A Study of R-tree Performance in Hybrid Flash/3DXPoint Storage

Athanasios Fevgas
Data Structuring & Engineering Lab
ECE Department
University of Thessaly
Volos, Greece
fevgas@e-ce.uth.gr

Leonidas Akritidis
Data Structuring & Engineering Lab
ECE Department
University of Thessaly
Volos, Greece
leoakr@e-ce.uth.gr

Miltiadis Alamaniotis
ECE Department
University of Texas at San Antonio
San Antonio, US
miltos.alamaniotis@utsa.edu

Panagiota Tsompanopoulou
Data Structuring & Engineering Lab
ECE Department
University of Thessaly
Volos, Greece
yota@e-ce.uth.gr

Panayiotis Bozanis
Data Structuring & Engineering Lab
ECE Department
University of Thessaly
Volos, Greece
pbozanis@e-ce.uth.gr

***Abstract*—The flash based solid state drives have become the storage medium of choice for many applications, replacing traditional HDDs in almost any data center. Their advent has motivated many research efforts in data management. 3DXPoint, a new non-volatile memory with even better specifications, is a breakthrough for storage systems; featuring low latency and high IOPS, 3DXPoint can create new lines of research. Towards this direction, hybrid storage systems combining both flash and 3DXPoint seem to be an adequate roadmap, since the cost of 3DXPoint remains high. In this paper, we study the performance of R-tree on both flash and 3DXPoint SSDs through careful experimentation. We also examine a simple yet illuminating approach to develop a hybrid index. The conducted experiments on one real and two synthetic datasets show that spatial indexes can achieve significant performance gains by exploiting 3DXPoint technology. To the best of our knowledge this is the first research effort that considers a hybrid flash/3DXPoint storage for R-tree.**

***Index Terms*—Data access methods, R-tree, SSD, flash, 3DX-Point, spatial indexing**

I. Introduction

In the recent years, the advent of Solid State Drives (SSDs) has revolutionized data storage systems. Traditional Hard Disk Drives (HDDS) are loosing their share in both consumer and enterprise markets, since SSDs provide high read and write speeds, low power consumption, high density and sock resistance. On the other hand, the read and write speeds of SSDs differ, while a certain workload can suffer unpredictable latency due to garbage collection. These performance characteristics of SSDs have created new lines of research in data management. Many studies pursue adapting data access methods to the intrinsic characteristics of this new storage medium. The initial efforts mostly intended to absorb the imbalance between read and write speeds, trading reads for writes [1], [16], [29], [31]. The next State-of-the-art works suggest employing the SSDs' internal parallelism [6], [21]–[23], while the latter ones [7], [24] additionally exploit the efficiency of the NVMe protocol to enhance query performance.

3DXPoint is a new non-volatile memory (NVM) introduced by Intel and Micron in 2015. Although it can be used as main memory, currently is widely available only on secondary storage products, under the "Optane" brand name. A key feature of this new NVM is its low latency. Specifically, an Optane SSD has read latency as low as 7μs; that is roughly 10 time less than the respective of a conventional flash SSD [15]. This property is of high importance, since low latency in information systems is associated with improved user experience. Another aspect of 3DXPoint SSDs' performance is their ability to deliver high throughput even at small queue depths (i.e. with a small number of outstanding I/O requests) [10], [14]. On the other hand, conventional flash devices perform better when high volumes of data requests are issued in batches [7], [21].

In our previous work, we studied the performance parameters of Grid File in hybrid flash/3DXPoint storage, introducing H-Grid [8] a hybrid point access method. In this paper we focus on tree indexes and specifically to R-tree. To the best of our knowledge, this is the first study that exploits 3DXPoint to improve the performance of a spatial access method.

The contributions of this work are summarized in the following:

- We study the performance characteristics of R-tree in both flash and 3DXPoint SSDs, employing one real and two synthetic datasets.
- We present a design for an efficient hybrid spatial index based on R-tree.
- We exploit a simple yet informative approach for constructing a hybrid R-tree to support our claim.

The remainder of this paper is organized as following. Section II provides a brief background on the basics of non-

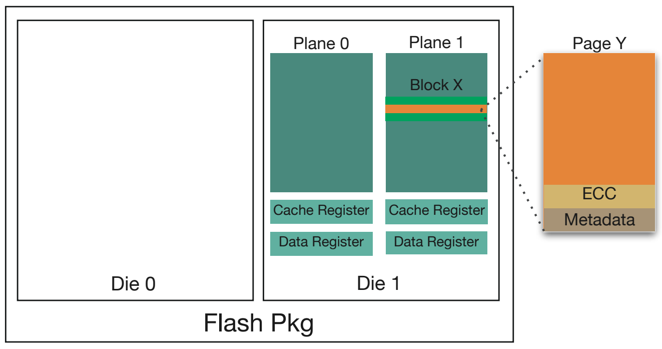


Fig. 1: The main components of a Flash package with 2 dies

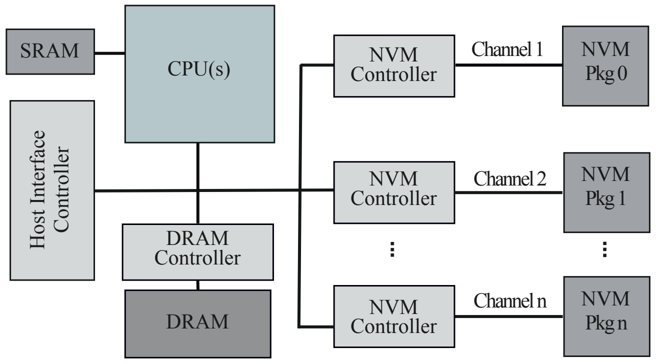


Fig. 2: SSD drive

volatile memories (flash and 3DXPoint), of SSD devices and R-trees. Section III discusses the design challenges of a hybrid index, and Section IV analyzes the experimental results. The related work on hybrid data management systems is presented in Section V. Finally, our conclusions are presented in Section VI.

II. Background

A. NVMs and SSDs

NAND Flash: The vast majority of SSDs utilize NAND flash to store information, rendering it as the most wide spread NVM today. NAND stores data by trapping electrical charge into flash cells. It exploits different voltage thresholds to store up to four bits in a single cell. Thus, four types of NAND exist today, particularly, SLC NAND stores one bit, MCL two bits, TLC and QLC, three and four bits, respectively. Flash cells are organized into pages, which are the smallest units that can be read or written [19]. In turn, pages are grouped into blocks that are the smallest erasable units. Figure 1 illustrates a typical NAND flash organization. As one can observe, a number of blocks along with registers assembles a plane, and several planes compose a die (chip). A flash package is assembled by two or more dies. The dies in a package can perform different operations (read, program, erase) at the same time forming a kind of of parallelism, which is often referred in the literature as package-level parallelism [21]. Moreover, the planes in a die are able to execute the same command simultaneously, providing an additional level of parallelism (plane level) [6], [11]. A flash characteristic that has motivated many research efforts is the asymmetry between read, program, and erase operation speeds. The read operation is faster than the programming, while the erase operation demands more time from both read and program. Successive program and erase operations impair flash cells, causing increased error rates. Other common sources of errors are aging and overheating. In the case a block presents high error rates, it is excluded from future usage.

3D Xpoint: 3DXPoint was developed with the aim to fill the gap between DRAM and flash. It provides lower latency than flash (10^3x) and higher density (10x) than DRAM [19]. It uses a cross-point architecture that permits bit-level addressing. Each 3DXPoint cell is capable to store one bit of information. The cells are stacked at successive layers. Unlike flash, 3DXPoint allows in-place writes. This feature, facilitates the design of storage devices and enables its usage as main memory. Specifically, the authors in [10] discern three application fields for 3DXPoint: i) within secondary storage devices, ii) as a chip volatile extension of DRAM, and iii) as persistent main memory beside DRAM. Today, there are some SSDs models that exploit 3DXPoint to store data. Within 2019, the first products for the main memory will be available in the market.

SSD Devices: Solid State Drives (SSDs) utilize non-volatile memories to store data. A typical SSD device is depicted in Figure 2. The most important parts are the NVM memory chips, the controller, and the DRAM. The SSD controller is assembled by a CPU, SRAM, interconnection controllers for the NVM and DRAM memory, and a host interface. The NVM chips are connected to communication buses called channels. The number of channels determines the internal parallelism of the drive. The flash memory controllers handle data transfers from/to NVM memory. The embedded CPU and the SRAM provide the execution environment for the firmware code that manages the drive. SSDs integrate a significant amount of DRAM intended to cache data, improving performance. Finally, the host controller interface interconnects the device to the host system, leveraging a bus interface (e.g. PCIe, SATA, SAS).

B. R-tree

R-tree was introduced by Guttman [9] in 1984, with the aim to facilitate VLSI design. However, very soon it became a popular data access method in both industry and academia with a wide range of applications. Geographical information systems and multimedia databases are considered being among them [26]. A large number of R-tree variants have been proposed since its introduction, pursuing to improve its efficiency or to modify it for different applications [18].

R-tree is a dynamic hierarchical data structure akin to that of B+tree. It uses multidimensional minimum bounding rectangles (MBRs) to organize spatial objects. Thus, each leaf node entry stores the smallest MBR that encloses a single

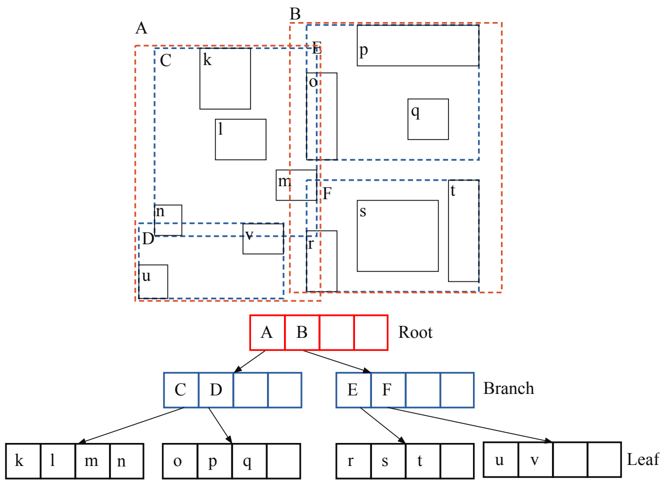


Fig. 3: A set of rectangles $\{k, l, m, n, o, p, q, r, s, t, u, v\}$ and the corresponding R-tree (m=2, M=4)

geometric object O, and a pointer to the address of the particular object rather than the object itself, i.e. $(MBR_O, \vec{O})$. Similarly, each internal node entry contains an MBR that encloses all MBRs of its descendants, and a pointer to their sub-tree T, namely $(MBR_T, \vec{T})$. If the R-tree resides in secondary storage, its nodes correspond to disk pages. Each R-tree node (not the root) can store at least m and at most M entries, with $m \leq M/2$. The root can store two records at minimum, unless it is a leaf; 0 or 1 entries are allowed in such a case.

MBRs from different nodes may overlap. Even if an object is enclosed by many MBRs, it is always associated with only one of them. Therefore, the search procedure for a spatial object O starts from the root and traverses the tree towards the leaves; however, it might follow several paths in order to ascertain the existence (or not) of O. This results in the worst case of retrieving a small number of objects to a cost that is linear to the size of the data.

The height of an R-tree determines the least number of pages that must be retrieved in order to touch a spatial object. Assuming that an R-tree accommodates N rectangles, then its maximum height is

$$H_{max} = \log_m N - 1 \tag{1}$$

Figure 3 illustrates a group of MBRs in the plane and one possible R-tree ($m = 2, M = 4$). Rectangles k, l, m, n, o, p, q, r, s, t, u and v enclose the spatial objects (not depicted here), forming the leaves of the tree. Similarly, C, D, E, F, organize the leaves forming R-tree's internal nodes, while A, B designate the root. It must be noted that alternative R-trees may be constructed indexing the same spatial objects. The structure of the resulting R-tree depends, to a large extent, on the order of the insert and/or delete operations issued to it.

Several variants of R-tree have been proposed in order to improve its efficiency; some representative examples are

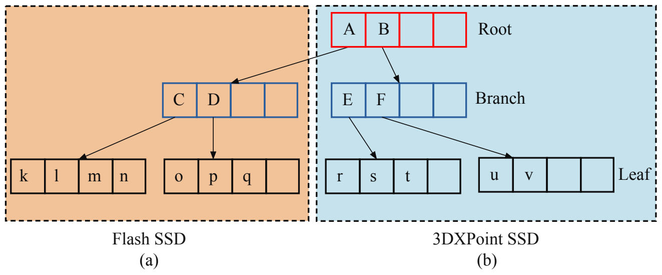


Fig. 4: Indicative example of a hybrid R-tree; a part of the tree is stored to the 3DXPoint storage.

R$^+$-tree [27], R*-tree [2], Hilbert R-tree [13], Cubetree [25], Historical R-tree [28] and LR-tree [3].

III. R-TREE FOR HYBRID STORAGE

Previous works for flash efficient data access methods target to: i) reduce the number of random writes, and ii) exploit the high throughput and internal parallelism of SSDs. For this reason, they employ logging techniques and batch update operations. Furthermore, some works group several queries together and process them at once [7], [22], [24]. The performance characteristics of 3DXPoint SSDs (i.e., low latency and high IOPS) and their cost, which remains significant higher than that of the flash-based ones, motivated us to study indexing methods under a different perspective. Therefore, we believe that a hybrid storage configuration that combines both NVM technologies (flash and 3DXPoint) can be a decent option. Next, we describe some key features that a hybrid index should have and we provide some implementation details.

A running example of a hybrid tree T is illustrated in Figure 4. A part of the tree is stored on the 3DXPoint SSD (Fig. 4(a)) while the rest of it is left on the flash based one (Fig. 4(b)). Such a design is based on two fundamental operations: i) a selection algorithm that detects hot data regions, and ii) a replacement policy that retains only the hottest data to the fast storage over time. The hot data selection algorithm can be based on usage statistics, combined with other factors, such as the spatial properties of the indexed objects. In more detail, the frequency of accesses in a specific region R, as well as the recency of these accesses, can provide a strong indication of its popularity. Let R be a sub-tree of T. As the available space in the performance tier (3DXPoint) is reduced, an adequate reclaim policy is needed to migrate cold regions to the storage tier (flash).

We present some implementation guidelines bellow. As we mentioned above, the selection of the hottest regions is performed using various criteria. As a result, a weight value is calculated for each region. The weights are maintained in-memory, using hash tables that guarantee fast access. Also, an in-memory buffer is exploited to keep recently accessed node pages. We consider LRU as replacement policy, however alternative algorithms can also be applied. Each tree node of can reside in either the main memory, or the flash, or the 3DXPoint storage.

Algorithm 1: RetrieveNode(N, MB)

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

Algorithm 1 describes node retrieval. If the requested node N is already in the in-memory buffer (MB), it is moved to the most recently used (MRU) position of MB. Otherwise, a fetch operation from the secondary storage is initiated. By the end of the operation, N is placed to the MRU position of the main buffer and a reference to it is returned.

Let $x_N \in \{0, 1\}$ denote whether node N is stored in the 3DXPoint storage or not. Assuming that N occupies a single page in the secondary storage, the cost of retrieving N is

$$C_N = x_N * R_x + R_f * (1 - x_N) = R_f - x_N * (R_f - R_x) \quad (2)$$

where R_f and R_x are the reading costs from the flash and 3DXPoint, respectively.

We also present a simple yet illuminating case of a hybrid R-tree (Fig. 5), where all non-leaf nodes are stored to the 3DXPoint SSD. We refer to it as sHR-tree from now on. The upper level nodes of an R-tree are referenced more often than the leaves and follow a small-size I/O access pattern, as the tree is traversed from the root to the leaves. Therefore, the efficiency of 3DXPoint at small queue depths can lead to considerable performance improvement. Using sHR-Tree, we aim at drawing useful conclusions about R-tree performance, when a hybrid storage scheme is applied.

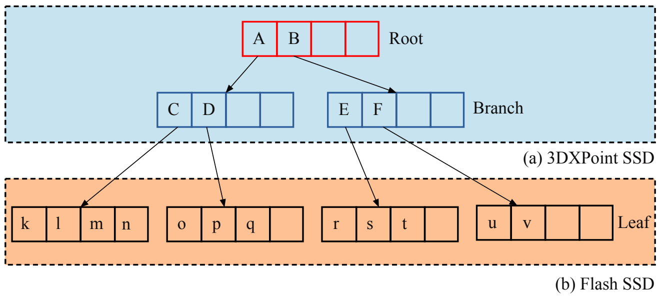


Fig. 5: A simple yet illuminating case of a hybrid R-tree index (sHR-tree); all non-leaf nodes are stored to the 3DXPoint storage.

IV. Evaluation

A. Methodology and setup

In this section we discuss the performance evaluation of R-tree in the various storage configurations. We conducted a series of experiments, using both flash and 3DXPoint storage devices. We aim at unfolding the benefits of hybrid index configurations, against an approach that considers single storage medium. For this reason, we evaluate two different workloads, concerning i) index construction, and ii) execution of 5000 range queries.

The experiments were run on a workstation equipped with an 4-core Intel Xeon CPU E3-1245 v6 3.70GHz CPU and 16GB of RAM. The workstation runs CentOS Linux 7, hosted on a separate SATA SSD. A flash NVMe SSD and an Optane counterpart device were used for the experiments. The performance characteristics of both devices are listed in Table I.

We utilized three datasets in the experiments: one real-world containing 300M points, and two synthetic, having Gaussian and Uniform distributions, of 50M points each. The real dataset is derived from Openstreetmap[1]. All experiments were executed using the Direct I/O (O_DIRECT) to avoid the effects of Linux OS caching layer. The total size of the in-memory buffers was configured to 4MB. We did not employ a special write buffer in the experiments.

TABLE I: SSD Specification

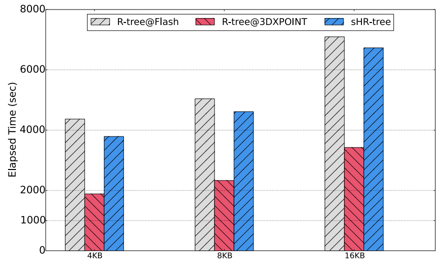
	Intel DC P3700 (FLASH)	Optane Memory series (3DXPOINT)
Seq. Read	up to 2700MB/s	up to 1350MB/s
Seq. Write	up to 1100MB/s	up to 290MB/s
Random Read	450K IOPS	240K IOPS
Random Write	75K IOPS	65K IOPS
Latency Read	120μs	7μs
Latency Write	30μs	18μs

B. Index Construction

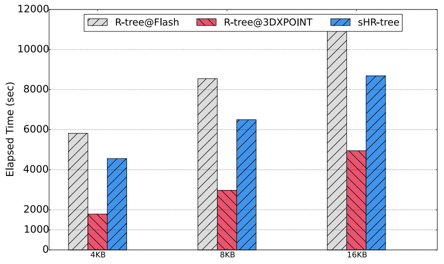
Regarding index construction, we examine three different test cases: i) construction of R-tree on flash, ii) construction of R-tree on 3DXPoint, and iii) construction of the sHR-tree, described in Section III, that uses both storages.

We present the execution times for three different page sizes (4KB, 8KB and 16KB) in Figure 6. The results are quite impressive for the run on the 3DXPoint SSD. It improves the execution time, compared to the run on the flash SSD, up to 57% for the real dataset and up to 69% and 68% for the Gaussian and Uniform datasets, respectively. The improvement is significant for the sHR-tree as well; it achieves a gain up to 13% for the real dataset and up to 24% for the synthetic ones compared to the flash run. The index construction time is less in the real dataset case, despite the fact that its size is quite bigger than the size of the synthetic ones. This occurs because the objects in the real dataset exhibit spatial locality, which contributes to a high number of cache hits. We also observe that the page size influences performance. Specifically,

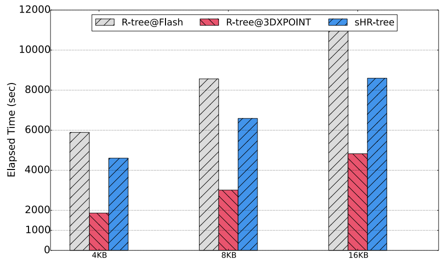
[1]http://spatialhadoop.cs.umn.edu/datasets.html



(a) Real dataset

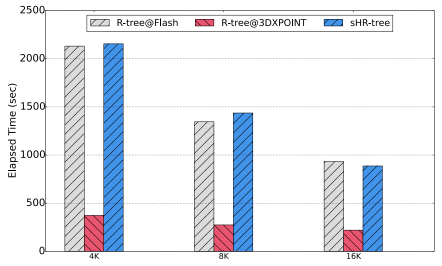


(b) Gaussian dataset

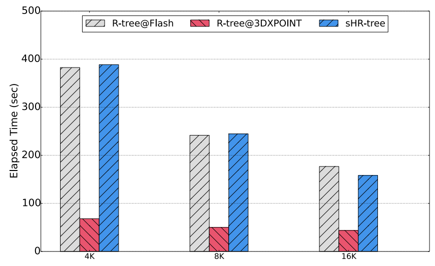


(c) Uniform dataset

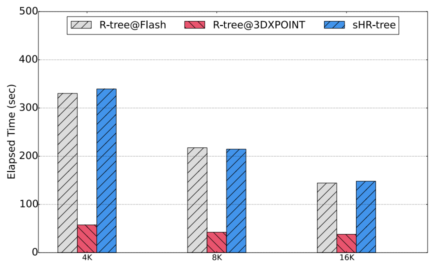
Fig. 6: Execution times of index construction for different workloads. A gain up to of 13% for the real dataset and up to 24% for the synthetics is achieved by the sHR-tree.



(a) Real dataset



(b) Gaussian dataset



(c) Uniform dataset

Fig. 7: Execution times of range queries for different workloads. R-tree@3DXPoint achieves an improvement up to 82% in comparison to the R-tree@Flash.

when 4KB pages are used the results are better in all test cases. This is expected to a certain point, since the page size determines the size of data written each time. The study of an implementation that exploits a write buffer of fixed size is left for future work.

C. *Range Queries*

In this section we discuss the performance of range queries (Fig. 7). The obtained results for the R-tree residing in the 3DXPoint SSD are quite impressive; we get an improvement in comparison to the flash SSD execution up to 82%. On the other hand, sHR-tree improves only in two cases w.r.t. the real and Gaussian datasets (16KB page runs). This happens because the amount of data on the fast 3DXPoint SSD is not enough to contribute substantial performance gain. Therefore, a hybrid approach that persists not only the upper-level nodes in 3DXPoint storage, but identifies and stores the hottest regions (including the leaves) on it, can significantly contribute to query performance. This argument is strengthened by results concerning the 3DXPoint execution. Moreover, the performance is improved in all test cases by increasing the page size. This happens since larger pages require less I/O operations to fetch the requested objects.

V. Related Work

The high speed of SSDs and the low cost of magnetic HDDs render hybrid storages an attractive solution for many data management systems [20]. In particular, two different approaches are found in the literature. The former suggests migrating of the hottest data to SSDs, while the latter employs SSDs in a caching layer between main memory and magnetic disks.

DBMSes exploit different placement methods [4], [30] to move hot data in SSDs, while preserving the cold data to low cost magnetic disks. In [4], the authors demonstrate a utility that suggests database object (i.e. tables or indexes) placement based on workload statistics. Specifically, the objects that impose the higher random I/O are moved to the SSD. A data placement mechanism integrated in the database engine is presented in [30]. The authors locate hot database objects by examining the SQL queries.

Following a different strategy, the authors in [5] propose SSDs for caching. Specifically, they exploit SSDs as write-through cache between the in-memory buffer pool and the HDD. They introduce a policy to pick up the hottest data using frequency and recency statistics, and a replacement policy to clean up the SSD when gets full. Similarly, the authors in [17] study the effects of DBMSes' caching algorithms on the performance of the SSDs. In the sequel, they propose a new caching policy. This policy takes into account the diverse properties of the devices in a hybrid storage system. Moreover, they present a page admission/eviction policy to/from the SSD that is adapted to the aforementioned caching algorithm.

The performance characteristics of hybrid storage systems render existing methods, designed for a specific medium, inadequate. For this reason, the HybridB tree, an indexing method for hybrid SSD/HDD storage, is proposed in [12]. HybridB tree utilizes a huge leaf approach, spanning leaf nodes among SSD and HDD drives. In opposition, it always maintains the internal nodes in the SSD. HybridB tree achieves good performance, while it also manages to reduce random write operations to SSD.

Finally, the authors in [32] study various performance parameters of database systems. Specifically, they recognize i) write amplification, ii) misuse of temporary tables, and iii) buffer pool cache misses, as factors that degrade query performance. Their experimental findings show that a 3DXPoint SSD can improve queries' execution up to 6.5x in comparison to a NAND flash counterpart.

VI. Conclusions

In this paper, we investigate R-tree performance in a hybrid storage, which is composed by a flash and a 3DXPoint SDD. To the best of our knowledge, this is the first work that considers 3DXPoint NVM as storage medium for a widely used spatial access method such as the R-tree. We experimentally evaluate three different cases, namely: i) R-tree on flash, ii) R-tree on 3DXPoint, and iii) a simple hybrid R-tree implementation (sHR-tree). The experimental results support our design hypothesis. Specifically, the R-tree execution exclusively on the 3DXPoint device improves index construction up to 69%. Similarly, the hybrid R-tree improves up to 24%. Regarding range queries, a gain up to 82% is achieved when 3DXPoint is the sole storage. However, the gain is marginal for the hybrid approach, since only a small number of nodes reside in the fast storage.

Our plans for future work include the development of a hot region detection algorithm that locates and migrates regions of high interest to a high performing 3DXPoint based storage. We also aim to further investigate spatial query processing, in order to take the full advantage of 3DXPoint properties.

References

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 322–331, Atlantic City, NJ, 1990.

[3] P. Bozanis, A. Nanopoulos, and Y. Manolopoulos. Lr-tree: a logarithmic decomposable spatial index method. *The computer journal*, 46(3):319–331, 2003.

[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. An object placement advisor for db2 using solid state storage. *Proceedings of the VLDB Endowment*, 2(2):1318–1329, 2009.

[5] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. Ssd bufferpool extensions for database systems. *Proceedings of the VLDB Endowment*, 3(1-2):1435–1446, 2010.

[6] F. Chen, B. Hou, and R. Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage*, 12(3):13, 2016.

[7] A. Fevgas and P. Bozanis. Lb-grid: An ssd efficient grid file. *Data & Knowledge Engineering*, 2019. available online.

[8] A. Fevgas and P. Bozanis. A spatial index for hybrid storage. In *23rd International Database Engineering & Applications Symposium. IDEAS 2019*. ACM, 2019.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 47–57, Boston, MA, 1984.

[10] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[11] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*, pages 96–107, Tucson, AZ, 2011.

[12] P. Jin, P. Yang, and L. Yue. Optimizing B$^+$-tree for hybrid storage systems. *Distributed & Parallel Databases*, 33(3):449–475, 2015.

[13] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.

[14] I. Koltsidas and V. Hsu. IBM storage and NVM express revolution. Technical report, IBM, 2017.

[15] K. Kourtis, N. Ioannou, and I. Koltsidas. Reaping the performance of fast NVM storage with udepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, Boston, MA, 2019.

[16] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Efficient indexing data structures for flash-based sensor devices. *ACM Transactions on Storage*, 2(4):468–503, 2006.

[17] X. Liu and K. Salem. Hybrid storage management for database systems. *Proceedings of the VLDB Endowment*, 6(8):541–552, 2013.

[18] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-trees: theory and applications*. Springer, 2010.

[19] R. Micheloni. *3D flash memories*. Springer, 2016.

[20] J. Niu, J. Xu, and L. Xie. Hybrid storage systems: A survey of architectures and algorithms. *IEEE ACCESS*, 6:13385–13406, 2018.

[21] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B$^+$-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[22] H. Roh, S. Park, M. Shin, and S.-W. Lee. Mpsearch: Multi-path search for tree-based indexes to exploit internal parallelism of flash SSDs. *IEEE Data Engineering Bulletin*, 37(2):3–11, 2014.

[23] G. Roumelis, A. Fevgas, M. Vassilakopoulos, A. Corral, P. Bozanis, and Y. Manolopoulos. Bulk-loading and bulk-insertion algorithms for xbr-trees in solid state drives. *Computing*, 2019. available online.

[24] G. Roumelis, M. Vassilakopoulos, A. Corral, A. Fevgas, and Y. Manolopoulos. Spatial batch-queries processing using xBR$^+$-trees in solid-state drives. In *Proceedings of the 8th International Conference on Model & Data Engineering (MEDI)*, pages 301–317, Marrakesh, Morocco, 2018.

[25] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: organization of and bulk incremental updates on the data cube. *ACM SIGMOD Record*, 26(2):89–99, 1997.

[26] H. Samet. Applications of spatial data structures. 1990.

[27] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. pages 507–518, 1987.

[28] Y. Tao and D. Papadias. Efficient historical r-trees. In *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*, pages 223–232. IEEE, 2001.

[29] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient r-tree implementation over flash-memory storage systems. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, GIS '03, pages 17–24, New York, NY, USA, 2003. ACM.

[30] C.-H. Wu, C.-W. Huang, and C.-Y. Chang. A data management method for databases using hybrid storage systems. *ACM SIGAPP Applied Computing Review*, 19(1):34–47, 2019.

[31] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems*, 6(3):19, 2007.

[32] J. Yang and D. J. Lilja. Reducing relational database performance bottlenecks using 3d xpoint storage technology. In *Proceedings of the 17th IEEE International Conference on Trust, Security & Privacy in Computing & Communications / 12th IEEE International Conference On Big Data Science & Engineering (TrustCom/BigDataSE)*, pages 1804–1808, 2018.