The VLDB Journal manuscript No.
(will be inserted by the editor)

Kyriakos Mouratidis · Dimitris Sacharidis · HweeHwa Pang

Partially Materialized Digest Scheme: An Efficient Verification Method for Outsourced Databases

Received: date / Accepted: date

Abstract In the outsourced database model, a data owner publishes her database through a third-party server; i.e., the server hosts the data and answers user queries on behalf of the owner. Since the server may not be trusted, or may be compromised, users need a means to verify that answers received are both *authentic* and *complete*, i.e., that the returned data have not been tampered with, and that no qualifying results have been omitted. We propose a result verification approach for one-dimensional queries, called *Partially Materialized Digest scheme* (PMD), that applies to both static and dynamic databases. PMD uses separate indexes for the data and for their associated verification information, and only partially materializes the latter. In contrast with previous work, PMD avoids unnecessary costs when processing queries that do not request verification, achieving the performance of an ordinary index (e.g., a B^+-tree). On the other hand, when an authenticity and completeness proof is required, PMD outperforms the existing state-of-the-art technique by a wide margin, as we demonstrate analytically and experimentally.

Furthermore, we design two verification methods for spatial queries. The first, termed *Merkle R-tree* (MR-tree), extends the conventional approach of embedding authentication information into the data index (i.e., an R-tree). The second, called *Partially Materialized KD-tree* (PMKD), follows the PMD paradigm using separate data and verification indexes. An empirical evaluation with real data shows that the PMD methodology is superior to the traditional approach for spatial queries too.

Kyriakos Mouratidis
Singapore Management University
E-mail: kyriakos@smu.edu.sg

Dimitris Sacharidis
National Technical University of Athens
E-mail: dsachar@dblab.ntua.gr

HweeHwa Pang
Singapore Management University
E-mail: hhpang@smu.edu.sg

1 Introduction

The outsourced database model [13] includes three main entities: (i) the *data owner*, (ii) the *third-party publisher* (i.e., the server), and (iii) the *users*. The owner uploads her data to the server, along with their associated index structures. The users can then pose queries to the server. Since the latter may not be trusted, or may be compromised, it is essential to enable verification of the results by the users. In particular, the server should be able to guarantee that the returned data are both *authentic* and *complete*. Authenticity implies that the result data tuples indeed exist in the owner's database, and that they have not been tampered with in any way. Completeness requires that no qualifying tuples have been omitted by the server. We term the above problem authenticity and completeness (AC) verification in outsourced databases.

The general approach taken requires that the owner uploads authentication information together with her data. Posed a query, the server returns, in addition to the result, pieces of this information in a *verification object* (VO). The user may then use the VO to determine whether the server's answer is authentic and complete. The performance factors involved in this general scheme are (i) construction time for the index and the verification structures, (ii) communication cost to upload the database to the server, (iii) storage overhead at the server, (iv) query processing and VO computation time, (v) communication overhead for sending the results and the VO to the users, and (vi) computation cost for AC verification by the users.

Existing methods fall into two categories. The first includes signature-based techniques, where the owner produces one signature for every tuple in the database, e.g., [26, 24]. Approaches of the second category use a Merkle hash tree (MHT, reviewed in Section 2.1) for VO computation, e.g., [9, 27]. The MHT is embedded into the data index and the VO is created during query processing. Recently, [17] advocated the use of MHT-based schemes, showing that signature-based methods are very costly,

due to the large storage and computation overheads incurred by the numerous signatures.

We propose a novel method, called *Partially Materialized Digest scheme* (PMD), that belongs to the second category. As opposed to existing approaches, PMD does not incorporate the MHT into the data index. Instead, it uses a *main index* (MI) solely for storing and querying the data, and a separate *digest index* (DI) that contains the MHT-based verification information in a compressed form. Our decision to decouple DI from MI is motivated by the following facts.

First, in a real-world scenario not all users require AC guarantees; e.g., some time-critical applications may favor a fast response over a verified one. In such situations, PMD uses directly the MI, achieving the performance of a standard, non-authenticated index. Second, in case an AC proof is requested, query processing and VO computation impose conflicting requirements on the index. For instance, a high fanout is desirable in order to reduce the query evaluation cost, but that leads to a large VO [19]. Employing a separate DI provides flexibility in designing its structure and optimizing its performance; we use a binary MHT (to keep the VO small) and materialize only a subset of its nodes in order to decrease the DI size and the I/O cost for VO generation. Based on the above design principles, PMD significantly outperforms the current state-of-the-art verification technique [17], reducing drastically the query response time, the storage overhead and the index construction cost.

In addition to performance advantages, PMD provides additional flexibilities that are essential to database outsourcing applications. Specifically, in many cases a database is already outsourced and deployed when the owner decides to authenticate it. With our technique, the owner needs only to compute and upload the DI, without having to authenticate and re-upload the entire database. Moreover, PMD provides architectural flexibility, since the MI may be hosted by a server responsible only for query processing, and the DI by another server that is dedicated to providing AC guarantees (i.e., producing the VO).

The basic PMD method is designed for verification of one-dimensional queries in static databases. The *dynamic PMD* (dPMD) adapts the PMD approach to dynamic datasets by using an alternative DI materialization technique. We analyze the performance of both methods, and verify experimentally their superiority over existing approaches for static and dynamic data. As an additional contribution, we extend our study to higher dimensions, and design comprehensive methods for verification in spatial databases. The first, termed *Merkle R-tree* (MR-tree), incorporates an MHT into an R-tree [11] index. The second, called *Partially Materialized KD-tree* (PMKD), applies a space partitioning strategy and extends the PMD concept to spatial queries. We evaluate the performance of the MR-tree and PMKD on real data, and demonstrate that the PMD methodology retains its benefits in the spatial domain.

To summarize, in this paper we propose the PMD methodology for database verification. Its advantages over existing methods are as follows:

- PMD achieves significant performance improvements in terms of query processing, space requirements and index construction cost over the current state-of-the-art.
- PMD separates the query processing and the VO computation tasks. This provides users with the option to receive a non-authenticated answer, without any performance degradation due to verification structures. Additionally, existing databases can be directly authenticated, without having to be modified and re-uploaded. Also, query answering and VO generation can be delegated to different servers.
- PMD applies to static and dynamic databases, and to one-dimensional and spatial data. In all cases, it retains its advantages over the conventional approach of combining the data and verification indexes.

The rest of the paper is organized as follows. Section 2 provides a background on cryptographic primitives and spatial indexes. Section 3 surveys previous verification methods. Sections 4 and 5 describe our technique for static one-dimensional data (PMD), and its adaptation to dynamic settings (dPMD), respectively. Section 6 presents the MR-tree and PMKD methods for AC verification in spatial databases. Section 7 experimentally evaluates our techniques, and Section 8 concludes the paper.

2 Background

In this section, we review some cryptographic essentials, as well as some basics of spatial access methods.

2.1 Cryptographic Primitives

Collision-resistant hash functions. A hash function $\mathcal{H}$ maps a message m of arbitrary size to a fixed-length bit vector $\mathcal{H}(m)$. The collision-resistance property guarantees that it is computationally infeasible to find two different messages that map into the same hash value. Additionally, a desirable property is that $\mathcal{H}(m)$ is fast to compute. The most commonly used hash function is SHA1 [25] with an 160-bit output, which satisfies both requirements. We refer to $\mathcal{H}(m)$ as the hash, hash value or digest of m.

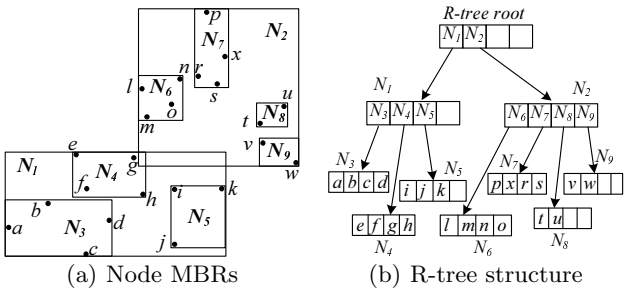
Public-key encryption. In public-key cryptography, a message owner creates a pair of *private* and *public* keys. She keeps the private key secret and publishes the public one. This allows the integrity and ownership of a message m to be authenticated as follows. The owner *signs* m with her private key. The recipients of m can verify

the signature using the public key to determine whether m has been tampered with. The most widely used algorithm for public-key encryption is RSA [29] with a signature length of 128 bytes.

Merkle hash tree. The Merkle hash tree (MHT) is a method for authenticating a set of messages (e.g., data tuples) collectively, without signing each one individually [20]. It is a binary tree over the digests of the messages, where each internal node equals the hash of the concatenation of its two children. The owner signs the root of the tree with her private key. Given a message and the sibling hashes to the path in MHT from the root to the message, one may verify its authenticity by reconstructing bottom-up the root digest of MHT, and checking whether it matches the owner's signature. The collision-resistance of the hash function guarantees that an adversary cannot modify/replace any message in a way that leads to an identical root digest. The MHT need not be binary, i.e., it can be a multi-way tree. However, the number of digests required for authentication is minimized at a fanout of 2. The MHT idea also applies to arbitrary DAG (directed acyclic graph) structures [19].

2.2 Spatial Indexes

Spatial indexes are used for efficient search among data in the Euclidean space. We survey two indexes, the R-tree and the KD-tree, because they are utilized by our techniques in Section 6. The R-tree and its variants [11, 2] are the most widely used spatial access methods. Figure 1 shows an R-tree for a dataset containing 23 objects, assuming a capacity of four entries per node. Objects that are close in space (e.g., l, m, n, o) are stored in the same leaf node (N_6). Nodes are then recursively grouped together in a similar fashion up to the top level, which consists of a single root. Each R-tree node is characterized by the minimum bounding rectangle (MBR) enclosing all the points in its subtree.



(a) Node MBRs (b) R-tree structure

Fig. 1 R-tree example

The KD-tree [3] is a space partitioning index. It is a binary tree, built top-down as follows. The root corresponds to the entire data-space. It is split according to the first dimension (e.g., an x value) into two halves (buckets) that contain an equal number of data. Each of these buckets, in turn, is partitioned into two halves according to the second dimension (e.g., a y value). The procedure continues recursively by cycling through splitting dimensions in successive tree levels, until each bucket contains one object only.

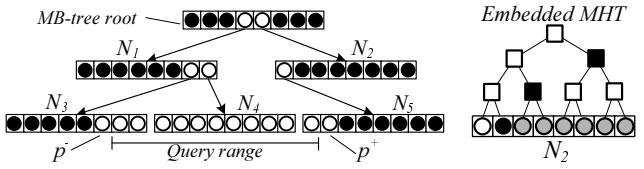
3 Related Work

Existing verification methods for outsourced databases follow two paradigms. The first is signature chaining [26, 24, 7]. Assuming that the data are ordered according to search attribute A, the owner hashes and signs every triple of consecutive tuples. Posed a range (selection) query on A, the server returns the qualifying data, along with (hashes of) the first tuple to the left and the first tuple to the right of the range. It then includes the corresponding signatures in the VO. The user inspects the result by verifying the signatures that "chain" consecutive tuples. The number of signatures sent (by the server) and verified (by the user) can be reduced by aggregating them into a single one [23]. Signature chaining approaches are shown to be particularly inefficient [17], because (i) generating the signatures incurs high computation cost for the owner, (ii) the large size of the signatures leads to excessive storage overhead for the server, and (iii) verifying multiple signatures (even aggregated ones) is expensive.

The second paradigm overcomes the above problems by utilizing an MHT for result verification. The MHT hashes are faster to compute and more concise than signatures, leading to shorter construction time and smaller storage overhead. [5, 6] extend the MHT idea to verify the authenticity of XML documents, while [4] to authenticate UDDI registries for publishing information about web services. [9, 10, 22] verify range selections over an ordered list of data tuples. [27] introduces a method that is suitable for database indexes, combining an MHT with a multi-way tree. [17] is the most recent work in this category, proposing two AC verification methods for one-dimensional queries over disk-resident data; the Merkle B-tree (MB-tree) and the Embedded Merkle B-tree (EMB-tree), the latter being the current state-of-the-art.

The MB-tree is a B$^+$-tree [8] where each internal node N_i additionally contains a hash for each child. The digest h_{N_i} of N_i is defined over (the concatenation of) the hashes of its children. h_{N_i} is stored in the parent node of N_i. The root digest is signed by the owner. Figure 2(a) shows an MB-tree with fanout $f = 8$ (omitting nodes irrelevant to our example). Posed a range query, the server returns, in addition to qualifying objects, two *boundary* ones, p^- and p^+, falling immediately to the left and to the right of the range. The reported objects are shown hollow in the leaf level. The VO contains all

left (right) sibling hashes to the path of p^- (p^+), i.e., the VO includes the digests of the solid entries. Upon receipt of the result, the user calculates the hashes of the returned (hollow) objects, and combines them with the solid hashes to derive, initially, the internal hollow hashes and, eventually, the root digest. If the latter matches the owner's signature, the result is complete and authentic, since (i) all objects between p^- and p^+ have been returned and are authentic (due to the collision-resistance of h), and (ii) the interval between the keys of p^- and p^+ completely covers the query range.



(a) MB-tree (b) Emb. MHT

Fig. 2 MB- and EMB-tree example

The second method proposed in [17] is the EMB-tree. It is motivated by the fact that the VO of the MB-tree is too large, containing up to $f-1$ digests for each node in the path toward a boundary point. In Figure 2(a), for instance, the MB-tree inserts 7 digests of N_2 in the VO. To remedy the problem, the EMB-tree embeds a binary MHT within each node, and includes in the VO only $\log f$ hashes per node. Figure 2(b) shows the embedded MHT for N_2. The VO contains the solid hashes, and skips the shaded ones. As noted in [17], the best EMB-tree variant is the EMB$^-$-tree. The EMB$^-$-tree does not explicitly store the embedded trees, but derives them on the fly during VO computation, e.g., in Figure 2(b), N_2 only stores the 8 digests of its children (shown circular) and skips the square hashes. In the following, we refer to the EMB$^-$-tree as EMB-tree for simplicity.

The EMB-tree has significantly smaller VO than the MB-tree, and similar size and query performance. Our technique, PMD, has even smaller VO than EMB-tree. The most important advantage of PMD, however, is that it incurs only a fraction of the query response time, the storage overhead, and the construction cost of the MB-/EMB-tree. Furthermore, our approach provides the processing and architectural flexibilities discussed in Section 1.

Regarding multi-dimensional queries, [10,19] propose methods for range verification, by incorporating MHTs into multi-dimensional range trees [3]. These techniques focus on memory-resident data; range trees incur high space overhead, and are not suitable for disk-based indexing. I/O efficiency is taken into account in adaptations of these approaches, which however process only special types of range queries, and are based on theoretical indexes. On the other hand, our spatial methods (in Section 6) target explicitly disk-resident data, build upon commonly used indexes, and process general range and nearest neighbor queries.

Another stream of related work focuses on probabilistic result verification, where tampering with the query results can/cannot be detected with a certain probability. [32] proposes that the owner inserts fake tuples in the outsourced database and shares these tuples with the users. Upon receipt of an answer from the server, the querying user searches in the result for the fake tuples that should be returned. If none of these tuples is missing, then the answer is considered legitimate. This approach requires that the users are trusted (an assumption we do not make) and, more importantly, there is a significant possibility that tampering is not detected (e.g., if the server modifies some real tuple but no fake one, or if no fake tuple satisfies the query). In [31] the user stores locally some (pre-computed) query results and identifies tampering if some tuples falling in the intersection of the pre-computed and the posed queries are missing from the server's responses. Similar to [32], tampering is not always detected.

In addition to result verification methods, research on outsourced databases has also considered other data security problems. For example, [12] proposes techniques that allow SQL query processing over encrypted data. [30,21] study security in conjunction with access control, while [1,15] address privacy-preservation issues. The above techniques consider related, yet different problems from ours, and provide complementary methods for securing outsourced databases.

4 Static One-dimensional Data

In this section we present the Partially Materialized Digest scheme (PMD), our AC verification method for static data. We analyze its performance and compare it qualitatively with existing methods. Table 1 summarizes the primary symbols used, along with their interpretation.

Symbol	Description
$\mathcal{S}(m)$	owner's signature on message m
$\mathcal{H}(m)$	digest of message m
N	data cardinality
f	fanout of the MI
n	number of external nodes in the MI
E_i	i-th external node of the MI
H_i	root digest of E_i's lower tree
d_{MI}, d_{DI}, d_u, d_l	height of MI, DI, upper tree, lower trees
S	query result (including boundary objects)
B	block size
$\|h\|$	size of hash value
$\|s\|$	size of signature
$\|num\|$	size of pointer, integer or real number
$C_{I/O}, C_h, C_s, C_v$	cost of I/O, $\mathcal{H}(m)$, $\mathcal{S}(m)$, $\mathcal{S}^{-1}(m)$ operation

Table 1 Primary symbols and functions

4.1 The PMD Approach

Let our database contain N objects of the form $\langle id, key \rangle$, where id is a unique object identifier and key is the indexed attribute. The main index (MI) is a standard B$^+$-tree on key, i.e., it does not store any verification information. Every external (i.e., lowest level) node E_i stores data objects, and a sibling pointer to the next external node E_{i+1}. The MI is used for query processing.

In this paper we consider equality and range selections. Equality selections are treated as a special case of range selections, so we focus on the latter. A range query requesting objects with key in interval $r = [a, b]$ is evaluated by descending the MI down to the first object with key larger than or equal to a, and then reporting all subsequent objects with keys up to b. If the search needs to proceed to the next data page (external node), we utilize the sibling pointers of the B$^+$-tree. The internal levels of MI are irrelevant to VO computation. The structure of the external level, however, affects the VO and the DI, as we describe next.

Digest Index and AC Verification. To prove authenticity and completeness, the server returns to the user, in addition to the objects inside the query range r, the two *boundary* objects, p^- and p^+, falling immediately to the left and to the right of r. AC can be proven if we show that (i) the objects between p^- and p^+ have not been tampered with, and (ii) all objects between p^- and p^+ are returned. To prove (i) and (ii), we compute a VO using the digest index (DI). *Our method guarantees that VO construction incurs a maximum of 2 disk accesses, and that the VO has near-minimal size.*

Conceptually, the DI is a composite tree. Letting n be the number of external nodes in MI, the DI consists of an *upper* MHT built on top of n *lower* ones. The DI is constructed as follows. For each external MI node E_i, we compute a (lower) MHT. In a second step, we build the (upper) MHT over the roots of the lower trees. The root of the upper tree is signed with the owner's private key. The upper and lower trees are binary. Thus, the DI is essentially an MHT with fanout 2. This is a desirable property, since the VO size is minimal for binary MHTs [19] (as discussed in Section 2.1). The DI, however, is usually not a perfect binary tree, leading to a near-minimal VO size.

Figure 3 illustrates the DI in a scenario where the database contains $N = 16$ objects (p_1 to p_{16}). Assuming that each MI leaf contains 4 objects, the MI has $n = 4$ external nodes. For each of them, we compute a lower tree on the objects $p_i = \langle id_i, key_i \rangle$ therein. In the figure, $h_{i|j} = \mathcal{H}(\mathcal{H}(p_i)|\mathcal{H}(p_j))$ (where | indicates concatenation) and H_i is the root of the i-th lower tree. The upper tree is built over all the H_i values; $H_{i|j} = \mathcal{H}(H_i|H_j)$ and $\mathcal{S}(H_{1|2|3|4})$ is its signed root.

The DI works like an ordinary MHT. Consider the paths that lead to the left and to the right boundary objects, p^- and p^+. The VO for a range query contains (i) the signed DI root, (ii) all left sibling hashes to the path of p^-, and (iii) all right sibling hashes to the path of p^+. Upon receipt of the query result S, the user combines it with VO components (ii) and (iii), to reconstruct the (missing) part of the DI between the paths of p^- and p^+. Then, she verifies with the owner's public key whether the root of DI (i.e., component (i) of the VO) matches the locally computed root hash. If they match, S is deemed both complete and authentic; the collision-resistance of the hash function ensures that it is computationally infeasible for the server to tamper with the result and yet manage to produce hashes that match the original ones.

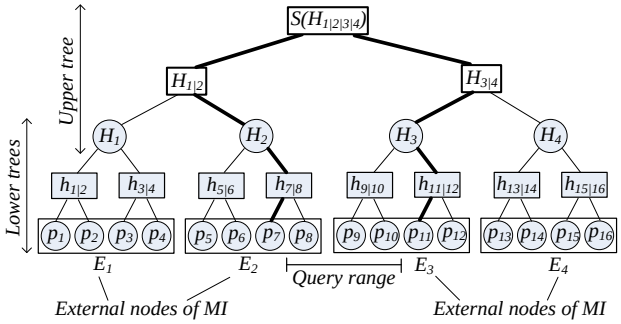


Fig. 3 Digest index example

Returning to our example, consider the range query in Figure 3. The query result is $\{p_8, p_9, p_{10}\}$, and the boundary objects are $p^- = p_7$ and $p^+ = p_{11}$. The DI paths leading to p^- and p^+ are shown bold. The server returns to the user objects p_7 to p_{11}, together with the VO. Component (i) of the VO is the signed root, $\mathcal{S}(H_{1|2|3|4})$. Component (ii) contains hash values H_1 and $h_{5|6}$, while (iii) consists of H_4 and $\mathcal{H}(p_{12})$. During AC verification, the user concatenates $h_{5|6}$ and $\mathcal{H}(\mathcal{H}(p_7)|\mathcal{H}(p_8))$ to calculate[1] H_2. Then, she appends H_2 to H_1, to derive $H_{1|2}$. Similarly, she computes $h_{9|10}$, $h_{11|12}$ and then H_3 and $H_{3|4}$. Finally, she calculates $\mathcal{H}(H_{1|2}|H_{3|4})$ and verifies whether it was signed by the data owner (using component (i) of the VO and the owner's public key).

We note here that the general idea to create a VO using the left and right sibling hashes of p^- and p^+ was first introduced in [9] (focusing on in-memory data), and has been used by subsequent research, including the MB-/EMB-tree. Verification in PMD follows the same paradigm, but introduces the aforementioned DI decomposition into upper tree and lower trees. This approach is suited to disk-resident data; it allows for an effective DI compression/materialization, which leads to low space consumption and fast VO computation, as we describe next.

Digest Index Compression. To reduce the storage overhead at the server and the I/O cost for VO computation, we do not materialize the entire DI structure.

[1] Note that the user knows objects p_i for $i = 7...11$, and she can locally compute the corresponding $\mathcal{H}(p_i)$.

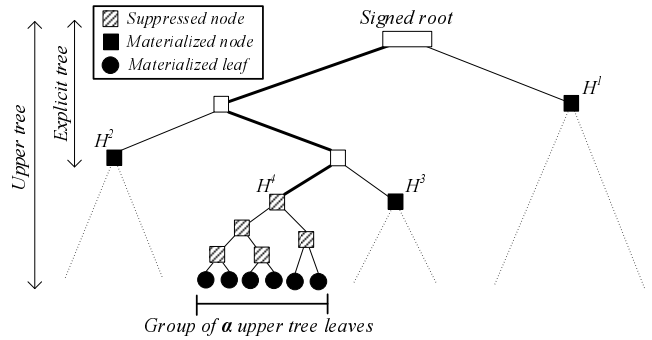


Fig. 4 Upper tree compression

Based on the fact that a page access is three orders of magnitude more expensive than a hashing operation [17, 26], we do not store the lower trees. Instead, when we access an external MI node (during query processing) and require values from the corresponding lower tree of the DI (to form the VO), we compute them on the fly. In the example of Figure 3, for instance, query processing loads the external node E_2 of MI to access p_7 and p_8. Since p_5 and p_6 are also necessarily read, the server computes $h_{5|6}$ and inserts it into the VO.

Following the same principle as above, we do not store the entire upper tree. Assuming that a disk page fits 4 hashes (i.e., $B = 4 \cdot |h|$) in Figure 3, we materialize only H_1, H_2, H_3, and H_4, and place them in a single page; this page constitutes the DI, and it is the only verification information stored in the server (in addition to the signed root). If $H_{3|4}$, for instance, is part of the VO for some query, then it is computed on the fly as $\mathcal{H}(H_3|H_4)$. In the general case, DI is still a flat structure, but it consists of multiple pages. Each DI block contains verification information for α consecutive MI leaves E_i (we discuss α's computation later).

In particular, we partition the n upper tree leaves (i.e., the n lower tree roots) into groups of α. For each group, we build a binary MHT (*suppressed tree*). In a second step, we construct one binary MHT (*explicit tree*) on top of all the suppressed ones, as shown in Figure 4. For every group (e.g., the group of the α leaves shown in the figure), we form one DI page by inserting (i) its α hashes (the solid circles), and (ii) the sibling digests to its path in the explicit tree (hashes H^1, H^2, H^3, shown as solid squares). In other words, the internal nodes of the suppressed tree (striped squares) are not materialized, but they are computed on the fly (using the α leaves/solid circles) when needed. Overall, the DI contains $\frac{n}{\alpha}$ disk pages.

The number of explicit nodes to be stored with a group (i.e., the cardinality of set (ii) above) equals the height of the explicit tree; this height is $\log \frac{n}{\alpha}$, since it is built over $\frac{n}{\alpha}$ suppressed tree roots. Each group and its corresponding explicit nodes should fit in a page, leading to the condition:

$$\left(\alpha + \log \frac{n}{\alpha}\right) \cdot |h| \leq B \ , \tag{1}$$

where B is the block size and $|h|$ is the size of a hash value. We set α to the largest integer that satisfies this condition, in order to minimize the size of DI. For all practical problem settings, $d_u \cdot |h| \ll B$, which means that α is always larger than 1 and that the verification information for an external MI node always fits in a single page[2]. Thus, we can compute VO components (ii) and (iii) for any range query with *at most 2 disk accesses*, one for each boundary object. If both p^- and p^+ are covered by the same suppressed tree, then only one page access is needed (for their common DI block).

A final remark concerns the linkage between MI and DI. Upon accessing the MI node E_i that contains a boundary data object, we need to read the DI page that stores verification information about E_i. To facilitate this operation, we keep in each external MI node a pointer to the corresponding DI block. The size and performance of the MI are practically unaffected by these extra pointers, since there are only n of them (where $n \ll N$), stored at its external level. If, however, we wish not to modify the structure of the MI at all (e.g., it is already deployed by the time the AC requirements are imposed), we need to maintain a directory. The directory contains a $\langle key, ptr \rangle$ pair for each DI block, where ptr is a pointer to the block, and key is the smallest key among the objects it represents. For most problem settings, the directory fits in a single page, and it could be kept either in main memory or on the disk.

4.2 Performance Analysis

This section analyzes the performance of PMD with respect to the cost factors listed in Section 1. To simplify the presentation, we assume that all MI nodes are full, and that all divisions have residual zero. We also assume that the sizes of a pointer, an integer and a real number are all equal to $|num|$.

In the MI, each internal node stores f pointers to its children and $f-1$ keys. Thus, the MI fanout is $f = \frac{B-|num|}{2 \cdot |num|} + 1$. Every external node contains $f-1$ data objects. Hence, MI has $n = \frac{N}{f-1}$ leaves, and height $d_{MI} = 1 + \log_f n \simeq \log_f N$. Assuming that the upper and lower trees of the DI are perfect binary trees, the height of the (conceptual) DI structure is $d_{DI} = \log N$. The heights of the upper and lower trees are $d_u = \log n$ and $d_l = \log(f-1)$, respectively.

Index construction time. Index construction includes computing and storing the MI and DI. The CPU time for

[2] For typical values $B = 1$ Kbyte and $|h| = 20$ bytes, n must be larger than 2^{50} for this statement to be false.

building the MI is negligible compared to the other involved factors and is ignored. The MI has $\sum_{i=0}^{i=d_{MI}-1} f^i = \frac{f^{d_{MI}}-1}{f-1}$ nodes, and storing it incurs $\frac{f^{d_{MI}}-1}{f-1} \cdot C_{I/O}$ cost. The conceptual DI structure has $2 \cdot N - 1$ nodes. Each node requires a digest computation. Thus, the CPU time to build the DI is $(2 \cdot N - 1) \cdot C_h + C_s$, where C_s is the cost of signing its root. To calculate the I/O for storing DI, let α be the maximum integer that satisfies Condition 1. The compressed DI occupies $\frac{n}{\alpha}$ disk pages. In total, the index construction cost is

$$\left(\frac{f^{d_{MI}}-1}{f-1} + \frac{n}{\alpha}\right) \cdot C_{I/O} + (2 \cdot N - 1) \cdot C_h + C_s \ . \quad (2)$$

Data uploading cost. The MI and DI could be either built at the server side or sent directly by the owner. In the first case, the owner needs to upload only the data objects and the signed root of DI, incurring a cost of

$$2 \cdot N \cdot |num| + |s| \ , \quad (3)$$

where $|s|$ is the size of the signed DI root. If the owner builds the MI and DI at her side and uploads them to the server, the communication cost equals the storage overhead and is given by Formula 4 below.

Space requirements. As previously discussed, the MI contains $\frac{f^{d_{MI}}-1}{f-1}$ nodes, and the DI consists of $\frac{n}{\alpha}$ disk pages. Hence, the storage overhead at the server is

$$\left(\frac{f^{d_{MI}}-1}{f-1} + \frac{n}{\alpha}\right) \cdot B + |s| \ . \quad (4)$$

Query processing and VO creation cost. Query processing time at the server breaks into I/O cost for the MI and DI, as well as CPU time for on-the-fly hash computations needed to build the VO. To evaluate a range query, we traverse MI down to p^-, incurring $d_{MI} - 1$ internal node accesses. Next, we serially load the external nodes to the right of p^- (following the sibling pointers) until we reach p^+. If $|S|$ denotes the number of objects returned to the user, including the two boundary ones, we access $\frac{|S|}{f-1} + 1$ pages in the worst case.

To create the VO we access a maximum of two DI pages. Regarding CPU time, for each boundary object p^- and p^+, VO computation involves on-the-fly digest computations (i) to construct the lower trees containing the boundary objects p^-, p^+, and (ii) to build the corresponding suppressed tree. The lower tree is computed over $f - 1$ objects, and we need a digest for each of its nodes, except for the root (whose value is materialized in the DI). Thus, we perform $2 \cdot f - 4$ hash operations. The suppressed tree is constructed on α digests, requiring as many hash computations as the number of its internal nodes, i.e., $\alpha - 1$. The overall query processing and VO computation cost is

$$\left(d_{MI} + \frac{|S|}{f-1} + 2\right) \cdot C_{I/O} + 2 \cdot (2 \cdot f + \alpha - 5) \cdot C_h \ . \quad (5)$$

Server-user communication cost. The server returns to the user set S (containing the objects that satisfy the query and the two boundary ones) and the VO. VO component (i) is the signed DI root of size $|s|$. Component (ii) (i.e., left sibling digests to the path of p^- in the conceptual DI) contains a maximum of d_{DI} hash values. Similarly, component (iii) consists of at most d_{DI} digests. Hashes common in components (ii) and (iii) are sent only once, but we consider the worst case where the paths of p^- and p^+ are disjoint. In addition to digests, VO also includes information about their position in the DI to facilitate proper verification at the user side. Similar to [17], we ignore this information in our analysis, as it is negligible compared to the size of the hashes in the VO. The total amount of information sent by the server to the user is

$$2 \cdot |S| \cdot |num| + 2 \cdot (d_{DI} - 1) \cdot |h| + |s| \ . \quad (6)$$

AC verification cost. Given the set S and the VO, the user has to compute the missing digests and combine them with the VO to retrieve the root of the DI. This procedure involves calculating $2 \cdot |S| - 1$ hashes on top of objects in S, and combining them (i.e., concatenating and then hashing) with the VO digests, incurring a maximum of $2 \cdot (d_{DI} - 1)$ additional digest computations. Finally, the user has to verify whether the computed DI root matches the one returned in the VO, using the owner's public key. Letting the cost of this operation be C_v, the total AC verification cost is

$$(2 \cdot |S| + 2 \cdot d_{DI} - 3) \cdot C_h + C_v \ . \quad (7)$$

4.3 Quantitative Comparison with Existing Methods

To provide an intuition about the performance of our technique, we use the above analysis and that in [17] to estimate the various costs of PMD, MB-, and EMB-tree in a typical problem setting. Table 2 contains expected values for the three methods in a scenario where data cardinality $N = 300$K objects, block size $B = 1$ Kbyte, and $|h|$, $|s|$, $|num|$ are 20, 128, and 4 bytes, respectively. We assume that a page access takes $C_{I/O} = 10$ msec, a hash computation $C_h = 3$ μsec, a signing operation $C_s = 3$ msec, and a signature verification $C_v = 200$ μsec. A space utilization of 67% is assumed for MI, MB- and EMB-tree (i.e., the average fanout is 67% of its maximum value shown in the first row of the table).

The fanout of MI in PMD is almost four times larger than its competitors. This is because the MI is a conventional B$^+$-tree, while the MB-/EMB-tree nodes store extra (verification) information, which limits their capacity. The reduced fanout of MB-/EMB-tree leads to over 3.5 times higher space requirements than PMD. The size of the latter breaks down to 3,583 Kbytes for MI and 81 Kbytes for DI, revealing that the PMD scheme enables

Property	PMD	MB	EMB
Fanout	128	36	36
Space Requirements (Kbytes)	3,664	13,538	13,538
Construction Time (sec)	38.44	135.42	137.18
Query Processing - no VO (sec)	3.57	13.05	13.05
Query Processing - with VO (sec)	3.59	13.05	13.05
VO Size (bytes)	848	2208	848
AC Verification (sec)	0.18	0.1	0.18

Table 2 Expected performance

AC verification with only 2% extra space (compared to a standard, non-authenticated B$^+$-tree). The construction time is dominated by the I/O cost to store the data-structures and, since PMD is much more concise than its competitors, it is built faster. The size of the indexes also determines the owner-server communication cost (if the indexes are built and uploaded by the owner), resulting in PMD being more efficient in terms of network overhead too. If the data-structures are built at the server, then the communication cost is the same for all methods.

Table 2 includes the processing cost of a range query with selectivity 10%, in situations where the user does or does not request for AC guarantees. PMD is more than three times faster than MB-/EMB-tree in both cases, mainly due to the higher fanout of the MI. In the non-authenticated case, PMD performs identically to a plain B$^+$-tree. In the authenticated case, it takes only 0.02 sec more (3.57 versus 3.59 sec), due to VO computation. This fact highlights that PMD provides AC guarantees at a marginal extra cost with respect to a non-authenticated B$^+$-tree. Note that the 0.02 sec spent for VO construction correspond essentially to the I/O cost for accessing 2 DI pages, since the 421 hash computations performed take only 0.0013 sec.

In terms of server-user communication cost, the difference lies in the VO size. The number of hashes in the VO increases with the MHT fanout. PMD and EMB-tree have binary (conceptual) MHTs and, thus, small and equal VO size[3]. MB-tree uses an MHT with a fanout of 36, leading to a much larger VO and, hence, higher network overhead. On the other hand, the AC verification cost at the user side decreases with the MHT fanout, and the MB-tree is the fastest method.

To summarize, according to our analysis, PMD is superior to its competitors in terms of all cost factors by a large margin, the only exception being the AC verification time. Our experiments in Section 7 verify the above empirically. In addition to the performance benefits of PMD over its competitors, it provides flexibility crucial to database outsourcing applications, as discussed in Section 1.

[3] As we show in the experiments, in practice PMD has smaller VO than EMB-tree because its MHT (i.e., the DI) is closer to a perfect tree.

4.4 Discussion

So far we assumed that our database contains objects of the form $\langle id, key\rangle$. An indexed relation, however, may have records with additional attributes (and size several times larger than $2{\cdot}|num|$). In this case, the MI is built on top of the relation file; its external nodes contain entries of the form $\langle ptr, key\rangle$, where ptr is a pointer to the actual record of the relation that corresponds to key.

The issue in this case is that the lower trees cannot be suppressed. Consider the example in Figure 3. An object digest $\mathcal{H}(p_i)$ is no longer $\mathcal{H}(id_i|key_i)$, but it is the hash value of the entire record in the relation that key_i corresponds to. In other words, we need to access the relation file in order to compute $\mathcal{H}(p_i)$. To cure this problem, we store into the DI all the object hashes. The DI structure is similar to Figure 4, the difference being that at the bottom level we have object digests instead of lower tree roots, and that the explicit tree becomes larger.

Every DI block contains verification information for a group of κ consecutive objects. There is a suppressed tree for every group, i.e., there are $\frac{N}{\kappa}$ suppressed trees in total. As such, the height of the explicit tree is $\log \frac{N}{\kappa}$. Each group and its corresponding explicit nodes (whose number is equal to the height of the explicit tree) are placed in a page, leading to the condition:

$$\left(\kappa + \log \frac{N}{\kappa}\right) \cdot |h| \leq B \ . \tag{8}$$

The size of the DI is $\frac{N}{\kappa}$ pages, which is considerably larger than before. However, the DI retains the property that VO construction requires a maximum of 2 page accesses (one for each boundary object). Apart from the structure of the DI, query processing, VO construction, and AC verification are similar to Section 4.1.

To provide some performance indications, let us consider the setting of Table 2 again. Here, κ is 38, which implies a DI of 7,895 Kbytes. MI is unchanged, and the total space consumption in PMD is 11,478 Kbytes (which is still 15% smaller than that for MB-/EMB-tree). The query processing time is not affected, and PMD remains more than three times faster than MB-/EMB-tree. The VO size and the AC verification cost are also identical to the ones shown in the first column of the table. Note that in the above discussion we focus on the index characteristics, and exclude from consideration the size of the indexed relation and the I/Os spent to fetch the entire result records (these costs are identical across all methods).

5 Dynamic One-dimensional Data

The basic PMD approach, as presented in Section 4, is targeted at and optimized for static databases. For dynamic datasets, however, its performance deteriorates,

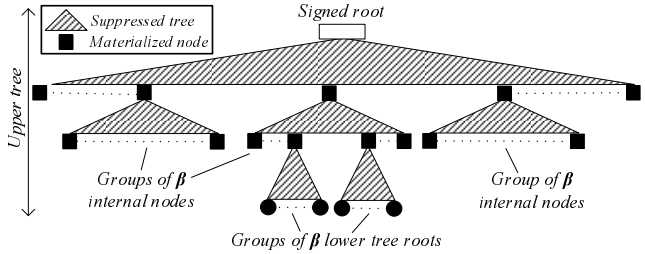


Fig. 5 Digest index in the dynamic case

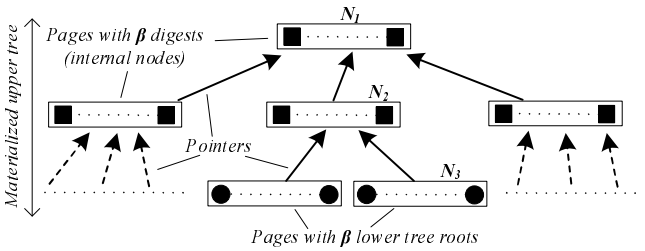


Fig. 6 Materialization of the DI

because each update (object insertion or deletion) requires modification of all DI pages. Consider the example in Figure 4, and assume that the owner inserts/deletes an object in the database. Let the external MI node that stores the object correspond to one of the leaves shown as solid circles. The updated value of this leaf causes digest changes that propagate upwards until the root of the DI, affecting H^4 and the explicit nodes shown as hollow squares. Each DI page materializes one of these three hashes; e.g., the DI block corresponding to the subtree of H^3 contains H^4, and DI pages below H^1 store the top hollow square (i.e., the left child of the root). Thus, we need to modify all DI blocks to reflect the update, incurring $(\frac{n}{\alpha})$ I/O operations.

To avoid this problem, we propose the dynamic PMD (dPMD). The main idea is to materialize the DI as a tree instead of a flat structure. To achieve this, while keeping the storage overhead and I/O cost for VO creation low, we materialize information only in specific levels of the DI, termed *materialized levels*, and suppress all the others (as opposed to suppressing only the lowest subtrees in the basic PMD technique).

Figure 5 illustrates this procedure. The DI is built bottom-up. The external level is a materialized level. The lower tree roots are partitioned into groups of β (we discuss β's calculation later). On top of each group, we construct a (suppressed) binary MHT[4]. The roots of these trees constitute the next materialized level; we partition them into groups of β and compute a (suppressed) binary MHT for each group. We repeat this procedure until the number of resulting root digests is no more than β. The suppressed trees are shown as striped triangles.

The DI skips the intermediate nodes of the suppressed trees, and stores only the digests in the materialized levels. It is essentially a tree with fanout β where each node corresponds to a suppressed tree; the node stores (i) the β leaf digests of the suppressed tree, and (ii) a pointer to its parent node. Note that the pointers in DI lead from the children to their parent. Figure 6 shows DI materialization for the example in Figure 5. Letting B be the block size, the materialization procedure requires that

$$\beta \cdot |h| + |num| \leq B \ . \tag{9}$$

[4] We only show two of these trees, and skip their intermediate levels.

VO construction is performed by bottom-up traversal of the DI. Consider Figure 6, and assume that we want to retrieve verification information for object p falling in an external MI node E_i. Let N_3 be the DI node that contains the root of E_i's lower tree. We first access N_3 and build its (suppressed) binary MHT on the fly. We insert into the VO the digests that are needed for p, and follow the pointer to N_2. We compute the suppressed tree over the hashes in N_2 and append the necessary digests to the VO. Finally, we access N_1 and retrieve the remaining digests by building its MHT. To form the VO for a range query, we need verification information for two boundary objects. We traverse the DI bottom-up for both objects simultaneously, ascending levels one by one, in order to avoid accessing common DI nodes (and computing their suppressed trees) twice.

Regarding update handling, assume that the owner inserts/deletes a point p into/from the database and that the update does not trigger reorganization of the MI. Let E_i be the external MI node that stores p. The lower tree of E_i must be recomputed and its root digest H_i updated in the DI. Consider Figure 6 and assume that H_i is stored in N_3. We access N_3, update the value of H_i, and recompute the suppressed tree of N_3. Following the parent pointer to N_2 and, repeating the same process for it, we reach N_1. Finally, we compute the root of the DI as the root of N_1's suppressed tree. The owner signs the new root and uploads it to the server. Modification of an object is treated as a deletion followed by an insertion.

It is possible that the updates incur reorganization in the external level of MI (i.e., node merges/splits). Consider the case where all DI nodes are packed and an external node of MI splits. This leads to recursive splits in the corresponding path of the DI. To avoid this problem, we utilize only a fraction of its node capacity when we build it for the first time. Determining this fraction depends on the expected update volume; in our experiments, setting it to 90% works well even for large numbers of updates. Another important remark regards the processing of multiple updates. The maintenance cost for both the MI and DI can be greatly reduced by processing updates in batches (rather than individually), since multiple updates may affect the same MI and DI nodes.

Accessing/modifying these nodes only once for many updates saves a considerable fraction of the I/O cost.

Performance analysis. The performance analysis of dPMD is similar to that in Section 4.2, the main difference being the number of nodes/pages in the DI. The DI is materialized as a tree with fanout β, built on top of n hashes. Thus, the materialized DI has height[5] $d'_{DI} = \log_\beta n$ and $\sum_{i=0}^{i=d'_{DI}-1} \beta^i = \frac{\beta^{d'_{DI}}-1}{\beta-1}$ nodes. Its construction time and storage overhead are derived by Formulae 2 and 4, respectively, by replacing factor $\frac{n}{\alpha}$ with $\frac{\beta^{d'_{DI}}-1}{\beta-1}$ as the DI size.

VO computation in dPMD incurs $2 \cdot d'_{DI} - 1$ I/O operations[6] instead of 2 for the basic PMD method. Furthermore, for each accessed DI node, we build a suppressed tree over its β hashes, requiring $\beta - 2$ hash computations (the root digest need not be computed as it is stored at the higher DI level). Thus, the CPU time for VO generation is $\frac{\beta^{d'_{DI}}-1}{\beta-1} \cdot (\beta-2) \cdot C_h$. The overall query processing and VO computation cost is given by Formula 5 subject to the above changes. The performance of dPMD for the remaining factors is given by the same formulae as in Section 4.2, since the height of the conceptual DI and the size of VO are the same as in PMD.

In the dynamic case, there is an extra performance factor, the update cost. It breaks down into MI and DI maintenance time, plus cost C_s for signing the new DI root. For both PMD and dPMD, the I/O for updating the MI is the same, and equal to d_{MI}, assuming that no reorganization takes place. The cost to update the DI in dPMD is $d'_{DI} \cdot (C_{I/O} + (\beta-2) \cdot C_h)$, due to accessing d'_{DI} DI nodes and computing their suppressed trees. This is much lower than that in PMD, where the required DI maintenance incurs $\frac{n}{\alpha} \cdot C_{I/O} + (2 \cdot f + \alpha - 5) \cdot C_h$ time, as all its pages are modified and the lower and suppressed trees of the inserted/deleted object are recomputed.

Discussion. To provide an intuition about the performance of dPMD, we use the scenario of Section 4.3 to compare it with alternative methods, starting with PMD. First, the DI in dPMD is smaller (73 versus 81 pages), because (i) it materializes fewer hashes (since it suppresses more subtrees/nodes), and (ii) PMD replicates some upper level hashes in multiple pages. This implies lower storage overhead and index construction time (requiring 3,656 versus 3,664 Kbytes, and 38.36 versus 38.44 sec, respectively). The dPMD technique is slightly slower only for VO computation, reading more DI pages and performing some extra hashing operations. In our example scenario, the difference is 0.03 sec, corresponding practically to three extra I/Os, since digest computations are very fast. The VO size and AC verification cost is the same between our methods, as the DI is binary in both cases. The most important difference is the update cost; updating the MI takes exactly the same time (0.03 sec) but maintaining the DI takes 0.03 versus 0.81 sec, for dPMD and PMD, respectively.

Essentially, dPMD achieves as large of a performance improvement over the MB-/EMB-tree as PMD does; i.e., the general picture is similar to Table 2. Also, it is superior in terms of update cost. The MB- and EMB-tree descend the index from the root down to the inserted/deleted object, updating the corresponding keys and digests in the visited nodes. Due to their smaller fanout (and, consequently, larger tree height), this procedure accesses more nodes than updating MI in dPMD. Furthermore, the low fanout of the MB-/EMB-tree implies more frequent index reorganization and, thus, higher cost.

A final remark concerns a critical point in updating outsourced databases. When the owner issues updates, a malicious server may ignore them, and continue to answer user queries over the obsolete dataset. Since the old version of the database was authenticated by the owner, the user considers the results legitimate. The problem has been identified in [17]. A practical solution is to append an expiry time to the tree root prior to signing, so that the users can verify the freshness of the results. Another common practice is for the owner to publish a list of revoked signatures, so that the users can reject obsolete verification information.

[5] Note that the height of the *conceptual* DI structure remains $d_{DI} = \log N$, assuming that all lower and suppressed trees are full.

[6] The root of DI is loaded once for both boundary objects, and thus the subtraction of one page access.

6 Verification in Spatial Databases

In this section we consider verification in spatial databases. We assume a unit data-space, i.e., all object coordinates (x, y) are in $[0, 1]^2$. We focus on range queries, which retrieve the objects that fall inside an axis-parallel rectangle, but our techniques extend easily to k nearest neighbor (k-NN) queries, as we discuss later. We present two methods, the Merkle R-tree (MR-tree) and the Partially Materialized KD-tree (PMKD). The first is an extension of the MB-tree method to R-tree indexes, and the second an adaptation of the PMD methodology for spatial data.

6.1 The Merkle R-tree

Our first method, termed Merkle R-tree (MR-tree), builds upon the R-tree index. We focus on the R-tree because it is the standard spatial access method. However, the MR-tree approach applies to other tree-like spatial indexes too. Intuitively, the MR-tree results from the MB-tree by treating node MBRs as the indexing keys. The difference is that the MR-tree also takes into account MBRs for hash calculations and VO computation, in order to provide AC guarantees.

The MR-tree is an R-tree, where each internal node N_i is associated with a digest h_{N_i} over (the concatenation of) the MBRs and digests of its children. To formalize, let e_j be the j-th entry (child) of N_i, and mbr_j be e_j's MBR. The digest of N_i is $h_{N_i} = \mathcal{H}(mbr_1|h_{e_1}|mbr_2|h_{e_2}|...)$, where hashes h_{e_j} are defined similarly for N_i's child nodes. Structurally, N_i contains tuples of the form $\langle mbr_j, h_{e_j}, ptr\rangle$, where mbr_j is represented by the coordinates of its bottom-left and top-right corners (i.e., by 4 real numbers), and ptr is a pointer to the disk block containing e_j. An exception is the external level, where each leaf node stores the data objects inside, and its digest is computed over the hashes of these objects (i.e., no MBR information is involved in object digests). The owner authenticates the MR-tree by signing (the concatenation of) the root's MBR and digest.

VO computation is performed during query processing, by inserting into the VO the MBR and digest of *each encountered* entry e_j that *does not intersect* the query range. In Figure 7 we demonstrate VO computation assuming that the structure of the MR-tree is the same as the R-tree in Figure 1(a), and that the query range is the shaded rectangle R. First, we access the root of the tree, and insert its signed digest into VO (the signed root is always included in the VO). The first root entry, N_1, does not intersect with R and, thus, we include its digest h_{N_1} and MBR mbr_{N_1} in VO. Note that h_{N_1} and mbr_{N_1} are stored in the root, and we do not need to access node N_1. The other root entry, N_2, overlaps with R and, hence, we visit it. N_2's entries N_8 and N_9 are disjoint from R and we append their MBRs and digests to VO (i.e., $mbr_{N_8}, h_{N_8}, mbr_{N_9}$ and h_{N_9}). Entries N_6 and N_7 of N_2, on the other hand, intersect R and are visited. Objects l, m, n, o, r form the query result, while encountered objects outside R (i.e., objects p, s, x) are included in VO.

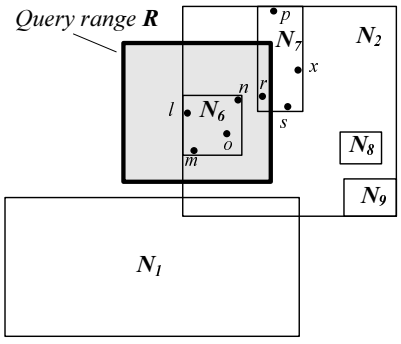


Fig. 7 Range query in MR-tree

Upon receipt of the result and its VO, the user can verify authenticity and completeness, following a procedure with as many steps as the MR-tree levels (i.e., three in our example). In a first step (corresponding to the leaf level), she computes the hashes of all returned objects (regardless of whether they belong to the result or the VO), and derives h_{N_6} and h_{N_7}. In a second step (for the second level), she uses the object coordinates to derive[7] mbr_{N_6} and mbr_{N_7}. Then, she derives h_{N_2} from calculated $\{mbr_{N_6}, h_{N_6}, mbr_{N_7}, h_{N_7}\}$ and VO entries $\{mbr_{N_8}, h_{N_8}, mbr_{N_9}, h_{N_9}\}$. Having ascended to the root level, in the third step, the user computes mbr_{N_2} (from mbr_{N_6}, mbr_{N_7} as calculated in the previous step) and VO entries mbr_{N_8}, mbr_{N_9}. Then, she computes the root MBR (from mbr_{N_2} and VO entry mbr_{N_1}). Finally, she concatenates the root MBR with its digest (derived from previously computed mbr_{N_2}, h_{N_2}, and VO entries mbr_{N_1}, h_{N_1}), and verifies whether it matches the signed root of the MR-tree.

An important remark regarding the VO in the MR-tree, is that (in contrast to the MB-tree case) it includes considerable extra information to indicate the position of its components in the R-tree. In our example, the user needs to know that the root contains two entries, and that the MBR and digest of the first entry are the ones stored in the beginning of the VO. Similarly, for N_2, she needs to know which its entries are and their ordering, and so on. This hierarchy information, essentially, provides the structure of the visited part of the MR-tree.

Similar to the MB-tree, the MR-tree has two main performance shortcomings. The first is that its fanout is low, since storing a digest per entry in each internal node consumes considerable space. In the typical case where $|num| = 4$ and $|h| = 20$ bytes, the fanout is reduced to half with respect to a simple R-tree, since every entry requires 40 instead of 20 bytes. The second drawback is that the VO contains MBRs and digests of numerous MR-tree entries, incurring large server-user communication cost and AC verification overhead. These problems motivate us to extend the PMD methodology to spatial databases, yielding PMKD (described in the next section). We consider PMKD our primary contribution for spatial databases, and use the MR-tree as a baseline that represents the conventional approach of integrating the data with the verification indexes. We note that a technique similar to the MR-tree appears in [33].

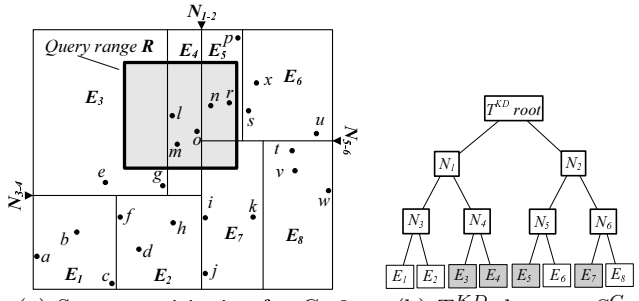
6.2 The Partially Materialized KD-tree

PMKD follows the PMD paradigm, extracting the hashes from the main index (MI), and storing them into a separate structure, the digest index (DI). The MI can be any spatial access method. Its leaf level, however, must be a partition of the data-space; i.e., the extents of external MI nodes should not overlap with each other, and their union should be the entire data-space. Therefore, prior to MI construction, we iteratively partition the space with a KD-tree, denoted as T^{KD}, until the number of objects in each bucket becomes less than or equal to $C = \frac{B}{3 \cdot |num|}$; C is the maximum number of objects that fit in one disk

[7] Note that a node MBR is unambiguously defined as the minimum box enclosing its entries.

page, assuming that each object is represented as a tuple $\langle id, x, y\rangle$, where x and y are its coordinates. Let S^C be the set of resulting T^{KD} buckets. The MI is built in two steps. First, we form a page (i.e., an external MI node) E_i for each bucket in S^C, by storing inside all the objects of the bucket. In a second step, we assign to every E_i the extent br_{E_i} of the corresponding bucket, and index them on br_{E_i} with a spatial access method (e.g., an R-tree). The resulting structure is used as the MI. Note that, at the external MI level, we use the br_{E_i} bounding boxes instead of tight MBRs, so that the space partitioning property is retained. This property is necessary for checking the completeness of query results.

Assuming the data of Figure 1(a) and that $C = 3$, Figure 8 shows the buckets in S^C, and the T^{KD} structure. We use notation N_j for internal T^{KD} nodes. The N_{i-j} marks in Figure 8(a) indicate the splitting lines between nodes N_i and N_j. We denote the leaves of T^{KD} (i.e., the buckets in S^C) by E_i, to indicate their connection with the MI; external MI node E_4, for instance, contains objects l, m, o and is indexed (in the MI) according to the extent of its bucket in Figure 8(a). Processing a range query R on the MI accesses (in addition to some internal nodes) leaves whose br_{E_i} overlaps with R, i.e., it loads nodes/pages E_3, E_4, E_5 and E_7.

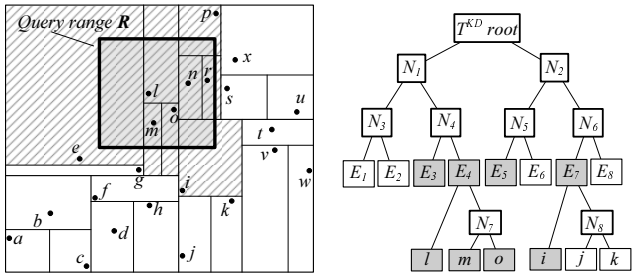


(a) Space partitioning for C=3 (b) T^{KD} down to S^C

Fig. 8 T^{KD} partitioning and structure

Having built MI, it remains to construct the DI for VO computation. The DI is derived by T^{KD} as follows. Given S^C, we continue KD-tree splitting within each E_i until every bucket contains exactly one object, i.e., we continue partitioning the space with T^{KD} all the way to the data level. Figure 9 illustrates the final T^{KD} partitioning and structure (we only show the subtrees of E_4 and E_7, to avoid cluttering the figure). Each object p resides in one and only one bucket. We denote as br_p the extent of p's bucket, and refer to it as the bounding region of p. We define as digest of object p the hash value $h_p = \mathcal{H}(p.id|p.x|p.y|br_p)$, i.e., we include its bounding region in the digest computation.

Conceptually, the DI is a binary MHT built on top of the object digests, and having the same structure as T^{KD}. We include br_p into h_p due to our AC verification mechanism described next. The DI is shown in Figure 10,



(a) Final space partitioning (b) Final T^{KD}

Fig. 9 Final T^{KD} partitioning and structure

skipping again some lower trees for clarity. Its upper tree follows T^{KD} down to S^C while the lower trees are given by the subtrees of each E_i in the final T^{KD}. The owner certifies the DI by signing its root. Lower trees (along with br_p for objects in S) are implicit, i.e., they are computed on the fly when necessary. The upper tree is materialized in the same way as in PMD (Section 4) for static data, and as in dPMD (Section 5) for dynamic ones.

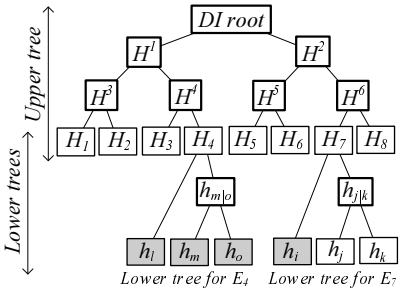


Fig. 10 DI built according to T^{KD}

Given a range query R, the server returns result S containing all objects p whose br_p overlaps with R. Result completeness can be proven if the union of the br_p of objects in S completely covers R. The disjointness of bounding regions guarantees that no object outside S falls in R. Thus, the server must (i) return regions br_p to the user as part of the VO, and (ii) show that they are authentic (i.e., that their extent and contained object have not been tampered with). Therefore, it verifies each returned object p individually, by including into the VO *all* sibling hashes (both left and right) to p's path in DI.

Consider the query range R in Figure 9. Result S contains all objects e, i, l, m, n, o, p, r, since their bounding regions (shown striped in Figure 9(a)) overlap with R. To verify m, for instance, according to the DI in Figure 10, the user needs h_o, h_l, H_3, H^3, and H^2. Since o and l are in S, and the VO contains their bounding regions, h_o and h_l can be computed at the user side. Thus, only H_3, H^3, and H^2 are inserted into the VO for m. Simi-

larly, we include hashes necessary for the verification of the remaining objects in S.

So far we considered static data. Both the MR-tree and PMKD, however, also work for dynamic databases. The former is updated in a similar way to the MB-tree. The latter can efficiently handle updates, provided that the DI is materialized as in Section 5 for dPMD. Furthermore, both techniques also apply to other types of queries, such as k Nearest Neighbors (k-NN). A k-NN query retrieves the k data objects that are closest to a user-specified query point. The MR-tree extends to k-NN processing using a standard NN search algorithm for R-trees (e.g., the best-first technique of [14]). PMKD is even more flexible, since its verification mechanism is independent of query processing in the MI. VO computation/AC verification is essentially identical to that of a circular range query with center at the query point and radius equal to the distance of the k-th (i.e., farthest) NN.

The focus of this section is on spatial (i.e., two-dimensional) databases, but our techniques extend directly to higher dimensions. They rely, however, on spatial indexes which are known to suffer from the dimensionality curse [16]. Their extension to specialized high-dimensional indexes is an interesting direction for future work.

7 Experimental Evaluation

In this section, we experimentally evaluate the performance of our methods. In Section 7.1 we focus on static one-dimensional data, while in Section 7.2 we consider dynamic ones. In Section 7.3 we compare our spatial methods, and in Section 7.4 we summarize our experimental results.

7.1 Static Data

First, we compare PMD and dPMD with the MB- and EMB-tree on static data. For the latter two, we use the implementation of [17] available at `http://cs-people.bu.edu/lifeifei/aisl/`. We center our empirical study around the space requirements, the index construction cost, and the query processing time, but we also cover the other performance factors listed in Section 1. In accordance with [17], we use synthetic datasets with uniformly distributed keys. The parameters in our simulations are the data cardinality N, the block size B, and the query selectivity σ. In each experiment, we vary a single parameter and set the remaining ones to a default value. N ranges between 100K and 500K objects, with default $N = 300$K. B varies between 512 bytes and 4 Kbytes, with default $B = 1$ Kbyte. σ varies from 1% to 50% of the database size, with default $\sigma = 10\%$. All digests are 20 bytes long and computed using SHA1. On our machine (with a 3 Ghz Pentium 4 CPU), a hash computation takes $C_h \simeq 3$ μsec, and a random disk I/O takes $C_{I/O} \simeq 10$ msec.

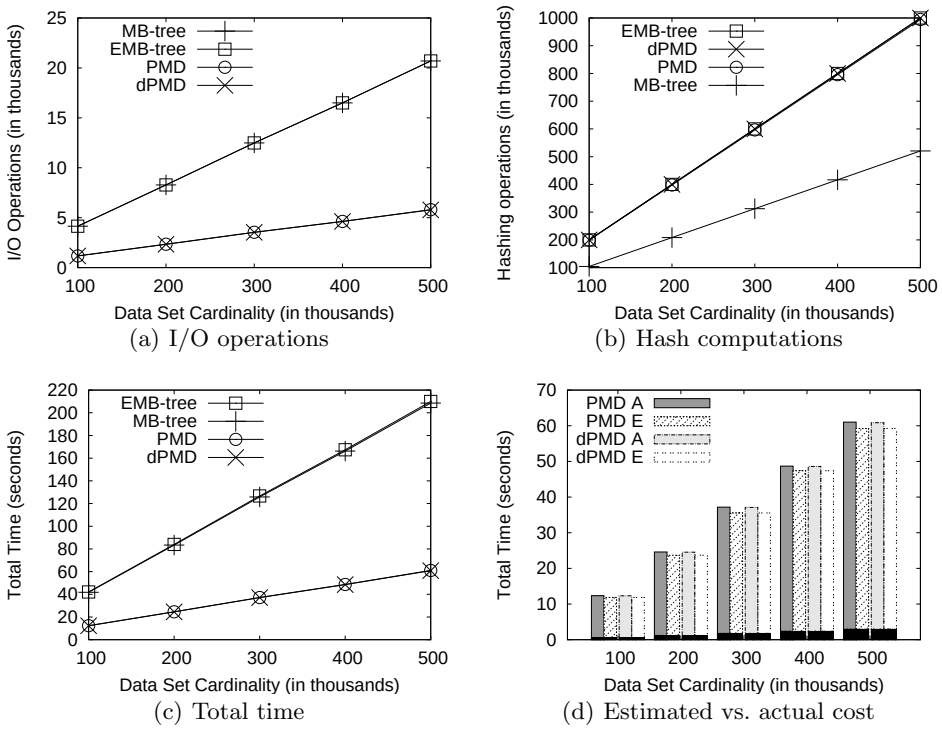
Property	PMD	dPMD	MB	EMB
Fanout	128	128	36	36
Average Fanout	87	87	24	24
Total Size (Kbytes)	3,538	3,530	12,487	12,487
MI Height	3	3	4	4
MI Nodes	3,458	3,458	12,487	12,487
DI Height	1	3	–	–
DI Nodes	80	72	–	–
Materialized Hashes	4,080	3,672	312,486	312,486

Table 3 Index characteristics

Space Requirements. In Table 3, we summarize the index characteristics for the 300K dataset. The low fanout of the MB- and EMB-tree, and the numerous materialized hashes, lead to a large storage overhead; they require around 3.5 times more space than PMD and dPMD. Note also that the verification information (i.e., the materialized hashes) take up less than 80 Kbytes in our methods, versus 6,103 Kbytes for MB-/EMB-tree. In addition to space consumption at the server, the size of the structures also affects the owner-server communication cost if the indexes are constructed at the owner side.

Another important observation about our techniques concerns the size ratio between DI and MI. Since MI is a plain B^+-tree, the relative storage requirements of DI and MI indicate that PMD and dPMD take up, respectively, only 2.3% and 2% additional space compared to a non-authenticated, conventional B^+-tree. This percentage does not exceed 3% in any of the cases that we tested.

Construction Cost. The factors that contribute to the total construction time, albeit with a different impact are (i) the number of I/O operations, and (ii) the number of hashing operations. To illustrate their individual effect, we vary N from 100K to 500K and plot the corresponding costs in Figures 11(a) and 11(b). The number of I/Os is directly related to the index fanout. Building the MB- and EMB-tree incurs the same number of I/Os, since they have the same structure and number of nodes. The cost of PMD is similar to dPMD, because they use the same MI. Due to its smaller DI though, the latter incurs 0.3% fewer I/Os. The increased fanout of our methods (128 versus 36 for the MB-/EMB-tree) leads to significantly lower I/O cost. The difference from the MB-/EMB-tree grows for larger N. Figure 11(b) depicts the number of hashing operations for the same experiment. PMD, dPMD and EMB-tree use binary MHTs and, hence, they require roughly the same number of hash operations. On the other hand, the MB-tree includes an MHT with fanout 36, and performs 1.9 times fewer computations. Figure 11(c) shows the total construction time, which is clearly dominated by the I/O cost. Overall, our methods are around 3.4 times faster than the MB-/EMB-tree.



(a) I/O operations (b) Hash computations (c) Total time (d) Estimated vs. actual cost

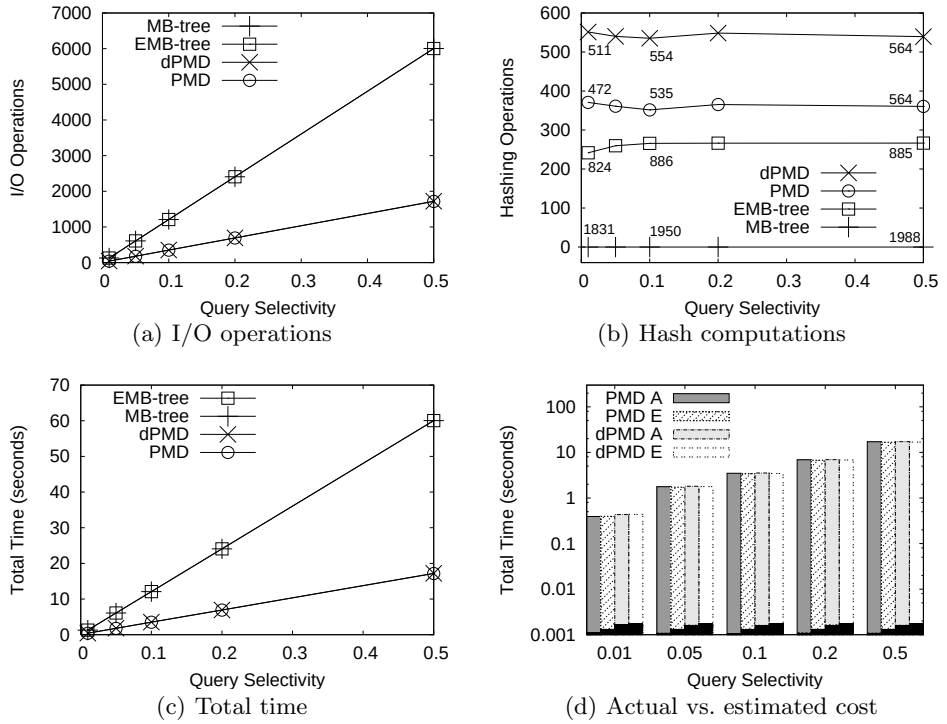
Fig. 11 Index construction cost

To assess the accuracy of our cost models, we plot in Figure 11(d) the actual versus the estimated construction cost, labeled as "A" and "E" in the bar-chart, respectively. The estimations are produced by Formula 2, taking into account that the average fanout is 87. The bars in Figure 11(d) depict the total cost, while the black portion of each bar corresponds to the CPU time (which is less than 5% of the overall cost). This figure demonstrates (i) the accuracy of our cost model (deviating by less than 3% from the actual measurements), and (ii) that I/O operations contribute more than 95% of the total construction time. The latter validates our fundamental design principle to avoid storing digests that can be computed on the fly.

Query and Verification Cost. To investigate the query processing/VO computation performance, we create workloads of range queries with selectivity σ varying between 1% and 50%. Figures 12(a) and 12(b) depict the effect of σ on the number of I/O and hash operations, respectively. The measurements correspond to average values over 100 randomly generated queries of the desired selectivity. PMD and dPMD require up to 3.5 times fewer I/Os than their competitors, due to their higher fanout. PMD and dPMD incur the same I/Os for query processing, since they use the same MI. PMD, however, is slightly better, incurring at most 2 I/Os for VO computation, versus 5 for dPMD (there are 3 levels in its DI). This performance gain of around 3 pages is independent of σ, and is not obvious in the figure. Regarding VO computation, as shown in Figure 12(b), the MB-tree requires no hashing operations. The runner up is the EMB-tree, followed by PMD and dPMD. As expected, our methods involve more digest computations, since they materialize fewer hashes. This approach pays off, as the overall cost (shown in Figure 12(c)) is dominated by the I/Os and follows a trend similar to Figure 12(a).

Figure 12(d) depicts the actual and estimated query processing time for our methods, and shows its breakdown into I/O and CPU cost. The plot is in logarithmic scale so that the CPU time (shown as the black portion at the bottom of each bar) is legible. Our model (Formula 5) is very accurate, with an error of less than 4%. The hashing cost is 1 msec and 1.6 msec for PMD and dPMD, respectively, corresponding to a small fraction of the total time.

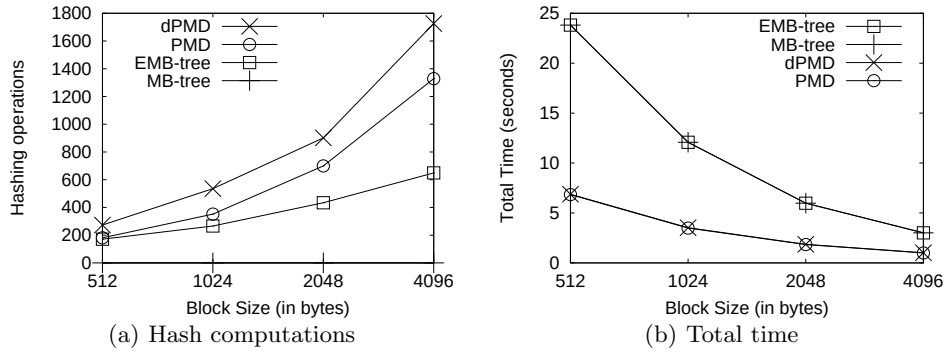
In addition to query response time, our methods are also preferable in terms of server-user communication cost. In Figure 12(b), the number next to each marker for selectivities 1%, 10%, and 50% indicates the VO size in bytes. The number of digests inside the VO increases with the MHT fanout. Thus, our methods construct the smallest VO, while the MB-tree the largest. The VO of the EMB-tree includes more digests than PMD/dPMD, even though it uses binary MHTs too. The reason is that its embedded MHTs are far from perfect trees, and the VO spans several of them. In contrast to the VO size, the verification time at the user side is reduced for a higher MHT fanout. Thus, the MB-tree requires fewer compu-



(a) I/O operations (b) Hash computations

(c) Total time (d) Actual vs. estimated cost

Fig. 12 Query processing and VO computation cost



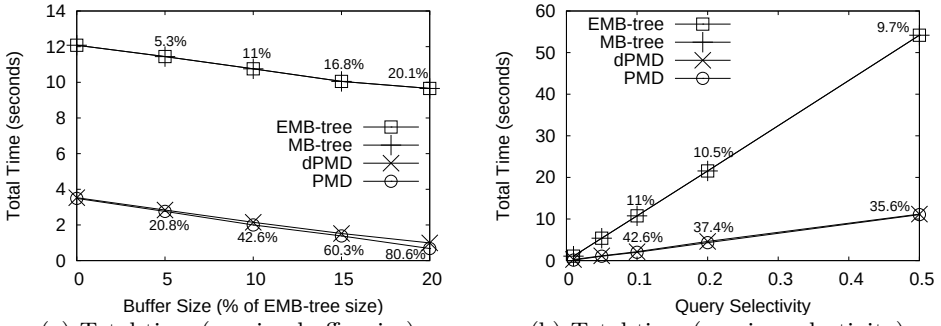
(a) Hash computations (b) Total time

Fig. 13 Query cost vs. block size

tations; for $\sigma = 10\%$, AC verification takes 0.09 seconds for MB-tree, and around 0.19 seconds for the other three methods.

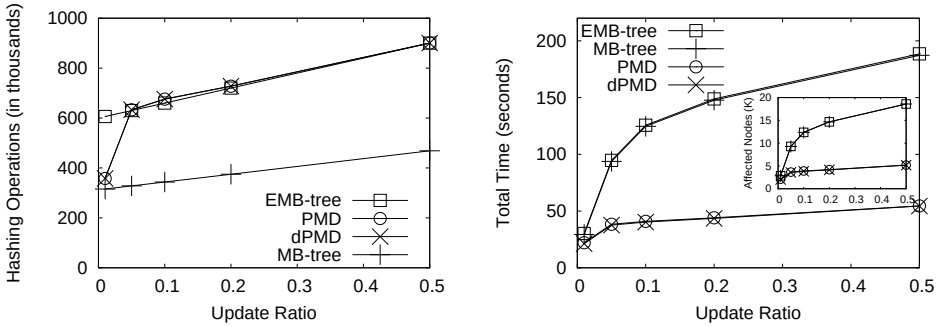
In Figure 13, we investigate the effect of block size B on the query answering time. We vary B from 512 bytes to 4 Kbytes, while keeping $\sigma = 10\%$. Figures 13(a) and 13(b) show the number of hash operations and the total time. The EMB-tree, PMD and dPMD compute more digests as B increases, since their conceptual MHTs grow. The MB-tree does not calculate any hashes in all cases. On the other hand, the I/O cost decreases with B for all methods. In terms of total time, our techniques are clearly better. The performance difference from our competitors decreases with B. However, PMD and dPMD are still 3 times faster for $B = 4$ Kbytes. Our competitors would become better only for unrealistically large block sizes (128 Kbytes for the MB-tree and 256 Kbytes for the EMB-tree, according to the analyses in [17] and Section 4.2).

The experiments so far do not use a buffer. Figure 14 now studies the performance impact of an LRU buffer. In Figure 14(a), we plot the overall query processing and VO creation time (in the default setting) for various buffer sizes. In each experiment, the buffer size is the same across all methods, and is expressed as a percentage over the size of the largest structure (i.e., MB-/EMB-tree); e.g., label 10 of the x-axis means that the buffer contains 1249 pages (10% of the MB-/EMB-tree size). The percentages next to the measurements represent the improvement achieved over the case where



(a) Total time (varying buffer size) (b) Total time (varying selectivity)

Fig. 14 Query cost in the presence of a buffer



(a) Hash computations (b) Total time

Fig. 15 Batch update cost

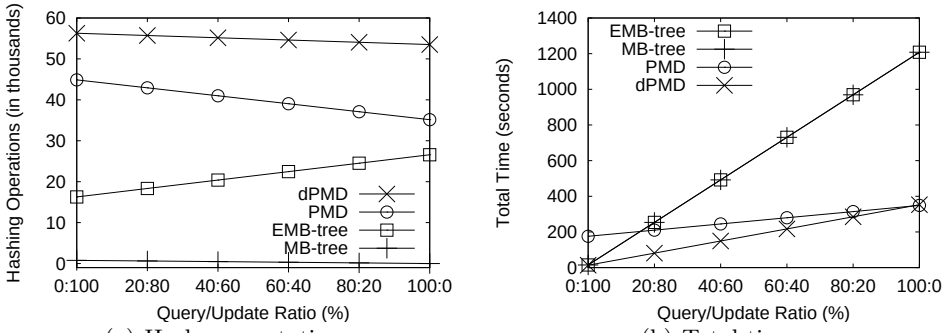
no buffer is used. All techniques utilize the buffer and their cost drops. However, as the buffer size increases, PMD/dPMD achieve a larger relative improvement compared to MB-/EMB-tree, because a greater fraction of their (concise) structures stays in main memory. For a 2497-page buffer (the largest tested) our methods are more than 10 times faster than their competitors. Figure 14(b) fixes the buffer size at 1249 pages (10% of MB-/EMB-tree space) and shows the total processing time for queries of different selectivities. The performance improves, but the trends are similar to Figure 12(c) (i.e., without the use of a buffer). PMD/dPMD are around 4.5 times better in all cases.

7.2 Dynamic One-dimensional Data

Update Cost. To investigate the dynamic scenario, we experiment on two settings; (i) batch updates, and (ii) mixed query/update workloads. The latter reflects a realistic dynamic scenario where queries and updates are intermixed.

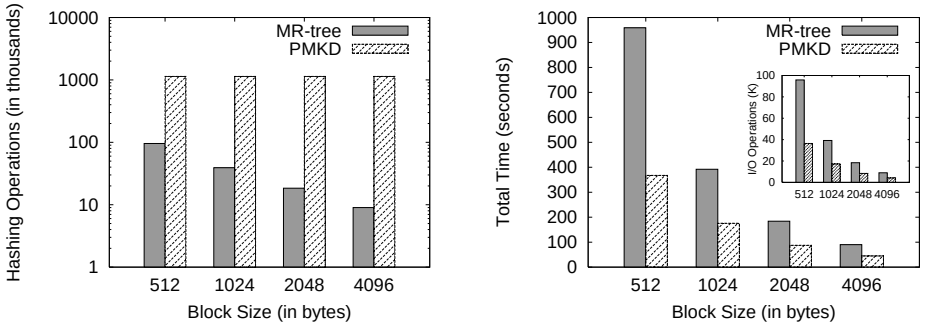
First, we generate batches of updates, similar to the experiments in [17]. Each batch contains $\upsilon \cdot N$ updates, where parameter υ varies from 1% to 50%; i.e., for default $N = 300$K, an υ of 50% implies that there are 150K updates. The updates are processed simultaneously, sharing computations and I/Os for common affected digests and index nodes. Figure 15(a) illustrates the number of hash computations. The MB-tree requires the fewest digest computations, since it performs a single hash operation per affected node, as opposed to rebuilding entire embedded or conceptual trees. However, the total update cost is dominated by the I/O operations and, thus, by the number of affected nodes. Figure 15(b) shows the overall update time, whereas the embedded figure depicts the number of affected nodes per method. In terms of total cost, dPMD is the best method. Interestingly, PMD performs almost as well as dPMD. The reason is that each batch of updates affects a large part of the DI in dPMD, leading to a cost comparable to rewriting the entire DI in PMD. Both our techniques are up to 3.4 times faster than the MB-/EMB-tree.

Figure 16 focuses on mixed workloads and assumes incremental processing (i.e., each update is processed as soon as it arrives at the server). We create workloads of 100 operations, consisting of queries (with $\sigma = 10\%$) and updates at a predefined ratio ρ. We vary ρ from 0:100 (updates only) to 100:0 (queries only). Figure 16(a) shows the number of digest computations, and Figure 16(b) plots the total time. Even though dPMD performs the largest number of hashing operations, it is overall the most efficient method for any ratio, except for 100:0 (queries only) where PMD is marginally faster.



(a) Hash computations (b) Total time

Fig. 16 Mixed query/update workload



(a) Hash computations (b) Total time

Fig. 17 Spatial index construction cost

Property	PMKD	MR
Fanout	50	25
Total Size (Kbytes)	17,232	39,204
MI Nodes	16,910	39,204
DI Nodes	322	–
Materialized Hashes	16,422	608,323

Table 4 Index characteristics for spatial data

Another interesting observation is that PMD performs worse than the MB- and EMB-tree in the 0:100 setting (updates only), because it modifies/rewrites the entire DI for every update.

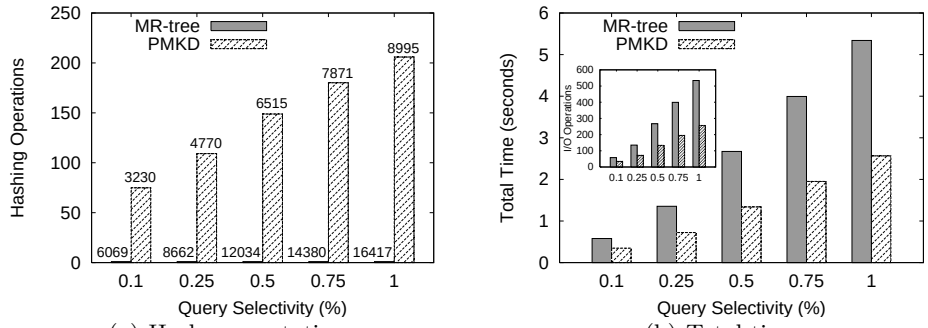
7.3 Spatial Data

In this section, we evaluate our spatial techniques. We compare the MR-tree and PMKD on a real dataset containing 569,120 two-dimensional points (available at `http://www.maproom.psu.edu/dcw`). PMKD uses an R-tree as the MI, and a PMD-like DI materialization. Table 4 shows the index characteristics of both methods for block size $B = 1$ Kbyte. The MI in PMKD has double the fanout, and less than half the size of the MR-tree, because it materializes just a few hashes. The DI (in PMKD) is very concise, with only 322 Kbytes of verification information, versus 11,881 Kbytes for the MR-tree.

Figure 17 plots the number of hash operations and the total time to construct the data-structures, varying the block size B between 512 bytes and 4 Kbytes. We observe that PMKD computes more hashes, but due to its smaller size, it incurs fewer I/Os. The I/Os dominate the overall construction cost and, thus, the PMKD structures are much faster to build than the MR-tree.

In Figure 18, we investigate the processing cost for range queries. We fix B to 1 Kbyte and vary the query selectivity from 0.1% to 1%. For each selectivity, we generate randomly 100 queries, and present the average measured values. In Figures 18(a) and 18(b), we show the number of hashing operations and the total time for processing/VO construction. The MR-tree does not compute any hash. Overall, however, (i.e., including the I/O cost) PMKD is the clear winner, due to its larger fanout.

PMKD is also superior to the MR-tree in terms of VO size and, thus, server-user communication cost. The numbers above the bars in Figure 18(a) indicate the VO size in bytes for both methods; PMKD owes its small VO to its binary DI (versus a fanout of 25 for the MHT of the MR-tree). To conclude the empirical study on the spatial methods, we need to mention that we also evaluated them for k-NN queries (implemented in the way described at the end of Section 6), but we omit the figures because the observed behaviors are similar to the range query experiments.



(a) Hash computations (b) Total time

Fig. 18 Query processing on spatial data

7.4 Summary of Experimental Results

To summarize, the best method for static data is PMD, with dPMD being a close runner up. Since the latter is significantly better for dynamic data, we consider dPMD to be the best technique overall, due to its generality and graceful performance on the whole. In the spatial case, PMKD is the preferred method, confirming that the PMD methodology retains its advantages in spatial databases too.

8 Conclusion

In this paper we propose a framework for authenticity and completeness (AC) verification in outsourced databases. We use separate indexes for the data and the verification information, and we compress the latter. Our approach applies to static and dynamic databases. As opposed to previous work, if a user does not request for AC guarantees, then the performance of our methods is similar to an ordinary, non-authenticated index. A theoretical analysis and an extensive experimental evaluation indicate that our one-dimensional techniques outperform the existing state-of-the-art by a wide margin; they achieve over 3 times smaller query processing cost, storage overhead, and index construction time in all examined settings. Furthermore, we design spatial verification techniques. We establish the generality of our methodology showing experimentally that its adaptation to spatial queries retains its advantages over the conventional approach of combining data and verification information in a single index.

A promising direction for future work is the authentication of continuous queries over streaming data. Two recent methods on this topic [18,28] employ concepts similar to the MB-/EMB-tree. It would be interesting to extend our methodology to data streams and compare with the existing approaches. There are several challenges arising in this setting, relating primarily to the highly dynamic nature of the data, the continuous updating of query results and verification objects, and the fact that processing takes place in main memory, ruling out I/O considerations.

References

1. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: SIGMOD Conference, pp. 439–450 (2000)
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The r*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD Conference, pp. 322–331 (1990)
3. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications (Second Edition). Springer-Verlag (2000)
4. Bertino, E., Carminati, B., Ferrari, E.: Merkle tree authentication in uddi registries. Int. J. Web Service Res. **1**(2) (2004)
5. Bertino, E., Carminati, B., Ferrari, E., Thuraisingham, B.M., Gupta, A.: Selective and authentic third-party distribution of xml documents. IEEE TKDE **16**(10), 1263–1278 (2004)
6. Carminati, B., Ferrari, E., Bertino, E.: Securing xml data in third-party distribution systems. In: CIKM, pp. 99–106 (2005)
7. Cheng, W., Pang, H., Tan, K.L.: Authenticating multi-dimensional query results in data publishing. In: DBSec, pp. 60–73 (2006)
8. Comer, D.: Ubiquitous b-tree. ACM Comput. Surv. **11**(2), 121–137 (1979). DOI http://doi.acm.org/10.1145/356770.356776
9. Devanbu, P.T., Gertz, M., Martel, C.U., Stubblebine, S.G.: Authentic third-party data publication. In: DBSec, pp. 101–112 (2000)
10. Devanbu, P.T., Gertz, M., Martel, C.U., Stubblebine, S.G.: Authentic data publication over the internet. Journal of Computer Security **11**(3), 291–314 (2003)
11. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD Conference, pp. 47–57 (1984)
12. Hacigümüs, H., Iyer, B.R., Li, C., Mehrotra, S.: Executing sql over encrypted data in the database-service-provider model. In: SIGMOD Conference, pp. 216–227 (2002)
13. Hacigümüs, H., Mehrotra, S., Iyer, B.R.: Providing database as a service. In: ICDE, pp. 29–40 (2002)
14. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM TODS **24**(2), 265–318 (1999)
15. Hore, B., Mehrotra, S., Tsudik, G.: A privacy-preserving index for range queries. In: VLDB, pp. 720–731 (2004)

16. Korn, F., Pagel, B.U., Faloutsos, C.: On the 'dimensionality curse' and the 'self-similarity blessing'. IEEE Trans. Knowl. Data Eng. **13**(1), 96–111 (2001)
17. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: SIGMOD Conference, pp. 121–132 (2006)
18. Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G.: Proof-infused streams: Enabling authentication of sliding window queries on streams. In: VLDB, pp. 147–158 (2007)
19. Martel, C.U., Nuckolls, G., Devanbu, P.T., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica **39**(1), 21–41 (2004)
20. Merkle, R.C.: A certified digital signature. In: CRYPTO, pp. 218–238 (1989)
21. Miklau, G., Suciu, D.: Controlling access to published data using cryptography. In: VLDB, pp. 898–909 (2003)
22. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. In: NDSS (2004)
23. Mykletun, E., Narasimha, M., Tsudik, G.: Signature bouquets: Immutability for aggregated/condensed signatures. In: ESORICS, pp. 160–176 (2004)
24. Narasimha, M., Tsudik, G.: Dsac: integrity for outsourced databases with signature aggregation and chaining. In: CIKM, pp. 235–236 (2005)
25. N.I.S.T.: Fips pub 180-1: Secure hash standard. national institute of standards and technology (1995)
26. Pang, H., Jain, A., Ramamritham, K., Tan, K.L.: Verifying completeness of relational query results in data publishing. In: SIGMOD Conference, pp. 407–418 (2005)
27. Pang, H., Tan, K.L.: Authenticating query results in edge computing. In: ICDE, pp. 560–571 (2004)
28. Papadopoulos, S., Yang, Y., Papadias, D.: Cads: Continuous authentication on data streams. In: VLDB, pp. 135–146 (2007)
29. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
30. Rizvi, S., Mendelzon, A.O., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: SIGMOD Conference, pp. 551–562 (2004)
31. Sion, R.: Query execution assurance for outsourced databases. In: VLDB, pp. 601–612 (2005)
32. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity auditing of outsourced data. In: VLDB, pp. 782–793 (2007)
33. Yang, Y., Papadopoulos, S., Papadias, D., Kollios, G.: Spatial outsourcing for location-based services. In: ICDE (2008)