Dumpy: A Compact and Adaptive Index for Large Data Series Collections

ZEYU WANG, School of Computer Science, Fudan University, China
QITONG WANG, LIPADE, Université Paris Cité, France
PENG WANG*, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China
THEMIS PALPANAS, LIPADE, Université Paris Cité & IUF, France
WEI WANG, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

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 pre-processing (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. Experiments with a variety of large, real datasets demonstrate that the Dumpy solutions achieve considerably better efficiency, scalability and search accuracy than its competitors.

CCS Concepts: • **Information systems → Data management systems**; **Temporal data**; **Database query processing**.

Additional Key Words and Phrases: data series indexing, similarity search

ACM Reference Format:
Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. 2023. Dumpy: A Compact and Adaptive Index for Large Data Series Collections. *Proc. ACM Manag. Data* 1, 1, Article 111 (May 2023), 27 pages. https://doi.org/10.1145/3588965

*Corresponding author

Authors' addresses: Zeyu Wang, School of Computer Science, Fudan University, Shanghai, China, zeyuwang21@m.fudan.edu.cn; Qitong Wang, LIPADE, Université Paris Cité, Paris, France, qitong.wang@etu.u-paris.fr; Peng Wang, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China, pengwang5@fudan.edu.cn; Themis Palpanas, LIPADE, Université Paris Cité & IUF, Paris, France, themis@mi.parisdescartes.fr; Wei Wang, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China, weiwang1@fudan.edu.cn.

2836-6573/2023/5-ART111 $15.00
https://doi.org/10.1145/3588965

1 INTRODUCTION

Massive data series collections are now being produced by applications across virtually every scientific and social domain [31, 43, 74], 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 [8, 44]. In this context, similarity search is an essential primitive operation, lying at the core of several other high-level algorithms, e.g., classification, clustering, motif discovery and outlier detection [10, 11, 43, 46, 59, 64].

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 [21, 22]. 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 [5, 7, 15, 37, 62]. In such applications, approximate query result quality is sufficient to support downstream applications [16]. 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 [54] 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% [69]. 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 [22], only a few of them can robustly support large data series collections, e.g., over 100GB (which is why techniques for approximate search [22], as well as progressive search for exact [20] and approximate [29] query answering have been studied). Among them, DSTree [65] and the iSAX index family [45] 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 [13, 14, 49, 50, 52, 67, 70], 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% [22]. 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 [56] does not in principle limit the fanout of a node, popular iSAX-family indexes [12, 14, 38, 48, 49, 52, 67, 70] 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 [12, 56] often lead to sub-optimal decisions (cf. [68], Section 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 [68]. 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., 128MB) [68] 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 [17, 24]. 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.

Our contributions 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) Our experimental evaluation with a variety of synthetic and real datasets demonstrates that Dumpy and its variants provide consistently faster index building times (4x on average), and higher approximate query accuracy (65% higher MAP on average) than the SOTA competitors, with query answering times in the order of milliseconds.

2 RELATED WORK

[Data series indexes] Dozens of methods have been proposed to index massive data series collections [21, 22]. Among these, the SAX-based indexes [45] have gained popularity and achieved SOTA performance. Following the initial iSAX [56] index, iSAX2.0 and iSAX2+ [12, 13] provide faster index building through novel bulk loading and node splitting strategies, ADS [70] optimizes the combined index building and query answering time, ULISSE [39] supports subsequence similarity search, SEAnet [63] improves query results quality for high-frequency time series using deep learning embeddings, while DPiSAX [66], Odyssey [14], PARIS [50], MESSI [51], SING [52], and Hercules [19] exploit distribution and modern hardware parallelism. These indexes all inherit the original binary structure of iSAX, which limits their intra-node series proximity.

ADS [70], 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 [21].)

TARDIS [68] 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 [33, 34] 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 [65] 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.

[High-dimensional vector indexes] According to recent studies [21, 22], 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 [41, 42, 62], 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 [23, 24, 57]. Product quantization family methods [6, 25, 28, 47] 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 [22]. LSH (Local Sensitive Hashing) family methods [26, 27, 36, 40, 58], though providing probabilistic guarantees, have been shown to fall behind data series indexes in terms of time and space [22].

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 $s = \{p_1, p_2, \ldots, p_l\}$ is a sequence of points $p_i = (x_i, t_i)$ where each point is associate with a value x_i and a position t_i, satisfying that $t_1 < t_2 < \cdots < t_l$. l denotes the length of data series s.*

In this paper, 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.

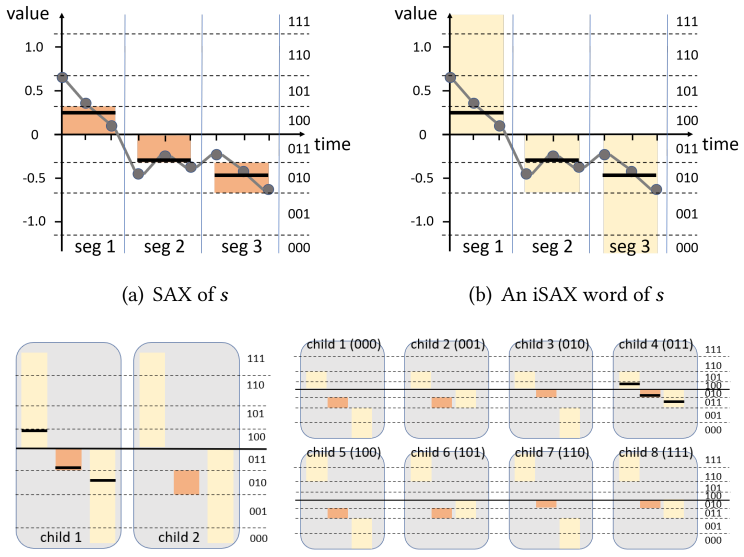
Definition 2 (kNN Query). *Given an integer k, a query data series q and a distance measure $dist$, a **kNN Query** retrieves from the database the set of series $R = \{r_1, r_2, \ldots, r_k\}$ such that for any other series s in the database and any $r_i \in R$, $dist(r_i, q) \leq dist(s, q)$.*

The choice of the distance measure depends on the particular application. However, Euclidean Distance (ED) is one of the most popular, widely studied and effective similarity measures for large data series collections [18]. We also support Dynamic Time Warping (DTW) [53] in the meanwhile as the inherent property of the iSAX index family [49]. 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 [56]. iSAX is a dynamic prefix of SAX words, and SAX is a symbolization of PAA (Piecewise Aggregate Approximation) [32]. We briefly review these techniques with the example in Figure 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 Figure 1(a), 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



(a) SAX of *s*

(b) An iSAX word of *s*

(c) Split on segment 2

(d) Split on all segments

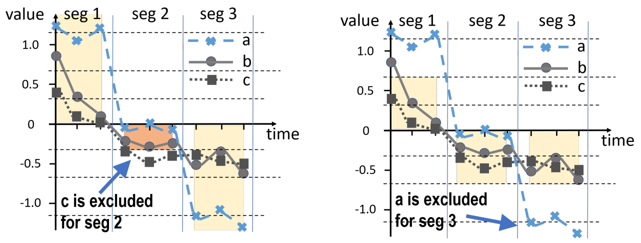
Fig. 1. (a) and (b) are the PAA, SAX and iSAX representation ($w = 3$, $b = 3$). (c) and (d) are the node splitting for the iSAX-index family in two- and full-ary structures, respectively.

coefficients on each segment. In Figure 1(a), SAX(s,3,8)=[100,011,010]. The SAX word represents a *region* formed by the value ranges in w segments, 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 Figure 1(b) is iSAX(s,3,8)=[1,01,0][1]. 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 [45] 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, besides the common iSAX word, stores the raw data and SAX words of every series in it. When the size of a leaf node exceeds the leaf size threshold th, the leaf gets transferred into an internal node, and all series in it are split accordingly. There are two splitting strategies in the iSAX-index family. One is the binary split (see Figure 1(c)) which splits a node by doubling the cardinality of the iSAX symbol on *one* segment. These two refined iSAX words representing smaller space on one segment, are assigned to two new leaf nodes.

Considering a node contains the iSAX word in Figure 1(b) (i.e., [1,01,0]) as an example. The binary strategy splits on the second segment and generates two new leaves with iSAX words [1,010,0] and [1,011,0], respectively. The classical iSAX index [56] adopts the binary split that leads to a binary tree structure below the first layer. The second strategy is to split on all w segments (i.e., double the cardinality of the iSAX symbol on each segment) and generate at most 2^w child nodes (see

[1]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.



(a) Skewed region: *a* and *b* are wrongly grouped in node 1-011-0
(b) Even region: *b* and *c* are correctly grouped in node 10-01-01

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.

Figure 1(d)). This split mode generates a totally full-ary structure, adopted by the recently proposed iSAX-family index, TARDIS [68].

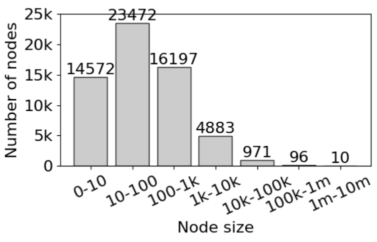
4 PROXIMITY-COMPACTNESS TRADE-OFF

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.

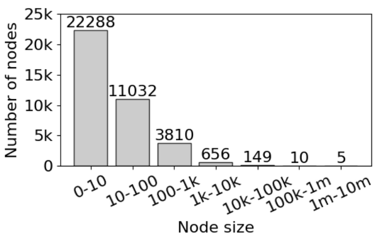
[Proximity problem of binary structures] In a binary structure index like iSAX, the SOTA splitting strategy [12] targets to balance the number of series in the two child nodes, by choosing a segment on which the mean value is close to the breakpoint. However, this strategy leads to skewed splits: it 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 Figure 2(a), 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 Figure 2(a)), 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 Figure 2(b), will successfully group series *b* and *c* together. It is important to note 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 [68] 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



(a) Random walk



(b) DNA

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

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 our solution Dumpy. Based on a novel adaptive multi-ary structure, Dumpy can hit the right balance of the proximity-compactness trade-off.

In the following, we introduce our basic idea in Section 5.1 and describe the workflow of Dumpy building in Section 5.2. In Section 5.3 we present the adaptive split strategy and the leaf node packing algorithm in Section 5.4. In Section 5.5 and 5.6, we describe the searching and updating algorithm of Dumpy, respectively.

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 Figure 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. Node N_1 is an internal node with iSAX word [0,1,1,1]. $csl(N_1) = [1, 3, 4]$ means λ_{N_1}=3 and we split N_1 on segments 1, 3, 4.

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. Section 5.3).

To better facilitate our adaptive splitting algorithm, we propose a new index-building workflow based on the information of all series (cr. Section 5.2). The building workflow of previous SAX-based

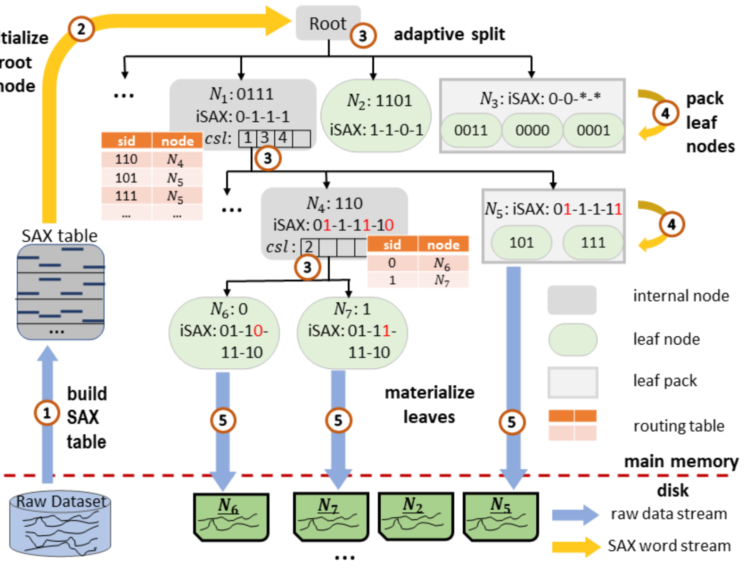


Fig. 4. Index structure (w=4) and building workflow.

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 [35]. 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. Section 5.4).

5.2 Workflow of Dumpy Building

The index-building workflow of Dumpy is demonstrated in Figure 4. Dumpy follows the most advanced building framework of the iSAX-family index [71] but changes two key designs that provide a better index structure and higher building efficiency. The classical framework is a two-pass procedure. In 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 the aforementioned 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,

Algorithm 1 Dumpy Construction

```
Input: Dataset $db$, leaf node capacity $th$, parameters $r$,$\rho$
Output: Dumpy root $N_r$
 1: for each series $s$ read in order from $db$ do
 2:     Compute SAX word of $s$, and insert it into SAX table $T_{sax}$
 3: Initialize root $N_r$.
 4: $Split(N_r, T_{sax}, th)$
 5: $NodePacking(N_r, r, \rho, th)$ and label the new leaf node.
 6: Initialize an empty hash table as $buf$
 7: for each series $s$ read in order from $db$ do
 8:     if $buf$ is full then
 9:         flush data in $buf$ to corresponding leaf node files.
10:     leaf node $N_l = N_i.route2Leaf(T_{sax}[s])$
11:     $buf[N_l].insert(s, T_{sax}[s])$
12: flush $buf$ and clear it.
13: free($T_{sax}$)
14: return $N_r$
```

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.

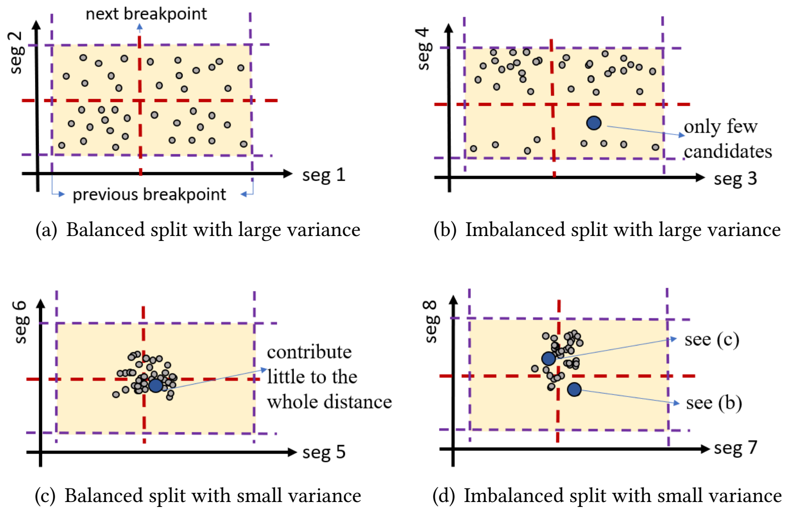
[Write to disk after leaf node packing] A large fanout usually generates numerous small nodes before leaf node packing, as shown in Figure 3. Considering in the second pass we flush the series of each relevant leaf node in a batch, the number of leaf nodes approximately decides how many random disk writes 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.

We now present the complete Dumpy index construction workflow using Algorithm 1. In Stage 1, the SAX table is built from the raw dataset (lines 1-2), and in Stage 2 the root node is initialized (line 3). In Stage 3, the adaptive split is first executed on the root node, and then recursively on all internal nodes whose size exceeds th (line 4 and Algorithm 2). In Stage 4, we traverse the index tree and pack leaf nodes under the same parent (line 5 and Algorithm 3). Now we finish the construction of index structure. In stage 5 we materialize leaves (line 6-12). Line 6 prepares a buffer for leaf nodes. Then the raw series are read from the disk, routed to leaf nodes, and cached in the buffer along with its SAX words (lines 7, 10-11). Once the buffer is full, it flushes data to the corresponding file of each node (lines 8-9).

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. And to measure compactness, we consider both the variance of fill factors of child nodes and the ratio of overflowed nodes to pursue a balanced split and avoid the bias for small or large fanouts. We now give the formal definition of the proposed objective function, and then introduce the design principles in detail.



(a) Balanced split with large variance (b) Imbalanced split with large variance

(c) Balanced split with small variance (d) Imbalanced split with small variance

Fig. 5. Four typical scenarios while choosing the segments to split. The grey points represent the same set of data series under four different split plans. The blue points are the query series.

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:

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

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[2], $o \in [0,1]$ is the ratio of overflowed child nodes (size $> th$), σ_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 [9, 25, 47], considering large variances usually mean large information entropy [55]. We also explain this in Figure 5. 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 Figure 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.

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

We further illustrate the heuristic behind the objective function in Figure 5, where we present four different split plans for the same group of data series. Without loss of generality, we assume all plans choose two segments, i.e., $|csl(N)| = 2$. Then we project the relevant data on a two-dimensional SAX space (different spaces for different plans). The plan chosen by our objective function is Figure 5(a), which has a large variance and balanced child nodes. The plan in Figure 5(b) is inferior to Figure 5(a) due to the imbalance of child nodes, which leads to lower search accuracy for a lack of sufficient candidate series. The problem of the plan in Figure 5(c) is its small variance, which means that series are rather close to each other in these dimensions, on which the distance between series contributes little to the whole distance that is accumulated by all w dimensions. That is, the distance of query and series in this subspace (small or large) is not informative for their actual distance in the whole space. Specifically, close series in this subspace may have a large distance actually. Therefore, the proximity of the series inside a node is crucially weakened. Figure 5(d) shows a worst split plan candidate, and our objective function can easily eliminate it.

5.3.2 Find the Optimal Split Plan. A naive solution to finding the optimal split plan under our objective function is to iterate and evaluate all 2^w-1 split plans. To evaluate the objective function of each plan, it needs at least four passes of all series, rendering the CPU calculation a bottleneck in index building. To reduce the complexity, 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. We first present each technique in detail, then wrap them up in the complete node splitting algorithm in Algorithm 2.

[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.

Proof.

$$\sum_{cs \in csl(N)} Var(\Pi_{cs}(\mathcal{X}_N)) = \frac{1}{c_N} \sum_{cs} \sum_{i=1}^{c_N} (x^i_{cs} - \mu_{cs})^2$$

$$= \frac{1}{c_N} \sum_{i=1}^{c_N} \sum_{cs} (x^i_{cs} - \mu_{cs})^2 = \frac{1}{c_N} \sum_{i=1}^{c_N} \|\boldsymbol{x}'^i - \boldsymbol{\mu}\|^2 = Var(\mathcal{X}'_N)$$

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] We now consider pruning impractical plans based on simple heuristics. Taking two extreme cases as examples: a plan that splits a node of size th+1 on w segments will generate excessively small nodes, and a plan that splits a million-sized node on one segment will generate two huge nodes of size far exceeding th. Hence, it is natural 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

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

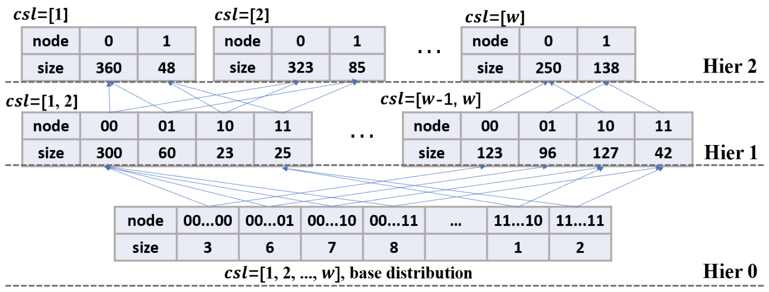


Fig. 6. Hierarchically compute child node sizes for each candidate split plan.

In practice, we empirically set $F_l = 50\%$ and $F_r = 300\%$, which generally achieves 16x speedup and 99% accuracy on average. The acceleration is especially remarkable on relatively small nodes, whose pruned search space is very small.

[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. We observe that 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 $csl^i(N)$. Supposing $csl^i(N) = [1]$ and $csl^j(N) = [1, 2]$ (the upper left part in Figure 6), and the sizes of four child nodes under $csl^j(N)$ are $c_{N_{00}} = 300$, $c_{N_{01}} = 60$, $c_{N_{10}} = 23$, $c_{N_{11}} = 25$, (nodes are represented by their *sid*s), then the sizes of two child nodes under $csl^i(N)$ can be computed as $c_{N_0} = 300 + 60 = 360$ and $c_{N_1} = 23 + 25 = 48$. 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 distribution we have gained in a hierarchical way and avoids traversing all the series for each plan.

We summarize our splitting strategy in Algorithm 2. The backbone is in lines 4-13 and 19-42 while lines 14-16 split data according to the split plan and lines 17-18 recursively split the overflowed child nodes. We first prepare the data variance on each segment (lines 4-6) and the base distribution (lines 7-10), and then determine the range of the fanout (line 11). Then we evaluate each split plan in a hierarchical way (lines 19-42) with a hash set to avoid duplicated evaluation. Lines 21-26 iterate each combination from the base plan as the current split plan and compute its distribution. After that, the objective function is evaluated to update the best-so-far answer. Finally, the sub-plans of the current plan are evaluated similarly.

5.4 Leaf Node Packing

As mentioned in Section 5.1, data series are usually distributed unevenly in child nodes when the fanout is large, leading to a large number of small leaf nodes and hence, degeneration of the indexing performance. TARDIS [68] adopts a node-size-based strategy to merge subtrees into a partition, while Coconut [33] proposes a sorting technique for iSAX words, both of which crucially weaken or even abandon the *pruning power* of resulting index. To tackle this problem, 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. For example, merging two nodes with *sid* 0010 and 0100 is better than merging 0010 and 0101, since we demote two bits and get $0**0$ instead of demoting three bits and

Algorithm 2 Split

```
Input: node $N$, SAX table $T_{sax}$, leaf node capacity $th$
1: if $N$ is root node then
2:     $csl(N) = [1, 2, \ldots, w]$
3: else
4:     segVar = []
5:     for $i$ = 1 to $w$ do
6:         segVar.push(variance of data in segment $i$)
7:     Initialize $baseDist$ as a list of size $2^w$: [0,0,...,0].
8:     for each data series $s$ in $N$ do
9:         $sid, \_ = promoteiSAX(iSAX(N), T_{sax}[s], [1, 2, ..., w])$
10:        $baseDist[sid]$++
11:    Calculate $\lambda_{max}$,$\lambda_{min}$ according to Equation 3
12:    Initialize $visit$ as an empty set
13:    $csl(N)$ = calcDist($baseDist$, $[1, 2, ..., w]$, $\lambda_{max}$, $\lambda_{min}$, $visit$, $segVar$, null)
14: for each data series $s$ in $N$ do
15:    $sid, isax_{new} = promoteiSAX(iSAX(N), T_{sax}[s], csl(N))$
16:    Insert $s$ to $N.children[sid]$
17:    Label $N.children[sid]$ with $isax_{new}$
18: for each of $N$'s child node $N_i$ satisfying $c_{N_i} > th$ do
19:    $Split(N_i, T_{sax}, th)$

Function calcDist($baseDist$, $basePlan$, $\lambda_{cur}$, $\lambda_{min}$, $visit$, $segVar$,$bsf$)
20: if $\lambda_{cur} < \lambda_{min}$ then
21:    return $bsf$
22: for each combination of cardinality $\lambda_{cur}$ from $basePlan$ as $csl^{cur}$ do
23:    if $visit$ does not contain($csl^{cur}$) then
24:        Initialize $Dist$ as a list of size $2^{\lambda_{cur}}$: [0,0,...,0]
25:        for each $item$ in $baseDist$ do
26:            $sid_{new}$ = extract bits in $csl^{cur}$ from $item.sid$
27:            $Dist[sid_{new}]$+=$item.size$
28:        Compute $score$ using Equation. 1
29:        if score > $bsf$.score then
30:            $bsf = csl^{cur}$
31:        calcDist($Dist$,$csl^{cur}$, $\lambda_{cur}$-1, $\lambda_{min}$, $visit$, $segVar$, $bsf$)
32:        $visit$.insert($csl^{cur}$)
33: return $bsf$

Function promoteiSAX(iSAX word $isax$, SAX word $sax$, chosen segments list $csl$)
34: $sid = 0$
35: $isax_{new} = isax$
36: for each segment $seg$ in $csl$ do
37:    $nb = len(isax[seg])$
38:    $sid = sid << 1$ + the $(nb + 1)$-th bit of $sax[seg]$
39:    $isax_{new}[seg]$ = first $nb + 1$ bits of $sax[seg]$
40: return $sid$, $isax_{new}$
```

getting the coarser $0 * **$. We define the demotion bits as the different bits between the *sid*s 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 this algorithm are shown in Algorithm 3. We first sum the sizes of all small leaf nodes (lines 1-5) and randomly initialize the pack list with $\lfloor sum\_size/th \rfloor$ leaf nodes, which is the least number to hold the small nodes (line 6). Then we iterate all the other small nodes and select or create a pack for them (lines 7-20). For node N, we check each of the existing packs in the pack list and select the one with the least demotion cost (lines 14-16, 19-20). When the pack demotes more than $\rho * \lambda_{N_p}$ bits, we simply give up this choice (lines 11-12). Besides, we also ensure the size

Algorithm 3 Node Packing

```
Input: parent node $N_p$, small node threshold $r$, demotion ratio $\rho$, leaf capacity $th$
 1: $nodes$ = [], $sum\_size$ = 0
 2: for each of $N_p$'s child node $N_l$ that is not splitted do
 3:     if $c_{N_l} < r * th$ then
 4:         $nodes$.push($N_l$)
 5:         $sum\_size$ += $c_{N_l}$
 6: Initialize pack list $pl$ with $\lfloor sum\_size/th \rfloor$ leaf nodes
 7: for each node $N$ in $nodes$ do
 8:     $bsf$ = (null, $\lambda_{N_p}$)
 9:     for each pack $p$ in $pl$ do
10:         $p' = p.insert(N)$
11:         if $c'_p > th$ or number of demotion bits of $p' > \rho * \lambda_{N_p}$ then
12:             continue
13:         else
14:             $cost$ = increased number of demotion bits in $p'$
15:             if $cost < bsf$.cost then
16:                 $bsf = (p, cost)$
17:     if $bsf$.pack == null then
18:         $pl$.push(new pack initialized with $N$)
19:     else
20:         $bsf$.pack.insert($N$)
21: update $N_p$'s routing table
22: for each of $N_p$'s child node $N_i$ satisfying $c_{N_i} > th$ do
23:     $NodePacking(N_i, r, \rho, th)$
```

of all the packs will not exceed th leading to new splits (lines 11-12). When there is no qualified pack, we will create a new pack initialized with node N (lines 17-18). Finally, we update the routing table of N_p.

The definition of small nodes can be defined by use cases. For a static dataset, the small node threshold r can be set to 1 to improve performance, while in a dynamic dataset, r can be set dynamically according to the historical updating frequency. That is, small r for intensive updating use cases and vice versa.

5.5 Search Algorithm

Dumpy supports two styles of query answering algorithms. The first style follows the classical pruning-based search algorithm [22]. As a SAX-based index, Dumpy can conduct an efficient search (including the exact, δ-ϵ-approximate search and etc. [22, 56]) by pruning irrelevant leaf nodes using lower-bounding distances of iSAX words [22, 56]. 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.

As shown in Algorithm 4, the input includes an additional parameter that restricts the visited node number. The search process starts from the root node and ends with a node that has fewer nodes than the input or an empty node (lines 1-4). Then we sort the sibling of the ending node according to their lower bound distance (as described in [56] for ED and [49] for DTW) in ascending order (line 5). Finally, we iterate the sorted sibling nodes or subtrees until reaching the maximal visited node number and return kNN (lines 6-9, the concrete search procedure is omitted since it is the same as in other search algorithms).

Algorithm 4 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 $k$NN among the visited series
```

5.6 Updates

As a fully functional index, Dumpy also supports updates (insertion and deletion) besides bulk loading. A major difference is that Dumpy no longer collects the information of all series beforehand on a dynamic dataset. Though, since all the SAX words are stored in leaves along with the raw series (as the same with iSAX-index family [56]), we can re-organize the index structure when an internal node's fanout and size do not satisfy the constraint in Equation 3.

Specifically, we read all the SAX words in the leaves rooted at this node and follow the same workflow in Algorithm 1. Note that this process can be executed in a background thread without blocking the front-end service. During this period, the query series of updates will come into both the old structure and the new but unfinished one. Once the background work is finished, we replace the old subtree with the new subtree, and free all the space occupied by the old one. Another difference is when the query series falls into a full pack. Then we can simply 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 [56, 70]. 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 Equation 3. Then the cost of computing the variance of each segment, the base distribution and the routing target is $O(4wc_N)$. The number of calculations of the *calcDist* function is $\binom{w}{\lambda_{max}} * 2^w + \sum_{i=\lambda_{min}}^{\lambda_{max}-1} \binom{w}{i} 2^{\lambda_{i+1}}$, where the first term corresponds to evaluating all possible plans of the max fanout, i.e., using λ_{max} segments (cf. Figure 6, level Hier 1 of the hierarchy), and the second term to evaluating all possible plans of smaller fanouts (cf. Figure 6, level Hier 2). 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 $O(\sum_N (4wc_N + \binom{w}{\lambda_{max}} * 2^w + \sum_{i=\lambda_{min}}^{\lambda_{max}-1} \binom{w}{i} 2^{\lambda_{i+1}} + 2^{\lambda_N} * np))$.

Random disk writes can have a significant cost when building Dumpy (cf. Figure 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 $2/\overline{\lambda}$x smaller, where $\overline{\lambda}$ 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. Section 7.1), 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 [17, 24]. 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. For example, a node with *sid* = 000 will check the series of neighboring nodes 100 on the first chosen segment and duplicate the series that are very close to 000 into 000. The same process applies to nodes 010 and 011 on the second and third chosen segments, respectively.

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.

We also claim that neither binary nor full-ary structures can easily adopt similar fuzzy boundary optimizations. In a binary structure, since each leaf node has only one sibling node, it is prone to produce excessive duplication series and generate more splits. This results in a deeper index, which rather decreases the search accuracy. As for the full-ary structure, the excessive number of leaf nodes translates into an overwhelming number of replication destinations, introducing unacceptable storage overhead. We have experimentally verified these observations but omit the results and the detailed algorithms due to the lack of space.

7 EXPERIMENTS

[Environment] Experiments were conducted on an Intel Core(R) i9-10900K 2.80GHz 10-core CPU with 4*32GB 2400MHz 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/Dumpy.

[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 to ensure sufficient hardness [73], 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 a standard Gaussian distribution $N(0,1)$. It has been extensively used in the existing works [2, 21, 22, 72]. We generate 50-800 million Rand series of different lengths (50GB-800GB). **DNA** [1] 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 (~113GB). The second real dataset, **ECG** (Electrocardiography), is extracted from the MIMIC-III Waveform Database [30]. It contains over 97 million series of length 320 (~117GB), sampled at 125Hz from 6146 ICU patients. The last real dataset, **Deep** [61], comprises 1 billion vectors of size 96, extracted from the last CNN (convolutional neural network) layers of images.

[Algorithms] In the iSAX-index family, we take **iSAX2+** as the SOTA binary structure [22]. We also implement a stand-alone version of **TARDIS** as the SOTA full-ary structure and use 100% sampling percent. **DSTree** [65] is also included as one of the SOTA data series indexes [22]. 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.

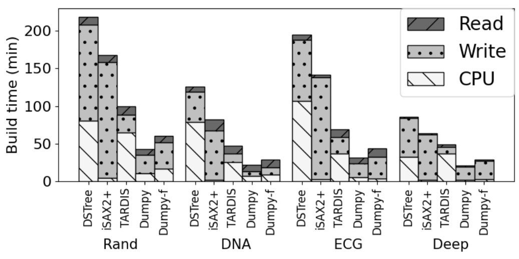
All the codes are open-source, implemented in C/C++, and compiled by g++ 9.4.0 with -O3 optimization. We use the optimized versions of DSTree and iSAX2+ [21].

[Parameters] We set the number of segments $w = 16$, SAX cardinality $c = 64$ (i.e., $b = 8$), and the leaf size threshold $th = 10000$. The memory buffer size for index building is set to $4GB$ unless specified. The replication times of each series in Dumpy-f is set to at most 3.

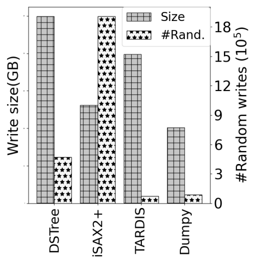
[Measures] Similar with other works [4, 22, 47], we use Mean Average Precision (MAP) [60] as the accuracy measure, which is defined as the mean value of AP on a group of queries. For query s_q, AP equals to $\frac{1}{k}\sum_{i=1}^{k} P(s_q, i) * rel(i)$, where $P(s_q, i)$ is the ratio of true neighbors 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, 68], 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 [49, 53].

7.1 Index Building

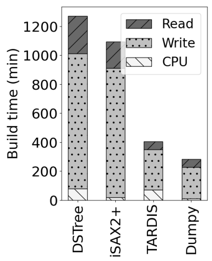
7.1.1 Efficiency. First, we evaluate the index building efficiency in four datasets on SSD, and the results are shown in Figure 7(a). In all four datasets, Dumpy outperforms the other three methods by a large margin, i.e, 5.3 times faster than DSTree, 3.8 times than iSAX2+ and 2.5 times than TARDIS on average. Dumpy-f only incurs small overheads (about **38%**) on Dumpy and is considerably faster than DSTree and iSAX2+. The advantage of Dumpy mainly comes from the reduction of random disk writes. As shown in Figure 7(b), the number of random writes (#Rand.) of iSAX2+ is



(a) Building time across four datasets on SSD



(b) Disk writes in building



(c) Building time on HDD

Fig. 7. Index building (4GB memory)

Table 1. Index structure statistics

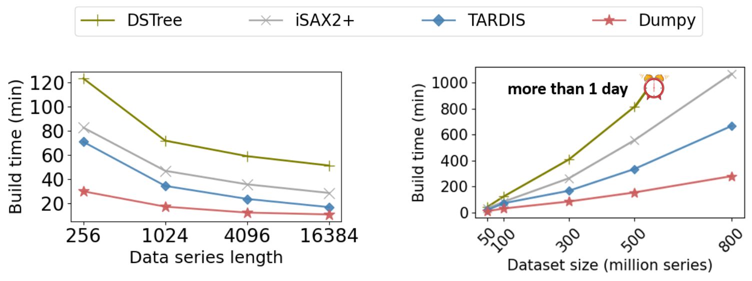
Dataset	Method	#Leaves	#Nodes	Height	Fill factor	Index Size (MB)[1]
Rand	iSAX2+	73563	86945	20	13.59%	16
	DSTree	17847	35693	32	56.03%	9
	TARDIS	8516867	8520065	**3**	0.11%	732
	Dumpy	**14106**	**19418**	7	**70.89%**	**3**
DNA	iSAX2+	42906	47885	25	6.14%	9
	DSTree	5833	11665	43	45.16%	3
	TARDIS	1011436	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**

[1] Size of in-memory index structure only.

~**3.5**x more than DSTree and ~**17**x more than Dumpy and TARDIS, making it the dominating factor in index building time than the number of writing bytes (#Seq.) in both SSD and HDD. Though TARDIS also enjoys low I/O costs for its compact and large partition, it needs more CPU time to serialize the enormous index structure into each partition. Results on HDD in Figure 7(c) are similar to those on SSD.

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. That of DSTree is slightly larger than Dumpy, and that of iSAX2+ is generally >**3**x more than DSTree. It verifies the complexity analysis in Section 5.2. Noting that the index building time of DSTree is even longer than iSAX2+ (Figure 7(a)), due to its high CPU cost and large writing bytes incurred by EAPCA calculations. TARDIS generates million-level leaves and has a low fill factor for its full-ary structure. These nodes are organized in large partitions where each partition is 128MB, as the setting of [68].

7.1.2 Scalability. Next, we test the scalability in Rand datasets by increasing data size from 50GB to 800GB, 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 100GB when the length is varied, as the same design with the benchmark [21].



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

Fig. 8. Index scalability (32GB memory)

Figure 8 presents the index building time with 32GB 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 **0.9904**, 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.

7.2 Query Processing

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

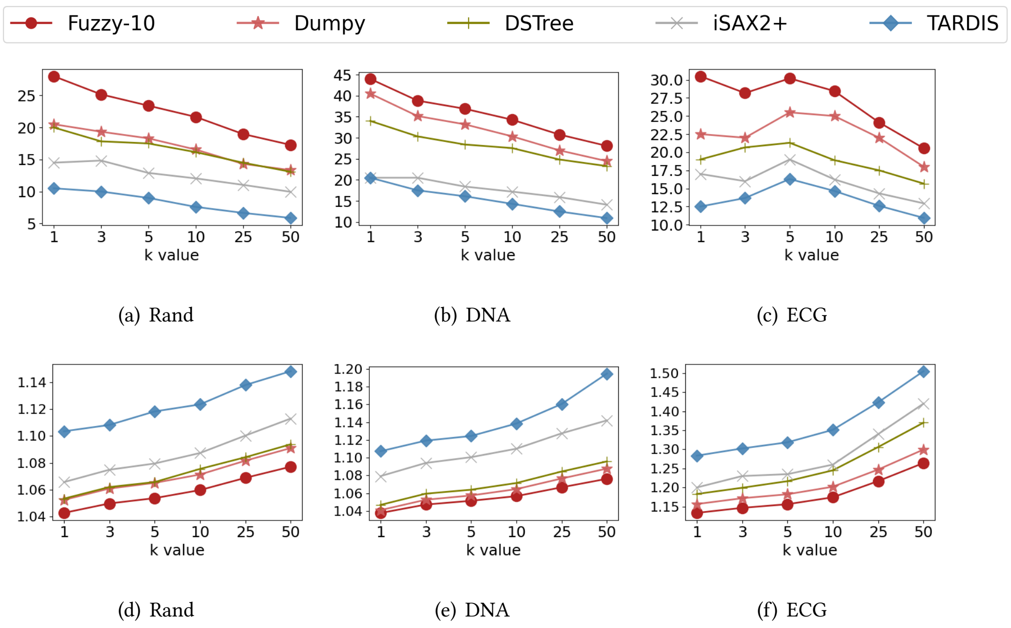
7.2.1 Approximate Search. First, we evaluate the accuracy of approximate results across four datasets.

[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 Figure 9. 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 three 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 as described in Figure 5 (>**20**% leaf nodes have one segment using more than 4 bits than other segments). Moreover, Dumpy-Fuzzy has higher accuracy than Dumpy and other approaches, which verifies our duplication strategy.

[Search multiple nodes] In Figure 10, 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 (Figure 11). The accuracy on different lengths shares similar rankings.

Figure 12 shows the average query time of different approaches for approximate top-50 queries. It can be seen that Dumpy has a similar query processing time with iSAX2+ and is faster than DSTree. That is, our approach can achieve higher accuracy with a smaller time cost. TARDIS is much slower because it pays lots of time to deserialize the large index partition.

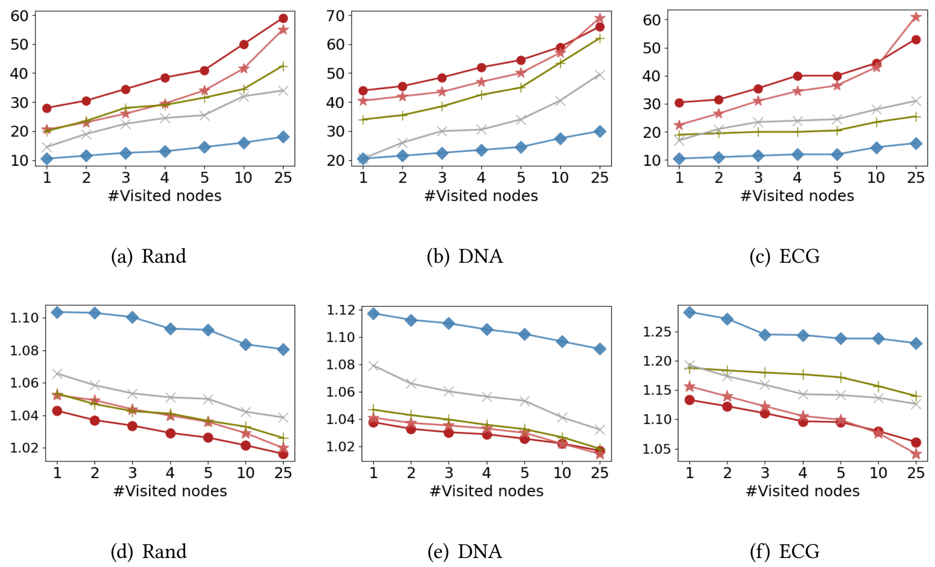
Our results show that Dumpy can achieve 60%-70% MAP within 100ms on TB-level datasets, while its index building time is 4x faster than the SOTA competitors. These results demonstrate that Dumpy fulfills the requirements of many kNN-based applications.



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

(d) Rand (e) DNA (f) ECG

Fig. 9. Approximate search under ED (search one node).



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

(d) Rand (e) DNA (f) ECG

Fig. 10. Extended approximate search under ED (k=1).

[Upper bound of approximate NN distance] Dumpy (like any other index) does not provide any accuracy guarantees for its (ng-)approximate search results [21, 22]. Nevertheless, we can provide an upper bound for the distance between the approximate NN and the query series. This upper bound can be interpreted as the worst-case accuracy of the approximate search results of Dumpy.

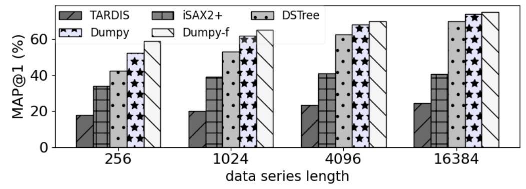


Fig. 11. Extended approx. Search vs. series lengths (search 25 nodes)

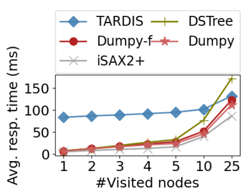


Fig. 12. Query time

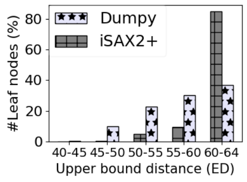
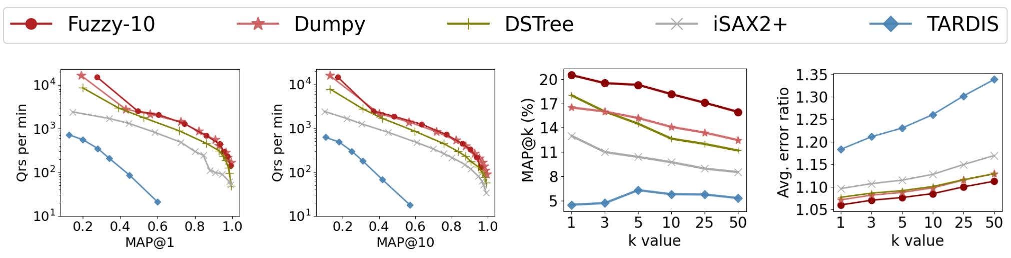
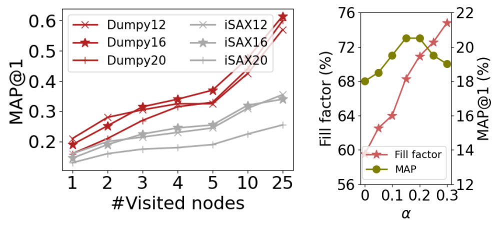


Fig. 13. Histogram of upper bounds of the distance



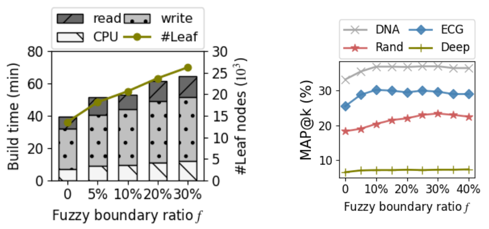
(a) $k = 1$ (b) $k = 50$ (a) MAP (b) Avg. error ratio

Fig. 14. Efficiency v.s. accuracy Fig. 15. Approx. Search under DTW



(a) Effect of w (b) Effect of α

Fig. 16. Influence of parameters



(a) On index building (b) On Search Accuracy

Fig. 17. Influence of f

Given the target leaf node, this upper bound distance is defined as $\sqrt{\frac{l}{w}\sum_{i=1}^{w} range_i^2}$, where $range_i$ is the distance between two breakpoints in the i-th segment. It corresponds to the distance when the query and the nearest neighbor series are located *on* the opposite boundaries of each segment.

In Figure 13, we use a histogram to show the distribution of these upper bound distances for the leaf nodes of the Dumpy and iSAX indexes. The histogram shows that over **80**% of the iSAX leaf nodes have loose bounds (60-64), whereas over **60%** of the Dumpy leaves have tight bounds (40-60). This is because Dumpy chooses to split the coarser segment, which leads to a smaller worst-case distance than iSAX.

[Efficiency vs 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 Figure 14. Benefiting from the high-proximity nodes and compact index structure, Dumpy surpasses its competitors in both low and high-precision intervals.

[Searching under DTW] In this experiment, we compare the accuracy under DTW distance in Figure 15. 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 in the error ratio among all the methods tend to be smaller (except for TARDIS).
[Influence of parameters] We also investigate the influence of parameters on the accuracy, including the number of segments w, the weight factor α in the objective function Equation. 1 and boundary ratio f for Dumpy-Fuzzy.

In Figure 16(a), we vary w for iSAX and Dumpy and observe that the best iSAX performance (when $w = 16$) is still worse than Dumpy (with any number of segments $w \in \{12, 16, 20\}$). This is explained by Dumpy's strategy to search for better-quality splits that fully exploit the iSAX summarization power of the available segments. We also observe that in both indexes, fewer segments tend to degrade the precision of iSAX summarizations, whereas more segments produce a large number of leaf nodes, which harms the index performance.

In Figure 16(b), when α increases from 0 to 0.3, the fill factor also keeps increasing, from 59% to 75%. The precision first increases since visiting more series in a node, and then decreases due to the reduction of intra-node proximity. As shown in Figure 5, when α decreases, the split plan tends to be like Figure 5(c), which means that the series in a node is not that similar. The sweet point for the precision occurs when α is about 0.2, which is used across different datasets in our experiments, showing the robust performance.

Second, we evaluate the stability of Dumpy-Fuzzy regarding different f, the fuzzy boundary ratio. In Figure 17(a) we observe that the number of leaves and the building time increases slowly as f increases. In Figure 17(b) ($k = 5$), the precision and error ratio improve with f increasing to **10**%, and remain relatively stable then. It implies that Dumpy-Fuzzy is stable regarding f, and choose f=10 for ECG and Deep and f=30 for Rand and DNA.

7.2.2 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. But only the node summarized with iSAX words can be pruned during searching. The results are reported using the average of 40 queries with $k = 50$ under ED and DTW. Overall, Dumpy achieves the best efficiency in all 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.

7.3 Complete Workloads

Finally, we compare different approaches when inserting new data series (Figure 18). 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.

Table 2. Exact Search. For response time, the first number in the brackets is the I/O time while the second is the CPU time.

	Method	Resp. time (s)	#Loaded Nodes	Pruning 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%

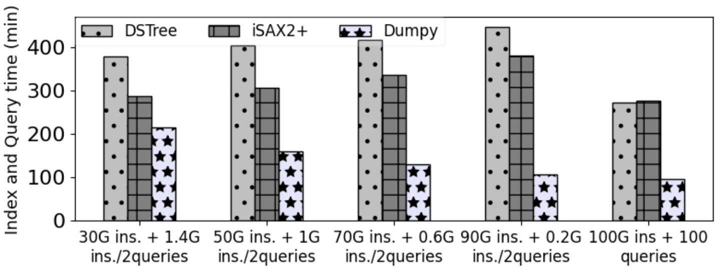


Fig. 18. Update performance (4GB memory)

8 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. A variant of Dumpy, Dumpy-Fuzzy can achieve even higher accuracy by clever partial duplication. Experiments with a variety of large, synthetic and real datasets demonstrate the efficiency, scalability, and accuracy of our solutions.

In future work, we plan to extend Dumpy to support subsequence matching. By designing a proper cost function and an efficient evaluation algorithm, Dumpy's adaptive splitting strategy can enhance the SOTA subsequence matching index, ULISSE [39], with higher-proximity nodes and hence better performance. Moreover, by absorbing the parallel paradigms of ParIS [50], SING [52] and TARDIS [68], Dumpy's overall performance can be further improved by exploiting modern hardware.

ACKNOWLEDGMENTS

This work is supported by the Ministry of Science and Technology of China, National Key Research and Development Program (No. 2020YFB1710001). Qitong Wang is supported by China Scholarship Council. We would also like to thank Prof. Karima Echihabi for her kind sharing of the optimized implementation of iSAX2+ and DSTree.

REFERENCES

[1] [n. d.]. National Center for Biotechnology Information (NCBI)[Internet]. https://www.ncbi.nlm.nih.gov/ Accessed March 14, 2022.

[2] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. 1993. Efficient similarity search in sequence databases. In *International conference on foundations of data organization and algorithms*. Springer, 69–84.

[3] Panagiotis Anagnostou, Petros Barbas, Aristidis G. Vrahatis, and Sotiris K. Tasoulis. 2020. Approximate kNN Classification for Biomedical Data. In *2020 IEEE International Conference on Big Data (Big Data)*. 3602–3607. https://doi.org/10.1109/BigData50022.2020.9378126

[4] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. 2018. HD-Index: Pushing the Scalability-Accuracy Boundary for Approximate kNN Search in High-Dimensional Spaces. *Proceedings of the VLDB Endowment* 11, 8 (2018).

[5] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (2023).

[6] Artem Babenko and Victor Lempitsky. 2014. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence* 37, 6 (2014), 1247–1260.

[7] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[8] Anthony J. Bagnall, Richard L. Cole, Themis Palpanas, and Konstantinos Zoumpatianos. 2019. Data Series Management (Dagstuhl Seminar 19282). *Dagstuhl Reports* 9, 7 (2019), 24–39. https://doi.org/10.4230/DagRep.9.7.24

[9] J.S. Beis and D.G. Lowe. 1997. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE, San Juan, PR, USA, 1000–1006.

[10] Paul Boniol, Michele Linardi, Federico Roncallo, and Themis Palpanas. 2020. Automated anomaly detection in large sequences. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1834–1837.

[11] Paul Boniol and Themis Palpanas. 2020. Series2graph: Graph-based subsequence anomaly detection for time series. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1821–1834.

[12] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn Keogh. 2010. iSAX 2.0: Indexing and mining one billion time series. In *2010 IEEE International Conference on Data Mining*. IEEE, 58–67.

[13] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with *i* sax2+. *Knowledge and information systems* 39, 1 (2014), 123–151.

[14] Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and Botao Peng. 2023. Odyssey: A Journey in the Land of Distributed Data Series Similarity Search. *Proc. VLDB Endow.* (2023).

[15] George Chen, Christina Lee, and Shah Devavrat. 2017. Nearest Neighbors for Modern Applications with Massive Data. https://nn2017.mit.edu/. In *Proceedings of the 31rd International Conference on Neural Information Processing Systems Workshop*.

[16] George H. Chen and Devavrat Shah. 2018. Explaining the Success of Nearest Neighbor Methods in Prediction. *Foundations and Trends® in Machine Learning* 10, 5-6 (2018), 337–588. https://doi.org/10.1561/2200000064

[17] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zhiyong Zheng, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. *Advances in Neural Information Processing Systems* 34 (2021).

[18] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1542–1552.

[19] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *Proc. VLDB Endow.* 15, 10 (2022), 2005–2018.

[20] Karima Echihabi, Theophanis Tsandilas, Anna Gogolou, Anastasia Bezerianos, and Themis Palpanas. 2023. ProS: Data Series Progressive k-NN Similarity Search and Classification with Probabilistic Quality Guarantees. *VLDBJ* (2023).

[21] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 112–127.

[22] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 403–420.

[23] Cong Fu, Changxu Wang, and Deng Cai. 2021. High Dimensional Similarity Search with Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), 1–1.

[24] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 461–474.

[25] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 744–755.

[26] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: indexable distance estimating codes for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[27] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 9, 1 (2015), 1–12.

[28] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[29] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. 2020. PANENE: A Progressive Algorithm for Indexing and Querying Approximate k-Nearest Neighbors. *IEEE Trans. Vis. Comput. Graph.* 26, 2 (2020), 1347–1360.

[30] Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific data* 3, 1 (2016), 1–9.

[31] Eamonn Keogh. 2006. A decade of progress in indexing and mining large time series databases. In *Proceedings of the 32nd international conference on Very large data bases*. 1268–1268.

[32] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems* 3, 3 (2001), 263–286.

[33] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-up Approach for Building Data Series Indexes. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 677–690.

[34] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *VLDB J.* 28, 6 (2019), 847–869. https://doi.org/10.1007/s00778-019-00573-w

[35] Flip Korn, Bernd-Uwe Pagel, and Christos Faloutsos. 2001. On the 'Dimensionality Curse' and the 'Self-Similarity Blessing'. *IEEE Trans. Knowl. Data Eng.* 13, 1 (2001), 96–111. https://doi.org/10.1109/69.908983

[36] Oleksandra Levchenko, Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Boyan Kolev, and Dennis Shasha. 2018. Spark-parsketch: a massively distributed indexing of time series datasets. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 1951–1954.

[37] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering* 32, 8 (2019), 1475–1488.

[38] Michele Linardi and Themis Palpanas. 2018. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *Proc. VLDB Endow.* 11, 13 (2018), 2236–2248. https://doi.org/10.14778/3275366.3284968

[39] Michele Linardi and Themis Palpanas. 2020. Scalable data series subsequence matching with ULISSE. *VLDB J.* 29, 6 (2020), 1449–1474.

[40] Wanqi Liu, Hanchen Wang, Ying Zhang, Wei Wang, Lu Qin, and Xuemin Lin. 2021. EI-LSH: An early-termination driven I/O efficient incremental c-approximate nearest neighbor search. *The VLDB Journal* 30, 2 (2021), 215–235.

[41] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[42] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[43] Themis Palpanas. 2015. Data series management: The road to big sequence analytics. *ACM SIGMOD Record* 44, 2 (2015), 47–52.

[44] Themis Palpanas. 2016. Big Sequence Management: A glimpse of the Past, the Present, and the Future. In *SOFSEM 2016: Theory and Practice of Computer Science - 42nd International Conference on Current Trends in Theory and Practice of Computer Science (Lecture Notes in Computer Science, Vol. 9587)*. 63–80.

[45] Themis Palpanas. 2020. Evolution of a Data Series Index. In *Information Search, Integration, and Personalization*. Springer International Publishing, Cham, 68–83.

[46] Themis Palpanas and Volker Beckmann. 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *SIGMOD Rec.* 48, 3 (2019), 36–40. https://doi.org/10.1145/3377391.3377400

[47] John Paparrizos, Ikraduya Edian, Chunwei Liu, Aaron J. Elmore, and Michael J. Franklin. 2022. Fast Adaptive Similarity Search through Variance-Aware Quantization. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE.

[48] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2018. Paris: The next destination for fast data series indexing and query answering. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 791–800.

[49] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. Messi: In-memory data series indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 337–348.

[50] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. Paris+: Data series indexing on multi-core architectures. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2020), 2151–2164.

[51] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. Fast data series indexing for in-memory data. *VLDB J.* 30, 6 (2021), 1041–1067.

[52] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. SING: Sequence Indexing Using GPUs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE, 1883–1888.

[53] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12).* 262–270. https://doi.org/10.1145/2339530.2339576

[54] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. 2015. Fast and Scalable Outlier Detection with Approximate Nearest Neighbor Ensembles. In *Database Systems for Advanced Applications.* 19–36.

[55] Claude Elwood Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[56] Jin Shieh and Eamonn Keogh. 2008. iSAX: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining.* 623–631.

[57] Suhas Jayaram Subramanya, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems.* 13766–13776.

[58] Yifang Sun, Wei Wang, Jianbin Qin, Ying Zhang, and Xuemin Lin. 2014. SRS: Solving c-Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *Proc. VLDB Endow.* 8, 1 (Sept. 2014), 1–12.

[59] Chang Wei Tan, Geoffrey I Webb, and François Petitjean. 2017. Indexing and classifying gigabytes of time series under time warping. In *Proceedings of the 2017 SIAM international conference on data mining.* SIAM, 282–290.

[60] Andrew Turpin and Falk Scholer. 2006. User performance versus precision measures for simple search tasks. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval.* 11–18.

[61] Skoltech Computer Vision. [n. d.]. Deep billion-scale indexing. http://sites.skoltech.ru/compvision/noimi Accessed March 14, 2022.

[62] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2101.12631* (2021).

[63] Qitong Wang and Themis Palpanas. 2021. Deep Learning Embeddings for Data Series Similarity Search *(KDD '21).* ACM, NY, USA, 1708–1716. https://doi.org/10.1145/3447548.3467317

[64] Qitong Wang, Stephen Whitmarsh, Vincent Navarro, and Themis Palpanas. 2022. iEDeaL: A Deep Learning Framework for Detecting Highly Imbalanced Interictal Epileptiform Discharges. *Proc. VLDB Endow.* 16, 3 (2022), 480–490.

[65] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proceedings of the VLDB Endowment* 6, 10 (2013), 793–804.

[66] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2017. Dpisax: Massively distributed partitioned isax. In *2017 IEEE International Conference on Data Mining (ICDM).* IEEE, 1135–1140.

[67] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2020. Massively Distributed Time Series Indexing and Querying. *IEEE Trans. Knowl. Data Eng.* 32, 1 (2020), 108–120. https://doi.org/10.1109/TKDE.2018.2880215

[68] Liang Zhang, Noura Alghamdi, Mohamed Y Eltabakh, and Elke A Rundensteiner. 2019. TARDIS: Distributed indexing framework for big time series data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE, 1202–1213.

[69] Kang Zhao, Liuyihan Song, Yingya Zhang, Pan Pan, Yinghui Xu, and Rong Jin. 2021. ANN Softmax: Acceleration of Extreme Classification Training. *Proc. VLDB Endow.* 15, 1 (2021), 1–10. https://doi.org/10.14778/3485450.3485451

[70] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* 1555–1566.

[71] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *The VLDB Journal* 25, 6 (2016), 843–866.

[72] Kostas Zoumpatianos, Yin Lou, Ioana Ileana, Themis Palpanas, and Johannes Gehrke. 2018. Generating data series query workloads. *VLDB J.* 27, 6 (2018), 823–846. https://doi.org/10.1007/s00778-018-0513-x

[73] Kostas Zoumpatianos, Yin Lou, Themis Palpanas, and Johannes Gehrke. 2015. Query workloads for data series indexes. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* 1603–1612.

[74] Kostas Zoumpatianos and Themis Palpanas. 2018. Data series management: Fulfilling the need for big sequence analytics. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE, 1677–1678.

Received July 2022; revised October 2022; accepted November 2022