The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art

Karima Echihabi
IRDA, Rabat IT Center,
ENSIAS, Mohammed V Univ.
karima.echihabi@gmail.com

Kostas Zoumpatianos
Harvard Univ.
kostas@seas.harvard.edu

Themis Palpanas
Paris Descartes Univ.
themis@mi.parisdescartes.fr

Houda Benbrahim
IRDA, Rabat IT Center,
ENSIAS, Mohammed V Univ.
houda.benbrahim@um5.ac.ma

ABSTRACT

Increasingly large data series collections are becoming commonplace across many different domains and applications. A key operation in the analysis of data series collections is similarity search, which has attracted lots of attention and effort over the past two decades. Even though several relevant approaches have been proposed in the literature, none of the existing studies provides a detailed evaluation against the available alternatives. The lack of comparative results is further exacerbated by the non-standard use of terminology, which has led to confusion and misconceptions. In this paper, we provide definitions for the different flavors of similarity search that have been studied in the past, and present the first systematic experimental evaluation of the efficiency of data series similarity search techniques. Based on the experimental results, we describe the strengths and weaknesses of each approach and give recommendations for the best approach to use under typical use cases. Finally, by identifying the shortcomings of each method, our findings lay the ground for solid further developments in the field.

PVLDB Reference Format:
Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB*, 12(2): 112-127, 2018.
DOI: https://doi.org/10.14778/3282495.3282498

1. INTRODUCTION

Data Series. A data series is an ordered sequence of data points[1]. Data series are one of the most common types of data, covering virtually every scientific and social domain, such as astrophysics, neuroscience, seismology, environmental monitoring, biology, health care, energy, finance, criminology, social studies, video and audio recordings, and many others [41, 72, 61, 75, 56, 39, 68, 8, 47, 53, 84, 38]. As more devices, applications, and users are connected with IoT technologies, an increasingly large number of data series is generated, leading to multi-TB collections [59]. We note that, when these collections are analyzed, the common denominator of most analysis algorithms and machine learning methods, e.g., outlier detection [17, 25], frequent pattern mining [66], clustering [45, 69, 67, 82], and classification [18], is that they are based on similarity search. That is, they require to compute distances among data series, and this operation is repeated many times.

Data Series Similarity Search. Similarity search is the operation of finding the set of data series in a collection, which is close to a given query series according to some definition of distance (or similarity). A key observation is that similarity search needs to process a sequence (or subsequence) of values as a single object, rather than as individual, independent points, which is what makes the management and analysis of data sequences a hard problem with a considerable cost. Therefore, improving the performance of similarity search can improve the scalability of data analysis algorithms for massive data series collections.

Nevertheless, despite the significance of data series similarity search, and the abundance of relevant methods that have been proposed in the past two decades [3, 22, 73, 63, 23, 13, 42, 66, 71, 81, 14, 89, 58, 85, 52, 51, 62], no study has ever attempted to compare these methods under the same conditions. We also point out that we focus on the *efficiency* of similarity search methods, whereas previous works studied the *accuracy* of dimensionality reduction techniques and similarity measures, focusing on classification [44, 27, 9].

In this experimental and analysis paper, we thoroughly assess different data series similarity search methods, in order to lay a solid ground for future research developments in the field. In particular, we focus on the problem of *exact whole matching similarity search in collections with a very large number of data series*, i.e., similarity search that produces exact (not approximate) results, by calculating distances on

[1] When the sequence is ordered on time, it is called a *time series*. However, the order can be defined by angle (e.g., in radial profiles), mass (e.g., in mass spectroscopy), position (e.g., in genome sequences), and others [60]. The terms *data series*, *time series* and *sequence* are used interchangeably.

Proceedings of the VLDB Endowment, Vol. 12, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3282495.3282498

the whole (not a sub-) sequence. This problem represents a common use case across many domains [1, 2, 38, 29]. This work is the most extensive experimental comparison of the efficiency of similarity search methods ever conducted.

Contributions. We make the following contributions:

1. We present a thorough discussion of the data series similarity search problem, formally defining its different variations that have been studied in the literature under diverse and conflicting names. Thus, establishing a common language that will facilitate further work in this area.

2. We include a brief survey of data series similarity search approaches, bringing together studies presented in different communities that have been treated in isolation from each other. These approaches range from smart serial scan methods to the use of indexing, and are based on a variety of classic and specialized data summarization techniques.

3. We make sure that all approaches are evaluated under the same conditions, so as to guard against implementation bias. To this effect, we used implementations in C/C++ for all approaches, and reimplemented in C the ones that were only available in other programming languages. Moreover, we conducted a careful inspection of the code bases, and applied to all of them the same set of optimizations (e.g., with respect to memory management, Euclidean distance calculation, etc.), leading to considerably faster performance.

4. We conduct the first comprehensive experimental evaluation of the efficiency of data series similarity search approaches, using several synthetic and 4 real datasets from diverse domains. In addition, we report the first large scale experiments with carefully crafted query workloads that include queries of varying difficulty, which can effectively stress-test all the approaches. Our results reveal characteristics that have not been reported in the literature, and lead to a deep understanding of the different approaches and their performance. Based on those, we provide recommendations for the best approach to use under typical use cases, and identify promising future research directions.

5. We make available online all source codes, datasets, and query workloads used in our study [28]. This will render our work reproducible and further help the community to agree on and establish a much needed data series similarity search benchmark [44, 91, 90].

2. DEFINITIONS AND TERMINOLOGY

Similarity search represents a common problem in various areas of computer science. However, in the particular case of data series, there exist several different flavors that have been studied in the literature, often times using overloaded and conflicting terms. This has contributed to an overall confusion, which hinders further advances in the field.

In this section, we discuss the different flavors of data series similarity search, and provide corresponding definitions, which set a common language for the problems in this area.

On Sequences. A ***data series*** $S(p_1, p_2, ..., p_n)$ is an ordered sequence of points, p_i, $1 \leq i \leq n$. The number of points, $|S| = n$, is the length of the series. We denote the i-th point in S by $S[i]$; then $S[i:j]$ denotes the ***subsequence*** $S(p_i, p_{i+1}, ..., p_{j-1}, p_j)$, where $1 \leq i \leq j \leq n$. We use $\mathbb{S}$ to represent all the series in a collection (dataset).

In the above definition, if each point in the series represents the value of a single variable (e.g., temperature) then each point is a scalar, and we talk about a ***univariate series***. Otherwise, if each point represents the values of multiple variables (e.g., temperature, humidity, pressure, etc.) then each point is a vector, and we talk about a ***multivariate series***. The values of a data series may also encode measurement errors, or imprecisions, in which case we talk about uncertain data series [7, 88, 70, 24, 25].

Especially in the context of similarity search, a data series of length n can also be represented as a single point in an n-dimensional space. Then the values and length of S are referred to as *dimensions* and *dimensionality*, respectively.

On Distance Measures. A data series ***distance*** is a function that measures the (dis)similarity of two data series. The distance between a query series, S_Q, and a candidate series, S_C, is denoted by $d(S_Q, S_C)$.

Even though several distance measures have been proposed in the literature [11, 26, 6, 19, 80, 57], the Euclidean distance is the one that is the most widely used, as well as one of the most effective for large data series collections [27]. We note that an additional advantage of Euclidean distance is that in the case of Z-normalized series (mean=0, stddev=1), which are very often used in practice [91], it can be exploited to compute Pearson correlation [63].

In addition to the distance used to compare data series in the high-dimensional space, some similarity search methods also rely on *lower-bounding* [14, 89, 71, 81, 85, 52, 22, 42] and *upper-bounding* distances [81, 42]. A ***lower-bounding distance*** is a distance defined in the reduced dimensional space satisfying the lower-bounding property, i.e., the distance between two series in the reduced space is guaranteed to be smaller than or equal to the distance between the series in the original space [30]. Inversely, an ***upper-bounding distance*** ensures that distances in the reduced space are larger than the distances in the original space [81, 42].

On Similarity Search Queries. We now define the different forms of data series similarity search queries. We assume a data series collection, $\mathbb{S}$, a query series, S_Q, and a distance function $d(\cdot, \cdot)$.

A ***k-Nearest-Neighbor (k-NN) query*** identifies the k series in the collection with the smallest distances to the query series.

Definition 1. *Given an integer k, a* ***k-NN query*** *retrieves the set of series $\mathbb{A} = \{\{S_{C_1}, ..., S_{C_k}\} \subseteq \mathbb{S} | \forall\ S_C \in \mathbb{A}$ and $\forall\ S_{C'} \notin \mathbb{A},\ d(S_Q, S_C) \leq d(S_Q, S_{C'})\}$.*

An ***r-range query*** identifies all the series in the collection within range r form the query series.

Definition 2. *Given a distance r, an* ***r-range query*** *retrieves the set of series $\mathbb{A} = \{S_C \in \mathbb{S} | d(S_Q, S_C) \leq r\}$.*

We additionally identify the following two categories of k-NN and range queries. In ***whole matching (WM)*** queries, we compute the similarity between an entire query series and an entire candidate series. All the series involved in the similarity search have to have the same length. In ***subsequence matching (SM)*** queries, we compute the similarity between an entire query series and all subsequences of a candidate series. In this case, candidate series can have different lengths, but should be longer than the query series.

Definition 3. *A* ***whole matching query*** *finds the candidate data series $S \in \mathbb{S}$ that matches S_Q, where $|S| = |S_Q|$.*

Definition 4. *A* ***subsequence matching query*** *finds the subsequence $S[i:j]$ of a candidate data series $S \in \mathbb{S}$ that matches S_Q, where $|S[i:j]| = |S_Q| < |S|$.*

In practice, we encounter situations that cover the entire spectrum: WM queries on large collections of short series [29, 2], SM queries on large collections of short series [1], and SM queries on collections of long series [32].

Note that SM queries can be converted to WM: create a new collection that comprises all overlapping subsequences (each long series in the candidate set is chopped into overlapping subsequences of the length of the query), and perform a WM query against these subsequences [52, 51].

On Similarity Search Methods. When a similarity search algorithm (k-NN or range) produces answers that are (by definition) always correct and complete: we call such an algorithm ***exact***. Nevertheless, we can also develop algorithms without such strong guarantees: we call such algorithms ***approximate***. As we discuss below, there exist different flavors of approximate similarity search algorithms.

An **ϵ-approximate** algorithm guarantees that its distance results have a relative error no more than ϵ, i.e., the approximate distance is at most $(1+\epsilon)$ times the exact one.

Definition 5. *Given a query S_Q, and $\epsilon \geq 0$, an **ϵ-approximate** algorithm guarantees that all results, S_C, are at a distance $d(S_Q, S_C) \leq (1+\epsilon)d(S_Q, [k\text{-}th\ NN\ of\ S_Q])$ in the case of a k-NN query, and distance $d(S_Q, S_C) \leq (1+\epsilon)r$ in the case of an r-range query.*

A **δ-ϵ-approximate** algorithm, guarantees that its distance results will have a relative error no more than ϵ (i.e., the approximate distance is at most $(1+\epsilon)$ times the exact distance), with a probability of at least δ.

Definition 6. *Given a query S_Q, $\epsilon \geq 0$, and $\delta \in [0,1]$, a **δ-ϵ-approximate** algorithm produces results, S_C, for which $Pr[d(S_Q, S_C) \leq (1+\epsilon)d(S_Q, [k\text{-}th\ NN\ of\ S_Q])] \geq \delta$ in the case of a k-NN query, and $Pr[d(S_Q, S_C) \leq (1+\epsilon)r] \geq \delta$) in the case of an r-range query.*

An ***ng-approximate*** (no-guarantees approximate) algorithm does not provide any guarantees (deterministic, or probabilistic) on the error bounds of its distance results.

Definition 7. *Given a query S_Q, an **ng-approximate** algorithm produces results, S_C, that are at a distance $d(S_Q, S_C) \leq (1+\theta)d(S_Q, [k\text{-}th\ NN\ of\ S_Q])$ in the case of a k-NN query, and distance $d(S_Q, S_C) \leq (1+\theta)r$ in the case of an r-range query, for an arbitrary value $\theta \in \mathbb{R}_{>0}$.*

In the data series literature, *ng-approximate* algorithms have been referred to as *approximate*, or *heuristic* search [14, 89, 71, 81, 85, 52]. Unless otherwise specified, for the rest of this paper we will refer to *ng-approximate* algorithms simply as approximate. Approximate matching in the data series literature consists of pruning the search space, by traversing one path of an index structure representing the data, visiting at most one leaf, to get a baseline best-so-far (bsf) match.

Observe that when $\delta = 1$, a δ-ϵ-approximate method becomes ϵ-approximate, and when $\epsilon = 0$, an ϵ-approximate method becomes exact [21]. It it also possible that the same approach implements both approximate and exact algorithms [73, 81, 14, 89, 71]. Methods that provide exact answers with probabilistic guarantees are considered δ-0-approximate. These methods guarantee distance results to be exact with probability at least δ ($0 \leq \delta \leq 1$ and $\epsilon = 0$). (We note that in the case of k-NN queries, Def. 5 corresponds to the *approximately correct NN* [21] and $(1+\epsilon)$-*approximate NN* [5], while Def. 6 corresponds to the *probably approximately correct NN* [21].)

Scope. In this paper, we focus on *univariate* series with *no uncertainty*, and we examine *exact* methods for *whole matching* in collections with a *very large number of series*, using *k-NN queries* and the *Euclidean distance*. This is a very popular problem that lies at the core of several other algorithms, and is important for many applications in various domains in the real world [82, 91, 60], ranging from fMRI clustering [35] to mining earthquake [40], energy consumption [48], and retail data [49]. Note also that some of the insights gained by this study could carry over to other settings, such as, r-range queries, dynamic time warping distance, or approximate search.

3. SIMILARITY SEARCH PRIMER

Similarity search methods can be classified into sequential, and indexing methods. Sequential methods proceed in one step to answer a similarity search query. Each candidate is read sequentially from the raw data file and compared to the query. Particular optimizations can be applied to limit the number of these comparisons [66]. Some sequential methods work with the raw data in its original high-dimensional representation [66], while others perform transformations on the raw data before comparing them to the query [58].

On the other hand, answering a similarity query using an index involves two steps: a filtering step where the pre-built index is used to prune candidates and a refinement step where the surviving candidates are compared to the query in the original high dimensional space [36, 83, 31, 14, 89, 71, 81, 10, 85, 52, 51]. Some indexing methods first summarize the original data and then index these summarizations [10, 71, 83, 31], while others interwine data reduction and indexing [14, 89, 81]. Some methods index high dimensional data directly [22]. We note that all indexing methods depend on lower-bounding, since it allows indexes to prune the search space with the guarantee of no false dismissals [30] (the DSTree index [81] also supports an upper-bounding distance, but does not use it for similarity search). Metric indexes (such as the M-tree [22]) additionally require the distance measure triangle inequality to hold. Though, there exist (non-metric) indexes for data series that are based on distance measures that are not metrics [46].

There also exist hybrid approaches that fall in-between indexing and sequential methods. In particular, multi-step approaches, where data are transformed and re-organized in levels. Pruning then occurs at multiple intermediate filtering steps as levels are sequentially read one at a time.

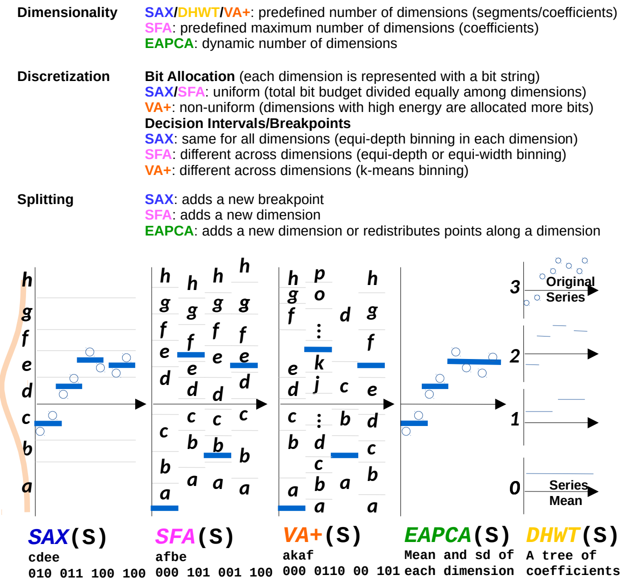
Stepwise is such a method [42], relying on Euclidean distance, and lower- and upper-bounding distances.

3.1 Summarization Techniques

We now briefly outline the summarization techniques used by the methods that we examine in this study.

The *Discrete Haar Wavelet Transform* (DHWT) [16] uses the Haar wavelet decomposition to transform each data series S into a multi-level hierarchical structure. Resulting summarizations are composed of the first l coefficients.

The *Discrete Fourier Transform* (DFT) [3, 30, 64, 65] decomposes S into frequency coefficients. A subset of l coefficients constitutes the summary of S. In our experiments, we use the Fast Fourier Transform (FFT) algorithm, which is optimal for whole matching scenarios (the MFT algorithm [4] is faster than FFT for computing DFT on sliding windows, thus beneficial for subsequence matching queries).



Dimensionality	SAX/DHWT/VA+: predefined number of dimensions (segments/coefficients) SFA: predefined maximum number of dimensions (coefficients) EAPCA: dynamic number of dimensions
Discretization	**Bit Allocation** (each dimension is represented with a bit string) SAX/SFA: uniform (total bit budget divided equally among dimensions) VA+: non-uniform (dimensions with high energy are allocated more bits) **Decision Intervals/Breakpoints** SAX: same for all dimensions (equi-depth binning in each dimension) SFA: different across dimensions (equi-depth or equi-width binning) VA+: different across dimensions (k-means binning)
Splitting	SAX: adds a new breakpoint SFA: adds a new dimension EAPCA: adds a new dimension or redistributes points along a dimension

Figure 1: Summarizations

The *Piecewise Aggregate Approximation* (PAA) [43] and *Adaptive Piecewise Constant Approximation* (APCA) [15] methods are segmentation techniques that divide S into l (equi-length and varying-length, respectively) segments. Each segment represents the mean value of the corresponding points. The *Extended Adaptive Piecewise Approximation* (EAPCA) [81] technique extends APCA by using more information to represent each segment. In addition to the mean, it also stores the standard deviation of the segment. With the *Symbolic Aggregate Approximation* (SAX) [50], S is first transformed using PAA into l real values, and then a discretization technique is applied to map PAA values to discrete set of symbols (alphabet) that can be succinctly represented in binary form. A SAX representation consists of l such symbols. An iSAX (indexable SAX) [74] representation can have an arbitrary alphabet size for each segment.

Similarly to SAX, the *Symbolic Fourier Approximation* (SFA) [71] is also a symbolic approach. However, instead of PAA, it first transforms S into l DFT coefficients using FFT (or MFT for subsequence matching), then extends the discretization principle of SAX to support both equi-depth and equi-width binning, and to allow each dimension to have its own breakpoints. An SFA summary consists of l symbols.

Using the VA+file method [31], S of length n is first transformed using the Karhunen–Loève transform (KLT) into n real values, which are then quantized to discrete symbols. As we will detail later, we modified the VA+file to use DFT instead of KLT, for efficiency reasons.

Figure 1 presents a high-level overview of the summarization techniques presented above.

3.2 Similarity Search Methods

In this study, we focus on algorithms that can produce exact results, and evaluate the ten methods outlined below (in chronological order). The properties of these algorithms are also summarized in Table 1.

We also point out that there exist several techniques dedicated to approximate similarity search [34, 23, 77, 33, 55, 86]. A thorough evaluation of all approximate methods deserves a study on its own, and we defer it to future work.

R*-tree. The R*-tree [10] is a height-balanced spatial access method that partitions the data space into a hierarchy of nested overlapping rectangles. Each leaf can contain either the raw data objects or pointers to those, along with the enclosing rectangle. Each intermediate node contains the minimum bounding rectangle that encompasses the rectangles of its children. Given a query S_Q, the R*-tree query answering algorithm visits all nodes whose rectangle intersects S_Q, starting from the root. Once a leaf is reached, all its data entries are returned. We tried multiple implementations of the R*-tree, and opted for the fastest [37]. We modified this code by adding support for PAA summaries.

M-tree. The M-tree [22] is a multidimensional, metric-space access method that uses hyper-spheres to divide the data entries according to their relative distances. The leaves store data objects, and the internal nodes store routing objects; both store distances from each object to its parent. During query answering, the M-tree uses these distances to prune the search space. The triangle inequality that holds for metric distance functions guarantees correctness. Apart from exact queries, it also supports ϵ-approximate and δ-ϵ-approximate queries. We experimented with four different code bases: two implementations that support bulk-loading [20, 25], the disk-aware mvptree [12], and a memory-resident implementation [25]. We report the results with the latter, because (despite our laborious efforts) it was the only one that scaled to datasets larger than 1GB. We modified it to use the same sampling technique as the original implementation [20], which chooses the number of initial samples based on the leaf size, minimum utilization, and dataset size.

VA+file. The VA+file [31] is an improvement of the VA-file method [83]. While both methods create a filter file containing quantization-based approximations of the high dimensional data, and share the same exact search algorithm, the VA+file does not assume that neighboring points (dimensions) in the sequence are uncorrelated. It thus improves the accuracy of the approximations by allocating bits per dimension in a non-uniform fashion, and partitioning each dimension using a k-means (instead of an equi-depth approach). We improved the efficiency of the original VA+file significantly by implementing it in C and modifying it to use DFT instead of KLT, since DFT is a very good approximation for KLT [31] and is much more efficient [54].

Stepwise. The Stepwise method [42] differentiates itself from indexing methods by storing DHWT summarizations vertically across multiple levels. This process happens in a pre-processing step. When a query S_Q arrives, the algorithm converts it to DWHT, and computes the distance between S_Q and the DHWT of each candidate data series one level at a time, using lower and upper bounding distances it filters out non-promising candidates. When leaves are reached, the final refinement step consists of calculating the Euclidean distance between the raw representations of S_Q and the candidate series. We modified the original implementation to load the pre-computed sums in memory and answer one query at a time (instead of the batch query answering of the original implementation). We also slightly improved memory management to address swapping issues that occurred with the out-of-memory datasets.

SFA trie. The SFA approach [71] first summarizes the series using SFA of length 1 and builds a trie with a fanout

Table 1: Similarity search methods

		Matching Accuracy				Matching Type		Representation		Implementation	
		exact	ng-appr.	ϵ-appr.	δ-ϵ-appr.	Whole	Subseq.	Raw	Reduced	Original	New
Indexes	ADS+	[89]	[89]			✓			iSAX	C	
	DSTree	[81]	[81]			✓			EAPCA	Java	C
	iSAX2+	[14]	[14]			✓			iSAX	C#	C
	M-tree	[22]		[21]	[21]	✓		✓		C++	
	R*-tree	[10]				✓			PAA	C++	
	SFA trie	[71]	[71]			✓	✓		SFA	Java	C
	VA+file	[31]				✓			DFT	MATLAB	C
Other	UCR Suite	[66]					✓	✓		C	
	MASS	[87]					✓		DFT	C	
	Stepwise	[42]				✓			DHWT	C	

equal to the alphabet size on top of them. As leaves reach their capacity and split, the length of the SFA word for each series in the leaf is increased by one, and the series are redistributed among the new nodes. The maximum resolution is the number of DFT coefficients given as a parameter. SFA implements lower-bounding to prune the search space, as well as a bulk-loading algorithm. We re-implemented SFA in C, optimized its memory management, and improved the sampling and buffering schemes. This resulted in a significantly faster implementation than the original one in Java.

UCR Suite. The UCR Suite [66] is an optimized sequential scan algorithm for exact subsequence matching. We adapted the original algorithm to support exact whole matching.

DSTree. The DSTree [81] approach uses the EAPCA summarization technique, which allows, during node splitting, the resolution of a summarization to increase along two dimensions: vertically and horizontally. (Instead, SAX-based indexes allow horizontal splitting by adding a breakpoint to the y-axis, and SFA allows vertical splitting by adding a new DFT coefficient.) In addition to a lower bounding distance, the DSTree also supports an upper bounding distance. It uses both distances to determine the optimal splitting policy for each node. We reimplemented the DSTree algorithm in C and we optimized its buffering and memory management, improving the performance of the algorithm by a factor of 4, compared to the original implementation (in Java).

iSAX2+. The iSAX family of indexes has undergone several improvements. The iSAX 2.0 index [13] improved the splitting policy and added bulk-loading support to the original iSAX index [73]. iSAX2+ [14] further optimized bulk-loading. In the literature, competing approaches have either compared to iSAX, or iSAX 2.0. This is the first time that iSAX2+ is compared to other exact data series indexes. The index supports ng-approximate and exact query answering. We reimplemented the original iSAX2+ algorithm from scratch using C, and optimized its memory management, leading to significant performance improvements.

ADS+. ADS+ [89] is the first query adaptive data series index. It first builds an index tree structure using only the iSAX summarizations of the raw data, and then adaptively constructs the leaves and incorporates the raw data during query answering. For exact query answering, the SIMS algorithm is proposed. It first performs a fast ng-approximate search in the tree in order to acquire an initial best-so-far (bsf) distance, then prunes the search space by using the bsf and the lower bounds between the query and all iSAX summaries. Using that, it performs a skip-sequential search on the raw data that were not pruned. In all our experiments involving ADS+ we use the SIMS algorithm for exact similarity search. ADS-FULL is a non-adaptive version of ADS, that builds a full index using a double pass on the data.

MASS. MASS [87] is an exact subsequence matching algorithm, which computes the distance between a query, S_Q, and every subsequence in the series, using the dot product of the DFT transforms of the series and the reverse of S_Q. We adapted it to perform exact whole matching queries.

4. EXPERIMENTAL EVALUATION

In order to provide an unbiased evaluation, we re-implemented in C all methods whose original language was other than C/C++. Our new implementations are more efficient (in space and time) than the original ones on all datasets we tested. All methods use single precision values, and the methods based on fixed summarizations use 16 segments/coefficients. The same set of known optimizations for data series processing are applied to all methods. All results, source codes, datasets and plots are available in [28].

4.1 Environment

All methods were compiled with GCC 6.2.0 under Ubuntu Linux 16.04.2 with level 2 optimization. Experiments were run on two different machines. The first machine, called *HDD*, is a server with two Intel Xeon E5-2650 v4 2.2GHz CPUs, 75GB[2] of RAM, and 10.8TB (6 x 1.8TB) 10K RPM SAS hard drives in RAID0. The throughput of the RAID0 array is 1290 MB/sec. The second machine, called *SSD*, is a server with two Intel Xeon E5-2650 v4 2.2Ghz CPUs, 75GB of RAM, and 3.2TB (2 x 1.6TB) SATA2 SSD in RAID0. The throughput of the RAID0 array is 330 MB/sec. All our algorithms are single-core implementations.

4.2 Experimental Setup

Scope. This work concentrates on exact whole-matching (WM) 1-NN queries. Extending our experimental framework to cover r-range queries, subsequence matching and approximate query answering is part of our future work.

Algorithms. This experimental study covers the ten methods described in Section 3, which all have native-support for Euclidean distance. Our baseline is the Euclidean distance version of the UCR Suite [66]. This is a set of techniques for performing very fast similarity computation scans. These optimizations include: a) avoiding the computation of square root on Euclidean distance, b) early abandoning

[2] We used GRUB to limit the amount of RAM, so that all methods are forced to use the disk. Note that GRUB prevents the operating system from using the rest of the RAM as a file cache, which is what we wanted for our experiments.

of Euclidean distance calculations, and c) reordering early abandoning on normalized data[3]. We used these optimizations on all the methods that we examined.

Datasets. Experiments were conducted using both synthetic and real datasets. Synthetic data series were generated as random-walks (i.e., cumulative sums) of steps that follow a Gaussian distribution (0,1). This type of data has been extensively used in the past [30, 14, 91], and it is claimed to model the distribution of stock market prices [30].

Our four real datasets come from the domains of seismology, astronomy, neuroscience and image processing. The seismic dataset, *Seismic*, was obtained from the IRIS Seismic Data Access archive [32]. It contains seismic instrument recording from thousands of stations worldwide and consists of 100 million data series of size 256. The astronomy dataset, *Astro*, represents celestial objects and was obtained from [76]. The dataset consists of 100 million data series of size 256. The neuroscience dataset, *SALD*, obtained from [78] represents MRI data, including 200 million data series of size 128. The image processing dataset, *Deep1B*, retrieved from [79], contains 267 million Deep1B vectors of size 96 extracted from the last layers of a convolutional neural network. All of our real datasets are of size 100 GB. In the rest of the paper, the size of each dataset is given in GB instead of the number of data series. Overall, in our experiments, we use datasets of sizes between 25-1000GB.

Queries. All our query workloads, unless otherwise stated, include 100 query series. For synthetic datasets, we use two types of workloads: *Synth-Rand* queries are produced using the same random-walk generator (with a different seed[4]), while *Synth-Ctrl* queries are created by extracting data series from the input data set and adding progressively larger amounts of noise, in order to control the difficulty of each query (more difficult queries tend to be less similar to their nearest neighbor [90]). For the real datasets, query workloads are also generated by adding progressively larger amounts of noise to data series extracted from the raw data, and we name them with the suffix *-Ctrl*. For the Deep1B dataset, we additionally include a real workload that came with the original dataset; we refer to it as *Deep-Orig*.

Scenarios. The experimental framework consists of three scenarios: parametrization, evaluation and comparison. In parametrization (§4.3.1), the optimal parameters for each method are identified. In evaluation (§4.3.2), the scalability and search efficiency for each method is evaluated under varying dataset sizes and data series lengths. Finally, in comparison (§4.3.3), methods are compared together according to the following criteria: a) scalability and search efficiency on more complex query workloads and more varied and larger datasets, b) memory and disk footprint, c) pruning ratio, and d) tightness of the lower bound.

Measures The measures we use are the following.

1. For scalability and search efficiency, we use two measures: *wall clock time* and the *number of random disk accesses*. *Wall clock time* is used to measure input, output and total elapsed times. Then CPU time is calculated as the difference between the total time and I/O time. The *number of random disk accesses* is measured for indexes. One random disk access corresponds to one leaf access for all indexes, except for the skip-sequential access method ADS+, for which one random disk access corresponds to one skip. As will be evident in the results, our measure of random disk accesses provides a good insight into the actual performance of indexes, even though we do not account for details such as caching, the number of disk pages occupied by a leaf and the numbers of leaves in contiguous disk blocks.
2. For footprint, the measures used are: *total number of nodes*, *number of leaf nodes*, *memory size*, *disk size*, *leaf nodes fill factor* and *leaf depth*.
3. We also consider the pruning ratio P, which has been widely used in the data series literature [43, 71, 81, 27, 42] as an implementation-independent measure to compare the effectiveness of an index. It is defined as follows:

$$P \;=\; 1 - \frac{\#\ of\ Raw\ Data\ Series\ Examined}{\#\ of\ Data\ Series\ In\ Dataset}$$

Pruning ratio is a good indicator of the number of sequential I/Os incurred. However, since relevant data series are usually spread out on disk, it should be considered along with the number of random disk accesses (seeks) performed.

4. The *tightness of the lower bound*, TLB has been used in the literature as an implementation independent measure in various different forms [73, 71, 80]. In this work we use the following version of the TLB measure that better captures the performance of indexes:

$$TLB \;=\; \frac{Lower\ Bounding\ Distance(Q\prime, N)}{Average\ True\ Distance(Q, N)}$$

Where Q is the query, $Q\prime$ is the representation of Q using the segmentation of a given leaf node N, and the average true distance between the query Q and node N is the average Euclidean distance between Q and all data series in N. We report the average over all leaf nodes for all 100 queries.

Procedure. Unless otherwise stated, experiments refer to answering 100 exact queries. Experiments with query workloads of 10,000 queries report extrapolated values. The extrapolation consists of discarding the best and worst five queries (of the original 100) in terms of total execution time, and multiplying the average of the 90 remaining queries by 10,000. Experiments involving an indexing method include a first step of building the index (or re-organizing the data as in the case of Stepwise). Caches are fully cleared before each experiment. During each experiment, the caches are warm, i.e., not cleared between indexing/preprocessing and query answering, nor after each query.

4.3 Results

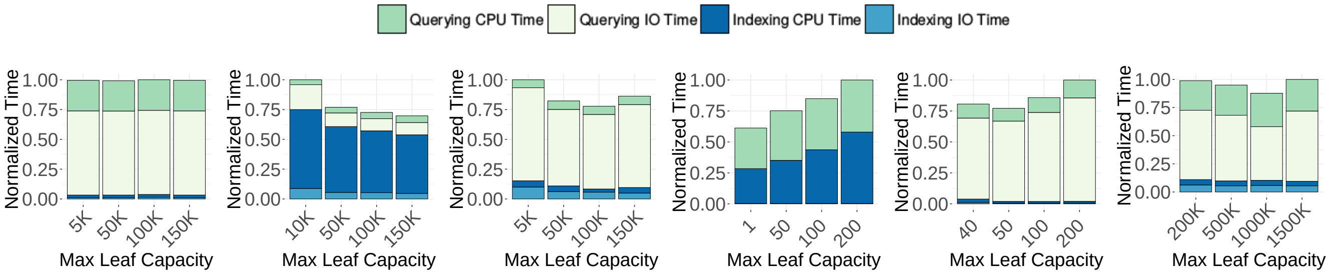
4.3.1 Parametrization

We start our experimentation by fine tuning each method. Methods that do not support parameters are ran with their default values. The methods that support parameters are ADS+, DSTree, iSAX2+, M-tree, R*-tree and SFA trie. We use a synthetic dataset of 100GB with data series of length 256. The only exceptions are M-tree and R*-tree, where we parametrize using 50GB, since experiments with 100GB, or above, take more than 24 hours to complete.

The most critical parameter for these methods is the leaf threshold, i.e., the maximum number of data series that an index leaf can hold. We thus vary the leaf size and study the tradeoffs of index construction and query answering for each method. Figure 2 reports indexing and querying execution times for each method, normalized by the largest total

[3] Early abandoning of Z-normalization is not used since all datasets were normalized in advance.

[4] All seeds can be found in [28].



(a) ADS+ Dataset = 100GB **(b) DSTree Dataset = 100GB** **(c) iSAX2+ Dataset = 100GB** **(d) M-tree Dataset = 50GB** **(e) R*-tree Dataset = 50GB** **(f) SFA trie Dataset = 100GB**

Figure 2: Leaf size parametrization

cost. The ratio is broken down into CPU and I/O times. Figure 2a shows that the performance of ADS+ is the same across leaf sizes. The leaf size affects indexing time, but not query answering. This is not visible in the figure, because index construction time is minimal compared to query answering time. This behavior is expected, since ADS+ is an adaptive index, which during querying splits the nodes until a minimal leaf size is reached. For M-tree, larger leaves cause both indexing and querying times to deteriorate. For all other methods, increasing the leaf size improves indexing time (because trees are smaller) and querying time (because several series are read together), but once the leaf size goes beyond the optimal leaf size, querying slows down (because some series are unnecessarily read and processed). For DSTree, the experiments execution logs indicate that querying is faster with the 100K leaf size. The optimal leaf size for iSAX2+ is also 100K, for SFA is 1M, and for M-tree and R*-tree are 1 and 50, respectively.

SFA takes two other parameters: the alphabet size and the binning method. We ran experiments with both equi-depth and equi-width binning, and alphabet sizes from 8 (default value), to 256 (default alphabet size of iSAX2+ and ADS+). Alphabet size 8 and equi-depth binning provided the best performance and were thus used for subsequent experiments.

Some of the evaluated methods also use internal buffers to manage raw data that do not fit in memory during index building and query processing. We ran experiments varying these buffer sizes from 5GB to 60GB. The maximum was set to 60GB (recall that total RAM was 75GB). All methods benefit from a larger buffer size except ADS+. This is because a smaller buffer size allows the OS to use extra memory for file caching during query processing, since ADS+ accesses the raw data file directly.

4.3.2 Evaluation of Individual Methods

We now evaluate the indexing and search efficiency of the methods by varying the dataset size. We used two datasets of size 25GB and 50GB that fit in memory and two datasets of size 100GB and 250GB that do not fit in memory (total RAM was 75GB), with the *Synth-Rand* query workload.

ADS+. Figure 3a shows that ADS+ is very efficient at index building, spending most of the cost for query answering, itself dominated by the input time. The reason is that ADS+ performs skip sequential accesses on the raw data file, performing a skip almost every time a data series is pruned.

DSTree. In contrast, DSTree answers queries very fast whereas index building is costly (Figure 3b). DSTree's cost for index building is mostly CPU, thus, offering great opportunities for parallelization.

iSAX2+. Figure 3c summarizes the results for iSAX2+, which is slower to build the index compared to ADS+, but faster compared to DSTree. Query answering is faster than ADS+ and slower than the DSTree.

MASS. Figure 3d reports the results for MASS, which has been designed mainly for subsequence matching queries, but we adapted it for whole matching. The very high CPU cost is due to the large number of operations involved in calculating Fourier transforms and the dot product cost.

M-tree. For the M-tree, we were only able to run experiments with in-memory datasets, because the only implementation we could use is a main memory index. The disk-aware implementations did not scale beyond 1GB. Figure 3e shows the M-tree experimental results for the 25GB and 50GB datasets, and the (optimistic) extrapolated results for the 100GB and 250GB datasets. Note that going from 25GB to 50GB, the M-tree performance deteriorates by a factor of 3, even though both datasets fit in memory. (The M-tree experiments for the 100GB and 250GB datasets were not able to terminate, so we report extrapolated values in the graph, by multiplying the 50GB numbers by 3 and 9, respectively, which is an optimistic estimation.) These results indicate that M-tree cannot scale to large dataset sizes.

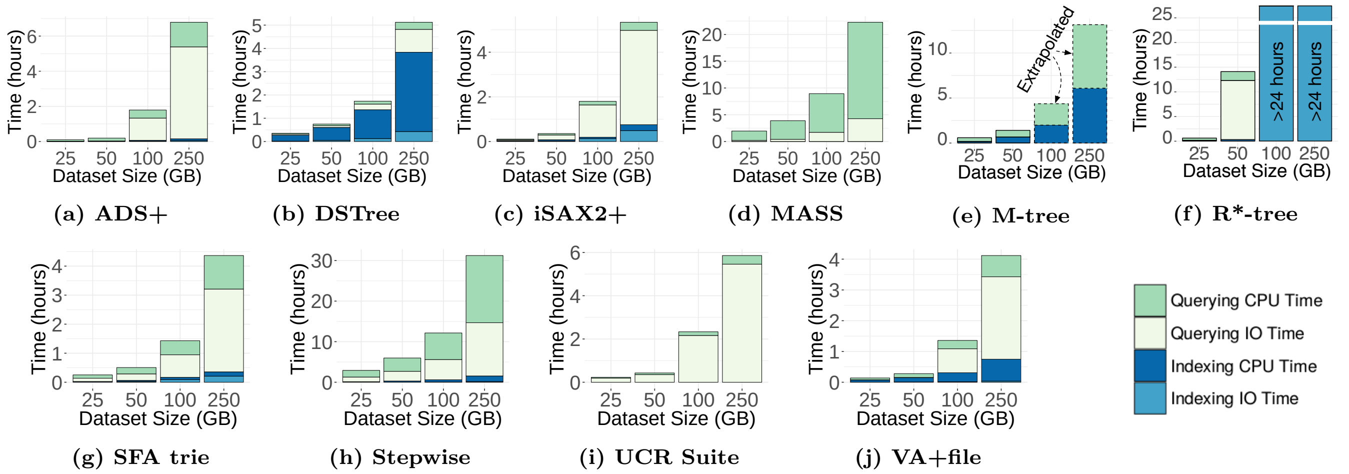
R*-tree. Figure 3f shows the results for the R*-tree. Its performance deteriorates rapidly as dataset sizes increase. Even using the best implementation among the ones we tried, when the dataset reaches half the available memory, swapping causes performance to degrade. Experiments on the 100GB and 250GB datasets were stopped after 24 hours.

SFA Trie. Figure 3g reports the cost of index building and query processing for SFA. We observe that query processing dominates the total cost and that query cost is mostly I/O, due to the optimal leaf size being rather large.

Stepwise. Figure 3h indicates the time it takes for Stepwise to build the DWHT tree and execute the workload. The total cost is high and is dominated by query answering. This is because answering one query entails filtering the data level by level and requires locating the remaining candidate data corresponding to higher resolutions through random I/O.

UCR Suite. Figure 3i shows the time it takes for the UCR-Suite to execute the workload. Its cost is naturally dominated by input time, being a sequential scan algorithm.

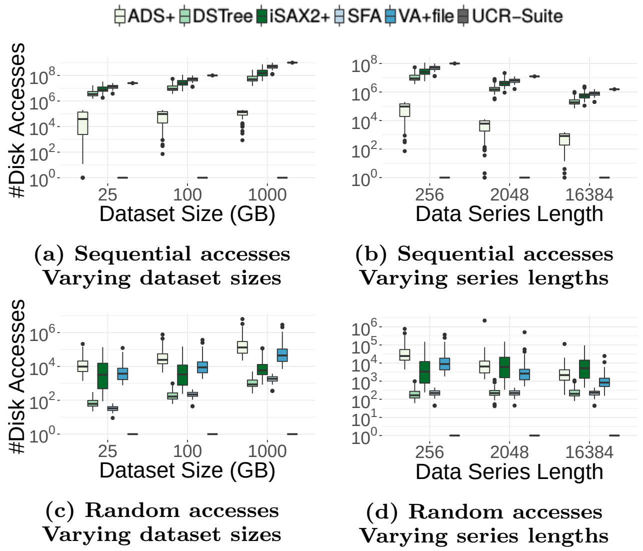
VA+file. We observe in Figure 3j that VA+file is efficient at index building, spending most of the cost for query answering. The indexing and querying costs are dominated by



(a) ADS+ **(b) DSTree** **(c) iSAX2+** **(d) MASS** **(e) M-tree** **(f) R*-tree**

(g) SFA trie **(h) Stepwise** **(i) UCR Suite** **(j) VA+file**

Figure 3: Scalability with increasing dataset sizes



(a) Sequential accesses Varying dataset sizes

(b) Sequential accesses Varying series lengths

(c) Random accesses Varying dataset sizes

(d) Random accesses Varying series lengths

Figure 4: Number of disk accesses

CPU and input time, respectively. The CPU cost is due to the time spent for determining the bit allocation and decision intervals for each dimension; the input time is incurred when accessing the non-pruned raw data series.

Summary. Overall, Figure 3 shows that it takes Stepwise, MASS, the M-tree and the R*-tree over 12 hours to complete the workload for the 250GB dataset, whereas the other methods need less than 7 hours. Therefore, in the subsequent experiments, we will only include ADS+, DSTree, iSAX2+, SFA, the UCR suite and the VA+file.

4.3.3 Comparison of the Best Methods

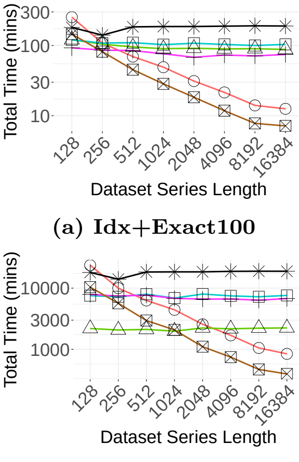
In the following experiments, we use the best methods as identified above, and compare them in more detail.

Disk Accesses vs Dataset Size/Sequence Length. Figure 4 shows the number of sequential and random disk accesses incurred by the 100 exact queries of the *Synth-Rand* workload for increasing dataset sizes and increasing lengths. When the dataset size is varied, the length of the data series is kept constant at 256, whereas the dataset size is kept at 100GB when the length is varied. We can observe that the VA+file and ADS+ perform the smallest number of sequential disk accesses across dataset sizes and data series lengths, with the VA+ performing virtually none. As expected, the UCR-Suite performs the largest number of sequential accesses regardless of the length of the series, or the size of the dataset. This number is also steady across queries, thus its boxplot is represented by a flat line. There is not a significant difference between the number of sequential operations needed by the DSTree, SFA or iSAX2+ (DSTree does the least, and SFA the most). SFA requires more sequential accesses, because its optimal leaf size is 1M, as opposed to 100K for DSTree and iSAX2+.

As far as random I/O for different dataset sizes is concerned, ADS+ performs the largest number of random accesses, followed by the VA+file. The DSTree and SFA incur almost the same number of operations. However, the DSTree has a good balance between the number of random and sequential I/O operations. It is interesting to point out that as the dataset size increases, the number of random operations for iSAX2+ becomes less skewed across queries. This is because of the fixed split-point nature of iSAX2+ that causes it to better distribute data among leaves when the dataset is large: in small dataset sizes, many leaves can contain very few series.

When the dataset size is set to 100GB and the data series length is increased, we can observe a dramatic decrease of the number of random operations incurred by ADS+ and VA+file. The reason is that both methods use a skip-sequential algorithm, so even if the pruning ratio stays the same, when the data series is long, the algorithm skips larger blocks of data, thus the number of skips decreases. The random I/Os across lengths for the other methods remain quite steady, with SFA and DSTree performing the least.

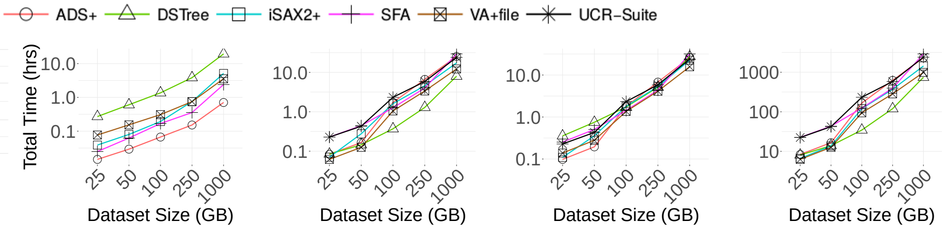
Scalability/Search Efficiency vs Sequence Length. Figure 5 depicts the performance of the different methods with increasing data series lengths. In order to factor out other parameters, we fix the dataset size to 100GB, and the dimensionality of the methods that use summarizations to 16, for all data series lengths. We observe that the indexing and querying costs for ADS+ and VA+file plummet as the



(a) **Idx+Exact100**

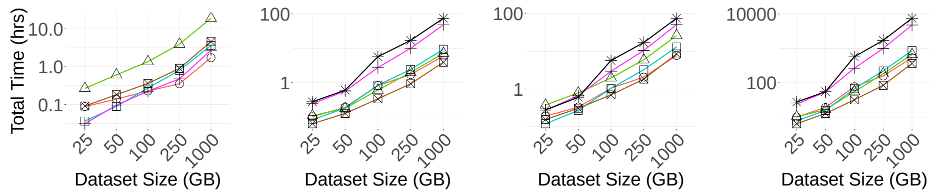
(b) **Idx+Exact10K**

Figure 5: Scalability with increasing lengths



(a) **Idx** (b) **Exact100** (c) **Idx+Exact100** (d) **Idx+Exact10K**

Figure 6: Scalability comparison (HDD)



(a) **Idx** (b) **Exact100** (c) **Idx+Exact100** (d) **Idx+Exact10K**

Figure 7: Scalability comparison (SSD)

data series length increases, whereas the cost of the other methods remains relatively steady across all lengths. This is because with increasing lengths, both algorithms perform larger sequential reads on the raw data file and fewer, contiguous skips. VA+file performs better than ADS+ since it incurs less random and almost no sequential I/Os (Figure 4).

Scalability/Search Efficiency vs Dataset Size - HDD. Figure 6 compares the scalability and search efficiency of the best methods on the HDD platform for the *Synth-Rand* workload on synthetic datasets ranging from 25GB to 1TB. There are 4 scenarios: indexing (Idx), answering 100 exact queries (Exact100), indexing and answering 100 exact queries (Idx+Exact100), and indexing and answering 10,000 queries (Idx+Exact10K). Times are shown in log scale to reveal the performance on smaller datasets.

Figure 6a indicates only the indexing times. ADS+ outperforms all other methods and is an order of magnitude faster than the slowest, DSTree. Figure 6b shows the times for running 100 exact queries. We observe two trends in this plot. For in-memory datasets, VA+file surpasses the other methods. For the larger datasets, the DSTree is a clear winner, followed by VA+file, while the performance of the other methods converge to that of sequential scan. Figure 6c refers to indexing and answering the 100 exact queries. For in-memory datasets, ADS+ shows the best performance, with iSAX2+ performing equally well on the 25GB dataset. However, for larger datasets, VA+file outperforms all other methods.

Figure 6d shows the time for indexing and answering 10K exact queries. The trends now change. For in-memory datasets, iSAX2+ and VA+file outperform all other methods, in particular ADS+. Both iSAX2+ and VA+file are slower than ADS+ in index building, but this high initial cost is amortized over the large query workload.

The DSTree is the best contender for large data sets that do not fit in memory, followed by VA+file and iSAX2+. The other methods perform similar to a sequential scan. The DSTree has the highest indexing cost among these methods, but once the index is built, query answering is very fast, thus being amortized for large query workloads. The strength of the DSTree is based on its sophisticated splitting policy, the upper/lower bounds used in query answering, and its parameter-free summarization algorithm.

Our results for in-memory datasets corroborate earlier studies [89] (i.e., ADS+ outperforms alternative methods), yet, we additionally bring in the picture VA+file, which is very competitive and had not been considered in earlier works. Moreover, for out-of-memory data, our results show that ADS+ is not faster than sequential scan, as was previously reported. The reason for this discrepancy in results lies with the different hardware characteristics, which can significantly affect the performance of different algorithms, both in relative, as well as in absolute terms. More specifically, the disks used in [89] had 60% of the sequential throughput of the disks used in this paper. As a result, ADS+ can be outperformed by a sequential scan of the data when the disk throughput is high and the length of the sequences is small enough, where ADS+ is forced to perform multiple disk seeks. Figures 4a and 4c clearly show that ADS+ performs the smallest number of sequential disk operations and the largest number of random disk operations across all datasets. In main-memory, SSDs, and with batched I/Os, ADS+ is expected to perform significantly better.

Scalability/Search Efficiency vs Dataset Size - SSD. In order to further study the effect of different hardware on the performance of similarity search methods, we repeated the experiments described in the last paragraph on the SSD machine. We once again tuned each index on the 100GB dataset to find the optimal leaf threshold, which this time was an order of magnitude smaller than the optimal leaf size for the HDD platform. However, we were not able to perform experiments with our larger datasets with these smaller leaf sizes, because the maximum number of possible split points was reached before indexing the entire dataset. Although small leaf sizes can improve performance on smaller datasets, they cannot be used in practice, since the index itself cannot be constructed. Therefore, we iteratively increased the leaf sizes, and picked the ones that worked for

Table 2: Controlled workloads experimental results summary (sequential scan algorithm is highlighted)

		Scenarios					
	Dataset	Idx	Exact 100	Idx+ Exact 100	Idx+ Exact 10K	Exact Easy-20	Exact Hard-20
HDD	Small	A	D	S	D	D	D
	Large	A	D	S	D	D	D
	Astro	A	U	U	V	V	U
	Deep1B	A	U	U	U	D	U
	SALD	A	D	I	D	D	D
	Seismic	A	D	S	D	D	U
SSD	Small	S	D	I	D	I	D
	Large	S	D	I	D	I	D
	Astro	I	V	V	V	V	V
	Deep1B	S	I	I	V	I	U
	SALD	S	I	I	I	I	V
	Seismic	A	V	V	V	D	V

A: ADS, **D:** DSTree, **I:** iSAX2+
S: SFA, **U:** UCR-Suite, **V:** VA+file

all datasets in our experiments: these leaf sizes proved to be the same as the ones for the HDD platform. We note that the SFA trie was particularly sensitive to parametrization.

There are two main observations on these results (see Figure 7). The first is that VA+file and ADS+ are now the best performers on most scenarios. The only exceptions are iSAX2+ surpassing ADS+ on the 25GB workload, and iSAX2+/SFA being faster in indexing the in-memory datasets. As discussed earlier, the bottleneck of ADS+ and VA+file is random I/O, so the fast performance of the SSD machine on random I/O explains why they both win over the other methods. ADS+ is faster than VA+file at indexing, while the opposite is true for query answering. The indexing cost of VA+file is amortized in the 10K workload. The second observation is that UCR-Suite performs poorly, due to the low disk throughput of the SSD server.

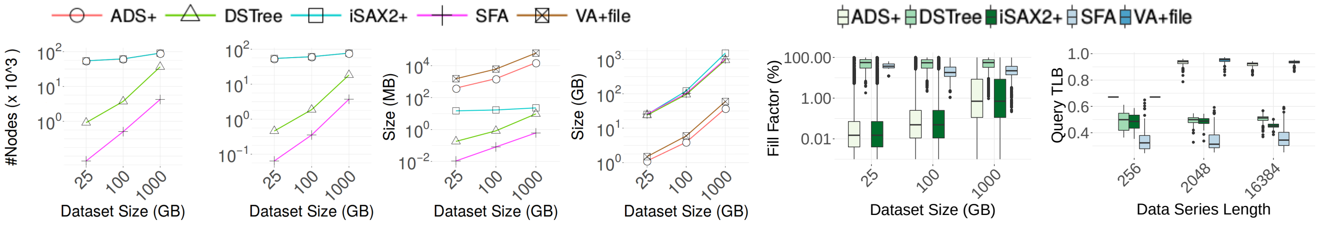
Memory/Disk Footprint vs Dataset Size. In this set of experiments, we compare the disk and memory footprints of all methods. Figure 8a shows that SAX-based indexes have the largest number of nodes. SFA has a very low number of nodes, because the leaf size we use is 1,000,000 (refer to Figure 2), whereas the leaf sizes for DSTree and iSAX2+ are both 100,000. The ADS+ index is indifferent to leaf size so we set its initial value to 100,000. For all methods, most nodes are leaves, as shown in Figure 8b. Note that ADS+ and iSAX2+ have the same tree structure with en equal number of nodes, since the leaf size is the same.

As shown in Figures 8c and 8d, the size of the indexes in memory and on disk follows the same trend as the number of nodes. Although ADS+ and iSAX2+ have the same tree shape, some of the data types and structures they use are not the same, thus the different sizes in memory. For the VA+file, we only report the size of the approximation file on disk, since it does not build an auxiliary tree structure.

We use two measures to compare the overall structure of the indexes. The first is the leaf nodes fill factor, which measures the percentage of the leaf that is full, and gives a good indication of whether the index distributes evenly the data among leaves. The second measure is the depth of the leaves, which can help evaluate how balanced an index is. While none of the best performing index trees is truly height-balanced, some are better balanced in practice than others. Figure 8e shows the leaf nodes fill factor for different dataset sizes and methods. (Note that VA+file is missing, since it has no tree; though, if we consider as leaves the pages, where it stores the data, then the fill factor of these pages is 100%.) We observe that SFA offers the least variability in the fill factor for the small datasets (as indicated by the size of the boxplot), but the median fill factor fluctuates as the data set changes. DSTree provides the highest median fill factor (as indicated by the line in the middle of the boxplot), which also remains steady with increasing data set sizes. DSTree also displays the least skew and virtually no outliers, which means that this index effectively partitions the dataset and distributes the series across all its leaves. The SAX-based indexes have many outliers, with some leaves being full and others being empty. The graph showing the depth of the indexes can be found elsewhere [28].

Tightness of the Lower Bound. Figure 8f shows the TLB (defined in Section 4.2) of each method for increasing data series lengths. We observe that the TLBs of ADS+ and VA+file increase rapidly with increasing lengths, then stabilize when they reach a value close to 1. This explains why the performance of both methods improves with longer series. We also note that VA+file has a slightly tighter lower bound than ADS+, thanks to its non-uniform discretization scheme, which helps explain why VA+file incurs less random I/O than ADS+, and thus performs better. The TLB of the SFA trie is low compared to the other methods, although we used the tight lower bounding distance of SFA (which uses the DFT MBRs). We believe this is due to the optimal alphabet size of 8, which is rather small compared to the default alphabet size of 256 for the SAX-based methods. As for iSAX2+ and DSTree, the main difference in the TLB is that it becomes virtually constant as the length increases.

Pruning Ratio. We measure the pruning ratio (higher is better) for all indexes across datasets and data series lengths. For the *Synth-Rand* workload on synthetic datasets, we varied the size from 25GB to 1TB and the length from 128 to 16384. We observed that the pruning ratio remained stable for each method and that overall ADS+ and VA+file have the best pruning ratio, followed by DSTree, iSAX2+ and SFA. We also ran experiments with a real workload (*Deep-Orig*), a controlled workload on the 100GB synthetic dataset (*Synth-Ctrl*), and controlled workloads on the real datasets (*Astro-Ctrl*, *Deep-Ctrl*, *SALD-Ctrl* and *Seismic-Ctrl*). In the controlled workloads, we extract series from the dataset and add noise. Figure 9 summarizes these results. For lack of space, we only report the pruning ratio for the real datasets (all of size 100GB) and the 100GB synthetic dataset. The pruning ratio for *Synth-Rand* is the highest for all methods. We observe that the *Synth-Ctrl* workload is more varied than *Synth-Rand* since it contains harder queries with lower pruning ratios. The trend remains the same with ADS+ and VA+file having the best pruning ratio overall, followed by DSTree, iSAX2+ then SFA. For real dataset workloads, ADS+ and VA+file achieve the best pruning, followed by iSAX2+, DSTree, and then SFA. The relatively low pruning ratio for the SFA is most probably due to the large leaf size of 1,000,000. Once a leaf is retrieved, SFA accesses all series in the leaf, which reduces the pruning ratio significantly. VA+file has a slightly better pruning ratio than ADS+, because it performs less random and sequential I/O, thanks to its tighter lower bound. We note that the pruning ratio alone does not predict the performance of an index. In fact, this ratio provides a good estimate of the number of sequential operations that a method will perform, but it



(a) Nodes (b) Leaf Nodes (c) Mem. Size (d) Disk Size (e) Leaf fill factor (f) TLB

Figure 8: Exact methods footprint and TLB for synthetic datasets

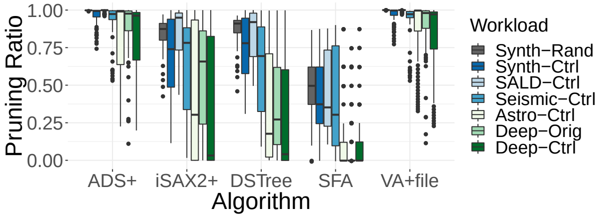


Figure 9: Pruning ratio (Dataset Size= 100GB, Workload = 100 Queries)

should be considered along with other measures like the number of random disk I/Os.

Scalability/Search Efficiency with Real Datasets. In Table 2, we report the name of the best method for each scenario. In addition to the four scenarios discussed earlier, we also consider two new scenarios: the average time of the 20 easiest queries (Easy-20) and the average time of the 20 hardest queries (Hard-20) of the corresponding workload. A query is considered easy, or hard, depending on its pruning ratio (computed as the average across all techniques) [90].

It is important to note that while queries are categorized as easy and hard, easy queries on one dataset may be harder than easy queries on another dataset, as the average pruning ratio for each dataset differs. This is because some datasets can be summarized more effectively than others. We averaged the results over 20 hard queries and 20 easy queries. In-memory datasets are labeled *small* and the others *large*.

We observe that UCR-Suite wins in exact query answering and on hard queries for the Astro/Deep1B scenarios. This is due to the very low pruning ratio for these workloads. DSTree is fast on easy queries and exact query answering on the SALD/Seimic scenarios. ADS+ always wins in indexing on HDD, but is sometimes surpassed by iSAX2+/SFA on SSD. Similarly to synthetic datasets, the methods behave differently on real datasets when experiments are ran on the SSD platform. VA+file and iSAX2+ have a superior performance overall. DSTree also performs well, while UCR-Suite wins only on hard queries on the Deep1B dataset.

5. DISCUSSION

In the data series literature, competing similarity search methods have never been compared together under a unified experimental scheme. The objective of this experimental evaluation is to consolidate previous work on data series whole-matching similarity search and prepare a solid ground for further developments in the field.

We undertook a challenging and laborious task, where we re-implemented from scratch four algorithms: iSAX2+, SFA trie, DSTree, and VA+file, and optimized memory management problems (swapping, and out-of-memory errors) in R*-tree, M-tree, and Stepwise. Choosing C/C++ provided considerable performance gains, but also required low-level memory management optimizations. We believe the effort involved was well worth it since the results of our experimental evaluation emphatically demonstrate the importance of the experimental setup on the relative performance of the various methods. To further facilitate research in the field we publicize our source code and experimental results [28]. This section summarizes the lessons learned in this study.

Unexpected Results. For some of the algorithms our experimental evaluation revealed some unexpected results.

(1) *The Stepwise method performed lower than our expectations.* This was both due to the fact that our baseline sequential scan was fully optimized for early abandoning and computation optimization, but most importantly because of *a different experimental setup.* The original implementation of Stepwise performed batched query answering. In our case we compared all methods on single query at a time workload scenario. This demonstrates the importance of *the experimental setup and workload type.*

(2) *The VA+file method performed extremely well.* Although an older method, VA+file is among the best performers overall. Our optimized implementation, which is much faster than the original version, helped unleash the best of this method; this demonstrates the importance of *the implementation framework.*

(3) *For exact queries on out-of-memory data on the HDD machine, ADS+ is underperforming.* The reason is that ADS+ performs multiple skips while pruning at a per series level and is thus significantly affected by the hard disk's latency. In the original study [89], ADS+ was run on a machine with 60% of the hard disk throughput of the one used in the current work. The HDD setup with the 6 RAID0 disks gave a significant advantage on methods that perform sequential scans on larger blocks of data and less skips. On the SSD machine, however, the trend is reversed, and ADS+ becomes one of the best contenders overall. These observations demonstrate the importance of *the hardware setup.*

(4) *The optimal parameters of most algorithms were different than the ones presented in their respective papers.* This is because some methods were not tuned before: the iSAX2+, DSTree and SFA papers have no tuning experiments. We tuned each for varying leaf and buffer sizes (for brevity, we only report results for leaf parametrization in Figure 2 (for buffer tuning experiments, see [28]). For SFA, we also tuned the sample size used to identify the break-

points, binning method (equi-depth vs. equi-width), and number of symbols for the SFA discretization. Another reason is that we studied in more detail methods that were partially tuned (e.g., ADS+ was tuned only for varying leaf size; we also varied buffer size and found that assigning most of RAM to buffering hurts performance). These findings further demonstrate the need for *careful parameter-tuning.*

(5) *The quality of the summarization, as measured by TLB and pruning, is not necessarily correlated to time performance.* An early experimental study [44] claimed that the tightness of the lower bound can be used alone to evaluate the efficiency of indexing techniques. While summarization quality is an important factor on its own, we demonstrate that it cannot alone predict the time performance of an index, even in the absence of data and implementation biases. For example, ADS+ achieves very high pruning and TLB, yet, in terms of time, it is outperformed by other methods in some scenarios. It is of crucial importance to consider summarization quality alongside the properties of the index structure and the hardware platform.

Speed-up Opportunities. Through our analysis, we identified multiple factors that affect the performance of examined methods. In turn, these factors reveal opportunities and point to directions for performance improvements.

(1) *Stepwise* offers many such avenues. Its storage scheme could be optimized to reduce the number of random I/O during query answering, and its query answering algorithm would benefit a lot from parallelization and modern hardware optimizations (i.e., through multi-core and SIMD parallelism), as 50%-98% of total time is CPU.

(2) *DSTree* is very fast at query answering, but rather slow at index building. Nevertheless, a large percentage of this time (85-90%) is CPU. Therefore, also the indexing performance of DSTree can be improved by exploiting modern hardware. Moreover, bulk loading during indexing, and buffering during querying, would also make it even faster.

(3) A similar observation holds for *VA+file MASS.* Even though MASS is not designed for whole-matching data series similarity search, its performance can be significantly enhanced with parallelism and modern hardware exploitation, since 90% of its execution time is CPU cost. Similarly, the indexing cost of VA+file can be further improved.

(4) Finally, we obtained a better understanding of the *ADS+* algorithm. Apart from being very fast in index building, our results showed that it also has a leading performance for whole-matching similarity search for *long* data series. We also discovered that the main bottleneck for ADS+ are the multiple skips performed during query answering. Its effects could be masked by controlling the size of the data segments skipped (i.e., skipping/reading large continuous blocks), and through asynchronous I/O. Moreover, because of its very good pruning power (that leads to an increased number of skips), we expect ADS+ to work well whenever random access is cheap, e.g., with SSDs and main-memory systems.

Data-adaptive Partitioning. While the SFA trie and iSAX-based index building algorithms are much faster than the DSTree index building algorithm, their performance during query answering is much worse than that of DSTree. DSTree spends more time during indexing, intelligently adapting its leaf summarizations when split operations are performed. This leads to better data clustering and as a result faster query execution. On the contrary, both iSAX and SFA have fixed maximum resolutions, and iSAX indexes can only perform splits on predefined split-points. Even though iSAX summarizations at full resolution offer excellent pruning power (see ADS+ in Figure 9), grouping them using fixed split-points in an iSAX-based index does not allow for effective clustering (see Figure 8e). This is both an advantage (indexing is extremely fast), but also a drawback as it does not allow clustering to adapt to the dataset distribution.

Access-Path Selection. Finally, our results demonstrate that the pruning ratio, along with the ability of an index to cluster together similar data series in large contiguous blocks of data, is crucial for its performance. Moreover, our results confirm the intuitive result that the smaller the pruning ratio, the higher the probability that a sequential scan will perform better than an index, as can be observed for the hard queries in Table 2. This is because it will avoid costly random accesses patterns on a large part of the dataset. However, the decision between a scan or an index, and more specifically, the choice of an index, is not trivial, but is based on a combination of factors: (a) the effectiveness of the summarization used by the index (which can be estimated by the pruning ratio); (b) the ability of the index to cluster together similar data series (which determines the access pattern); and (c) the hardware characteristics (which dictate the data access latencies). This context gives rise to interesting optimization problems, which have never before been studied in the domain of data series similarity search.

Recommendations. Figure 10 presents a decision matrix that reports the best approach to use for problems with different data series characteristics, given a specific hardware setup (i.e., HDD) and query workload (i.e., Indexing + 10K synthetic queries). In general though, choosing the best approach to answer a similarity query on massive data series is an optimization problem, and needs to be studied in depth.

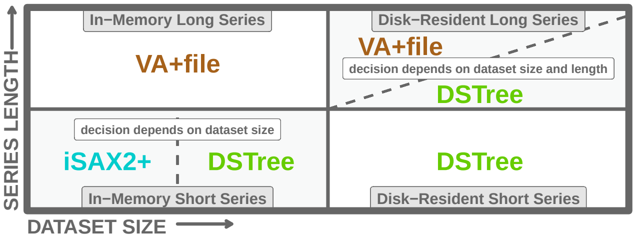

In–Memory Long Series
VA+file
Disk–Resident Long Series
VA+file
decision depends on dataset size and length
DSTree
decision depends on dataset size
iSAX2+
DSTree
In–Memory Short Series
DSTree
Disk–Resident Short Series
SERIES LENGTH
DATASET SIZE

Figure 10: Recommendations (Indexing and answering 10K queries on HDD)

6. CONCLUSIONS AND FUTURE WORK

In this work, we unified and formally defined the terminology used for the different flavors of data series similarity search problems, and we designed and executed a thorough experimental comparison of several relevant techniques from the literature, which had never before been compared at equal footing to one another. Our results paint a clear picture of the strengths and weaknesses of the various approaches, and indicate promising research directions. Part of our future work is the experimental comparison of approximate methods, r-range queries and sub-sequence matching.

Acknowledgments. We sincerely thank all authors for generously sharing their code, and M. Linardi for his implementation of MASS [87]. Work partially supported by EU project NESTOR (Marie Curie 748945).

References

[1] Adhd-200. `http://fcon_1000.projects.nitrc.org/indi/adhd200/`, 2018.

[2] Sloan digital sky survey. `https://www.sdss3.org/dr10/data_access/volume.php`, 2018.

[3] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. pages 69–84, 1993.

[4] S. Albrecht, I. Cumming, and J. Dudas. The momentary fourier transformation derived from recursive matrix transformations. In *Proceedings of 13th International Conference on Digital Signal Processing*, volume 1, pages 337–340 vol.1, Jul 1997.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, Nov. 1998.

[6] J. Aßfalg, H. Kriegel, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Similarity search on time series based on threshold queries. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, pages 276–294, 2006.

[7] J. Aßfalg, H. Kriegel, P. Kröger, and M. Renz. Probabilistic similarity search for uncertain time series. In *Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, New Orleans, LA, USA, June 2-4, 2009, Proceedings*, pages 435–443, 2009.

[8] M. Bach-Andersen, B. Romer-Odgaard, and O. Winther. Flexible non-linear predictive models for large-scale wind turbine diagnostics. *Wind Energy*, 20(5):753–764, 2017.

[9] A. J. Bagnall, J. Lines, A. Bostrom, J. Large, and E. J. Keogh. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Min. Knowl. Discov.*, 31(3):606–660, 2017.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 322–331. ACM, 1990.

[11] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *AAAIWS*, pages 359–370, 1994.

[12] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. *SIGMOD Rec.*, 26(2):357–368, June 1997.

[13] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh. isax 2.0: Indexing and mining one billion time series. In G. I. Webb, B. Liu, C. Zhang, D. Gunopulos, and X. Wu, editors, *ICDM*, pages 58–67. IEEE Computer Society, 2010.

[14] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with isax2+. *Knowl. Inf. Syst.*, 39(1):123–151, 2014.

[15] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27(2):188–228, June 2002.

[16] K.-P. Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pages 126–133, Mar 1999.

[17] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[18] Y. Chen, E. K. Garcia, M. R. Gupta, A. Rahimi, and L. Cazzanti. Similarity-based classification: Concepts and algorithms. *J. Mach. Learn. Res.*, 10:747–776, June 2009.

[19] Y. Chen, M. A. Nascimento, B. C. Ooi, and A. K. H. Tung. Spade: On shape-based pattern detection in streaming time series. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 786–795, 2007.

[20] P. Ciaccia and M. Patella. Bulk loading the M-tree. pages 15–26, Feb. 1998.

[21] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *ICDE*, pages 244–255, 2000.

[22] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In M. Jarke, M. Carey, K. R. Dittrich, F. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, Aug. 1997. Morgan Kaufmann Publishers, Inc.

[23] R. Cole, D. E. Shasha, and X. Zhao. Fast window correlations over uncooperative time series. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, Illinois, USA, August 21-24, 2005*, pages 743–749, 2005.

[24] M. Dallachiesa, B. Nushi, K. Mirylenka, and T. Palpanas. Uncertain time-series similarity: Return to the basics. *PVLDB*, 5(11):1662–1673, 2012.

[25] M. Dallachiesa, T. Palpanas, and I. F. Ilyas. Top-k nearest neighbor search in uncertain data series. *PVLDB*, 8(1):13–24, Sept. 2014.

[26] G. Das, D. Gunopulos, and H. Mannila. Finding similar time series. *Principles of Data Mining and Knowledge Discovery*, pages 88–100, 1997.

[27] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.

[28] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. `http://www.mi.parisdescartes.fr/~themisp/dsseval/`, 2018.

[29] ESA. SENTINEL-2 mission, 2018.

[30] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, New York, NY, USA, 1994. ACM.

[31] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E.

Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *In Proceedings of the 9th ACM Int. Conf. on Information and Knowledge Management*, pages 202–209. ACM Press, 2000.

[32] I. R. I. for Seismology with Artificial Intelligence. Seismic Data Access. http://ds.iris.edu/data/access/, 2018.

[33] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(4):744–755, Apr. 2014.

[34] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[35] X. Golay, S. Kollias, G. Stoll, D. Meier, A. Valavanis, and P. Boesiger. A new correlation-based fuzzy logic clustering algorithm for fmri. *Magnetic Resonance in Medicine*, 40(2):249–260, 1998.

[36] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

[37] M. Hadjieleftheriou. The libspatialindex api, January 2014. http://libspatialindex.github.io/.

[38] G. Hébrail. *Practical data mining in a large utility company*, pages 87–95. Physica-Verlag HD, Heidelberg, 2000.

[39] P. Huijse, P. A. Estévez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Comp. Int. Mag.*, 9(3):27–39, 2014.

[40] Y. Kakizawa, R. H. Shumway, and M. Taniguchi. Discrimination and clustering for multivariate time series. *Journal of the American Statistical Association*, 93(441):328–340, 1998.

[41] K. Kashino, G. Smith, and H. Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.

[42] S. Kashyap and P. Karras. Scalable knn search on vertically stored time series. In C. Apt, J. Ghosh, and P. Smyth, editors, *KDD*, pages 1334–1342. ACM, 2011.

[43] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.

[44] E. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, Oct. 2003.

[45] E. Keogh and M. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, *Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98)*, pages 239–241, New York City, NY, 1998. ACM Press.

[46] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowl. Inf. Syst.*, 7(3):358–386, Mar. 2005.

[47] S. Knieling, J. Niediek, E. Kutter, J. Bostroem, C. Elger, and F. Mormann. An online adaptive screening procedure for selective neuronal responses. *Journal of Neuroscience Methods*, 291(Supplement C):36 – 42, 2017.

[48] K. Košmelj and V. Batagelj. Cross-sectional approach for clustering time varying data. *Journal of Classification*, 7(1):99–109, 1990.

[49] M. Kumar, N. R. Patel, and J. Woo. Clustering seasonality patterns in the presence of errors. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 557–563, 2002.

[50] J. Lin, E. J. Keogh, S. Lonardi, and B. Y. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, DMKD 2003, San Diego, California, USA, June 13, 2003*, pages 2–11, 2003.

[51] M. Linardi and T. Palpanas. Scalable, variable-length similarity search in data series: The ULISSE approach. *PVLDB*, 11(13):2236–2248, 2018.

[52] M. Linardi and T. Palpanas. ULISSE: ULtra compact Index for Variable-Length Similarity SEarch in Data Series. In *ICDE*, 2018.

[53] M. Linardi, Y. Zhu, T. Palpanas, and E. J. Keogh. Matrix profile X: Valmod - scalable discovery of variable-length motifs in data series. 2018.

[54] C. Maccone. Advantages of karhunenlove transform over fast fourier transform for planetary radar and space debris detection. *Acta Astronautica*, 60(8):775 – 779, 2007.

[55] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016.

[56] K. Mirylenka, V. Christophides, T. Palpanas, I. Pefkianakis, and M. May. Characterizing home device usage from wireless traffic time series. In *EDBT*, pages 551–562, 2016.

[57] K. Mirylenka, M. Dallachiesa, and T. Palpanas. Data series similarity using correlation-aware measures. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, pages 11:1–11:12, 2017.

[58] A. Mueen, Y. Zhu, M. Yeh, K. Kamgar, K. Viswanathan, C. Gupta, and E. Keogh. The fastest similarity search algorithm for time series subsequences under euclidean distance, August 2017. http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html.

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

[60] T. Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In R. M. Freivalds, G. Engels, and B. Catania, editors, *SOFSEM*, volume 9587 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2016.

[61] P. Paraskevopoulos, T.-C. Dinh, Z. Dashdorj, T. Palpanas, and L. Serafini. Identification and characteri-

zation of human behavior patterns from mobile phone data. In *D4D Challenge session, NetMob*, 2013.

[62] B. Peng, T. Palpanas, and P. Fatourou. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. *IEEE BigData*, 2018.

[63] D. Rafiei. On similarity-based queries for time series data. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 23-26, 1999*, pages 410–417, 1999.

[64] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. *SIGMOD Rec.*, 26(2):13–25, June 1997.

[65] D. Rafiei and A. O. Mendelzon. Efficient retrieval of similar time sequences using DFT. *CoRR*, cs.DB/9809033, 1998.

[66] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In Q. Yang, D. Agarwal, and J. Pei, editors, *KDD*, pages 262–270. ACM, 2012.

[67] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 547–556. IEEE, 2011.

[68] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical data prediction for real-world wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, accepted for publication, 2015.

[69] P. P. Rodrigues, J. Gama, and J. P. Pedroso. Odac: Hierarchical clustering of time series data streams. In J. Ghosh, D. Lambert, D. B. Skillicorn, and J. Srivastava, editors, *SDM*, pages 499–503. SIAM, 2006.

[70] S. R. Sarangi and K. Murthy. DUST: a generalized notion of similarity between uncertain time series. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 383–392, 2010.

[71] P. Schäfer and M. Högqvist. Sfa: A symbolic fourier approximation and index for similarity search in high dimensional datasets. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 516–527, New York, NY, USA, 2012. ACM.

[72] D. Shasha. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.*, 22(2):40–46, 1999.

[73] J. Shieh and E. Keogh. isax: Indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 623–631, New York, NY, USA, 2008. ACM.

[74] J. Shieh and E. Keogh. isax: Indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 623–631, New York, NY, USA, 2008. ACM.

[75] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[76] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[77] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: Solving C-approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB*, 8(1):1–12, Sept. 2014.

[78] S. University. Southwest University Adult Lifespan Dataset (SALD). `http://fcon_1000.projects.nitrc.org/indi/retro/sald.html?utm_source=newsletter&utm_medium=email&utm_content=See%20Data&utm_campaign=indi-1`, 2018.

[79] S. C. Vision. Deep billion-scale indexing. `http://sites.skoltech.ru/compvision/noimi`, 2018.

[80] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Min. Knowl. Discov.*, 26(2):275–309, Mar. 2013.

[81] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 6(10):793–804, 2013.

[82] T. Warren Liao. Clustering of time series dataa survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[83] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[84] B. M. Williams and L. A. Hoel. Modeling and forecasting vehicular traffic flow as a seasonal arima process: Theoretical basis and empirical results. *Journal of Transportation Engineering*, 129(6):664–672, 2003.

[85] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Dpisax: Massively distributed partitioned isax. 2017.

[86] A. B. Yandex and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2055–2063, June 2016.

[87] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh. Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile. *Data Mining and Knowledge Discovery*, pages 1–41, 2017.

[88] M. Yeh, K. Wu, P. S. Yu, and M. Chen. PROUD: a probabilistic approach to processing similarity queries over uncertain data streams. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 684–695, 2009.

[89] K. Zoumpatianos, S. Idreos, and T. Palpanas. ADS: the adaptive data series index. *VLDBJ*, 25(6):843–866, 2016.

[90] K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and

J. Gehrke. Generating data series query workloads. *VLDBJ*, 2018.

[91] K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke. Query workloads for data series indexes. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1603–1612, 2015.