, for each node n_i^a, $\mathcal{P}(n_i^a, id_j)$ will progressively move toward value $\frac{1}{|\mathcal{T}^a|}$ after each access and each reconciliation.

It is natural to study the evolution of these probabilities using the concept of entropy, which allows us to identify at an aggregate level the knowledge of the server and its degradation due to shuffling and merging. In particular, we are interested in the impact of delta versions over the entropy, which we evaluated – as common in the study of codes and channels when analytical models become unmanageable – experimentally. We then designed a set of experiments with an initial configuration corresponding to a worst case assumption where the server has a precise knowledge about the node-block correspondence (the entropy is then equal to zero) and evaluated how the entropy increases with access execution (for the serial index) and with access execution and merging after reconciliation (for our proposal). The experiments have considered a variety of configurations, with different numbers of nodes, numbers of versions, values for *num_cover*, and access profiles. Access profiles have been simulated by synthetically generating a sequence of accesses that follow a self-similar probability distribution with skewness γ in the range $[0.25, 0.5]$ (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). We then applied the same sequence of accesses to the serial and concurrent shuffle index and evaluated the growth of the entropy. Figure 6 illustrates the experimental results using 4 covers, 4 versions, 1000 nodes, skewness γ equal to 0.5 and 0.25, and varying the number of accesses. Experiments with different configurations presented a similar behavior.



(a) (b)

Fig. 6. Evolution of the entropy for values of γ equal to 0.5 (a) and 0.25 (b).

As visible from the figure, the evolution of entropy in the concurrent scenario does not have a smooth trend, but it is characterized by two growing trends: a lower growing rate, characterizing the time window between two reconciliations, and a higher growing rate, characterizing reconciliations. Before the first reconciliation, the entropy is slightly lower in the concurrent scenario than in the serial index. The reason is that each delta version serves a smaller number of accesses than the shuffle index in the serial version (assuming uniform distribution of load among versions, each transaction has one fourth of the accesses operating on the main index). However, already at the first reconciliation, the entropy for the concurrent scenario becomes higher than that of the serial scenario, and remains higher. While an even higher entropy might sound not intuitive and an unexpected advantage (more protection with better performance), the explanation for such a behavior is simply that reconciliation and merging enjoy shuffling over a larger number of nodes all at one time. In fact, reconciliation makes the concurrent shuffle index stronger because this phase applies a shuffle over all the nodes in the conflict set. The size of this set depends on the number of conflicts and our model forces it to be for each delta version at least as large as the number of covers used for every shuffle. The size of the conflict set will often be greater than the number of covers, and the growth of entropy produced by a shuffle increases more than linearly with the number of blocks involved in the shuffle (i.e., the execution of two shuffles over two sets of m distinct elements produces lower entropy than a single shuffle over a set of $2m$ elements). The cost of such better protection can be due to reconciliation, which is below 10% of the access cost in the configuration that maximizes the server throughput (Section 9).

9. Performance analysis

We implemented the search and reconciliation algorithms with Java programs. To assess the system performance, we used a data set of 1TB stored in the leaves of a shuffle index with 4 levels. The size of the nodes of the shuffle index was 8 KB. The hardware used in the experiments included a server machine with 2 Intel Xeon Quad 2.0 GHz L3-4 MB, 12 GB RAM, four 1TB disks, 7200RPM, 32 MB cache, and Linux Ubuntu with the ext4 file system, and a client machine with an Intel Core 2 Duo CPU T5500 at 1.66 GHz, 2 GB DRAM, and Linux Ubuntu. The client and the server operate in a local area network (100 Mbps Ethernet, with average RTT of 0.48 ms). The results reported in this section have been obtained as the average over 50 runs and, for each run, the number of accesses is 5000 and the number of covers adopted at each access is 4. The inverse of the average disk time needed to perform a single search gives an upper bound of 52 tps to the maximum throughput of the system.

To emulate the workload of an outsourcing service, we designed a generator scheme, modeling the number of access requests per second as a random variable following a Poisson distribution with mean arrival rate λ (the time when an access request arrives is independent from the time of arrival of previous requests). In our experiments, we considered $\lambda = 16$ tps and $\lambda = 32$ tps, which correspond to 30% and to 60%, respectively, of the physical maximum throughput (52 tps). These are sensible workloads for a service hosted on a single machine and a robust test for the deployment of the proposed solution in a real world scenario. In fact, a workload of 60% of the maximum disk service rate is known to be optimal with respect to the upper bound of the physical maximum throughput [18]. Due to the value of the maximum throughput, well below the ability of the program to simulate requests, a single emulator in the client can adequately model the requests originating from multiple transactions in different network locations.

To evaluate the performance gain obtained with the support of concurrent searches and the overhead due to reconciliation, we compare the results obtained in three different scenarios: (i) serial shuffle index [9]; (ii) concurrent shuffle index where delta versions are never reconciled; and (iii) concurrent shuffle index where delta versions are periodically reconciled. In the experiments, delta versions are reconciled every 128 and every 256 access requests, for the configuration with $\lambda = 16$ tps and $\lambda = 32$ tps, respectively. A higher reconciliation frequency increases the overhead because it more often requires write locks on the disk blocks to be re-written. On the other hand, a lower frequency requires less often such locks but over a considerably larger number of blocks (conflicts among versions grow more than linearly with respect to the number of access requests). Figure 7 shows as the chosen threshold values enable us to balance these two aspects and optimize the server throughput for the employed operating setup.

To assess the performance of our model we analyzed the following three parameters: (i) the throughput; (ii) the average service time (server-side); and (iii) the average response time (client-side). The results are described in the following.

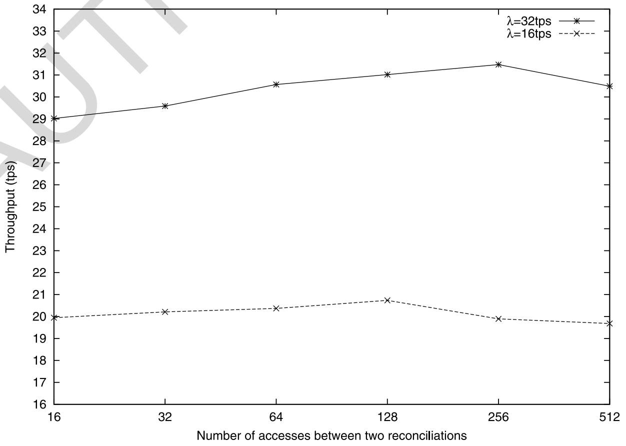
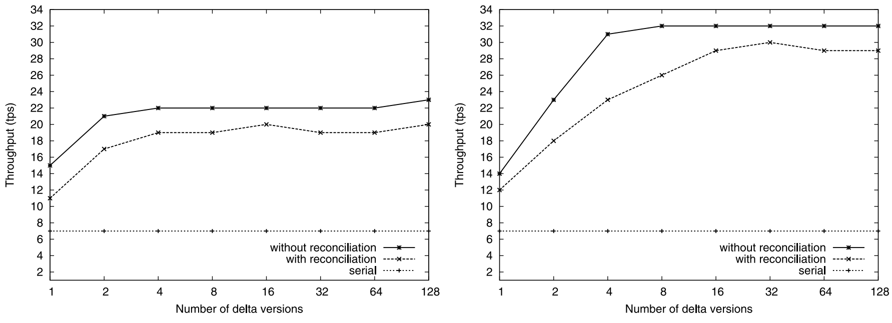


Fig. 7. Server throughput varying the number of access requests between two subsequent reconciliations.



(a) (b)

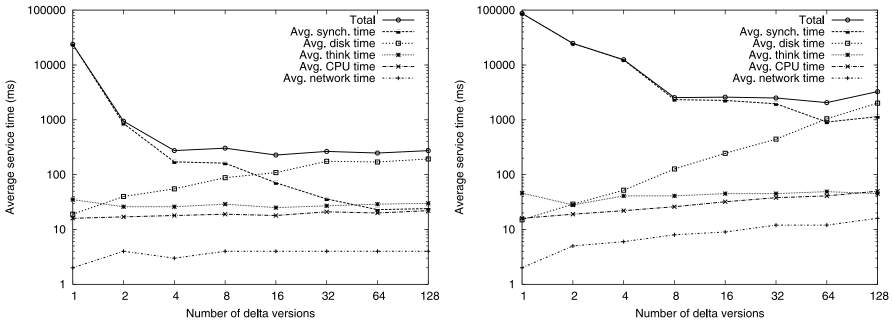
Fig. 8. Server throughput varying the number of delta versions between 1 and 128, with access request arrival rate equal to $\lambda = 16$ tps (a) and $\lambda = 32$ tps (b).

9.1. Throughput

Figure 8(a)–(b) report the server throughput, varying the maximum number of delta versions between 1 and 128 with access request arrival rate equal to $\lambda = 16$ tps and $\lambda = 32$ tps, respectively. Although the performance overhead of concurrent applications highly depends on the random disk access patterns required to execute read and write accesses to blocks, Fig. 8 demonstrates how the adoption of our concurrency support offers a threefold (fourfold, respectively) increase of the server throughput compared to the serial shuffle index when $\lambda = 16$ tps ($\lambda = 32$ tps, respectively). Note that the server throughput is higher than or equal to the mean arrival rate λ of client requests, meaning that the time necessary to the server to process an access request is lower than the time between two consecutive accesses. Figure 8 also highlights the limited cost due to reconciliation, which has a maximum of 25% and is 6% in the configuration that maximizes the server throughput.

9.2. Average service time

The service time necessary to the server to provide a response to an access request depends on different factors: disk time, synchronization time (i.e., time for locking the data structures in main memory and time spent for the periodic reconciliation procedures), network time, CPU time, and think time (i.e., time due to the protocol latencies at the client side). Figure 9(a)–(b) report the components influencing the average service time, varying the maximum number of delta versions between 1 and 128, with access request arrival rate equal to $\lambda = 16$ tps and $\lambda = 32$ tps, respectively. It is immediate to see that the component that most affects the average service time is the average disk time, which grows exponentially with the maximum number of delta versions (this trend is worse in configurations with a high system workload, that is, for $\lambda = 32$ tps). Although the average disk time increases exponentially with the maximum number of delta versions, the average service time does not increase.



(a) (b)

Fig. 9. Average service time varying the number of delta versions between 1 and 128, with access request arrival rate equal to $\lambda = 16$ tps (a) and $\lambda = 32$ tps (b).

This is due to the fact that the average synchronization time quickly decreases as the maximum number of delta versions grows, since resource contention drops. The average network, CPU, and think times have instead a low impact on the average service time. Indeed, the average network time only slightly grows with the maximum number of delta versions and the average CPU and think times are constant regardless of the number of ongoing transactions. In fact, the process communication costs are negligible with respect to the computational time demands of the server application. Analogously, the latencies imposed by the interactive access protocol at the client side are independent from the concurrent management of transactions.

9.3. Average response time

To minimize the time required to complete an access request, we need to carefully choose the number of delta versions to adopt for supporting concurrency. Figure 10 illustrates the average response time, varying the maximum number of delta versions between 1 and 128, when the access request arrival rate is equal to $\lambda = 16$ tps and $\lambda = 32$ tps. Figure 10 shows that the average response time is minimized by using 32 delta versions, and it is equal to 250 ms for $\lambda = 16$ tps and to 2000 ms for $\lambda = 32$ tps.

10. Related work

Previous work is related to the definition of indexing structures for the execution of queries on encrypted outsourced data (e.g., [1,8,16,17,23,24,26]). The proposals in [8,26] specifically adopt the $B+$-tree and the B-tree data structures to define an index able to efficiently support search operations on the key attribute. Although these solutions efficiently support accesses to the outsourced data, they suffer from inference attacks when even a limited number of indexes is published [4]. Indeed,

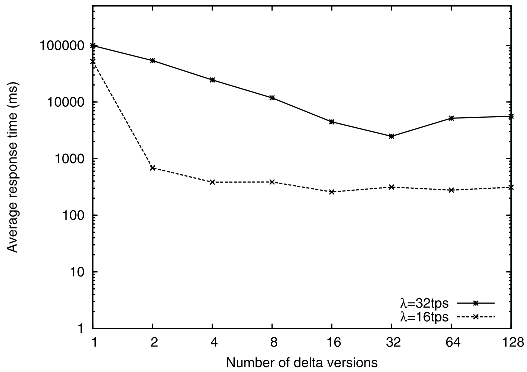


Fig. 10. Average response time varying the number of delta versions between 1 and 128.

they are static and do not offer protection against the attacks based on the frequency of the accesses. Another line of work related to our is represented by cryptographic techniques proposed for hiding to the server the value (or set thereof) in which a user querying outsourced data is interested [12,21,27]. These proposals however do not address access confidentiality, and are therefore not applicable to the considered scenario. Other related works are in the area of Private Information Retrieval (PIR) [6]. In these works, a database is typically modeled as a N-bit string and a user is interested in retrieving the i-th bit of the collection without allowing the server to know/infer which is the bit the user is interested in. In general, PIR proposals can be classified in two main classes: *information-theoretic* PIR and *computational* PIR. Information-theoretic PIR protocols prevent an attacker with unlimited computing power to learn any information about the user's query [2,6]. Computational PIR protocols preserve the privacy of queries against adversaries restricted to polynomial-time computations [3,5]. Recently, traditional PIR protocols have been integrated with relational databases, to the aim of protecting sensitive data (i.e., constant values) within SQL query conditions while providing the client with efficient query evaluation [22]. The original query formulated by the client is properly sanitized before execution and the client resorts to traditional PIR protocols, operating on the sanitized query result only, to extract the tuples of interest. The difference between PIR proposals and our solution is that PIR protocols suffer from a high computational overhead [25] and typically protect the confidentiality of users' queries, while data confidentiality is not considered an issue.

The proposals in [9,11,13,19,20,28,29] aim at protecting data confidentiality and the accesses realized by the client over the data. The solution in [19] is based on the definition of a B-tree index and of a technique for accessing the content of a node in the tree that prevents the server from inferring which node has been accessed. However, the server can observe repeated accesses to the same physical block, which correspond to repeated searches for the same values, and launch a frequency attack to infer information about the values stored in each node of the B-tree. To counteract this shortcoming, in [20] the authors propose a solution based on the definition of a fixed query plan, which is however impractical. The proposal in [28], aimed at preserving both access and pattern confidentiality, adopts the pyramid-shaped database layout of Oblivious RAM [15] and an enhanced reordering technique between adjacent levels of the data structure to protect both data confidentiality and the secrecy of users' queries. The performance of a search operation is however highly affected by the reordering of lower levels of the database, since this reordering can take hours and needs to be periodically performed. This appears a strong obstacle to the real deployment of such a solution. Also, this proposal assumes the presence of a secure coprocessor on the server, trusted by the client. The proposal in [13] exploits Oblivious RAM layout as well and proposes an enhanced management of the shuffling-based approach, by limiting the shuffling to accessed records only. To this aim, it exploits a cache managed by a secure coprocessor operating on the server. The main drawback of this solution is that it relies on a secure coprocessor for guaranteeing access pattern confidentiality, and its security naturally depends on the size of the local cache, which will typically be limited. The first proposal combining shuffling, cover searches, and cache to offer an extensive protection of confidentiality with a limited overhead in response times is illustrated in [9], where data are organized according to a novel data structure whose management does not rely on a trusted component at the server side. A similar solution has been proposed subsequently in [29], where the authors combine cover searches and shuffling to protect access confidentiality. This approach is however less flexible and less efficient than the proposal in [9] as it does not adopt a $B+$-tree index structure and it uses a constant number of cover searches (equal to two). Although all these proposals aim at providing access confidentiality, they offer support neither for concurrent accesses nor for the efficient evaluation of searches for values of attributes different from the candidate key on which the primary index has been defined.

11. Conclusions

Dynamically allocated data structures have recently emerged as a promising solution to provide privacy protection of data whose storage and management are delegated to external servers. Such solutions, working on an index defined over the data and requesting write locks at every access (to enforce dynamic allocation) may result limited and affect performance in scenarios where multiple transactions need to operate concurrently or searches based on attributes different from the primary key need to be supported. In this paper, we have addressed these limitations and extended the recently proposed shuffle index approach to efficiently support concurrent transactions and multiple indexes. Our solution provides data and access privacy, also against frequency attacks by the server, comparable to or better than the original (serial) shuffle index approach. Furthermore, it provides support for the evaluation at the server side of a wider set of queries and a up to fourfold throughput in case of concurrent accesses, thus providing a convincing argument for its adoption.

Acknowledgments

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 2010–2011 project "GenData 2020" (2010RTFWBH), and by Google under the Google Research Award program.

Appendix. Proofs of theorems

Theorem 5.1. *Given a logical index $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ and a delta version $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ of $\mathcal{T}$, the set $(\mathcal{T} \cup \Delta_i) \setminus (\{\langle id, V, P\rangle \in \mathcal{T}\colon\ id \in \mathcal{ID}_i\} \cup \{\langle id, V, T\rangle \in \mathcal{T}\colon\ id \in \mathcal{ID}_i\})$ of logical nodes represents logical index $\mathcal{T}' = \mathcal{T} \oplus \Delta_i$.*

Proof. We first note that by Definition 3.2, $n^a \in \Delta_i^a$ iff $\phi(n^a) \in \mathcal{ID}_i$. Therefore, in the set $(\mathcal{T} \cup \Delta_i) \setminus (\{\langle id, V, P\rangle \in \mathcal{T}\colon\ id \in \mathcal{ID}_i\} \cup \{\langle id, V, T\rangle \in \mathcal{T}\colon\ id \in \mathcal{ID}_i\})$ of logical nodes there is not an abstract node represented by two or more logical nodes and there is not a logical identifier repeated in two or more logical nodes. By Definition 3.3, $\mathcal{T}' = \mathcal{T} \oplus \Delta_i = (\mathcal{T}^a, \mathcal{ID}, \phi_i)$. The correct allocation for all the nodes $n^a \in (\mathcal{T}^a \setminus \Delta_i^a)$ in $\mathcal{T}'$ is $id = \phi(n^a)$. Therefore, if n^a is a leaf, logical node $\langle id, V, T\rangle \in \mathcal{T}$ correctly represents n^a. If n^a is an internal node, all the children of n^a do not belong to Δ_i (Definition 3.2) and logical node $\langle id, V, P\rangle \in \mathcal{T}$ correctly represents n^a. Analogously, the correct allocation for all the nodes $n^a \in \Delta_i^a$ in $\mathcal{T}'$ is $id = \phi_i(n^a)$. Therefore, if n^a is a leaf, the corresponding logical node correctly represents n^a. If n^a is an internal node, its children either belong to Δ_i or to the main index. Since pointers to children are defined according to ϕ_i, the logical node $\langle id, V, P\rangle \in \Delta_i$ storing n^a correctly represents n^a. The structure is then consistent. □

Theorem 6.1. *Given a logical index $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$, a set $\{\Delta_1, \dots, \Delta_n\}$ of delta versions of $\mathcal{T}$, and $\Delta_r = (\Delta_r^a, \mathcal{ID}_r, \phi_r)$ a reconciled delta version of $\{\Delta_1, \dots, \Delta_n\}$ with $\mathcal{T}$, the logical index $\mathcal{T}' = (\mathcal{T}^a, \mathcal{ID}, \phi_\rho)$ computed by Algorithm 2 represents $\mathcal{T} \oplus \Delta_\rho$, where $\Delta_\rho = (\Delta_\rho^a, \mathcal{ID}_\rho, \phi_\rho)$ with $\Delta_\rho^a = \Delta_r^a$, $\mathcal{ID}_\rho = \mathcal{ID}_r$, and $\phi_\rho(n^a) = \phi(n^a)$ if $n^a \notin \Delta_\rho^a$.*

Proof. The proof of the theorem is based on the following observations. Given a logical index $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ and a delta version $\Delta_i = (\Delta_i^a, \mathcal{ID}_i, \phi_i)$ of $\mathcal{T}$:

(1) the parent of each node in Δ_i^a belongs to Δ_i^a: $n_x^a \in \Delta_i^a \Rightarrow n_y^a = \langle Values_y, Children_y\rangle \in \Delta_i^a$ s.t. $n_x^a \in Children_y$ (by Definition 3.2);

(2) if node n^a belongs to delta version Δ_i, the logical identifier $\phi(n^a)$ associated with n^a in the main shuffle index belongs to Δ_i: $n^a \in \Delta_i^a \Leftrightarrow \phi(n^a) \in \mathcal{ID}_i$ (by Definition 3.2);

(3) if a node n^a and its identifier $\phi_i(n^a)$ in Δ_i belong to *ToShuffle*$_i$[l], also node n^a_j allocated to $\phi(n^a)$ in Δ_i belongs to *ToShuffle*$_i$[l]: $\langle n^a, \phi_i(n^a)\rangle \in$ *ToShuffle*$_i$[l] $\Leftrightarrow \exists n^a_j \in \mathcal{T}^a$ s.t. $\langle n^a_j, \phi(n^a)\rangle \in$ *ToShuffle*$_i$[l], $i = 1, \ldots, n$, $l = 0, \ldots, h$ (the same relationship holds for *ToAdjust*$_i$[l] and *Unchanged*$_i$[l]).

To prove that $\mathcal{T}'$ is a logical index resulting from $\mathcal{T} \oplus \Delta_\rho$, we first need to prove that $\mathcal{T}'$ is a correct logical representation of $\mathcal{T}^a$ where nodes are allocated to blocks according to ϕ_ρ. To this purpose, we consider separately the blocks in *ToShuffle*$_i$[l], *ToAdjust*$_i$[l], *Unchanged*$_i$[l], and the blocks that do not belong to any delta version.

- *Blocks that do not belong to any delta version*. These nodes/blocks are not modified by the algorithm. Given a node $n^a \in \mathcal{T}^a$ such that $n^a \notin \Delta^a_i$, $i = 1, \ldots, n$, by observation 2 above, also $\phi(n^a) \notin \mathcal{ID}_i$, $i = 1, \ldots, n$. Furthermore, if n^a is an internal node, its children *Children*[j], $j = 1, \ldots, q + 1$ (and the blocks to which they are allocated in $\mathcal{T}$) do not belong to Δ^a_i, $i = 1, \ldots, n$. Then, the subtree rooted at $\langle id, V, P\rangle$ ($\langle id, V, T\rangle$, respectively) representing node n^a in $\mathcal{T}$ is consistent and correctly represents the result of the reconciliation.
- *Unchanged*$_i$[l]. These blocks are written on the main index, directly from the delta version without the client's intervention. Given a node/block allocation $\langle n^a, \phi_i(n^a)\rangle$ that belongs to *Unchanged*$_i$[l], $\nexists \Delta_j$, $i \neq j$, such that $n^a \in \Delta^a_j$ or $\phi(n^a) \in \mathcal{ID}_j$, since otherwise $\langle n^a, \phi_i(n^a)\rangle$ and $\langle n^a_i, \phi(n^a)\rangle$ would belong to *ToShuffle*$_i$[l]. This observation also holds for all the children of the node (if n^a is an internal node), since otherwise $\langle n^a, \phi_i(n^a)\rangle$ would belong to *ToAdjust*$_i$[l]. If n^a is an internal node, its children are stored in blocks that do not belong to any delta version, or that belong to *Unchanged*$_i$[$l + 1$] or to *ToAdjust*$_i$[$l + 1$]. Since all these blocks are not re-allocated by the algorithm, the block $\langle id, V, P\rangle$ representing n^a in Δ_i correctly refers to the blocks storing the children of n^a and therefore correctly represents the result of the reconciliation. Analogously, if n^a is a leaf node, block $\langle id, V, T\rangle$ representing n^a in Δ_i correctly represents the result of the reconciliation.
- *ToAdjust*$_i$[l]. These blocks are downloaded by the client, which updates the pointers according to the reconciled blocks at level $l + 1$ if $l < h$, and writes them on the main index without changing their allocation. Given a node/block allocation $\langle n^a, \phi_i(n^a)\rangle$ that belongs to *ToAdjust*$_i$[l], $\nexists \Delta_j$, $i \neq j$, such that $n^a \in \Delta^a_j$ or $\phi(n^a) \in \mathcal{ID}_j$, since otherwise $\langle n^a, \phi_i(n^a)\rangle$ and $\langle n^a_i, \phi(n^a)\rangle$ would belong to *ToShuffle*$_i$[l]. However, if n^a is an internal node, its children may belong to *ToShuffle*$_i$[$l + 1$]. Hence, the pointers in the logical node $\langle id,V,P\rangle$ representing n^a in Δ_i must be updated according to the reconciliation of the nodes at level $l + 1$. More precisely, if $P[j] \notin \mathcal{ID}_i$, then *Children*[$j$] of n^a is a node that does not belong to any delta version and therefore its allocation has not been modified. Consequently, $P[j]$ remains unchanged. If $P[j] \in \mathcal{ID}_i$ then *Children*[j] of n^a belongs to Δ^a_i. If $\langle Children[j], P[j]\rangle$ either belongs to *Unchanged*$_i$[$l + 1$] or to *ToAdjust*$_i$[$l + 1$], then $P[j]$ refers to the correct block in $\mathcal{T}'$ and therefore it remains unchanged. Otherwise, if $\langle Children[j], P[j]\rangle$ be-

longs to $\mathit{ToShuffle}_i[l+1]$, then $\mathit{Children}[j]$ has been allocated to a different identifier. Since nodes at level $l+1$ have already been reconciled with the main index, $P[j]$ can easily be updated to the identifier of the node in $\mathcal{T}'$ storing $\mathit{Children}[j]$ (i.e., $\phi_\rho(\mathit{Children}[j])$). If n^a is a leaf node, block $\langle id, V, T\rangle$ representing n^a in Δ_i already correctly represents the result of the reconciliation.

- $\mathit{ToShuffle}_i[l]$. These blocks are downloaded by the client, reconciled with $\mathit{ToShuffle}_1[l], \ldots, \mathit{ToShuffle}_n[l]$, shuffled, and written on the main index. We note that a node/block allocation $\langle n^a, \phi_i(n^a)\rangle$ belongs to $\mathit{ToShuffle}_i[l]$ in three cases: (i) $n^a \in \Delta_j$, $i \neq j$; (ii) $\phi_i(n^a) \in \mathcal{ID}_j$, $i \neq j$; or (iii) it is a cover node. In the first case, the algorithm combines all the blocks storing n^a in all the delta versions, obtaining one block that is written in the main index. If n^a is an internal node, block $\langle id, V, P\rangle$ computed by the algorithm is such that $id = \phi_\rho(n^a)$, $V = \mathit{Values}$, and the pointers to children are obtained as described for the blocks in $\mathit{ToAdjust}_i[l]$, considering the reallocation defined when reconciling nodes at level $l+1$. If n^a is a leaf, block $\langle id, V, T\rangle$ computed by the algorithm is such that $id = \phi_\rho(n^a)$, $V = \mathit{Values}$ and $T = \mathit{Tuples}$. In the second and third cases, the logical node storing $\langle n^a, \phi_i(n^a)\rangle$ is reallocated to $\phi_\rho(n^a)$ (i.e., $id = \phi_\rho(n^a)$) and, if n^a is an internal node, the pointers to children are updated as described for the blocks in $\mathit{ToAdjust}_i[l]$.

We note that $\mathcal{T}'$ does not have replicated nodes, since $\mathcal{T}^a, \Delta_1^a, \ldots, \Delta_n^a$ do not include replicas. Also, if a node n^a belongs to two (or more) delta versions, $\langle n^a, \phi_i(n^a)\rangle$ belongs to $\mathit{ToShuffle}_i[l]$ for all delta versions Δ_i such that $n^a \in \Delta_i$. Therefore, only one copy of these nodes is allocated to a block in $\mathcal{T}'$. We conclude that $\mathcal{T}'$ is a correct logical representation of $\mathcal{T}^a$. Also, $\phi_\rho(n^a) = \phi(n^a)$ if $n^a \notin \Delta_\rho^a$, since the blocks in $\mathcal{T} \setminus \Delta_\rho$ are not modified by the algorithm. Since all the blocks in $\mathit{ToShuffle}_1[l], \ldots, \mathit{ToShuffle}_n[l]$ are shuffled, $l = 0, \ldots, h$, and $\mathit{ToShuffle}_i[l]$ includes all the blocks in $\mathcal{C}_i$ at level l, the algorithm shuffles all the blocks/nodes with conflicting allocation. □

Theorem 8.1. *When the server detects m conflicts over a single physical block in m versions, the probability that any pair of them is due to accesses to the same logical node as a target is $\frac{1}{(num\_cover+1)^2}$. The probability that all of them are due to accesses to the same logical node as a target is $\frac{1}{(num\_cover+1)^m}$.*

Proof. As it was shown in [9], the shuffle index relies on the indistinguishability hypothesis, which guarantees that accesses to covers are not distinguishable from accesses to targets. Experimental support is provided in [9] that guarantees that covers can follow the same statistical profile as targets. We can then see that each logical access is indistinguishable from both the target and any of the *num_cover* covers. Having 1 target and *num_cover* covers, each physical access will have a probability equal to $\frac{1}{(num\_cover+1)}$ of being associated with the logical node representing the target and a probability equal to $\frac{num\_cover}{(num\_cover+1)}$ of being associated with a logical node representing a cover. Based on the indistinguishability hypothesis, the probabilities will be independent. Then, the probability that two accesses to the same block correspond to two accesses to the node as a target will be the product of them, that is, $p = \frac{1}{(\mathit{num\_cover}+1)^2}$. The same reasoning applied over all the conflicts in the m delta versions permits to show that the probability that all m accesses refer to the same target is $p = \frac{1}{(\mathit{num\_cover}+1)^m}$. □

Theorem 8.2. *Let $\mathcal{T} = (\mathcal{T}^a, \mathcal{ID}, \phi)$ be a logical index, $\{\Delta_1, \ldots, \Delta_n\}$ be a set of delta versions of $\mathcal{T}$, $\mathcal{T}'$ be the logical index resulting from reconciling $\{\Delta_1, \ldots, \Delta_n\}$ and $\mathcal{T}$ through Algorithm* 2, *and $n^a \in (\Delta_i \cap \Delta_j)$, $i, j = 1, \ldots, n$, $i \neq j$. The server has probability lower than $\frac{1}{\mathit{num\_cover}+1}$ to identify the block where n^a is allocated.*

Proof. Since $n^a \in (\Delta_i \cap \Delta_j)$, $\langle n^a, \phi_i(n^a)\rangle \in$ *ToShuffle*$_i$[*l*] and $\langle n^a, \phi_j(n^a)\rangle \in$ *ToShuffle*$_j$[*l*]. Since we include cover nodes, *ToShuffle*$_k$[*l*], $k = 1, \ldots, n$, includes at least *num_cover* + 1 pairs. As a consequence, in the worst case, there are at least *num_cover* + 1 nodes that must be reallocated and *num_cover* + 1 possible allocations. Since nodes/blocks in *ToShuffle*$_1$[*l*] $\cup \cdots \cup$ *ToShuffle*$_n$[*l*] are shuffled by the algorithm, n^a has the same probability of being allocated by the shuffling to any of the (at least *num_cover* + 1) available blocks. □

References

[1] R. Agrawal, J. Kierman, R. Srikant and Y. Xu, Order preserving encryption for numeric data, in: *Proc. of ACM SIGMOD 2004*, Paris, France, June 2004.

[2] A. Ambainis, Upper bound on communication complexity of private information retrieval, in: *Proc. of ICALP 1997*, Bologna, Italy, July 1997.

[3] C. Cachin, S. Micali and M. Stadler, Computationally private information retrieval with polylogarithmic communication, in: *Proc. of EUROCRYPT 1999*, Prague, Czech Republic, May 1999.

[4] A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi and P. Samarati, Modeling and assessing inference exposure in encrypted databases, *ACM TISSEC* **8**(1) (2005), 119–152.

[5] B. Chor and N. Gilboa, Computationally private information retrieval (extended abstract), in: *Proc. of STOC 1997*, El Paso, TX, USA, May 1997.

[6] B. Chor, E. Kushilevitz, O. Goldreich and M. Sudan, Private information retrieval, *JACM* **45**(6) (1998), 965–981.

[7] S. Cimato, M. Gamassi, V. Piuri, R. Sassi and F. Scotti, Privacy-aware biometrics: Design and implementation of a multimodal verification system, in: *Proc. of ACSAC 2008*, Anaheim, CA, USA, December 2008.

[8] E. Damiani, S. De Capitani Vimercati, S. Jajodia, S. Paraboschi and P. Samarati, Balancing confidentiality and efficiency in untrusted relational DBMSs, in: *Proc. of CCS 2003*, Washington, DC, USA, October 2003.

[9] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Efficient and private access to outsourced data, in: *Proc. of ICDCS 2011*, Minneapolis, MN, USA, June 2011.

[10] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi and P. Samarati, Supporting concurrency in private data outsourcing, in: *Proc. of ESORICS 2011*, Leuven, Belgium, September 2011.

[11] S. De Capitani di Vimercati, S. Foresti and P. Samarati, Managing and accessing data in the cloud: Privacy risks and approaches, in: *Proc. of CRiSIS 2012*, Cork, Ireland, October 2012.

[12] E. De Cristofaro, Y. Lu and G. Tsudik, Efficient techniques for privacy-preserving sharing of sensitive information, in: *Proc. of TRUST 2011*, Pittsburgh, PA, USA, June 2011.

[13] X. Ding, Y. Yang and R.H. Deng, Database access pattern protection without full-shuffles, *IEEE TIFS* **6**(1) (2011), 189–201.

[14] M. Gamassi, V. Piuri, D. Sana and F. Scotti, Robust fingerprint detection for access control, in: *Proc. of RoboCare 2005*, Rome, Italy, May 2005.

[15] O. Goldreich and R. Ostrovsky, Software protection and simulation on oblivious RAMs, *JACM* **43**(3) (1996), 431–473.

[16] H. Hacigümüs, B. Iyer and S. Mehrotra, Providing database as a service, in: *Proc. of ICDE 2002*, San Jose, CA, USA, February 2002.

[17] H. Hacigümüs, B. Iyer, S. Mehrotra and C. Li, Executing SQL over encrypted data in the database-service-provider model, in: *Proc. of SIGMOD 2002*, Madison, WI, USA, June 2002.

[18] E.D. Lazowska, J. Zahorjan, G.S. Graham and K.C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[19] P. Lin and K.S. Candan, Hiding traversal of tree structured data from untrusted data stores, in: *Proc. of WOSIS 2004*, Porto, Portugal, April 2004.

[20] P. Lin and K.S. Candan, Secure and privacy preserving outsourcing of tree structured data, in: *Proc. of SDM 2004*, Toronto, Canada, August 2004.

[21] Y. Lu and G. Tsudik, Privacy-preserving cloud database querying, *JISIS* **1**(4) (2011), 5–25.

[22] F. Olumofin and I. Goldberg, Privacy-preserving queries over relational databases, in: *Proc. of PETS 2010*, Berlin, Germany, July 2010.

[23] P. Samarati and S. De Capitani di Vimercati, Data protection in outsourcing scenarios: Issues and directions, in: *Proc. of ASIACCS 2010*, Beijing, China, April 2010.

[24] E. Shmueli, R. Waisenberg, Y. Elovici and E. Gudes, Designing secure indexes for encrypted databases, in: *Proc. of IFIP DBSec 2005*, Storrs, CT, USA, August 2005.

[25] R. Sion and B. Carbunar, On the computational practicality of private information retrieval, in: *Proc. of NDSS 2007*, San Diego, CA, USA, February/March 2007.

[26] H. Wang and L.V.S. Lakshmanan, Efficient secure query evaluation over encrypted XML databases, in: *Proc. of VLDB 2006*, Seoul, Korea, September 2006.

[27] S. Wang, D. Agrawal and A. El Abbadi, A comprehensive framework for secure query processing on relational data in the cloud, in: *Proc. of SDM 2011*, Seattle, WA, USA, September 2011.

[28] P. Williams, R. Sion and B. Carbunar, Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage, in: *Proc of CCS 2008*, Alexandria, VA, USA, October 2008.

[29] K. Yang, J. Zhang, W. Zhang and D. Qiao, A light-weight solution to preservation of access pattern privacy in un-trusted clouds, in: *Proc. of ESORICS 2011*, Leuven, Belgium, September 2011.