The VLDB Journal
https://doi.org/10.1007/s00778-024-00874-9

REGULAR PAPER

DumpyOS: A data-adaptive multi-ary index for scalable data series similarity search

Zeyu Wang[1] · Qitong Wang[2] · Peng Wang[3] · Themis Palpanas[4] · Wei Wang[3]

Received: 29 August 2023 / Accepted: 7 August 2024

Abstract

Data series indexes are necessary for managing and analyzing the increasing amounts of data series collections that are nowadays available. These indexes support both exact and approximate similarity search, with approximate search providing high-quality results within milliseconds, which makes it very attractive for certain modern applications. Reducing the preprocessing (i.e., index building) time and improving the accuracy of search results are two major challenges. DSTree and the iSAX index family are state-of-the-art solutions for this problem. However, DSTree suffers from long index building times, while iSAX suffers from low search accuracy. In this paper, we identify two problems of the iSAX index family that adversely affect the overall performance. First, we observe the presence of a *proximity-compactness trade-off* related to the index structure design (i.e., the node fanout degree), significantly limiting the efficiency and accuracy of the resulting index. Second, a skewed data distribution will negatively affect the performance of iSAX. To overcome these problems, we propose Dumpy, an index that employs a novel multi-ary data structure with an adaptive node splitting algorithm and an efficient building workflow. Furthermore, we devise Dumpy-Fuzzy as a variant of Dumpy which further improves search accuracy by proper duplication of series. To fully leverage the potential of modern hardware including multicore CPUs and Solid State Drives (SSDs), we parallelize Dumpy to DumpyOS with sophisticated indexing and pruning-based querying algorithms. An optimized approximate search algorithm, DumpyOS-F that prominently improves the search accuracy without violating the index, is also proposed. Experiments with a variety of large, real datasets demonstrate that the Dumpy solutions achieve considerably better efficiency, scalability and search accuracy than its competitors. DumpyOS further improves on Dumpy, by delivering several times faster index building and querying, and DumpyOS-F improves the search accuracy of Dumpy-Fuzzy without the additional space cost of Dumpy-Fuzzy. This paper is an extension of the previously published SIGMOD paper [81].

Keywords Time series index · Big data management · Similarity search

1 Introduction

Massive data series collections are now being produced by applications across virtually every scientific and social domain [40, 53, 94], making data series one of the most common data types. The problems of managing and analyzing large-volume data series have attracted the research interest of the data management community in the past three decades [9, 54]. In this context, similarity search is an essential primitive operation, lying at the core of several other

✉ Peng Wang
pengwang5@fudan.edu.cn

Zeyu Wang
zeyuwang21@m.fudan.edu.cn

Qitong Wang
qitong.wang@etu.u-paris.fr

Themis Palpanas
themis@mi.parisdescartes.fr

Wei Wang
weiwang1@fudan.edu.cn

[1] School of Computer Science, Fudan University, Shanghai, China

[2] LIPADE, Université Paris Cité, Paris, France

[3] Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China

[4] LIPADE, Université Paris Cité & IUF, Paris, France

Published online: 21 August 2024

high-level algorithms, e.g., classification, clustering, motif discovery and outlier detection [11, 12, 53, 56, 70, 76].

Similarity search aims to find the nearest neighbors in the dataset, given a query series and a distance measure. The naive solution is to sequentially calculate the distances of all series to the query series. However, sequential scan quickly becomes intractable as the dataset size increases. To facilitate similarity search at scale, a data series index can be used to prune irrelevant data and thus, reduce the effort required to answer the queries. Moreover, as researchers pay more attention to data exploration, the importance of approximate similarity search grows rapidly [27, 28]. It is widely employed in real-world applications since it can provide high-quality approximate query results within the interactive response time, in the order of milliseconds [7, 8, 17, 48, 75]. In such applications, approximate query result quality is sufficient to support downstream applications [18]. Recent examples include (1) a k-nearest-neighbors (kNN) classifier [3], whose accuracy converges to the best when kNN mean average precision (MAP) reaches 60%; (2) an outlier detector [65] that achieves the best ROC-AUC with 50% MAP; and (3) a kNN-based SoftMax approximation technique for large-scale classification, which achieves nearly the same accuracy as the exact SoftMax when kNN recall reaches 80% [88]. For these applications, the core requirements for the kNN-index are the query time under the above precision (should be in the order of milliseconds), the index building time, and the scalability to support large datasets.

Although there are dozens of approaches in the literature that can index data series [28], only a few of them can robustly support large data series collections, e.g., over 100 GB (which is why techniques for approximate search [28], as well as progressive search for exact [25] and approximate [38] query answering have been studied). Among them, DSTree [78] and the iSAX index family [55] show the best query performance on the approximate search and support exact search at the same time. Due to the dynamic segmentation technique, DSTree requires a long index building time (over one order of magnitude slower than iSAX) and is hard to optimize. On the contrary, benefiting from fast index building and rich optimizations [14, 15, 59, 60, 62, 85, 90], the iSAX index family has become the most popular data series index in the past decade. Nonetheless, iSAX still suffers from unsatisfactory approximate search accuracy when visiting a small portion of data (one or several nodes, ensuring millisecond-level delay), e.g., its MAP is less than 10% when visiting one node while query time exceeds one second when improving MAP to $\geq$50% [28]. In this work, we identify the intrinsic problem of the index structure and building workflow of the iSAX index family and propose our novel solution, Dumpy, to tackle those.

First, we observe that the design of the index structure is an inherent but overlooked problem that significantly limits the performance of the iSAX index family. Although iSAX [67] does not in principle limit the fanout of a node, popular iSAX-family indexes [13, 15, 49, 58, 59, 62, 85, 90] still adopt a binary structure (except for the first layer that has a full fanout). When a node contains more series than the leaf size threshold th, it selects one SAX segment and splits the node into two child nodes.

However, under this binary structure, the splitting policies being used [13, 67] often lead to sub-optimal decisions (cf. [87], Sect. 4), that hurt the proximity (i.e., similarity) of series inside a node, and finally the quality of approximate query results.

Recently, a full-ary SAX-based index has been proposed to tackle this problem [87]. A full-ary structure splits a full node on all segments, such that it avoids the problems of focusing on a single segment that leads to sub-optimal splitting decisions. However, it generates too many nodes (at most 2^w, w is the total number of segments) in each split, leading to an excessive number of leaf nodes, and hence extremely low leaf node fill factors. This leads to an underperforming (disk-resident) index, due to inefficient disk utilization and overwhelming disk accesses. Although subtrees in the index can be merged into larger partitions (e.g., 128 MB) [87] to reduce random I/Os, it still incurs substantial overhead to store and load its large internal index structure and introduce many application limitations at the same time.

We term the aforementioned problems as the *proximity-compactness trade-off*. Both proximity and compactness contribute to similarity search since proximity provides closer series to the query and compactness provides more candidate series when visiting a node.

The binary index structure aims at providing compact child nodes, but impairs the accuracy of query results, whereas the full-ary structure splits the node to preserve the proximity of series inside nodes, but fails to provide leaf nodes of high fill factors (i.e., compactness). As a result, both structures fail to exploit the proximity-compactness trade-off, limiting their performance on search accuracy and also building efficiency.

In this work, we break the limits of a single fixed fanout for the iSAX-family indexes and propose an adaptive split strategy that leads to a multi-ary index structure. Specifically, we design a novel objective function to estimate the qualities of candidate split plans in the aspects of both proximity and compactness. We use the average variances of data on selected segments to measure the intra-node series proximity and the variance of fill factors of child nodes to measure the compactness. Moreover, we propose an efficient search algorithm comprised of three speedup techniques to find the optimal split plan according to our quality estimation.

Besides the index structure design, we identify two other problems of the iSAX-index family preventing the best exploitation of the novel adaptive multi-ary index structure.

The first observation is that when the fanout is large (e.g., the first layer in the binary structure and all layers in the full-ary structure), data series are often distributed among the child nodes in a highly imbalanced way, which cannot be entirely avoided, even when we choose the best split plan. That is, most data series concentrate on only a few nodes while most nodes are slight in size. It usually leads to a large number of small nodes that impair the performance of the resulting index. The other problem is that the common iSAX index building workflow splits a node by relying only on the distribution of a tiny portion of data, which actually makes the splitting decisions sub-optimal for the data as a whole. For example, iSAX2+ tries to balance two child nodes in splitting according to the first $th + 1$ series (i.e., split once it is full), but the final average fill factor is usually less than 20% as verified in our experiments.

To avoid these two problems, we design a flexible and efficient index-building workflow along with a leaf packing algorithm. Benefiting from the static segmentation of iSAX, our workflow can collect the global SAX word tables without incurring any additional overhead, and make our adaptive split strategy better fit the whole dataset. Moreover, our leaf node packing algorithm can pack small sibling leaves without losing the pruning power, contributing to fewer random disk accesses during index building and querying.

In summary, by combining the adaptive split strategy with the new index building workflow, we present our data series indexing solution, Dumpy (named after its short and compact structure). Dumpy advances the State-Of-The-Art (SOTA) in terms of index building efficiency, approximate search accuracy, and exact search performance, making it a fully-functional and practical solution for extensive data series management and analysis applications.

Moreover, generally as a space-partition-based approach, Dumpy also suffers from a common boundary issue [19, 32]. That is, the kNN of a query may locate in the adjacent node or subtree and near the partition boundary. Since we only search one to several nodes, these true neighbors may be missing. To alleviate this effect, we propose a variant of Dumpy, Dumpy-Fuzzy, which transfers the *hard* partition boundary to a *fuzzy range*, and adopts a duplication strategy in each split to further improve the search accuracy, at the cost of a small overhead on index building and storage.

Modern parallel hardware, such as multi-core CPUs and NVMe SSDs, becomes pervasive nowadays on commodity machines and data centers. To fully exploit the potential of these devices, we extend Dumpy to a well-designed parallel solution, DumpyOS (short for Dumpy On Steroids), to further accelerate the index building and pruning-based querying. Parallel computing can accelerate near-linearly the adaptive split when indexing, and distance calculation when querying while a proper utilization of SSD can resolve the I/O bottleneck. Moreover, computations and I/Os are separated in DumpyOS so that they can be designed to overlap with each other leading to even higher performance.

Furthermore, by extending the fuzzy mechanism we design a novel approximate search algorithm DumpyOS-F (short for DumpyOS-Fuzzy), which directly operates on the Dumpy index without any physical duplication of the series in the index. In contrast to Dumpy-Fuzzy, which statically finds series in the fuzzy boundary for a given node during indexing, DumpyOS-F only works during querying by dynamically finding the series that are close to the query series, but located in different leaf nodes. Therefore, DumpyOS-F is more flexible to select candidate series with better proximity, and gets rid of the compactness constraint of Dumpy's structure. As a result, DumpyOS-F achieves better accuracy and avoids the data duplication strategy that Dumpy-Fuzzy uses (and which violates the integrity of the Dumpy index structure).

An extensive experimental evaluation with several synthetic and real datasets shows that DumpyOS (with DumpyOS-F) outperforms the state-of-the-art competitors across all measures.

Our contributions[1] can be summarized as follows.

(1) We identify the inherent proximity-compactness trade-off in the structural designs of the current SOTA iSAX-index family, and demonstrate that it limits the quality of approximate query results, as well as the index building efficiency.
(2) We present Dumpy, a novel multi-ary data series index that hits the right balance of the proximity-compactness trade-off by adaptively and intelligently determining the splitting strategy on-the-fly.
(3) We design a powerful and efficient index-building workflow for the iSAX-index family with a novel leaf packing algorithm to handle data skewness and achieve robust performance.
(4) We devise Dumpy-Fuzzy to further improve search accuracy by proper data duplication.
(5) We develop DumpyOS, a parallel, materialized time series index, which fully leverages multi-core CPUs and NVMe SSDs to achieve significantly higher performance in both index building and pruning-based querying.
(6) We design DumpyOS-F, a novel approximate search algorithm that uses the existing Dumpy index structure with no additional space costs. DumpyOS-F leads to more accurate search results than Dumpy and Dumpy-Fuzzy.
(7) Our experimental evaluation with a variety of synthetic and real datasets demonstrates that Dumpy and its variants provide consistently faster index building times (up to 5.3x; 4x on average), and higher approximate query

[1] A preliminary version of this paper has appeared elsewhere [81].

accuracy (up to 130%; 65% higher MAP on average) than the SOTA competitors, with query answering times in the order of milliseconds. DumpyOS further achieves on average 3.7x faster building time and 5.8x faster exact query time than Dumpy on multi-core CPU and SSD, and DumpyOS-F is 18% more accurate than Dumpy on average in approximate search.

2 Related work

[Data series indexes] Dozens of methods have been proposed to index massive data series collections [27, 28]. Among these, the SAX-based indexes [55] have gained popularity and achieved SOTA performance. Following the initial iSAX [67] index, iSAX2.0 and iSAX2+ [13, 14] provide faster index building through novel bulk loading and node splitting strategies, ADS [90] optimizes the combined index building and query answering time, ULISSE [50] supports subsequence similarity search, SEAnet [77] improves query results quality for high-frequency time series using deep learning embeddings, while DPiSAX [84], Odyssey [15], PARIS [60], MESSI [61], SING [62], and Hercules [26] exploit distribution and modern hardware parallelism. These indexes all inherit the original binary structure of iSAX, which limits their intra-node series proximity.

ADS [90], as a *query-adaptive* index, builds and materializes only the leaf nodes visited by the examined queries. However, in the case of a huge query workload that visits all leaf nodes of the index, ADS becomes the same as an iSAX index, with the same query answering properties. On the contrary, as a *data-adaptive* index, Dumpy adapts its structure based on the data collection rather than the queries. Therefore, its performance is independent of workloads. (We omit ADS in the experiments since it is not superior to iSAX2.0 and DSTree [27].)

TARDIS [87] first notices the drawbacks of the binary structure and proposes a full-ary structure along with a size-based partitioning strategy to merge different subtrees to be applied in a distributed cluster. However, TARDIS is only for analyzing a static dataset and the enormous structure decreases the building and query efficiency. We implement a stand-alone version of TARDIS in our experiments. Coconut [43, 44] builds a B+-tree after sorting the dataset using the InvSAX representations and gains remarkable performance improvement from sequential I/Os in bulk loading. However, the sequential layout on disk will be destroyed by further insertions, and the scan-based exact-search algorithm requires a complete InvSAX table to be kept in memory and the raw dataset in place. And it seems no easy way to restore the classical tree-based pruning in Coconut. Hence, we do not include Coconut in our experiments.

DSTree [78] achieves remarkable search accuracy by adopting a highly adaptive summarization EAPCA and increasing the number of segments on the fly. While the side effect is that DSTree cannot skip costly split operations on raw data series. Bulk loading algorithms and many other optimizations we mentioned are therefore hard to be applied on DSTree. As evaluated in our experiments, Dumpy provides higher-quality query results than DSTree even on the static summarization iSAX, with a much faster building time.

[Parallel disk-based indexes] Many methods are designed to index and query data series in a parallel environment. DPiSAX [84] and TARDIS [87] explore the proper way of data distribution on a cluster of machines. MESSI [61], SING [62] and Odyssey [15] build and query the in-memory iSAX index in parallel, but they cannot be extended to the disk index. PARIS [60] is a disk-resident solution that parallelizes the ADS+ index, which only materializes the SAX words when building the index. PARIS relies on serial scanning of the raw dataset to support exact search, which leads to an additional cost during query answering time (just like ADS+).

On the other hand, dozens of classical indexes (for very low-dimensional data, i.e., not suitable for data series) have been extended and optimized to a parallel environment with multi-cores and SSDs. For example, PA-Tree [74] optimizes the execution paradigm of B+-Tree on NVMe, TreeLine [86] is designed as an update-in-place key-value store on SSD, and FOR-Tree [37] optimizes R-Tree on SSD by reducing random writes.

Recently, Zheng et al. proposed DecLog [89], which is a novel decentralized logging technique for time series database management systems. Declog is also designed based on the merits of NVM to improve the I/O throughput. However, the problem of data series similarity search is not considered.

[High-dimensional vector indexes] According to recent studies [24, 27, 28], similarity search algorithms for data series and high-dimensional vectors could be employed interchangeably. Representative algorithms for high-dimensional vector search include proximity graph-based methods [51, 75], showing excellent query performance on small datasets, but consuming excessive time and memory to build and store the graph. Now they are not easy to scale in billion-scale datasets in commodity machines [32, 69]. Product quantization family methods [34, 35, 57] achieve better query performance on minute-level near-exact search than data series indexes in advanced research. However, the building time is still over one magnitude slower than DSTree [28]. Locality Sensitive Hashing (LSH) family methods [46, 47, 82], provide probabilistic guarantees for approximate similarity search, but have been shown to fall behind data series indexes in terms of time and space in the general case

[28], and even more importantly, cannot support exact query answering.

3 Background

We first provide some definitions necessary for the rest of this paper, and then explain the iSAX summarization and index.

Definition 1 (*Data Series*) A data series $s = \{x_1, x_2, \ldots, x_l\}$ is a sequence of points where x_i is the i-th point and l denotes the length of data series s.

We assume a data series database db contains numerous data series of equal length n. We use the kNN (k-Nearest Neighbor) query to denote a specific similarity search query with an explicit number of nearest neighbors.

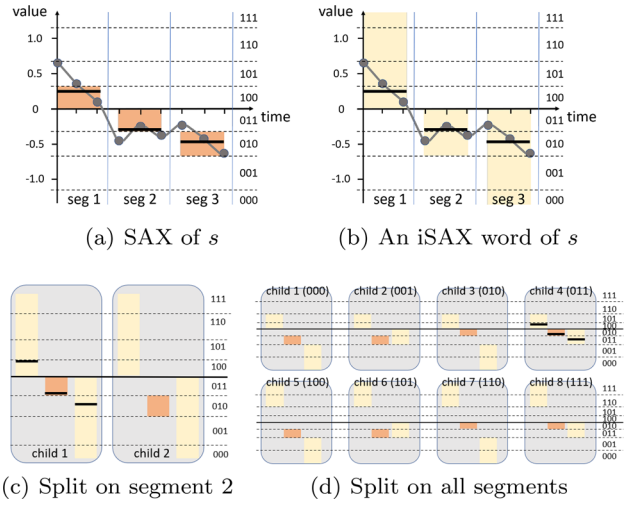
Definition 2 (*kNN Query*) Given an integer k, a query data series q and a distance measure $dist$, a **kNN Query** retrieves from the database the set of series $R = \{r_1, r_2, \ldots, r_k\}$ such that for any series $s \in db \backslash R$ and $r_i \in R$, $dist(r_i, q) \leq dist(s, q)$.

The choice of the distance measure depends on the particular application. Euclidean Distance (ED) is one of the most popular, widely studied and effective similarity measures for large data series collections [22]. Dynamic Time Warping (DTW) [63] is also widely adopted to analyze data series. Our solution supports both ED and DTW using the same data structure, like other iSAX indexes. Besides the exact kNN query, the approximate kNN query that accelerates the query processing by checking a small subset of the whole database has attracted intensive interest from researchers. The approximate query result, $A = \{a_1, \ldots, a_k\}$, is expected to be close to the ground truth result R.

[iSAX summarization] In this paper, we build Dumpy using the iSAX summarization technique [67]. iSAX is a dynamic prefix of SAX words, and SAX is a symbolization of PAA (Piecewise Aggregate Approximation) [41]. We briefly review these techniques with the example in Fig. 1.

PAA(s, w) divides data series s into w disjoint equal-length segments, and represents each segment with its mean value. Hence, PAA reduces s to a lower-dimensional summarization. As the black solid line shown in Fig. 1a, PAA(s,3)=[0.28,−0.31,−0.49].

SAX(s,w,c) is the representation of PAA by w discrete symbols, drawn from an alphabet of cardinality c. The main idea of SAX is that the real-value space can be split by $c-1$ breakpoints (subject to $N(0, 1)$) into c regions, that are labeled by distinct symbols. For example, when c=4 the available symbols (represented in bit-codes) are {00,01,10,11}. SAX assigns symbols to the PAA coefficients on each segment. In Fig. 1a, SAX(s,3,8) = [100,011,010]. The SAX word represents a *region* formed by the value ranges in w segments,



(a) SAX of s (b) An iSAX word of s

(c) Split on segment 2 (d) Split on all segments

Fig. 1 **a**, **b** are the PAA, SAX and iSAX representation ($w = 3$, $b = 3$). **c**, **d** are the node splitting for iSAX-index family in two- and full-ary structure, respectively

drawn in orange background. iSAX(s,w,c) uses variable cardinality ($\leq c$) in each segment. That is, an iSAX word is a prefix of the corresponding SAX word. The iSAX word in Fig. 1b is iSAX(s,3,8)=[1,01,0].[2] Due to the decreased cardinality of the alphabet, an iSAX word represents a larger range (more coarse-grained) than the corresponding SAX word.

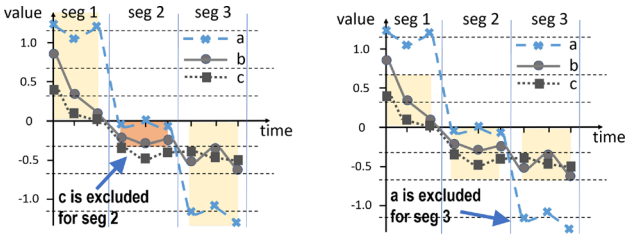
[iSAX index family] The iSAX index family [55] uses the tree structure to organize data series, which consists of three types of nodes. The root node representing the whole value space, points to at most 2^w child nodes by splitting on all w segments. Each internal node contains the common iSAX word of all the series in it, and pointers to its child nodes. Each leaf node stores the raw data and the complete SAX words of the series inside. The iSAX index is built by inserting series to the target leaf node one by one. Once the size of a leaf node exceeds the capacity th (a user-defined parameter), the leaf node gets transferred into an internal node and splits the series inside into several child nodes. The child node occupies a subspace of the space represented by its parent. There are two splitting strategies in the iSAX-index family. The first is the binary split (see Fig. 1c), which splits a node by doubling the cardinality of the iSAX symbol on *one* segment, and thus, the two child nodes represent disjoint ranges on the specific segment while remaining the same for other segments.

Figure 1b, c show an example. The node shown in Fig. 1b indicates that the series inside the node have a common iSAX word [1,01,0]. If the number of series in this node is more than th, it will become an internal node and split the series inside. With the binary split, it will choose one segment (say, segment 2) and double the cardinality of the iSAX word on this segment. That is, the iSAX words of the two child nodes

[2] A special case for the symbol of iSAX word is $*$, at which segment we use only one symbol $*$ ($c = 1$) to represent the whole value range.

are [1,010,0], [1,011,0] (see Fig. 1c). Then the series inside the parent node is split into these two child nodes according to the value of the second segment.

The second splitting strategy is the full split, where the cardinalities of all the segments are doubled and thus at most 2^w child nodes are produced, i.e., [10,010,00], [10,010,01] and so on (see Fig. 1d). Similarly, the child nodes occupy disjoint subspaces of the parent subspace so that the series inside the parent node will go into the unique child node. This splitting strategy leads to a completely full-ary structure, adopted by the recently proposed iSAX-family index, TARDIS [87].



(a) Skewed region (b) Even region

Fig. 2 Illustration of the adverse effect of skewed splits to the intra-node series proximity. Series b and c are similar to one another, while series a is dissimilar to them. In subfigure (**a**), series a and b are wrongly grouped in node 1-011-0, whereas in subfigure (**b**) b and c are correctly grouped in node 10-01-01

4 Proximity-compactness trade-off

We now present the proximity-compactness trade-off, based on the analysis of the binary and full-ary index structures. More specifically, we claim that neither of them can achieve a high leaf node fill factor (i.e., high compactness) and high intra-node series similarity (i.e., high proximity) simultaneously, which limits the index building efficiency and the approximate query accuracy.

[Proximity problem of binary structures] In a binary structure index like iSAX, the SOTA splitting strategy [13] targets to balance the number of series in the two child nodes, by choosing a segment on which the mean value is close to the breakpoint. However, this strategy leads to skewed splits: it may split on several specific segments multiple times, leading to an iSAX word with several very high-granularity and other very low-granularity segments. This situation is depicted in Fig. 2a, where segment 2 has been split three times, while the other segments only once. Choosing segment 2 may be the best choice for the parent node, yet, this choice is not beneficial for the overall proximity of the series inside the child node. In our example, the series b and c are similar overall, but not grouped together due to the slight difference in segment 2, whereas the distant series a and b are grouped into the same node.

Intuitively, this happens because the split decision considers the similarity of the series in an individual segment (segment 2 in Fig. 2a), while proximity is determined by the overall similarity among series across all segments. In other words, all segments should be of approximately the same granularity to better reason about similarity (or equivalently, proximity). On the contrary, a node with a more even subdivision as in Fig. 2b, will successfully group series b and c together. It is important that, given a binary fanout, no splitting strategy can provide balanced splits while avoiding the skewness problem. Thus, binary fanout structures inherently suffer from the proximity problem.

[Compactness problem of full-ary structures] Contrary to the binary fanout, a full-ary structure [87] splits a node on all segments. Hence, it intrinsically avoids the skewness problem by creating a strictly even region. However, it quickly generates too many small nodes with low fill factors, severely damaging the index compactness. Table 1 in our experiments shows the fill factor of a full-ary structure (TARDIS) is below 0.5% on four public large datasets. Consequently, the resulting index cannot provide enough candidate series in approximate search, leading to low accuracy when visiting a handful of nodes. In terms of efficiency, although merging subtrees into larger partitions can significantly reduce random I/Os, storing and loading the enormous structure in a partition file incurs heavy overhead on index building and querying, let alone such dense node packs almost prevent further insertions.

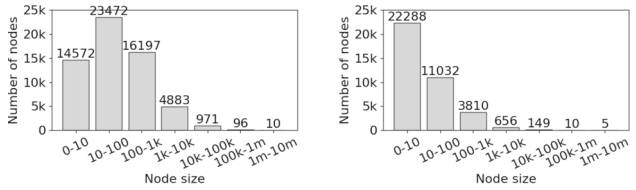
5 Dumpy

In this section, we introduce Dumpy. Based on a novel adaptive multi-ary structure, Dumpy can hit the right balance of the proximity-compactness trade-off.

5.1 Index structure and design overview

Dumpy organizes data series hierarchically and adopts top-down inserting and splitting as in other SAX-based indexes. Once a node N is full (its size c_N exceeds the leaf node capacity th), Dumpy adaptively selects λ_N segments and splits node N on these segments to generate child nodes. So the fanout of $N \leq 2^{\lambda_N}$.

We demonstrate an example Dumpy tree with $w = 4$ segments in Fig. 4. The internal node of Dumpy N maintains a list of chosen segments, $csl(N) = [cs_1, cs_2, \ldots, cs_{\lambda_N}]$ where cs_i is the id of segments (numbered from 1 to w), and $csl(N)$ is sorted by the id of segments in ascending order. When we concatenate the increased bit of each symbol on $csl(N)$, we can get a λ_N-length bit-code, denoted by sid in Dumpy.



(a) Random walk (b) DNA

Fig. 3 Node size distribution in the first layer on two 100 GB datasets ($w = 16$)

In the physical layout, a leaf node corresponds to continuous disk pages storing the raw series and SAX words. An internal node maintains a hash table to support tree traversal, named *routing table*, mapping *sid* to its corresponding child node.

We now present the intuitions behind our adaptive node splitting algorithm. To find the best balance between the proximity-compactness trade-off, we design an objective function to evaluate each possible split plan, where we use the variance of data on certain subspace to estimate the proximity of series inside child nodes and use the variance of fill factors of leaf nodes to surrogate the compactness. Considering the whole search space is 2^w+1, we first eliminate unpromising plans and then employ the relationship between different split plans to accelerate searching (cr. Sect. 5.3).

To better facilitate our adaptive splitting algorithm, we propose a new index-building workflow based on the information of all series (cr. Sect. 5.2). The building workflow of previous SAX-based indexes split a node once it is *just* full, i.e., the th+1 series arrives. Considering a node in the index may be mapped by much more series than th, the conventional split decisions will lose effect as the first $th + 1$ series soon become a small portion of all series falling into this node.

Last but not least, even if supported by the optimal adaptive splits, there might still exist a large number of small leaf nodes. This is coming from the fact that data series, similar to high-dimensional vectors, are usually unevenly distributed, generating many different dense and sparse regions [45]. Figure 3 shows the node size distribution in the first layer of iSAX-type indexes. >60% nodes in Rand and >80% nodes in DNA have <100 series while <2% nodes cover 80% series. To fully avoid this problem, we propose a novel leaf node packing algorithm, to provide high-quality leaf packs by bounding the maximal demotion bits of them (cr. Sect. 5.4).

5.2 Workflow of dumpy building

The index-building workflow of Dumpy is demonstrated in Fig. 4. Dumpy follows the most advanced building framework of the iSAX-family index [91] but changes two key designs. The classical framework is a two-pass procedure. In

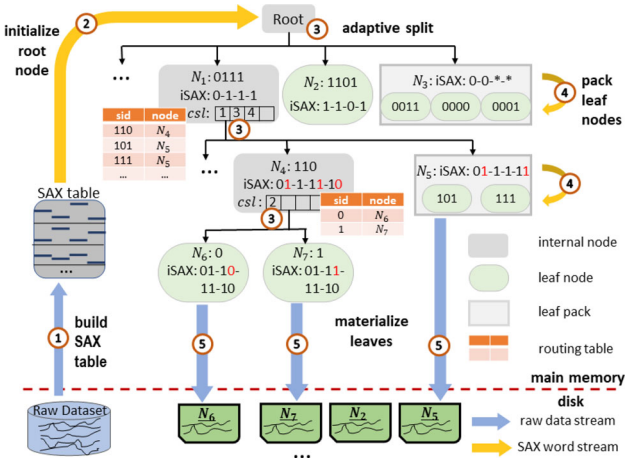


Fig. 4 Index structure (w=4) and building workflow

the first pass, it reads data series in batch from the raw dataset and computes the SAX words of each series. Then the SAX words are inserted into the destination leaf node one by one and nodes will be split once it is full. After the first pass, the index structure is in its final form. In the second pass, data series are again read in batch, routed to the correct leaf nodes, and written to corresponding files finally.

[Split nodes using a complete SAX word table] One key point of this framework is to only keep the SAX words in the index (the first pass) and use the SAX words to split nodes, which takes full advantage of the static property of iSAX summarization and significantly reduces disk I/Os. Dumpy further extends this workflow by separating the SAX words collection and node splitting into two non-overlapping steps, i.e., only collecting all SAX words into a SAX table in the first pass and then using the SAX table to build the index structure before the second pass. Hence, when splitting a node, we know the exact size and the distribution of the series inside it, making our adaptive splitting algorithm take effect actually as we expect.

[Write to disk after leaf node packing] A large fanout usually generates numerous small nodes before leaf node packing, as shown in Fig. 3. Considering in the second pass we flush the series of each relevant leaf node in a batch, the number of leaf nodes approximately decides how many random disk writes per batch. To reduce random writes, before materializing leaves as in the second pass, Dumpy merges sibling small leaf nodes in a proper way to be bigger packs and builds a routing table for the internal node. Then in the second pass, the series will directly be routed to the leaf pack by the routing table, largely reducing the random disk writes.

5.3 Adaptive node splitting

We now present our adaptive strategy of determining fanouts and splits (on which segments) based on the SAX words

of all relevant series. Our strategy is to select the best split plan based on a novel objective function, which considers the proximity of series inside child nodes and the compactness of child nodes at the same time. Since the number of all possible split plans is as large as $2^w - 1$, we also propose an efficient search algorithm by restricting the candidate space and reusing the shared information.

5.3.1 Objective function

Our objective function targets to achieve the best balance between the trade-off of proximity and compactness. We measure the proximity based on the average variances of data on candidate segments. To measure the compactness of the children nodes after a split, we consider both the variance of fill factors of child nodes and the ratio of overflowed nodes to pursue a balanced split and avoid bias for small or large fanouts.

Given a node N containing c_N series $\mathcal{X}_N = \{\boldsymbol{x^1}, \boldsymbol{x^2}, \ldots, \boldsymbol{x^{c_N}}\}$ where $\boldsymbol{x^i} = SAX(s^i, w, c)$ is the SAX word of series s^i and x^i_j is j-th symbol of $\boldsymbol{x^i}$, and a split plan $csl(N) = [cs_1, cs_2, \ldots, cs_{|csl(N)|}]$, we first project each series of $\mathcal{X}_N$ onto the segments of $csl(N)$ and get $\mathcal{X'}_N$, that is, $x'^i_j = x^i_{cs_j}$ for any i and $1 \le j \le |csl(N)|$. Then our objective function is as follows:

$$\max_{csl(N)} \; e^{\sqrt{\frac{1}{|csl(N)|} Var(\mathcal{X'}_N)}} + \alpha * e^{-(1+o)\sigma_{\boldsymbol{F}}} \tag{1}$$

where e is the Euler's number, the variance of projected data is defined as $Var(\mathcal{X'}_N) = \frac{1}{c_N} \sum_{i=1}^{c_N} \|\boldsymbol{x'^i} - \boldsymbol{\mu}\|^2$ and $\boldsymbol{\mu}$ is a vector of mean values of data on each chosen segments,[3] $o \in [0, 1]$ is the ratio of overflowed child nodes (size $> th$), $\sigma_{\boldsymbol{F}}$ is the standard deviation of the fill factors of child nodes, and α is a weight factor to balance the influence of these two measurements.

The first term estimates the proximity of a split plan. It evaluates the average variance of relevant data series on the projected SAX space, which is equivalent to the average distance of all the data series to their centroid, i.e., the mean vector $\boldsymbol{\mu}$. The variance is an indicator of data informativeness on certain dimension [10, 34, 57], considering large variances usually mean large information entropy [66]. Since different plans may choose different numbers of segments, we divide the variance by the number of chosen segments to make the evaluation fair.

The second term is to evaluate the compactness of a split plan. The standard deviation of fill factors of child nodes prevents extremely imbalanced splits and avoids the severe data skewness like Fig. 3: the value will be very large in this case. Informally, the vector of fill factors is defined as $\boldsymbol{F} = (F_1, F_2, \ldots, F_{2^{|csl(N)|}})$ where $F_i = c_{N_i}/th$ and N_i is the i-th child node. However, it shows bias for small fanout, which generates fewer but larger child nodes and leads to an unnecessary deep tree. To resolve this problem, we add a penalty term $(1+o)$ that uses the ratio of overflowed child nodes to avoid the bias for plans of small fanout.

5.3.2 Find the optimal split plan

To reduce the complexity of finding the optimal split plan under our objective function, we propose a novel searching algorithm composed of three practical speedup techniques, that are, pre-computing the variance for each segment, restricting the search space by a user-defined fill-factor range, and hierarchically computing the sizes of child nodes.

[Pre-compute variance] We find that in the first term of the objective function, $Var(\mathcal{X'}_N)$ can be computed by linearly accumulating the variance of data on each segment.

$$Var(\mathcal{X'}_N) = \sum_{cs \in csl(N)} Var(\Pi_{cs}(\mathcal{X}_N)) \tag{2}$$

where $\Pi_{cs}(\mathcal{X}_N)$ indicates the projection of $\mathcal{X}_N$ onto segment cs.

Hence, we can pre-compute the variance of data series on each segment when we start to split a node. When evaluating a specific plan, we simply fetch the corresponding segments' variances and sum them up with constant complexity.

[Restrict the search space] The second speedup technique is to restrict the average fill factor of child nodes to be in a reasonable range and avoid the particular evaluation. We introduce a pair of parameters F_l, F_r to bound the average fill factor of child nodes. Then the range of the number of chosen segments $|csl(N)|$ can be deduced as

$$\max(1, \log \frac{c_N}{F_r * th}) \le |csl(N)| \le \min(w, \log \frac{c_N}{F_l * th}) \tag{3}$$

In practice, we empirically set $F_l = 50\%$ and $F_r = 300\%$, which generally achieves 16x speedup and 99% accuracy on average.

[Hierarchically compute sizes of child nodes] So far for each plan, we still need to iterate all the data series to get the sizes of child nodes. If a split plan $csl^i(N)$ is a subset of another plan $csl^j(N)$, then the size distribution of child nodes of plan $csl^j(N)$ can be reused to calculate the distribution of plan $csl^i(N)$. Since the whole w segments are a superset of split plans, we first compute child node sizes for w segments as a base distribution in each split and then traverse other plans in a depth-first manner, starting from the plan with the largest fanout to the smallest. Hence, we can reuse the size

[3] We use the midpoint of the range represented by the SAX symbol to calculate the mean value and other statistics.

distribution we have gained in a hierarchical way and avoids traversing all the series for each plan.

5.4 Leaf node packing

In this subsection, we propose a simple yet effective algorithm to pack small leaf nodes without losing the pruning ability of packed nodes. The intuition is to minimize the value range in the SAX space occupied by the packed nodes, i.e., make them have the tightest iSAX representation. Tighter iSAX representation directly translates to higher pruning power. We define the demotion bits as the different bits between the sids of two or more nodes considered to be merged into the same pack. In our node packing algorithm, we limit the number of demotion bits to be smaller than $\rho\lambda$, where ρ is a user-defined parameter trading off pack quality and fill factor. Specifically, given a list of packs and a small node N to be packed, we decide N's belonging by the demotion cost, which is defined as the increased number of demotion bits of the pack if we add N into it. A leaf node pack forbids any node's insertion request if it will make the pack demote more than $\rho\lambda$ bits or overflow (size $> th$). Finally, if no existing pack can satisfy the requirements to insert N, we will create a new pack and insert N into it. The details of our leaf packing algorithm can be found in [81].

5.5 Search algorithm

Dumpy supports two styles of query-answering algorithms. The first style follows the classical pruning-based search algorithm [28]. As a SAX-based index, Dumpy can conduct an efficient search (including the exact, δ-ϵ-approximate search and etc. [28, 67]) by pruning irrelevant leaf nodes using lower-bounding distances of iSAX words [28, 67]. Besides that, Dumpy also supports traditional approximate search, i.e., querying one target leaf node. Moreover, we extend it to allow searching more nodes, called *extended approximate search*, to improve query answer quality while maintaining response time in milliseconds. We limit the search range of extended approximate search in the smallest subtree of the target leaf node to reduce the complexity and avoid traversing the whole tree and evaluating the nodes one by one as in the bound-based search style. Benefiting from Dumpy's multi-ary structure and fill factor, it brings prominent improvement in search accuracy. The details of our algorithm is shown in Algorithm 1.

Algorithm 1 Extended Approximate Search

Input: root node N_r, node number nbr, query series q
1: node $N = N_r$
2: **while** N!= null and N.leafNbr $> nbr$ **do**
3: $sid = promoteiSAX(iSAX(N), SAX(q), csl(N))$
4: $N = N.routingtable[sid]$
5: sort N's siblings according to lower bound distance into list l
6: **while** number of searched nodes $< nbr$ **do**
7: N_c = pop the head node of l
8: fetch all nodes rooted at N_c and search the series inside
9: **return** kNN among the visited series

5.6 Updates

As a fully functional index, Dumpy also supports updates (insertion and deletion) besides bulk loading. For insertion, we first insert the series to the target leaf node. If the leaf overflows, we read all the SAX words inside and follow the same workflow of index building. When the query series falls into a full pack, we extract the target leaf node in the pack and redo the node packing for the siblings after a large number of such extractions. These operations are very fast since these small nodes usually cover a small number of series.

The deletion is almost the same as the iSAX-index family [67, 90]. In particular, we mark the deleted data series in the corresponding leaf via a bit-vector and further insertions can exploit the space occupied by the deleted series while queries ignore these entries. When a node is empty, we free the occupied space. The only difference for Dumpy is to update the routing table.

5.7 Complexity analysis

In this section, we first analyze the time complexity of index building and querying, and then analyze the space complexity.

[Time complexity] As a disk-based index, the time cost of Dumpy depends on both in-core complexity and disk accesses. In the following, we first discuss the theoretical time cost in index building and then querying.

The complexity of Dumpy index building could be summed over sub-modules. In the adaptive split algorithm, let node N is to be split, and $|csl(N)|$ is between λ_{min} and λ_{max} according to Eq. 3. Then the cost of computing the variance of each segment, the base distribution and the routing target is $O(wc_N)$. The calling number of the $calcDist$ function is $\binom{w}{\lambda_{max}} * 2^w + \sum_{i=\lambda_{min}}^{\lambda_{max}-1} \binom{w}{i} 2^{\lambda_{i+1}} = O(2^w)$, where the first term corresponds to evaluating all possible plans of the max fanout, i.e., using λ_{max} segments, and the second term to evaluating all possible plans of smaller fanouts. In the leaf node packing algorithm, given that the final pack number is np, the time complexity of node packing is $O(2^{\lambda_N} * np)$. In summary, the total in-core complexity is $O(\sum_N (wc_N + 2^w + 2^{\lambda_N} * np))$.

Random disk writes can have a significant cost when building Dumpy (cf. Fig. 4, Stage 5). Assume the number of data series in the database is $|db|$, the number of leaf nodes is n_l and the memory buffer can contain B series. In each batch, Dumpy generates n_l random writes at most, and in total, $O(\frac{|db|}{B} * n_l)$ random writes for the whole index building.

For querying, the approximate search goes down a single path from the root node to a target leaf node. Let the length of this path be $|p|$; then the cost is $O(|p|w)$. The I/O cost is a single disk read of size $O(th)$. Compared to the iSAX indexes (with binary fanout), the length $|p|$ of the Dumpy path is $2/\bar{\lambda}$x smaller, where $\bar{\lambda}$ denotes Dumpy's average fanout. In addition to the target leaf node search cost, the complexity of the exact search comprises of $O((1-pr)*n_l)$ random disk reads of size $O(th)$, where pr is the pruning ratio, and $O(w*n_{total}\log n_{total})$ in-core calculations, where n_{total} is the total number of nodes.

In practice, Dumpy is a more compact index (smaller n_l and n_{total} values) than other SAX-based indexes (cf. Sect. 9.1), and therefore, faster in both building and querying times.

[Space complexity] The space occupied by Dumpy (in addition to the raw data size) is as follows. The SAX words are persisted on disk, occupying $\lceil wb*|db|/8\rceil$ bytes. The internal nodes of the index store the routing table, the iSAX word, and the list of segments used in the split (i.e., the chosen segments), for a total of $\sum_N(8*2^{\lambda_N}+wb/8+\lambda_N)$ bytes. The leaf nodes store a single iSAX word summary, for a total of $n_l*(wb/8)$ bytes. Since the number of nodes is small, Dumpy introduces very little additional storage in practice.

6 Dumpy-fuzzy

As partition-based indexes, data series indexes also suffer from the so-called *boundary issue* in approximate search [19, 32]. That is, the data series located near the boundary of a query's resident node are also good candidates, but cannot be considered in approximate search since they may be located in different subtrees. To overcome this problem, we propose a variant of Dumpy, named Dumpy-Fuzzy, that views the static partition boundary (i.e., the SAX breakpoints) as a range (fuzzy boundary) and places the series lying on this range into the nodes of both sides. Dumpy-Fuzzy further improves the approximate search accuracy compared with Dumpy at the expense of a small overhead on index building and disk space.

Specifically, Dumpy-Fuzzy adds a duplication procedure after splitting. For each newly-generated internal node N, it checks the series lying on N's neighboring nodes (i.e., the nodes whose sid is 1-bit different from N) and duplicates the series near the boundary into itself. We introduce a hyper-parameter $f\in(0,1)$, the fuzzy boundary ranges regarding the original node ranges, to control which series is qualified to be duplicated. In addition, duplication also applies after leaf node packing. The series near the boundaries of a leaf pack can also be placed redundantly into the pack in the same way as above. we ensure that no additional split will be introduced in this procedure (i.e., the leaf pack will not overflow).

Note that Dumpy-Fuzzy does not damage Dumpy's pruning power for exact search. Duplicated series do not change the iSAX words of nodes or packs. Hence, the lower bound calculations are kept the same. Therefore, without violating the pruning-based exact search, Dumpy-Fuzzy improves the approximate search accuracy by examining more promising candidates.

7 DumpyOS

In this section, we introduce DumpyOS, which extends Dumpy in a parallel way to maximize its performance under modern multi-core architectures and NVMe devices. We first introduce some preliminary materials about the characteristics of NVMe SSD devices, and present an overview of our methods in Sect. 7.1. Then, we describe the concurrent index building workflow in Sect. 7.2, and the novel parallel pruning-based query answering techniques in Sect. 7.3.

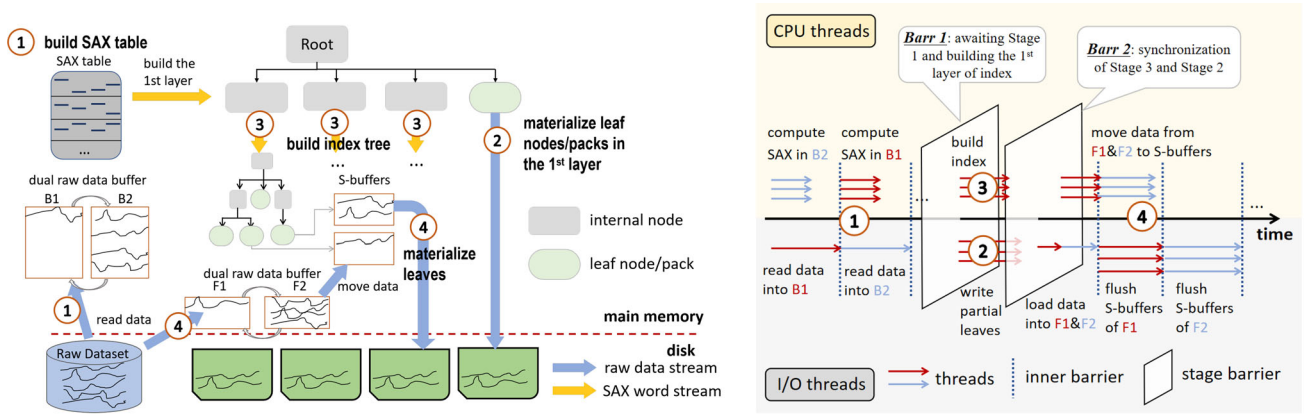
7.1 Overview

In the following, we briefly describe the characteristics of NVMe SSDs, and summarize key rules to better exploit them. Then we introduce the rationale of DumpyOS based on these key rules.

Rule 1: Issue large batches of parallel I/Os to ensure the internal parallelization is fully exploited [16, 30]. The most prominent feature of SSD is the *internal parallelism*. The storage unit of SSD (i.e., the flash memory) is organized in a highly-hierarchically manner to maximize I/O concurrency [71]. The storage units are organized into different levels (e.g., page, block, die, and channel from bottom to top). The upper levels can operate independently to serve various requests simultaneously, while the lower levels can only work in parallel if they are executing identical commands.

Rule 2: Separate read and write operations. Although parallel read and write operations are beneficial to the exploitation of SSDs, the interference between reads and writes may cause access conflicts and hence hurt the parallelism and I/O performance [33].

Rule 3: Buffer small random writes into large sequential writes. Another peculiarity of flash memory is garbage collection. Given that flash memory must be erased before it can be rewritten, with erasures at the block granularity and writes at the page granularity, the original data need to be copied to another block, known as the write amplification on SSD [21, 83]. Writes smaller than a single page actually perform a read-modify-write operation on SSD, and result in one invalid page. This leads to not only more unnecessary



(a) DumpyOS: Index building workflow

(b) DumpyOS: Index building timeline of (time scale does not reflect reality)

Fig. 5 Illustration of building DumpyOS. The inner barriers are inside a loop within a stage and we only show one iteration. The stage barrier is used for synchronization between stages

reads and writes, but also to more frequent garbage collection [1, 42].

Based on these rules, our proposed method, DumpyOS, divides the operations in index building and querying into two separate parts, i.e., I/O and CPU operations, and then adopts different parallel techniques to accelerate each one of these two parts. Furthermore, we design a dedicated workflow to overlap CPU calculation time with I/O time by interleaved threads scheduling.

7.2 Concurrent index building

To better exploit the parallelism of modern hardware, we design a new concurrent workflow for DumpyOS, shown in Fig. 5.

The DumpyOS workflow consists of four stages. In Stage 1, we build the SAX table with a dual raw data buffer (one reads data, while the other computes the SAX words [60]), and then we build the first layer of the index tree .[4] After synchronization of these threads (Barr 1), Stages 2 and 3 are executed in parallel. Stage 2 reads raw data in batches, routes them into leaf nodes, or packs in the first layer, and materializes these leaves to SSD. At the same time, in Stage 3, we employ a group of threads as index-building workers to grow the index tree from internal nodes of the first layer (one worker for each subtree). Since there is a significant size difference between different internal nodes (cf. Fig. 3), we schedule the threads in a dynamic manner. Each worker fetches an internal node, conducts adaptive splits and node packing, and then repeats this process until all the subtrees are built. Once all threads from the two stages return (Barr 2), Stage 4 starts. Similar to Stage 2, Stage 4 reads raw data and materializes other leaves (below the first layer).

Note that Stage 2 is additionally incurred to materialize the leaves in the first layer. This stage involves an I/O-bound process that utilizes the available I/O time during index structure construction (Stage 3). Although one extra pass of sequential read is introduced, it is very fast compared to leaf materialization (random writes). This stage has the additional advantage that it reduces the burden of Stage 4.

[Speedup materialization with parallel I/Os] Since data movements on secondary storage are usually the bottleneck in index construction [60, 81], improving the I/O throughput with SSDs becomes a promising direction to accelerate the leaf materialization process. Recall that the materialization process consists of three steps: (i) read a series from the raw dataset, (ii) find the leaf to which the series belongs, and (iii) append the data series to the leaf file once the memory is full.

To reduce the number of I/Os, a buffer pool is usually prepared to store the data read from the dataset. However, such a serial design does not fully use the SSD's internal parallelism. Following **Rule 1**, a naive I/O-parallel method is to divide the buffer pool into several pieces and assign each of them to a single thread, which repeats the three steps. Although this method achieves I/O parallelization, it mixes read and write requests issued from different threads, and thus, generates severe I/O conflicts among threads, leading to a degradation of the SSD performance (**Rule 2**).

Consequently, we can set barriers between reads and writes in order to isolate them. That is, we parallelize each one of the three steps with barriers between them. Specifically, we (sequentially) read a batch of data into a unified buffer pool (rather than small buffer pieces), find the target leaf nodes of the series, and finally write these leaves one by one on the SSD. For the second step, we can partition

[4] Recall that the root node always chooses all w segments without costly adaptive splitting.

the buffer in equally-sized partitions, and assign these partitions to a group of threads (one for each partition). The threads find the target leaf for each series, and a concurrent hash map (where the key is a leaf and the value is a set of pointers to the series in the buffer pool) can be used to collect the results. As for the third step, the prepared leaves can be materialized in parallel by threads without any conflicts.[5]

The writing process, which is still a bottleneck, can be further optimized. In the algorithm above, we issue one write request for a single series (≈1KB for 256-length), leading to very small writes and performance degradation (**Rule 3**). Therefore, we optimize it by buffering small writes. The second step is modified to move the full data series (rather than pointers) to the buffers of target leaves, named S-buffers. Finally, the third step can directly flush S-buffers to the SSD (i.e., one I/O request for each buffer). In this way, the internal parallelism of modern SSDs is fully leveraged and the throughput is significantly improved. However, when we move data to the S-buffers of leaves (the second step described above), an additional pass of data-moving operations is introduced in memory, which incurs a substantial overhead. In the following, we explain how we address this by interleaved thread scheduling.

[Mask CPU time with interleaved threads] As shown in Fig. 5 (Stage 4), we divide the unified memory buffer into two parts (F1 and F2). In each cycle, we first read data to fill F1. Next, a group of CPU-bound threads is initiated as routing workers to move data from F1 to the corresponding S-buffers. In the meanwhile, another I/O-bound thread reads data into F2 (interleaving read and computing). We set an inner barrier here to await the completion of these threads. Then, the routing workers start to move data from F2 to the S-buffers, while we start a group of I/O threads to flush the prepared S-buffers originating from F1 (interleaving writes and computing). Another inner barrier is set here to synchronize these threads. After that, the threads flush the S-buffers from F2. This process is repeated until a full pass of the raw dataset is completed. In this way, we mask the extra CPU time and always keep the SSD busy with maximal throughput.

7.3 Parallel pruning-based query answering

In this subsection, we focus on accelerating the pruning-based query algorithm based on modern parallel hardware. SOTA parallel query algorithms like PARIS+ [60] and Coconut [43] achieve speedup by serially scanning the dataset. However, this search style has three limitations: (1) it cannot serve approximate queries like ng- and $\delta\epsilon$-search [28]; (2) it is impractical to always maintain a sequential data layout and aligned in-memory SAX table for a dynamic workload in a real situation; and (3) it requires additional space in the memory and secondary storage to store the SAX table and dataset, respectively, which limits scalability.

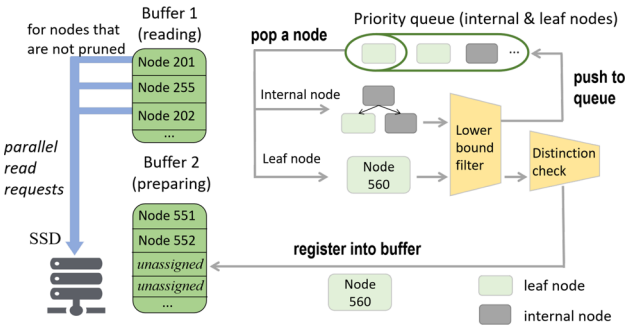


Fig. 6 An illustration of parallel pruning-based query algorithm of DumpyOS. It can also work on Dumpy on HDD to mask the CPU time, but the I/O time is still the bottleneck

In our case, we decided to optimize the classical pruning-based query algorithm to achieve higher performance, while avoiding these limitations. Our core idea is shown in Fig. 6. We divide the threads into two groups, loading and computing workers respectively. Each group gets allocated a memory buffer at a time that contains η slots to accommodate leaf nodes or packs (η is a user-defined parameter depending on the parallelism of the SSD). First, we initialize a priority queue with the root node (ordered by the lower bound distance in ascent). Then one computing worker takes a node from the priority queue and if it is an internal node, then we perform a qualification check on its children by comparing their lower bound distances to the query. Qualified children nodes are pushed into the queue. If the node is a leaf that has not been visited before, it will be registered into a slot of the current buffer (i.e., Preparing Buffer), along with the file position of this leaf. Once the buffer is filled with η leaves, the computing workers will wait for loading workers, who are meanwhile reading raw data from data files into the other buffer (i.e., Reading Buffer), which has been filled with candidate leaves beforehand. In this way, we can parallelize the read requests in querying to increase the I/O performance (**Rule 1**). Once loading workers finish, we exchange the two buffers, and immediately let loading workers read data for the new buffer. Consequently, we wake the computing workers to compute the distances of the data-ready buffer in parallel with SIMD techniques [59], and update current BSF answers. Finally, the buffer is cleared and the above process is repeated until the queue head violates the condition of the lower bound filter. When this happens, it is impossible to produce better answers from the nodes of the queue; for the nodes already in the buffer, we will still fetch the data from SSDs and compute distances for them before terminating the algorithm. Overall, in our algorithm, both CPU computations and I/O requests are

[5] There are many ways to implement parallel I/O requests (e.g., asynchronous I/O with io_uring [5] or libaio [29]). For simplicity and clarity, we use multi threads to issue I/O requests in this paper.

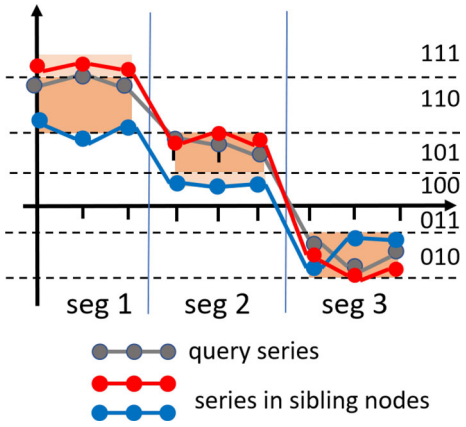


Fig. 7 Illustration of the downside of Dumpy-Fuzzy. The red series is closer to the query series than the blue one

fully parallelized, enjoying the performance improvements stemming from modern hardware.

8 DumpyOS-F

In this section, we introduce an enhanced algorithm for extended approximate search, DumpyOS-F. Compared to Dumpy-Fuzzy, the most prominent characteristic of DumpyOS-F is that it does not need any modification of Dumpy's structure, and thus no additional space cost. At the same time, it provides more accurate results.

While Dumpy-Fuzzy mitigates the boundary issue by physically duplicating series into neighboring subtrees or nodes, it still has three limitations that hinder the improvement of search accuracy:
(1) The number of duplicated series is limited to prevent excessive dilation of the original index structure. The duplication in a leaf node or pack is restricted to avoid overflow by abandoning checking other neighboring series despite a higher similarity. The selection of duplicated series is random since it is not possible to know which series is closer to the query during indexing. For example, in Fig. 7, we cannot decide whether it would be better to duplicate the red, or the blue series into the node, without knowing the query series. This leads to possible loss of promising candidate series.
(2) The similarity between duplicated series and the query is limited. As shown in Fig. 7, consider that the capacity of the node is sufficient, and thus both the red and blue series are contained in this node. Given that the blue series is not as close to the grey query series as the red one, duplicating the blue series does not provide any benefit for this query. Nevertheless, Dumpy-Fuzzy will duplicate the blue series, because of its proximity to the middle segment (seg 2) of the node.
(3) The same series may be checked multiple times. Since the duplicated series come from the sibling nodes, given our extended search Algorithm 1, it is possible to obtain the same series during searching (e.g., the red series exists in two nodes and can be visited twice), making the duplication meaningless in this case.

In summary, since Dumpy-Fuzzy statically selects similar series for a node to duplicate during indexing, it cannot accurately identify the series that are similar to a given query series, yet lying in different nodes.

In DumpyOS-F, we determine the candidate series during querying rather than indexing. DumpyOS-F estimates the distance between the query series and sibling nodes on-the-fly, and further prioritizes these nodes. That is, we only check the series in the fuzzy boundary that are close to the query itself rather than the node where the query lies. As a result, DumpyOS-F is more accurate and efficient in approximate search, and most importantly, it works directly on Dumpy's index structure. Therefore, users do not need to build a specialized index for approximate search like Dumpy-Fuzzy.

As shown in Fig. 8 (left), DumpyOS-F accesses three kinds of nodes in search: normal, fuzzy, and sibling nodes. Normal nodes are the nodes we visit when routing a query series from the root to a leaf node. That is, the query series is located in the range of normal nodes. At each layer, we mark the nodes that are close to the query series as fuzzy nodes. Specifically, we compute the distance between the PAA value of the query series and the corresponding breakpoint at each segment, and check whether the query series is located in the fuzzy boundary of a sibling node (according to parameter f). If so, the sibling node is marked as a fuzzy node. For example, in Fig. 8 (right), the second segment of the query series PAA is very close to the breakpoint. Thus, among all sibling nodes, we select the one that only differs from the normal node (marked in orange) on the *second* segment: this selected node becomes the fuzzy node (marked in green). Finally, some other siblings of the normal nodes are marked as sibling nodes that will also be visited (as described in Algorithm 1), which are shown in blue in Fig. 8.

The pseudocodes of DumpyOS-F are shown in Algorithm 2. In the first step (lines 1–16), we route the query series to the target leaf, while collecting the fuzzy nodes at each layer into a priority queue according to their estimated distances in ascent (lines 6–12). Since a pack containing multiple small leaves may be put into the queue many times, among these entries, we only keep the one with the smallest estimated distance (line 15). In the second step (lines 14–21), we search the target leaf and then pop the fuzzy nodes from the priority queue (line 17). If the popped fuzzy node is an internal node, we called the adapted routing algorithm (Algorithm 3) to obtain the nearest leaf to the query series and search it (line 19). If the popped node is a leaf, we will directly search the series inside (line 20). Finally, if the num-

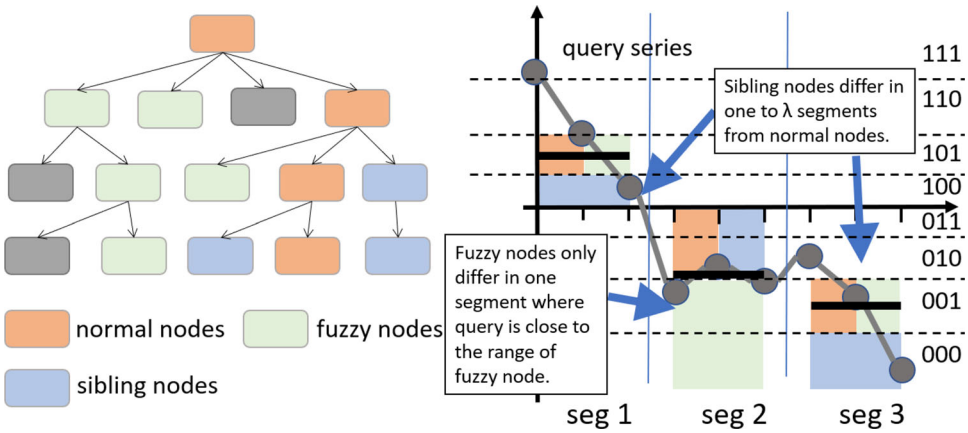


Fig. 8 Extended approximate search with DumpyOS-F. (left)Three kinds of nodes visited during searching. (right) Illustration of fuzzy and sibling nodes; cells marked in two colors indicate that they are occupied by two nodes

ber of accessed nodes is still within budget, we search sibling nodes in a bottom-up fashion by calling Algorithm 1.

The adapted routing algorithm is shown in Algorithm 3. In line 1, we first identify the special segment, where the fuzzy node's iSAX label is not a prefix of the query's corresponding SAX label, and use bit b=1 (line 2) to record the (series inside the) node is smaller than the query in this segment, b=0 otherwise. For example, consider the green fuzzy node in Fig. 8 (right) as N. Then the second segment will be identified since the node's second iSAX symbol 00 is not a prefix of the query's 010. b is set to 1 as N is below the query series in this segment. Then, if N is split on the second segment, we always route the query series to the upper child (i.e. whose second iSAX symbol is 00**1**), because it is closer to the query than the lower child. To achieve this, we slightly modify the iSAX promotion algorithm as shown in Function PromoteiSAXF, where we always set the bit of the special segment to the value b. Since at each layer we have at most λ fuzzy nodes and in the worst case they are all internal nodes, then the extra time cost is $O(|p| * \bar{\lambda})$, which is negligible compared to node loading and distance calculation.

Algorithm 2 DumpyOS-F

Input: root node N_r, node number nbr, query series q, fuzzy boundary parameter f

```
1: node N = N_r
2: initialize an empty priority queue pq
3: while N is an internal node do
4:   sid = promoteiSAX(iSAX(N), SAX(q), csl(N))
5:   N_c = N.routingtable[sid]
6:   for each segment seg in csl(N) do
7:     bp = next breakpoint in seg
8:     r = value range of iSAX(N_c)[seg]
9:     es_dis = |PAA(q)[seg] − bp|
10:    if es_dis < f * r then
11:      sib_id = flip the bit at seg of sid
12:      push < N.routingtable[sib_id], es_dis > into pq
13:  N = N_c
14: search node N
15: remove the entries in pq that point to the same node but have larger distances
16: while pq is not empty and nbr > 1 do
17:   < N_c, _ > = pop the first element from pq
18:   if N_c is an internal node then
19:     N_c = AdaptedRouting(N_c, q)
20:   search node N_c
21:   nbr− = 1
22: ExtendedApproximateSearch(N_r, nbr, q) while skipping the visited nodes
23: return kNN among the visited series
```

9 Experiments

[Environment] Experiments were conducted on an Intel Core(R) i9-10900K 2.80 GHz 10-core CPU with 4*32 GB 2400 MHz main memory, running Windows Subsystem of Linux (Ubuntu Linux 20.04 LTS). The machine has a Samsung PCIe 2TB SSD (default), and a Seagate SATAIII 7200RPM 2TB HDD. Our codes are available at https://github.com/DSM-fudan/DumpyOS.

[Datasets] We use one synthetic and three real datasets. All series are z-normalized before indexing and querying. In each dataset, we prepare 200 queries that are not part of the dataset of varying hardness [93], and obtain the ground truth kNN results using brute-force search. **Rand** is a synthetic dataset, generated as cumulative sums of random walk steps following $N(0, 1)$. It has been extensively used in the existing works [2, 27, 28, 92]. We generate 50–800 million Rand series of different lengths (50–800 GB). **DNA** [52] is a real dataset collected from DNA sequences of two plants, Allium sativum and Taxus wallichiana. It comprises 26 million data series of length 1024 (∼113 GB). The second real dataset, **ECG** (Electrocardiography), is extracted from the MIMIC-III Waveform Database [39]. It contains over 97 million series of length 320 (∼117 GB), sampled at 125 Hz from 6146 ICU patients. The last real dataset, **Deep** [73], comprises 1 billion vectors of size 96, extracted from the last CNN (convolutional neural network) layers of images.

Algorithm 3 Adapted Routing

Input: fuzzy node N, query series q
1: seg = the segment where $iSAX(N)[seg]$ is not a prefix of $SAX(q)$
2: b = the inversion of the last bit at $iSAX(N)[seg]$
3: **while** N is an internal node **do**
4: sid=$PromoteiSAXF(iSAX(N),SAX(q),csl(N),seg,b)$
5: $N = N.routingtable[sid]$
6: **return** N

Function PromoteiSAXF(iSAX word $isax$, SAX word sax, chosen segments list csl, segment s, fixed symbol b)
7: $sid = 0$
8: **for** each segment seg in csl **do**
9: $nb = len(isax[seg])$
10: **if** $seg = s$ **then**
11: $sid = (sid << 1) + b$
12: **else**
13: sid = ($sid << 1$) + the $(nb + 1)$-th bit of $sax[seg]$
14: **return** sid

[Algorithms] In iSAX-index family, we take **iSAX2+** as the SOTA binary structure [28]. We also implement a stand-alone version of **TARDIS** as the SOTA full-ary structure and use 100% sampling percent. **DSTree** [78] is also included as one of the SOTA data series indexes [28]. For simplicity, Dumpy-Fuzzy with parameter f is abbreviated as **Dumpy-f**. To evaluate the quality of these indexes, we implement extended approximate search, as well as pruning-based search on them. We also include PARIS+ [60] as the SOTA parallel (serial-scan-based) disk index to be compared to DumpyOS.

All the codes are open-source, implemented in C and C++, compiled by g++ 9.4.0 with -O3 optimization.

[Parameters] We set the number of segments $w = 16$, SAX cardinality $c = 64$ (i.e., $b = 8$), $\alpha = 0.2$, and the leaf size threshold $th = 10000$. The replication factor of each series in Dumpy-f is set to at most 3, and f is set to 10 for ECG and Deep and 30 for Rand and DNA. The discussion about the influence and the selection of parameters can be found in [81]. The memory buffer size for index building is set to 4 GB unless specified. The default number of threads for DumpyOS is set to 5 for index building and 8 for querying. The queue size η is set to 24 for our SSD.

[Measures] Similar with other works [4, 28, 57], we use Mean Average Precision (MAP) [72] as the accuracy measure, which is defined as the mean value of AP on a group of queries. For query s_q, AP equals to $\frac{1}{k}\sum_{i=1}^{k} P(s_q, i) * rel(i)$, where $P(s_q, i)$ is the ratio of true neighbors among the top-i nearest results and $rel(i)$ is 1 if the i-th nearest result is the true exact kNN result and 0 otherwise. It can be proved that MAP is equivalent to the average recall rate when the returned results are sorted by the actual distances. Another similarity measure we use is the average error ratio which measures the difference between approximate and exact results, commonly used in approximate search [4, 87], and defined as $\frac{1}{k}\sum_{i=1}^{k} \frac{dist(a_i, s_q)}{dist(r_i, s_q)}$. We measure both ED and DTW, where the DTW warping window size is set to 10% of the series length as a common setting [59, 63].

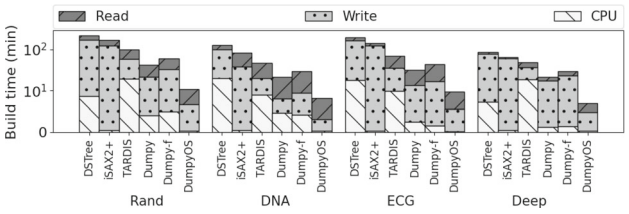


Fig. 9 Index building time across four datasets on SSD

Table 1 Index structure statistics

Data	Method	#Leaves	#Nodes	Ht	Fill factor	Size[a]
Rand	iSAX2+	73563	86945	20	13.59%	16
	DSTree	17847	35693	32	56.03%	9
	TARDIS	8516867	8520065	**3**	0.11%	732
	Dumpy	**14106**	**19418**	7	**70.89%**	**3**
DNA	iSAX2+	42906	47885	25	6.14%	9
	DSTree	5833	11665	43	45.16%	3
	TARDIS	1011436	1312989	**5**	0.26%	278
	Dumpy	**4367**	**6228**	9	**60.32%**	**1**
ECG	iSAX2+	69786	74042	9	13.98%	14
	DSTree	20740	41479	48	47.04%	13
	TARDIS	3178628	3182368	**4**	0.33%	749
	Dumpy	**12112**	**15050**	7	**80.55%**	**3**
Deep	iSAX2+	68096	71188	8	19.08%	11
	DSTree	16324	32647	33	61.26%	8
	TARDIS	824458	827094	**3**	0.27%	546
	Dumpy	**11590**	**13664**	8	**86.28%**	**3**

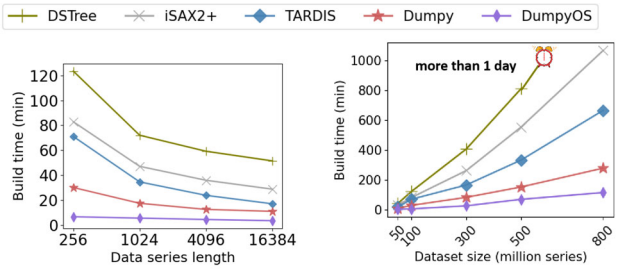
The best method is marked in bold
[a] Size of in-memory index structure only, in MB unit

9.1 Index building

9.1.1 Efficiency

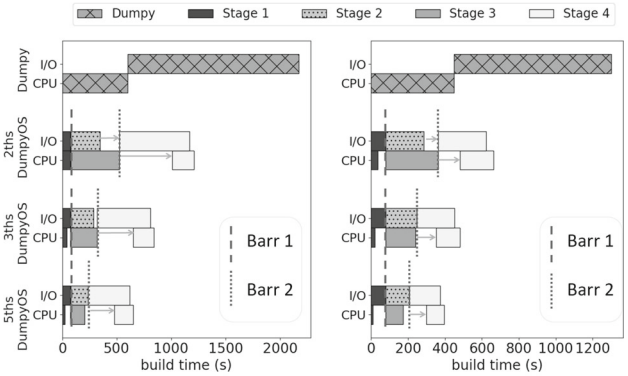
First, we evaluate the index-building efficiency in four datasets on SSD, and the results are shown in Fig. 9. In all four datasets, Dumpy outperforms the other three methods by a large margin, i.e, 5.3 times faster than DSTree, 3.8 times than iSAX2+ and 2.5 times than TARDIS on average. Dumpy-f only incurs small overheads (about **38%**) on Dumpy and is considerably faster than DSTree and iSAX2+. Moreover, DumpyOS is **3.7** times faster than Dumpy (with 5 threads). Note that in this figure, the CPU time is only for the part that does not overlap with I/O time. (A detailed analysis of DumpyOS will be presented in Sect. 9.1.3.)

We present the detailed index information in Table 1. Dumpy has the fewest leaf nodes (i.e., the highest fill factor), which verifies the good compactness of Dumpy. The num-



(a) On data series length (b) On dataset size

Fig. 10 Index scalability (32 GB memory)



(a) Rand (b) DNA

Fig. 11 Parallel building with DumpyOS with 2, 3, and 5 threads, on two 100 GB datasets (Rand and DNA). Dotted lines denote stage barriers. Grey arrows indicate that the corresponding threads are waiting for synchronization (inner barriers), and for clarity, we do not show the detailed timelines between inner barriers. That is, the I/O and the CPU time of each stage are summed as a complete period

ber of leaf nodes of DSTree is slightly larger than Dumpy, and that of iSAX2+ is generally >**3**x more than DSTree. As for the full-ary structure like TARDIS, it generates millions of leaves and has a low fill factor in the leaves, as analyzed in Sect. 4. These nodes are further packed into large partitions of capacity 128MB, as the setting of the original paper [87]. These results verify the space complexity analysis in Sect. 5.2.

9.1.2 Scalability

Next, we test the scalability in Rand datasets by increasing data size from 50 to 800 GB, and series length from 256 to 16384. When the dataset size is varied, the series length is kept constant at 256, whereas the dataset size is kept at 100 GB when the length is varied, as the same design with the benchmark [27].

Figure 10 presents the index building time with 32 GB memory. Dumpy has the best scalability in both cases. In a linear regression test for the building time and dataset size, Dumpy's coefficient of determination R^2 is **0.99**, verifying its linear growth of the building time. The reason is that the number of leaves increases linearly as the dataset scales up, indicating a nearly constant average fill factor. This also supports the complexity analysis. The performance when varying series length also follows this rule. Besides that, DumpyOS shows a stable reduction in indexing time as the dataset size increases (about **3** times faster than Dumpy).

9.1.3 Parallel construction with DumpyOS

In Fig. 11, we show the building time breakdown of DumpyOS. In the first two rows, we display the I/O and the CPU time of Dumpy, respectively. For the rest rows, we show the timelines of building DumpyOS with different numbers of threads. We sum the I/O or CPU time of each stage and show them in the figure. Note that in Stages 1 and 4, the tasks are executed in batches, and in each batch, CPU and I/O threads need to synchronize on the inner barriers (see Fig. 5b). To make the figure clear, we omit the detailed timelines between inner barriers, and sum the total time consumed in these stages. In the synchronization process of these two stages, CPU threads always need to wait for I/O threads. We mark the (sum) waiting time with the grey arrows. For Stage 1, CPU time can be totally masked by I/O time, while in Stage 4, a small portion of CPU time cannot be overlapped by I/O time; we illustrate this with a small non-overlapping part at the ends of the bars in the figure.

Using parallel computation, the CPU time costs for Stages 1 and 3 reduce almost linearly with the number of threads and can be fully masked by the simultaneous I/O-bound work. Note that DumpyOS introduces one more pass of sequential read of the dataset in Stage 2, but this time can be overlapped by the CPU time. Thus, this design improves the overall wall-clock time by reducing the writing time in Stage 4. In Stage 4, our buffering technique significantly reduces the write time to **40%** (see the comparison between Dumpy and 2-threads DumpyOS, where only one thread is used to issue I/O requests). By parallelizing writing requests, the write time is further reduced by up to **2**x when using 5 threads. Notice that with 5 threads, the CPU time is totally masked by the I/O time except for Stage 4, and the SSD is nearly saturated (where the write time is close to the sequential read time in Stage 4). Only marginal benefits can be gained as the number of threads increases. Finally, we note that more than **95%** of the additionally introduced CPU time in Stage 4 is masked by the I/O time, verifying the effectiveness of our thread-scheduling algorithm.

9.2 Query processing

In this section, we verify the accuracy of the query processing.

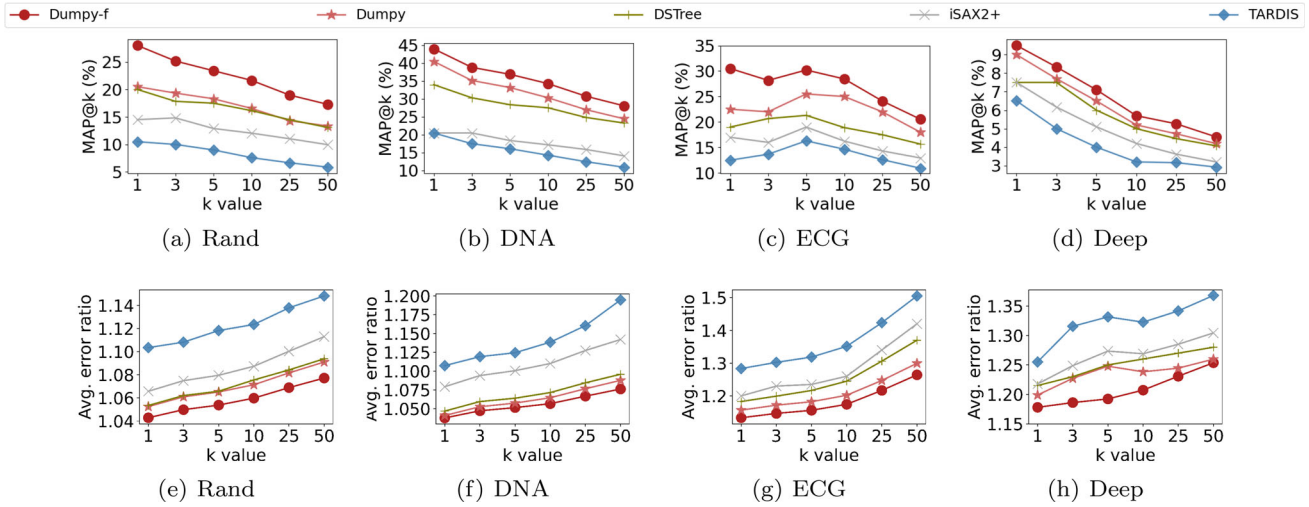


Fig. 12 Approximate search under ED (search one node)

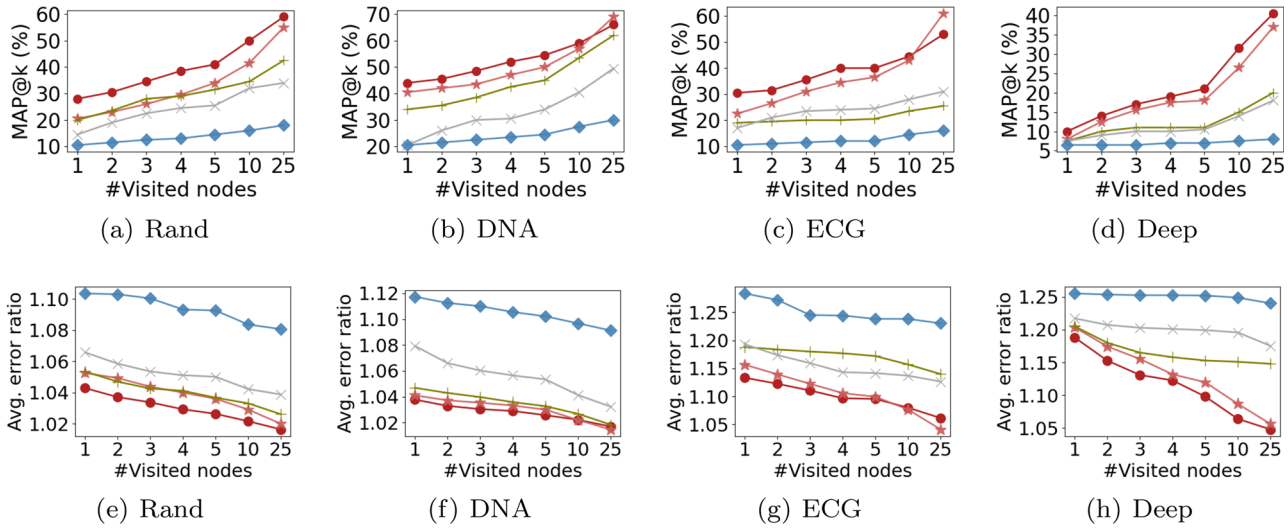


Fig. 13 Extended approximate search under ED (k = 1)

9.2.1 Approximate search

First, we evaluate the accuracy of approximate results across four datasets.

[Search one node] We compare these approaches when searching one node to obtain the approximate top-*k* result on three datasets with ED distance, and results are shown in Fig. 12. It can be seen that Dumpy consistently outperforms other approaches. Specifically, Dumpy improves the average accuracy by **84**%, **46**%, **11**% and reduces the average error ratio by **7.3**%, **3.4**% and **1.4**% on the four datasets compared with TARDIS, iSAX2+ and DSTree respectively. Tardis has the lowest performance, due to its low fill factor. iSAX2+ suffers from insufficient intra-node series proximity, characterized by the number of uneven nodes (>**20**% leaf nodes have one segment using more than 4 bits than other segments). This number is only **1.4**% for Dumpy. As for TARDIS, the small-sized leaves cannot provide enough candidate series, though the series in the target leaf have a superior quality (small distance to the query). Moreover, Dumpy-Fuzzy has higher accuracy than Dumpy and other approaches, which verifies our duplication strategy.

[Search multiple nodes] In Fig. 13, we compare the accuracy of searching multiple nodes (1 to 25) for top-1 result with ED distance. The MAP value of Dumpy and Dumpy-*f* increases remarkably faster than the competitors, attributed to our multi-ary structure that provides closer sibling nodes. When visiting 25 nodes, Dumpy and Dumpy-*f* improve the accuracy by **58**%, **65**% and reduce the average error ratio by **3.6**% and **3.7**% on average of four datasets respectively compared with the second-best approach, DSTree. We also compare the accuracy as the series length varies (Fig. 14). The accuracy on different lengths shares similar rankings.

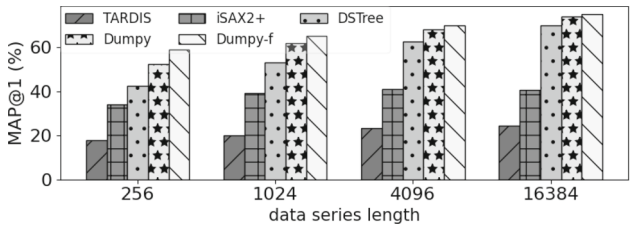
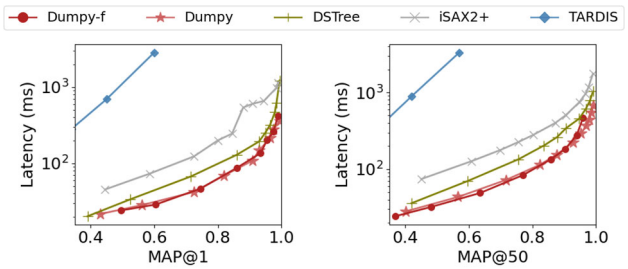


Fig. 14 Approx. search versus series lengths (search 25 nodes)



(a) $k = 1$ (b) $k = 50$

Fig. 15 Efficiency versus accuracy

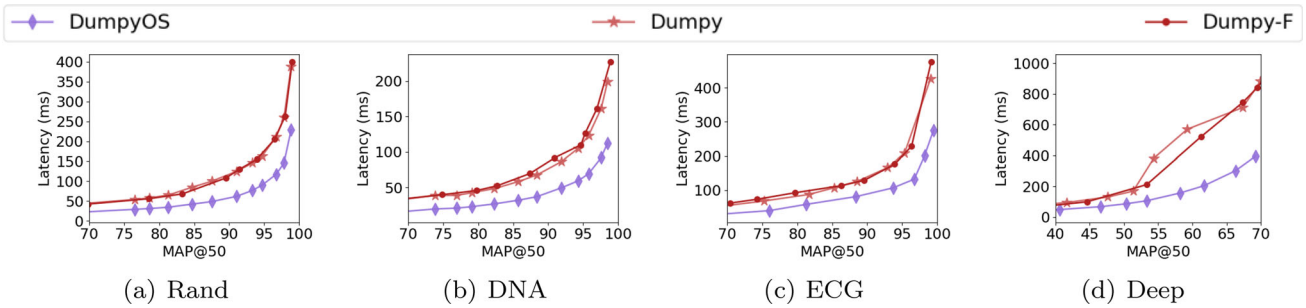
[Efficiency versus accuracy] We extend the approximate search to all leaf nodes with lower bound pruning to evaluate the indexes' response time under the whole MAP range. The results are depicted in Fig. 15. Benefiting from the high-proximity nodes and compact index structure, Dumpy surpasses its competitors in both low and high-precision intervals. Our results show that Dumpy can achieve 60–70% MAP within 100ms on TB-level datasets, while its index building time is 4x faster than the SOTA competitors. These results demonstrate that Dumpy fulfills the requirements of many kNN-based applications.

[Parallel search of DumpyOS] In Figs. 16 and 17, we show the *ng*-approximate query performance improvements of DumpyOS against Dumpy and Dumpy-*f*. The queries are processed sequentially. DumpyOS accelerates the pruning-based query process with the parallel search algorithm in Sect. 7.3. DumpyOS reduces the query latency of Dumpy by **41**%, **44**%, **36**%, **55**% on Rand, DNA, ECG and Deep datasets, respectively when $k = 50$ on the high-recall range. When $k = 1$, DumpyOS reduces the query latency of Dumpy by **43**%, **43**%, **35**%, and **46**% on Rand, DNA, ECG, and Deep datasets, respectively on the high-recall range. The speedup of querying becomes larger as MAP goes higher since a higher precision requires more nodes to be loaded and more distance calculations, which can be accelerated by the buffering technique and the parallel computation of DumpyOS.

[Searching under DTW] In this experiment, we compare the accuracy under DTW distance in Fig. 18. Due to the inherent hardness, the precision is lower than ED generally. However, Dumpy and Dumpy-*f* still achieve better precision and error ratio under DTW. Since the absolute distance of DTW is smaller than ED, the differences of the error ratio among all the methods tend to be smaller (except for TARDIS).

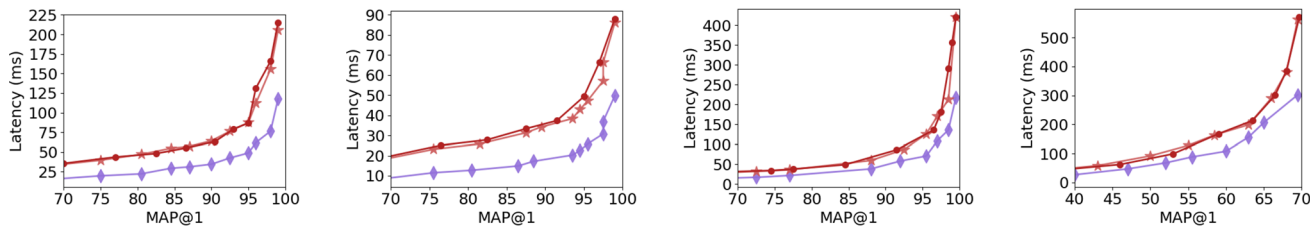
9.2.2 Approximate search on DumpyOS-F

We evaluate our novel, extended approximate search algorithm, DumpyOS-F, on Rand and DNA datasets with Dumpy



(a) Rand (b) DNA (c) ECG (d) Deep

Fig. 16 Parallel pruning-based *ng*-approximate search with DumpyOS ($k = 50$)



(a) Rand (b) DNA (c) ECG (d) Deep

Fig. 17 Parallel pruning-based *ng*-approximate search with DumpyOS ($k = 1$)

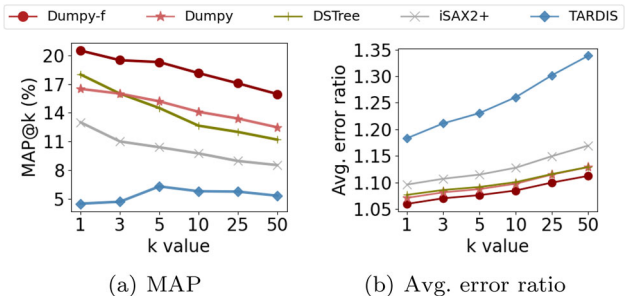


Fig. 18 Approx. search under DTW (search one node)

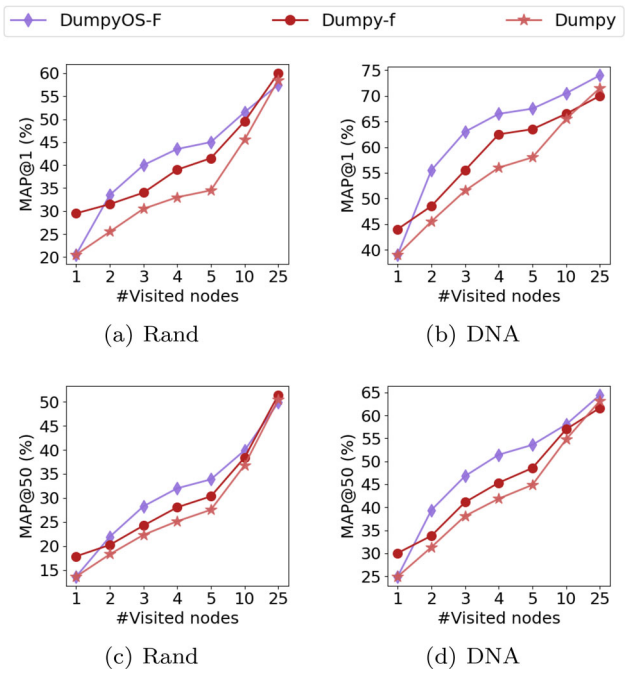


Fig. 19 Extended approximate search with DumpyOS-F

and Dumpy-Fuzzy, as shown in Fig. 19. When visiting a single leaf, DumpyOS-F shares the same accuracy with Dumpy, because of the invariance of the index structure. Dumpy-Fuzzy is the most accurate benefiting from the duplicated series stored in the node. When we visit two or more nodes, DumpyOS achieves the highest accuracy, that is, an average (over our four settings) of **18**% and **8.7**%, respectively, higher MAP than Dumpy and Dumpy-Fuzzy.

9.2.3 Exact search

We evaluate the exact search efficiency of Dumpy against other methods in Table 2. Since TARDIS does not support exact kNN search in the original paper, we implement a similar algorithm as the iSAX-index family, where the nodes summarized with iSAX words can be pruned during searching. The results are reported on average of 40 queries with $k = 50$. Overall, Dumpy achieves the best efficiency in all

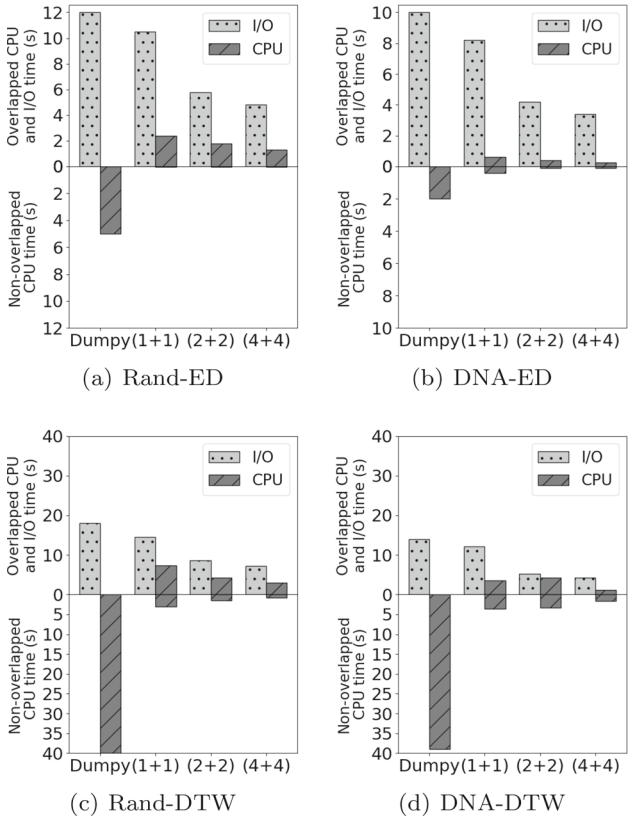


Fig. 20 Parallel exact search time with DumpyOS. The first column is for single-thread Dumpy and the rest three columns are for DumpyOS. On x-axis, $A + B$ indicates A threads for reading, and B threads for computing

cases. It is worth noting that although DSTree has a higher pruning ratio than Dumpy, the response time is still slower than Dumpy. The reason is as follows. DSTree takes a longer time to compute the lower bound of distance due to computing the standard deviation frequently. iSAX2+ suffers from the low fill factor and needs to read about **3** times nodes more than Dumpy and DSTree.

[Multi-thread exact search with DumpyOS] In Fig. 20, we show the exact query performance of DumpyOS when varying the number of threads. To gain more insights, we show the I/O and CPU time of the query time components of the total query time, as well. I/O time is shown in the upper part. CPU time is broken into two parts. One part overlaps with the I/O time, shown in the upper part. The other, non-overlapped part, is shown in the lower part.

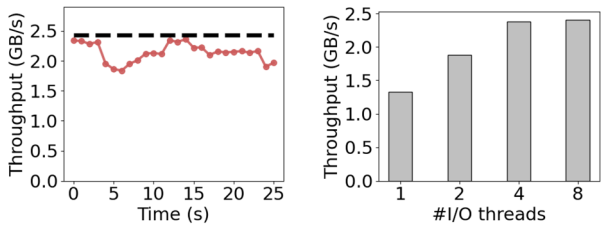
As a single-thread algorithm, Dumpy's I/O and CPU time are not overlapped at all. In contrast, DumpyOS allows I/O threads and CPU threads to work simultaneously, therefore, their execution times are overlapping. The overall query time is computed by the sum of I/O time (in the upper part) and non-overlapping CPU time (in the bottom part).

Compared with Dumpy, DumpyOS achieves **5.8x** faster query time on average with "4+4" threads, where I/O time is

Table 2 Exact search statistics

	Method	Resp. time (s)	#Loaded nodes	Prune ratio (%)
Rand-ED	iSAX2+	65 (50+15)	7595	81.51
	DSTree	33 (20+13)	2027	86.06
	TARDIS	53 (15+38)	1665744	59.30
	Dumpy	**17** (**12+5**)	2641	83.70
Rand-DTW	iSAX2+	151 (85+66)	18660	72.58
	DSTree	79 (24+65)	3678	75.12
	TARDIS	208 (22+186)	2846868	49.24
	Dumpy	**58** (**18+40**)	3997	73.61
DNA-ED	iSAX2+	42 (26+16)	1077	91.04
	DSTree	21 (16+5)	326	94.00
	TARDIS	40 (11+29)	161959	71.64
	Dumpy	**12** (**10+2**)	433	91.69
DNA-DTW	iSAX2+	116 (60+56)	2163	87.77
	DSTree	63 (18+45)	497	90.93
	TARDIS	143 (16+127)	194645	68.78
	Dumpy	**53** (**14+39**)	528	89.41

The best method is marked in bold
The response time is also broken down into I/O time (the first number in the parentheses) and CPU time (the second number)



(a) I/O throughput monitoring during search. (b) The increase of I/O throughput with parallelization.

Fig. 21 I/O throughput on parallel exact search with DumpyOS

2.8x faster while CPU time is **7.2x** faster. The acceleration is obvious from one to two I/O threads (see DumpyOS "1+1" to "2+2"), which comes from our buffering and parallel reading algorithms. As we keep increasing the number of threads, the SSD's throughput turns saturated, and the performance reaches a peak. The SIMD technique significantly reduces computing time, especially under DTW distance (see Dumpy to DumpyOS "1+1"). Observe that the CPU time is almost totally masked by the I/O time (see DumpyOS "4+4"), as expected.

[Hardware bottleneck] Since in the pruning-based search process of DumpyOS, the I/O time becomes the major part of the total wall-clock time, we further study the bottleneck of the hardware that limits the I/O performance. Given that nearly all the I/O requests are sequential I/Os, we measure the I/O performance with the (read) throughput. The results are shown in Fig. 21. In Fig. 21a, we monitor the I/O throughput during a period of the pruning-based parallel exact search with 4 threads on the Rand100GB dataset. It can be seen that the I/O throughput is very close to the I/O bandwidth, which is measured using the fio [6] command under the condition of multi-threads sequential read. This indicates that the bandwidth of the NVMe SSD is nearly saturated, especially when the query needs to load a large number of nodes.

In Fig. 21b, we compare the I/O throughput with a different number of I/O threads. Before 4 threads, the read throughput increases as more threads are used to submit I/O requests. This is because more threads produce a number of I/O requests that lead to a better use of the parallelization ability of the NVMe SSD. However, with more than 4 I/O threads, the SSD bandwidth becomes saturated, which limits further improvement in query performance. This indicates that under a better hardware environment, including a more advanced PCIe channel (e.g., version of $\geq$ 4.0), a more efficient I/O system (e.g., SPDK [20]) or a better SSD, DumpyOS can further improve its I/O time, and thus the overall query performance.

[Pruning-based search versus scan-based search] We further compare DumpyOS with SOTA parallel-scan-based exact search algorithm PARIS+, and report the results in Fig. 22. During the search process, PARIS+ needs to materialize the leaves that have been visited, by flushing these data when the memory buffer is full. In our experiments, PARIS+ shows slight superiority when the number of served queries is small. In this case, the buffer is not full, hence no disk writing. Yet, as more queries come, DumpyOS becomes more efficient since the random disk writes hurt the perfor-

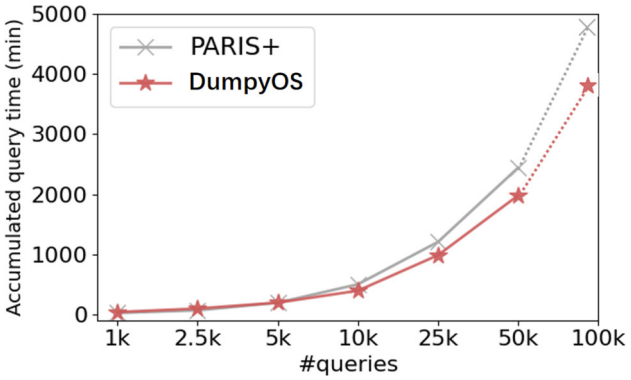


Fig. 22 DumpyOS versus PARIS+

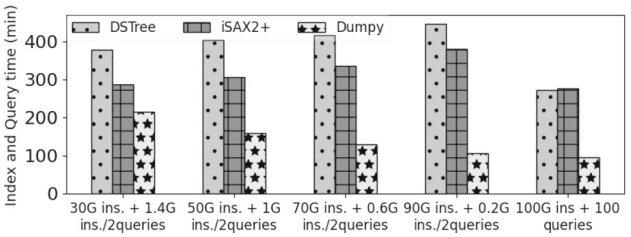


Fig. 23 Update performance (4 GB memory)

mance of PARIS+. Even after serving 50,000 queries, the ratio of full leaves in PARIS+ is only **16%**, at which time PARIS+ has generated over 300,000 leaves. On the contrary, DumpyOS that leverages parallel techniques to achieve faster index building, leaves no indexing burden on the query stage and hence achieves better query performance with intensive query workloads.

9.3 Complete workloads

Finally, we compare different approaches when inserting new data series (Fig. 23). We omit TARDIS since it is designed for the static dataset and not easy to be extended. To be fair, we implement all methods using a single thread (even though Dumpy is multi-threaded). We use different synthetic workloads consisting of 100 exact queries, and a total of 100 million series, where queries are interleaved by a batch of insertions. The results show Dumpy outperforms the competitors for all workloads, thanks to its compact structure. Although the re-splitting and re-packing procedures add an additional cost, this cost is balanced by the efficiency improvements that these two designs bring along. Moreover, Dumpy shows better performance when the initial batch size increases (while iSAX and DSTree show worse performance), because fewer insertions incur fewer re-splitting and re-packing actions.

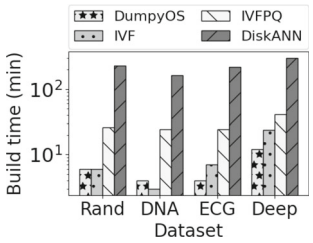


Fig. 24 Index construction time on four 25 GB datasets

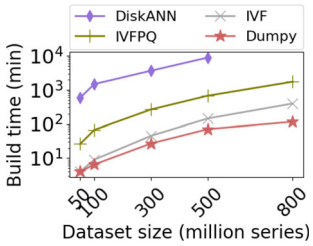


Fig. 25 Index construction on Rand datasets

9.4 Comparison with high-d vector indexes

In this subsection, we compare our DumpyOS solutions with SOTA high-dimensional vector indexes w.r.t. the building efficiency and *ng*-approximate query performance. Note that SOTA high-dimensional vector indexes do not support neither exact search nor approximate search with quality guarantees. Therefore, these approaches target a different set of applications than DumpyOS. Nevertheless, we compare to them for completeness.

9.4.1 Experimental settings

[Algorithms and implementations.] We use **Disk- ANN** [69] as a representative disk-based graph-based index and adopt the most recent implementation.[6] We use **IVF** [23] and **IVFPQ** [35] from the family of partition-based indexes. We train the centroids of both, as well as the codebooks of IVFPQ with the Faiss library.[7] Since there are no disk-based implementations of IVF and IVFPQ, we provide our own implementations; we store sequentially the data, as well as the quantized data of each cluster. In this way, data in each cluster can be read with sequential I/Os when querying, similarly to DumpyOS.

[Parameters.] For DiskANN, we set $R = 32$ and $L = 100$ for constructing the graph. For IVF and IVFPQ, we randomly select 1 million data series from the datasets to train the centroids, and the number of centroids is set to be 1/10,000 of the number of the series. For IVFPQ, the number of segments is set to 16 like DumpyOS. The memory constraints when

[6] https://github.com/microsoft/DiskANN.

[7] https://github.com/facebookresearch/faiss.git.

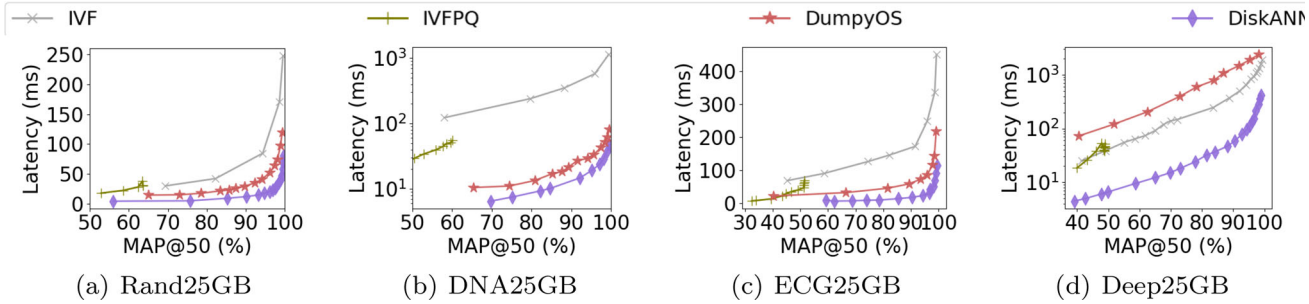


Fig. 26 Query performance comparisons with high-dimensional vector indexes ($k = 50$)

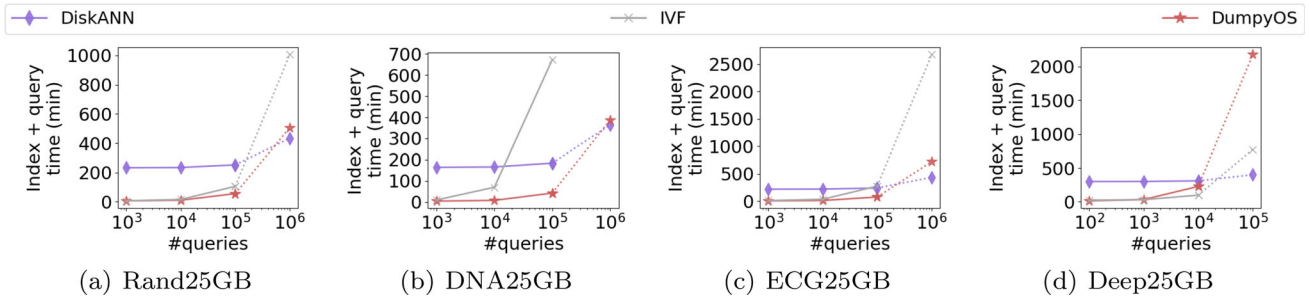


Fig. 27 Accumulated time of indexing and *ng*-approximate queries under 90% recall@50

building indexes and running queries are controlled to be the same as Dumpy. When building the index, all 10 threads are utilized.

[Datasets.] Since building DiskANN on 100 GB datasets costs over one day which significantly exceeds the time limit of common data series similarity search applications, we extract subsets of the datasets of 25 GB for the experiments in this subsection.

9.4.2 Experimental results

[Index building.] In Fig. 24, we show the index-building time of the four indexes. DumpyOS and IVF are the most efficient indexes, followed by IVFPQ, and finally, DiskANN. Compared with IVFPQ and DiskANN, DumpyOS reduces the building time by 79% and 97%, respectively. Note that the index-building complexity of IVF and IVFPQ is highly sensitive to the number of centroids. In contrast, DumpyOS builds the index based on the iSAX summarization without any real distance calculations and thus achieves superior efficiency and robustness. In Fig. 25, we observe that DumpyOS has the best scalability, followed by IVF, IVFPQ, and DiskANN. Note that on the Rand500GB dataset, DiskANN requires over 6 days to build the index, which is impractical for many scenarios.

[Query performance.] In Fig. 26, we compare the *ng*-approximate query performance of Dumpy with the other three indexes. We observe that DiskANN exhibited the best performance, followed by DumpyOS, IVF and IVFPQ. Note that only the quantized codes are stored in the IVFPQ index, and this information loss of the quantization prevents IVFPQ from achieving a high recall. DiskANN is slightly better than DumpyOS on data series datasets, while IVF and IVFPQ are much slower than DumpyOS. In the classical long data series dataset like DNA, IVF suffers from the curse of dimensionality [31] and thus a poor clustering quality, which results in performance degradation. On the other hand, DumpyOS can capture the differences on different segments between different data series and thus provide a superior query performance. Note that a prominent characteristic of these data series is that they contain temporal semantics, i.e., the value continuity among the adjacent time axis. Therefore, data series indexes that use summarization techniques such as PAA and SAX work well on these datasets. On the contrary, on the Deep dataset, whose series exhibit high frequencies, DumpyOS is less effective in producing the series summarizations, and performs worse than IVF.

In Fig. 27, we show the accumulated time of indexing and *ng*-approximate query answering for 90% recall@50. Note that IVFPQ is skipped in this experiment since it cannot reach 90% recall@50. We extract 100,000 queries from the base 100 GB datasets for this experiment, and we extrapolate the time on 1 million queries based on the average time of the 100,000 queries (i.e. the dotted lines). As we observe, on Rand25GB, DNA25GB, and ECG25GB, only after running nearly 1 million queries, DiskANN is faster than DumpyOS. For Deep25GB, DiskANN still needs 100,000 queries to compensate for the long building time.

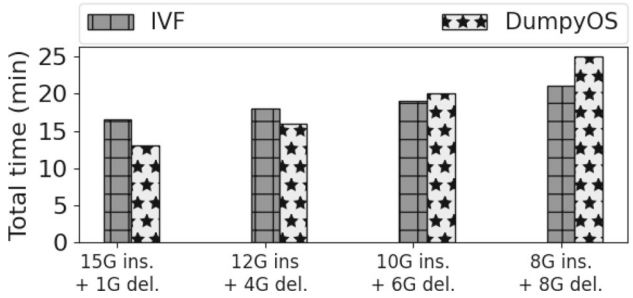


Fig. 28 Update performance: the overall time of building a static index on 10 GB dataset, updating (inserting and deleting) 16 GB data with batches of size 1 GB, and executing 10,000 *ng*-approximate queries with 90% recall@50

[Streaming scenario.] We finally test the performance of the indexes on streaming scenarios under workloads of different insertion/deletion/update ratios. We skip DiskANN since the disk-version updating algorithm (i.e. FreshDiskANN [68]) is not publicly available, and IVFPQ since it cannot reach a recall of 90%. For the workloads, we fix the initial dataset to be 10 GB, the number of *ng*-approximate queries to be 10,000 under 90% recall@50, and the size of the updated data to be 16 GB, including insertions and deletions. For IVF, we do not update the centroids during updating by assuming the data distribution remains the same. In Fig. 28, we show the total time of initial index building, updating and querying. As observed, DumpyOS's overall performance is as good as the simple IVF, though DumpyOS adaptively adjusts its index structure to achieve the best point of the proximity-compactness trade-off. With a higher deletion ratio, DumpyOS needs to make more adjustments while IVF simply marks the deleted entries. Nevertheless, the performance difference is marginal.

9.4.3 Discussion

It is important to note that data series and ANN indexes are designed for different application scenarios, though the problems are conceptually the same. ANN indexes are mainly designed for the approximate similarity search of valuable deep learning embedding vectors. They are supposed to answer online queries in the common scenario, e.g., recommendation systems. In this case, the fast approximate query performance is useful.

On the other hand, data series indexes are designed to manage massive data series datasets of low value-density originating from monitoring devices and natural sequences like IoT devices, etc. They usually serve for offline data exploration and mining tasks [90], such as classification and pattern recognition as we introduce in Sect. 1. For example, for the large and fast IoT series data produced by sensors, a series index with *high ingestion throughput* like DumpyOS is necessary [36, 64, 79]. The same is true for data exploration tasks, where the analyst may need to build an index several times from scratch, on different data subsets. Therefore, having solutions that achieve excellent performance in index building time and exact query answering time is very important. Note that these abilities are *not* supported by ANN indexes like DiskANN.

In a nutshell, data series and ANN indexes are the most suitable techniques of choice for their native applications (though, some technical designs can be borrowed or inspired from each other). A more detailed discussion can be found in our previous work [80].

10 Conclusions and future work

We propose a novel multi-ary data series index Dumpy with an adaptive split strategy that hits the right balance in the proximity-compactness trade-off. By mitigating the boundary issue, Dumpy-Fuzzy and DumpyOS-F achieve even higher accuracy by checking series in the fuzzy boundary. Moreover, the DumpyOS parallel solution fully leverages modern multi-core CPUs and NVMe SSDs, resulting in higher efficiency on both computations and I/Os. Experiments with a variety of large, synthetic and real datasets demonstrate the efficiency, scalability, and accuracy of our solutions.

Acknowledgements This work is supported by the Ministry of Science and Technology of China, National Key Research and Development Program (No. 2021YFB3300503). We especially thank Prof. Botao Peng in Institute of Computing Technology, Chinese Academy of Sciences, for his assistance in reproducing PARIS+.

References

1. Agrawal, N., Prabhakaran, V.E.: Design tradeoffs for ssd performance. In: USENIX ATC (2008)
2. Agrawal, R., Faloutsos, C., Swami, A.: Efficient similarity search in sequence databases. In: FODO (1993)
3. Anagnostou, P., Barbas, P.E.: Approximate KNN classification for biomedical data. In: Big Data (2020)
4. Arora, A.: Hd-index: pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. PVLDB **11**(8), 906–919 (2018)
5. Axboe, J.: Efficient io with io_uring. https://kernel.dk/io_uring.pdf. Accessed April 7, 2023
6. Axboe, J.: Flexible I/O Tester (2022). https://github.com/axboe/fio
7. Azizi, I., Echihabi, K., Palpanas, T.: ELPIS: graph-based similarity search for scalable data science. PVLDB **16**(6), 1548–1559 (2023)
8. Babenko, A., Lempitsky, V.: Efficient indexing of billion-scale datasets of deep descriptors. In: CVPR (2016)
9. Bagnall, A.J., Cole, R.L., Palpanas, T., Zoumpatianos, K.: Data series management (Dagstuhl seminar 19282). Dagstuhl Rep. **9**(7), 24–39 (2019)
10. Beis, J., Lowe, D.: Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In: CVPR, pp. 1000–1006 (1997)

11. Boniol, P., Linardi, M.: Automated anomaly detection in large sequences. In: ICDE (2020)
12. Boniol, P., Palpanas, T.: Series2graph: graph-based subsequence anomaly detection for time series. PVLDB **13**(12), 1821–1834 (2020)
13. Camerra, A., Palpanas, T.E.: isax 2.0: indexing and mining one billion time series. In: ICDM (2010)
14. Camerra, A., Shieh, J., Palpanas, T.E.: Beyond one billion time series: indexing and mining very large time series collections with *i*sax2+. KAIS (2014). https://doi.org/10.1007/s10115-012-0606-6
15. Chatzakis, M., Fatourou, P., Kosmas, E., Palpanas, T., Peng, B.: Odyssey: a journey in the land of distributed data series similarity search. PVLDB (2023)
16. Chen, F., Hou, B., Lee, R.: Internal parallelism of flash memory-based solid-state drives. TOS **12**(3), 1–39 (2016)
17. Chen, G., Lee, C.e.: Nearest neighbors for modern applications with massive data. In: NeurIPS (2017)
18. Chen, G.H., Shah, D.: Explaining the success of nearest neighbor methods in prediction. Found. Trends Mach. Learn. **10**(5–6), 337–588 (2018)
19. Chen, Q., Zhao, B., Wang, H., Li, M., Liu, C.: Spann: highly-efficient billion-scale approximate nearest neighborhood search. NeurIPS **34**, 5199–5212 (2021)
20. Community, S.: Storage performance development kit. https://spdk.io/. Accessed March 4, 2024
21. Didona, D., Ioannou, N., Stoica, R., Kourtis, K.: Toward a better understanding and evaluation of tree structures on flash ssds. PVLDB **14**(3), 364–377 (2020)
22. Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., Keogh, E.: Querying and mining of time series data: experimental comparison of representations and distance measures. PVLDB **1**(2), 1542–1552 (2008)
23. Douze, M., et al.: The faiss library (2024)
24. Echihabi, K., et al.: Big sequence management: scaling up and out. In: EDBT (2021)
25. Echihabi, K., et al.: ProS: data series progressive k-NN similarity search and classification with probabilistic quality guarantees. VLDB J **32**, 763–789 (2023)
26. Echihabi, K., Fatourou, P., Zoumpatianos, K., Palpanas, T., Benbrahim, H.: Hercules against data series similarity search. PVLDB **15**(10), 2005–2018 (2022)
27. Echihabi, K., Zoumpatianos, K., Palpanas, T., Benbrahim, H.: The lernaean hydra of data series similarity search: an experimental evaluation of the state of the art. PVLDB **12**(2), 112–127 (2018)
28. Echihabi, K., Zoumpatianos, K., Palpanas, T., Benbrahim, H.: Return of the lernaean hydra: experimental evaluation of data series approximate similarity search. PVLDB **13**(3), 403–420 (2019)
29. Ehrenberg, D.: The asynchronous input/output (aio) interface. https://github.com/littledan/linux-aio. Accessed April 7, 2023
30. Fevgas, A., Akritidis, L., Bozanis, P., Manolopoulos, Y.: Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. VLDB J. **29**, 273–311 (2020)
31. Francois, D., Wertz, V.: The concentration of fractional distances. TKDE **19**(7), 873–886 (2007)
32. Fu, C., Xiang, C., Wang, C., Cai, D.: Fast approximate nearest neighbor search with the navigating spreading-out graph. PVLDB **12**(5), 461–474 (2019)
33. Gao, C., Shi, L., Ji, C., Di, Y., Wu, K.: Exploiting parallelism for access conflict minimization in flash-based solid state drives. TCAD **37**(1), 168–181 (2018)
34. Ge, T., He, K., Ke, Q., Sun, J.: Optimized product quantization. TPAMI **36**(4), 744–755 (2013)
35. Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. TPAMI **33**(1), 117–28 (2010)
36. Jensen, S.K., Pedersen, T.B., Thomsen, C.: Time series management systems: a survey. TKDE **29**(11), 2581–2600 (2017)
37. Jin, P., Xie, X., Wang, N., Yue, L.: Optimizing r-tree for flash memory. Expert Syst. Appl. **42**(10), 4676–4686 (2015)
38. Jo, J., Seo, J., Fekete, J.: PANENE: a progressive algorithm for indexing and querying approximate k-nearest neighbors. TVCG **26**(2), 1347–1360 (2020)
39. Johnson, A.E., Pollard, T.J., Shen, L., Lehman, L.: Mimic-iii, a freely accessible critical care database. Sci. Data **3**(1), 1–9 (2016)
40. Keogh, E.: A decade of progress in indexing and mining large time series databases. In: PVLDB (2006)
41. Keogh, E., Chakrabarti, K., Pazzani, M., Mehrotra, S.: Dimensionality reduction for fast similarity search in large time series databases. KAIS **3**(3), 263–286 (2001)
42. Kim, J., Seo, S., Jung, D., Kim, J.S., Huh, J.: Parameter-aware i/o management for solid state disks (ssds). IEEE Trans. Comput. **61**(5), 636–649 (2012)
43. Kondylakis, H., Dayan, N., Zoumpatianos, K., Palpanas, T.: Coconut: a scalable bottom-up approach for building data series indexes. PVLDB **11**(6), 677–690 (2018)
44. Kondylakis, H., et al.: Coconut: sortable summarizations for scalable indexes over static and streaming data series. VLDB J. **28**(6), 847–869 (2019)
45. Korn, F., Pagel, B., Faloutsos, C.: On the 'dimensionality curse'nd the 'self-similarity blessing'. TKDE **13**(1), 96–111 (2001)
46. Levchenko, O., Kolev, B., Yagoubi, D.E., et al.: Bestneighbor: efficient evaluation of knn queries on large time series databases. KAIS **63**(2), 349–378 (2021)
47. Levchenko, O., Yagoubi, D.E., Akbarinia, R.: Spark-parsketch: a massively distributed indexing of time series datasets. In: CIKM (2018)
48. Li, W., Zhang, Y., Sun, Y., Wang, W., Li, M., Zhang, W., Lin, X.: Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement. TKDE **32**(8), 1475–1488 (2019)
49. Linardi, M., Palpanas, T.: Scalable, variable-length similarity search in data series: the ulisse approach. PVLDB **11**(13), 2236–2248 (2018)
50. Linardi, M., Palpanas, T.: Scalable data series subsequence matching with ULISSE. VLDB J. **29**(6), 1449–1474 (2020)
51. Malkov, Y.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. TPAMI **42**(4), 824–836 (2018)
52. NCBI: National library of medicine. https://www.ncbi.nlm.nih.gov/. Accessed March 14, 2022
53. Palpanas, T.: Data series management: the road to big sequence analytics. SIGMOD Rec. **44**(2), 47–52 (2015)
54. Palpanas, T.: Big sequence management: a glimpse of the past, the present, and the future. SOFSEM **9587**, 63–80 (2016)
55. Palpanas, T.: Evolution of a data series index. In: ISIP, pp. 68–83 (2020)
56. Palpanas, T., Beckmann, V.: Itisa. SIGMOD Rec. **48**(3), 36–40 (2019)
57. Paparrizos, J., Edian, I., Liu, C., Elmore, A.J., Franklin, M.J.: Fast adaptive similarity search through variance-aware quantization. In: ICDE (2022)
58. Peng, B., Fatourou, P., Palpanas, T.: Paris: the next destination for fast data series indexing and query answering. In: Big Data, pp. 791–800 (2018)
59. Peng, B., Fatourou, P., Palpanas, T.: Messi: in-memory data series indexing. In: ICDE, pp. 337–348 (2020)
60. Peng, B., Fatourou, P., Palpanas, T.: Paris+: data series indexing on multi-core architectures. TKDE **33**(5), 2151–2164 (2020)
61. Peng, B., Fatourou, P., Palpanas, T.: Fast data series indexing for in-memory data. VLDB J. **30**(6), 1041–1067 (2021)

62. Peng, B., Fatourou, P., Palpanas, T.: Sing: sequence indexing using gpus. In: ICDE, pp. 1883–1888 (2021)
63. Rakthanmanon, T., Campana, B., Mueen, A.: Searching and mining trillions of time series subsequences under dynamic time warping. In: SIGKDD (2012)
64. Raza, U., Camerra, A.: Practical data prediction for real-world wireless sensor networks. TKDE **27**(8), 2231–2244 (2015)
65. Schubert, E., Zimek, A., Kriegel, H.P.: Fast and scalable outlier detection with approximate nearest neighbor ensembles. In: DASFAA, pp. 19–36 (2015)
66. Shannon, C.E.: A mathematical theory of communication. BSTJ **27**(3), 379–423 (1948)
67. Shieh, J., Keogh, E.: Isax: indexing and mining terabyte sized time series. In: SIGKDD, pp. 623–631 (2008)
68. Singh, A., et al.: Freshdiskann: a fast and accurate graph-based ANN index for streaming similarity search (2021)
69. Subramanya, S.J., Kadekodi, R.: Diskann: fast accurate billion-point nearest neighbor search on a single node. In: NeurIPS (2019)
70. Tan, C.W., Webb, G.I., Petitjean, F.: Indexing and classifying gigabytes of time series under time warping. In: SDM, pp. 282–290 (2017)
71. Tavakkol, A., Gómez-Luna, J., et.al., M.S.: MQSim: a framework for enabling realistic studies of modern Multi-Queue SSD devices. In: FAST, pp. 49–66 (2018)
72. Turpin, A., Scholer, F.: User performance versus precision measures for simple search tasks. In: SIGIR (2006)
73. Vision, S.C.: Deep billion-scale indexing. http://sites.skoltech.ru/compvision/noimi. Accessed March 14, 2022
74. Wang, L., Zhang, Z., He, B.: Pa-tree: Polled-mode asynchronous b+ tree for nvme. In: ICDE (2020)
75. Wang, M., Xu, X., Yue, Q.: A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. PVLDB **14**(11), 1964–1978 (2021)
76. Wang, Q., et al.: iEDeaL: a deep learning framework for detecting highly imbalanced interictal epileptiform discharges. PVLDB **16**(3), 480–490 (2022)
77. Wang, Q., Palpanas, T.: Deep learning embeddings for data series similarity search. In: SIGKDD (2021)
78. Wang, Y., Wang, P., Pei, J.: A data-adaptive and dynamic segmentation index for whole matching on time series. PVLDB **6**(10), 793–804 (2013)
79. Wang, Z., He, Z., Wang, P., Wang, Y., Wang, W.: Static and streaming discovery of maximal linear representation between time series. TKDE **36**(1), 401–415 (2024)
80. Wang, Z., Wang, P., Palpanas, T., Wang, W.: Graph-and tree-based indexes for high-dimensional vector similarity search: analyses, comparisons, and future directions. IEEE Data Eng. Bull. **46**(3), 3–21 (2023)
81. Wang, Z., Wang, Q., Wang, P., Palpanas, T., Wang, W.: Dumpy: a compact and adaptive index for large data series collections. Proc. ACM Manag. Data **1**(1), 1–27 (2023)
82. Wei, J., Peng, B., Lee, X., Palpanas, T.: Det-lsh: a locality-sensitive hashing scheme with dynamic encoding tree for approximate nearest neighbor search. PVLDB **17**(9), 2241–2254 (2024)
83. Write amplification (2023). https://en.wikipedia.org/w/index.php?title=Write_amplification&oldid=1190580363. Accessed March 15, 2024
84. Yagoubi, D.E., Akbarinia, R., Masseglia, F., Palpanas, T.: Dpisax: massively distributed partitioned isax. In: ICDM, pp. 1135–1140 (2017)
85. Yagoubi, D.E., Akbarinia, R., Masseglia, F., Palpanas, T.: Massively distributed time series indexing and querying. TKDE **32**(1), 108–120 (2020)
86. Yu, G.X., Markakis, M., Kipf, A., Larson, P., Minhas, U.F., Kraska, T.: Treeline: an update-in-place key-value store for modern storage. PVLDB **16**(1), 99–112 (2022)
87. Zhang, L., Alghamdi, N., Eltabakh, M.Y., Rundensteiner, E.A.: Tardis: distributed indexing framework for big time series data. In: ICDE, pp. 1202–1213 (2019)
88. Zhao, K., Song, L., Zhang, Y., Pan, P., Xu, Y., Jin, R.: Ann softmax: acceleration of extreme classification training. PVLDB **15**(1), 1–10 (2021)
89. Zheng, B., Gao, Y.: Declog: decentralized logging in non-volatile memory for time series database systems. Proc. VLDB Endow. **17**(1), 1–14 (2023)
90. Zoumpatianos, K., Idreos, S., Palpanas, T.: Indexing for interactive exploration of big data series. In: SIGMOD, pp. 1555–1566 (2014)
91. Zoumpatianos, K., Idreos, S., Palpanas, T.: Ads: the adaptive data series index. VLDB J. **25**(6), 843–866 (2016)
92. Zoumpatianos, K., Lou, Y., Ileana, I., Palpanas, T., Gehrke, J.: Generating data series query workloads. VLDB J. **27**(6), 823–846 (2018)
93. Zoumpatianos, K., Lou, Y.: Query workloads for data series indexes. In: SIGKDD (2015)
94. Zoumpatianos, K., Palpanas, T.: Data series management: fulfilling the need for big sequence analytics. In: ICDE, pp. 1677–1678 (2018)