Neural Computing and Applications (2023) 35:133–145
https://doi.org/10.1007/s00521-021-05804-2

S.I. : INFORMATION, INTELLIGENCE, SYSTEMS AND APPLICATIONS

HyR-tree: a spatial index for hybrid flash/3D XPoint storage

Athanasios Fevgas[1] · Leonidas Akritidis[3] · Miltiadis Alamaniotis[2] · Panagiota Tsompanopoulou[1] · Panayiotis Bozanis[3]

Received: 15 February 2020 / Accepted: 5 February 2021 / Published online: 25 February 2021

Abstract
Flash-based SSDs have become well established in the storage market, replacing magnetic disks in both enterprise and consumer computer systems. The performance characteristics of these new devices have prompted a considerable amount of research that aims at developing efficient data access methods. Early works attempted to reduce the expensive random writes, exploiting logging and batch write techniques, while more recent ones addressed query processing, taking advantage of the high internal parallelism of SSDs. 3D XPoint is a new nonvolatile memory technology that has emerged recently, featuring smaller access times and higher durability compared with flash. It is available both as block addressable secondary storage and as byte addressable persistent main memory. However, the high cost of 3D XPoint prevents, for the moment, its adoption in large scales. This renders hybrid storage systems utilizing NAND flash and 3D XPoint a sufficient alternative. In this work, we propose HyR-tree, a hybrid variant of R-tree that persists a part of the tree in the high performing 3D XPoint storage. HyR-tree identifies repeated access pattern to the data and uses these patterns to locate the most important nodes. The importance of a node is determined by the performance gain that derives from its placement within a 3D XPoint-based device. We experimentally evaluated HyR-tree using real devices and four different datasets. The acquired results show that our proposal achieves significant performance gains up to 40% in tree construction and up to 56% in range queries.

Keywords Data access methods · R-tree · SSD · Flash · 3D XPoint · Time series

1 Introduction

The term data structure refers to data organization as well as a set of algorithms that operate on these data. The basic operations include insertions, deletions, and searches, as well as some potentially useful functions such as rebalancing and traversals. An index is a special purpose data structure especially designed to facilitate and accelerate the access to the contents of a file. Among a huge number of indexes, R-tree has been introduced with the aim of organizing, managing, and searching multi-dimensional data. Indicative applications include (geo)spatial data, multimedia databases, feature vectors for machine learning applications, and navigational information systems [33].

Moreover, during the past decade, the appealing characteristics of the modern flash-based secondary storage devices led to a gradual replacement of the traditional magnetic hard drives in both commercial and scientific computer systems. Broadly known as solid-state drives (SSDs), these units adopt a completely different technology

✉ Athanasios Fevgas
fevgas@uth.gr

Leonidas Akritidis
lakritidis@ihu.gr

Miltiadis Alamaniotis
miltos.alamaniotis@utsa.edu

Panagiota Tsompanopoulou
yota@uth.gr

Panayiotis Bozanis
pbozanis@ihu.gr

1 Data Structuring and Eng. Lab., Department of Electrical and Computer Engineering, University of Thessaly, 38221 Volos, Greece

2 Department of Electrical and Computer Engineering, University of Texas at San Antonio, San Antonio, USA

3 School of Science and Technology, International Hellenic University, Thessaloniki, Greece

for storing digital information. More specifically, instead of using magnetic rotating disks coupled by one or more read/write heads, SSDs employ flash memory cells consisting of floating-gate MOSFETs to achieve nonvolatile storage. Consequently, in the absence of movable electromechanical parts, SSDs combine high data transfer rates, low power consumption, increased durability, and improved shock resistance.

Apart from these beneficial features, SSDs also exhibit significant differences in their read and write rates (i.e., reads are consistently faster than writes), whereas their continuous usage leads to degradation in performance and even non-recoverable damages in the device. These effects have attracted numerous researchers to propose enhanced versions of traditional data structures and algorithms with the aim of maximizing their performance. The relevant literature includes several state-of-the-art approaches that attempt to achieve this goal by exploiting the natural internal parallelism of SSDs [7, 29, 30]. Another portion of the relevant works focuses on balancing the difference between read and write speeds [23, 39, 41], whereas some recent works additionally employ the NVMe protocol to improve the efficiency of query processing [10, 31].

Simultaneously, the continuous advances in hardware technology led to the introduction of new nonvolatile memories (NVM) that further improves the characteristics of the current SSDs. 3D XPoint is a new type of NVM that features even lower latencies and higher data transfer rates. More precisely, according to [20], the current 3D XPoint-based storage devices have a read latency that is more than an order of magnitude lower than the respective of a conventional SSD. Additionally, 3D XPoint SSDs offer high throughput even at small queue depths (i.e., with a small number of pending I/O requests), as indicated by [13, 18]. In contrast, the typical SSDs exhibit better performance when the requests for data are submitted in batches [6, 10, 29].

Motivated by the efficiency of 3D XPoint memory we investigated indexing methods, that utilize both nonvolatile memory technologies, thus balancing performance and cost. In [11], we proposed a hybrid variant of Grid File (H-Grid), that employs flash as primary storage and a small amount of 3D XPoint for the hottest data. We introduced a hot region detection algorithm that identifies the most important sub-directories and data buckets. The algorithm uses the number of disk accesses to calculate a weight value for each sub-directory and data bucket. In continuing, it compares the computed values with the cumulative moving average (CMA) of the weights to locate the hottest regions. The experimental results have shown that H-Grid can reduce up to 43% the search time for a single point and up to 28% the execution time of range queries.

Moreover, in [8], we described a road-map for composing a hybrid index based on R-tree and we experimented with a simple implementation (sHR-tree), that accommodates all non-leaf nodes to a 3D XPoint SSD (Fig. 1). This design is based on the observation that the upper level nodes are accessed more often than the leaves. The experimental results demonstrated substantial improvements when 3D XPoint is utilized as sole storage medium for R-tree, e.g., a gain of up to 82% is achieved in range queries. However, the improvement for sHR-tree is marginal, since the small number non-leaf nodes is not capable of making any considerable performance contribution. This reveals the necessity for a new data access method, that stores a part of the leaf-nodes to the fast storage.

To address this problem, in this paper, we introduce a new index for storing and managing spatial data in hybrid storage scenarios. The proposed data structure, named HyR-tree (Hybrid R-tree), adopts an unsupervised learning approach that identifies the most important of its nodes and subsequently transfers them to the 3D XPoint storage. In short, the key idea is to exploit the temporal nature of the data accesses and model them as a time series. In this way, we are able to apply a wide range of time series mining unsupervised algorithms. More precisely, we record the number of the tree node accesses in the five past epochs and we assign them time-decaying weights according to the weighted moving average method. The computed weights are then used to locate the tree nodes that exhibit high probability to be re-referenced in a short period of time. In the sequel, these “hot” nodes are transferred to the high performing 3D XPoint storage with the aim of improving the efficiency of query processing.

Notice that the majority of the existing hybrid data structures migrate data to a high performance storage medium either by exploiting usage and workload statistics [4], or by applying well-established cache replacement policies such as LFU or LRU. Examples include the research works of [5, 24, 40]. In contrast, to the best of our knowledge, the proposed HyR-tree is among the first hybrid data structures that employs unsupervised learning techniques to identify the nodes of the tree to be moved to a fast storage medium. Although the proposed weighted moving average is a fairly simple unsupervised learning technique, it was proved quite efficient in our experiments. Nonetheless, the strong element of this research is that additional efforts from other researchers may derive from this point, with the aim of introducing hybrid solutions based on other more robust unsupervised methods.

The contributions of this work are summarized into the following list:

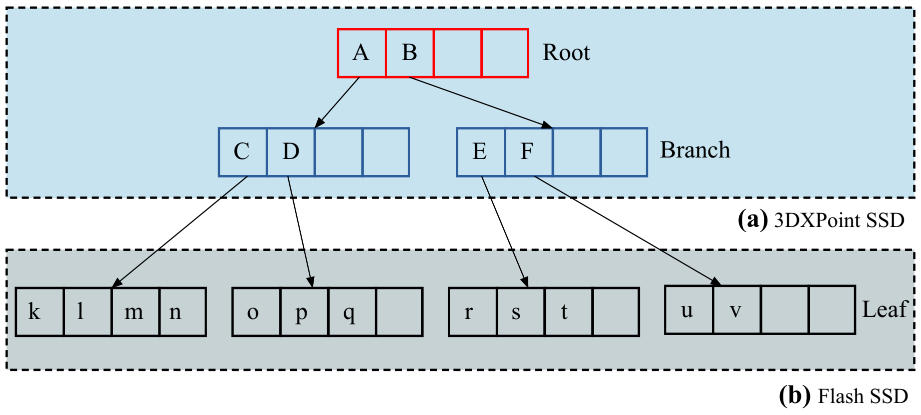


Fig. 1 A simple yet illuminating case of a hybrid R-tree index (sHR-tree); all non-leaf nodes are stored to the 3D XPoint storage

- We introduce HyR-tree (Hybrid R-tree), a new R-tree variant for efficiently managing multi-dimensional data in hybrid storage installations. The proposed data structure models the past data accesses as a time series and employs the weighted moving average (WMA) approach to assign weights and learn the most important of its nodes.
- We present a data migration policy that moves data to the high performing 3D XPoint secondary storage according to WMA. In contrast to other hybrid data structures that employ standard usage statistics, or simple LRU/LFU data migration policies, HyR-tree is among the first that bases its operation on unsupervised learning techniques.
- We experimentally evaluate our design by utilizing an NVMe flash SSD and a 3D XPoint device as testbed, and several large datasets containing up to 500 million data points. The acquired results demonstrate significant performance gains that approach 44.6% for range queries.

The remainder of this paper is organized as follows. Section 2 contains brief overviews of the nonvolatile memories, SSD devices, R-trees, and time series. The basic design and implementation details of the proposed HyR-tree are presented in Sect. 3. Furthermore, Sect. 4 describes the experimental evaluation and analyzes the obtained results. The related work on data management in hybrid storage systems is presented in Sect. 5, and finally, Sect. 6 summarizes our work and presents our future plans.

2 Overview

In this section, we discuss some preliminary elements that are necessary for the introduction of the proposed solution. The presentation is divided into three parts: Initially, in Sect. 2.1, we describe the basic features of the nonvolatile memories and we refer to their most important characteristics. In the second part, we present the traditional R-tree structure for indexing multi-dimensional data, whereas in the last part, we refer to the key elements of time series analysis that provide the tools for developing a robust variant of R-tree.

2.1 Nonvolatile memories

The term nonvolatile memory (NVM) refers to storage devices that are capable of retaining their content even after the interruption of power. Solid-state drives (SSDs) are the most popular example of utilizing NVMs and are primarily implemented by employing NAND flash. According to this technology, the data are persistently stored in the device via the charging of flash cells. The capacity of a flash cell is determined by different voltage thresholds that can be applied to it and ranges from 1 (SLC NAND) to 4 bits (QLC NAND).

The architecture of flash-based storage includes an hierarchy of storage units, from the elementary to the more complex ones. More precisely, the fundamental storage unit is the *page*, an organization of flash cells that constitutes the smallest readable/writable unit in an SSD [27]. On the other hand, a collection of pages is called a *block*, the smallest erasable unit of an SSD. As we move toward the higher levels of the hierarchy, a number of blocks accompanied by some additional special registers assemble a *plane*, and a series of planes compose a *die (chip)*. Finally, two or more dies are combined to form a flash *package*. Figure 2 illustrates the aforementioned hierarchy by depicting the main components of a flash package with two dies.

One of the most interesting properties of flash is its internal capability for parallelism. In fact, there are two levels of parallelism in a flash package. The first one is called *plane-level parallelism* and derives from the capability of the planes in a die to execute the same command simultaneously [7, 14]. Furthermore, the dies are allowed to perform different operations (read, program, erase) at the same time; this property is frequently encountered as *package-level parallelism* in the relevant literature [29].

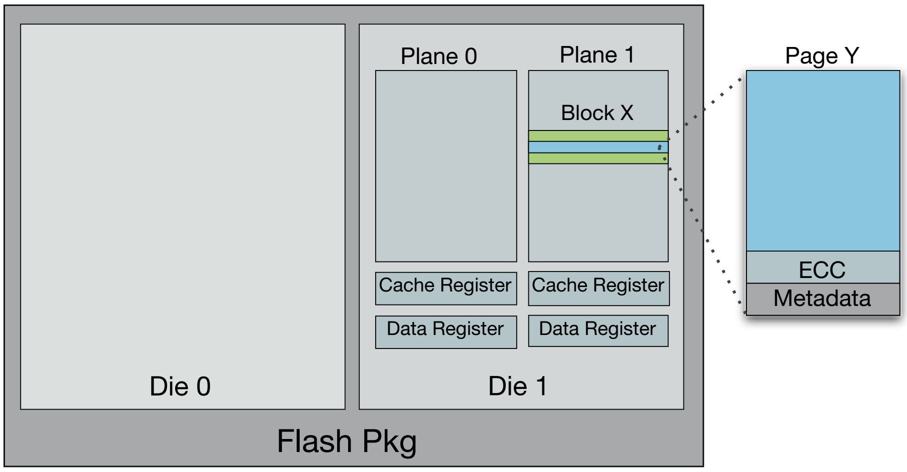


Fig. 2 The main components of a flash package with two dies

Another feature of NAND flash SSDs (Fig. 3) lies in the speed difference of the various operations. More specifically, writes are considerably slower than reads, whereas deletions are even slower. In addition, the successive program and erase operations have a negative impact on the integrity of flash cells and they lead to increased error rates that subsequently decrease the performance of the device. If such a damaged block exhibits prohibitively high error rates, it is marked as defective by the system and it is excluded from further usage. For this reason, manufacturers often provide with extra space their drives.

In this paper, we focus on a recently introduced type of NVM, i.e., *3D XPoint*. 3D XPoint was developed by Intel and Micron primarily to bridge the gap between the characteristics (latency times and density) of the main DRAM memories and the ones of the flash-based storage devices. In 3D XPoint memories, the cells are organized in layers and each cell can store one bit of information. The primary difference between 3D XPoint and flash NVM lies into the capability of the former to perform in-place writes, allowing its usage as a main memory unit. This property increases the usefulness of 3D XPoint, since it enables its exploitation as a both primary and secondary storage system and also, as a caching mechanism for other types of secondary storage [13].

2.2 R-tree

R-trees are indexes designed for storing and accessing spatial data. They were proposed by [12] and since then, they have been broadly used in a wide variety of applications including geographical and navigation systems, databases, handling of multi-dimensional feature vectors in machine learning algorithms, multimedia applications, and so on [33]. R-trees have been proved quite efficient in nearest neighbor query processing, including great-circle

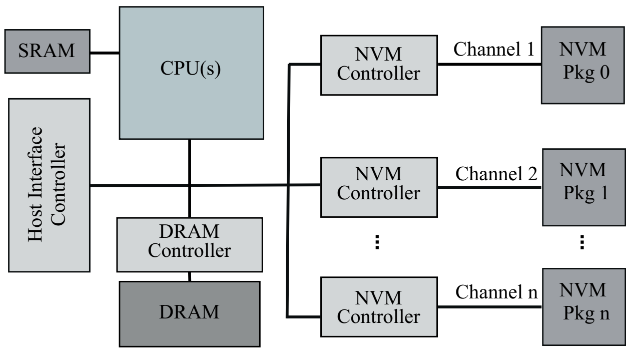


Fig. 3 An SSD drive

distance [34]. A considerable wealth of scientific research has focused on the proposal of numerous R-tree variants with the aim of improving its performance on specific applications [26].

The data structure operates by creating groups of proximal records and in the sequel, it represents these records by using their minimum bounding rectangle. The hierarchy within an R-tree is created by storing one such rectangle in each node of the tree, in a manner that the rectangle of a node encloses all the bounding rectangles of its children. Consequently, it is possible that a record belongs to multiple rectangles; nevertheless, it is always associated with only one of them.

The rationale of this functionality is that in the case a query does not intersect a bounding rectangle, then it also cannot intersect any of the contained records. In this context, a look-up operation for an object S within the R-tree begins by accessing the root, and then it follows multiple paths toward the leaves to ensure the existence of S or not. In the worst case, these multiple traversals retrieve a small number of objects with a cost that is linear to the size of the data.

R-trees share some common properties with B-trees: They are balanced search trees (all leaf nodes reside in the same depth) and use pages to accommodate the data. If an R-tree is stored on disk, then these pages correspond to disk pages and the number of records that can be stored in them ranges from m to M, where $m = M/2$. Similar to the majority of the tree structures, the height is an important performance factor, since it affects the number of pages that will be accessed during a search operation. The maximum height $h_{\max}$ of an R-tree is given by the following equation:

$$h_{\max} = \log_m N - 1, \quad (1)$$

where N represents the total number of the bounding rectangles that are stored within the data structure.

A typical R-tree with $m = 2$ and $M = 4$ is presented in Fig. 4. The tree stores 12 objects (not depicted in the figure), accommodated at its leaf nodes, whereas its height is equal to 3. According to our previous discussion, there exist 12 elementary rectangles (from k to v) each one enclosing one of 12 aforementioned objects. In addition, the internal nodes of the structure store four larger rectangles (from C to F) which store the most proximal objects. Finally, these four groups are in turn placed into two larger groups; the respective rectangles are placed on the root of this R-tree.

Notice that the final form of the tree is greatly affected by the order of the operations. Consequently, in the case where the insertions of the objects are performed in a different order, then different R-trees will derive. The effectiveness of this data structure in indexing spatial objects has triggered a significant amount of research toward its improvement and specialization for multiple types of applications. In short, some indicative variants include the R$^+$-tree introduced by [35], R*-tree by [2], Hilbert R-tree by [17], Cubetree by [32], Historical R-tree by [36], and LR-tree by [3].

2.3 Time series forecasting

Time series is a term referring to a sequence of objects sorted in chronological order. These objects may be any type of information with temporal characteristics, including web site traffic, stock market prices, scheduled temperature recordings for a location, blood pressure measurements, and more generally, values recorded in specific (and usually equal) time intervals. In the particular occasion that we examine in this article, the objects are chucks of data requested from the secondary storage. As we shall discuss shortly in Sect. 3, we decided to model these requests by utilizing time series, with the aim of discovering potentially repeated patterns over time. These patterns will assist us in the prediction of the most appropriate data which will be stored and retrieved to/from a 3D XPoint SSD device.

Time series data can be easily visualized by plotting the respective line charts. With these tools, the data are analyzed by employing methods which are frequently called *time series analysis*. These methods attempt to identify the most significant properties of the temporal data, and then, exploit these properties to predict future values based on previously observed values. Therefore, an entire family of algorithms known as *time series forecasting algorithms* has been proposed in the relevant literature.

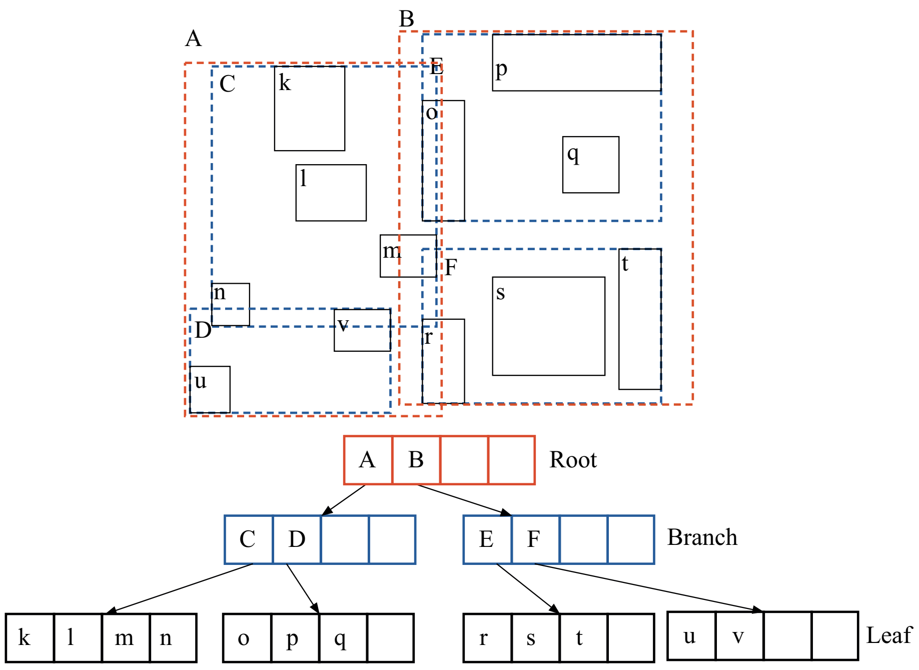


Fig. 4 An R-tree with 12 records, 18 bounding rectangles, $h = 3$, $m = 2$, and $M = 4$

The most elementary approach to the prediction of a future value based on the values of the present and past values of a time series is the computation of their average value. However, this simplistic method is sensitive to short-term fluctuations of the values and it is considered rather unreliable. A common workaround to this problem is the calculation of the *simple moving average (SMA)*. This is performed by sliding a fixed-length window over the data points of the series, and then, by computing the mean of the values that are enclosed by this window. In other words, as the window slides, the most recent values are included in the computation of the mean, whereas the older ones are dropped. In this way, the short-term fluctuations are smoothed out and the longer-term patterns are highlighted.

In this work, we employ a variant of *SMA* method, which is known as *weighted moving average, or WMA*. WMA calculates the average of the k most recent points, multiplying each data point with a weight value. Though it depends on the application, it is usual to assign higher values to the weights associated with the most recent observed values.

3 The HyR-tree

The low latency and the high IOPS of 3D XPoint even at small queue depths motivated us to develop a multi-dimensional point access method that efficiently exploits hybrid I/O [11]. Continuing our research in the same direction, we presented a proposal for a spatial access method based on R-tree in [8]. In this work, we present HyR-tree, a hybrid index that profits from both NVM technologies (flash and 3D XPoint). In the following, we describe its main features and we give some implementation details.

3.1 Overview of HyR-tree

A logical representation of HyR-tree is depicted in Fig. 5. The internal nodes (Fig. 5a) and a selected small part of the leaf nodes (Fig. 5b) are stored in the performance tier, that is constituted by a 3D XPoint device, while the vast majority of the leaves (Fig. 5c) is persisted in the storage tier, that is a flash SSD. The idea behind this design is to move the high priority data to the 3D XPoint device. Therefore, careful selection of the leaf nodes for the 3D XPoint is necessary. To attain this, HyR-tree monitors all read requests and automatically identifies hot regions, i.e., tree nodes that have high probability to be referenced in a short period of time. Only these nodes are gradually migrated to the performance tier.

We use two values to decide whether a leaf node will be migrated or not; these are denoted as S and T. Specifically, S is a score that is assigned to each leaf node. S is calculated by using the number of previous node accesses. On the other hand, T is a threshold that controls the number of migrated nodes.

As we will see shortly, our algorithm requires the computation and storage of various statistics and scores for each leaf node.

For this reason, we introduce an additional auxiliary data structure, named *Node Score Table* (*NCT*), which maps each leaf node to a structure that contains (i) the number of node accesses for the current epoch, (ii) a list that holds node accesses for past epochs, and (iii) the node's score S. The node access list and the score are updated at the end of each epoch. NCT resides in the main memory and is implemented as a hash table permitting fast lookups.

HyR-tree employs also a small in-memory buffer to keep the recently accessed nodes. We consider LRU as eviction policy for the buffer, however, different policies can be utilized. As a result, the leaves of the tree can reside in either the main memory, or the flash, or the 3D XPoint storage. On the other hand, upper level nodes can be in the main memory or the 3D XPoint. We assume that each tree node corresponds to one disk page.

In the following subsection, we describe the migration policy of HyR-tree.

3.2 Data placement in HyR-tree

According to the previous discussion, the upper level nodes of HyR-tree are placed in the fast 3D XPoint storage. However, the leaves can reside either on 3D XPoint or flash, with their position depending on their popularity. The popularity of a leaf node is determined by a weight-based policy for the identification of the nodes that are most likely to be accessed in the future. This policy is inspired by the weighted moving averages method in time series. Thus, we record the number of accesses of each leaf for the t most recent epochs (e.g., $t = 5$ in Fig. 6). Each epoch may represent a period of time or a predefined number of I/O requests. We currently use the second option in our implementation. At the end of an epoch, we calculate a score S^{ρ}_{t+1} for each leaf node ρ according to the equation given as follows:

$$S^{\rho}_{t+1} = \sum_{i=0}^{t-1} \frac{2(t-i)}{t(t+1)} Y_{t-i} \qquad (2)$$

where Y_{t-i} denotes the number of read requests to ρ, at the epoch $t-i$, with $i = 0, \ldots, t-1$. This policy assigns higher weights to the most recent values and lower to the old ones, implementing in practice an online learning process driven of the most recent values.

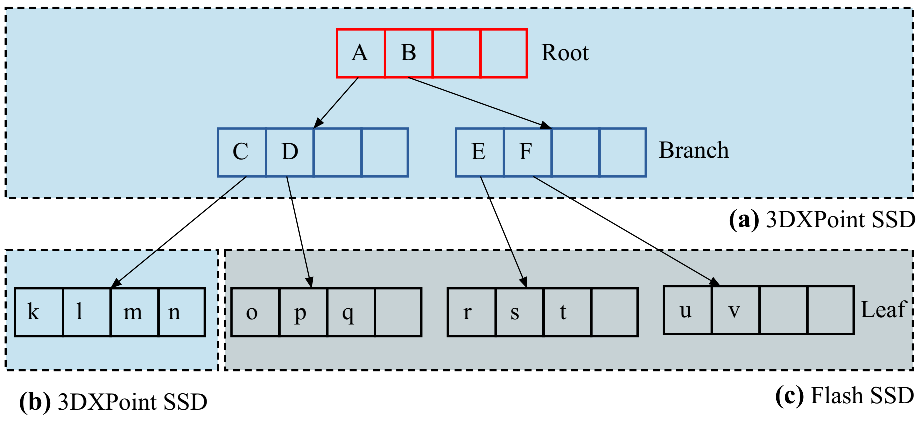


Fig. 5 HyR-tree; a part of the tree is stored to the 3D XPoint SSD

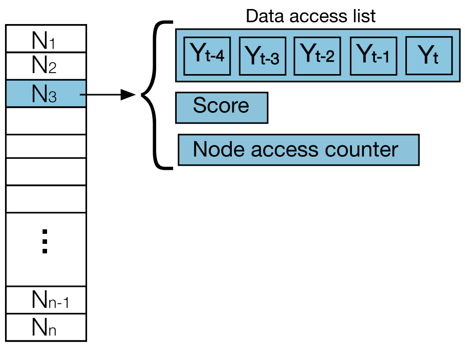


Fig. 6 The Node Score Table. It holds the score for each leaf node

Algorithm 1 describes how the scores and the threshold T are determined. At the end of an epoch, we process all nodes in *NCT* updating (push) *data access lists* with the respective epoch's I/O counter values (*ctn*) (line 3). If a list is larger than t, we discard (pop) the oldest value (lines 5–7). We calculate the score S according to Eq. 2 (line 8) and for each node, we update its corresponding record in *NCT* (line 9). We also store S to a score array (line 10). In the sequel, we sort the score array and we calculate the Nth percentile value of the scores. Finally, we set this value as threshold T for the next epoch.

Algorithm 2 details the eviction policy, which releases space from the memory buffer for a newly created node, or a node read from secondary storage. During eviction modified (dirty) nodes are persisted to the secondary storage, while the unmodified ones (clean) are simply discarded from the main memory. We adapt the LRU eviction policy to the hybrid storage of HyR-tree, so that each dirty evicted node to be written to the appropriate storage. Lines 2–7 of the algorithm actually apply the proposed placement policy, locating the nodes that are eligible for migration. In particular, a leaf node n of the tree is candidate for migration if it is (i) located at the LRU position of the in-memory buffer (namely it is the oldest element of the buffer), and (ii) written in the flash storage. In this case, its score (in NCT) is compared with threshold T and if the score is higher than T, then the node is stored to the 3D XPoint and its state is set to clean. If node is dirty (lines 8–18), we check whether is an upper level node (lines 15–17) or was previously migrated to 3D XPoint, where in these cases, the node is written to 3D XPoint; otherwise, it is persisted in flash.

```
Algorithm 1: Calculate Migration Threshold($NCT$)
    Data: $NCT$ the node score table, $t$ the number of epochs
    Result: the migration threshold for the next epoch $T$
 1  if epoch have been elapsed then
 2      foreach node n in NCT do
 3          $NCT[n].DataAccessList.push(NCT[n].ctn)$;
 4          $NCT[n].ctn = 0$;
 5          if $NCT[n].DataAccessList.size > t$ then
 6              $NCT[n].DataAccessList.pop()$;
 7          end
 8          calculate score $S$ (Eq. 2);
 9          $NCT[n].score \leftarrow S$;
10          scoreArray.push(score);
11      end
12      sort(scoreArray);
13      $T = N^{th} percentilevalue(scoreArray)$
14  end
15  return $T$
```

```
Algorithm 2: Evict($MB, NCT, T$)
    Data: $MB$ the main buffer, $NCT$ the node score table, $T$ the threshold for the
          current epoch
 1  select victim node $n$ from the LRU position of the $MB$;
 2  if node n is leaf AND n is in FLASH then
 3      if $NCT[n].score > T$ then
 4          write $n$ to 3DXPOINT;
 5          set $n$ clean;
 6      end
 7  end
 8  if n is dirty then
 9      if n is leaf then
10          if n is in 3DXPOINT then
11              write $n$ in 3DXPOINT ;
12          else
13              write $n$ to FLASH;
14          end
15      else if n is upper level node then
16          write $n$ in 3DXPOINT;
17      end
18  end
19  evict $n$;
```

3.3 Node retrieval

Queries like range searches are fundamental in spatial data processing. Given a rectangle Q, a range query returns all rectangles in R-tree that intersect with Q. Therefore, such queries usually involve large numbers of node retrievals. HyR-tree improves query performance by placing the most popular nodes at the highly efficient secondary storage. In algorithm 3, we describe how the node fetching operation is executed in HyR-tree. Recalling from the above, HyR-tree incorporates a small main memory buffer. Thus, if the requested node N is already in the main memory, it is moved to the most recently used (MRU) position of the buffer (lines 1–2). In a different case, a fetch operation from secondary storage (flash or 3D XPoint) is needed to retrieve the node (lines 3–9). At the end of the fetching process, N is placed in the main buffer (MRU position).

Assuming that the nodes of HyR-tree occupy a single page in the secondary storage, the cost of fetching an upper level node n is $C_n = R_x$ where R_x is the reading cost from 3D XPoint. Now let $x_n \in \{0, 1\}$ be a variable that denotes whether n is stored in the 3D XPoint storage or not. In the case that n is a leaf, then the cost of retrieving n is given by the following formula:

$$C_n = x_n R_x + R_f(1 - x_n) = R_f - x_n(R_f - R_x) \tag{3}$$

where R_f is the reading cost from flash.

Algorithm 3: RetrieveNode(n, MB)

```
Data: $n$ the node to be retrieved, the in-memory buffer MB
Result: the node $n$
1  if $n$ is in the in-memory buffer $MB$ then
2      move $n$ to the MRU position of $MB$;
3  else if $n$ is in FLASH then
4      read $n$ from FLASH;
5      move $n$ to the MRU position of $MB$;
6  else
7      read $n$ from 3DXPOINT;
8      move $n$ to the MRU position of $MB$;
9  end
10 return $n$
```

4 Experimental evaluation

4.1 Methodology and setup

In this section, we discuss the performance evaluation of HyR-tree for various storage configurations. We conducted a series of experiments by employing both flash and 3D XPoint storage devices. All the utilized devices were real. Here, we aim at unfolding the benefits of HyR-tree, against an R-tree implementation that utilizes a single storage medium, and against two HyR-tree's variants. The first variant, (sHyR-tree), stores only the upper level nodes to the 3D XPoint storage, while the second one (rHyR-tree) randomly selects leaf nodes for migration. To ensure fairness, rHyR-tree persists the same amount of data with HyR-tree in the fast 3D XPoint SSD. We evaluate two different workloads, regarding (i) index construction and (ii) execution of 5000 range queries.

We utilized four datasets in the conducted experiments: two real-world and two synthetic. The real datasets contain 300 and 500 million two-dimensional geographical points (latitude and longitude) in the globe. These datasets are publicly available on the official Open Street Map project page.[1] Regarding the synthetic datasets, both contain 20 million two-dimensional records which have Gaussian and uniform distributions, respectively.

The experiments were conducted on a workstation with a 4-core CPU (Intel Xeon E3-1245 v6 3.70 GHz) and 16 GB of main memory, running CentOS Linux 7. The OS was hosted on a separate SATA SSD. A flash NVMe SSD (Intel DC P3700) and a 3D XPoint counterpart device (Optane memory series 32 GB) were employed for the experiments. The performance characteristics of the two storage devices are listed in Table 1.

All the experiments were executed using the Direct I/O (O_DIRECT) to bypass Linux OS caching layer. The total size of the in-memory buffer (MB) was configured to 4 MB. We set the epoch at 500 K node reads for the runs using the 500 M point dataset, and at 250 K node reads for the rest test cases. The migration threshold T was set differently for each dataset type. Specifically, for the 300 M real dataset, we selected a threshold the 99th percentile of the scores in each epoch, for the bigger 500 M dataset the 97th percentile, while for the smaller synthetics, the 95th percentile, respectively. These threshold values resulted in migrating approximately one-third of the nodes to 3D XPoint.

The results are presented in the following paragraphs. The single hatched section of each bar corresponds to the I/O time in the flash SSD, while the double hatched section represents the I/O time in the 3D XPoint SSD.

4.2 Index construction

In this subsection, we discuss the performance of the index construction process. Specifically, we examine five different index construction test cases: (i) R-tree on flash, (ii) R-tree on 3D XPoint, (iii) sHyR-tree, (iv) rHyR-tree, and (v) HyR-tree. In Fig. 7, we present the execution times for three different page sizes (4, 8, and 16 KB). In the following discussion, we consider the execution of the flash SSD as the baseline case.

As expected, the best results were obtained when the 3D XPoint SSD was utilized as a sole storage medium. However, index construction with 500 M points failed, because the index size (42 GB) exceeded the storage capacity.

[1] http://spatialhadoop.cs.umn.edu/datasets.html.

Table 1 SSD Specification, from manufacturer's datasheets

	Intel DC P3700 (Flash)	Optane memory series (3D XPoint)
Seq. read	Up to 2700 MB/s	Up to 1350 MB/s
Seq. write	Up to 1100 MB/s	Up to 290 MB/s
Random read	450 K IOPS	240 K IOPS
Random write	75 K IOPS	65 K IOPS
Latency read	120 μs	7 μs
Latency write	30 μs	18 μs

Compared with the baseline case, the execution time was improved up to 57% for the 300 M real dataset, and up to 69% and 68% for the Gaussian and uniform datasets, respectively. Similarly, the construction of the sHyR-tree attains a satisfactory improvement. In particular, the process achieves a gain of up to 20% for the real 500 M point dataset, and up to 21% for the two synthetic ones, compared with the baseline.

Our proposed method, i.e., HyR-tree, exhibits a significant performance gain that ranges between 18 and 21% for the real 300 M point dataset, and between 24 and 25% for the 500 M point one. Regarding the synthetics, the gain is up to 40% in the Gaussian, and up to 39% in the uniform dataset, respectively. HyR-tree and its random variant (rHyR-tree) exhibit similar behavior, which is expected to an extent, since both persist the same amount of data in each SSD. Remarkably, the index construction in the real dataset was completed in a shorter time than the two synthetic ones, although its size is considerably larger. This occurred because the objects in the real dataset exhibit spatial locality, which results in a high number of cache hits. We also observe that the I/O time increases with respect to the page size. This is expected up to a point, since the page size determines the amount of data written at each time.

4.3 Range queries

Next, we examine the performance of the various scenarios in the execution of range queries. The results of this experiment are illustrated in Fig. 8. The first observation concerns the huge performance difference between the tested cases where the R-tree is stored on the flash SSD and the 3D XPoint SSD. More specifically, in the latter case, an improvement up to 82% is attained in comparison with the flash SSD execution.

The performance benefits of using a sHR-tree are marginal in the test cases using the three smaller datasets. This is mainly explained by the fact that the amount of the I/O operations on the fast 3D XPoint storage is not enough to make essential performance contribution. On the other hand, a gain up to 10% is achieved in the case of the 500 M geographical points. This means that even a simple approach as the sHR-tree can be satisfactory when large data are involved.

HyR-tree confirmed its design hypothesis [8] that a hybrid index identifying and storing the hottest regions on a 3D XPoint-based storage can significantly accelerate query performance. More precisely, our proposed index improves the processing times of the submitted range queries by a margin that ranges from 35 to 42% in the real 300 M point dataset, and from 42 to 56% in the 500 M points one. Similarly, the gain for the Gaussian dataset is up to 44.6% and for the uniform one is up to 41%.

Comparing HyR-tree with its random version, HyR-tree is 17–25% faster in the experiment with the 300 M real dataset and 9–20% in the experiment using the 500 M point dataset. On the other hand, both indexes provide similar results in the test cases employing the smaller synthetic datasets. The acquired results indicate that the performance of HyR-tree depends on the size and type of data. Moreover, parameters like epoch size and the threshold T (Nth percentile) influence performance and should be further investigated.

5 Related work

During the past few years, the design and implementation of effective and efficient data structures for NVMs have attracted the attention of the researchers. Specifically, [9] provided a detailed overview of the implementations of 62 indexes in flash storage devices. A significant amount of research works focused on the introduction of solutions that combine the high data transfer rates of the SSDs in comparison with the low cost of the traditional magnetic hard drives (HDDs). The outcome of these works led to hybrid storage scenarios in numerous data management systems; a survey of the state-of-the-art architectures and algorithms was presented by [28].

The two primary architectures for hybrid storage systems either propose the movement of the most frequently

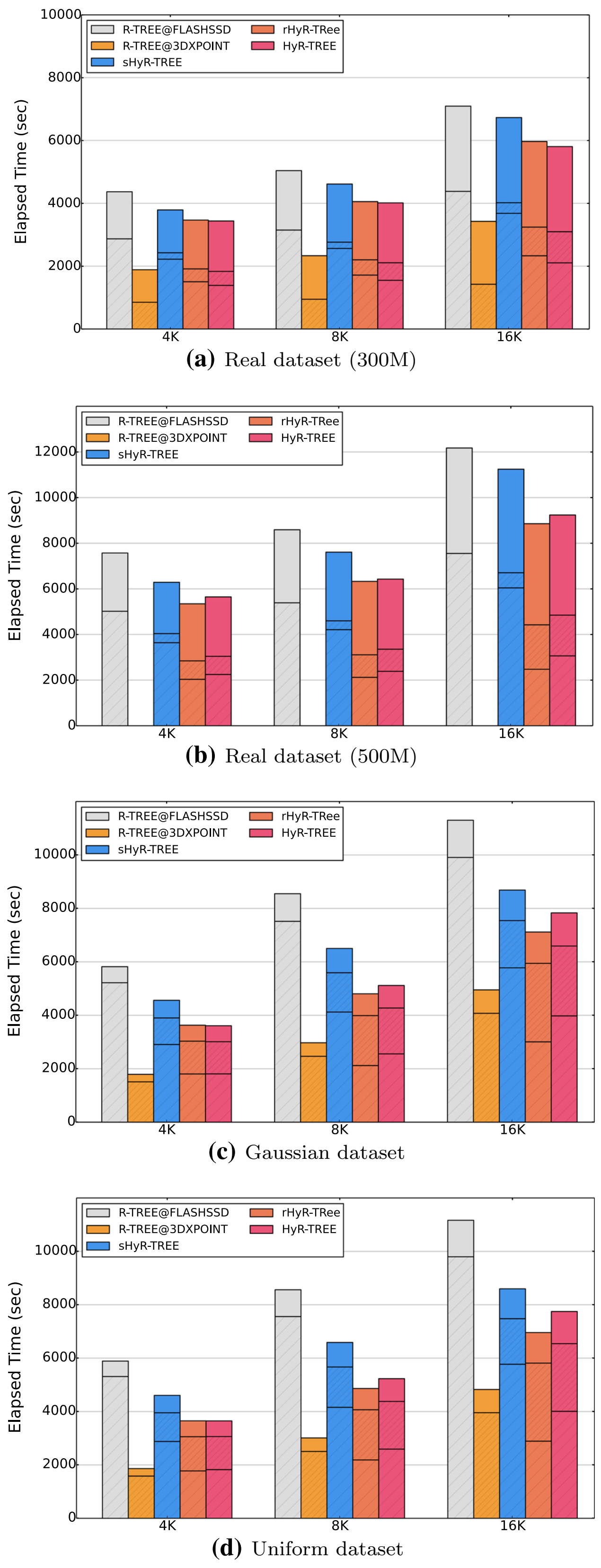


Fig. 7 Execution times of index construction. HyR-tree achieves a gain up to 25% for the real-world datasets, and up to 40% for the synthetics, compared with R-tree execution on the flash SSD. The 500 M dataset **b** failed to run using 3D XPoint as single storage, because index size exceeded device's capacity

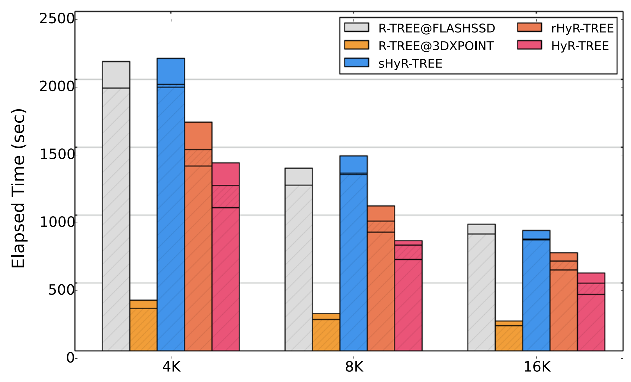
used data to SSDs (leaving the rest of the data in the cheaper and slower HDDs) or transform the SSDs into a caching mechanism that stands between the main memory and HDDs. For example, the works of [4, 40] suggested robust migration policies for transferring the most recently/frequently used data to SSDs in database management systems (DBMSs). In addition, the work of [4] considered several usage and workload statistics and introduced a utility that places the hottest database objects (e.g., tables) to the SSDs. Regarding the relational DBMSs, [40] proposed a method that analyzes the submitted SQL queries and transfers the most frequently used objects to SSDs.

Regarding the utilization of SSDs as a caching mechanism, [5] employed these devices as a write cache between main memory and HDDs. This work adopts the well-known LRU and LFU eviction policies for the suggested cache and applies them when the storage becomes full. Furthermore, [24] investigated the possibility of utilizing SSDs for the caching requirements of the DBMSs. They introduced an algorithm that is based on the properties of the various devices that participate in a hybrid storage scheme and it also includes a page eviction policy.

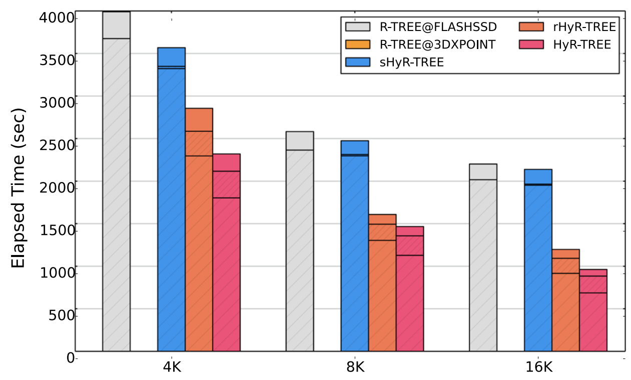
One of the most popular data structures employed by the majority of DBMSs is the B-tree and its variants. Nevertheless, [16] showed that on hybrid storage systems, the performance of their original forms can be significantly improved. In this work, the authors introduced the Hybrid B-tree, a data indexing technique that is more suitable on these cases, since it offers improved performance and reduced random write operations. In particular, Hybrid B-tree is aware of the medium that stores its nodes. All its internal nodes are stored in SSD devices, whereas its leaves are spanned across both the SSD and HDD drives.

Another work that explored several performance issues in DBMSs was conducted by [44]. More precisely, the authors highlighted write amplification, bad usage of temporary tables, and buffer pool cache misses as three crucial parameters that affect the efficiency of query execution in a negative manner. In their experimental evaluation, they demonstrated the speed superiority of 3D XPoint in query processing over the standard SSD devices.

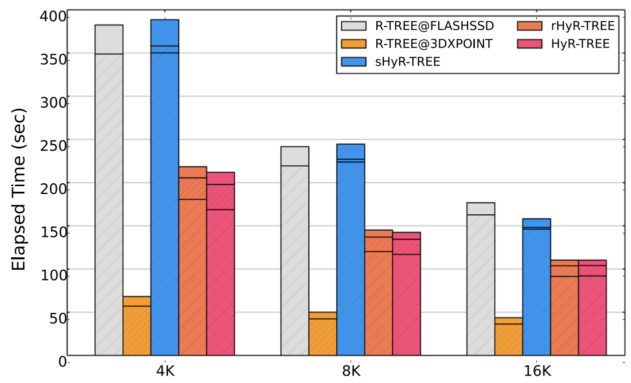
An interesting branch of the relevant research includes the multi-armed bandit methods, that attempt to optimally make decisions, while they simultaneously learn new knowledge [21, 25]. These methods fall into the broad category of reinforcement learning and have been widely



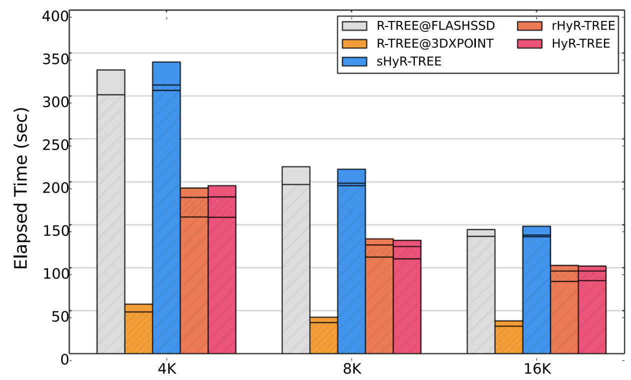
(a) Real dataset (300M points)



(b) Real dataset (500M points)



(c) Gaussian dataset



(d) Uniform dataset

◀**Fig. 8** Execution times of range queries for different workloads. HyR-tree achieves an improvement up to 56% for the real-world datasets and up 44.6% to for the synthetics, compared with R-tree execution on the flash SSD

adopted in addressing collaborative filtering problems such as in recommender systems [22] and P2P networks [19], as well as in cache replacement strategies [38]. The application of a multi-armed bandit model for identifying repeated data access patterns in HyR-tree constitutes an attractive alternative to the proposed moving average approach.

Finally, there are alternative spatial data structures that offer guaranteed worst-case performance for range queries. A representative example is the external memory range tree of [1], which improved the duration of query processing for three and higher dimensional orthogonal range reporting. The authors of this work proposed a dimensionality reduction method via projection, and they introduced a method that outperformed other schemes in both the pointer machine and I/O models. Moreover, [37] proposed data structures for external memory range searching in two and three dimensions. These structures were based on specific manifolds that partition space into regions, based on the output size of queries at points within the space. However, these data structures are difficult to implement, so in practice, their usage scenarios are limited compared with R-tree.

6 Conclusions and future work

In this paper, we highlight the large value of contemporary nonvolatile memory technologies in data management. Continuing our previous work on hybrid spatial indexes, we introduce HyR-tree, an R-tree variant for compound flash-3D XPoint storage installations.

The key element in HyR-tree is that it models the submitted data requests as time series and employs the weighted moving average approach to detect repeated access patterns. In this way, it is capable of identifying the hottest tree nodes. By subsequently transferring these nodes to the faster 3D XPoint storage medium, it achieves substantial improvements to its overall performance. To the best of our knowledge, this is the first R-tree variant that employs unsupervised machine learning techniques to combine the benefits of a hybrid storage configuration.

We evaluated our approach through a series of experiments by using four datasets, two real, and two synthetic ones. We considered five different cases, namely (i) R-tree on flash, (ii) R-tree on 3D XPoint, (iii) a simple hybrid R-tree implementation (sHyR-tree), (iv) HyR-tree with random node placement (rHyR-tree), and (v) HyR-tree.

The experimental results supported our design hypothesis. More specifically, we measured the performance of two basic operations, namely the index construction and the execution of range queries.

Regarding the first operation, the R-tree execution exclusively on the 3D XPoint device (best case) improved index construction by up to 69% compared with the flash SSD execution. Similarly, the sHR-tree achieved improvements of up to 21%, whereas the gain for HyR-tree was up to 40%. In the second case, i.e., the execution of 5000 range queries, the best results were obtained when the R-tree was stored on a single 3D XPoint SSD, gaining up to 82%. Our proposed HyR-tree improved range query execution by up to 56% in the real 500 M world dataset, and up to 44.6% in the two synthetic ones. On the other hand, the sHR-tree achieves remarkable gains up to 10% only in the 500 M point cases. This is due to the higher number of internal nodes hosted in the high-performance storage. Thus, is clear that even a small amount of 3D XPoint can substantially improve performance at affordable costs.

The obtained results unfold the value of hybrid indexes in spatial data processing, but definitely there is room for improvement. Our future work plans include the investigation of additional learning methods for hot node detection. The study of reinforcement learning techniques, like the multi-armed bandit method, is in our priorities. Specifically, data placement can be modeled as an online learning problem associated with a reward credited in each page read. Thus, a page can be migrated to the fast storage either by considering the received rewards (exploitation) or based on a random selection (exploration). Another interesting research direction is the study of additional query types like kNN, Joins, etc. Last but not least, a cooling process that periodically moves nodes with declining popularity back to flash is also in our plans.

Our work shows that combining traditional data structures with machine learning techniques and modern hardware can create new lines of research. The first 3D XPoint products for the memory bus (Optane DC Persistent Memory) are already in the market, enabling many new applications. These memory modules can operate either as low-cost volatile alternative of DRAM, or as a new persistent memory layer, or as high performing block storage. The latest option eliminates the latency of data transfers through the I/O bus and allows existing applications to use them as a block storage medium. As a result, new file systems for hybrid memory configurations have been introduced [43] lately. Different evaluations have shown that using Optane DC PM as secondary storage provides several times higher read performance compared with conventional flash and 3D XPoint SSDs [15, 42]. As the performance gap between the two storage layers is widened, we can safely conclude that hybrid indexes can be further benefited, and new research challenges arise.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Afshani P, Arge L, Larsen KD (2009) Orthogonal range reporting in three and higher dimensions. In: Proceedings of the 50th annual IEEE symposium on foundations of computer science, pp 149–158
2. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on management of data, pp 322–331
3. Bozanis P, Nanopoulos A, Manolopoulos Y (2003) LR-tree: a logarithmic decomposable spatial index method. Comput J 46(3):319–331
4. Canim M, Mihaila GA, Bhattacharjee B, Ross KA, Lang CA (2009) An object placement advisor for DB2 using solid state storage. Proc VLDB Endow 2(2):1318–1329
5. Canim M, Mihaila GA, Bhattacharjee B, Ross KA, Lang CA (2010) SSD bufferpool extensions for database systems. Proc VLDB Endow 3(1–2):1435–1446
6. Carniel AC, Ciferri RR, Ciferri CD (2018) A generic and efficient framework for flash-aware spatial indexing. Inf Syst 52:102–120
7. Chen F, Hou B, Lee R (2016) Internal parallelism of flash memory-based solid-state drives. ACM Trans Storage 12(3):13
8. Fevgas A, Akritidis L, Alamaniotis M, Tsompanopoulou P, Bozanis P (2019) A study of R-tree performance in hybrid Flash/3D XPoint storage. In: Proceedings of the 10th international conference on information, intelligence, systems and applications (IISA), pp 1–6
9. Fevgas A, Akritidis L, Bozanis P, Manolopoulos Y (2020) Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. VLDB J 29(1):273–311
10. Fevgas A, Bozanis P (2019) LB-Grid: an SSD efficient grid file. Data Knowl Eng 121:18–41. https://doi.org/10.1016/j.datak.2019.04.002
11. Fevgas A, Bozanis P (2019) A spatial index for hybrid storage. In: Proceedings of the 23rd international database applications and engineering symposium, pp 1–8
12. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data, pp 47–57
13. Hady FT, Foong A, Veal B, Williams D (2017) Platform storage performance with 3D XPoint technology. Proc IEEE 105(9):1822–1833
14. Hu Y, Jiang H, Feng D, Tian L, Luo H, Zhang S (2011) Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In: Proceedings of the 25th international conference on supercomputing, pp 96–107
15. Izraelevitz J, Yang J, Zhang L, Kim J, Liu X, Memaripour A, Soh YJ, Wang Z, Xu Y, Dulloor SR, et al (2019) Basic performance measurements of the intel optane DC persistent memory module. arXiv:1903.05714

16. Jin P, Yang P, Yue L (2015) Optimizing B+-Tree for hybrid storage systems. Distrib Parallel Databases 33(3):449–475
17. Kamel I, Faloutsos C (1994) Hilbert R-tree: an improved R-tree using fractals. In: Proceedings of the 20th international conference on very large data bases, pp 500–509
18. Koltsidas I, Hsu V (2017) IBM storage and NVM express revolution. Technical reports, IBM
19. Korda N, Szorenyi B, Li S (2016) Distributed clustering of linear bandits in peer to peer networks. In: Proceedings of The 33rd international conference on machine learning, pp 1301–1309
20. Kourtis K, Ioannou N, Koltsidas I (2019) Reaping the performance of fast NVM storage with uDepot. In: Proceedings of the 17th USENIX conference on file and storage technologies, pp 1–15. Boston, MA
21. Li S (2016) The art of clustering bandits. Ph.D. Thesis, Università degli Studi dell'Insubria
22. Li S, Karatzoglou A, Gentile C (2016) Collaborative filtering bandits. In: Proceedings of the 39th international ACM SIGIR conference on research and development in information retrieval, pp 539–548
23. Lin S, Zeinalipour-Yazti D, Kalogeraki V, Gunopulos D, Najjar WA (2006) Efficient indexing data structures for flash-based sensor devices. ACM Trans Storage 2(4):468–503
24. Liu X, Salem K (2013) Hybrid storage management for database systems. Proc VLDB Endow 6(8):541–552
25. Mahadik K, Wu Q, Li S, Sabne A (2020) Fast distributed bandits for online recommendation systems. In: Proceedings of the 34th ACM international conference on supercomputing, pp 1–13
26. Manolopoulos Y, Nanopoulos A, Papadopoulos AN, Theodoridis Y (2010) R-trees: theory and applications. Springer, Berlin
27. Micheloni R (2016) 3D flash memories. Springer, Berlin
28. Niu J, Xu J, Xie L (2018) Hybrid storage systems: a survey of architectures and algorithms. IEEE Access 6:13385–13406
29. Roh H, Park S, Kim S, Shin M, Lee SW (2011) B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. Proc VLDB Endow 5(4):286–297
30. Roh H, Park S, Shin M, Lee SW (2014) MPSearch: multi-path search for tree-based indexes to exploit internal parallelism of flash SSDs. IEEE Data Eng Bull 37(2):3–11
31. Roumelis G, Vassilakopoulos M, Corral A, Fevgas A, Manolopoulos Y (2018) Spatial batch-queries processing using xBR+-trees in solid-state drives. In: Proceedings of the 8th international conference on model and data engineering, pp 301–317
32. Roussopoulos N, Kotidis Y, Roussopoulos M (1997) Cubetree: organization of and bulk incremental updates on the data cube. ACM SIGMOD Record 26(2):89–99
33. Samet H (1990) Applications of spatial data structures: Computer graphics, image processing, and GIS. Addison-Wesley Longman Publishing Co., Inc., USA
34. Schubert E, Zimek A, Kriegel HP (2013) Geodetic distance queries on R-trees for indexing geographic data. In: Proceedings of the 16th international symposium on spatial and temporal databases, pp 146–164
35. Sellis T, Roussopoulos N, Faloutsos C (1987) The R+-tree: a dynamic index for multi-dimensional objects. In: Proceedings of the 13th VLDB conference, pp 507–518
36. Tao Y, Papadias D (2001) Efficient historical R-trees. In: Proceedings of the 13th international conference on scientific and statistical database management, pp 223–232
37. Vengroff DE, Vitter JS (1996) Efficient 3-D range searching in external memory. In: Proceedings of the 28th annual ACM symposium on theory of computing, pp 192–201
38. Vietri G, Rodriguez LV, Martinez WA, Lyons S, Liu J, Rangaswami R, Zhao M, Narasimhan G (2018) Driving cache replacement with ml-based lecar. In: 10th {USENIX} workshop on hot topics in storage and file systems (HotStorage 18)
39. Wu CH, Chang LP, Kuo TW (2003) An efficient R-tree implementation over flash-memory storage systems. In: Proceedings of the 11th ACM international symposium on advances in geographic information systems, pp 17–24
40. Wu CH, Huang CW, Chang CY (2019) A data management method for databases using hybrid storage systems. ACM SIGAPP Appl Comput Rev 19(1):34–47
41. Wu CH, Kuo TW, Chang LP (2007) An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans Embed Comput Syst 6(3):19
42. Wu Y, Park K, Sen R, Kroth B, Do J (2020) Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In: Proceedings of the 16th international workshop on data management on new hardware, DaMoN '20. Association for Computing Machinery, New York. https://doi.org/10.1145/3399666.3399898
43. Xu J, Swanson S (2016) NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In: 14th USENIX conference on file and storage technologies (FAST 16). USENIX Association, Santa Clara, pp 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu
44. Yang J, Lilja DJ (2018) Reducing relational database performance bottlenecks using 3D XPoint storage technology. In: Proceedings of the 17th IEEE international conference on trust, security and privacy in computing and communications and 12th IEEE international conference on big data science and engineering, pp 1804–1808