Efficient Range and kNN Twin Subsequence Search in Time Series

Georgios Chatzigeorgakidis, Dimitrios Skoutas, Kostas Patroumpas, Themis Palpanas, Spiros Athanasiou and Spiros Skiadopoulos

Abstract—Analyzing time series data is crucial for many applications. In particular, subsequence search refers to finding subsequences within an input time series T that are similar to a query sequence Q. Existing subsequence search approaches typically employ Euclidean distance or Dynamic Time Warping as similarity measures and address range queries. In this paper, we focus on Chebyshev distance, which is the largest difference between each individual pair of points across the entire length of two compared subsequences. We call such similar pairs *twins*. We first show how existing time series indices can be extended to perform twin subsequence search. Then, we introduce TS-Index, a novel index tailored to the computation of twin subsequence search queries. Moreover, given that specifying a distance threshold is often not straightforward, we show how TS-Index can also be used to evaluate kNN queries. Our extensive experimental evaluation compares these approaches using real time series datasets. The results demonstrate that TS-Index can retrieve twin subsequences faster than all other methods under various conditions.

Index Terms—time series indexing, subsequence matching, similarity search.

1 Introduction

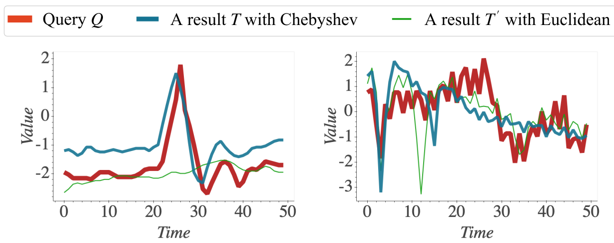
A time series is a sequence of time-ordered data points. It can represent various types of measurements, ranging from household resource consumption sensor readings to human body measurements using special medical instruments such as an electrocardiogram. In the last years, generation of time series data has grown exponentially due to rapid technological advancements in mining and monitoring applications, including sensors and IoT. Analyzing time series can provide various insights, such as the discovery of trends and patterns, which has also led to an increasing scientific research interest [5], [9], [18].

One of the fundamental problems over time series data is *subsequence search*. Given an input time series T and a query sequence Q ($|Q| \ll |T|$), subsequence search (or matching) refers to finding subsequences within T that are similar to Q. Most existing approaches employ Euclidean distance or Dynamic Time Warping (DTW) as similarity measure [27], [32]. Nevertheless, as observed in [35], no single measure of similarity is suitable for every application or dataset (for instance, different $\mathcal{L}_p$ norms capture different patterns of similarity), hence even a single user may want to examine different measures. This is also shown experimentally in [8], where different similarity measures achieve different classification accuracy in different datasets.

In this work, we focus on *Chebyshev* distance (also known as $\mathcal{L}_\infty$ norm or maximum norm). It requires that the respective values of two time series are always close to each other, while Euclidean distance or DTW allow a few values to deviate as long as the rest are sufficiently close. More specifically, the Chebyshev distance between two subsequences of equal length is the maximum difference of their values across their entire duration. We call two subsequences *twins* with respect to a distance threshold ϵ, if their Chebyshev distance is not greater than ϵ, i.e., their values do not differ by more than ϵ at any timestamp.

Various applications can benefit from twin subsequence search. One example concerns identifying *doublet* earthquakes. Two earthquakes are characterized as doublets if their epicenter is located relatively close and they have almost identical waveforms [36], [37]. Twin subsequence search could help identify such cases, facilitating seismologists that study changes in the Earth's inner core. Another application comes from smart Traffic Light System (TLS) cameras at road intersections, which count the number of vehicles crossing towards each direction [10]. Modern TLS sensors record several values per second, for tens of thousands of traffic lights in a large city. Finding subsequences of traffic data that are nearly identical across their entire length can provide road network analysts with useful insights regarding recurring events at specific road intersections in a city, which can subsequently lead to improved traffic control and more accurate congestion estimates. Furthermore, twin subsequence search could be used to detect patterns in medical applications. It could be useful in cases where doctors search for very similar, known previous patterns of the same

- *Georgios Chatzigeorgakidis was with the Information Management Systems Institute of the "Athena" Research and Innovation Center, Athens, Greece.*
E-mail: see gchatzi@athenarc.gr
- *Dimitrios Skoutas was with the Information Management Systems Institute of the "Athena" Research and Innovation Center, Athens, Greece.*
E-mail: see dskoutas@athenarc.gr
- *Kostas Patroumpas was with the Information Management Systems Institute of the "Athena" Research and Innovation Center, Athens, Greece.*
E-mail: see kpatro@athenarc.gr
- *Themis Palpanas was with LIPADE, Université Paris Cité and French University Institute (IUF), Paris, France.*
E-mail: see themis@mi.parisdescartes.fr
- *Spiros Athanasiou was with the Information Management Systems Institute of the "Athena" Research and Innovation Center, Athens, Greece.*
E-mail: see spathan@athenarc.gr
- *Spiros Skiadopoulos was with the Department of Informatics and Telecommunications of the University of the Peloponnese, Tripoli, Greece.*
E-mail: see spiros@uop.gr



(a) Absence of desired spike (b) Presence of undesired spike

Fig. 1. Examples of false positives obtained with Euclidean distance compared to results with Chebyshev distance on subsequences of the EEG dataset.

person in Electroencephalography (EEG) or Electrocardiography (ECG) sequences [25], [28] that indicate a previous medical condition, or try to detect irregularities where the difference between a normal and an abnormal waveform exceeds a given threshold [31]. Finally, Chebyshev distance has been used in [17], revealing interesting findings on time series representing closing prices of US mutual funds, as well as in [7] for hyperspectral imaging classification.

A question that naturally arises is whether the same results can be obtained by subsequence search using Euclidean distance. We investigate this empirically by performing the following indicative experiment on an EEG time series [21] with a length of 1,801,999 timestamps. Considering a query sequence Q and an initial Chebyshev distance threshold ϵ, we identify all twin subsequences, producing 1,034 results in total. We then attempt to retrieve the same results by subsequence search using Euclidean distance. As will be shown later (see Lemma 1 in Section 3.1), we need to set the Euclidean distance threshold to $\epsilon' = \epsilon \times \sqrt{|Q|}$, where $|Q|$ is the query subsequence length, in order to ensure no false negatives. Searching with this relaxed threshold under Euclidean distance yields 127,887 results, which include too many false positives, thus requiring an expensive post-processing step to identify the correct 1,034 twin subsequences. We provide a more detailed analysis in Section 6.2.

Figure 1 exemplifies the intuition behind the false positives. Assume a query sequence Q and two matches, T, T', obtained under Chebyshev and Euclidean distance, respectively. As shown, T matches Q in all timestamps. Instead, although T' is a match under the corresponding Euclidean distance threshold ϵ', it either lacks a spike that is present in Q (Fig. 1a), or exhibits one not present in Q (Fig. 1b).

Given a query sequence Q and a time series T, a naïve process for finding twin subsequences of Q across T is by performing a sweepline scan. This scans T using a sliding window of length $|Q|$, comparing at each timestamp the query with the current subsequence extracted from T, and adding it to the results if it satisfies the given threshold ϵ on Chebyshev distance. However, this approach is clearly inefficient for very long time series.

In this work, we investigate index-based methods to efficiently execute twin subsequence search. First, we show how two state-of-the-art time series indices, namely KV-Index [32] and iSAX [30] can be adapted for this purpose. Then, we present our proposed index, called TS-Index, which is tailored and optimized for this problem. TS-Index, initially introduced in [6], is a tree structure that summarizes the subsequences contained within each node using *Minimum Bounding Time Series* (MBTS) [4], consisting of an upper and lower bounding sequence. Furthermore, we describe optimizations to the TS-Index in terms of memory footprint and index construction time. The former is based on Piecewise Aggregate Approximation (PAA) [14], while the latter on bulk loading, utilizing an embedding of subsequences to a low-dimensional space followed by an ordering using a space filling curve. We introduce efficient exact algorithms for twin subsequence search, addressing both threshold-based and k-nearest neighbor (kNN) queries. We also consider the execution of variable-length queries. To evaluate the performance of these methods, we conduct a comprehensive experimental study against both real-world and synthetic time series, including also a case study for qualitatively assessing the results in certain real-world scenarios. Our experiments show that twin subsequence search with TS-Index is faster than with previous indices, often by orders of magnitude.

Summarizing, our main contributions are as follows:

- We introduce the problem of twin subsequence search, and propose a filter-verification algorithm that can be applied on state-of-the-art indices.
- We introduce TS-Index, a tree-based index tailored to twin subsequence search, which utilizes appropriate bounds in its nodes to prune the search space.
- We propose optimizations that improve the memory footprint of the index and reduce its construction cost via bulk loading.
- We present an algorithm for executing twin subsequence kNN queries over TS-Index.
- We experimentally evaluate our proposed methods using real-world and synthetic datasets and comparing them against appropriately adapted state-of-the-art time series indices in terms of query execution, memory footprint and index construction time.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem. Section 4 presents how it can be addressed based on existing indices. Section 5 presents the proposed TS-Index. Section 6 reports our experimental results. Finally, Section 7 concludes the paper.

2 Related Work

A straightforward method for subsequence search is with a sweepline that scans the time series using a sliding window. UCR suite [27] offers such a framework, comprising various optimizations. Furthermore, Matrix Profile [34] includes methods to detect the nearest neighbor subsequence for each subsequence of a time series. However, the optimizations introduced in these methods are specifically tailored to Euclidean distance and therefore cannot be applied for twin subsequence search. Furthermore, the lack of an index poses efficiency and scalability limitations.

Indexing is advantageous for various time series search operations; a detailed survey and experimental evaluation of time series indices for similarity search can be found in [9]. One family of methods is based on Discrete Wavelet Transform [11]. This reduces the dimensionality of the time series and generates an index based on the transformed sequences. For instance, the *Haar wavelet* [12] has been used to gradually reduce the dimensionality of time series and build an index on the obtained coefficients [3]. Precision and performance improvements can be achieved via bi-orthonormal wavelets [26]. Furthermore, kNN search can be performed by accessing the coefficients of Haar-wavelet-transformed time series through a sequential scan over step-wise increasing resolutions [13].

A more recent approach is based on the *Symbolic Aggregate Approximation* (SAX) [16] representation. A SAX *word* is a multi-resolution summary of a time series, quantized on the value domain. It is derived from the Piecewise Aggregate Approximation (PAA) [14], which segments a time series on the time axis and approximates it by retaining only the mean value per segment. This has led to the iSAX index [30], a tree-based structure built over the SAX words of a set of time series, leveraging their multi-resolution characteristics. Each node in iSAX contains a SAX word that guarantees a lower bound in terms of Euclidean distance for all the time series indexed by it. To answer similarity search queries, the index is traversed in a top-down fashion, comparing at each step the SAX representation of the query against the ones contained in each visited node.

Several extensions to iSAX have been proposed [22]. iSAX 2.0 [1] enables bulk loading, while iSAX2+ [2] minimizes the I/O operations caused by node splitting during construction. ADS+ [38] is an adaptive version that builds the index on-demand, while processing the queries. DPiSAX [33] is a parallel and distributed version. ParIS [23] and MESSI [24] take advantage of modern multi-core architectures and hardware parallelization to accelerate processing times for disk-based and in-memory indices, respectively. Coconut [15] introduces sortable SAX representations and utilizes a space filling curve to sort the time series and build an index in a bottom-up fashion. ULISSE [19] can answer similarity search queries of varying length.

Another recent method for subsequence search is KV-Index [32]. After extracting all subsequences of a given length from a time series and deriving their corresponding mean values, it generates an index containing key-value pairs. Each key represents a range of mean values for a group of subsequences, pointing to starting positions of these subsequences along the original time series.

As we show in Section 4, it is possible to execute twin subsequence search queries using iSAX or KV-Index. However, since these indices are tailored to similarity search using Euclidean distance, this approach is suboptimal, as indicated also in our experiments in Section 6. Moreover, an index for arbitrary $\mathcal{L}_p$ norms is described in [35]. It divides each sequence into a fixed number of equi-sized segments, and takes the mean of each segment to form a feature vector. Such a generic approach favors flexibility; instead, our focus in this paper is on optimizing performance specifically for queries using Chebyshev distance.

In a previous work [5], we studied the problem of discovering pairs and bundles of similar time-aligned subsequences within a collection of time series, based on Chebyshev distance, using a sweepline approach. In this paper, we focus on searching for twin subsequences in an input time series T that are similar to a query subsequence Q, which is a different problem, and we propose an index-based approach. In another previous work [4], we have developed a hybrid index, called BTSR-Tree, which also employs the concept of Minimum Bounding Time Series (MBTS) to prune the search space. However, this is a spatial-first index specifically tailored to queries over geo-located time series. Moreover, it is based on Euclidean distance instead of Chebyshev, and it does not support bulk-loading.

TABLE 1
Basic notations

T	input time series
n	length of time series T
Q	query subsequence
l	subsequence length
l'	query length ($l' \geq l$)
ϵ	distance threshold
k	number of nearest subsequences to retrieve
$T_{p,l}$	subsequence of T with length l starting at position p
S	(sub)sequence
$d(S, S')$	Chebyshev distance between equi-length subsequences
μ	mean value of a sequence
$B^{\sqcap}$	upper bounding time series of an MBTS
$B^{\sqcup}$	lower bounding time series of an MBTS
V_S	embedding of subsequence S
μ_c	minimum node capacity in TS-Index
M_c	maximum node capacity in TS-Index
m	number of segments

3 Problem Definition

Next, we formally introduce the twin subsequence search problem and describe a generic filter-verification approach. Table 1 lists basic notations used throughout the paper.

3.1 Problem Statement

A *time series* is a time-ordered sequence $T = \{T_1, T_2, ..., T_n\}$, where T_i is the value at the i-th timestamp and $n = |T|$ is the length of the series (i.e., number of timestamps). We use $T_{p,l}$ to denote the *subsequence* $\{T_p, ..., T_{p+l-1}\}$ starting at timestamp p and having length l, where $1 \leq p \leq p + l - 1 \leq n$. For brevity, we also use S to generally refer to a (sub)sequence.

Given two sequences S and S' of equal length l, we call them *twins* if their Chebyshev distance is not greater than a given threshold ϵ. The Chebyshev distance of two vectors is their maximum difference along any dimension. Hence, if S and S' are twin sequences with respect to ϵ, their values at any timestamp should not differ by more than ϵ. Formally:

Definition 1 (Twin Sequences). *Two sequences S and S' of equal length l are called* twins *with respect to a given threshold ϵ, denoted as $S_1 \sim_\epsilon S_2$, if their Chebyshev distance d is not greater than ϵ, i.e., $d(S, S') := \max_{i=0}^{l-1}(|S_i - S'_i|) \leq \epsilon$.*

We can now formally define the problem:

Problem 1 (Twin Subsequence Search). *Given a query sequence Q of length l, a time series T of length $n \gg l$, and a distance threshold ϵ, find all subsequences S in T ($|S| = l$) such that $Q \sim_\epsilon S$.*

The following lemma establishes a relation between a given Chebyshev distance threshold and a corresponding Euclidean distance threshold.

Lemma 1. *Given two twin sequences $S \sim_\epsilon S'$ of length l, their Euclidean distance is $ED(S, S') \leq \epsilon \times \sqrt{l}$.*

Proof. If $S \sim_\epsilon S'$, then $ED(S, S') = \sqrt{\sum_i (S_i - S'_i)^2} \leq \sqrt{\sum_i \epsilon^2} = \epsilon \times \sqrt{l}$. □

Moreover, from Definition 1, it is straightforward to derive the following property, stating that any pair of time-aligned subsequences across two twin sequences are also twins.

Lemma 2. *Given two twin series $T \sim_\epsilon T'$, then $T_{p,l} \sim_\epsilon T'_{p,l}$ for any $l \in [1, |T|]$ and $p \in [1, |T| - l]$.*

Notice that z-normalization is often used when comparing time series. Throughout the paper, we consider various possibilities: (a) working with the raw values, (b) z-normalizing the entire time series, (c) z-normalizing each individual subsequence. We discuss the implications of each case where relevant.

3.2 Filter-Verification Approach

We can detect twin subsequences following a filter-verification framework: the first step (*filtering*) generates candidate subsequences, which are then evaluated in the second step (*verification*) to identify those satisfying the Chebyshev distance threshold. A straightforward approach for generating candidates is to scan the entire time series T with a *sweepline* and consider each subsequence $T_{p,l}$ for $p \in [1, |T| - l]$ as a candidate.

Verification is done by checking all pairwise value differences between Q and $T_{p,l}$. If the difference found at a timestamp exceeds ϵ, then candidate $T_{p,l}$ is rejected, otherwise it is accepted. Verification can be accelerated by detecting false positives as early as possible. To this end, if the values are z-normalized, we can prioritize those points in Q having the highest absolute value, since these are less likely to have a match with the respective points in $T_{p,l}$. This optimization is also used in *UCR Suite* [27], and is known as *reordering early abandoning*.

Clearly, the drawback of this sweepline approach is that it generates an excessive number of candidates (specifically, $|T| - l$), thus incurring a prohibitive cost when dealing with very long series. To filter candidates more effectively, in the following sections we consider methods based on indexing the subsequences of T. First, we address the problem using existing state-of-the-art indices, and then we introduce a novel index tailored to twin subsequence search.

4 Searching with Existing Indices

Next, we focus on two representative state-of-the-art indices for time series similarity search, namely KV-Index [32] and iSAX [2], showing how they can be used for twin subsequence search without altering their structure.

4.1 KV-Index

KV-Index is a state-of-the-art index for subsequence matching [32]. Given a time series T, it is built by considering all its subsequences of a pre-defined length l. Each subsequence S is represented by a pair (p, μ), where p is its starting position (i.e., timestamp) in T and μ is its mean value over the next l timestamps. KV-Index is an inverted index constructed over these pairs. Each key is a range of mean values, whereas each inverted list entry contains intervals of positions.

Twin subsequence search can be performed with KV-Index based on the following lemma.

Lemma 3. *Let two subsequences S and S' of length l. If $S \sim_\epsilon S'$, then $|\mu - \mu'| \leq \epsilon$, where μ and μ' are the mean values of S and S', respectively.*

Proof. Let S and S' be twin subsequences of length l. We assume that $|\mu - \mu'| > \epsilon$, and prove the lemma by contradiction: $|\mu - \mu'| > \epsilon \Rightarrow \frac{1}{l} \times |\sum_i S_i - \sum_i S'_i| > \epsilon \Rightarrow \sum_i |S_i - S'_i| > l \times \epsilon$. The latter can only hold if there exists at least one timestamp i where $|S_i - S'_i| > \epsilon$, in which case it cannot hold that $S \sim_\epsilon S'$. □

Based on Lemma 3, we can use a KV-Index built over a time series T to generate candidates for detecting twin subsequences. Specifically, assume a query sequence Q with mean value μ_q. The candidate subsequences in T are those included in the inverted lists with keys $[\mu_{min}, \mu_{max}]$, such that $\mu_{min} - \epsilon \leq \mu_q \leq \mu_{max} + \epsilon$. Then, the obtained candidates must be verified to derive the final results. Notice that Lemma 3 is not effective if each individual subsequence has been z-normalized, because then all mean values are zero. Hence, KV-Index is applicable when working with raw values or if the entire sequence is z-normalized.

4.2 iSAX Index

iSAX is a tree index structure for time series similarity search [2]. Time series are z-normalized and indexed using their *Symbolic Aggregate approXimation (SAX)* [30]. The SAX representation of a series is derived in two steps. The first applies *Piecewise Aggregate Approximation* (PAA) [14], which splits the series in a specified number m of segments and approximates each one with the mean value over the corresponding time interval. The second step applies quantization to assign each mean value to a discrete SAX *symbol*. Hence, each SAX symbol X corresponds to a range of mean values $[\mu_{X_{min}}, \mu_{X_{max}})$. The SAX representation of a series is a sequence of m SAX symbols (one symbol per segment), and is called SAX *word*. Notice that, by default, SAX words are derived using precomputed breakpoints that are selected assuming z-normalized values; nevertheless, non-normalized values can also be handled by adjusting the breakpoints accordingly.

Twin subsequence search can be performed over the iSAX index based on the following lemma.

Lemma 4. *Let two subsequences S and S' of length l, and their SAX representations $SAX(S) = \{X_1, X_2, ..., X_m\}$ and $SAX(S') = \{X'_1, X'_2, ..., X'_m\}$. If $S \sim_\epsilon S'$, then the following conditions must hold $\forall i \in [1, m]$: $\mu_{X_{i_{max}}} \geq \mu_{X'_{i_{min}}} - \epsilon$ and $\mu_{X_{i_{min}}} \leq \mu_{X'_{i_{max}}} + \epsilon$.*

Proof. According to Lemma 3, if two sequences are twins with respect to a threshold ϵ, then the difference between their mean values is also bounded by ϵ. Besides, according to Lemma 2, any pair of time-aligned segments across two twin sequences are also twins. Hence, if $S \sim_\epsilon S'$, then for each pair of symbols X_i and X'_i in the respective

SAX representations, the mean values denoted by these symbols must not differ by more than ϵ, which means that $\mu_{X_{i_{max}}} \geq \mu_{X'_{i_{min}}} - \epsilon$ and $\mu_{X_{i_{min}}} \leq \mu_{X'_{i_{max}}} + \epsilon$. □

Based on the above, we can perform twin subsequence search using iSAX as follows. Given a time series T, we construct an iSAX index over all its l-length subsequences. Then, for a query sequence Q, we traverse the iSAX index starting from its root. At each node, we check the SAX word of Q against the SAX word of that node, applying Lemma 4. If the check fails, the node and its subtree can be safely pruned; otherwise, the search continues at the node's children. Once a leaf node is reached, and qualifies according to this check, all subsequences indexed therein are retrieved as candidates for verification.

5 The TS-Index

As discussed in Section 4, it is possible to use KV-Index or iSAX to identify candidates for twin subsequence queries. However, since these indices are not tailored to the matching criterion, they tend to generate a large number of false positives, incurring a significant verification cost, as confirmed in our experiments (Section 6). In the following, we introduce TS-Index, which is specifically designed for twin subsequence search. First, we provide an overview of its structure and explain how it is constructed. Then, we present an algorithm to evaluate twin subsequence queries specifying a distance threshold. We also propose optimizations for reducing the index size and its construction cost. Finally, we discuss how the TS-Index can be used to evaluate kNN queries. The latter is very useful in practice, because it is often not intuitive for the user to specify a distance threshold.

5.1 Index Structure

The core concept in TS-Index is that of *Minimum Bounding Time Series* (MBTS) [4]. An MBTS is a pair of sequences that fully encloses a set of time series $\mathcal{T}$ by indicating the maximum and minimum values at each timestamp. Figure 2a depicts an example of an MBTS enclosing a set of four time series. Formally:

Definition 2 (MBTS)**.** *Given a set $\mathcal{T}$ of time series with equal length l, its MBTS $B = (B^{\sqcap}, B^{\sqcup})$ consists of an* upper bounding time series $B^{\sqcap}$ *and a* lower bounding time series $B^{\sqcup}$, *constructed by respectively selecting the maximum and minimum values at each timestamp $i \in \{1, \ldots, l\}$ among all time series in $\mathcal{T}$ as follows:*

$$B^{\sqcap} = \{\max_{T \in \mathcal{T}} T_1, \ldots, \max_{T \in \mathcal{T}} T_l\}$$
$$B^{\sqcup} = \{\min_{T \in \mathcal{T}} T_1, \ldots, \min_{T \in \mathcal{T}} T_l\} \tag{1}$$

The TS-Index has a tree structure. Each internal node points to a set of children nodes, whereas each leaf node points to a set of subsequences (more specifically, to the starting positions of its indexed subsequences along the input time series T). All leaf nodes are at the same level. Each node is associated with an MBTS, which encloses all the sequences indexed therein. Clearly, MBTS get tighter when descending from the root to the leaf level. Figure 3a illustrates an example of TS-Index for eight input sequences. The MBTS of each node is depicted as a grey band.

5.2 Index Construction

Assume an input time series T and a subsequence length l. The TS-Index over T is constructed in a top-down fashion, by sequentially inserting all l-length subsequences of T. When inserting a sequence S, we traverse the index from the root, selecting at each level the node whose MBTS has the smallest distance from S, until a leaf node is reached. The distance between a sequence S and an MBTS B is calculated using the following formula:

$$d(S, B) = \max_i \begin{cases} S_i - B_i^{\sqcap} & \text{if } S_i > B_i^{\sqcap} \\ B_i^{\sqcup} - S_i & \text{if } S_i < B_i^{\sqcup} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $B_i^{\sqcap}$ and $B_i^{\sqcup}$ are the i^{th} values of the upper and lower bounds of the MBTS B, respectively.

Each node has a minimum capacity μ_c and a maximum *capacity* M_c, specifying the minimum and maximum number of children it can point to. Once a node exceeds M_c, it is split in two nodes. This may cause the parent node to also exceed the maximum capacity M_c, in which case it is split too. Hence, this process recursively propagates upwards until no further splits occur. This procedure ensures that all leaves are placed on the same level of the tree.

During node splitting, the goal is to make the MBTS of each new sibling node as tight as possible. If this is a leaf node, we identify the two subsequences within the original node having the highest Chebyshev distance and use them as seeds for the two sibling nodes. Each remaining subsequence is assigned to the node where it causes the smallest expansion of its MBTS, which gets updated accordingly. For an internal node, the process is similar. Yet, adjusting its MBTS in this case involves the MBTS of children nodes instead of individual sequences. To accommodate this, the distance between two MBTS B_1 and B_2 is defined as:

$$d(B_1, B_2) = \max_i \begin{cases} B_{1,i}^{\sqcup} - B_{2,i}^{\sqcap} & \text{if } B_{1,i}^{\sqcup} > B_{2,i}^{\sqcap} \\ B_{2,i}^{\sqcup} - B_{1,i}^{\sqcap} & \text{if } B_{1,i}^{\sqcap} < B_{2,i}^{\sqcup} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

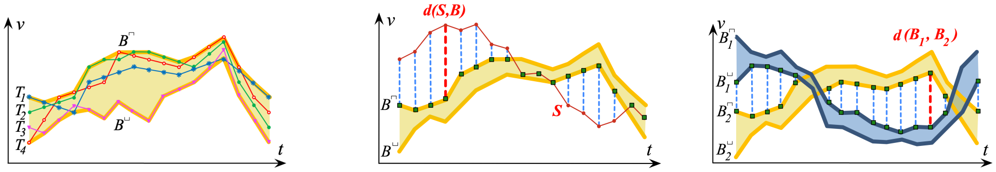
where $B_{1,i}^{\sqcup}, B_{1,i}^{\sqcap}$ and $B_{2,i}^{\sqcup}, B_{2,i}^{\sqcap}$ are the i^{th} values of the upper and lower bounds of the MBTS B_1 and B_2, respectively. Figures 2b and 2c exemplify the calculation of the distance of a sequence S to an MBTS B and the calculation of the distance between two MBTS (B_1, B_2) respectively; in both cases, the distance is the length of the dashed red line.

Figure 3b depicts an example where inserting subsequence p_{10} into leaf node A_3 of the TS-Index in Figure 3a, causes it to split into two new nodes, A'_3 and A_4 (we assume $\mu_c = 2$ and M_c=3). This process is then propagated upwards, splitting the root into B_1 and B_2. To keep the MBTS tight –according to Eq. 3–, nodes A_1, A_4 have become children of B_1 and A_2, A'_3 are now children of B_2. Finally, a new root is added, increasing the index height by one.

5.3 Query Execution

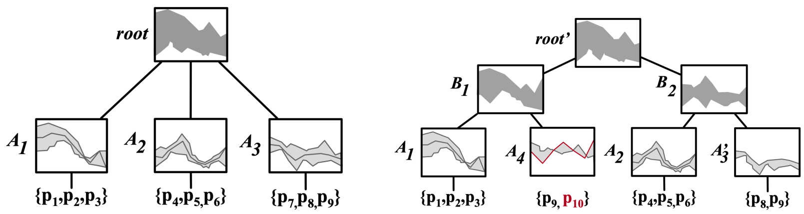
Twin subsequence search can be performed with a TS-Index based on the following lemma.

Lemma 5. *Assume a query sequence Q and a node N of the TS-Index with MBTS B. If there exists a sequence S indexed at N such that $Q \sim_\epsilon S$, then $d(Q, B) \leq \epsilon$.*



(a) (b) (c)

Fig. 2. (a) MBTS enclosing a set of 4 time series. Distance between (b) a sequence S and an MBTS B, (c) MBTS B_1 and B_2.



(a) (b)

Fig. 3. (a) TS-Index for 9 input sequences. (b) Inserting p_{10} causes a split at leaf A_3 and splits propagate upwards.

Algorithm 1: `TwinSubsequenceSearch`

Input : Time series T, TS-Index I, query Q, threshold ϵ
Output: List R of twin subsequences to Q

1 $R \leftarrow \emptyset$
2 $L \leftarrow I.root$.getChildren()
3 **while** $L \neq \emptyset$ **do**
4   $N \leftarrow L$.getNext()
5   $B \leftarrow N.MBTS$
6   **if** $d((Q,B) \leq \epsilon$ **then**
7   **if** N *is not leaf* **then**
8       $L \leftarrow L \cup \{N$.getChildren()$\}$
9   **else**
10   **foreach** $p \in N.getPositions()$ **do**
11   **if** $d((Q, T_{p,l}) \leq \epsilon$ **then**
12           $R \leftarrow R \cup T_{p,l}$
13 **return** R

Proof. Assume that $Q \sim_\epsilon S$ for a sequence S indexed by node N. From Def. 2, it follows that $S_i \in [B_i^{\sqcup}, B_i^{\sqcap}]$ for each timestamp i. Moreover, from Def. 1, it follows that $|Q_i - S_i| \leq \epsilon$. Hence, from Eq. 2, we derive $d(Q,B) \leq \epsilon$. □

Given a query sequence Q, we traverse the index in a top-down fashion, starting from its root. For each visited node N, we compare Q against N's MBTS, applying Lemma 5 to prune its subtree. Note that this check can be accelerated, since it is not necessary to fully compute distance $d(Q,B)$; instead, if the indexed values have been z-normalized, we apply early abandoning (see Section 3.2) to prune the node as soon as the value difference exceeds ϵ in at least one timestamp. Multiple paths starting from the root may need to be explored, depending on the query and the tightness of the bounds in the visited nodes.

Algorithm 1 describes the search process. The input includes the query sequence Q, the constructed TS-Index I, the given time series T and the threshold ϵ. We start by initializing a list L with the root's children (Line 2). Then, we traverse the index by iterating over this list (Lines 3-12). For each node N currently in the list, we obtain its MBTS (Lines 4-5). Then, we check whether the distance between this MBTS and the query is higher than the specified threshold ϵ (Line 6). If so, the subtree under the current node N is pruned; otherwise, it is examined as explained next. If N is not a leaf node, we insert its children in list L for probing (Lines 7-8). Once a leaf node is reached, we iterate over all the subsequence positions it contains and check whether each corresponding subsequence is a twin of Q with respect to ϵ. If so, we add this subsequence to the final results (Lines 9-12). The results are returned once all candidate nodes in list L have been either probed or pruned (Line 13).

Cost Analysis. Assume a TS-Index built on a sequence T of length n, using subsequence length l and minimum capacity μ_c. The total number of indexed subsequences is $n - l + 1$. Then, its maximum height is $h_{max} = \lceil log_{\mu_c}(n-l+1) \rceil$. The maximum number of nodes of a tree depends on its fanout f, which in our case is $f \leq M_c$. Specifically, the maximum number of nodes is $\mathcal{O}(f^{h-1})$, where h is the height of the tree. Thus, in our case, the maximum number of nodes is $\mathcal{O}(M_c^{\lceil log_{\mu_c}(n-l+1) \rceil - 1})$. For each visited node, we must compare the query sequence against its two bounding MBTS subsequences. Each such filtering requires a number of comparisons in the range $[1, l]$. Thus, in the worst case, we need to perform $\mathcal{O}(2l \times M_c^{\lceil log_{\mu_c}(n-l+1) \rceil - 1})$ to reach leaf level during a range search. Finally, if at each leaf we have to check all indexed subsequences, we have a final worst case complexity of $\mathcal{O}(l \times (2M_c^{\lceil log_{\mu_c}(n-l+1) \rceil - 1}) + n - l + 1)$.

Answering Queries with Length $l' > l$. We can exploit a TS-Index built over subsequences of length l and also answer queries for any length $l' > l$. According to Lemma 2, if two sequences are twins, then any pair of respective segments within them is also a twin pair with respect to the same distance threshold ϵ. Based on this, we can subdivide the query sequence Q of length $l' > l$ into a number of disjoint consecutive subsequences, each of length l. If the right-most subsequence has length lower than l, it is ignored. Then, for each of these subsequences, we execute a search against the TS-Index to retrieve its twins. For each subsequent execution, the starting timestamp of the resulting subsequences must coincide with the last timestamp of the previous execution, otherwise they are discarded. The final candidate positions on the input sequence are those such that for each execution we obtained consecutive subsequences. Verification on these candidate positions yields the final results.

The correctness of the results follows from Lemma 2. Consider a subsequence S and a query Q having Chebyshev distance not higher than ϵ. If we segment both S and Q as discussed above, then all corresponding pairs of segments must also be within distance ϵ. Therefore, we can treat each query segment as a subquery and then take the intersection of these results.

5.4 Reducing Memory Footprint

For an input time series T and subsequence length l, we need to extract and index $|T| - l$ subsequences. Moreover, each node in the index stores upper and lower bounds (MBTS) that have the form of two sequences of length l. This incurs high memory footprint when $|T|$ is large.

To overcome this issue, we can apply Piecewise Aggregate Approximation (PAA) [14] to the subsequences extracted from T before inserting them into the index. In particular, we split each sequence into m segments and represent each segment by the mean value over the corresponding part of the sequence. As PAA is also applied on query sequence Q, query execution can still be performed as described above, but now involving the PAA representations instead of the original subsequences. While traversing the TS-Index, we compare the query's PAA to each corresponding MBTS of each node, which also consists of an upper and lower PAA bound. If the Chebyshev distance exceeds the given threshold (according to Eq. 2), we can safely prune the node. The correctness of this can be easily derived from Lemma 2 (which implies that each pair of respective segments should also be twins) combined with Lemma 3 (which implies that the pairwise differences between mean values are also bounded by ϵ).

Using PAA, compressed MBTS representations are stored in each node, with the segmentation factor being determined by the number m of segments. Inevitably, this also reduces to some extent the resolution of the bounds, since now each MBTS is based on mean values. This implies a trade-off between index size and pruning effectiveness. Nevertheless, in our experiments (see Section 6), we are able to significantly reduce the index size with negligible performance drop.

5.5 Bulk Loading

The process described in Section 5.2 assumes that the index is built by inserting each subsequence individually and according to a predetermined order, typically the order in which they are extracted from the input time series T. However, building the index in this manner is slow, due to the overhead from node splits. Splitting nodes is computationally expensive, as it requires calculating multiple pairwise distances among subsequences, as well as updating the MBTS of the involved nodes.

However, we can accelerate index construction through *bulk-loading*. The idea is to follow a *bottom-up* processing in three stages: (a) re-order the subsequences based on similarity; (b) insert the subsequences in this order to populate all the leaf nodes; (c) construct the internal nodes at each level in a bottom-up fashion to complete the tree structure.

Algorithm 2: `BulkLoading`

Input : Time series T, subsequence length l, node capacity μ_c and M_c
Output: TS-Index I
$L \leftarrow \emptyset$
foreach $T_{p,l} \in T$ **do**
    $S \leftarrow T_{p,l}$
    $V_S \leftarrow \texttt{generateEmbedding}(S)$
    $Z_S \leftarrow \texttt{getZcode}(S)$
    $L \leftarrow L \cup \{S\}$ (sorted by Z_S)
$\mathcal{N}_L \leftarrow \emptyset$
$count \leftarrow 0$
$N \leftarrow \texttt{newNode}()$
foreach $S \in L$ **do**
    if $\mathit{requireNewNode}(\mu_c, M_c, S, N) == True$ **then**
        $\texttt{calculateMBTS}(N)$
        $\mathcal{N}_L \leftarrow \mathcal{N}_L \cup \{N\}$
        $N \leftarrow \texttt{newNode}()$
        $count = 0$
    $N \leftarrow N \cup \{S\}$
    $count$++
$\mathcal{I} \leftarrow \texttt{createParentNodes}(\mathcal{N}_L, M_c)$
return $\mathcal{I}$
Procedure $\mathit{createParentNodes}(\mathcal{N}, M_c)$
    $\mathcal{N}^{curr} \leftarrow \emptyset$
    $N^{curr} \leftarrow \texttt{newNode}()$
    $count \leftarrow 0$
    foreach $N \in \mathit{topLevel}(\mathcal{N})$ **do**
        if $\mathit{requireNewNode}(\mu_c, M_c, S, N) == True$ **then**
            $\texttt{calculateMBTS}(N^{curr})$
            $\mathcal{N}^{curr} \leftarrow \mathcal{N}^{curr} \cup \{N^{curr}\}$
            $N^{curr} \leftarrow \texttt{newNode}()$
            $count = 0$
        $N^{curr} \leftarrow N^{curr} \cup \{N\}$
        $N.parent \leftarrow N^{curr}$
        $count + +$
    if $|\mathcal{N}^{curr}| > 1$ **then** $\texttt{createParentNodes}(\mathcal{N}^{curr}, M_c)$
    else return $\mathcal{N}^{curr}$

The main challenge is how to determine the insertion order of subsequences so that the resulting nodes have tight MBTS. To this end, we use a *Space Filling Curve* (SFC) to map each subsequence to a 1-dimensional point, such that similar subsequences are more likely to be mapped to nearby points. Specifically, we use the Z-order curve [29], as it can be very efficiently computed, while providing a good approximation, as indicated by our experiments.

However, since subsequences have a relatively high number of dimensions, directly applying a Z-order curve on them to reduce the dimensions from l to 1, results in low accuracy. To avoid this, we first embed each subsequence S to a 5-dimensional vector $V_S = [\mu, v_{min}, v_{max}, p_{min}, p_{max}]$, where μ is the mean value of S, v_{min} and v_{max} are its minimum and maximum values, and p_{min} and p_{max} are the respective timestamp positions in which these extreme values are observed. The intuition is that, if two subsequences have a similar mean, maximum and minimum values, and those maximum and minimum also appear close in time, then their Chebyshev distance is more likely to be smaller. Finally, we traverse these 5-dimensional points according to the Z-order curve to determine the insertion order of subsequences into the index.

Algorithm 2 describes the bulk-loading process. We start by extracting all subsequences of length l from the input time series T. For each subsequence S, we compute its embedding V_S and use it to determine its Z-order code Z_S. All subsequences are then inserted to a list according to this order (Lines 2-6). Then, we populate the leaf nodes by inserting the subsequences in this order (Lines 10-17).

Clearly, there is a trade-off when building the tree: placing less subsequences per leaf tends to yield tighter MBTS, but creates many more nodes overall. Ideally, we would like to populate each leaf above its minimum capacity μ_c, and stop further insertions once the next subsequence expands the existing MBTS significantly, even before reaching the maximum capacity M_c. Towards this, we employ a heuristic based on a dynamically adjusted "jump" threshold that can assess such changes between consecutive Z-order code differences. While a leaf or internal node is being updated with new entries, we maintain the range of their corresponding Z-order codes. Thus, we can calculate a "jump" threshold by dividing this range by $(\mu_c + M_c)/2$, assuming that nodes in the constructed TS-Index will be half full on average. If the difference in Z-order codes between the last subsequence S' added to this node and the next one S exceeds this threshold, a new node is created with S as its first entry (Line 11). Thus, two successive, but dissimilar subsequences get indexed in separate nodes, keeping their bounds tighter, as well as those in parent nodes at upper levels.

Once all leaf nodes are populated, we compute their corresponding MBTS, and we recursively create the internal nodes at each level (Line 18). We iterate over the nodes of the previous level (Line 24) and add them to a new node, which becomes their parent node (Lines 30-31). When a new node is needed (Line 25), we calculate the MBTS of the new node and add it to the list of nodes of the current level (Lines 25-29). If there are more than one nodes at the current level, we recursively invoke the same function to create their parent nodes for the next level upwards. Once a single node is obtained at a given level, it becomes the root (Lines 33-34).

5.6 Extension to k-Nearest Neighbor Queries

So far, we have assumed that twin subsequences are found according to a user-specified distance threshold ϵ. In practice, selecting appropriate distance thresholds is often not straightforward. Next, we explain how TS-Index can also process kNN queries, where the user only specifies the number k of subsequences to be returned as most similar to a query subsequence Q under the Chebyshev distance.

Algorithm 3 outlines the process. The input includes the query sequence Q, the constructed TS-Index, the input time series T and the parameter k. A priority queue R, sorted by distance, holds the resulting subsequences (Line 1). The algorithm recursively traverses the index in a depth-first manner starting from the root (Line 2). For each current node, we check whether it is a leaf or not. If it is an internal node (Lines 13-25), we employ a local priority queue P_{local} to keep its children sorted by their Chebyshev distance to the query. If we already have k resulting subsequences (Line 15), we only need to push the child nodes that are closer than the k-th closest element (Lines 16, 17); in this case, we use early abandoning (if the values are z-normalized) to discard a node as soon as the value difference exceeds the distance of the k-th closest element in at least one timestamp. After inserting the node's children to the priority queue, we iterate

Algorithm 3: `TwinKNNSearch`

Input : Time series T, TS-Index I, query sequence Q, integer k
Output: List R of the k most similar twin subsequences to Q

1 $R \leftarrow \emptyset$
2 KNNTraversal($I.root, T, R, Q, k$)
3 **Procedure** *KNNTraversal*(N, T, R, Q, k)
4   **if** N *is* $leaf$ **then**
5   **foreach** $p \in N.getPositions()$ **do**
6   **if** $R.getSize() == k$ **then**
7   **if** $d(Q, T_{p,l}) < R.peekLast().dist$ **then**
8   R.pollLast()
9   R.push($T_{p,l}$, d($Q, T_{p,l}$))
10   **else**
11   R.push($T_{p,l}$, d($Q, T_{p,l}$))
12   **else**
13         $P_{local} \leftarrow \emptyset$
14   **foreach** $N' \in N.getChildren()$ **do**
15   **if** $R.getSize() == k$ **then**
16   **if** $d(Q, N'.MBTS) < R.peekLast().dist$ **then**
17   P_{local}.push(N', d(($Q, N'.MBTS$)))
18   **else**
19   P_{local}.push(N', d(($Q, N'.MBTS$)))
20   **foreach** $N'' \in P_{local}$ **do**
21   **if** $R.getSize() == k$ **then**
22   **if** $d(Q, N''.MBTS) < R.peekLast().dist$ **then**
23   KNNTraversal(N'', T, R, Q, k)
24   **else**
25   KNNTraversal(N'', T, R, Q, k)
26 **return** R

over it and recursively traverse the tree starting from the closest element (Lines 21-25). As above, when the result list size is equal to k, we can prune nodes (Lines 21-23). If the current node is a leaf (Lines 5-11), we push each qualifying subsequence contained in the current node to the results list R (Lines 10-11). If we already have k resulting subsequences (Line 6), we only keep subsequences that are closer than the k-th closest element, which gets evicted (Lines 7-9). In this case, we also use early abandoning (if the values are z-normalized) to discard non-qualifying subsequences faster.

6 Experimental Evaluation

Next, we present a comprehensive evaluation of our methods against four real-world and one synthetic dataset.

6.1 Experimental Setup

6.1.1 Datasets

We performed experiments against the time series listed in Table 2, which contain diverse patterns and differ in their total duration [1]. In particular:

Insect Movement [21]: 64,436 insect telemetry readings spanning around 30 minutes (36 readings/sec).

Electroencephalography (EEG) [21]: 1,801,999 EEG readings at 500Hz lasting one hour.

Electrooculography (EOG) [20]: 8,099,500 readings of the electrical potential between front and back of a human eye.

1. We have also performed similar experiments on all time series from the UCR Archive (https://www.cs.ucr.edu/%7Eeamonn/time_series_data_2018/). The source code, instructions, and the full list of results are publicly available at https://github.com/chgeorgakidis/ts_index

TABLE 2
Datasets and distance thresholds.

Dataset	n	ϵ (norm)	ϵ (non-norm)
Insect	64,436	0.5,**0.75**,1,1.25,1.5	50, **100**,150,200,250
EEG	1,801,999	0.1,0.2,**0.3**,0.4,0.5	20, **40**,60,80,100
EOG	8,099,500	0.1,0.2,**0.3**,0.4,0.5	3,**4**,5,6,7
ECG	20,140,000	0.01,0.02,0.03,**0.04**,0.05	0.03,0.04,**0.05**,0.06,0.07
Synthetic	100,000,000	**0.5**	**N/A**

TABLE 3
Other parameters.

Parameter	*Value*
Number m of segments	5, **10**, 20, 25, 50
Subsequence length l	50, **100**, 150, 200, 250
Query length l'	**100**, 200, 300, 400, 500
Count k of results in kNN queries	10, 20, 30, 40, 50

Electrocardiography (ECG) [27]: 20,140,000 ECG data points recorded at 256Hz, lasting 22 hours and 23 minutes.

Synthetic. To examine scalability, we generated a synthetic time series by extending the EOG data. Specifically, we appended replicas of the original EOG time series, after randomly altering each data point by up to 25% of the time series standard deviation. In total, we obtained a synthetic series of 100,000,000 points.

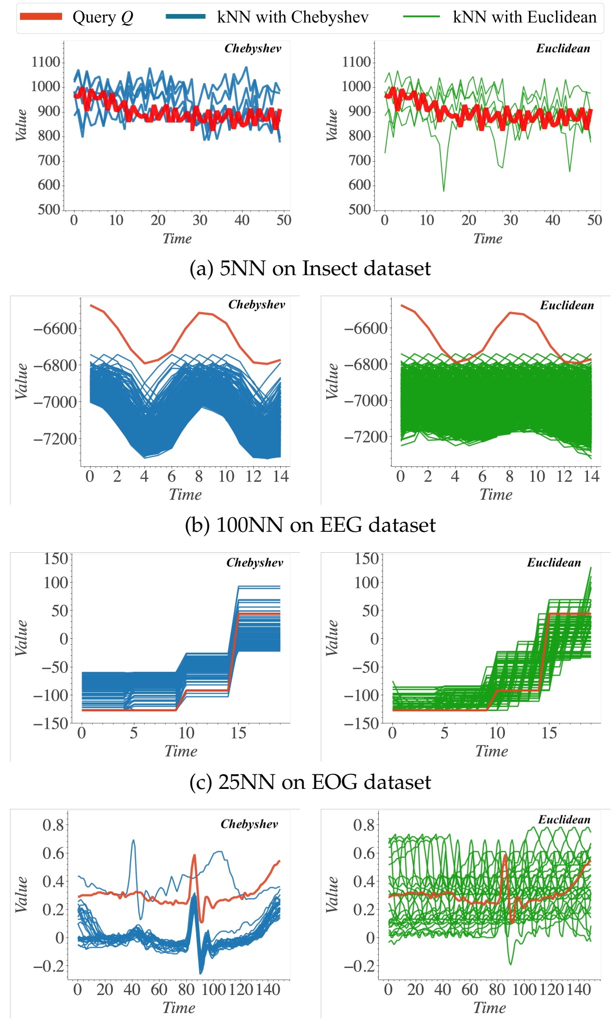
Unless stated otherwise, we z-normalize the time series to facilitate selection of distance thresholds.

6.1.2 Parameters

Table 2 indicates the different values for the distance threshold ϵ used in the experiments against each dataset, for z-normalized (*norm) or original values (*non-norm*)*. Note that choosing a value for ϵ is more intuitive for z-normalized values as it is expressed in standard deviations. Table 3 contains the values for subsequence length l, query length l' and number of segments m, which are common in the experiments on all datasets, as well as the number k of results to fetch for kNN queries. Default values have been selected by running preliminary tests, and are shown in bold. In particular, the default subsequence length l is set to the minimum length of queries in the workload. Moreover, for iSAX, the maximum node capacity is set to 10,000 to enable index construction in reasonable time even for larger datasets. The default values for minimum and maximum node capacity in TS-Index are set to $\mu_c = 10$ and $M_c = 30$, respectively.

6.1.3 Methodology

For each dataset, we randomly picked 100 subsequences, each of length $l = 100$ points, and used them as the query workload in all tests against that dataset. We report average response time per query (in milliseconds). We implemented all methods, including KV-Index, iSAX, and TS-Index, in Java. In all implementations, the structure of the index is kept in memory, while the original input dataset is stored on disk. Leaf nodes in the index contain the starting positions of the subsequences in the input time series. Thus, when a leaf is reached at query time, its corresponding subsequences are obtained from the input time series file using random access. All experiments were conducted on a server with 4 CPUs, each equipped with 8 cores clocked at 2.13GHz, and 256 GB RAM running Debian Linux.



(a) 5NN on Insect dataset

(b) 100NN on EEG dataset

(c) 25NN on EOG dataset

(d) 25NN on ECG dataset

Fig. 4. Top-k similar subsequences.

6.2 Case Study

We first provide some intuition on the qualitative differences in the kNN results obtained when using Chebyshev distance and Euclidean distance. For the Insect dataset (Figure 4a), some of the matching subsequences returned with Euclidean distance (right column) contain spikes that deviate significantly from the value of the query sequence at the respective timestamp. Instead, the results retrieved with Chebyshev distance (left column) closely match the query across all timestamps. Regarding the EEG, EOG and ECG datasets (Figures 4b, 4c and 4d, respectively), we observe that the resulting subsequences using Chebyshev distance tend to follow the pattern of the query more closely compared to those obtained with Euclidean distance.

Next, we measure how much the result sets differ when executing both range and kNN queries against each dataset in Section 6.1, using either Chebyshev or Euclidean distance. For range queries, we examine whether it is feasible to discover approximately the same twin subsequences by appropriately tuning a Euclidean distance threshold. For each dataset, we first execute twin subsequence search using a threshold ϵ on Chebyshev distance. Then, we repeat the process using Euclidean distance and varying the threshold from $\epsilon \times \sqrt{|Q|}$ to ϵ (see Lemma 1), reducing it by a specific

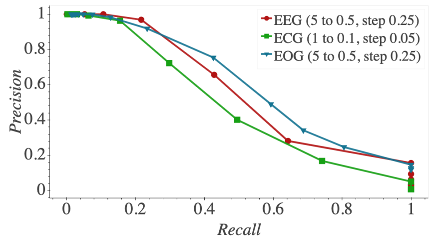


Fig. 5. Precision/recall for various distance thresholds.

TABLE 4
Average correlation among kNN rankings.

Dataset	*k value*				
	10	20	30	40	50
Insect	0.1534	0.1481	0.0527	0.0454	0.0272
EEG	0.3761	0.1615	0.1071	0.0716	0.0558
EOG	0.2726	0.1959	0.1406	0.1144	0.0937
ECG	0.1495	0.1287	0.0995	0.0882	0.0595

step each time. Figure 5 shows the resulting recall-precision diagram for each dataset. The corresponding ϵ threshold ranges and step for Euclidean distance are indicated in the legend. As we can see, when starting from a low threshold that ensures high precision (above 0.8), recall is very low (below 0.3), meaning that there is a very high number of false negatives; on the other hand, as we increase the threshold, recall eventually reaches 1 but precision drops below 0.4, meaning that the majority of the results are false positives.

For kNN queries, we use Spearman's rank correlation coefficient to measure the correlation between the two lists of k nearest neighbors sorted by Chebyshev and Euclidean distance, respectively. For different values of k, we average the results over 100 randomly selected query subsequences. The obtained values for Insect, EEG, EOG and ECG are shown in Table 4, indicating very low correlation in all cases.

6.3 Performance

We compare the average execution time per query for varying values of each parameter, setting the rest to their default values.

6.3.1 Effect of bulk loading on TS-Index

We first examine the impact of bulk loading using the EEG dataset. Figure 6 compares the variant employing bulk loading (*TS-Index Bulk*) against the one (*TS-Index*) that inserts subsequences in the order they have been extracted from the input time series. To evaluate the behavior of the two variants, we examine query execution time, memory footprint and construction cost at different resolutions, i.e., by varying the number m of PAA segments (up to no compression at all, i.e., $l = m = 100$). We observe that both variants have similar memory footprint, as shown in Figure 6a. The size of each index grows as the number of segments increases. Indicatively, when using only $m = 50$ segments (i.e., each spanning a pair of successive time points), the index size is reduced by approximately 40%. Of course, the finer the segmentation, the more detailed the resolution of MBTS per node. Then, query execution time improves by more than an order of magnitude as Figure 6b testifies. The trends in performance are roughly similar for both TS-Index and TS-Index Bulk. However, with a coarser segmentation (5, 10, and 20 segments), bulk loading gives a further advantage to the index. This indicates that MBTS generated using embeddings after sorting and grouping the subsequences are tighter even at lower resolutions. Naturally, this difference is gradually eliminated as we further increase the number of segments. Then, subsequence representation becomes too detailed and little deviation should be expected in values among those subsequences grouped together according to the original ordering. Thus, the resulting MBTS get almost as tight as those derived from the embeddings.

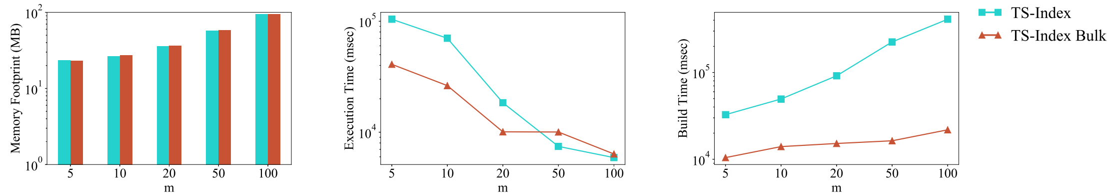
Nevertheless, the two TS-Index variants differ significantly in their construction cost, as illustrated in Figure 6c. When subsequences are inserted in the order they are extracted from the input dataset, index construction takes much longer as the number of segments increases. Indeed, finer segmentation implies more node splits, which become very expensive across many dimensions. In contrast, no node splits occur in TS-Index Bulk, which reduces construction time by orders of magnitude compared to TS-Index. Due to its bottom-up construction, MBTS of nodes at a given level in TS-Index only need to combine the MBTS of their descendants, which is far less costly than node splitting. Given that TS-Index with bulk loading is advantageous in terms of construction time, and answers queries faster, in the sequel we only compare this variant against other approaches.

6.3.2 Varying threshold ϵ

Figure 7 depicts query execution time (in logarithmic scale) for varying threshold ϵ. As expected, searching with the Sweepline approach has a fixed cost per dataset regardless of ϵ, since it needs to scan all subsequences extracted from the input time series. Relaxing the threshold incurs an overhead when an index is involved. Queries against KV-Index perform poorly compared to other indices, since filtering based on mean values achieves less pruning. Searching with TS-Index outperforms the rest in every setting for all tested datasets, with the only exception being iSAX for $\epsilon = 0.01$ in the ECG dataset. Overall, TS-Index is at least an order of magnitude more efficient in twin subsequence search compared to the KV-Index and Sweepline approaches. It is also consistently better than iSAX as it is less susceptible to fluctuations in the input sequences.

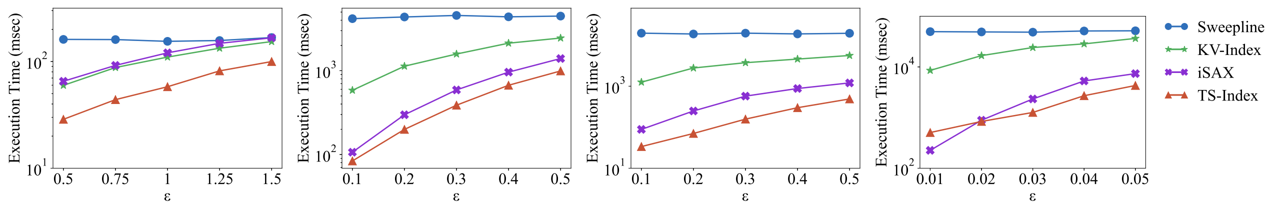
6.3.3 Varying Number of Segments

This experiment involves iSAX and TS-Index, since only these indices apply segmentation of subsequences. Figure 8 plots performance results for varying number of segments. Again, TS-Index outperforms iSAX in almost every setting, except for the coarsest segmentation in the ECG dataset. Finer segmentation generally improves performance of TS-Index, as its inner nodes contain more detailed MBTS and can prune the search space faster. Yet, for the finest segmentation tested ($m = 25$), the cost increases for the EOG and ECG datasets, indicating a trade-off between the segmentation and actual performance. This is due to the PAA being applied on the MBTS of TS-Index, reducing its memory footprint, but allowing pruning on less nodes. The more the segments, the better the pruning, but also



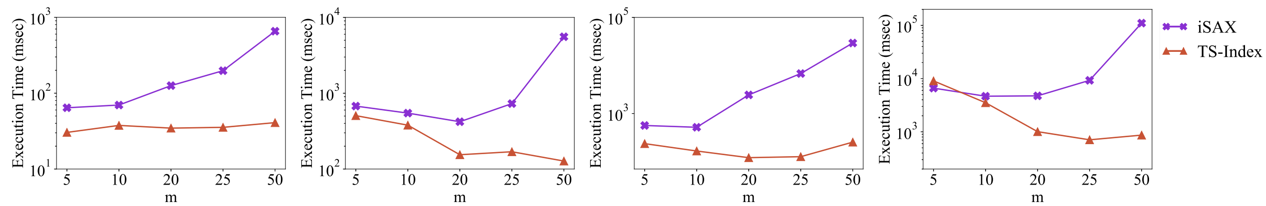
(a) Memory footprint (b) Query execution cost (c) Index construction cost

Fig. 6. Performance of TS-Index with and without bulk loading against EEG dataset for varying number of segments m.



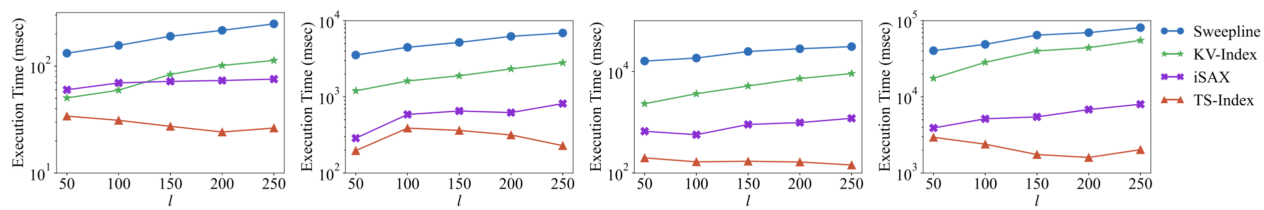
(a) Insect Dataset (b) EEG Dataset (c) EOG Dataset (d) ECG Dataset

Fig. 7. Performance results with varying distance threshold ϵ.



(a) Insect Dataset (b) EEG Dataset (c) EOG Dataset (d) ECG Dataset

Fig. 8. Performance results for varying number m of segments.



(a) Insect Dataset (b) EEG Dataset (c) EOG Dataset (d) ECG Dataset

Fig. 9. Performance results for varying subsequence length l (query length $l' = l$).

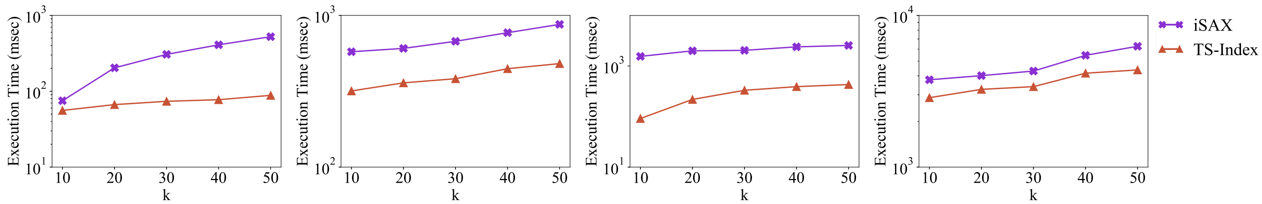
more time points have to be checked per node. Thus, increasing the number of segments up to a certain value can be beneficial; past that number, performance deteriorates. Interestingly, too fine segmentation seems to harm iSAX even more, as in most cases execution cost significantly worsens with more than 10 segments. With 25 segments, TS-Index responds more than an order of magnitude faster compared to iSAX. In general, if index size is not critical, the default number of segments can be equal to the subsequence length ($m = l$), i.e., not applying PAA at all.

6.3.4 Varying Subsequence Length

Figure 9 plots performance results with a varying length l for subsequences obtained from the input time series. In all cases, the query length is set to be equal to l. Increasing l seems to slightly negatively affect all approaches, except for TS-Index. Since longer subsequences are extracted, more checks are required, both in nodes (in case of iSAX) and raw subsequences during verification. Instead, TS-Index is faster when longer subsequences are specified, as it becomes less likely to find matching twins. Thanks to the Chebyshev distance, TS-Index has more pruning capability and can skip non-qualifying subtrees earlier at higher levels in the tree hierarchy. Thus, much fewer leaf nodes are accessed and need to be verified, sparing much of the verification cost for checks per timestamp.

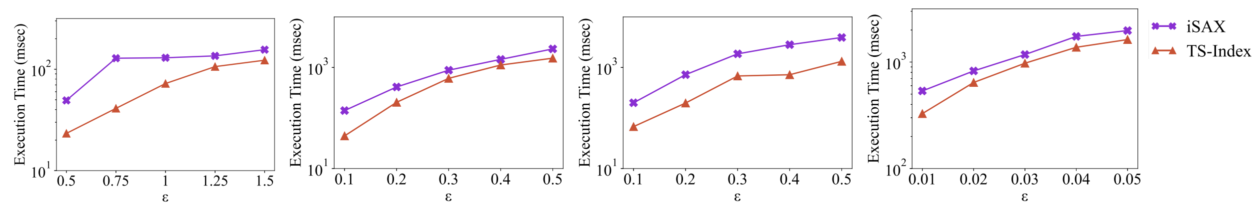
6.3.5 Varying Parameter k for kNN Queries

Figure 10 compares performance of twin subsequence kNN queries against TS-Index and iSAX for different values of k. TS-Index outperforms iSAX in every setting, with smaller differences noticed for the ECG dataset (Figure 10d), which is in line with the previous experiments. Increasing k has a negative impact on performance in most cases, except for



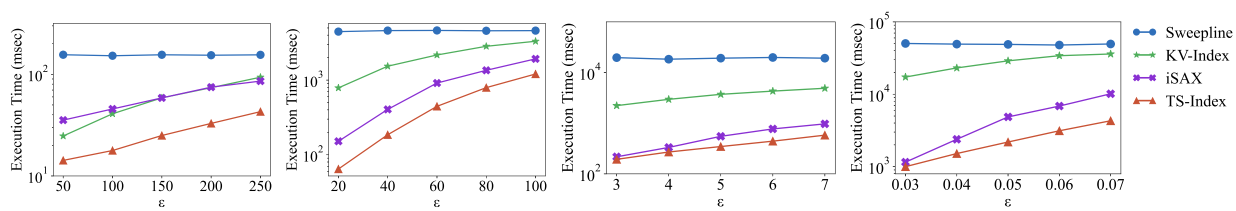
(a) Insect Dataset (b) EEG Dataset (c) EOG Dataset (d) ECG Dataset

Fig. 10. Performance of kNN search for varying values of parameter k.



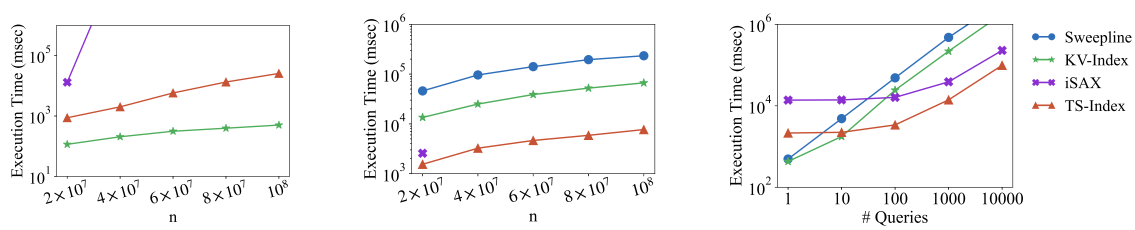
(a) Insect Dataset (b) EEG Dataset (c) EOG Dataset (d) ECG Dataset

Fig. 11. Performance results for varying distance threshold ϵ on z-normalized subsequences.



(a) Insect Dataset (b) EEG Dataset (c) EOG Dataset (d) ECG Dataset

Fig. 12. Performance results for varying distance threshold ϵ on non-normalized data.



(a) Index construction time (b) Average query response time (c) Total cost

Fig. 13. Scalability results against the synthetic dataset for various time series lengths (n) and query workloads.

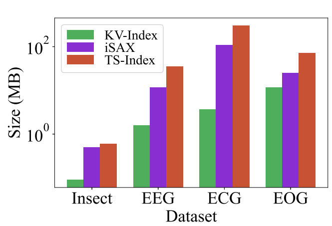


Fig. 14. Memory footprint per index constructed for the various datasets.

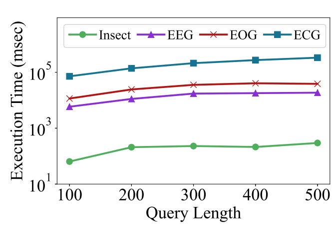


Fig. 15. Performance for varying query length l' (all indices built with $l = 100$).

iSAX on the EOG dataset (Figure 10c), where performance seems rather stable. This could be due to the rather large leaf sizes in iSAX, which increases the probability of finding (almost) all k results in a single leaf node, thus requiring less node accesses for k values of the same order of magnitude.

6.3.6 Searching over z-normalized subsequences

We repeat the experiment for varying distance threshold ϵ, this time applying z-normalization over each individual subsequence, before inserting it in the index. As mentioned in Section 4.1, KV-Index cannot be built on such data since the mean value per subsequence would always be zero; thus, we only compare TS-Index with iSAX. The results are depicted in Figure 11. Clearly, z-normalizing the subsequences separately has no significant effect on the performance of TS-Index; the results are similar to those in Figure 7, with TS-Index outperforming iSAX in all cases.

6.3.7 Searching on Non-Normalized Data

Query execution cost for identifying twin subsequences against the raw (non-normalized) time series is depicted in Figure 12. Some variations in performance are observed depending on the dataset characteristics. For instance, iSAX is closely competitive to TS-Index against the EOG data, but its performance worsens and resembles that of KV-Index against the Insect dataset. Overall, TS-Index copes better than all the rest even for raw data, confirming its suitability for twin subsequence search in various settings.

6.3.8 Varying Query Length

Figure 15 depicts the impact on performance when constructing each TS-Index with a fixed subsequence length $l = 100$ and then executing queries of varying length $l' \in [100, 500]$. As expected, the execution time increases when $l' > l$, since this translates to multiple queries, one for each consecutive part of the query. However, more pruning also occurs, since all parts need to satisfy the distance threshold, hence the overall extra cost is not very high.

6.3.9 Index Size

Figure 14 presents the memory footprint of TS-Index, iSAX and KV-Index for each dataset. KV-Index requires significantly less space than TS-Index and iSAX, as it only keeps in memory the mean value and position range per indexed subsequence. Instead, TS-Index and iSAX occupy more space due to their more complex structures. From these results, it turns out that iSAX requires two to three times less space than TS-Index. Indeed, iSAX needs to store one SAX word per node, whereas a node in TS-Index is represented by an MBTS, hence its increased memory footprint. Nevertheless, even with tens of millions of subsequences indexed (e.g., for the larger ECG dataset), all indices, including TS-Index, have sizes that easily fit in main memory.

6.3.10 Scalability

Finally, we examine the scalability of the various methods against the *synthetic* dataset. Figure 13a depicts the index construction time for various subsets of the dataset, ranging from 20 to 100 million points. KV-Index can be constructed fast even for larger time series, as it only needs to calculate and store a mean value per subsequence. TS-Index, despite its complex hierarchical tree structure, can still be constructed within minutes even for the largest dataset (100 million points). In contrast, iSAX failed to build in reasonable time; in this test, it takes more than 5 hours to index the subsequences for input time series having more than 20 million points. We should note, however, that this may be a limitation of our implementation, which does not support bulk loading for iSAX.

Regarding execution cost, Figure 13b shows average query response times with varying data sizes. In this case, as the number of indexed subsequences also grows linearly, this has an impact on execution cost. Note that TS-Index is a clear winner regardless the data size, being at least one order of magnitude faster than the rest. iSAX seems close in terms of efficiency, but it could be built only for the smallest subset of the data, as mentioned before.

Figure 13c depicts the total time of each method for (i) building the index (if any) against the data subset of 20 million points and (ii) executing a workload with varying number of queries of the same length. For a single query, Sweepline and KV-Index are superior, since KV-Index is built very fast and Sweepline does not need to construct any index at all. However, as more queries are added to the workload, the total cost is dominated by query execution. Note that these two methods struggle to finish when the workload contains more than 1,000 queries. iSAX is advantageous over them when more than 100 queries are specified. However, TS-Index scales better with increasing query workloads. Indeed, it emerges as the most suitable solution when more than 10 twin subsequence search queries need to be answered. Hence, it is more affordable to build this index from scratch and utilize it to answer several queries than to employ any of its competitors.

7 Conclusions

We have introduced the twin subsequence search problem. Given a query sequence Q, an input time series T and a distance threshold ϵ, this task retrieves all subsequences in T with Chebyshev distance to Q not higher than ϵ. To answer this query efficiently, we have introduced the TS-Index, and proposed optimizations for reducing its memory footprint and improving its construction cost. We have also shown how TS-Index can be used for answering kNN queries, which is useful when specifying a distance threshold is not straightforward. Our extensive experimental evaluation confirms the superiority of TS-Index for twin subsequence search queries when compared to existing indices. In the future, we plan to extend our method to efficiently answer variable length queries of any size.

Acknowledgments

This work was supported by the EU H2020 project SmartDataLake (825041), the EU H2020 project OpertusMundi (870228) and the NSRF 2014-2020 project HELIX (5002781).

References

[1] A. Camerra, T. Palpanas, J. Shieh, and E. J. Keogh, "iSAX 2.0: Indexing and mining one billion time series," in *ICDM*, 2010, pp. 58–67.
[2] 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 i SAX2+," *KAIS*, vol. 39, no. 1, pp. 123–151, 2014.
[3] K. Chan and A. W. Fu, "Efficient time series matching by wavelets," in *ICDE*, 1999, pp. 126–133.
[4] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, S. Athanasiou, and S. Skiadopoulos, "Indexing geolocated time series data," in *SIGSPATIAL*, 2017, pp. 19:1–19:10.
[5] G. Chatzigeorgakidis, D. Skoutas, K. Patroumpas, T. Palpanas, S. Athanasiou, and S. Skiadopoulos, "Local pair and bundle discovery over co-evolving time series," in *SSTD*, 2019, pp. 160–169.
[6] ——, "Twin subsequence search in time series," in *EDBT*, 2021, pp. 475–480.
[7] S. Demirci, I. Erer, and O. Ersoy, "Weighted Chebyshev distance classification method for hyperspectral imaging," in *Next-Generation Spectroscopic Technologies VIII*, vol. 9482. SPIE, 2015, pp. 314 – 320.
[8] 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*, vol. 1, no. 2, pp. 1542–1552, 2008.
[9] 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," *PVLDB*, vol. 12, no. 2, pp. 112–127, 2018.
[10] B. Ghazal, K. ElKhatib, K. Chahine, and M. Kherfan, "Smart traffic light control system," in *EECEA*. IEEE, 2016, pp. 140–145.

[11] A. Graps, "An introduction to wavelets," *CiSE*, vol. 2, no. 2, pp. 50–61, 1995.

[12] A. Haar, "Zur theorie der orthogonalen funktionensysteme," *Math. Ann.*, vol. 69, no. 3, pp. 331–371, 1910.

[13] S. Kashyap and P. Karras, "Scalable kNN search on vertically stored time series," in *SIGKDD*, 2011, pp. 1334–1342.

[14] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *KAIS*, vol. 3, no. 3, pp. 263–286, 2001.

[15] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas, "Coconut: A scalable bottom-up approach for building data series indexes," *PVLDB*, vol. 11, no. 6, pp. 677–690, 2018.

[16] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi, "Experiencing SAX: a novel symbolic representation of time series," *Data Min. Knowl. Discov.*, vol. 15, no. 2, pp. 107–144, 2007.

[17] R. A. K.-l. Lin and H. S. S. K. Shim, "Fast similarity search in the presence of noise, scaling, and translation in time-series databases," in *VLDB*, 1995, pp. 490–501.

[18] M. Linardi and T. Palpanas, "Scalable, variable-length similarity search in data series: The ULISSE approach," *PVLDB*, vol. 11, no. 13, pp. 2236–2248, 2018.

[19] ——, "Scalable data series subsequence matching with ULISSE," *VLDBJ*, vol. 11, no. 13, pp. 2236–2248, 2020.

[20] A. Mueen and E. Keogh, "Online discovery and maintenance of time series motifs," in *SIGKDD*, 2010, pp. 1089–1098.

[21] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover, "Exact discovery of time series motifs," in *SIAM*, 2009, pp. 473–484.

[22] T. Palpanas, "Evolution of a Data Series Index," *CCIS*, vol. 1197, 2020.

[23] B. Peng, P. Fatourou, and T. Palpanas, "Paris: The next destination for fast data series indexing and query answering," in *IEEE BigData*, 2018.

[24] ——, "MESSI: In-memory data series indexing," in *ICDE*, 2020, pp. 337–348.

[25] T. Penzel, J. McNames, P. De Chazal, B. Raymond, A. Murray, and G. Moody, "Systematic comparison of different algorithms for apnoea detection based on electrocardiogram recordings," *MBEC*, vol. 40, no. 4, pp. 402–407, 2002.

[26] I. Popivanov and R. J. Miller, "Similarity search over time-series data using wavelets," in *ICDE*, 2002, pp. 212–221.

[27] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *SIGKDD*, 2012, pp. 262–270.

[28] M. J. Ribeiro, I. R. Violante, I. Bernardino, R. A. Edden, and M. Castelo-Branco, "Abnormal relationship between GABA, neurophysiology and impulsive behavior in neurofibromatosis type 1," *Cortex*, vol. 64, pp. 194–208, 2015.

[29] H. Sagan, *Space-filling curves*. Springer-Verlag New York, 1994.

[30] J. Shieh and E. J. Keogh, "*i*SAX: indexing and mining terabyte sized time series," in *SIGKDD*, 2008, pp. 623–631.

[31] H. Sivaraks and C. A. Ratanamahatana, "Robust and accurate anomaly detection in ecg artifacts using time series motif discovery," *CMMM*, vol. 2015, 2015.

[32] J. Wu, P. Wang, N. Pan, C. Wang, W. Wang, and J. Wang, "KV-Match: A subsequence matching approach supporting normalization and time warping," in *ICDE*, 2019, pp. 866–877.

[33] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Massively distributed time series indexing and querying," *TKDE*, vol. 32, no. 1, pp. 108–120, 2020.

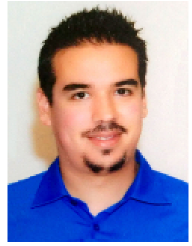
[34] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh, "Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets," in *ICDM*, 2016.

[35] B. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary Lp norms," in *VLDB*, 2000, pp. 385–394.

[36] J. Zhang, P. G. Richards, and D. P. Schaff, "Wide-scale detection of earthquake waveform doublets and further evidence for inner core super-rotation," *Geophys. J. Int.*, vol. 174, no. 3, pp. 993–1006, 2008.

[37] J. Zhang, X. Song, Y. Li, P. G. Richards, X. Sun, and F. Waldhauser, "Inner core differential motion confirmed by earthquake waveform doublets," *Science*, vol. 309, no. 5739, pp. 1357–1360, 2005.

[38] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for interactive exploration of big data series," in *SIGMOD*, 2014, pp. 1555–1566.



Georgios Chatzigeorgakidis Georgios Chatzigeorgakidis is a postdoc researcher at the Information Management Systems Institute of Athena Research Center. He acquired his diploma at the department of Electronics and Computer Engineering of TUC, Greece, his MSc degree at the department of Computer Science and Engineering of DTU, Denmark and his PhD degree at the Department of Informatics and Telecommunications of UoP, Greece.

Dimitrios Skoutas is a Principal Researcher at the Information Management Systems Institute of Athena Research Center. He received his Diploma and PhD in Electrical and Computer Engineering at the National Technical University of Athens, Greece, and has worked as a postdoctoral researcher at the L3S Research Center, Germany. His research interests include big data integration and mining, having more than 70 publications in these areas.

Kostas Patroumpas joined the Information Management Systems Institute at Athena Research Center in 2012 and has collaborated in many research projects. Previously, he worked in the industry as developer, IT manager, and GIS consultant. He has served on the program committees of several conferences and has more than 60 publications in the areas of data stream processing, trajectory data management, spatial analytics, and geospatial data integration.

Themis Palpanas is Director of the Data Intelligence Institute of Paris (diiP), Senior Member of the French University Institute (IUF), and Professor of computer science at the University of Paris (France). He is the author of 9 US patents, the recipient of 3 Best Paper awards, and the IBM SUR Award. He has served as Editor in Chief for BDR Journal, General Chair for VLDB 2013, and Associate Editor for TKDE and DSE journals, as well as PVLDB 2022, 2019 and 2017.

Spiros Athanasiou Spiros Athanasiou is a Research Associate and Project Manager at the Information Management Systems Institute of Athena Research Center. He received his diploma in Electrical Engineering at NTUA, Greece, and has worked as a researcher, advisor and project manager in R&I projects of the public and private sector. His research interests include (among others) Big Data and Semantic Web infrastructures.

Spiros Skiadopoulos is a Professor at the Dept. of Informatics and Telecommunications at University of the Peloponnese and the director of the M.Sc. program in Data Science. He has served in the program committee of several venues and participated in various research and development projects. His scientific contribution received a large number of citations. He obtained his PhD degree from the National Technical University of Athens (NTUA) and his MPhil degree from the Manchester Institute of Science and Technology (UMIST). More information is available at www.uop.gr/~spiros.