Karlsruhe Institute of Technology

# Efficient *k*-NN Search of Time Series in Arbitrary Time Intervals

Master's Thesis of

Janek Bettinger

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

| | |
|---|---|
| Reviewer: | Prof. Dr.-Ing. Klemens Böhm |
| Second reviewer: | Jun.-Prof. Dr.-Ing. Anne Koziolek |
| Advisor: | Jens Willkomm, M.Sc. |

September 1, 2017 – February 28, 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

**Karlsruhe, February 28, 2018**

. . . . . .

(Janek Bettinger)

# Abstract

The $k$ nearest neighbors ($k$-NN) of a time series are the $k$ closest sequences within a dataset regarding a distance measure. Often, not the entire time series, but only specific time intervals are of interest, e.g., to examine phenomena around special events. While numerous indexing techniques support the $k$-NN search of time series, none of them is designed for an efficient interval-based search. This work presents the novel index structure *Time Series Envelopes Index Tree* (TSEIT), that significantly speeds up the $k$-NN search of time series in arbitrary user-defined time intervals. The basic idea is to store groups of similar time series in the leaf nodes of a height-balanced tree. Each group is represented by an envelope that tightly wraps the group's time series. Hence, an envelope is formed by the maximum and the minimum values of the enclosed time series. The inner nodes hold envelopes that tightly wrap the envelopes of their child nodes. For a faster index lookup and a reduction of the index size, the envelopes are stored with reduced dimensionality. The dimensionality is increased with deeper tree levels to improve the accuracy of distance calculations. The tree structure enables TSEIT to exclude entire subtrees during the $k$-NN search without any false dismissals. Consequently, the algorithm needs to examine only a fraction of all time series. TSEIT uses dynamic time warping (DTW), which is one of the best distance measures for time series. The evaluation proves that TSEIT significantly outperforms rival techniques regarding querying effort and the amount of data which it can handle. It also shows that TSEIT is able to index a real-world dataset with more than 100 million time series effortlessly. For a particular set of $k$-NN queries on this massive dataset, TSEIT computes the DTW distance to merely 0.006% of all time series on average.

# Zusammenfassung

Die $k$ nächsten Nachbarn (KNN) einer Zeitreihe sind die $k$ Sequenzen innerhalb eines Datensatzes mit der geringsten Distanz zur entsprechenden Zeitreihe. Oftmals sind nicht die vollständigen Zeitreihen von Interesse, sondern lediglich bestimmte Zeitintervalle; etwa, um Phänomene rundum besondere Ereignisse zu untersuchen. Zwar gibt es zahlreiche Indexstrukturen, die eine KNN-Suche von Zeitreihen ermöglichen, jedoch ist keine auf eine effiziente intervallbasierte Suche ausgelegt. Diese Arbeit präsentiert die neuartige Indexstruktur *„Time Series Envelopes Index Tree“* (TSEIT), welche die KNN-Suche von Zeitreihen in beliebigen Zeitintervallen signifikant beschleunigt. Die Grundidee ist, Gruppen ähnlicher Zeitreihen in den Blattknoten eines höhenbalancierten Baumes zu speichern. Jede Gruppe wird dabei durch einen Umschlag repräsentiert, der die jeweiligen Zeitreihen eng umschließt. Ein Umschlag wird also durch die maximalen und minimalen Werte der enthaltenen Zeitreihen definiert. Die inneren Baumknoten speichern Umschläge, welche die Umschläge ihrer jeweiligen Kindknoten eng umschließen. Zur schnelleren Abfrage und Reduzierung der Indexgröße werden die Umschläge mit verringerter Dimensionalität gespeichert. Dabei ist die Dimensionalität in tieferen Baumebenen höher, um die Genauigkeit von Distanzberechnungen zu erhöhen. Die Baumstruktur ermöglicht es TSEIT, ganze Teilbäume bei der KNN-Suche fehlerfrei auszuschließen. Folglich muss der Suchalgorithmus nur einen Bruchteil aller Zeitreihen genauer untersuchen. TSEIT nutzt „Dynamic Time Warping“ (DTW), was als das beste Distanzmaß für Zeitreihen gilt. Die Evaluierung beweist, dass TSEIT konkurrierenden Techniken hinsichtlich Suchaufwand und Größe der Datensätze, die es verarbeiten kann, deutlich überlegen ist. Es wird gezeigt, dass TSEIT einen Datensatz mit mehr als 100 Millionen Zeitreihen mühelos indexieren kann. Bei einer Menge beispielhafter KNN-Anfragen zu diesem riesigen Datensatz, berechnet TSEIT die DTW-Distanz zu durchschnittlich lediglich 0,006% aller Zeitreihen.

# Contents

# List of Figures

# List of Equations

# 1 Introduction

Most data is inherently temporal and thus representable in the form of a time series. Nowadays, time series arise in almost every domain like industry or science in vast quantities. A time series might describe measured sensor values such as the temperature or energy usage over time. In medicine, the number of heartbeats or the oscillation of a brainwave can be represented as a time series. Further examples are sequences of stock prices or election polls at successive points in time.

Datasets themselves usually have no explicit value, but only the insights about real-world phenomena they reveal by analysis. Instead of entire time series, often only specific time intervals are of interest. Concerning the previous examples, this might be the temperature during a particular month or the energy usage between Christmas and New Year. Worth analyzing might also be the period just before a special event such as a heart attack or epileptic seizure, a market collapse or voting day. Moreover, as time series usually become longer over time due to continuous data acquisition, only the latest period might be appropriate for analysis. Another scenario could be the examination of separate time intervals to comprehend changes.

A basic operation in data analysis is obtaining the $k$ nearest neighbors ($k$-NN) of a given data object. The neighbors are those data objects that are closest to the query object concerning a distance measure. While the Euclidean distance is often used, the dynamic time warping (DTW) distance usually performs better for time series. The $k$-NN algorithm can be used for classification, clustering or regression, as well as for exploratory data analysis.

Usually, spatial index structures allow efficient $k$-NN querying without the need for a sequential scan through all data. Especially for time series, numerous techniques for indexing and querying have been developed in recent decades. However, they either operate on full time series only or retrieve similar subsequences at arbitrary positions within other time series. These methods are, however, not suitable for a $k$-NN search in specific time intervals. Most related approaches transform time series in a way that does not allow interval-based querying while guaranteeing the correctness of the result. Furthermore, they either require standardized value ranges or impose restrictions on the search intervals. Otherwise, they scale poorly with increasing data size.

This work proposes a novel index structure that allows an efficient $k$-NN search of time series in arbitrary time intervals. It guarantees an exact search without any restrictions. The main idea is to wrap groups of similar time series by so-called envelopes, which are conceptually comparable to minimum bounding rectangles. The envelopes themselves are

stored in a height-balanced tree structure, where the envelope of an inner node spans those of its child nodes. For a $k$-NN search given a query time series and a time interval, the index tree is traversed from the root node to find promising envelopes and time series at the leaf level. At each level, the distance between the query time series and each envelope is calculated to decide which subtrees to follow. A valuable property is that the distance to an envelope lower bounds the distance to each of the time series inside the envelope. This allows omitting entire subtrees below envelopes with a distance larger than the minimum distance so far, while still guaranteeing no false dismissals. Thus, the number of time series that need to be examined can be significantly reduced to a fraction of the total time series count. Experiments show that the proposed index structure can efficiently index and query more than 100 million time series, even for the computationally expensive DTW distance.

The remainder of this work is organized as follows. Chapter 2 presents related work including approaches that did not completely fulfill our demands. Chapter 3 introduces the fundamental concepts and techniques this work is based on. In Chapter 4, the novel index structure including the creation and query algorithm is explained, before implementation details are revealed in Chapter 5. Chapter 6 evaluates the proposed method using real-world datasets. Finally, Chapter 7 summarizes this work and suggests future work.

# 2 Related Work

This chapter introduces related and proven techniques for indexing and searching time series. It explains that current approaches are conceptually incompatible to interval-based querying, as they either work on whole time series or fixed-sized intervals. Most of the dimensionality reduction and indexing techniques introduced beyond, do not fit into the scenario of this work either.

## 2.1 Whole Matching and Subsequence Matching

Research mainly focuses on two different problems regarding similarity search of time series. While whole matching [YJF98; KPC01; SYF05; KR05; SH12] compares entire time series of equal length, subsequence matching [FRM94; KP99; KS01; AiJ+02; LPK07; Du+08; Rak+12; XC13; Gil+15] retrieves similar subsequences contained at arbitrary positions in other time series.

Subsequence matching is often converted to a whole matching problem by moving a sliding window over each time series and materializing the underlying subsequence. For example, a more or less individual index might be built for each position and size of the sliding window. However, this does not scale well with increasing number or length of the time series. Moreover, often not every possible window size is considered to keep the index size manageable.

As subsequence matching is more general than an interval-based search, it seems natural to use those techniques for the latter. However, the runtime and space complexity is too high to be applicable for large datasets. The high-performance and scalable querying approach proposed in [Rak+12], on the other hand, requires the standardization of the value range, making it less appealing for this work. Furthermore, any restrictions on the window size or interval for the $k$-NN search are not desired either.

An alternative application is the interval-based similarity search, where the target time intervals are fixed, which is similar to this work. While it is not that present yet, it gains in importance due to ever-increasing data acquisition times and longer time series. [Aßf+07] proposes an interval-based technique which, however, requires a costly transformation of each time series before indexing. Since it was evaluated with datasets holding a few thousand time series only, it does not seem to be suitable for processing massive datasets.

## 2.2 Dimensionality Reduction and Indexing of Time Series

A time series of length $n$ can be considered as a point in an $n$-dimensional space. However, multidimensional index structures or spatial access methods such as the R-tree [Gut84] suffer from the so-called *Curse of Dimensionality*: The number of time series that need to be examined grows exponentially with the number of dimensions, i.e., the length of the time series [Sch15; Spr91]. Usually, spatial index structures already degenerate with 10–20 dimensions [WSB98; AFS93]. Therefore, it is common practice to reduce the number of dimensions by approximating the time series in the first place. The reduced data can then be stored in a multidimensional index structure.

An established framework for this procedure is the *Generic Multimedia Indexing Method* (GEMINI) [FRM94; Fal96]. It requires a distance function and a feature-extraction or dimensionality reduction method, together with a distance function in the feature space that lower-bounds the actual distance. Most of the techniques introduced in the following implicitly follow the GEMINI concept.

### 2.2.1 Transformation-Based Approximation

[AFS93] and [FRM94] propose an indexing method that uses *Discrete Fourier Transform* (DFT) to map time series to the frequency domain. The $n$-point DFT [OS75] of a signal or time series is an equally long sequence of complex numbers, called the Fourier coefficients. Each coefficient describes the share of a frequency averaged over the entire duration of the signal, where the first coefficients quantify low frequencies. The real part of a coefficient gives the amplitude of a cosine wave, while the imaginary part is the amplitude of a sine wave. An essential property of the DFT is that the Euclidean distance between two signals in the frequency domain is equivalent to their distance in the time domain. Since the values of most real-world signals depend on their neighbors, they are not white noise. Therefore, these signals have only a few strong frequencies and Fourier coefficients. Thus, keeping only the first $c$ coefficients turns each time series into a point (with a real and an imaginary part) in a $2c$-dimensional space without great information loss. Experiments show that less than six coefficients are already adequate to calculate a lower bound for the true distance in the time domain. Having reduced the dimensionality, the time series can be indexed well. A symbolic representation for time series based on DFT together with a modified prefix tree for indexing is proposed in [SH12].

While DFT gives frequencies describing the general shape of a time series, it does not preserve any temporal information due to its periodic and infinite basis functions. In contrast, the *Discrete Wavelet Transform* (DWT) [Gra95] represents a time series as a linear combination of so-called wavelets, which are localized in frequency *and* time domain. Localized in time means that the function has a non-zero amplitude in only a small time interval. For time series, the most common wavelet basis function is the Haar wavelet, which is a combination of two rectangular functions [Haa10]. The Haar transform can also be considered as iteratively averaging adjacent values of the original time series. Like for

DFT, it is sufficient to index only the first coefficients. [KA99] first applies DWT with Haar wavelets to time series and states that it outperforms indexing with Fourier transform concerning complexity and the performance of similarity search.

DFT and DWT both transform single data objects independently from the rest of the dataset. *Single Value Decomposition* (SVD) [KJF97], in contrast, is a global transformation examining the entire dataset. For dimensionality reduction and indexing, only the overall most important axis or principal components are kept. While the reconstruction error is smaller than for linear transformations like DFT and DWT, the computational effort is comparatively high. Furthermore, since the index usually needs to be rebuilt whenever a new time series is inserted, SVD is unfeasible for massive datasets.

The transformation methods introduced above have in common that they are not directly applicable for indexing time series for an interval-based $k$-NN search. As they approximate time series, fine structures of high frequency get lost. This is no problem for similarity search on entire time series, as the distance measures on the approximations underestimate the true distance. However, two time series might have a far distance as a whole and therefore a comparatively large approximated distance—but they might be very close in some arbitrary small time intervals. Moreover, for raw subsequences, the frequency spectrum might strongly differ from the one of the entire sequence. This is especially not covered anymore after discarding most coefficients for dimensionality reduction. Furthermore, there is no known way to obtain the frequencies of a subsequence from the frequency spectrum of the entire sequence without performing an inverse DFT in the first place. Similar issues argue against the use of SVD in this work. Also with DWT, it is difficult to deal with peaks in subsequences due to the averaging of raw values. Moreover, DWT requires the length of the time series to be a power of two. While it is possible to use padding to fill sequences that are too short, this is not optimal.

### 2.2.2 Piecewise Approximation

Piecewise approximation methods divide a time series into segments and apply a function on each segment to define its value.

[YF00] and [Keo+01] suggest using segments of equal length and representing each one by its mean value. While former authors call this approach *Segmented Means*, the term *Piecewise Aggregate Approximation* (PAA) by the latter is prevailing in literature. PAA can alternatively be interpreted as a linear combination of differently shifted box functions.

[Lin+07] introduces the symbolic representation *Symbolic Aggregate Approximation* (SAX) on top of PAA. SAX first standardizes the time series to a mean value of zero and a standard deviation of one, before it translates them to the PAA representation. The coefficients of PAA are then mapped to a small alphabet of symbols such as *a, b, c*, whereby the discretization breakpoints are chosen so that the symbols are evenly distributed. The concatenation of symbols representing a sequence is called word. An *indexable SAX* (iSAX) is presented in [SK08; Cam+10; Cam+14]. Instead of alphanumeric characters, iSAX uses binary numbers as symbols, which allows deriving a word of reduced cardinality by simply

removing trailing bits. Raw time series that share the same iSAX representation are stored in a text file that has the particular iSAX word encoded as the file name. If a file (e.g., representing $\langle \mathtt{10}, \mathtt{11}, \mathtt{10} \rangle$) exceeds the maximum size allowed, it is split into two files each having an additional bit in its iSAX word (e.g., $\langle \mathtt{100}, \mathtt{11}, \mathtt{10} \rangle$ and $\langle \mathtt{101}, \mathtt{11}, \mathtt{10} \rangle$). *Adaptive Data Series* (ADS, ADS+) indexing [ZIP14; ZIP16] suggests a lazy index creation on top of the iSAX representation. To avoid a time-consuming initial index creation, the index is built and refined iteratively on query time.

*Adaptive Piecewise Constant Approximation* (APCA) [Cha+02] extends PAA by using varying segment lengths. Areas of high activity are resolved finer to reduce the reconstruction error. The *extended APCA* (EAPCA) representation [Wan+13] additionally includes the standard deviation besides the mean value per segment. Thus, it is possible to define not only a lower bound but also an upper bound for the true distance. An index structure based on the EAPCA representation is the *Dynamic Splitting Tree* (DSTree) [Wan+13]. In a DSTree each node holds the minimum and maximum mean and standard deviation of the subtree below. Whenever a leaf node holding raw time series exceeds the maximum capacity, it is split in either horizontal or vertical direction. A horizontal split partitions a set of time series by either their mean value or their standard deviation. Vertical splitting, in contrast, divides a segment into two and assigns the time series according to their mean value in the left subsegment to a new leaf.

The PAA-based representations and indexing techniques introduced above have some drawbacks regarding similarity search within intervals. As PAA represents segments by their mean value, it is prone to false dismissals, when peaks or outliers lie within the query time interval. Two time series might be wide apart in a segment regarding their mean values, but be close if the query interval within that segment contains a peak in one time series. This might result in false dismissals if the candidate time series is discarded only based on the mean values. Furthermore, the z-standardization performed by SAX manipulates the overall value range so that it is not possible to differentiate time series with the same trend but different vertical intercepts. For example, two horizontal lines at $-10$ and 42 are both mapped to a line at zero. As long as only the course of time series is of interest, this is not a disadvantage. However, in this work, the $k$-NN time series are supposed to be obtained in the original value range.

Instead of representing segments by their mean value, they can be defined by straight lines [SZ96; Keo97]. However, it seems like such approaches have not been pursued in research. They further suffer from the same problems regarding peaks as PAA-based techniques.

[KS10] proposes the concept of envelopes using the minimum and the maximum value of each segment as its representative. Thanks to this, peaks in time series get not lost and can be taken into account for querying. Likewise, the index structure *Time Warping in Indexed Sequential Structure* (TWIST) [NRR10] introduced in the same year is based on envelopes. TWIST calculates envelopes for groups of time series and stores them in a simple index structure for a $k$-NN search under DTW. To fasten query processing, the envelopes are segmented like in [KS10]. As this work adopts several concepts introduced by TWIST, including a lower-bound function for the distance to the time series inside an envelope, Section 3.3 on page 9 introduces it more detailed.

# 3 Fundamentals

This chapter first introduces the $k$-nearest neighbor search and the distance measure dynamic time warping. It motivates the need for an efficient $k$-NN algorithm under DTW using lower-bound techniques and presents the index structures TWIST and R-tree.

## 3.1 $k$-Nearest Neighbors Algorithm

The $k$-nearest neighbors ($k$-NN) algorithm is a fundamental and simple similarity search method that is usually used for classification or regression [FH51; CH67]. Given an input object, its $k$ closest neighbors in regard to a distance function are determined. For classification, the object is assigned the class of the majority of its neighbors; for regression, their average value. The Euclidean distance is commonly used as distance measure; however, more sophisticated and domain-specific ones are applicable, too.

## 3.2 Dynamic Time Warping

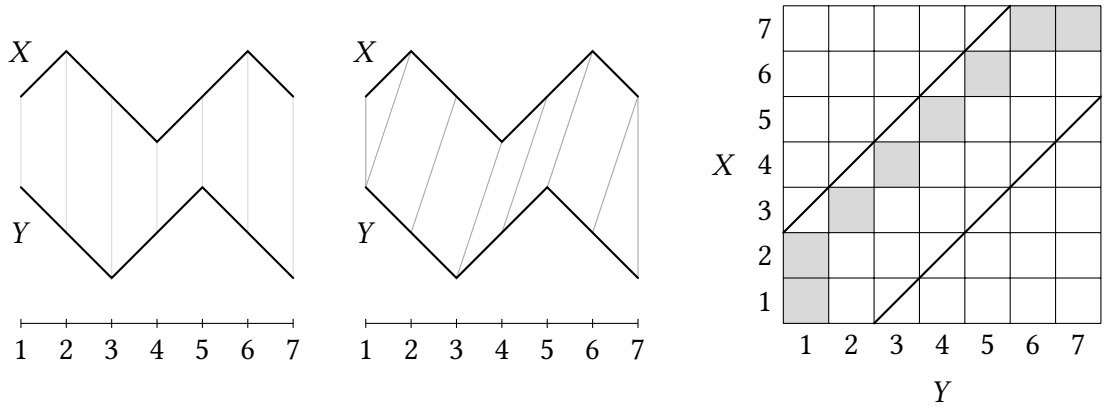
Dynamic time warping (DTW) is one of the best similarity measures for sequences, as it is less sensitive to typical transformations like shifting and scaling than other distance measures [YC15; SA14]. Initially developed for speech recognition [SC78], it was soon applied to time series [BC94]. In contrast to the Euclidean distance, it does not perform a one-to-one point comparison but a many-to-many comparison [Cas+12], as visualized in Figure 3.1a and 3.1b.

DTW aligns two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ in the temporal domain so that the total accumulated cost or distance is minimal. For this purpose, a distance matrix $D$ is calculated, where each cell $D_{i,j}$ corresponds to the minimum cumulative distance between the sequence elements $x_i$ and $y_j$.

The distance matrix $D$ of size $(m+1) \times (n+1)$ can be computed by dynamic programming recursively applying

$$D_{i,j} = d(x_i, y_j) + \min(D_{i,j-1}, D_{i-1,j}, D_{i-1,j-1}) \tag{3.1}$$

for $i = 1 \ldots m$ and $j = 1 \ldots n$, with the lengths $m$ and $n$ of the two sequences. $D_{0,0}$ is initialized with 0, whereas the other cells are initially set to $\infty$. $d(x_i, y_j)$ is the cost of matching two sequence elements $x_i$ and $y_j$ and is usually calculated with the squared




**(a)** Euclidean distance **(b)** DTW distance **(c)** DTW distance matrix

**Figure 3.1** Given two sequences $X$ and $Y$, (a) shows the warping path of the Euclidean distance, (b) shows a more suitable warping path by DTW, (c) shows the warping path in the DTW distance matrix constrained by a Sokoe-Chiba band (graphics based on [LY14]).

Euclidean distance $(x_i - y_j)^2$ [NRR10]. The final DTW distance corresponds to the total accumulated cost, i.e., $DTW(X, Y) = D_{m,n}$. Some publications finally compute the $p$-th root; hence, in this case, $DTW(X, Y) = \sqrt[p]{D_{m,n}}$ (with $p = 2$, if $d$ is the Euclidean distance) [NRR10]. Optionally, the optimal warping path $\langle (i_1, j_1), \ldots, (i_w, j_w) \rangle$ (with $\max(m, n) \leq w \leq n+m-2$ [LY14]) can be obtained by tracing back in the matrix choosing the cell with the lowest accumulated cost in each step, as visualized in Figure 3.1c .

DTW is reflexive and symmetric, but not transitive, as it does not satisfy the triangular inequality $DTW(X, Y) + DTW(Y, Z) \geq DTW(X, Z)$. Latter property can cause false dismissals for many index structures including most spatial access methods and metric-based trees. Only a costly sequential scan through all sequences can avoid this [YJF98].

### 3.2.1 Constraints

The canonical DTW has a high, quadratic computational complexity of $O(m \times n)$ for matching two sequences of length $m$ and $n$. Furthermore, the alignment might match sequence elements that are wide apart in time, which usually degrades the results of a similarity search. To overcome these issues, it is common to constrain the computation of the matrix. The so-called Sakoe-Chiba band [SC78] defines a warping window $0 \leq |i_k - j_k| \leq \omega \leq \min(m, n)$ so that the warping path is at each position $k$ allowed to be at most $\omega$ beyond from the distance matrix's main diagonal (in both horizontal and vertical direction). This is illustrated in Figure 3.1c. As a result, fewer matrix cells have to be computed and the time difference between two aligned sequence elements does not exceed the limit. [RK04] presents a general constraint model that can represent several distance measures such as the Euclidean distance or DTW, without or together with various bands such as the Sakoe-Chiba band or the Itakura parallelogram [Ita75].

### 3.2.2 Lower Bounds

To speed up the iterative $k$-NN search of time series under DTW, one can make use of one or more lower-bound functions for pruning off time series that cannot be the best match. A lower-bound function $LB(X, Y)$ returns an estimated value less than or equal to the exact value $DTW(X, Y)$. Besides, it satisfies—in contrast to DTW—the triangular inequality. If the lower-bound distance of a time series is already larger than the best-so-far distance in the nearest neighbor search, the time series can be omitted since it is guaranteed that its exact distance is not smaller. Only if the lower bound is smaller, the exact DTW has to be calculated.

A lower-bound distance should be much faster to compute than the exact DTW. To ensure pruning of many time series, it is desirable that the lower bound is tight, in other words, that it is large and close to the exact value. The tightness can be computed by $LB(X, Y)/DTW(X, Y)$ which gives a value in $[0, 1]$ where 1 is best [LY14].

[YJF98] proposes the first lower-bound technique for DTW, usually referred to as LB_Yi. This lower bound is the sum of the distances between the maximum of one sequence and those elements of the other sequence that are greater than this maximum, plus vice versa the summed distances for elements smaller than the minimum.

A fast to compute but less effective lower bound regarding tightness and pruning power is LB_Kim [KPC01]. For two sequences $X$ and $Y$ of length $n$, it is defined as the maximum of the distances between the sequences' first, last, maximum, and minimum value:

$$LB\_Kim(X, Y) = \max \begin{cases} |x_1 - y_1| \\ |x_n - y_n| \\ |\max(x_1, \ldots, x_n) - \max(y_1, \ldots, y_n)| \\ |\min(x_1, \ldots, x_n) - \min(y_1, \ldots, y_n)| \end{cases} \tag{3.2}$$

Further lower-bound functions, as well as a comparison, can be found in [NB14; LY14].

## 3.3 TWIST

*Time Warping in Indexed Sequential Structure* (TWIST) is an index structure for time series that allows a $k$-NN search under DTW with no false dismissals guaranteed [NRR10]. It stores groups of similar time series and defines a lower-bound function for the distance between a query sequence and all time series of a group. For a $k$-NN search, this allows not only the pruning of single time series but of entire groups reducing runtime and computational costs.

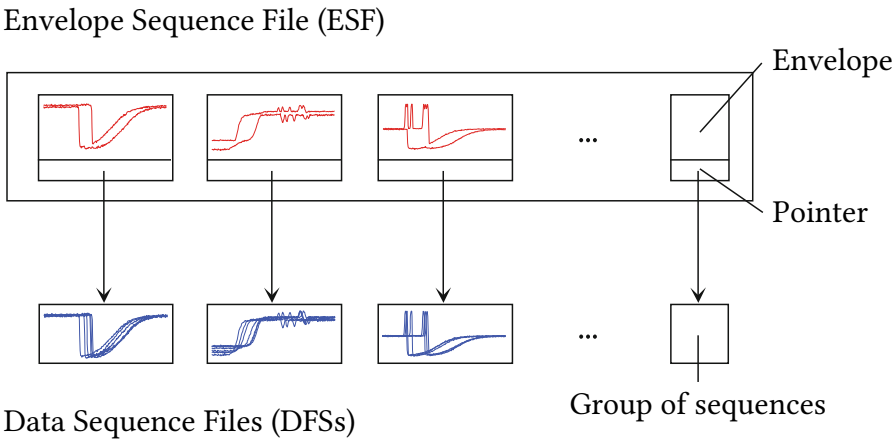



**Figure 3.2** A sample data structure of TWIST (graphic by [NRR10])

### 3.3.1 Data Structure

A group of similar time series is wrapped by a so-called envelope that is defined by the minimum and maximum value of the group's time series at each time step. Hence, an envelope consists of a lower and an upper sequence. Conceptually, an envelope is comparable to a minimum bounding rectangle as known from R-trees (cf. Section 3.4 on page 14).

An envelope $E$ with a lower sequence $E_l$ and an upper sequence $E_u$, wrapping $\mathbb{X}$, a set of time series of length $n$, is defined as follows:

$$E = \{E_l, E_u\} \tag{3.3}$$

where

$$E_l = \left\langle \ldots, \min_{X \in \mathbb{X}}(x_i), \ldots \right\rangle = \langle \ldots, e_{l,i}, \ldots \rangle \qquad \text{with } 1 \leq i \leq n$$

$$E_u = \left\langle \ldots, \max_{X \in \mathbb{X}}(x_i), \ldots \right\rangle = \langle \ldots, e_{u,i}, \ldots \rangle$$

As shown in Figure 3.2, each envelope is stored in a flat index structure called *envelope sequence file* (ESF) together with a pointer to a so-called *data sequence file* (DSF) holding the grouped raw time series.

### 3.3.2 Index Construction

The following sections describe the insertion of time series into the index structure TWIST, as well as their deletion from the index.

#### 3.3.2.1 Insertion

TWIST inserts a time series into the envelope for which the insertion cost is minimal and updates the envelope's lower and upper bound accordingly. Once an envelope or rather its

DSF contains more time series than allowed by the maximum page size, it is split into two DSFs, which are stored in the ESF again. The maximum page size is the only user-defined parameter.

The cost of inserting a time series $X = \langle x_1, \ldots, x_n \rangle$ into an envelope $E$ with a lower sequence $E_l = \langle e_{l,1}, \ldots, e_{l,n} \rangle$ and an upper sequence $E_u = \langle e_{u,1}, \ldots, e_{u,n} \rangle$ is calculated from the envelope area after the insertion and the associated area enlargement:

$$InsertionCost(X, E) = \sqrt[p]{\sum_{i=1}^{n} \begin{cases} |x_i - e_{l,i}|^p & \text{if } x_i > e_{u,i} \\ |e_{u,i} - x_i|^p & \text{if } x_i < e_{l,i} \\ 0 & \text{otherwise} \end{cases}} \tag{3.4}$$

$p$ is the dimension of $L_p$ norms and usually $p = 2$.

For splitting a DSF if it exceeds the maximum page size, TWIST adopts $k$-means clustering [Mac67] with $k = 2$ and the Euclidean distance. It aims to create two groups of time series so that both new envelopes are tight and overlap only slightly. Instead of $k$-means, other algorithms such as in R-tree [Gut84] or R*-tree [Bec+90] can be used; however, they are computational more complex.

#### 3.3.2.2 Deletion

To delete a time series from the index, it is deleted from the containing DSF. Moreover, the ESF needs to be adapted, too, so that the envelope remains as tight as possible. However, as the minimum and maximum functions are not self-maintainable concerning deletions [XE00], all remaining time series of the DSF need to be accessed to recalculate the envelope. Besides an eager deletion algorithm that recalculates the envelope immediately, TWIST alternatively provides a lazy algorithm that does not update the envelope at all. The lazy algorithm is much faster and, nevertheless, does not lead to false dismissals during the $k$-NN search. Hence, the trade-off between both deletion algorithms is runtime versus tightness of the envelope.

### 3.3.3 $k$-NN Querying

When a $k$-NN query is issued, TWIST calculates lower-bound distances to each envelope. If the lower-bound distance to an envelope is larger than the current best-so-far distance, it is pruned, as it cannot contain any target time series. The best-so-far distance is the exact or a lower-bound distance to the (candidate) $k$-nearest neighbor time series. By pruning one single envelope, a large number of candidate time series is omitted while still guaranteeing no false dismissals. To further improve the runtime, the lower bound is calculated with a segmented representation of both the query time series and the enclosing envelope, starting with a coarse segmentation on the time axis. As long as the lower-bound distance is smaller or equal to the best-so-far distance, the segmentation is refined iteratively to increase the accuracy of the estimate. Once the segmentation has reached the finest resolution, all time series of the current envelope are accessed.

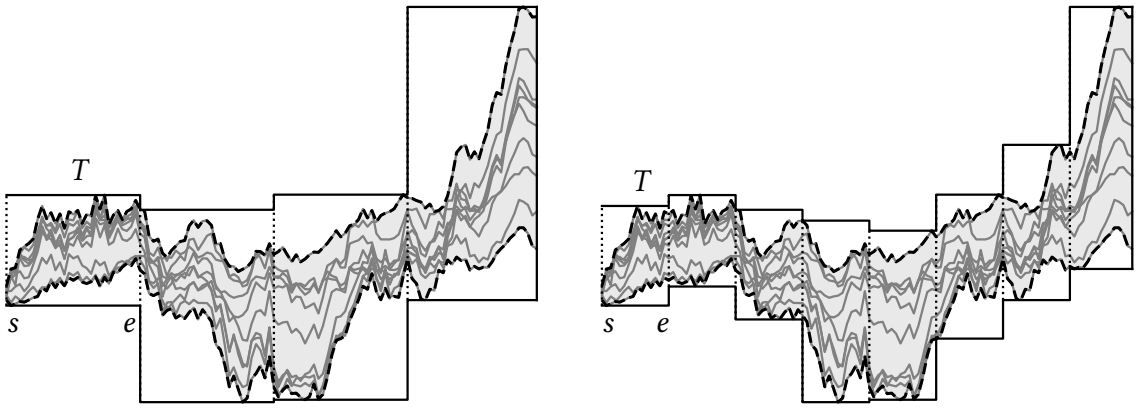



**Figure 3.3** Segmented envelope $E^{\mathcal{T}}$ with different segment lengths $T$. The dashed lines illustrate the lower ($E_l$) and upper sequence ($E_u$) of the envelope $E$ wrapping a group of time series (gray lines). The solid horizontal lines illustrate the lower ($E_l^{\mathcal{T}}$) and upper sequence ($E_u^{\mathcal{T}}$) of the segmented envelope.

#### 3.3.3.1 Segmentation

Segmentation of a time series or envelope is a simple dimensionality reduction technique and illustrated in Figure 3.3. The given time series $X = \langle x_1, \ldots, x_n \rangle$ is first split into adjacent time intervals—called segments—of length $T$. Solely the last interval might be of smaller length. The segmented time series $X^{\mathcal{T}}$ is finally composed of two sequences, where the lower sequence $X_l^{\mathcal{T}}$ contains the minimum value within each interval and the upper sequence $X_u^{\mathcal{T}}$ holds the maximum values:

$$X^{\mathcal{T}} = \left\{ X_l^{\mathcal{T}}, X_u^{\mathcal{T}} \right\} \tag{3.5}$$

where

$$X_l^{\mathcal{T}} = \langle \ldots, \min(x_s, \ldots, x_e), \ldots \rangle = \left\langle \ldots, x_{l,i}^{\mathcal{T}}, \ldots \right\rangle$$

$$X_u^{\mathcal{T}} = \langle \ldots, \max(x_s, \ldots, x_e), \ldots \rangle = \left\langle \ldots, x_{u,i}^{\mathcal{T}}, \ldots \right\rangle$$

$$s = (i-1) \cdot T + 1 \qquad \text{with } 1 \leq i \leq \lceil n/T \rceil$$

$$e = \min(i \cdot T, n)$$

For segmenting the envelope $E = \{E_l, E_u\} = \{\langle e_{l,1}, \ldots, e_{l,n} \rangle, \langle e_{u,1}, \ldots, e_{u,n} \rangle\}$, the minimum values of its lower sequence intervals are used, together with the maximum values of its upper sequence intervals. Hence, the segmented representation $E^{\mathcal{T}}$ is defined as follows:

$$E^{\mathcal{T}} = \left\{ E_l^{\mathcal{T}}, E_u^{\mathcal{T}} \right\} \tag{3.6}$$

where

$$E_l^{\mathcal{T}} = \langle \ldots, \min(e_{l,s}, \ldots, e_{l,e}), \ldots \rangle = \left\langle \ldots, e_{l,i}^{\mathcal{T}}, \ldots \right\rangle$$

$$E_u^{\mathcal{T}} = \langle \ldots, \max(e_{u,s}, \ldots, e_{u,e}), \ldots \rangle = \left\langle \ldots, e_{u,i}^{\mathcal{T}}, \ldots \right\rangle$$

#### 3.3.3.2 Lower-Bound Distance for a Group of Sequences

TWIST introduces a lower-bound function for the true DTW distance between a query sequence and a group of time series enclosed by an envelope. It is a fundamental concept

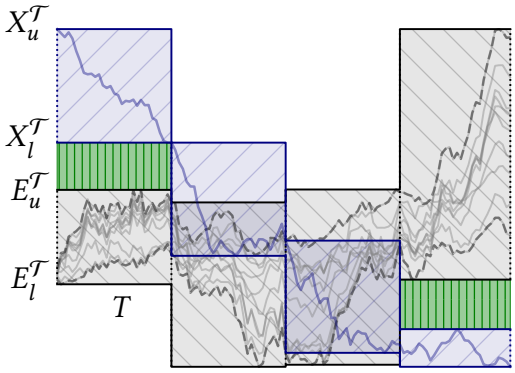



**Figure 3.4** Visualization of the lower-bound distance for a group of sequences and a query sequence. The black boxes (\\) represent a segmented envelope $E^{\mathcal{T}}$, while the blue boxes (//) represent a segmented query sequence $X^{\mathcal{T}}$. The green areas (||) represent the distance $d$ between a query segment and an envelope segment.

and crucial for $k$-NN querying. The lower bound works on segmented representations and is based on a lower-bound function for two segmented sequences [SYF05].

The equation for the lower-bound distance $LBG(X^{\mathcal{T}}, E^{\mathcal{T}})$ between a segmented query sequence $X^{\mathcal{T}}$ and a segmented envelope $E^{\mathcal{T}}$ is similar to the original DTW function introduced in Section 3.2 on page 7. Again, a distance matrix $D$ is calculated as in Equation 3.1. However, instead of using the Euclidean distance between two sequence elements, another local cost function $d$ is applied. Now, $d$ gives the distance between a query segment and an envelope segment. Effectively, $d$ is a lower bound for the distance between the query and all time series in the envelope within the interval covered by the segment.

$$\begin{aligned} d\left(X_i^{\mathcal{T}}, E_j^{\mathcal{T}}\right) &= d\left(\left\{x_{l,i}^{\mathcal{T}}, x_{u,i}^{\mathcal{T}}\right\}, \left\{e_{l,i}^{\mathcal{T}}, e_{u,i}^{\mathcal{T}}\right\}\right) \\ &= T \cdot \begin{cases} \left|x_{l,i}^{\mathcal{T}} - e_{u,j}^{\mathcal{T}}\right|^p & \text{if } x_{l,i}^{\mathcal{T}} > e_{u,j}^{\mathcal{T}} \\ \left|e_{l,j}^{\mathcal{T}} - x_{u,i}^{\mathcal{T}}\right|^p & \text{if } e_{l,j}^{\mathcal{T}} > x_{u,i}^{\mathcal{T}} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.7}$$

The lower bound LBG is illustrated in Figure 3.4. Besides the just introduced LBG, TWIST additionally proposes an alternative lower-bound function $LBG_K$ that makes use of a global constraint. As $LBG_K$ is not used for this work, it is omitted in this chapter.

#### 3.3.3.3 *k*-NN Query

The $k$-NN search starts with an initial estimated best-so-far distance. The best-so-far distance is a lower-bound distance to the $k$-nearest neighbor; hence, not necessarily to the closest neighbor. For this purpose, TWIST calculates the LBG lower-bound distance to each envelope using a maximal segment length $T$. Although not mentioned in [NRR10], if the query sequence was indexed, the LBG distance equals to zero for the envelope that contains the sequence. Since a best-so-far distance of zero is no advantage, the initialization can be omitted in this case.

Afterward, the LBG lower bound is repeatedly calculated for each envelope, as long as it does not exceed the current best-so-far distance. After each iteration, the segment length is reduced, and the LBG distance is recalculated with a finer representation of the query sequence and the current envelope. Since [NRR10] does not explicitly define how to reduce the segment length, it is assumed that it is halved. Reducing the segment length makes the lower bound larger and thus tighter. Once the lower-bound distance is larger than the best-so-far distance, the segment length is reset to the initial $T$, and the current envelope is omitted—together with all time series it holds since they cannot be a $k$-NN.

If the segments have reached their minimum length, and the LBG is still smaller than the best-so-far distance, all time series of the current envelope are accessed. For each raw time series, a lower-bound distance is calculated, e.g., using LB_Yi or LB_Kim as described in Section 3.2.2 on page 9. If also this lower bound is smaller than the current best-so-far distance, the true DTW distance is calculated. Otherwise, the next time series is examined.

When also the true DTW distance is smaller than the best-so-far distance, the time series is pushed onto a max-heap. The max-heap holds the $k$ current result sequences, where the first one always has the largest distance, and thus is the $k$-nearest neighbor. Subsequently, the first element is removed from the max-heap to make sure that it does not contain more than $k$ time series. Finally, the best-so-far distance is set to the exact distance of the new first time series of the max-heap.

A pseudo-code describing this algorithm is listed in the original publication [NRR10].

### 3.3.4 Evaluation

Unlike most multidimensional index structures, the index structure TWIST can be seen as a tree with only one level. Therefore, all nodes need to be examined for choosing the best envelope for inserting a time series. This gives a runtime complexity of $O(N)$ for a dataset size of $N$. In contrast, most spatial index structures have an average complexity in $O(\log N)$ for updates. Experiments confirm that the performance of TWIST already drops drastically after inserting hundred-thousands of time series. Not only for inserts but also for $k$-NN queries it is necessary to access all envelopes, even though distances are calculated on segmented representations. However, as the envelope's segments are not materialized, further computations are required for segmentation—at least one per envelope and often even more frequently for smaller segment lengths. The novel index structure in this work, in contrast, does not suffer from these weaknesses that make TWIST unable to handle vast amounts of data.

## 3.4 R-Tree

The R-tree [Gut84] is together with its variants the probably most popular index structure for multidimensional data, such as spatial objects like points or polygons [Man+06]. The

height-balanced tree is based on the $B^+$-tree [BM70; Com79] and corresponds to a hierarchy of $d$-dimensional minimum bounding rectangles (MBR). An MBR is the smallest rectangle that tightly bounds its content. In contrast to structures like the $k$-d-tree [Ben75], the MBRs might overlap. An R-tree enables a straightforward search for all data objects that are intersected or overlapped by a given query rectangle. Besides these so-called range queries, $k$-NN searches are applicable as well [RKV95; HS99].

An R-tree of order $(m, M)$ is characterized as follows. Each node, except the root, contains between $m$ and $M/2$ entries, where an entry is composed of an MBR and a child pointer. For leave nodes, the pointer refers to a single data object. The root node needs to contain at least two entries unless it is a leaf, in which case it may hold nothing or a single entry.

To insert a data object, the R-tree is traversed starting with the root node to find an appropriate leaf node. At each level, the node, whose MBR requires the minimum area enlargement to cover the new object, is chosen. If the leaf node found does not already contain $M/2$ entries, the data object is inserted, and the MBRs of all parental nodes up to the root are updated accordingly. Otherwise, the leaf node is split into two new leaf nodes, whereupon the object is inserted, and the MBRs are updated likewise. Whenever a leaf node was split, its parent might overflow and be split as well. Consequently, the splits might propagate upwards until a new root node is created.

While the R-tree especially tries to minimize the area of each MBR, the R*-tree [Bec+90] introduces further optimization criteria including the minimization of the overlap between the MBRs. On insertion, the R*-tree chooses the leaf node whose MBR enlargement leads to the minimum overlap among the sibling nodes. If a leaf node already contains the maximum number of objects, it is not split in the first place. Instead, the leave node's outer 30% of data objects are deleted and reinserted into the tree. While reinsertion is a kind of rebalancing and restructuring, it is a costly operation and therefore applied only once per tree level.

# 4 Time Series Envelopes Index Tree

This chapter unveils the novel index structure *Time Series Envelopes Index Tree* (TSEIT[1]) that allows an efficient interval-based $k$-NN search of time series.

After an overview of the fundamental concepts of TSEIT, the data structure is introduced in detail. The index construction, including traversal policies and splitting algorithms, is explained subsequently, before the index-based $k$-NN query algorithm is presented.

## 4.1 Overview

The basic idea of TSEIT is to store groups of time series in the leaf nodes of a height-balanced tree. An exemplary TSEIT tree is illustrated in Figure 4.1. Each group is represented by an envelope that tightly wraps the time series. Hence, an envelope consists of an upper sequence and a lower sequence. The upper one is defined by the maximum values of the enclosed time series, while the lower sequence is defined by the minimum values. Each inner node holds an envelope that tightly wraps the envelopes of its child nodes. Whenever a leaf node contains more time series than it is allowed to, it is either split into two nodes, or some of its time series are reinserted into the tree.

To save runtime and storage space, the envelopes are by default stored with reduced dimensionality by segmenting them as introduced in Section 3.3.1 on page 10. In order to improve the accuracy of distance calculations, the number of segments per envelope increases in deeper tree levels

For $k$-NN querying, the distance between a query time series and an envelope is a lower bound of the true distance to any of the enclosed time series. Hence, if the distance to an envelope is larger than the distance to the current $k$-NN, the enclosed time series have the same or an even larger distance to the query time series. In this case, the envelope cannot contain any $k$-NN time series. This allows omitting the entire subtree below the envelope without false dismissals. Thanks to this, TSEIT needs to calculate the costly DTW distance to only a fraction of all time series. As an envelope preserves the extrema of its time series, the distance calculation and thus the $k$-NN search works on arbitrary time intervals.

The optimal TSEIT tree is compact, holding well-filled leaf envelopes of a small area that overlap only slightly. This ensures low traversal cost and reduces the chance that a query time series hits many envelopes. However, some conditions contradict each other: storing only a few time series in each envelope minimizes the average envelope area, but degrades

[1] *TSEIT* is pronounced *[tsait]*, like the German word *Zeit* for *time*

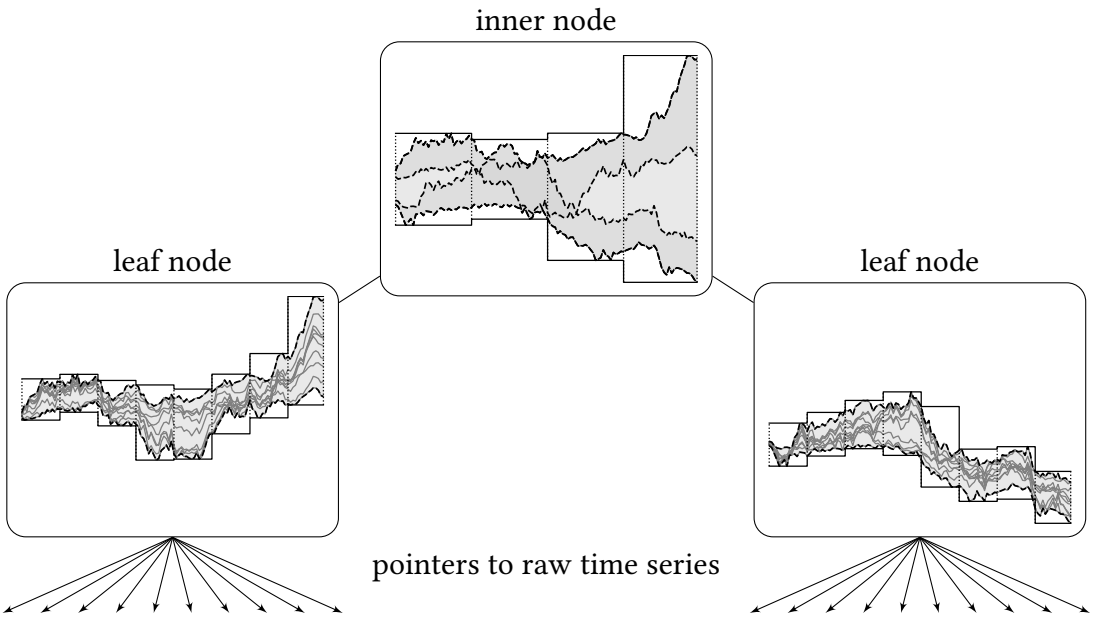



**Figure 4.1** Schematic illustration of a TSEIT tree with three nodes. The envelope of the inner node is formed by the envelopes of the two leaf nodes, each referencing a set of raw time series. The nodes hold segmented envelopes where the segment length of a parent node is twice the segment length of its child nodes. In this illustration, the raw envelopes and time series are depicted for a better understanding only.

the query runtime due to an increased tree size, on the other hand. TSEIT balances this by a sophisticated and tailored index creation algorithm.

## 4.2 Data Structure

The TSEIT tree is always height-balanced and thus its leaf nodes have the same depth. Its topology and the way it is balanced is based on concepts of the R-tree. TSEIT consists of inner nodes and leaf nodes each holding an envelope. Leaf nodes additionally store a set of pointers to raw time series. The envelopes are usually segmented and wrap either the envelopes of the child nodes or, in case of leaf nodes, a set of time series. Figure 4.2 compares the different data representations.

Each leaf node holds pointers to at least $l_{\min} \geq 1$ and at most $l_{\max} \geq 2l_{\min}$ raw time series, and each inner node has between $c_{\min} \geq 2$ and $c_{\max} \geq 2c_{\min} - 1$ child nodes. The indexed time series have the same length.

In the default configuration, the envelopes are stored with reduced dimensionality by segmenting them on the time axis (cf. Section 3.3.3.1 on page 12). The segments have the length $T_{\min} \geq 1$ for leaf-level envelopes, while the length doubles with each level up to the root node. Doubling the segment length halves the length of the resulting sequences, which reduces the accuracy of lower-bound distance calculations, but saves storage and runtime for traversal on the other hand.

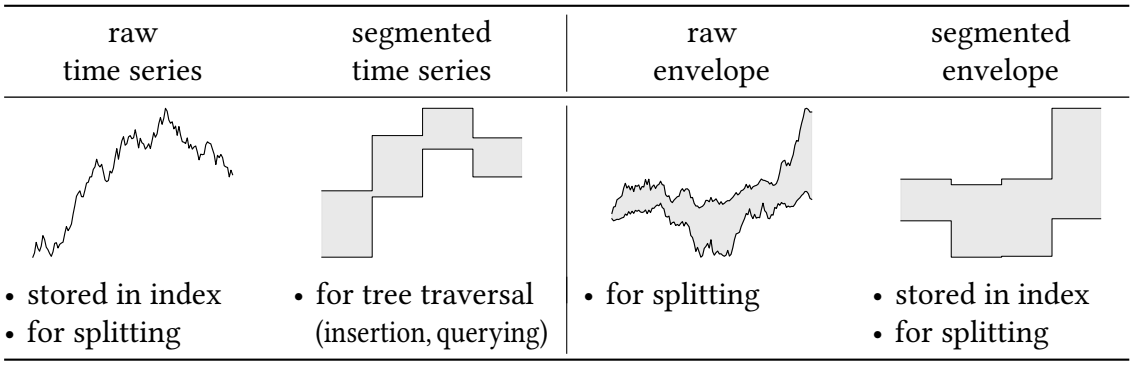



**Figure 4.2** Comparison of raw and segmented time series and envelopes.

In general, it is differentiated between balanced and unbalanced trees. An unbalanced tree does not impose any constraints on the height, which thus might be $O(N)$ at worst. For unevenly distributed data together with an unfavorable insertion order (e.g., sorted), subtrees can become very deep, which leads to high traversal cost. Moreover, the runtime for insertions and searches might strongly differ between successive executions depending on the length of the traversal path. Since most real-world datasets do not follow a normal distribution, an unconstrained structure often degenerates. In contrast, a height-balanced tree, like the R-tree or TSEIT, ensures that all leaf nodes have the same depth. The tree height and thus the length of the paths to the leaf level is limited by $O(\log N)$. Balancing ensures scalability, as operations like insertion and search stay efficient with increasing amount of inserted data. This is also the reason why the most common database systems[2] such as Oracle Database, Microsoft SQL Server, MySQL, PostgreSQL or MongoDB use balanced index structures (usually variants of the B-tree [BM70; Com79]).

There are different kinds of balanced trees. Many balanced trees such as the AVL-tree or the Red-Black-tree [Bay72] are binary trees. Always having two child nodes, the height of the tree can only be controlled by the capacity of the leaf nodes. This, however, has a significant impact on the query performance and does not influence the general topology. In contrast, a B-tree or the derived R-tree support—like TSEIT—more than two child nodes and thus allow regulating whether a tree becomes high (with few child nodes) or broad (with many child nodes). For TSEIT, this is important, as the height of a tree influences the segment length in the upper tree levels. A broad TSEIT tree with few levels and many child nodes has finer resolved envelopes in the upper levels than a high tree with few child nodes.

## 4.3 Index Construction

Inserting a time series into the index is all about finding an appropriate leaf node that can hold the time series so that the overall tree structure is optimal. Already while descending the tree, each segmented envelope $E^{\mathcal{T}}$ on the insertion path (including the target leaf

[2] Database ranking: `https://db-engines.com/en/ranking` (archived in January 2018: `https://web.archive.org/web/20180103030915/https://db-engines.com/en/ranking`)

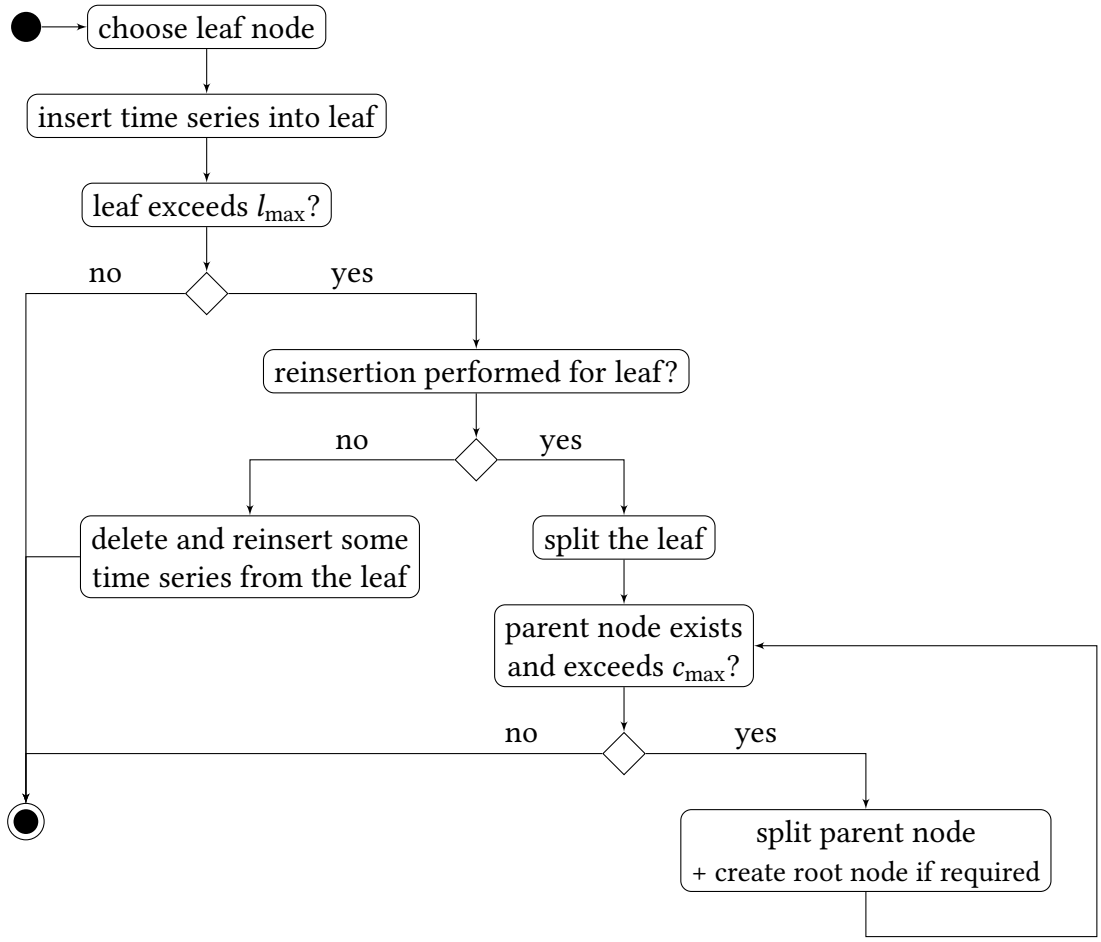



**Figure 4.3** Activity diagram of the insertion algorithm for index construction.

envelope) is updated according to the segmented representation $X^{\mathcal{T}}$ of the time series $X$ with the segment length $T$ of the current tree level:

$$UpdateEnvelope\left(E^{\mathcal{T}}, X^{\mathcal{T}}\right) = \begin{cases} E_u^{\mathcal{T}} \leftarrow \left\langle \max\left(e_{u,1}^{\mathcal{T}}, x_{u,1}^{\mathcal{T}}\right), \ldots, \max\left(e_{u,|E^{\mathcal{T}}|}^{\mathcal{T}}, x_{u,|X^{\mathcal{T}}|}^{\mathcal{T}}\right) \right\rangle \\ E_l^{\mathcal{T}} \leftarrow \left\langle \min\left(e_{l,1}^{\mathcal{T}}, x_{l,1}^{\mathcal{T}}\right), \ldots, \min\left(e_{l,|E^{\mathcal{T}}|}^{\mathcal{T}}, x_{l,|X^{\mathcal{T}}|}^{\mathcal{T}}\right) \right\rangle \end{cases} \tag{4.1}$$

Finally, a pointer to the raw time series is added to the leaf node found. If the leaf node exceeds $l_{\max}$—the maximum number of time series it is allowed to hold—after the insertion, either some time series are deleted from the node and reinserted, or the leaf is split into two leaf nodes. This might result in an overflow of the parent node, which is in consequence split, too. If splitting propagates up to the root node, a new root is created. While splitting a leaf node reorganizes the time series locally, reinsertion improves the overall tree by moving some time series to more appropriate leaf nodes.

For traversal and splitting, several algorithms are proposed in the following, whereas Chapter 6 evaluates them in detail. A schematic illustration of the overall insertion algorithm is given in Figure 4.3.

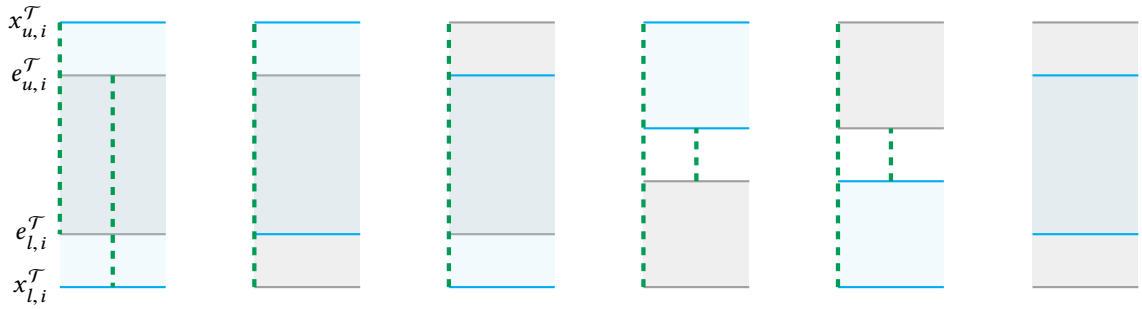



**Figure 4.4** Schematic illustration of *InsertionCost*, where the dashed green lines visualize the distance. The $i$-th image corresponds to the $i$-th case of Equation 4.3.

### 4.3.1 Traversal

For traversing the index tree from the root to a leaf node to insert a time series, the optimal subtree to follow needs to be determined at each tree level. For each child node of the current node, the cost of inserting the segmented time series $X^{\mathcal{T}}$ into the child's segmented envelope $E^{\mathcal{T}}$ is calculated. The subtree below the node for which the cost is minimal is then traversed. Different cost functions, each focusing on other aspects, are proposed as follows.

**Area after insertion** This is the most simple cost function and equivalent to the envelope area after inserting the segmented time series into the segmented envelope. It results in choosing the smallest envelope.

$$AreaAfterInsertion\left(E^{\mathcal{T}}, X^{\mathcal{T}}\right) = \sum_{i=1}^{|E^{\mathcal{T}}|} \max\left(e^{\mathcal{T}}_{u,i}, x^{\mathcal{T}}_{u,i}\right) - \min\left(e^{\mathcal{T}}_{l,i}, x^{\mathcal{T}}_{l,i}\right) \tag{4.2}$$

**Insertion cost** This function is based on the cost function proposed by TWIST (cf. Equation 3.4 on page 11) and is illustrated in Figure 4.4. It is extended to handle segmented representations of the time series and the envelope. The cost is a combination of the envelope area after insertion and the required area enlargement.

$$InsertionCost\left(E^{\mathcal{T}}, X^{\mathcal{T}}\right) = \tag{4.3}$$

$$\sum_{i=1}^{|E^{\mathcal{T}}|} \begin{cases} \left(x^{\mathcal{T}}_{u,i} - e^{\mathcal{T}}_{l,i}\right)^2 + \left(e^{\mathcal{T}}_{u,i} - x^{\mathcal{T}}_{l,i}\right)^2 & \text{if } x^{\mathcal{T}}_{u,i} > e^{\mathcal{T}}_{u,i} > e^{\mathcal{T}}_{l,i} > x^{\mathcal{T}}_{l,i} \quad \left[X^{\mathcal{T}}_i \text{ wraps } E^{\mathcal{T}}_i\right] \\ \left(x^{\mathcal{T}}_{u,i} - e^{\mathcal{T}}_{l,i}\right)^2 & \text{if } x^{\mathcal{T}}_{u,i} > e^{\mathcal{T}}_{u,i} > x^{\mathcal{T}}_{l,i} > e^{\mathcal{T}}_{l,i} \quad \left[X^{\mathcal{T}}_i \text{ overlaps } E^{\mathcal{T}}_i \text{ on top}\right] \\ \left(e^{\mathcal{T}}_{u,i} - x^{\mathcal{T}}_{l,i}\right)^2 & \text{if } e^{\mathcal{T}}_{u,i} > x^{\mathcal{T}}_{u,i} > e^{\mathcal{T}}_{l,i} > x^{\mathcal{T}}_{l,i} \quad \left[X^{\mathcal{T}}_i \text{ overlaps } E^{\mathcal{T}}_i \text{ at bottom}\right] \\ \left(x^{\mathcal{T}}_{u,i} - e^{\mathcal{T}}_{l,i}\right)^2 + \left(x^{\mathcal{T}}_{l,i} - e^{\mathcal{T}}_{u,i}\right)^2 & \text{if } x^{\mathcal{T}}_{l,i} > e^{\mathcal{T}}_{u,i} \quad \left[X^{\mathcal{T}}_i \text{ is above } E^{\mathcal{T}}_i\right] \\ \left(e^{\mathcal{T}}_{u,i} - x^{\mathcal{T}}_{l,i}\right)^2 + \left(e^{\mathcal{T}}_{l,i} - x^{\mathcal{T}}_{u,i}\right)^2 & \text{if } e^{\mathcal{T}}_{l,i} > x^{\mathcal{T}}_{u,i} \quad \left[X^{\mathcal{T}}_i \text{ is below } E^{\mathcal{T}}_i\right] \\ 0 & \text{otherwise} \quad \left[E^{\mathcal{T}}_i \text{ wraps } X^{\mathcal{T}}_i\right] \end{cases}$$

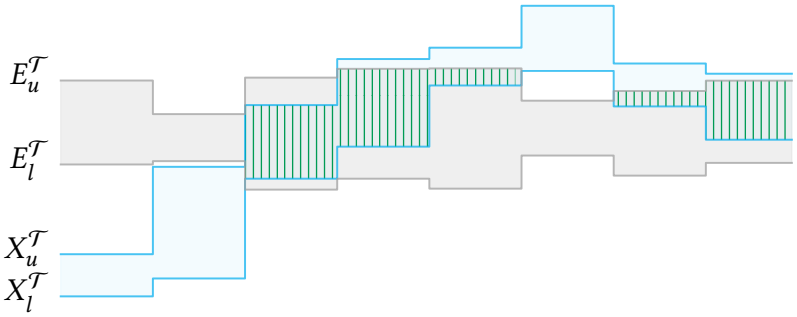



**Figure 4.5** Visualization of the overlap area (||) between a segmented envelope $E^{\mathcal{T}}$ and a segmented time series $X^{\mathcal{T}}$.

**Overlap and insertion cost** As the envelopes of the resulting index tree shall overlap only slightly, it is intuitive to take the overlap into account for creating the index. Therefore, this cost function chooses the subtree for which the total overlap between all candidate subtrees after insertion is minimal. However, if the envelopes of two subtrees do already overlap and the time series to be inserted falls within the overlapping area, the total overlap does not differ, regardless of which subtrees is chosen. Hence, in case of a tie regarding the total overlap, the subtree for which the value of a second cost function is minimal, is chosen for further traversal. Here, the second cost function is *InsertionCost* (cf. Equation 4.3).

The overlap area between a segmented envelope $E^{\mathcal{T}}$ and a segmented time series $X^{\mathcal{T}}$ is illustrated in Figure 4.5 and can be calculated as follows:

$$Overlap\left(E^{\mathcal{T}}, X^{\mathcal{T}}\right) = \sum_{i=1}^{|E^{\mathcal{T}}|} \max\left(0, \min\left(e_{u,i}^{\mathcal{T}}, x_{u,i}^{\mathcal{T}}\right) - \max\left(e_{l,i}^{\mathcal{T}}, x_{l,i}^{\mathcal{T}}\right)\right) \tag{4.4}$$

Notice that this equation also holds for the overlap area between two segmented envelopes, since a segmented time series is conceptually an envelope, too.

**Overlap and area after insertion** Instead of using *InsertionCost* as the second cost function like in the previous paragraph, *AreaAfterInsertion* (cf. Equation 4.2) is used. Thanks to this function, smaller envelopes are preferred and a continuous enlargement of one or more already large envelopes is mitigated, while keeping the total overlapping small.

**Area enlargement (abolished)** For inserting a data object, the original R-tree always chooses the node, whose bounding box needs the least enlargement. While this seems natural at first glance, this often leads to an uneven data distribution. Applied to TSEIT, an envelope that has been enlarged early is larger than other envelopes and thus requires less area enlargement to include further time series. So it is chosen by the algorithm and enlarged again. This continues and causes the algorithm to always—or mostly—choose the same envelope. However, as an index tree with only a few large and full leaf nodes, while the majority is sparse, is not desired, the cost function *area enlargement* is not used.

### 4.3.2 Splitting

A leaf node only holds a limited number of time series to ensure that the index tree grows. If a leaf node exceeds its maximum capacity $l_{\max}$, it is split into two nodes. The node's time series are assigned to one of both, guaranteeing that each new node holds at least $l_{\min}$ time series after splitting. For inner nodes, the number of child nodes is between $c_{\min}$ and $c_{\max}$ which requires splitting in case of an overflow, too. To this, the envelope of each child node is converted to a single time series by computing the central sequence. Thanks to this transformation, the same algorithms can be used for splitting both inner nodes and leaf nodes.

Splitting aims to group similar time series so that the resulting envelopes are small and overlap only slightly. It can also be considered as clustering of time series.

In the following, three different splitting algorithms are proposed, including a novel approach called $k$-envelopes.

#### 4.3.2.1 $k$-Means

$k$-means [Mac67] and its implementation by Lloyd [Llo82] is a simple and popular clustering algorithm that iteratively adds each data objects to the closest of $k$ clusters. Both TSEIT and TWIST use $k$-means clustering for splitting, with $k = 2$ and the Euclidean distance function.

While it was revealed that $k$-means performs well for time series regarding runtime and clustering quality [PG15], it does not allow the definition of a minimum cluster size. Therefore, $k$-means can only be used if $l_{\min} = 1$. For splitting inner nodes, the original $k$-means is not applicable at all, as $c_{\min}$ is required to be at least two.

#### 4.3.2.2 $k$-Envelopes

$k$-envelopes is a novel clustering algorithm based on ideas of Lloyd's $k$-means algorithm [Llo82] and tailored to the needs of TSEIT. Unlike $k$-means, it supports constraints on the minimum or maximum cluster size and thus enables a minimum filling degree for the TSEIT nodes. While $k$-means minimizes the variance within a cluster, $k$-envelopes also allows the optimization of the overlap and the total cluster size by using different cost functions. Similar to $k$-means, sequences are repeatedly assigned to one of two clusters—the envelopes—whereby the initial cluster centers are recalculated after each iteration. For the use with TSEIT, the algorithm is designed for two clusters ($k = 2$), but it can be extended to support an arbitrary number.

Figure 4.6 lists a slightly simplified version of the algorithm as pseudo-code. The sequences are first sorted by their sum. While sorting is by default ascending, the sequences can optionally be sorted alternating, so that the sequence with the smallest sum is followed by the sequence with the largest one and so on $(1, N, 2, (N-1), \ldots$ for $N$ sequences).

```
Algorithm KEnvelopes(sequences[], min_size):
    bsf_labels ← [ ];      // holds for each sequence the index of the envelope it is assigned to
    sequences, sort_indices ← sort_by_sum(sequences);
    prev_centers ← [ sequences.first(), sequences.last() ];
    prev_total_area ← ∞;

    while not maximum number of iterations do
        labels ← [ ];
        envelopes ← [ { upper: center, lower: center } foreach center in prev_centers ];

        foreach sequence in sequences do
            // assign the sequence to the envelope for which the cost is minimal
            min_cost ← ∞;    min_envelope ← None;
            foreach envelope in envelopes do
                cost ← calc_cost(envelope, sequence);
                if cost < min_cost then
                    min_cost ← cost;    min_envelope ← envelope;
                end
            end
            labels.append(min_envelope.index());
            min_envelope.update(sequence);

            // if min_envelope is full, assign remaining sequences to the other envelope
            if min_envelope.size() == sequences.length() − min_size then
                other_envelope ← (min_envelope.index() == 0) ? envelopes[1] : envelopes[0];
                for 1 to min_size - other_envelope.size() do
                    labels.append(other_envelope.index());
                    other_envelope.update(sequences.next());
                end
                break;
            end
        end

        total_area ← envelopes[0].area() + envelopes[1].area();
        if total_area ≥ prev_total_area then break;

        bsf_labels ← labels;
        prev_centers ← [ calc_central_sequence(envelope) foreach envelope in envelopes ];
        prev_total_area ← total_area;
    end

    labels ← undo_sorting(bsf_labels, sort_indices);
    return labels;
```

**Figure 4.6** Slightly simplified *k*-envelopes algorithm for assigning a list of sequences to two envelopes. Unlike this pseudo-code, the full algorithm supports a second cost function in case of a tie and optionally allows alternated sorting.

Alternated sorting aims to improve the clustering quality, as especially in the beginning strongly different time series are assigned successively to one of the envelopes. The initial centers are the first and the last sequence after sorting, or the first two in case of alternated sorting. In this way, the two initial envelopes are prone to have a large distance, which facilitates discrimination and the creation of envelopes with a minimum area and a minimum overlap.

The algorithm then repeatedly executes the following steps as long as the total envelope area decreases and the maximum number of iterations is not reached:

- *Initialize the envelopes by the central sequences.*

  For each envelope, both its upper and lower sequence is set to one of the central sequences of the previous iteration (or to the initial one in the first iteration). Therefore, the envelopes have an initial area of zero.

- *Assign each sequence to the envelope for which the cost is minimal.*

  All cost functions introduced in Section 4.3.1 on page 21 can be used, whereby *AreaAfterInsertion* is the default one, as it tends to create uniform sized envelopes. The assignment is recorded, e.g., using a "label" list mapping each sequence to an envelope index. The chosen envelope $E$ with the upper sequence $E_u$ and the lower sequence $E_l$ is then updated according to the raw time series $X$, similar to Equation 4.1 on page 20:

  $$UpdateEnvelope(E, X) = \begin{cases} E_u \leftarrow \langle \max(e_{u,1}, x_1), \ldots, \max(e_{u,|E|}, x_{|X|}) \rangle \\ E_l \leftarrow \langle \min(e_{l,1}, x_1), \ldots, \min(e_{l,|E|}, x_{|X|}) \rangle \end{cases} \tag{4.5}$$

- *If one envelope is full, assign the remaining sequences to the other envelope.*

  If an envelope already contains the maximum allowed number of sequences, assign the remaining sequences to the other envelope. This ensures that the latter one complies with the minimum cluster size.

- *Stop, if the quality decreases.*

  After having assigned all sequences to an envelope, compute the total area of both envelopes. If the area is larger than in the previous iteration, the iterations are stopped immediately, and the previous labels are used, as they give a local minimum area.

- *Calculate the new central sequence for each envelope.*

  The central sequence is the sequence in the middle between an envelope's upper and lower sequence $E_u$ and $E_l$, respectively:

  $$CentralSequence(E) = E_l + \frac{E_u - E_l}{2} \tag{4.6}$$

  where $+, -, /$ are applied element-wise.

Depending on the implementation, the label list must eventually be re-sorted so that the mapping corresponds to the original sequence order.

Like Lloyd's $k$-means algorithm, $k$-envelopes is a heuristic and, therefore, does not necessarily converge to the global optimum. When the algorithm stops, only a local minimum for the total envelope area might have been found. For $k$-means, it is common to run the algorithm several times with different initial cluster centers that are usually generated with some degrees of randomness. In an early development stage of $k$-envelopes, the envelopes have been initialized by randomly chosen sequences, and even a random processing order was evaluated. However, this has not improved the clustering quality. Furthermore, it has been tested to stop the algorithm only once the labels do not change anymore. However, it turned out that the assignment of time series to the envelopes can oscillate and, moreover, that the overall quality can decrease in later iterations.

$k$-envelopes has an overall runtime complexity of $O(N \log N)$ for splitting a set of $N$ sequences into two groups. Without taking the initial sorting of the input sequences and the final sorting of the labels into account, the complexity amounts to $O(iNn)$, with $i$ iterations and sequences of length $n$. As the number of iterations is usually low—1 to 10 in most cases—and the sequence length is fixed within a dataset, $k$-envelopes is almost a linear time algorithm in practice concerning the number of sequences (that is limited by $l_{\max}$ or $c_{\max}$, anyway).

#### 4.3.2.3 Overlap Split

Overlap Split is an adaption of the splitting algorithm of the R*-tree [Bec+90]. Like $k$-envelopes, it supports a minimum cluster size *min_size* making it suitable for TSEIT (with $l_{\min}$ and $c_{\min}$ for leaf nodes and inner nodes, respectively).

The $N$ sequences are first sorted by their sum. Then, $N - 2 \cdot min\_size + 1$ different splits into two clusters, i.e., envelopes, are calculated, where the first envelope of the $k$-th split contains the first $(min\_size - 1) + k$ sequences, and the second envelope contains the remaining ones. The $k$-th split is illustrated in Figure 4.7. Finally, the split with the minimum overlap between its envelopes and—in case of a tie—the minimum total envelope area, is chosen like in the original approach. An efficient algorithm that only requires a minimum number of costly envelope updates, by iteratively extending the envelopes instead of recalculating them in each iteration, is shown in Figure 4.8.

Similar to $k$-envelopes, the total runtime complexity of Overlap Split is in $O(N \log N)$ for splitting a set of $N$ sequences. Without considering the initial and final sort operations, the complexity is in $O(Nn)$ for $N$ sequences of length $n$.

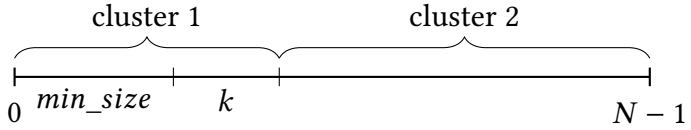



**Figure 4.7** Schematic illustration of the $k$-th split of Overlap Split.

```
Algorithm OverlapSplit(sequences[], min_size):
    N ← sequences.length();
    min_index ← None;    min_overlap ← ∞;    min_area ← ∞;
    sequences, sort_indices ← sort_by_sum(sequences);

    envelope_one ← initialize_envelope(sequences[0]);
    for i from 1 to min_size - 2 do
    |   envelope_one.update(sequences[i]);
    end
    // envelope_one "contains" the first min_size − 1 sequences now

    envelope_two ← initialize_envelope(sequences[N - 1]);
    for i from N − 2 to N - min_size do
    |   envelope_two.update(sequences[i]);
    end
    // envelope_two "contains" the last min_size sequences now

    envelope_two_list ← [ envelope_two ];
    for i from N − min_size − 1 to min_size do
    |   envelope_two_list.append(envelope_two_list.last().copy().update(sequences[i]))
    end
    // envelope_two_list holds N − 2 min_size + 1 precomputed envelopes now

    for k from 0 to N − 2 min_size do // generate and evaluate different splits
    |   i ← min_size − 1 + k;
    |   envelope_one.update(sequences[i]);
    |   envelope_two ← envelope_two_list[envelope_two_list.length() −k − 1];
    |
    |   overlap ← calc_overlap(envelope_one, envelope_two);
    |   if overlap < min_overlap then
    |   |   min_index ← i;    min_overlap ← overlap;
    |   |   min_area ← envelope_one.area() + envelope_two.area();
    |
    |   else if overlap == min_overlap then
    |   |   area ← envelope_one.area() + envelope_two.area();
    |   |   if area < min_area then
    |   |   |   min_index ← i;    min_area ← area;
    |   |   end
    |   end
    end

    labels ← [ 0 for _ from 0 to min_index ].append(
     [ 1 for _ from min_index + 1 to N − 1 ]);
    labels ← undo_sorting(labels, sort_indices);
    return labels;        // holds for each sequence the index of the envelope it is assigned to
```

**Figure 4.8** Overlap Split algorithm for assigning a list of sequences to two envelopes.

### 4.3.3 Reinsertion

Early inserted time series might have a stronger impact on the characteristics of the index tree than later inserted ones. This can be harmful and degrade the query performance if the envelopes are much larger than they would be with a better distribution of the time series among the leaf nodes. Although splitting a leaf node reorganizes time series, it is only a local optimization without taking the overall structure of the tree into account. Even the split of an inner node cannot solve the problem that time series might be contained by not the best fitting subtree, which unnecessarily increases the area of all upper-level envelopes. To overcome these issues, [Bec+90] proposes to force the reinsertion of data objects—the time series in this work—during the insertion process. By deleting time series from a leaf node and reinserting them, they might land up in another part of the tree leading to overall tighter envelopes. While this usually compensates the increase of the insertion time, reinsertion is an optional feature of TSEIT.

Reinsertion is triggered whenever a leaf node for which no reinsertion was already performed exceeds $l_{\max}$, the maximum time series count allowed per leaf. Executing the reinsertion only once per leaf node prevents infinite loops if the time series are reinserted into the same leaf node again. After an envelope's outer 30% of the time series are deleted, all parental envelopes up to the root node are adjusted accordingly. Afterward, the deleted time series are reinserted into the updated tree.

In greater detail, the following steps are performed:

- *Sort the time series by their distance to the envelope center.*

  The envelope center is the central sequence and calculated as in Equation 4.6 on page 25. If the envelope is segmented, the resulting sequence is shorter than the raw time series and needs to be extended by repeating each sequence element $T_{\min}$ times. The last sequence element might be repeated less often. The Euclidean distance is used as distance measure.

- *Remove the last 30% of the time series from the envelope.*

  The last time series after sorting are the outer ones, having the maximum distance to the envelope center. As an envelope is not self-maintainable with respect to deletion (cf. Section 3.3.2.2 on page 11), it needs completely be recreated by the first 70% of the time series. Removing the outer 30% seems reasonable and is borrowed by the R*-tree.

- *Adjust all covering envelopes up to the root node.*

  By deleting time series from a leaf node, the associated envelope usually gets smaller. Therefore, all parent envelopes up to the root must also be adjusted to still enclose their child envelopes tightly. For this, each parent node needs to access the envelopes of all of its child nodes.

  If envelope segmentation is used, the segment length of a parent envelope is twice the child nodes' segment length. Hence, to recreate a parent envelope, all child

envelopes have to be transformed into envelopes with the corresponding length. This is done by combining every two consecutive values of a child envelope $E^{\mathcal{T}}$ as follows:

$$CombineValues\left(E^{\mathcal{T}}\right) = \begin{cases} E_u^{\mathcal{T}} \leftarrow \left\langle \max(e_{u,1}^{\mathcal{T}}, e_{u,2}^{\mathcal{T}}), \max(e_{u,3}^{\mathcal{T}}, e_{u,4}^{\mathcal{T}}), \ldots, \max(e_{u,n-1}^{\mathcal{T}}, e_{u,n}^{\mathcal{T}}) \right\rangle \\ E_l^{\mathcal{T}} \leftarrow \left\langle \min(e_{l,1}^{\mathcal{T}}, e_{l,2}^{\mathcal{T}}), \min(e_{l,3}^{\mathcal{T}}, e_{l,4}^{\mathcal{T}}), \ldots, \min(e_{l,n-1}^{\mathcal{T}}, e_{l,n}^{\mathcal{T}}) \right\rangle \end{cases} \tag{4.7}$$

Alternatively, the parent envelope can first be recreated with the original child envelopes and be likewise transformed afterward.

- *Insert the deleted time series into the tree.*

  The reinsertion of the orphaned time series is now performed as usual.

## 4.4 *k*-NN Querying

The following describes the interval-based search for the $k$-NN time series using the TSEIT index. An illustration of the overall procedure is presented in Figure 4.9.

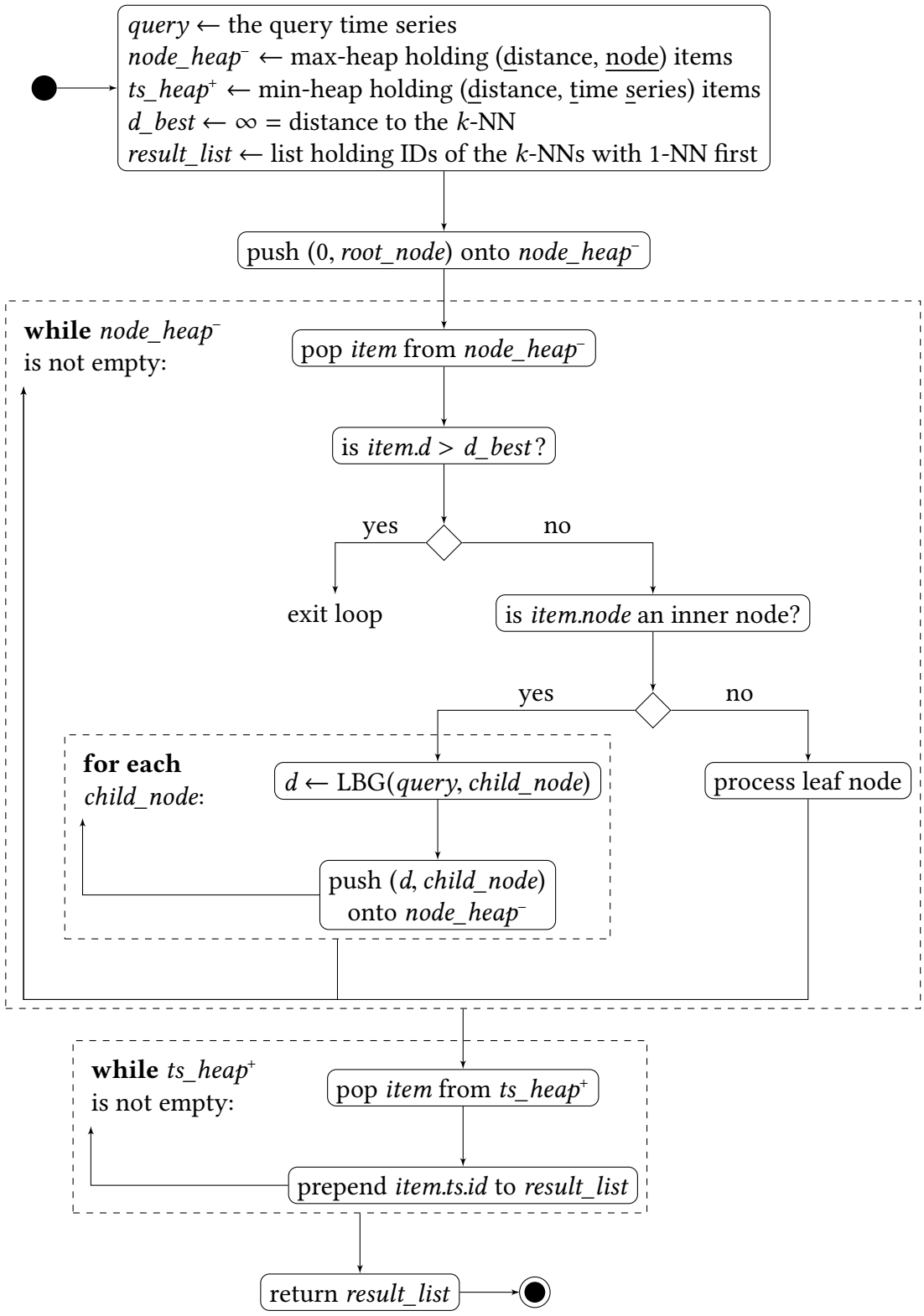
Given a time series $X$, a time interval from $a$ to $b$ and a $k \geq 1$, the $k$ time series $Y_i, \ldots$ for which the distance $DTW(X[a{:}b], Y_i[a{:}b])$ is minimal, shall be found. $X[a{:}b]$ denotes the subsequence of $X$ from $a$ to $b$ inclusive:

$$X[a{:}b] = \langle x_a, x_{a+1}, \ldots, x_{b-1}, x_b \rangle \text{ with } 1 \leq a \leq b \leq n = |X| \tag{4.8}$$

### 4.4.1 Fundamental Query Algorithm

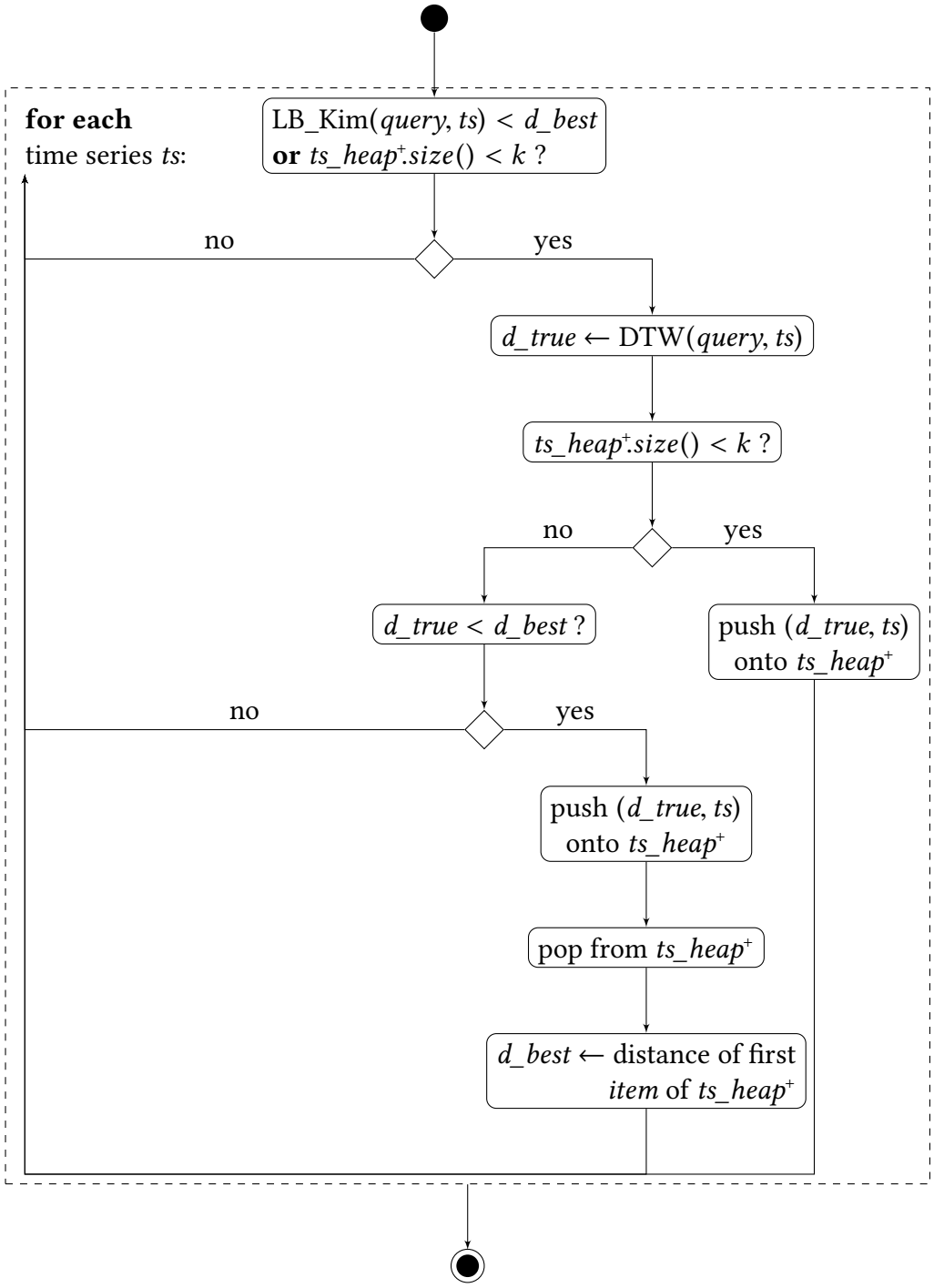
The main idea of the query algorithm is to calculate lower-bound distances to envelopes, representing entire subtrees, and to follow only those with a distance smaller than the current best-so-far distance. The best-so-far distance is the distance between the query time series and the $k$-nearest neighbor. Note that for all distance calculations only the subsequence within the specified time interval is considered. For receiving the closest candidate node, the algorithm maintains a min-heap holding nodes together with their lower-bound distance to the query time series. The heap invariant ensures that the first node in the heap is always the one with the smallest distance.

Initially, each child node of the root node is pushed onto the min-heap together with the LBG lower-bound distance of its envelope to the query time series. As long as the min-heap is not empty, the first and thus closest node is popped, in other words, returned and removed from the heap. The algorithm stops if the closest node's lower-bound distance is larger than the best-so-far distance. In this case, all remaining nodes in the queue have an even larger distance and cannot hold any nearest neighbor time series. If the popped node is an inner node, all of its child nodes are pushed together with their lower-bound




**(a)** The overall querying process.

**Figure 4.9** Activity diagram of the $k$-NN querying process.




**(b)** The activity *process leaf node* only.

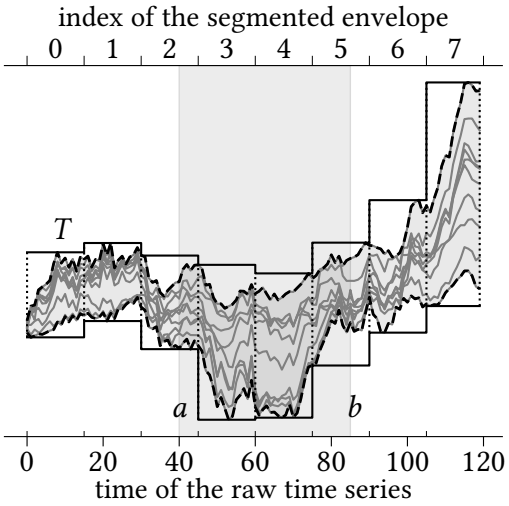



**Figure 4.10** Exemplary segmented envelope with segment length $T$, along with a query interval from $a$ to $b$ specified in the time space of the raw time series as annotated by the bottom axis. The top axis describes the indices of the upper and lower envelope sequence.

distances onto the min-heap. Whenever a leaf node is popped from the heap, all of its time series are accessed. For each time series, a lower bound for the distance to the query time series is calculated. While LB_Kim is used in this work, other lower-bound functions introduced in Section 3.2.2 on page 9 can be used, alternatively or additionally. If the lower bound is already larger than the best-so-far distance, the time series cannot be one of the $k$-NN and is therefore omitted. However, whenever the lower bound is smaller, the exact but computational extensive DTW distance is calculated. In the case that also the exact distance is smaller than the best-so-far distance, the time series is pushed onto a max-heap along with the distance. The max-heap holds the $k$ (or less) closest time series among the ones examined so far, where the first one is the farthest away. Afterward, the first time series in the max-heap is removed, as it has a larger distance to the query than the just inserted time series. Finally, the best-so-far distance is set to the distance of the new first time series in the heap. The algorithm then continues with the next time series of the current leaf node, or the next candidate node if any exits. It stops popping nodes from the min-heap, once the heap is empty or once the first node has a lower-bound distance larger than the best-so-far distance. In this case, all remaining nodes have an even larger distance and thus cannot hold the nearest neighbors. The max-heap now contains the $k$-NN time series, that can be obtained in descending order by repeated pop operations.

Appendix A.3 on page 80 lists the execution of exemplary $k$-NN queries.

### 4.4.2 Querying with Segmented Envelopes

With segmentation enabled, the query sequence is segmented while traversing the tree. The segment length equals to $T = \lfloor \min(n, 2^l \cdot T_{\min}) \rfloor$ for the tree level $l \geq 0$, where the leaf nodes are at level 0. However, the first and last segment in the query interval

might be covered only partially. Moreover, the segment length and thus the length of the envelope sequences differs for different tree levels. All of this must be taken into account for calculating the lower-bound distance between the query and an envelope, and for comparing envelopes at different tree levels.

Figure 4.10 illustrates these circumstances for an envelope with eight segments of length $T = 15$ along with a query interval from $a = 40$ to $b = 85$, and time series of length 120. This results in segment lengths of 5, 15, 15 and 10 for the four segments from index 2 to 5 in the query interval.

To support different segment lengths in the query interval, the LBG lower-bound distance of TWIST (cf. Section 3.3.3.2 on page 12) is modified as in [SYF05]. Instead of multiplying the distance between two segments by the current level's full segment length $T$, the minimum of the lengths $T_i$ and $T_j$ of the segments $i$ and $j$, respectively, is used. This lower bounds the true distance between the segments. The complete LBG lower bounds the distance between the segmented query sequence $X^{\mathcal{T}}$ and the segmented envelope $E^{\mathcal{T}}$ including the time series the latter holds. The full equation is given below for the sake of clarity.

$$LBG\left(X^{\mathcal{T}}, E^{\mathcal{T}}\right) = D_{n,m} \tag{4.9}$$

with

$$D_{0,0} = 0, \quad D_{i,0} = D_{0,j} = \infty$$

$$D_{i,j} = d\left(X_i^{\mathcal{T}}, E_j^{\mathcal{T}}\right) + \min(D_{i,j-1}, D_{i-1,j}, D_{i-1,j-1})$$

$$d\left(X_i^{\mathcal{T}}, E_j^{\mathcal{T}}\right) = \min\left(T_i, T_j\right) \cdot \begin{cases} \left|x_{l,i}^{\mathcal{T}} - e_{u,j}^{\mathcal{T}}\right|^2 & \text{if } x_{l,i}^{\mathcal{T}} > e_{u,j}^{\mathcal{T}} \\ \left|e_{l,j}^{\mathcal{T}} - x_{u,i}^{\mathcal{T}}\right|^2 & \text{if } e_{l,j}^{\mathcal{T}} > x_{u,i}^{\mathcal{T}} \\ 0 & \text{otherwise} \end{cases}$$

# 5 Implementation

This chapter introduces details on the implementation of TSEIT. First, the third-party tools used are presented, followed by the overall architecture and further insights into various parts of the developed application, including performance optimizations. Finally, a technical evaluation concerning code quality and runtime behavior is given.

## 5.1 Tools and Languages

The time series and the TSEIT index are stored in a PostgreSQL database and maintained by a server-side application written in Python. Furthermore, a client-side module provides several helper tools. In the following, the database system is introduced first, before the programming language Python and libraries used are presented.

### 5.1.1 PostgreSQL

PostgreSQL[1] is one of the most popular relational database systems[2] and available under an open source license. It is founded on the relational data model, where data is organized in tuples that are grouped into relations.

PostgreSQL provides an interface for *Generalized Search Trees* (GiST) [HNP95] that allows the creation of custom balanced index trees by implementing a set of predefined methods. While this seems useful for implementing the TSEIT index tree at first glance, GiST indexes are not flexible enough and do not allow debugging, as they are managed fully internally by the database system. Therefore, TSEIT does not use the GiST interface.

In this work, PostgreSQL is used in version 9.6 to store the time series and the TSEIT index itself. In contrast to other relational database systems such as MySQL, it supports writing custom functions in Python instead of procedural SQL, which is a major reason why it was chosen.

[1] Homepage of PostgreSQL: `https://www.postgresql.org/`

[2] Database ranking: `https://db-engines.com/en/ranking` (archived in January 2018: `https://web.archive.org/web/20180103030915/https://db-engines.com/en/ranking`)

#### 5.1.1.1 PL/Python

The PL/Python[3] procedural language allows writing PostgreSQL functions in Python that are executed by the operating system's Python interpreter. As it is possible to use any Python module that is present in the `PYTHONPATH`, PL/Python functions can be arbitrary powerful and sophisticated. Furthermore, the language module automatically imports a module for accessing the database that also provides several utility methods for logging. While both Python 2 and Python 3 are supported, the latter is used in this work.

For security reasons, PL/Python is only available as *untrusted language* requiring database superuser permissions to define functions of that type. PL/Python functions are executed with the permissions of the database administrator, instead of with the possibly restricted permissions of the database user calling the functions. Thus, a malicious or flawed PL/Python function might provoke privilege escalation or data leaks by reading or manipulating sensitive data in the file system or database.

### 5.1.2 Python

Python[4] is an interpreted general-purpose programming language with a convenient syntax and semantic. Together with a considerable number of excellent libraries, it is well-suited for rapid development. Among other paradigms, it supports functional and object-oriented programming.

In this work, Python in version 3.5 is widely used for initializing the database, inserting time series, maintaining the index structure, performing $k$-NN queries and evaluating the index.

Python applications can be structured by modules and packages. A Python module is a file containing definitions and statements, whereas a Python package is a module which can contain submodules or subpackages that are saved in the same directory. A collection of packages or modules is often referred to as library.

#### 5.1.2.1 NumPy

In comparison to compiled languages such as C or Fortran, traversing lists or multidimensional arrays with standard Python loops is slow. Remedy offers the Python library NumPy[5] by providing a special multidimensional array type for fast vectorized arithmetic operations, usually without the need for loops. Following the *single instruction, multiple data* paradigm [Fly72; Dun90], NumPy operates on whole blocks of data instead of processing array elements individually. Moreover, NumPy is the foundation of several scientific libraries for machine learning and data analysis.

[3] Documentation of PL/Python: `https://www.postgresql.org/docs/9.6/static/plpython.html`
[4] Homepage of Python: `https://www.python.org/`
[5] Homepage of NumPy: `http://www.numpy.org/`

The main reason for the efficiency of NumPy arrays is their internal memory layout. While the items of a standard Python list are spread across the system memory, the data of a NumPy array is stored in a contiguous block of memory. Together with the condition that all array items must be of the same type and size, this allows accessing every part of the array by some simple arithmetic on the address of the memory block. The NumPy arrays and most operations on them are written in the low-level language C, as it performs index arithmetic efficiently. Furthermore, storing the data as blocks enables optimizations of the CPU, leading to additional performance gains. Many access patterns like an array traversal or manipulation have spatial locality, meaning that it is likely that nearby memory locations will be accessed soon. Since the CPU usually loads not only individual elements but additionally some adjacent elements into its cache, the cache is always hit in these scenarios. Some CPUs even implement vectorized arithmetical operations that can operate on NumPy's memory blocks as efficient CPU instructions.

With this in mind, it is not surprising that in this work, the time series and envelope sequences are processed as NumPy arrays. Furthermore, almost all `for`-loops for traversal are replaced by vectorized NumPy operations. Despite a small overhead for converting standard Python lists to NumPy arrays, this gives a significant performance gain.

#### 5.1.2.2 Further packages

In addition to NumPy, this work uses the following Python packages. The first two are used on the server side, whereas the other packages are used on the client side.

**scikit-learn** Built on top of NumPy, scikit-learn[6] is a powerful library for data mining and data analysis, including a *k*-means implementation that is used in this work.

**FastDTW** The Python module FastDTW[7] implements the same-named algorithm by [SC07]. Besides a fast approximated DTW distance by constraining the search radius, an exact DTW search is supported.

**Psycopg** Psycopg[8] is an adapter mostly written in C for accessing a PostgreSQL database from within Python modules.

**Graphviz** The graph visualization tool Graphviz[9] is used together with a Python adapter[10] for debugging the structure of the index tree. The graphs are specified with the graph description language DOT.

**Matplotlib** Matplotlib[11] is a general-purpose plotting library for Python and used to visualize time series and envelopes, as well as the insertion progress over time.

[6] Homepage of scikit-learn: `http://scikit-learn.org/`
[7] Code repository of the FastDTW implementation for Python: `https://github.com/slaypni/fastdtw`
[8] Homepage of Psycopg: `http://initd.org/psycopg/`
[9] Homepage of Graphviz: `https://graphviz.gitlab.io/`
[10] Code repository of the Graphviz interface for Python: `https://github.com/xflr6/graphviz`
[11] Homepage of Matplotlib: `https://matplotlib.org/`

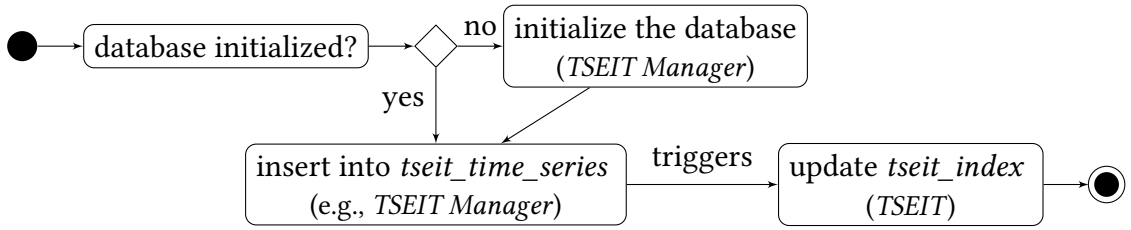



**Figure 5.1** Once the database has been initialized using the *TSEIT Manager* Python package, each insertion of a time series into the *tseit_time_series* table triggers an update of the *tseit_index* table by the *TSEIT* package.

**pandas** pandas[12] (in lower case) provides efficient data structures and analysis tools for tabular data. In this work, it is mainly used for processing CSV files holding evaluation results.

**XlsxWriter** XlsxWriter[13] allows generating Excel spreadsheets by writing Python code. As part of this work, it is used to convert raw CSV files to spreadsheets with conditional formatting for an easier perception of evaluation results.

## 5.2 Architecture

The reference implementation of TSEIT stores the time series and the index structure itself inside two PostgreSQL tables. As shown in Figure 5.1, a PL/Python trigger function maintains the index by updating the index table after each insertion of a time series into the time series table. The $k$-NN search is implemented as a PostgreSQL function written in PL/Python, too, and can be executed by a particular SQL statement. A server-side Python package called *TSEIT* implements the PL/Python functions, whereas a client-side package called *TSEIT Manager* provides several tools for database initialization, time series insertion, monitoring, and analysis.

Many publications introducing novel index structures such as [NRR10] or [ZIP16] store all data, including the index, in plain text files. Using a database system like PostgreSQL instead, allows making use of the provided data structures and efficient retrieval methods. The use of trigger functions and stored procedures, instead of an external client application that connects to the database to both insert time series and to update the index table, does not necessarily result in a significant performance gain. However, it allows using any database client without the need to know that an index is being managed in the background. Whereas using the *TSEIT Manager* is one option, time series can be inserted in any manner, e.g., using a graphical user interface or a custom script, locally or remote. After initialization, TSEIT is transparent to the user when running insertion and $k$-NN search statements in the database.

[12] Homepage of pandas: `http://pandas.pydata.org/`
[13] Code repository of XlsxWriter: `https://github.com/jmcnamara/XlsxWriter`

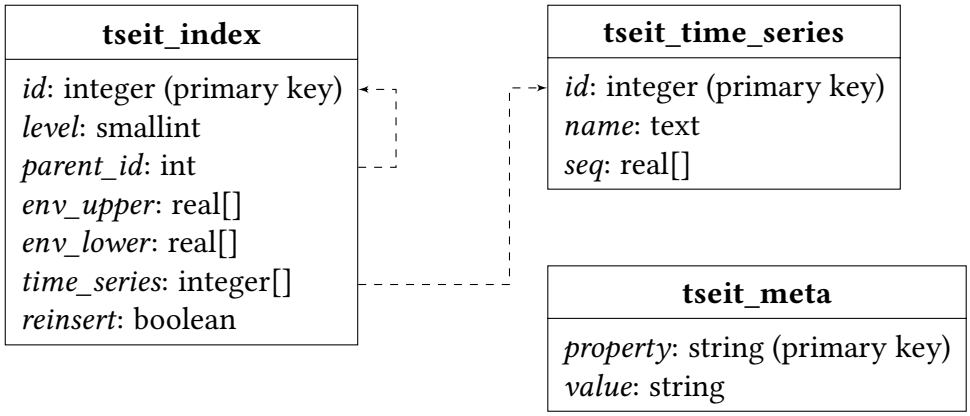



**Figure 5.2** The database schema of TSEIT. For performance reasons, the illustrated pointers are not implemented with foreign keys constraints, but maintained by the *TSEIT* module. The table *tseit_meta* is read-only.

## 5.3 Database Design

TSEIT's database contains two major tables and a read-only meta table with information on the configuration and creation time, as illustrated in Figure 5.2. The time series are stored as arrays of floating-point numbers in the table *tseit_time_series* together with an optional name and an auto-generated numeric identifier (ID). The index tree is held by the table *tseit_index* and represented as an adjacency list, where each table row corresponds to a tree node. Each node refers to the ID of its parent node, except the former is the root node. Furthermore, each row contains the level of the corresponding node. An envelope is stored as separate upper and lower sequences, each as an array of floats. Finally, a Boolean flag indicates whether reinsertion was already performed for the node. If the node is a leaf node, a further column contains an array of numeric IDs, each referring to a time series in the *tseit_time_series* table.

For performance reasons, the parent ID and the time series IDs are not maintained by foreign keys. Foreign key constraints are useful in many applications and allow, for example, cascaded deletions. However, they add overhead due to additional integrity checks. Since only the TSEIT implementation is accessing the index table, it can dispense with the foreign keys.

To improve the performance of tree traversal, an internal index is created on the ID and the parent ID column. For each index, PostgreSQL is building a B-tree specialized for high-concurrency [LY81] that holds column values together with identifiers to the physical row locations.

While there are numerous ways to represent hierarchical structures in the relation model, each has its own drawbacks. Adjacency lists enable efficient retrieval of the immediate parent or child node of a given node. Furthermore, inserting and removing single nodes is straightforward. On the downside, retrieving *all* child or parent nodes is expensive, as it requires either a join per tree level or a recursive query. However, this access scenario

does not exist for TSEIT since both the insertion and the query algorithm never process nodes from multiple levels at once. A fast tree traversal in descending or ascending level order is more critical and enabled by the chosen structure.

One-to-many relationships like from one leaf node to many time series are usually mapped by a separate table in order not to violate the first normal form [Cod70] by holding non-atomic values. However, as always *all* time series of a leaf node are processed, e.g., for recalculating the envelope after deletion or for a $k$-NN search, and as it is not necessary to find the leaf that holds a specific time series, the introduced design is sufficient.

## 5.4 TSEIT

The *TSEIT* package is the key component, as it implements the index creation and the $k$-NN query algorithm. It is built as a distributable and installable module that is called by the PL/Python functions. The insertion trigger, for example, basically consists of an `import` statement and a call to the `run` method of *TSEIT* passing the new row of the *tseit_time_series* table. Global data, such as the user-defined configuration or cached query plans, is shared using a global module-level variable that can be imported by any submodule. This is a standard approach and also applied by large frameworks such as Django.

*TSEIT* is lightweight and organized by subpackages that group similar modules. It follows the procedural programming paradigm and dispenses with a complex object-oriented class hierarchy, as it does not introduce great advantages, but adds overhead. Exceptions are the splitting algorithms $k$-envelopes and Overlap Split, which are implemented in object-oriented style to provide the same interface as the $k$-means module from scikit-learn. Tree nodes and time series are represented as dictionaries—associative arrays that map keys to values. Since they have the same structure as the database tables, no additional object-relational mapping is required.

### 5.4.1 Performance Optimizations

Even an efficient algorithm can have a poor runtime in practice if the implementation is not well-designed. When dealing with massive datasets, every tiny performance gain for processing a single time series can have a significant impact on the overall runtime. Thanks to the numerous minor and major code optimizations presented below, the final implementation has no avoidable bottlenecks and is around ten times faster than the first prototypes.

Alternative solutions concerning single methods have been evaluated using Python's `timeit` module that allows measuring the execution time of small code snippets. The overall implementation has been examined using `cProfile`, which provides statistics on how long and often different parts of a program are executed. Profiling results are presented at the end of this chapter in Section 5.6.2 on page 46.

```python
def overlap_naive(env_one, env_two):
    overlap = 0.0
    for u1, l1, u2, l2 in zip(env_one['upper'], env_one['lower'],
                              env_two['upper'], env_two['lower']):
        overlap += max(0.0, min(u1, u2) - max(l1, l2))
    return overlap

def overlap_vectorized(env_one, env_two):
    return ((np.minimum(env_one['upper'], env_two['upper']) - # element-wise
             np.maximum(env_one['lower'], env_two['lower']))  #  min / max
            .clip(min=0).sum())
```

**(a)** Computing the size of the overlap area of two given envelopes.

```python
def segment_naive(sequence, segment_length):
    segments_upper, segments_lower = [], []
    for i in range(len(sequence) // segment_length):
        x = i * segment_length
        y = x + segment_length
        sub_sequence = sequence[x:y]
        segments_upper.append(max(sub_sequence))
        segments_lower.append(min(sub_sequence))
    return (segments_upper, segments_lower)

def segment_vectorized(sequence, segment_length):
    # transform to `len(sequence)/segment_length × segment_length` matrix
    matrix = sequence.reshape(-1, segment_length)
    return (matrix.max(axis=1), matrix.min(axis=1)) # min / max per row
```

**(b)** Segmenting a given sequence. For the sake of simplicity, it is assumed in this example that the sequence length is a multiple of the segment length.

**Figure 5.3** Examples for standard Python and corresponding vector-based implementations using NumPy.

**Vector-based operations** The traditional way of processing each value of a list is to loop through the data structure. However, as standard Python loops are executed by an interpreter, the runtime performance is poor, especially compared to compiled languages such as C. Operating on entire sequences instead of accessing single items, is the core concept of NumPy. Replacing loops by vector-based NumPy operations can reduce the execution time to only $2 - 15\%$ of the original time, depending on the complexity and size of the input. Usually, the larger the input, the more pays the application of NumPy off.

Nearly all loops for traversing sequences or envelopes have been replaced by NumPy-based vector operations. Figure 5.3 shows two different methods optimized in this way.

```
def process(el, kind):
  if kind == 'env':
    # process envelope
  else:
    # process sequence

for el in data:
  process(el, kind)
```

**(a)**

```
for el in data:
  if kind == 'env':
    process_env(el)
  else:
    process_seq(el)
```

**(b)**

```
if kind == 'env':
  process = process_env
else:
  process = process_seq

for el in data:
  process(el)
```

**(c)**

**Figure 5.4** Various ways to write conditions on the type when iterating through the input data. While (a) and (b) require a test in each iteration, variant (c) tests the type only once before looping.

**Compute all segmentations at once** While the segmented representation of an envelope or sequence can already be computed efficiently using NumPy, the performance can be further improved by taking advantage of the fact that the segment length doubles with each upper tree level. By combining every two consecutive values of the representation for the level below, it is not necessary to compute the segments for each level independently, and the length of the sequences to process is reduced with each level.

**Efficient iterators** It is not always possible to replace slow Python loops by efficient vector-based operations. The built-in Python module `itertools` provides a set of functional tools for creating and using fast and memory-efficient iterators. All of them are implemented in C. This module allows, for example, replacing complex nested loops with a single one.

**Abandoning conditionals** Repeated `if` statements, especially inside loops, can degrade the performance, but are sometimes easy to replace. At many places in the code, it is necessary to differentiate whether the input data describes sequences or envelopes. For example, the Overlap Split algorithm iteratively updates the target envelopes by either input sequences or by input envelopes. Instead of testing the input type in each iteration to decide which function to call (cf. Figure 5.4a and 5.4b), the test can be moved outside the loop (cf. Figure 5.4c). Depending on the type, the proper function is then bound to a local variable and called inside the loop without the need for repeated tests.

**Optimize SQL queries** When a query is issued, the database system needs to parse it and create an execution plan. By using prepared statements, this work has to be done only once per database session, since subsequent calls use an already compiled plan. Therefore, *TSEIT* creates a prepared statement for each query the first time it is issued, and stores a reference in the global configuration module for reusing the statement subsequently. Using the global configuration module for caching the prepared statement is possible, as the *TSEIT* module is loaded only once per session and not for each single time series insertion (assuming that multiple time series are inserted in one session). This approach improves the total runtime by around 10%.

Using an index on the `parent_id` column is clearly advantageous, as it speeds up the traversal, especially for large index trees.

The SQL queries themselves are quite simple CRUD statements without complex joins or recursion and therefore offer only little room for optimization. Nevertheless, improvement is possible. The table *tseit_index* contains a column holding for each leaf node a list of time series IDs. As the column is not filled for inner nodes, it is not necessary to always include it in the result set. Receiving it only when required slightly improves the runtime.

### 5.4.2 Configurable Parameters

The TSEIT index is highly configurable, following the variations introduced in Chapter 4. With a configuration file, the user can specify, among other things, the topology of the tree, the splitting algorithms or the cost functions. The default configuration file together with all possible options is listed in Appendix A.1 on page 77.

### 5.4.3 TWIST

To allow a fair comparison between TSEIT and TWIST, both are implemented on the same code basis. The implementation of TWIST thus shares the design with TSEIT and uses the same data structures and helper methods, e.g., for accessing the database, including the mentioned optimizations. The use of TWIST can be switched per configuration file.

## 5.5 TSEIT Manager

The *TSEIT Manager* is a client-side companion module for *TSEIT*. It provides several utility modules, whereby most of them can be executed independently from each other from the command line. The provided modules are briefly introduced in the following.

**`create_database`** This module provides the routines for initializing *TSEIT* by creating the database, tables and trigger functions.

Optionally, the following PostgreSQL helper functions can be installed. As they require additional PostgreSQL extensions and special permissions they are not created by default.

- `get_relation_sizes()` returns the relation sizes in bytes and pretty-printed. A relation's total size is the sum of the actual table size, the size of related TOAST data (*The Oversized-Attribute Storage Technique*), plus the size of related indexes.
- `prewarm()` calls `pg_prewarm()` for each table of TSEIT to load them into the buffer cache of PostgreSQL. This is useful for accelerating querying, especially after a database restart.

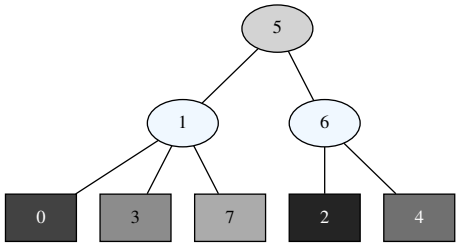



**Figure 5.5** Generated plot of a small TSEIT tree, where the numbers are the node IDs. The darker a rectangle is, the more time series the corresponding leaf node contains.

- `get_buffer_usage()` returns for each TSEIT table the size of related pages in the buffer cache, together with the ratio of cached data to the total relation size.

All SQL queries are not hard-coded in the module code but moved to separate template files, which are plain SQL scripts with `{}` placeholders that are replaced by concrete values using Python's `format()` string method.

`helpers.`**`insert_ts_into_db`** This module allows inserting time series from a CSV file into the TSEIT index. The size of a transaction and a connection, both in terms of the number of time series, can optionally be set. It is furthermore possible to write the progress, including the number of inserted time series and the current node count, into a CSV file in intervals. The monitoring is performed in a separate thread with a function that calls itself every given number of seconds.

When inserting many new rows in one single long-living database session, the memory usage of PostgreSQL grows continuously. This behavior can not only be observed for the *TSEIT* trigger functions, but also for simple stored PostgreSQL functions written in procedural SQL instead of PL/Python. Therefore, it is *not* likely that the increase of memory usage is caused by a bug in the implementation of TSEIT. A solution to avoid running out of memory is reestablishing the connection in intervals. By default, a new connection is established after 100,000 insertions.

`helpers.analysis.`**`visualize_progress`** This module provides a method for generating a plot with Matplotlib that visualizes the insertion progress. It includes the number of processed time series per second, TSEIT-related events such as splittings and reinsertions, as well as PostgreSQL-related events like auto-vacuum executions and reconnects. It makes use of pandas for parsing and filtering a PostgreSQL CSV log file and the CSV file generated with the logging option of `insert_ts_into_db`. For retrieving PostgreSQL events, it is necessary that PostgreSQL writes its logs into a CSV file (`logging_collector = on` and `log_destination = 'csvlog'`) and that the logging of auto-vacuum actions is enabled (`log_autovacuum_min_duration = 0`).

`helpers.analysis.`**`evaluate_index`** This module provides several methods for evaluating the characteristics of the built index tree. Besides statistics on the nodes and distribution of time series among the leaf nodes, the total overlap between the leaf nodes can be computed. Moreover, the module allows plotting the index tree like in Figure 5.5 using Graphviz, and it allows creating plots of envelopes and time series using Matplotlib. The distribution of nodes per level and time series per leaf node can additionally be plotted.

`helpers.analysis.`**`integrity_tests`** This module implements the following tests that shall ensure the integrity of the created index tree.

- Test, whether the number of time series held by the leaf nodes is equal to the number of time series in the *tseit_time_series* table.
- Test, whether the envelope of each leaf node wraps the time series it holds. This is done by calculating the target envelope for a leaf node's time series and by comparing the resulting upper and lower envelope sequences with the ones stored for the leaf node.
- Test, whether the envelope of each inner node wraps the envelopes of its child nodes. This is done analogously to the previous test.

`helpers.analysis.`**`config_tester`** The configuration tester makes it possible to build and evaluate the TSEIT index with different configurations. It optionally generates all parameter combinations specified in a particular configuration file. For example, with the following settings, six different indexes are created and evaluated with three different splitting algorithms and two different cost functions for traversal.

```
split_algo_leaf_node = kenvelopes, overlap_split, kmeans
choose_subtree_cost_fct = overlap_than_area, overlap_than_insertion_cost
```

The evaluation results are finally written to a CSV file, which is additionally converted to an Excel spreadsheet using XlsxWriter with columns containing metric values appropriately formatted. For example, in the metric column `total_overlap`, the cells' background colors are interpolated from green for the smallest value to red for the largest value, indicating the quality of the value.

`helpers.analysis.`**`analyze_config_values`** This module provides a method for analyzing the results of the configuration tester to facilitate the detection of suitable configuration values. For this, average normalized metric values are calculated for each configuration value. A metric is, for example, the total overlap area or the node count. In Figure 5.6 following the configuration tester settings above, the option `split_algo_leaf_node` is to be analyzed. First, multiple row groups are filtered so that the rows of a group only differ in the column of interest. Per row group, the metrics are then transformed to values in the interval $[0, 1]$, in order to make the metric values of different groups comparable. Finally, the mean of the normalized metrics is calculated for each possible value of the target configuration option.

**`preprocessing`** This folder contains a specific Python script for converting the datasets that are used for evaluating TSEIT to CSV files. It furthermore holds two general shell scripts for sorting a CSV file, optionally alternating, and for sampling rows of a CSV file by taking every $i$-th line and optionally shuffling them in the end.

| split_algo_leaf_node | choose_subtree_cost_fct | reinsertion | metric1 | ⟶ metric1_norm |
|---|---|---|---|---|
| kenvelopes | overlap_than_area | True | 0.1 | 0.0 |
| overlap_split | overlap_than_area | True | 0.3 | 0.23 |
| kmeans | overlap_than_area | True | 0.8 | 1.0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| kenvelopes | overlap_than_insertion. | True | 1 | 0.0 |
| overlap_split | overlap_than_insertion. | True | 7 | 1.0 |
| kmeans | overlap_than_insertion. | True | 4 | 0.5 |

**Figure 5.6** Example supporting the introduction of the module `analyze_config_values`. The mean normalized values for `metric1` regarding `split_algo_leaf_node` are 0.0, 0.62, 0.75 for `kenvelopes`, `overlap_split` and `kmeans`, respectively. If lower values are better for `metric1`, this result would indicate that `kenvelopes` is the best option.

## 5.6 Technical Evaluation

For the implementation of TSEIT, it was placed importance on high code quality. This is confirmed by static and dynamic program analysis as explained in the following.

### 5.6.1 Static Code Analysis

The source code satisfies Python's PEP8 style guide[14] and follows reasonable recommendations by the static code analysis tool Pylint[15]. Compliance with established coding standards improves not only the readability but also the maintainability. Pylint, furthermore, helps to detect code smells and possible implementation errors.

The Python tool Radon[16] allows computing the cyclomatic complexity [McC76]. The metric corresponds to the number of linearly independent paths through the code and should be low to support code comprehension. It is incremented with each statement that introduces a decision, e.g., `if` or `for`. The *TSEIT* package has an average cyclomatic complexity of 3.5, whereas it is 6.4 for the *TSEIT Manager* (and 3.1 without the `helpers` submodule). A cyclomatic complexity of up to 10 indicates a simple structure [For+97].

Certainly, code standards and metrics give only a hint on the actual quality of a software. However, they, facilitate the detection of problematic parts of the code, which then can be improved. This has been the case for the implementation process of TSEIT.

### 5.6.2 Profiling

Profiling is a dynamic program analysis technique for examining the runtime behavior of a program. For Python, `cProfile` monitors how often and how long methods of a program

[14] PEP 8 – Style Guide for Python Code: `https://www.python.org/dev/peps/pep-0008/`
[15] Homepage of Pylint: `https://www.pylint.org/`
[16] Radon in the Python package index: `https://pypi.python.org/pypi/radon`

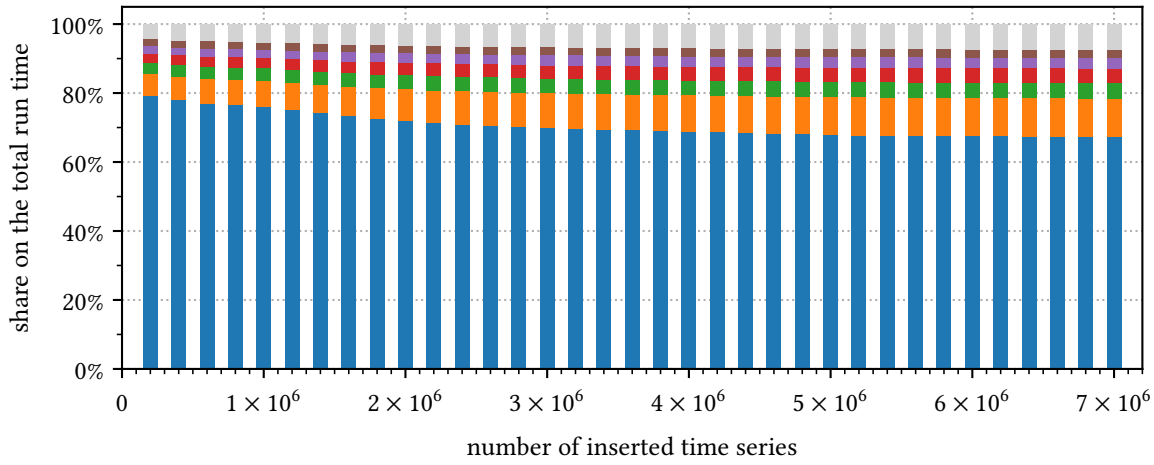



**Figure 5.7** Relative share of different operations on the accumulated runtime for inserting 7 million time series. The stacked bars describe from top to bottom:

- miscellaneous operations (4.2% – 7.4%),
- split a leaf node (2.3% – 2.4%),
- convert a Python list to a NumPy array (2.1% – 3.2%),
- calculate the area after insertion (2.7% – 4.2%),
- compute the segmentation of a time series for all levels (3.2% – 4.4%),
- calculate the overlap after insertion (6.3% – 11.3%),
- execute a database query (79.3% – 67.2%).

are executed. It is embedded in the *TSEIT* module and can be enabled in the configuration file. As the insertion of one single time series takes only a small fraction of a second, it is subject to fluctuations of the runtime. Therefore, the statistics for single insertions are aggregated for reliable results. The insertion routine of *TSEIT* is wrapped by statements for starting and stopping the profiling. Subsequently, a dump of previous profiling runs is loaded from a file, updated by the latest run, and resaved to disk.

The profiling results in Figure 5.7 using TSEIT's default configuration show that around 70% to 80% of the total runtime are spent for executing the prepared database statements. This number can be interpreted in two ways: either the database is quite slow, or the remaining parts of the program are very efficient. As the share of the query execution time on the total runtime has been significantly lower (30% to 50%) in early stages of development, it suggests that the remaining *TSEIT* code is indeed high-performing since its share has reduced. Notice that both the creation of prepared statements and the splitting of inner nodes are not shown in the plot, as their shares amount to merely 0.0002% and 0.06%, respectively.

Moreover, the absolute runtimes over the entire insertion process show that the execution of database queries takes longer for the first one million time series, before stabilizing at a lower value after around three million inserts. The execution time for database queries furthermore fluctuates more than the execution time of other operations.

# 6 Evaluation

This chapter evaluates the index creation and querying process, and presents detailed results of numerous test runs. First, the test environment and the datasets used are introduced. A target definition is given subsequently, before metrics describing the index and query evaluation are presented. An intensive test series evaluating different configurations for TSEIT is discussed hereinafter, including the effects of specific parameters. Following this, the consequences of various insertion orders are revealed. Runtime behavior and the size of the index structure is explained afterward, before the insertion and querying of more than 100 million time series is presented. The chapter ends with a comparison between TSEIT and the competing approach TWIST.

## 6.1 Setup

### 6.1.1 Environment

Most tests—especially all runtime tests—were executed on a virtual machine with the following hardware and software specification:

- host system:
  - Intel Xeon CPU E5-2630 v4, with a single core performance of 2.20 GHz
  - iSCSI over 10 GBit/s Ethernet
  - $8 \times 8$ TB (Seagate ST8000DM005 with 7,200 rpm) as RAID 10
  - Citrix XenServer 7.1 (hypervisor)
- guest system:
  - 62.9 GB of RAM (+ 1.34 GB of swap)
  - 345 GB data partition for PostgreSQL
  - Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-109-generic x86_64)
  - PostgreSQL server 9.6.5
    (the configuration file is listed in Appendix A.2 on page 79)
  - Python 3.5.2

### 6.1.2 Datasets

The datasets used for evaluating the TSEIT index are subsets of the *Google Books* American English unigram (or 1-gram) and bigram (or 2-gram) dataset from July 2012[1]. An $n$-gram is a contiguous sequence of $n$ words from a given sentence. For example, the sentence "This is great" contains the unigrams "This", "is" and "great", as well as the bigrams "This is" and "is great". The original datasets give for each $n$-gram occurring in a corpus of books the number of occurrences per year, from 1505 to 2008. This results in one time series per $n$-gram. As more books are published in later years, the counts are normalized by dividing them by the number of books published in each particular year.

The following three subsets are used for evaluation. $n$-grams containing special characters or additional annotations concerning the word type are left out.

**7-million dataset** This dataset contains 7,015,720 time series of length 209. They are created from the unigram dataset based on the American English corpus, version 2012, for the years 1800 to 2008 inclusive. The 1-grams satisfy the regular expression `[^\W\d_]+` and thus neither contain any character that is not a word character, nor any decimal digit or underscore. The uncompressed dataset has a size of 11 GB.

**1-million dataset** This dataset contains every 7-th time series of the 7-million dataset in the original order for the years 1800 to 2000 inclusive, which results in 1,000,951 time series of length 201. The uncompressed dataset has a size of 1.5 GB.

**103-million dataset** This dataset holds 103,462,118 time series of length 209. It is based on the bigram dataset of the American English corpus, version 2012, and contains bigrams for the years 1800 to 2008 inclusive. The 2-grams satisfy the regular expression `[a-zA-Z ]+` and thus contain only ASCII letters and spaces. From the 10,000 time series with the largest sequence sum, every 100-th time series is moved to the beginning of the dataset, as this is required for a special test concerning $k$-NN queries. The uncompressed dataset has a size of 155 GB.

#### 6.1.2.1 Characteristics of the Datasets

As provided by *Google Books*, the $n$-grams are basically sorted by their first letter. However, $n$-grams starting with the same letter are not always in perfect order. This almost corresponds to a random order of the time series concerning their value ranges, with some exceptions, e.g., for common $n$-grams starting with "the" or "a".

The 7-million dataset contains far more unigrams than the English language has words. Besides of containing rare words such as names, this is caused by errors in the text recognition when creating the corpus on which the original dataset is based on. Therefore, most of the time series have many tiny values close or equal to zero, as illustrated in Figure 6.1. Similar observations hold for the bigram dataset. The uneven distribution

[1] The *Google Books* n-gram datasets are freely available at `https://books.google.com/ngrams/`

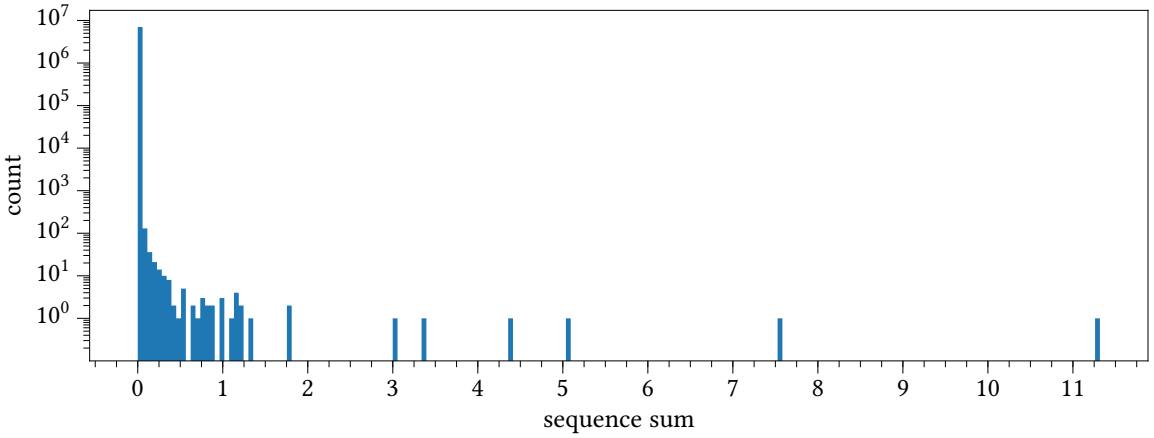



**Figure 6.1** Distribution of time series in the 7-million dataset. Note that the y-axis is log-scaled. More than 99.99% of the time series have a sequence sum less than 0.05, represented by the first bar. 90% of the time series have a sequence sum less than $10^{-6}$, while around 75% have a sum less than $2 \times 10^{-7}$.

makes indexing and querying more challenging, especially for those time series that are located in high-density areas where the sequences are concentrated.

### 6.1.3 *k*-NN Queries

Besides other metrics, the quality of the index is evaluated by the effort for different $k$-NN queries. The query time series are sampled from the 1-million dataset and are executed with $k = 1$, and interval limits $a = 1$ and $b = 201$. Since the length of the interval corresponds to the full length of the time series in the dataset, this is a worst-case scenario. With smaller intervals, the querying effort is usually smaller as the query time series tend to hit fewer leaf node envelopes.

Due to the highly uneven distribution of time series in the dataset, that contains a huge number of rare $n$-grams which are unlikely to be queried in practice, two different samples are used for evaluation. While Sample 1 contains time series of existing words, that are probably more interesting in actual practice, Sample 2 covers the remaining low-value time series. Taken together, the entire value range is covered.

**Sample 1** This sample holds 30 time series from the 1-million dataset for which the sequence sum is $\geq 10^{-3}$. After sorting in descending order, every 52-th time series is chosen from the 1,545 time series that satisfy the condition. The sample contains time series representing real words such as “he” or “amazing”.

**Sample 2** This sample holds 20 time series from the 1-million dataset for which the sequence sum is $< 10^{-3}$. After sorting in descending order, every 49,970-th time series is chosen from the 999,406 time series that satisfy the condition. The sample contains time series representing misspelled words and rare names such as “employd” or “Mauville”.

| Parameter | | Default Value |
|---|---|---|
| min. number of child nodes | $c_{\min}$ | 2 |
| max. number of child nodes | $c_{\max}$ | 3 |
| min. number of time series per leaf node | $l_{\min}$ | 1 |
| max. number of time series per leaf node | $l_{\max}$ | 1000 |
| segmentation | | enabled |
| minimum segment length | $T_{\min}$ | 1 |
| cost function for traversal | | overlap and area after insertion |
| reinsertion | | enabled |
| splitting algorithm for leaf nodes | | $k$-envelopes |
| splitting algorithm for inner nodes | | $k$-envelopes |
| $k$-envelopes: cost function | | area after insertion |
| $k$-envelopes: alternating sorting | | disabled |

**Figure 6.2** The default configuration of TSEIT.

### 6.1.4 Default Configuration

If not mentioned otherwise, the default configuration as shown in Figure 6.2 is used for building a TSEIT index.

## 6.2 Metrics

The primary aim of using an index structure is to accelerate querying. In this work, the runtime for $k$-NN querying is mainly influenced by two factors: the number of tree nodes to which an LBG lower-bound distance is calculated, and the number of exact DTW computations. An efficient query visits only a minimal number of inner nodes and leaf nodes, and calculates the DTW distance to merely a fraction of all time series.

Solely minimizing the DTW count does not necessarily result in the best performance, as this can favor trees with many leaf nodes holding only a few time series. In the worst case, each leaf node contains only one time series. This, on the one hand, leads to few DTW calculations, as most leaf nodes can already be eliminated with the LBG lower-bound distance. On the other hand, traversal costs are high, and the LBG calculations still cause effort. Similar effects occur vice versa. Therefore, both the LBG and the DTW count should be minimized.

It is assumed that a tree with well-filled leaf nodes and small envelopes that overlap only slightly leads to optimal query performance. A large tree with many poorly filled leaf nodes provokes high traversal cost, whereas large envelopes are more likely to be hit by a query time series, even though they do not necessarily contain a $k$-NN time series.

| *k*-NN Metrics | | Index Metrics | | | |
|---|---|---|---|---|---|
| | | total / leaf node count | mean number of t.s. per leaf | mean leaf density | mean leaf area |
| mean LBG count | Sample 1 | 0.89 | −0.76 | −0.79 | −0.62 |
| | Sample 2 | 1.00 | −0.96 | −0.89 | −0.83 |
| mean DTW count | Sample 1 | −0.40 | 0.73 | 0.68 | 0.75 |
| | Sample 2 | −0.52 | 0.70 | 0.63 | 0.69 |

**Figure 6.3** Pearson correlation coefficients for the relationship between index metrics and *k*-NN metrics for all test runs of this work. A value of 0 means no correlation, a value of ±1 means a perfect linear relationship. As the total node count and the leaf node count have almost equal correlation coefficients, they are combined in this table.

### 6.2.1 Index and *k*-NN Metrics

To evaluate the quality of an index, the following metrics are obtained. The terms "envelope" and "node" are used synonymously.

- index metrics:
  - number of nodes, leaf nodes, and tree levels
  - leaf area
  - number of time series per leaf node
  - density of a leaf node: number of contained time series divided by its area
  - total overlap area of the leaf nodes
- *k*-NN metrics:
  - number of LBG, LB_Kim, and DTW calculations

More sophisticated metrics such as the total length of the subsequences within overlapping envelope areas were less informative than expected. They were therefore discarded.

### 6.2.2 Correlation between Metrics

Performing and evaluating many *k*-NN queries can be more time-consuming than actually building the index. Thus, it is desirable to have a metric on the built index that gives an estimate on the querying effort. For this reason, the Pearson correlation between the *k*-NN metrics LBG and DTW count, and index metrics such as the leaf count or overlap was computed regarding almost all tests ever performed for this work.

Figure 6.3 shows the strongest correlations between the *k*-NN metrics and index metrics. The results are intuitive: the more nodes an index tree has, the more LBG calculations are required. On the other hand, fewer DTW calculations are needed, as many nodes—and thus time series—can already be discarded by the LBG distance. The other index metrics

themselves are more or less correlated to the node count. The more nodes, the fewer time series are held by the leaf nodes, which is a negative correlation. Leaf nodes with a low density have either large envelopes or contain only a few time series resulting in many nodes, again. The mean leaf area is also inversely correlated to the LBG count; probably, as small envelopes tend to contain few time series, which leads to more nodes to which an LBG distance needs to be calculated. On the other hand, this reduces the DTW count for the same reasons as above.

Interestingly, the overlap between leaf nodes was not strongly correlated to the $k$-NN metrics in most test series. This could be caused by the fact that the time series in the used datasets are highly unevenly distributed, which is why overlapping cannot be avoided. In contrast, for the test series presented in Figure 6.4 on page 57, the total leaf overlap is strongly positively correlated to the LBG and DTW count for Sample 1 by 0.89 and 0.72, respectively. However, for Sample 2, no strong correlation is measurable (0.25 and 0.32 for the LBG and DTW count, respectively). Note that in this test series the overlap and thus the correlation was not computed for all runs.

## 6.3 Parameter Evaluation

Chapter 4 introduces many parameters concerning the index construction. There are different tree topologies, multiple cost functions and splitting algorithms, and the optional reinsertion and segmentation. This results in a large number of possible configurations for building a TSEIT index. Using metrics describing the resulting index tree and executed $k$-NN queries, the numerous parameter combinations were evaluated.

Figure 6.4 on page 57 lists detailed evaluation results for runs with different parameter values, whereby some parameters are fixed for all runs. Latter ones are introduced first, before the effects of different parameter values are discussed.

As implied by the table, using the default configuration, TSEIT calculates the DTW distance to averagely 0.1% and 19.5% of the time series of Sample 1 and Sample 2, respectively.

### 6.3.1 Common Parameter Values

**Maximum number of time series per leaf node** First tests showed that for the 1-million dataset, a maximum of 1,000 time series per leaf node results in fewer DTW calculations than a larger limit of 5,000 or 10,000 time series. Thus, all following tests used $l_{\max} = 1{,}000$.

**Number of child nodes** Regarding the minimum and the maximum number of child nodes, $c_{\min}$ and $c_{\max}$ respectively, the results were less clear after tests with $(c_{\min}, c_{\max})$ values of $(2, 3)$, $(3, 5)$ and $(5, 10)$. As the results differed not drastically, $c_{\min} = 2$ and $c_{\max} = 3$ were chosen for subsequent tests since results tended to be slightly better with these values.

**Reinsertion** After determining that reinsertion clearly improves the overall results, it was used for all following test runs.

**Segmentation** Segmentation with a minimum segment length of $T_{\text{min}} = 1$ was used for most tests, as this is a core concept of TSEIT to save processing time and storage space. Moreover, further tests indicated that the segmentation does not necessarily increase the LBG and DTW count significantly.

### 6.3.2 Varying Parameter Values

**Minimum number of time series per leaf node** Not enforcing a minimum number of time series per leaf nodes results in smaller leaf envelopes that furthermore overlap less. This is not surprising, as with $l_{\text{min}} = 1$, the splitting algorithms do not necessarily have to assign time series with very different value ranges to the same node. However, only the splitting algorithms $k$-envelopes and, to some extent, $k$-means can produce a compact tree, where the leaf nodes are well-filled. Overlap Split with $l_{\text{min}} = 1$ leads to leaf nodes that contain on average fewer than 35 time series, although up to 1,000 are allowed. As a consequence, the resulting tree is very large and deep, which especially increases the number of required LBG calculations and thus degrades the runtime. For $k$-envelopes it is crucial to use the cost function *area after insertion* for assigning time series to one of the two envelopes. This ensures, in contrast to the other possible cost functions, that the nodes are in most cases evenly sized after splitting. With these settings, $k$-envelopes creates leaf nodes holding merely a few time series, only if necessary.

A larger minimum time series count of $l_{\text{min}} = 300$ does not necessarily impair the results, as it guarantees that the tree does not grow extensively. It limits the number of nodes and thus the number of required LBG calculations for $k$-NN queries.

By default, $l_{\text{min}} = 1$ as it gives good results together with a proper configuration of $k$-envelopes. It furthermore works well with an arbitrary value for $l_{\text{max}}$. Moreover, additional tests not listed here have shown that smaller lower limits lead to better results on average.

**Cost function for traversal** Concerning the cost function for choosing the subtree to insert a new time series (cf. Section 4.3.1 on page 21), the results are quite clear. Previous tests already showed that the simple cost functions *area after insertion* and *insertion cost* do not yield good results. However, applying those as a second cost function after the *overlap* cost function significantly improves the outcome. Comparing *overlap and area after insertion* and *overlap and insertion cost*, the table clearly shows that the former works better, especially for the $k$-NN metrics. In case of a tie for the overlap, the former cost function chooses the envelope that would be smaller after insertion, while the latter also takes the delta into account. *overlap and area after insertion* also results in 15% fewer leaf nodes on average, which consequently hold more time series. Since this is usually a preferred property, that function is used by default.

**Splitting algorithm for leaf nodes** Regarding runs with a minimum leaf size $l_{\text{min}} = 300$, the Overlap Split algorithm is superior to $k$-envelopes in many metrics. It produces leaf nodes with smaller areas, the highest density, and still small overlap. On the other hand, the leaf nodes often contain fewer time series on average, and the number of required

| | min. leaf size | traversal cost fct. | split leaf node | split inner node | $k$-env. cost fct. | $k$-env. alternating | leaf node count | mean leaf density | mean leaf area | max. leaf area | total leaf area | total leaf overlap | Sample 1: mean LBG count | Sample 1: mean DTW count | Sample 2: mean LBG count | Sample 2: mean DTW count |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 300 | oa | ke | ke | a | × | **1,409** | 1.3e+9 | 6.8e−4 | 0.92 | 0.96 | **1.59** | 25 | 1,296 | **2,369** | 194,387 |
| ★ | 1 | oa | ke | ke | a | × | 1,420 | 1.3e+9 | 4.3e−4 | 0.29 | 0.61 | 1.72 | 32 | 1,172 | 2,384 | 194,845 |
| | 300 | oa | ke | os | a | ✓ | 1,434 | 1.4e+9 | 7.9e−4 | 0.92 | 1.13 | 2.32 | 41 | 1,949 | 2,425 | 196,848 |
| | 1 | oa | ke | ke | a | ✓ | 1,435 | 1.6e+9 | 4.5e−4 | 0.28 | 0.65 | 1.81 | 37 | 1,474 | 2,427 | 198,617 |
| | 1 | oa | ke | os | a | ✓ | 1,440 | 1.4e+9 | 4.5e−4 | 0.28 | 0.64 | 1.88 | 37 | 1,332 | 2,447 | 195,013 |
| | 300 | oa | ke | ke | a | ✓ | 1,450 | 1.2e+9 | 7.8e−4 | 0.92 | 1.13 | 2.39 | 38 | 1,949 | 2,471 | 196,808 |
| | 300 | oa | ke | ke | oa | ✓ | 1,460 | 1.3e+9 | 8.5e−4 | 0.92 | 1.25 | 2.17 | 26 | 1,204 | 2,484 | 198,369 |
| | 300 | oa | ke | ke | oi | ✓ | 1,467 | 1.2e+9 | 8.5e−4 | 0.92 | 1.25 | 2.18 | 28 | 1,204 | 2,490 | 195,342 |
| | 300 | oa | ke | ke | i | ✓ | 1,487 | 9.2e+8 | 7.6e−4 | 0.92 | 1.13 | 2.54 | 40 | **3,102** | 2,514 | 203,537 |
| | 300 | oa | ke | ke | oi | × | 1,498 | 1.3e+9 | 6.4e−4 | 0.92 | 0.96 | 1.88 | 29 | 1,405 | 2,525 | 199,039 |
| | 300 | oa | os | ke | a | ✓ | 1,491 | 1.5e+9 | 6.4e−4 | 0.92 | 0.96 | 1.85 | 26 | 1,332 | 2,540 | 199,595 |
| | 300 | oa | ke | ke | i | × | 1,505 | 1.1e+9 | 6.4e−4 | 0.92 | 0.96 | 1.92 | 27 | 1,389 | 2,540 | 200,618 |
| | 300 | oa | ke | ke | oa | × | 1,494 | 1.3e+9 | 6.4e−4 | 0.92 | 0.96 | 1.89 | 29 | 1,405 | 2,548 | 198,659 |
| | 300 | oa | os | os | – | – | 1,503 | **1.6e+9** | 6.4e−4 | 0.92 | 0.96 | 1.87 | **25** | 1,341 | 2,562 | 196,866 |
| | 300 | oi | os | ke | a | ✓ | 1,525 | 1.5e+9 | 6.3e−4 | 0.92 | 0.96 | 1.90 | 26 | 1,445 | 2,655 | 203,202 |
| | 1 | oi | ke | ke | a | ✓ | 1,537 | 1.1e+9 | 4.1e−4 | 0.28 | 0.63 | 2.33 | 44 | 1,729 | 2,658 | 198,440 |
| | 300 | oi | ke | ke | a | × | 1,533 | 1.2e+9 | 6.2e−4 | 0.92 | 0.96 | 1.91 | 27 | 1,351 | 2,694 | 194,827 |
| | 300 | oi | ke | os | a | ✓ | 1,540 | 1.0e+9 | 1.2e−3 | 0.92 | 1.86 | 7.66 | 72 | 2,680 | 2,707 | 201,222 |
| | 300 | oi | ke | ke | a | ✓ | 1,544 | 1.2e+9 | **1.2e−3** | 0.92 | **1.90** | **8.15** | 71 | 2,610 | 2,710 | 199,486 |
| | 300 | oi | os | os | – | – | 1,551 | 1.4e+9 | 6.2e−4 | 0.92 | 0.96 | 2.03 | 25 | 1,440 | 2,724 | 201,552 |
| | 1 | oi | ke | os | a | ✓ | 1,573 | 1.0e+9 | 4.1e−4 | 0.28 | 0.64 | 2.60 | 47 | 1,818 | 2,744 | 198,194 |
| | 1 | oi | ke | ke | a | × | 1,587 | 1.1e+9 | 3.9e−4 | 0.29 | 0.62 | 2.47 | 39 | 1,169 | 2,779 | 197,050 |
| | 300 | oi | ke | ke | oi | ✓ | 1,697 | 1.0e+9 | 8.0e−4 | 0.92 | 1.37 | 3.15 | 40 | 1,674 | 2,983 | 199,995 |
| | 300 | oi | ke | ke | i | ✓ | 1,698 | 9.1e+8 | 8.6e−4 | **0.92** | 1.46 | 7.03 | 81 | 2,729 | 2,996 | 203,324 |
| | 1 | oa | km | ke | a | × | 1,787 | 1.0e+9 | 3.2e−4 | 0.29 | **0.57** | 2.29 | 28 | 1,276 | 3,044 | 198,426 |
| | 300 | oi | ke | ke | oa | ✓ | 1,721 | 9.5e+8 | 6.5e−4 | 0.92 | 1.11 | 2.76 | 36 | 1,512 | 3,073 | 197,269 |
| | 300 | oi | ke | ke | oi | × | 1,775 | 1.1e+9 | 5.5e−4 | 0.92 | 0.97 | 2.59 | 32 | 1,350 | 3,151 | 196,836 |
| | 300 | oi | ke | ke | oa | × | 1,783 | 1.0e+9 | 5.5e−4 | 0.92 | 0.97 | 2.60 | 32 | 1,350 | 3,163 | 197,655 |
| | 300 | oi | ke | ke | i | × | 1,883 | 9.1e+8 | 5.2e−4 | 0.92 | 0.97 | 2.89 | 29 | 1,395 | 3,401 | 201,302 |
| | 1 | oi | km | ke | a | × | 2,088 | 5.1e+8 | 2.7e−4 | 0.29 | 0.57 | 3.07 | 32 | 1,272 | 3,765 | 212,329 |
| | 1 | oa | ke | ke | i | ✓ | 26,574 | 3.1e+8 | 2.6e−5 | 0.27 | 0.69 | * | 109 | 1,204 | 47,052 | 187,066 |
| | 1 | oi | ke | ke | i | ✓ | 28,350 | 3.3e+8 | 2.4e−5 | 0.26 | 0.67 | * | 95 | 1,497 | 51,005 | 177,879 |
| | 1 | oa | ke | ke | oa | ✓ | 34,029 | 3.5e+8 | 1.8e−5 | 0.24 | 0.60 | * | 122 | 1,098 | 52,189 | 181,496 |
| | 1 | oi | os | os | – | – | 28,799 | 5.4e+8 | 2.3e−5 | 0.24 | 0.67 | * | 116 | 1,100 | 52,967 | 176,882 |
| | 1 | oa | os | os | – | – | 28,661 | 6.4e+8 | 2.4e−5 | 0.26 | 0.69 | * | 121 | **888** | 53,054 | 176,180 |
| | 1 | oa | ke | ke | oi | ✓ | 34,422 | 3.7e+8 | 1.7e−5 | 0.24 | 0.59 | * | 124 | 1,040 | 53,344 | 177,548 |
| | 1 | oa | ke | ke | i | × | 36,723 | 1.8e+8 | 2.0e−5 | 0.24 | 0.74 | * | 237 | 1,140 | 54,367 | 192,193 |
| | 1 | oa | ke | ke | oa | × | 36,258 | 2.3e+8 | 2.2e−5 | 0.26 | 0.79 | * | 241 | 1,169 | 54,780 | 190,774 |
| | 1 | oi | os | ke | a | ✓ | 30,094 | 6.9e+8 | 2.4e−5 | 0.24 | 0.72 | * | 130 | 1,252 | 55,491 | 176,214 |
| | 1 | oa | os | ke | a | ✓ | 30,084 | 5.1e+8 | 2.3e−5 | 0.24 | 0.71 | * | 112 | 1,030 | 56,124 | **173,510** |
| | 1 | oa | ke | ke | oi | × | 36,879 | 1.5e+8 | 2.1e−5 | 0.26 | 0.79 | * | 337 | 1,305 | 56,630 | 195,797 |
| | 1 | oi | ke | ke | oi | ✓ | 37,181 | 2.6e+8 | 1.6e−5 | 0.24 | 0.59 | * | 119 | 1,201 | 58,271 | 184,826 |
| | 1 | oi | ke | ke | oa | ✓ | 38,098 | 3.0e+8 | 1.5e−5 | 0.24 | 0.58 | * | 107 | 1,350 | 59,717 | 177,252 |
| | 1 | oi | ke | ke | i | × | 52,163 | **1.2e+8** | **1.4e−5** | 0.24 | 0.70 | * | 280 | 1,386 | 81,609 | 192,220 |
| | 1 | oi | ke | ke | oi | × | 53,818 | 1.6e+8 | 1.5e−5 | 0.24 | 0.81 | * | 382 | 1,095 | 83,244 | 184,224 |
| | 1 | oi | ke | ke | oa | × | **54,858** | 1.4e+8 | 1.4e−5 | 0.24 | 0.79 | * | 395 | 1,081 | **85,999** | 185,475 |
| TWIST: | | | | | | | 2,134 | 7.9e+8 | 2.8e−4 | **0.22** | 0.60 | 3.69 | **2,557** | 2,345 | 12,301 | **213,498** |

★ default configuration, – not required, * not calculated (too many leaf nodes)

**Figure 6.4** *(on previous page)* Table with evaluation results for different configurations of TSEIT with the 1-million dataset, ordered by the column *Sample 2: mean LBG count*.

The following abbreviations are used:

Cost functions (cf. Sec. 4.3.1, p. 21):
- a: area after insertion
- i: insertion cost
- oi: overlap and insertion cost
- oa: overlap and area after insertion

Splitting algorithms (cf. Sec. 4.3.2, p. 23):
- km: *k*-means
- ke: *k*-envelopes
- os: Overlap Split

The runs share the following settings (time series per leaf holds for TWIST, too):
- min. number of child nodes: 2
- max. number of child nodes: 3
- max. number of t.s. per leaf: 1,000
- segmentation: enabled
- min. segment length: 1
- reinsertion: enabled

The green-yellow-red shading indicates the goodness of a value in regard to the other values of the respective column, while the blue shades are for better visual distinction only. The minimum and maximum values per column are highlighted in bold. The LBG and DTW counts refer to a set of *k*-NN queries (cf. Section 6.1.3 on page 51) and are rounded to the nearest whole number.

LBG and DTW calculations is higher, especially for Sample 2. The *k*-envelopes algorithm, on the other hand, performs better on the *k*-NN queries and supports $l_{\min} = 1$, which in general allows a better adaption of the leaf envelopes to the dataset. *k*-means performs well on Sample 1, as it produces envelopes of a small area. However, it produces 25% to 60% more leaf nodes, which especially degrades the performance for Sample 2. To sum up, the novel *k*-envelopes algorithm developed in this work gives the best results overall.

**Splitting algorithm for inner nodes** Both *k*-envelopes and Overlap Split perform well on splitting inner nodes. As in this test series splitting is always performed with four envelopes only (due to $c_{\max} = 3$), both algorithms often yield the same results. Comparing runs that only differ in the splitting algorithm for inner nodes, *k*-envelopes tends to produce trees with slightly more leaf nodes which, however, are smaller and overlap marginally less. Furthermore, the number of LBG and DTW calculations is in many cases slightly lower. Hence, for the evaluation dataset, the best results were achieved using the novel algorithm *k*-envelopes.

Remember that *k*-means cannot be used for splitting inner nodes, as it does not support a minimum cluster size. However, this is required, as $c_{\min}$, the minimum number of child nodes, must be at least two.

**Settings for *k*-envelopes** Using *k*-envelopes for splitting leaf nodes together with a minimum leaf size $l_{\min} = 1$, the results clearly indicate that the cost function *area after insertion* performs best. Using any of the other three cost functions results in a degenerated tree with a huge number of sparse leaf nodes, that provoke high traversal costs. For larger values of $l_{\min}$, the results are less clear since all cost functions lead to good results.

| insertion order | leaf node count | mean leaf density | median leaf area | total leaf overlap | Sample 1: mean LBG count | Sample 1: mean DTW count | Sample 2: mean LBG count | Sample 2: mean DTW count |
|---|---|---|---|---|---|---|---|---|
| original | **1,420** | 1.3e+9 | 1.4e−6 | 1.72 | 32 | 1,172 | 2,384 | **194,845** |
| shuffled | 1,426 | **1.4e+9** | **1.4e−6** | **1.69** | **30** | **1,102** | 2,400 | 195,202 |
| descending | 1,444 | 1.2e+9 | **1.6e−6** | 2.67 | 54 | **1,323** | **2,347** | **213,057** |
| alternating | **1,472** | **1.2e+9** | 1.6e−6 | **2.79** | **59** | 1,141 | **2,405** | 202,566 |

**Figure 6.5** Metrics of runs with differing insertion order using the default configuration. The test run with shuffled insertion order produces few large leaf nodes, which is why the median is listed instead of the mean value.

Sorting the sequences in alternating order before assigning them to the clusters or envelopes aims to facilitate the creation of strongly differing clusters. With a minimum leaf size $l_{\min} = 300$, the algorithm is forced to fill one envelope once the other has reached its maximum size. In this case, the remaining sequences usually have a larger distance to each other than they would have with ascending order. This generally results in larger envelopes as listed in the table. With alternating order, the DTW count is higher for Sample 1, but lower for Sample 2. Ascending order, on the other hand, leads to more, yet smaller leaf nodes.

## 6.4 Insertion Order

Like many index trees such as the R-tree, TSEIT is influenced by the order in which the data is inserted. While a TSEIT tree is always height-balanced, the node count, area and overlap of the leaf nodes can differ. Figure 6.5 lists metrics after having inserted the time series in original order, shuffled, or ordered by descending or by alternating sequence sum (schematic: $N, 1, (N-1), 2, \ldots$ for $N$ sequences). Remember, that the original order is almost random regarding the value ranges as explained in Section 6.1.2.1 on page 50.

The results indicate that a random insertion order works best and that there is no benefit in sorting the time series in the first place. However, although there are differences, especially for the DTW count with Sample 2, they are not that strong with respect to the corresponding value ranges of the table in Figure 6.4.

## 6.5 Insertion Time

Using a tree structure together with segmentation and numerous performance optimizations of the implementation are worth it. TSEIT is able to index large numbers of time

| *indexing 7 million time series* | insertion time | mean insertion rate | index size |
|---|---|---|---|
| TSEIT default | 13.5 h | 144 ts/sec | 56 MB |
| TSEIT without segmentation | 25 h | 77 ts/sec | 65 MB |
| TWIST *(estimation)* | > 770 h | < 2.5 ts/sec | ~ 55 MB |

**Figure 6.6** Comparison of the runtime for inserting the time series from the 7-million dataset, together with the total size of the index table *tseit_index*, including the PostgreSQL index on the column `parent_id`. TSEIT was built with the default configuration, once without segmentation; TWIST with a maximum node size of 1,000 time series.

TWIST was manually canceled after having inserted 2.8 million time series in 190 hours (cf. Figure 6.9 on page 64), resulting in an index size of 22 MB. The values in the table are optimistically extrapolated assuming a constant—non-decreasing–insertion rate of 2.0 ts/sec (the mean rate of the last 6 hours before aborting) for the remaining 4.2 million time series. The true insertion time would be even higher than estimated.

series at a nearly constant insertion rate. As illustrated in Figure 6.7, more than 7 million time series are indexed in 13.5 hours with an average insertion rate of 145 time series per second. This is a remarkable result, especially in comparison to TWIST that during the same time only indexes around 11% of the data, as described in Section 6.8 on page 63. The plot furthermore shows that reinsertions and splittings of leaf nodes occur evenly distributed, indicating a reasonable index creation. Inner node splittings sometimes occur slightly more frequently, which, however, can be explained by the propagation of splittings.

It seems that the increase of the insertion rate after around one million insertions is caused by PostgreSQL, which executes more analysis operations and presumably creates more storage pages in the early beginning. This is also indicated by the profiling results in Figure 5.7 on page 47, and was observed to varying degrees in all runs.

Disabling segmentation significantly reduces the average insertion rate to 77 ts/sec, resulting in a total insertion time of around 25 hours for the 7-million dataset. As the envelopes are stored in full resolution now, the effort for retrieving and updating rows in the database and the cost of calculations on the envelopes is much higher.

The query runtime was not monitored in most cases for practical reasons, as this requires executing only one query at the same time. Usually, however, tests were executed in parallel making it unfeasible to obtain reliable time measurements. Appendix A.3.1 on page 81, on the other hand, lists some exemplary $k$-NN queries on the 7-million dataset—together with the required execution time for exclusive runs.

## 6.6 Index Size

For the evaluation runs, the TSEIT index is approximately 8 MB in size per one million time series (of length 201 or 209). As listed in Figure 6.6, the 7-million dataset requires

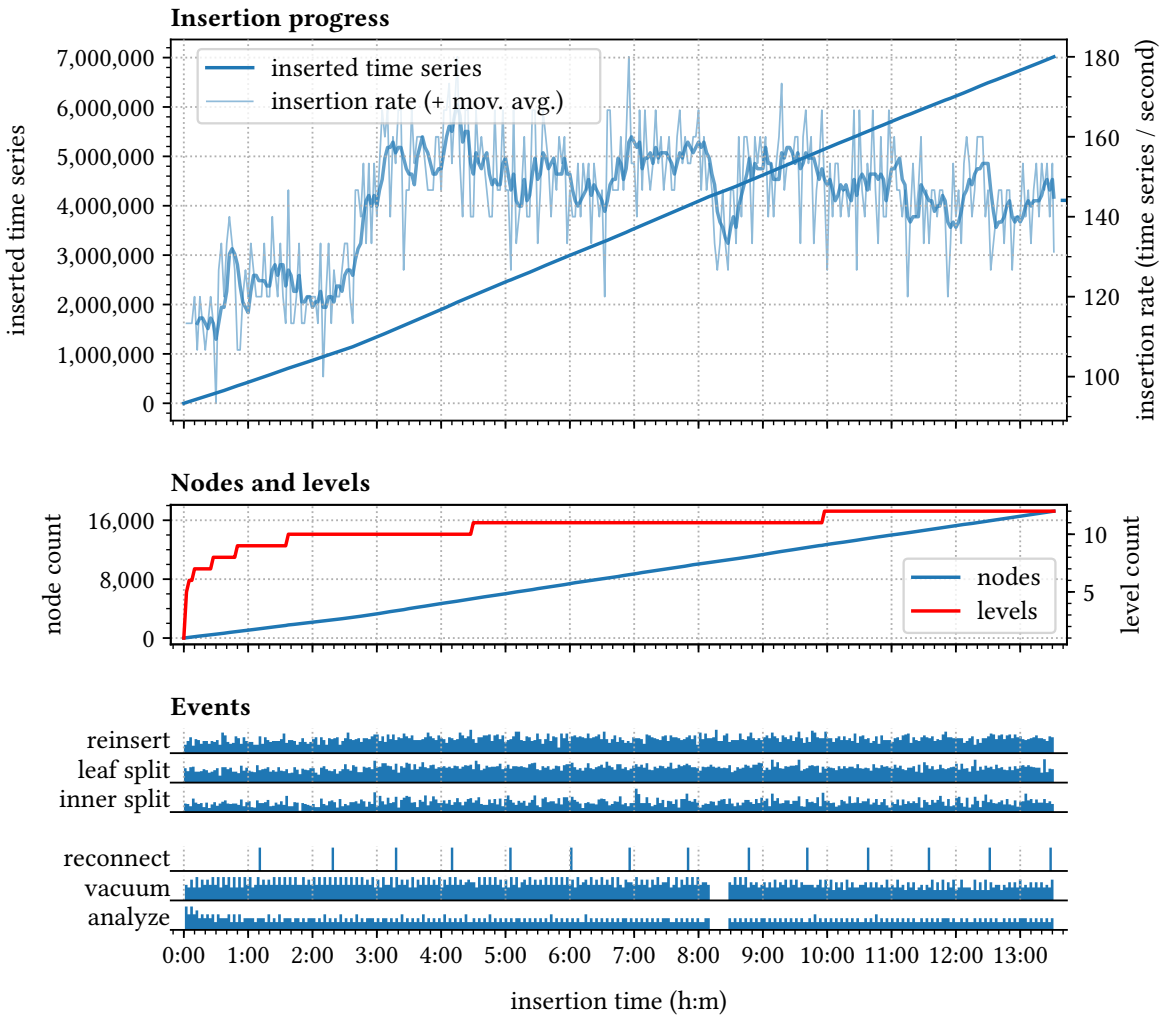



**Figure 6.7** Indexing the 7-million dataset using TSEIT with the default configuration. The upper part shows the number of inserted time series, together with the insertion rate and a moving average. The middle part plots the logarithmically increasing number of tree levels together with the total number of tree nodes.

In the bottom part, several histograms describe index and database related events. The height of a bar indicates the number of events within a time interval of 150 seconds. Note that the vertical axes of the histograms are scaled differently.

The reconnects are explicitly triggered by the insertion module of the *TSEIT Manager*, while the vacuum and analyze operations for garbage collection and table analysis are automatically executed by PostgreSQL.

56 MB of disk space, while the index of the 103-million dataset (cf. Section 6.7) is 821 MB in size. Consequently, the index table can entirely be loaded into the shared buffer cache of PostgreSQL (see `get_buffer_usage()`). As the buffer is located in memory, slow disk access is minimized.

Moreover, increasing the minimum segment length $T_{\text{min}}$ further reduces the storage size. This enables the user to effectively control the index size, which may be useful for indexing even larger datasets.

For retrieving the actual index size by the custom `get_relation_sizes()` function, a `VACUUM FULL` was executed at first to rewrite the tables and reclaim unused disk space. Otherwise, the total size would include dead tuples, too.

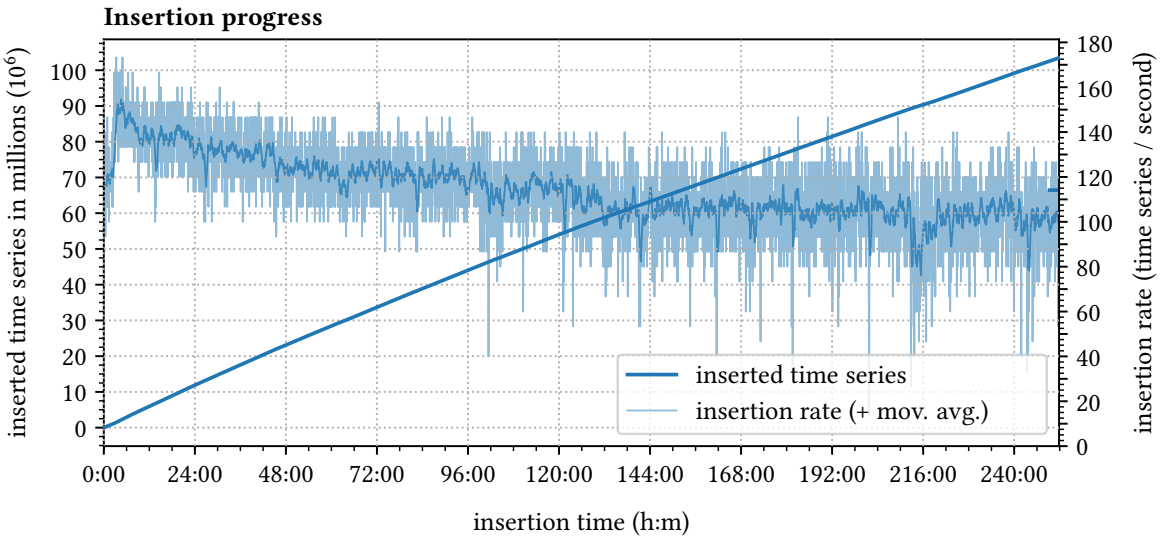
## 6.7 Inserting and Querying 103 Million Time Series

As visualized in Figure 6.8a, TSEIT indexes the time series of the 103-million dataset in 251.8 hours (10.5 days) with an average insertion rate of 113.3 ts/sec. The fact that the insertion rate decreases (almost) only linearly proves that TSEIT is efficient and able to index large datasets. The final tree has 257,437 nodes at 16 tree levels, including 146,929 leaf nodes that hold on average 704 time series (25-th, 50-th, and 75-th percentile: 574, 695, and 835 time series per leaf node, respectively).

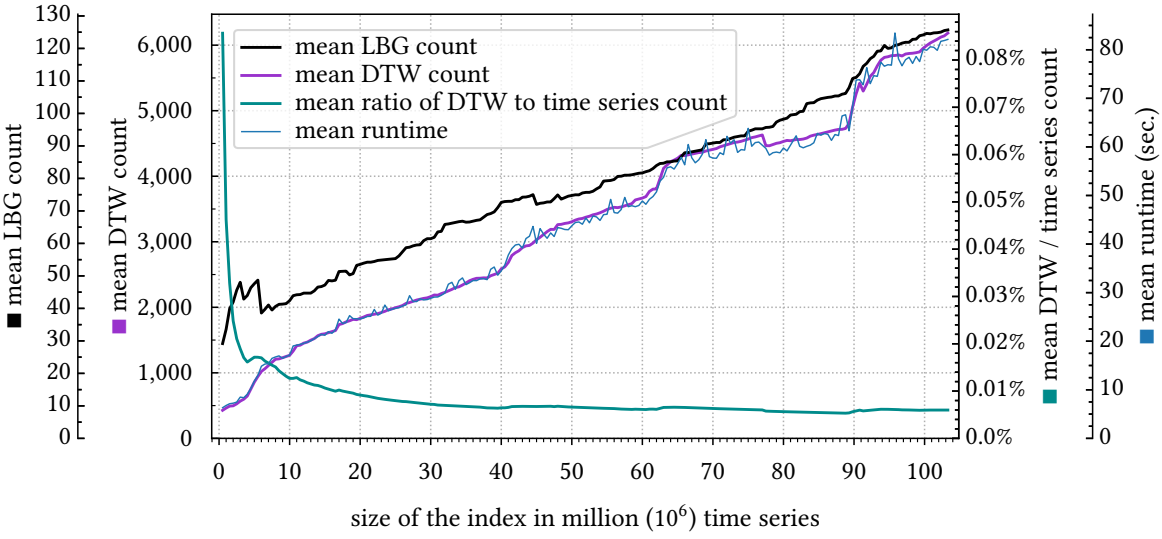
To evaluate the efficiency of the $k$-NN queries with growing dataset size, 100 1-NN queries were executed after every 500,000-th insertion for the 103-million dataset. The query time series were chosen by sorting the dataset by ascending sequence sum and taking 100 times every 100-th time series. They have been indexed first.

The evaluation results in Figure 6.8b show that both the number of required LBG and DTW calculations increases approximately linear with the size of the dataset, whereby the LBG count grows slower than the DTW count. The query runtime is mainly affected by the number of DTW calculations, as indicated by the apparent correlation of both lines in the plot. Substantial increases in the number of the DTW calculations are probably explained by the value ranges of the inserted time series, which are not entirely evenly distributed. When the tree is reorganized by the reinsertion of time series, the number of LBG and DTW calculations can drop when, as a result, fewer subtrees need to be visited. This happens several times as the plot shows. The in the end rather long average runtime for the $k$-NN queries is put into perspective, considering the ratio of the required DTW calculations to the actual size of the dataset: it decreases logarithmically and settles at around 0.006%. In other words, TSEIT calculates the DTW distance to merely 0.006% of all time series in this dataset on average. For other queries, especially with a smaller search interval, this ratio might be even smaller like presented in Appendix A.3.2 on page 86.

Note that in this test run the TSEIT index was built with the default configuration, which, however, is not necessarily optimal for such a vast dataset. Especially a larger number of child nodes might be useful to limit the depth of the tree and to increase the resolution of the segmented envelopes in the upper tree levels.




**(a)** The insertion process with a total runtime of 251.8 hours (10.5 days) and an average insertion rate of 113.3 ts/sec.




**(b)** At intervals of 500,000 inserts, 1-NN queries were executed for 100 time series. The strong increases of the DTW calculations are caused by the insertion of time series with large values, the decreases by tree restructurings by splits and reinsertions.

**Figure 6.8** Indexing and querying the 103-million dataset in two separate runs using TSEIT with the default configuration.

## 6.8 Comparison with TWIST

Unlike TSEIT, TWIST is a flat data structure, which strongly impacts the scalability and query performance.

As shown in Figure 6.4 on page 57, querying with TWIST is more costly than using a TSEIT index. While TSEIT requires between 25 and 80 LBG calculations for querying the time series of Sample 1, TWIST calculates the distance up to a hundred times more often. Not using a tree structure, all envelopes need to be accessed, what consequently sets a lower limit for the number of LBG calculations. TSEIT, in contrast, omits entire subtrees and thus does not need to visit all nodes. Note that TWIST uses the LBG function as introduced in Section 3.3.3.2 on page 12, while TSEIT applies the modified LBG distance as in Section 4.4.2 on page 32 in order to handle subsequences and segments of different lengths. Besides the LBG count, also the number of DTW calculations is higher for TWIST in the evaluation runs. This is even the case, if a larger leaf size of 10,000 is used (resulting in 1,642 and 228,381 DTW calculations on average for Sample 1 and Sample 2, respectively).

TWIST can be considered as a one-level tree where all nodes are leaf nodes (the root node is not explicitly modeled). From this perspective, the evaluation shows that a TWIST index holds more leaf nodes than a corresponding TSEIT index. Consequently, the nodes contain fewer time series on average and thus, the leave envelopes tend to be smaller while overlapping more in total. On the other hand, regarding the entire structures, TSEIT holds more nodes, as it is a multi-level tree with additional inner nodes. Therefore, given that leaf node envelopes are stored with full resolution (either due to $T_{\text{min}} = 1$ or without segmentation at all), a corresponding TWIST index is smaller in terms of storage space. However, since TSEIT allows increasing the minimum segment length $T_{\text{min}}$, the required storage space can be reduced as desired. This is not possible with TWIST that always stores envelopes in original length without segmentation. Only when a query is evaluated, TWIST computes the segments for all envelopes. This not only eliminates the possibility to save storage space but especially affects the query runtime.

The most significant drawback of TWIST is its poor scalability of the index creation. As visualized in Figure 6.9, the insertion rate decreases logarithmically since the algorithm needs to iterate through all envelopes in order to determine an appropriate one. Already after a few hundred-thousand inserted time series, the rate drops to less than 20 ts/sec and soon to impracticable 2 ts/sec and even less. This makes TWIST unfit for processing big data. In contrast, TSEIT traverses a tree examining only a small fraction of nodes to insert a new time series. Figure 6.10 shows that during the time in which TSEIT indexes the entire 7-million dataset, TWIST inserts only 11% of the time series (750,000). After 190 hours, TWIST has indexed only 2.8 million time series, whereas in the same time TSEIT can index around 80 million time series at still around 100 ts/sec (103-million dataset, cf. Figure 6.8a). An additional comparison of TSEIT's and TWIST's insertion rate and index size is given in Figure 6.6 on page 59.

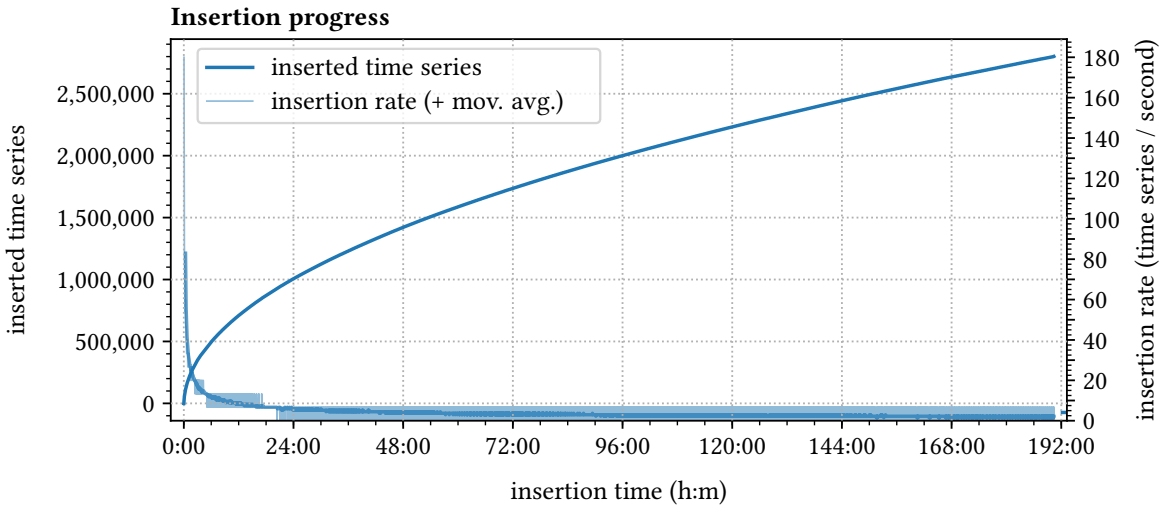



**Figure 6.9** Indexing the first 2.8 million time series of the 7-million dataset using TWIST. The insertion rate is logarithmically decreasing from 180 ts/sec in the very beginning to only 2.0 ts/sec after 190 hours.

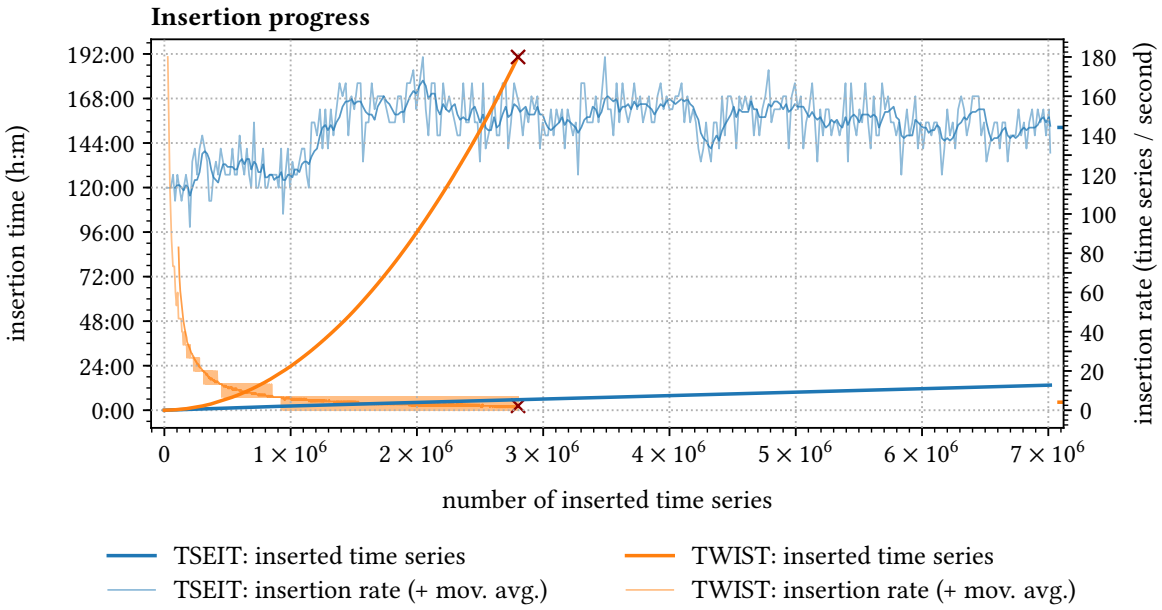



**Figure 6.10** Direct comparison of TSEIT and TWIST concerning the insertion of the time series from the 7-million dataset. The runs are the same as in Figure 6.7 on page 60 and Figure 6.9. TWIST was manually canceled after having inserted 2,8 million time series in 190 hours.

# 7 Conclusion and Future Work

This last chapter gives a brief conclusion on the content and findings of this work and finally proposes future work.

## 7.1 Conclusion

The aim of this work was the development of an efficient technique for finding the $k$ nearest neighbors of a given time series within a specified time interval. In recent decades numerous approaches for whole matching or subsequence matching have been proposed. Former ones compare entire time series of equal length, while the latter retrieve similar subsequences contained in other time series, no matter at which position. Focusing on a search within arbitrary user-defined time intervals, in contrast, is a topic not exhausted in research.

In this work, the height-balanced index structure TSEIT was designed, implemented and evaluated. It enables exact $k$-NN searches in arbitrary time intervals without false dismissals. The basic idea is to group similar time series and to store a representative of each group—a so-called envelope—in an R-tree-like data structure. A leaf node envelope tightly wraps the time series held by the leaf, while the envelope of an inner node spans the envelopes of its child nodes. The index creation algorithm aims to create leaf node envelopes that are well-filled, have a small area and overlap only slightly. As it can handle typical sequence transformations like shifting and scaling, the DTW distance is used as the similarity measure for time series.

To find the nearest neighbors of a given query time series, the TSEIT tree is traversed, descending only into those subtrees to which the LBG lower-bound distance is not larger than the minimum DTW distance found so far. The lower-bound distance to an envelope representing a subtree or leaf node is guaranteed to be smaller than (or equal to) the distance to any of the time series it wraps. Conversely, this means that a $k$-NN time series cannot be within an envelope for which the lower-bound distance is already larger than the exact best-so-far distance. In this case, the subtree or leaf node represented by the envelope can be omitted. Thus, the exact DTW distance has to be calculated for only a fraction of all time series in the TSEIT tree.

To accelerate tree traversal and to reduce storage requirements, the envelopes are usually stored with reduced dimensionality. They are approximated by so-called segments that are defined by the maximum and minimum envelope values within consecutive intervals. While the segment length is minimal at the leaf level, it doubles with each level up to the

root node. Hence, the upper tree levels hold envelopes of lower dimensionality, whereas the tightness of the lower bound increases when descending to deeper tree levels. The quality of the index tree is improved by occasionally deleting time series and reinserting them into the tree. This results in overall tighter envelopes, as the reinserted time series often land up in a more suitable part of the tree.

A novel clustering algorithm called *k*-envelopes for splitting nodes during index creation was developed in this work. It is based on ideas of Lloyd's *k*-means algorithm but tailored to the needs of TSEIT. For example, it allows defining a minimum cluster size and supports various cost functions for assigning sequences to clusters. The alternatively supported clustering algorithm Overlap Split is an adaption of the R*-tree's splitting technique. Although this algorithm works better than the established *k*-means algorithm, it did slightly worse in the evaluation than *k*-envelopes, which outperformed both methods.

The TSEIT index is stored together with the time series in a PostgreSQL database and maintained by a custom Python module executed by the database system. The *k*-NN queries are evaluated by a PostgreSQL function that is also written in Python. In addition, numerous helper functions support the analysis and evaluation of the index.

The evaluation supports the claim that TSEIT is able to efficiently search the *k*-NN of time series in arbitrary time intervals. In any case, only a fraction—often 0.1% or much less—of all time series is accessed, even though an exact search without false dismissals is guaranteed. TSEIT scales well and is able to index and query millions of time series, being many times faster than the competing approach TWIST. Since the insertion rate decreases only linearly by a small factor, TSEIT is able to index more than 100 million time series effortlessly.

## 7.2 Future Work

Although this work achieves its goals, there is potential for conceptual and implementation-related extensions and modifications. The following gives hints on possible future work.

**Deletion of time series** The intention of this work was the one-time indexing of static datasets. While later insertions are covered, the deletion of time series from the index has not been part of this work. The implementation even prevents the deletion of rows from the time series table to maintain the integrity of the index. After deleting a time series, the following invariants might need to be restored:

- An envelope (tightly) wraps the time series it holds.
- A leaf node contains at least $l_{\min}$ time series.
- An inner node has at least $c_{\min}$ child nodes.

The first point is not that critical, as the envelope can easily be recalculated. Alternatively, the envelope could even stay unmodified, as it still would not violate the lower-bound property of LBG. If a leaf node contains fewer time series than specified by $l_{\min}$, it needs to be deleted. The now orphaned time series can either be reinserted from the root node

or, if possible, distributed among the sibling nodes. The deletion of a leaf node may cause its parent node to violate the $c_{\min}$ constraint, provoking a deletion of this node, too. In this case, each orphaned child node can be reinserted as a whole. Alternatively, all time series below the deleted inner node can be reinserted individually. As the deletion of a time series can propagate up in the tree, it is worth to examine efficient approaches to deal with the costly reorganization of the index tree.

**Time series of different lengths** Currently, the time series in the TSEIT index are required to be of the same length. However, it is conceivable to add support for indexing time series of different lengths or with gaps. For example, this would allow inserting time series with differing start times. When the time series are defined for the entire query interval, there is conceptually no difference to the current solution. However, subsequences that are incomplete within the query interval need special treatment, e.g., by extrapolating the distance for the full interval. Moreover, envelopes of different lengths are worth considering, too.

**Maximum segment length** Starting with the minimum segment length $T_{\min}$ at the leaf level, the segment length doubles with each higher tree level. Once the segments are as long as the indexed time series, the segment length cannot double anymore and thus each further upper tree level holds envelopes consisting of only one segment. It might be useful to define a maximum segment length for a higher envelope resolution in upper levels.

**Cascade of lower bounds** Examining the time series of a candidate leaf node on $k$-NN query evaluation, the LB_Kim lower bound is calculated before possibly computing the exact but costly DTW distance. While LB_Kim is fast to compute, it is not as close to the exact distance as other lower-bound functions, which, however, are computational more intensive. It can be examined whether it is worth to either replace LB_Kim or to build a cascade of multiple lower-bound functions as in [Rak+12].

**Implementing LBG in C** Due to its recursive definition, it is not possible to efficiently implement the LBG function using NumPy. However, as this method is essential for querying, it might be beneficial to re-implement it in plain C to overcome the overhead of interpreted Python code. While there are several ways to wrap C code in Python, the simplest way is probably the use of NumPy's C-API. Since the query evaluation was not profiled, it is not clear how significant the performance gain could be.

**Parallelizing DTW computations** On $k$-NN query processing, both the LB_Kim and especially the DTW distances are always computed sequentially for all time series of a leaf node. The parallelization of those computations should considerably accelerate the query execution. For this, it might be useful to replace the Python module `heapq` with `queue`. The latter module implements the required locking mechanism for safely sharing data, like the current best-so-far distance, between multiple threads.

# Bibliography


[Haa10] Alfred Haar. "Zur Theorie der orthogonalen Funktionensysteme". In: *Mathematische Annalen* 69.3 (Sept. 1, 1910), pp. 331–371. DOI: 10.1007/BF01456326.

[FH51] Evelyn Fix and J. L. Hodges Jr. *Discriminatory Analysis - Nonparametric Discrimination: Consistency Properties*. Tech. rep. University of California, Berkeley, Feb. 1951. URL: http://www.dtic.mil/docs/citations/ADA800276.

[CH67] T. Cover and P. Hart. "Nearest Neighbor Pattern Classification". In: *IEEE Transactions on Information Theory* 13.1 (Jan. 1967), pp. 21–27. DOI: 10.1109/TIT.1967.1053964.

[Mac67] J. MacQueen. "Some Methods for Classification and Analysis of Multivariate Observations". In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. The Regents of the University of California, 1967. URL: https://projecteuclid.org/euclid.bsmsp/1200512992.

[BM70] R. Bayer and E. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control*. SIGFIDET '70. ACM, 1970, pp. 107–141. DOI: 10.1145/1734663.1734671.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. DOI: 10.1145/362384.362685.

[Bay72] Rudolf Bayer. "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms". In: *Acta Informatica* 1.4 (Dec. 1, 1972), pp. 290–306. DOI: 10.1007/BF00289509.

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[Ben75] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517. DOI: 10.1145/361002.361007.

[Ita75] F. Itakura. "Minimum Prediction Residual Principle Applied to Speech Recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23.1 (Feb. 1975), pp. 67–72. DOI: 10.1109/TASSP.1975.1162641.

[OS75] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing*. Prentice-Hall, 1975. ISBN: 978-0-13-214635-7.

[McC76] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[SC78] H. Sakoe and S. Chiba. "Dynamic Programming Algorithm Optimization for Spoken Word Recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (Feb. 1978), pp. 43–49. DOI: 10.1109/TASSP.1978.1163055.

[Com79] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Computing Surveys* 11.2 (June 1979), pp. 121–137. DOI: 10.1145/356770.356776.

[LY81] Philip L. Lehman and s. Bing Yao. "Efficient Locking for Concurrent Operations on B-Trees". In: *ACM Transactions on Database Systems* 6.4 (Dec. 1981), pp. 650–670. DOI: 10.1145/319628.319663.

[Llo82] S. Lloyd. "Least Squares Quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.

[Gut84] Antonin Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. ACM, 1984, pp. 47–57. DOI: 10.1145/602259.602266.

[Bec+90] Norbert Beckmann et al. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles". In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD '90. ACM, 1990, pp. 322–331. DOI: 10.1145/93597.98741.

[Dun90] R. Duncan. "A Survey of Parallel Computer Architectures". In: *Computer* 23.2 (Feb. 1990), pp. 5–16. DOI: 10.1109/2.44900.

[Spr91] Robert F. Sproull. "Refinements to Nearest-Neighbor Searching in k-Dimensional Trees". In: *Algorithmica* 6.1 (June 1, 1991), pp. 579–589. DOI: 10.1007/BF01759061.

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. "Efficient Similarity Search in Sequence Databases". In: *Foundations of Data Organization and Algorithms*. International Conference on Foundations of Data Organization and Algorithms. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Oct. 13, 1993, pp. 69–84. DOI: 10.1007/3-540-57301-1_5.

[BC94] Donald J. Berndt and James Clifford. "Using Dynamic Time Warping to Find Patterns in Time Series". In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*. AAAIWS'94. AAAI Press, 1994, pp. 359–370. ISBN: 0-929280-73-3.

[FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. "Fast Subsequence Matching in Time-Series Databases". In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. SIGMOD '94. ACM, 1994, pp. 419–429. DOI: 10.1145/191839.191925.

[Gra95] A. Graps. "An Introduction to Wavelets". In: *IEEE Computational Science and Engineering* 2.2 (1995), pp. 50–61. DOI: 10.1109/99.388960.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. "Generalized Search Trees for Database Systems". In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. Morgan Kaufmann Publishers Inc., 1995, pp. 562–573. ISBN: 978-1-55860-379-0.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. "Nearest Neighbor Queries". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. ACM, 1995, pp. 71–79. DOI: 10.1145/223784.223794.

[Fal96] Christos Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, 1996. ISBN: 0-7923-9777-0.

[SZ96] H. Shatkay and S.B. Zdonik. "Approximate Queries and Representations for Large Data Sequences". In: *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE Comput. Soc. Press, 1996, pp. 536–545. DOI: 10.1109/ICDE.1996.492204.

[For+97] John T. Foreman et al. *C4 Software Technology Reference Guide - A Prototype*. Handbook CMU/SEI-97-HB-001. Carnegie Mellon University, Software Engineering Institute, Jan. 1997, p. 438. URL: http://www.dtic.mil/docs/citations/ADA320732.

[Keo97] E. Keogh. "Fast Similarity Search in the Presence of Longitudinal Scaling in Time Series Databases". In: *Proceedings Ninth IEEE International Conference on Tools with Artificial Intelligence*. IEEE Comput. Soc, 1997, pp. 578–584. DOI: 10.1109/TAI.1997.632306.

[KJF97] Flip Korn, H. V. Jagadish, and Christos Faloutsos. "Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences". In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD '97. ACM, 1997, pp. 289–300. DOI: 10.1145/253260.253332.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen 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. Morgan Kaufmann Publishers Inc., 1998, pp. 194–205. ISBN: 978-1-55860-566-4.

[YJF98] Byoung-Kee Yi, H V Jagadish, and C Faloutsos. "Efficient Retrieval of Similar Time Sequences under Time Warping". In: *Proceedings 14th International Conference on Data Engineering*. Feb. 1998, pp. 201–208. DOI: 10.1109/ICDE.1998.655778.

[HS99] Gísli R. Hjaltason and Hanan Samet. "Distance Browsing in Spatial Databases". In: *ACM Transactions on Database Systems* 24.2 (June 1999), pp. 265–318. DOI: 10.1145/320248.320255.

[KP99] E.J. Keogh and M.J. Pazzani. "An Indexing Scheme for Fast Similarity Search in Large Time Series Databases". In: *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*. IEEE Comput. Soc, 1999, pp. 56–67. DOI: 10.1109/SSDM.1999.787621.

[KA99] Kin-Pong Chan and Ada Wai-Chee Fu. "Efficient Time Series Matching by Wavelets". In: *Proceedings 15th International Conference on Data Engineering*. IEEE, Mar. 1999, pp. 126–133. DOI: 10.1109/ICDE.1999.754915.

[XE00] Mei Xu and C. I. Ezeife. "Maintaining Horizontally Partitioned Warehouse Views". In: *Data Warehousing and Knowledge Discovery*. International Conference on Data Warehousing and Knowledge Discovery. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Sept. 4, 2000, pp. 126–133. DOI: 10.1007/3-540-44466-1_13.

[YF00] Byoung-Kee Yi and Christos Faloutsos. "Fast Time Sequence Indexing for Arbitrary Lp Norms". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. Morgan Kaufmann Publishers Inc., Sept. 1, 2000, pp. 385–394. ISBN: 978-1-55860-715-6.

[KS01] T. Kahveci and A. Singh. "Variable Length Queries for Time Series Data". In: *Proceedings 17th International Conference on Data Engineering*. IEEE Comput. Soc, 2001, pp. 273–282. DOI: 10.1109/ICDE.2001.914838.

[Keo+01] Eamonn Keogh et al. "Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases". In: *Knowledge and Information Systems* 3.3 (Aug. 1, 2001), pp. 263–286. DOI: 10.1007/PL00011669.

[KPC01] Sang-Wook Kim, Sanghyun Park, and Wesley W. Chu. "An Index-Based Approach for Similarity Search Supporting Time Warping in Large Sequence Databases". In: *Proceedings 17th International Conference on Data Engineering*. 2001, pp. 607–614. DOI: 10.1109/ICDE.2001.914875.

[Cha+02] Kaushik Chakrabarti et al. "Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases". In: *ACM Transactions on Database Systems* 27.2 (June 2002), pp. 188–228. DOI: 10.1145/568518.568520.

[AiJ+02] Ai-Jun Li et al. "An Approach for Fast Subsequence Matching through KMP Algorithm in Time Series Databases". In: *Proceedings. International Conference on Machine Learning and Cybernetics*. Vol. 3. IEEE, 2002, pp. 1292–1295. DOI: 10.1109/ICMLC.2002.1167412.

[RK04] Chotirat Ann Ratanamahatana and Eamonn Keogh. "Making Time-Series Classification More Accurate Using Learned Constraints". In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. Ed. by Michael W. Berry et al. Society for Industrial and Applied Mathematics, Apr. 22, 2004, pp. 11–22. DOI: 10.1137/1.9781611972740.2.

[KR05] Eamonn Keogh and Chotirat Ann Ratanamahatana. "Exact Indexing of Dynamic Time Warping". In: *Knowledge and Information Systems* 7.3 (2005), pp. 358–386. DOI: 10.1007/s10115-004-0154-9.

[SYF05] Yasushi Sakurai, Masatoshi Yoshikawa, and Christos Faloutsos. "FTW: Fast Similarity Search Under the Time Warping Distance". In: *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '05. ACM, 2005, pp. 326–337. DOI: 10.1145/1065167.1065210.

[Man+06] Yannis Manolopoulos et al. *R-Trees: Theory and Applications*. Red. by Lakhmi Jain and Xindong Wu. Advanced Information and Knowledge Processing. Springer London, 2006. DOI: 10.1007/978-1-84628-293-5.

[Aßf+07] Johannes Aßfalg et al. "Interval-Focused Similarity Search in Time Series Databases". In: *Advances in Databases: Concepts, Systems and Applications*. International Conference on Database Systems for Advanced Applications. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Apr. 9, 2007, pp. 586–597. DOI: 10.1007/978-3-540-71703-4_50.

[LPK07] Seung-Hwan Lim, Heejin Park, and Sang-Wook Kim. "Using Multiple Indexes for Efficient Subsequence Matching in Time-Series Databases". In: *Information Sciences* 177.24 (Dec. 15, 2007), pp. 5691–5706. DOI: 10.1016/j.ins.2007.07.004.

[Lin+07] Jessica Lin et al. "Experiencing SAX: A Novel Symbolic Representation of Time Series". In: *Data Mining and Knowledge Discovery* 15.2 (2007), pp. 107–144. DOI: 10.1007/s10618-007-0064-z.

[SC07] Stan Salvador and Philip Chan. "Toward Accurate Dynamic Time Warping in Linear Time and Space". In: *Intelligent Data Analysis* 11.5 (Jan. 1, 2007), pp. 561–580. ISSN: 1088-467X.

[Du+08] Yi Du et al. "Effective Subsequence Matching in Compressed Time Series". In: *2008 Third International Conference on Pervasive Computing and Applications*. IEEE, Oct. 2008, pp. 922–926. DOI: 10.1109/ICPCA.2008.4783742.

[SK08] Jin Shieh and Eamonn 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. ACM, 2008, pp. 623–631. DOI: 10.1145/1401890.1401966.

[Cam+10] Alessandro Camerra et al. "iSAX 2.0: Indexing and Mining One Billion Time Series". In: *2010 IEEE International Conference on Data Mining*. IEEE, Dec. 2010, pp. 58–67. DOI: 10.1109/ICDM.2010.124.

[KS10] Maciej Krawczak and Grazyna Szkatula. "Time Series Envelopes for Classification". In: *2010 5th IEEE International Conference Intelligent Systems*. IEEE, July 2010, pp. 156–161. DOI: 10.1109/IS.2010.5548371.

[NRR10] Vit Niennattrakul, Pongsakorn Ruengronghirunya, and Chotirat Ann Ratanamahatana. "Exact Indexing for Massive Time Series Databases under Time Warping Distance". In: *Data Mining and Knowledge Discovery* 21.3 (Nov. 2010), pp. 509–541. DOI: 10.1007/s10618-010-0165-y.

[Cas+12] Carmelo Cassisi et al. "Similarity Measures and Dimensionality Reduction Techniques for Time Series Data Mining". In: *Advances in Data Mining Knowledge Discovery and Applications*. Ed. by Adem Karahoca. InTech, 2012. DOI: 10.5772/49941.

[Rak+12] Thanawin Rakthanmanon et al. "Searching and Mining Trillions of Time Series Subsequences Under Dynamic Time Warping". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. ACM, 2012, pp. 262–270. DOI: 10.1145/2339530.2339576.

[SH12] Patrick Schäfer and Mikael 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. ACM, 2012, pp. 516–527. DOI: 10.1145/2247596.2247656.

[Wan+13] Yang Wang et al. "A Data-Adaptive and Dynamic Segmentation Index for Whole Matching on Time Series". In: *Proceedings of the VLDB Endowment* 6.10 (Aug. 2013), pp. 793–804. DOI: 10.14778/2536206.2536208.

[XC13] Xiao-Ying Liu and Chuan-Lun Ren. "Fast Subsequence Matching under Time Warping in Time-Series Databases". In: *2013 International Conference on Machine Learning and Cybernetics*. IEEE, July 2013, pp. 1584–1590. DOI: 10.1109/ICMLC.2013.6890855.

[Cam+14] Alessandro Camerra et al. "Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+". In: *Knowledge and Information Systems* 39.1 (Apr. 1, 2014), pp. 123–151. DOI: 10.1007/s10115-012-0606-6.

[LY14] Hailin Li and Libin Yang. "Extensions and Relationships of Some Existing Lower-Bound Functions for Dynamic Time Warping". In: *Journal of Intelligent Information Systems* 43.1 (2014), pp. 59–79. DOI: 10.1007/s10844-014-0306-7.

[NB14] Happy Nath and Ujwala Baruah. "Evaluation of Lower Bounding Methods of Dynamic Time Warping (DTW)". In: *International Journal of Computer Applications* 94.20 (May 30, 2014), pp. 12–17. DOI: 10.5120/16550-6168.

[SA14] Joan Serrà and Josep Ll. Arcos. "An Empirical Evaluation of Similarity Measures for Time Series Classification". In: *Knowledge-Based Systems* 67 (Supplement C Sept. 1, 2014), pp. 305–314. DOI: 10.1016/j.knosys.2014.04.035.

[ZIP14] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. "Indexing for Interactive Exploration of Big Data Series". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. ACM, 2014, pp. 1555–1566. DOI: 10.1145/2588555.2610498.

[Gil+15] Myeong-Seon Gil et al. "Fast Index Construction for Distortion-Free Subsequence Matching in Time-Series Databases". In: *2015 International Conference on Big Data and Smart Computing*. IEEE, Feb. 2015, pp. 130–135. DOI: 10.1109/35021BIGCOMP.2015.7072822.

[PG15] John Paparrizos and Luis Gravano. "K-Shape: Efficient and Accurate Clustering of Time Series". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015, pp. 1855–1870. DOI: 10.1145/2723372.2737793.

[Sch15] Patrick Schäfer. "Scalable Time Series Similarity Search for Data Analytics". PhD thesis. Humboldt-Universität zu Berlin, 2015. DOI: 10.18452/17338.

[YC15] Sho Yoshida and Basabi Chakraborty. "A Comparative Study of Similarity Measures for Time Series Classification". In: *New Frontiers in Artificial Intelligence*. JSAI International Symposium on Artificial Intelligence. Lecture Notes in Computer Science. Springer International Publishing, Nov. 16, 2015, pp. 397–408. DOI: 10.1007/978-3-319-50953-2_27.

[ZIP16] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. "ADS: The Adaptive Data Series Index". In: *The VLDB Journal* 25.6 (Dec. 1, 2016), pp. 843–866. DOI: 10.1007/s00778-016-0442-5.

# A Appendix

## A.1 Configuration of TSEIT

Below, the default configuration file for TSEIT is given. The user can override all parameters in a separate user configuration file.

```
; DATABASE ------------------------------------------------------------------
[super_database]
; <String> (default: postgres)
name = postgres

; <String> (default: postgres)
user = postgres

; <String> (default: localhost)
host = localhost

[database]
; <String> (default: tseit)
name = tseit

; <String> (default: postgres)
user = postgres

; <String> (default: localhost)
host = localhost

; INDEX ---------------------------------------------------------------------
[index]
; <Integer> (default: 200)
sequence_length = 200

; TREE ----------------------------------------------------------------------
; <Integer> (default: 2)
min_number_of_children = 2

; <Integer> (default: 3)
max_number_of_children = 3
```

```
; <Integer> (default: 1)
min_leaf_size = 1

; <Integer> (default: 1000)
max_leaf_size = 1000

; SEGMENTATION ----------------------------------------------------------
; true (default) | false
use_segmentation = true

; <Integer> (default: 1)
min_segment_length = 1

; SPLITTING -------------------------------------------------------------
; kenvelopes (default) | overlap_split | kmeans
split_algo_leaf_node = kenvelopes

; kenvelopes (default) | overlap_split
split_algo_inner_node = kenvelopes

; overlap_than_area | overlap_than_insertion_cost
; | area (default) | insertion_cost
kenvelopes_cost_fct = area

; true | false (default)
kenvelopes_sort_alternating = false

; MISCELLANEOUS ---------------------------------------------------------
; overlap_than_area (default) | overlap_than_insertion_cost
; | area | insertion_cost
choose_subtree_cost_fct = overlap_than_area

; true (default) | false
use_reinsertion = true

; TWIST -----------------------------------------------------------------
; true | false (default)
twist = false

; none | <Integer> (default: 7592)
twist_rand_seed = 7592

; <Integer> (default: 1000)
twist_max_node_size = 1000
```

```
; PROFILING ------------------------------------------------------------------
; false (default) | cprofile
profile = false

; <String> (default: profile.stats)
profile_path = profile.stats
```

## A.2 Configuration of PostgreSQL

The listing below shows the modifications on the default PostgreSQL configuration file. See `https://www.postgresql.org/docs/9.6/static/runtime-config.html` for a detailed description of the parameters.

```
#------------------------------------------------------------------------------
# CONNECTIONS AND AUTHENTICATION
#------------------------------------------------------------------------------

# - Connection Settings -
max_connections = 16  # default: 100

#------------------------------------------------------------------------------
# RESOURCE USAGE (except WAL)
#------------------------------------------------------------------------------

# - Memory -
shared_buffers = 15GB  # default: 128MB
work_mem = 2GB  # default: 4MB
maintenance_work_mem = 64GB  # default: 64MB

# - Asynchronous Behavior -
effective_io_concurrency = 10  # default: 1
max_worker_processes = 16  # default: 8
max_parallel_workers_per_gather = 16  # default: 2

#------------------------------------------------------------------------------
# WRITE AHEAD LOG
#------------------------------------------------------------------------------

# - Settings -
fsync = off  # default: on
synchronous_commit = off  # default: on
full_page_writes = off  # default: on
```

```
# - Checkpoints -
max_wal_size = 8GB  # default: 1GB
min_wal_size = 4GB  # default: 80MB
checkpoint_completion_target = 0.9  # default: 0.5

#------------------------------------------------------------------------------
# QUERY TUNING
#------------------------------------------------------------------------------

# - Planner Cost Constants -
parallel_setup_cost = 500.0  # default: 1000.0
effective_cache_size = 64GB  # default: 4GB

#------------------------------------------------------------------------------
# ERROR REPORTING AND LOGGING
#------------------------------------------------------------------------------

# - Where to Log -
log_destination = 'csvlog'  # default: stderr
logging_collector = on  # default: off
log_directory = '/var/log/postgresql'  # default: 'pg_log'
log_file_mode = 0644  # default: 0600

# - When to Log -
client_min_messages = debug  # default: notice
log_min_error_statement = fatal  # default: error

# - What to Log -
log_error_verbosity = terse  # default: default

#------------------------------------------------------------------------------
# AUTOVACUUM PARAMETERS
#------------------------------------------------------------------------------

log_autovacuum_min_duration = 0  # default:  -1
autovacuum_vacuum_cost_limit = 1000  # default: -1
```

## A.3 Exemplary *k*-NN Queries

In the following, the raw output of exemplary *k*-NN queries on the 7-million and the 103-million dataset is given. Leading `NOTICE:` strings, the content of the `stats` column (holding statistics as JSON-formatted string for client-side processing) and a few empty lines have been removed from the listings for space reasons.

The queries were executed after having loaded the database tables into PostgreSQL's buffer cache with the custom function `prewarm()`. The execution time was measured using the `timing` option of PostgreSQL's command-line client *psql*.

### A.3.1 Queries on the 7-Million Dataset

```
# SELECT * FROM knn('and', start_index => -1, end_index => -1,
#                   k => 4, radius => -1, verbose => True);
get time series...
[knn bsf: inf        ] level 10 - node 5318           LBG dist: 0.10528418421862709
[knn bsf: inf        ] level 10 - node 12631          LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 5319           LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 8744           LBG dist: 0.1052840820344287
[knn bsf: inf        ] level  8 - node 1777           LBG dist: 0.10528379661854458
[knn bsf: inf        ] level  8 - node 5317           LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 12597          LBG dist: 0.10528373423066201
[knn bsf: inf        ] level  7 - node 1743           LBG dist: 0.11486075082392017
[knn bsf: inf        ] level  7 - node 5316           LBG dist: 0.0
[knn bsf: inf        ] level  7 - node 11617          LBG dist: 0.11486044749520077
[knn bsf: inf        ] level  6 - node 1742           LBG dist: 0.12207004377429354
[knn bsf: inf        ] level  6 - node 3388           LBG dist: 0.12207867085282809
[knn bsf: inf        ] level  6 - node 11219          LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 587            LBG dist: 0.12505069448716366
[knn bsf: inf        ] level  5 - node 6588           LBG dist: 0.12500526367371476
[knn bsf: inf        ] level  5 - node 13497          LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 7220           LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 13496          LBG dist: 0.12700928642094733
[knn bsf: inf        ] level  3 - node 147            LBG dist: 0.12790301702559537
[knn bsf: inf        ] level  3 - node 7219           LBG dist: 0.12847742738133625
[knn bsf: inf        ] level  3 - node 8420           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 134            LBG dist: 0.1276305340647704
[knn bsf: inf        ] level  2 - node 6622           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 14926          LBG dist: 0.12407300009380696
[knn bsf: inf        ] level  1 - node 6482           LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 6621           LBG dist: 0.09944154604173609
[knn bsf: inf        ] level  0 - node 6448           LBG dist: 0.0
[knn bsf: inf        ] level  0 - node 6481           LBG dist: 0.051786927665999495
[knn bsf: 2.25377e-02] level  0 - node 6448           min. DTW: 0.001016920096783093

LBG calculations:         28
examined inner nodes:     11
examined leaf nodes:      1
LB_Kim calculations:      5
DTW calculations:         5

 k |  ts_id  | name |      distance       | start_index | end_index | stats
---+---------+------+---------------------+-------------+-----------+-------
 1 | 6076076 | to   | 0.00101692009678309 |           0 |       208 |  ...
 2 | 2896104 | in   |  0.0155376628564426 |           0 |       208 |  ...
 3 |  459271 | a    |  0.0220354034133428 |           0 |       208 |  ...
 4 | 4412877 | of   |  0.0225376950785585 |           0 |       208 |  ...

Time: 367.575 ms
```

**Listing A.1** Nearest neighbors of *and* for the years 1800 – 2008.

```
# SELECT * FROM knn('Germany', start_index => 130, end_index => 150,
#                   k => 2, radius => -1, verbose => True);
get time series...
[knn bsf: inf        ] level 10 - node 5318           LBG dist: 2.452010665370238e-07
[knn bsf: inf        ] level 10 - node 12631          LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 5319           LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 8744           LBG dist: 2.451516379639861e-07
[knn bsf: inf        ] level  8 - node 1777           LBG dist: 2.4501360276886693e-07
[knn bsf: inf        ] level  8 - node 5317           LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 12597          LBG dist: 2.4498343537627007e-07
[knn bsf: inf        ] level  7 - node 1743           LBG dist: 2.44831742689711e-07
[knn bsf: inf        ] level  7 - node 5316           LBG dist: 0.0
[knn bsf: inf        ] level  7 - node 11617          LBG dist: 2.446023303215416e-07
[knn bsf: inf        ] level  6 - node 1742           LBG dist: 2.402493560015593e-07
[knn bsf: inf        ] level  6 - node 3388           LBG dist: 2.44474989604464e-07
[knn bsf: inf        ] level  6 - node 11219          LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 587            LBG dist: 2.442392613272447e-07
[knn bsf: inf        ] level  5 - node 6588           LBG dist: 1.9259370377786145e-07
[knn bsf: inf        ] level  5 - node 13497          LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 7220           LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 13496          LBG dist: 2.1726206695260976e-07
[knn bsf: inf        ] level  3 - node 147            LBG dist: 5.5831918343947615e-08
[knn bsf: inf        ] level  3 - node 7219           LBG dist: 1.7181443237849322e-07
[knn bsf: inf        ] level  3 - node 8420           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 134            LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 6622           LBG dist: 1.6939984863798723e-07
[knn bsf: inf        ] level  2 - node 14926          LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 129            LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 133            LBG dist: 1.1953057372942287e-07
[knn bsf: inf        ] level  0 - node 9291           LBG dist: 0.0
[knn bsf: inf        ] level  0 - node 14924          LBG dist: 0.0
[knn bsf: 1.04658e-08] level  0 - node 9291           min. DTW: 8.793723645729129e-09
[knn bsf: 5.56403e-09] level  0 - node 14924          min. DTW: 4.720827730487925e-09
[knn bsf: 5.56403e-09] level  1 - node 7720           LBG dist: 0.0
[knn bsf: 5.56403e-09] level  1 - node 14925          LBG dist: 0.0
[knn bsf: 5.56403e-09] level  0 - node 127            LBG dist: 9.584258272321103e-07
[knn bsf: 5.56403e-09] level  0 - node 6966           LBG dist: 0.0
[knn bsf: 3.43002e-09] level  0 - node 6966           min. DTW: 3.044039632242908e-09
[knn bsf: 3.43002e-09] level  0 - node 128            LBG dist: 0.0
[knn bsf: 3.43002e-09] level  0 - node 7719           LBG dist: 0.0
[knn bsf: 3.04404e-09] level  0 - node 128            min. DTW: 1.5346986393409813e-09
[knn bsf: 3.04404e-09] level  0 - node 7719           min. DTW: 4.5235610212726406e-09

LBG calculations:         34
examined inner nodes:     14
examined leaf nodes:      5
LB_Kim calculations:      2358
DTW calculations:         504

 k |  ts_id  | name  |       distance       | start_index | end_index | stats
---+---------+-------+----------------------+-------------+-----------+-------
 1 | 4756547 | peace | 1.53469863934098e-09 |         130 |       150 |  ...
 2 |  583527 | blood | 3.04403963224291e-09 |         130 |       150 |  ...

Time: 191.804 ms
```

**Listing A.2** Nearest neighbors of *Germany* for the years 1930 – 1950.

```
# SELECT * FROM knn('Germany', start_index => 155, end_index => -1,
#                   k => 3, radius => -1,  verbose => True);
get time series...
[knn bsf: inf        ] level 10 - node 5318           LBG dist: 2.1535228272938325e-07
[knn bsf: inf        ] level 10 - node 12631          LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 5319           LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 8744           LBG dist: 2.152780041990043e-07
[knn bsf: inf        ] level  8 - node 1777           LBG dist: 2.1507060073667799e-07
[knn bsf: inf        ] level  8 - node 5317           LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 12597          LBG dist: 2.1502527854316913e-07
[knn bsf: inf        ] level  7 - node 1743           LBG dist: 2.1479741207794437e-07
[knn bsf: inf        ] level  7 - node 5316           LBG dist: 0.0
[knn bsf: inf        ] level  7 - node 11617          LBG dist: 2.1445289381883806e-07
[knn bsf: inf        ] level  6 - node 1742           LBG dist: 2.509842683932561e-07
[knn bsf: inf        ] level  6 - node 3388           LBG dist: 2.5740939092144853e-07
[knn bsf: inf        ] level  6 - node 11219          LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 587            LBG dist: 2.0284476321066255e-07
[knn bsf: inf        ] level  5 - node 6588           LBG dist: 1.7008854777886197e-07
[knn bsf: inf        ] level  5 - node 13497          LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 7220           LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 13496          LBG dist: 1.035989691849883e-07
[knn bsf: inf        ] level  3 - node 147            LBG dist: 1.914804104559071e-08
[knn bsf: inf        ] level  3 - node 7219           LBG dist: 1.6010274127281743e-07
[knn bsf: inf        ] level  3 - node 8420           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 134            LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 6622           LBG dist: 5.06737662439716e-07
[knn bsf: inf        ] level  2 - node 14926          LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 129            LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 133            LBG dist: 8.348610342357409e-10
[knn bsf: inf        ] level  0 - node 9291           LBG dist: 0.0
[knn bsf: inf        ] level  0 - node 14924          LBG dist: 0.0
[knn bsf: 6.57233e-10] level  0 - node 9291           min. DTW: 4.740883488792752e-10
[knn bsf: 4.79416e-10] level  0 - node 14924          min. DTW: 4.681301632545856e-10
[knn bsf: 4.79416e-10] level  1 - node 7720           LBG dist: 7.530247245002756e-10
[knn bsf: 4.79416e-10] level  1 - node 14925          LBG dist: 0.0
[knn bsf: 4.79416e-10] level  0 - node 128            LBG dist: 0.0
[knn bsf: 4.79416e-10] level  0 - node 7719           LBG dist: 0.0
[knn bsf: 4.79416e-10] level  0 - node 128            min. DTW: 6.012067502745996e-10
[knn bsf: 4.79416e-10] level  0 - node 7719           min. DTW: 7.180280417408994e-10

LBG calculations:         32
examined inner nodes:     13
examined leaf nodes:      4
LB_Kim calculations:      2142
DTW calculations:         244

 k |  ts_id  |  name   |       distance       | start_index | end_index | stats
---+---------+---------+----------------------+-------------+-----------+-------
 1 | 4831023 | Paris   | 4.68130163254586e-10 |         155 |       208 |  ...
 2 | 1430013 | demand  | 4.74088348879275e-10 |         155 |       208 |  ...
 3 | 2269496 | freedom | 4.79415545679705e-10 |         155 |       208 |  ...

Time: 474.690 ms
```

**Listing A.3** Nearest neighbors of *Germany* for the years 1955 – 2008. Note that the resulting neighbors differ from the ones in Listing A.2 where another interval is queried.

```
# SELECT * FROM knn('Microsoft', start_index => 190, end_index => -1,
#                   k => 5, radius => -1, verbose => True);
get time series...
[knn bsf: inf        ] level 10 - node 5318           LBG dist: 2.0242240655347458e-10
[knn bsf: inf        ] level 10 - node 12631          LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 5319           LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 8744           LBG dist: 2.010737231312652e-10
[knn bsf: inf        ] level  8 - node 1777           LBG dist: 1.9733052407232198e-10
[knn bsf: inf        ] level  8 - node 5317           LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 12597          LBG dist: 1.9651699696673137e-10
[knn bsf: inf        ] level  7 - node 1743           LBG dist: 1.9245100340249056e-10
[knn bsf: inf        ] level  7 - node 5316           LBG dist: 0.0
[knn bsf: inf        ] level  7 - node 11617          LBG dist: 1.8638020480582976e-10
[knn bsf: inf        ] level  6 - node 1742           LBG dist: 6.785666430238277e-10
[knn bsf: inf        ] level  6 - node 3388           LBG dist: 8.462378866770938e-10
[knn bsf: inf        ] level  6 - node 11219          LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 587            LBG dist: 6.559819011635096e-10
[knn bsf: inf        ] level  5 - node 6588           LBG dist: 3.2580359813949974e-11
[knn bsf: inf        ] level  5 - node 13497          LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 7220           LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 13496          LBG dist: 1.452001381955064e-11
[knn bsf: inf        ] level  3 - node 147            LBG dist: 0.0
[knn bsf: inf        ] level  3 - node 7219           LBG dist: 0.0
[knn bsf: inf        ] level  3 - node 8420           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 146            LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 8419           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 13304          LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 145            LBG dist: 4.16814561026661e-10
[knn bsf: inf        ] level  1 - node 9323           LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 3622           LBG dist: 7.165357774200008e-11
[knn bsf: inf        ] level  2 - node 7218           LBG dist: 4.3254330487266917e-10
[knn bsf: inf        ] level  2 - node 10227          LBG dist: 5.1099610597444514e-11
[knn bsf: inf        ] level  1 - node 140            LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 8418           LBG dist: 0.0
[knn bsf: inf        ] level  0 - node 138            LBG dist: 0.0
[knn bsf: inf        ] level  0 - node 4252           LBG dist: 0.0
[knn bsf: 8.20201e-10] level  0 - node 138            min. DTW: 4.070376999478622e-10
[knn bsf: 5.41523e-10] level  0 - node 4252           min. DTW: 5.017445068844095e-10
[knn bsf: 5.41523e-10] level  0 - node 6335           LBG dist: 0.0
[knn bsf: 5.41523e-10] level  0 - node 7368           LBG dist: 8.165076328021279e-11
[knn bsf: 4.99888e-10] level  0 - node 6335           min. DTW: 4.633895522821579e-10
[knn bsf: 4.99888e-10] level  2 - node 134            LBG dist: 5.1672611011303917e-11
[knn bsf: 4.99888e-10] level  2 - node 6622           LBG dist: 4.5083040824916254e-07
[knn bsf: 4.99888e-10] level  2 - node 14926          LBG dist: 0.0
[knn bsf: 4.99888e-10] level  0 - node 139            LBG dist: 0.0
[knn bsf: 4.99888e-10] level  0 - node 9306           LBG dist: 1.8053602633340352e-11
[knn bsf: 4.99888e-10] level  0 - node 13666          LBG dist: 1.6383938682866842e-12
[knn bsf: 3.57666e-10] level  0 - node 139            min. DTW: 1.3477038452488 23e-10
[knn bsf: 3.57666e-10] level  1 - node 6336           LBG dist: 0.0
[knn bsf: 3.57666e-10] level  1 - node 13303          LBG dist: 0.0
[knn bsf: 3.57666e-10] level  0 - node 132            LBG dist: 1.6130893524017 59e-09
[knn bsf: 3.57666e-10] level  0 - node 13302          LBG dist: 0.0
[knn bsf: 3.57666e-10] level  0 - node 13302          min. DTW: 3.9043201487942157e-10
[knn bsf: 3.57666e-10] level  0 - node 8417           LBG dist: 0.0
[knn bsf: 3.57666e-10] level  0 - node 11059          LBG dist: 0.0
[knn bsf: 3.57666e-10] level  0 - node 8417           min. DTW: 5.679729295600564e-10
[knn bsf: 1.79091e-10] level  0 - node 11059          min. DTW: 1.1112064920617997e-10
[knn bsf: 1.79091e-10] level  1 - node 7720           LBG dist: 2.208433518027373e-08
```

```
[knn bsf: 1.79091e-10] level  1 - node 14925          LBG dist: 0.0
[knn bsf: 1.79091e-10] level  0 - node 128            LBG dist: 0.0
[knn bsf: 1.79091e-10] level  0 - node 7719           LBG dist: 1.2724662036862344e-08
[knn bsf: 1.79091e-10] level  0 - node 128            min. DTW: 4.469617605410457e-10
[knn bsf: 1.79091e-10] level  0 - node 13666          min. DTW: 3.15338366423286e-10
[knn bsf: 1.79091e-10] level  3 - node 3578           LBG dist: 2.479937062711201e-10
[knn bsf: 1.79091e-10] level  3 - node 13495          LBG dist: 2.3786070683897696e-10
[knn bsf: 1.79091e-10] level  3 - node 13577          LBG dist: 8.566082210474352e-11
[knn bsf: 1.79091e-10] level  0 - node 9306           min. DTW: 5.288324962409504e-10
[knn bsf: 1.79091e-10] level  4 - node 611            LBG dist: 6.000479074288123e-11
[knn bsf: 1.79091e-10] level  4 - node 2492           LBG dist: 1.5306423573601883e-10
[knn bsf: 1.79091e-10] level  4 - node 16388          LBG dist: 1.6956650552269606e-10
[knn bsf: 1.79091e-10] level  1 - node 3621           LBG dist: 3.656143360858362e-10
[knn bsf: 1.79091e-10] level  1 - node 10226          LBG dist: 1.8919848929042113e-10
[knn bsf: 1.79091e-10] level  1 - node 129            LBG dist: 5.1672611011303917e-11
[knn bsf: 1.79091e-10] level  1 - node 133            LBG dist: 2.0819389046355206e-08
[knn bsf: 1.79091e-10] level  0 - node 9291           LBG dist: 5.73694450778397e-11
[knn bsf: 1.79091e-10] level  0 - node 14924          LBG dist: 3.696261480771607e-10
[knn bsf: 1.79091e-10] level  0 - node 9291           min. DTW: 5.454295872557366e-10
[knn bsf: 1.79091e-10] level  3 - node 302            LBG dist: 4.479325417496809e-10
[knn bsf: 1.79091e-10] level  3 - node 7573           LBG dist: 2.0284994281592104e-10
[knn bsf: 1.79091e-10] level  3 - node 14789          LBG dist: 4.3496402346397064e-10
[knn bsf: 1.79091e-10] level  1 - node 170            LBG dist: 1.1233211648369213e-09
[knn bsf: 1.79091e-10] level  1 - node 9762           LBG dist: 3.1510635881383756e-10
[knn bsf: 1.79091e-10] level  0 - node 7368           min. DTW: 8.842193648181429e-10
[knn bsf: 1.79091e-10] level  2 - node 301            LBG dist: 4.172609075245131e-10
[knn bsf: 1.79091e-10] level  2 - node 13576          LBG dist: 4.3011495719803036e-10
[knn bsf: 1.79091e-10] level  3 - node 1068           LBG dist: 4.808619308227447e-10
[knn bsf: 1.79091e-10] level  3 - node 5487           LBG dist: 3.928554057541039e-10
[knn bsf: 1.79091e-10] level  3 - node 2317           LBG dist: 4.489510651596557e-10
[knn bsf: 1.79091e-10] level  3 - node 16387          LBG dist: 4.204858013159738e-10
[knn bsf: 1.79091e-10] level  3 - node 16976          LBG dist: 4.4345386367506687e-10

LBG calculations:         75
examined inner nodes:     31
examined leaf nodes:      12
LB_Kim calculations:      7989
DTW calculations:         2954

 k |  ts_id  |  name  |       distance       | start_index | end_index | stats
---+---------+--------+----------------------+-------------+-----------+-------
 1 | 2968781 | IP     |  1.1112064920618e-10 |         190 |       208 |  ...
 2 | 1146578 | Click  |  1.1620426968856e-10 |         190 |       208 |  ...
 3 | 3146497 | Java   | 1.34770384524882e-10 |         190 |       208 |  ...
 4 | 1531883 | dialog | 1.40257448743664e-10 |         190 |       208 |  ...
 5 | 5465057 | Server | 1.79090613709668e-10 |         190 |       208 |  ...

Time: 900.352 ms
```

**Listing A.4** Nearest neighbors of *Microsoft* for the years 1990 – 2008. Note that the last examined nodes are inner nodes, as their parent nodes have a LBG distance smaller than the best-so-far distance (`bsf`). The algorithm breaks once the first node on the min-heap has a LBG distance larger than the best-so-far distance.

### A.3.2 Queries on the 103-Million Dataset

```
# SELECT * FROM knn('and the', start_index => 190, end_index => -1,
#                   k => 1, radius => -1, verbose => True);
get time series...
[knn bsf: inf        ] level 14 - node 119218         LBG dist: 5.4282091029398164e-05
[knn bsf: inf        ] level 14 - node 235501         LBG dist: 0.0
[knn bsf: inf        ] level 13 - node 119219         LBG dist: 5.428154159372357e-05
[knn bsf: inf        ] level 13 - node 235499         LBG dist: 0.0
[knn bsf: inf        ] level 12 - node 119217         LBG dist: 5.428048887737655e-05
[knn bsf: inf        ] level 12 - node 235498         LBG dist: 0.0
[knn bsf: inf        ] level 11 - node 119216         LBG dist: 5.427937674565127e-05
[knn bsf: inf        ] level 11 - node 235497         LBG dist: 0.0
[knn bsf: inf        ] level 10 - node 119215         LBG dist: 5.427636669450485e-05
[knn bsf: inf        ] level 10 - node 160225         LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 97259          LBG dist: 5.42730964034773e-05
[knn bsf: inf        ] level  9 - node 149173         LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 25106          LBG dist: 5.426609570241282e-05
[knn bsf: inf        ] level  8 - node 120530         LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 170893         LBG dist: 5.425029704579764e-05
[knn bsf: inf        ] level  7 - node 61280          LBG dist: 5.422558682749819e-05
[knn bsf: inf        ] level  7 - node 120529         LBG dist: 5.426512058400254e-05
[knn bsf: inf        ] level  7 - node 143518         LBG dist: 0.0
[knn bsf: inf        ] level  6 - node 86030          LBG dist: 0.0
[knn bsf: inf        ] level  6 - node 143517         LBG dist: 5.47076576741661e-05
[knn bsf: inf        ] level  5 - node 13256          LBG dist: 5.460572426394529e-05
[knn bsf: inf        ] level  5 - node 86029          LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 163040         LBG dist: 5.460570010581543e-05
[knn bsf: inf        ] level  4 - node 810            LBG dist: 1.8246882339278587e-05
[knn bsf: inf        ] level  4 - node 86028          LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 207152         LBG dist: 1.8191058734563358e-05
[knn bsf: inf        ] level  3 - node 457            LBG dist: 0.0
[knn bsf: inf        ] level  3 - node 86027          LBG dist: 2.735854560604522e-05
[knn bsf: inf        ] level  3 - node 221902         LBG dist: 2.7242370486059292e-05
[knn bsf: inf        ] level  2 - node 14             LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 456            LBG dist: 4.406678191897728e-05
[knn bsf: inf        ] level  2 - node 222965         LBG dist: 4.225945913650783e-05
[knn bsf: inf        ] level  1 - node 6              LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 9              LBG dist: 4.6773398496441175e-05
[knn bsf: inf        ] level  1 - node 227024         LBG dist: 4.2412721726216795e-05
[knn bsf: inf        ] level  0 - node 2              LBG dist: 0.0006023584819906798
[knn bsf: inf        ] level  0 - node 3              LBG dist: 0.0
[knn bsf: 4.44487e-07] level  0 - node 3              min. DTW: 4.4448727202399743e-07

LBG calculations:         37
examined inner nodes:     15
examined leaf nodes:      1
LB_Kim calculations:      325
DTW calculations:         181

 k |  ts_id   |  name  |       distance       | start_index | end_index | stats
---+----------+--------+----------------------+-------------+-----------+-------
 1 | 63806928 | on the | 4.44487272023997e-07 |         190 |       208 |  ...

Time: 58.644 ms
```

**Listing A.5** Nearest neighbors of *and the* for the years 1990 – 2008.

```
# SELECT * FROM knn('the United', start_index => -1, end_index => -1,
#                   k => 1, radius => -1, verbose => True);
get time series...
[knn bsf: inf        ] level 14 - node 119218          LBG dist: 1.826339364954095e-07
[knn bsf: inf        ] level 14 - node 235501          LBG dist: 0.0
[knn bsf: inf        ] level 13 - node 119219          LBG dist: 1.8252825135969383e-07
[knn bsf: inf        ] level 13 - node 235499          LBG dist: 0.0
[knn bsf: inf        ] level 12 - node 119217          LBG dist: 1.82325843055561e-07
[knn bsf: inf        ] level 12 - node 235498          LBG dist: 0.0
[knn bsf: inf        ] level 11 - node 119216          LBG dist: 1.8211213065783284e-07
[knn bsf: inf        ] level 11 - node 235497          LBG dist: 0.0
[knn bsf: inf        ] level 10 - node 119215          LBG dist: 1.8153432291668496e-07
[knn bsf: inf        ] level 10 - node 160225          LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 97259           LBG dist: 1.809075821485947e-07
[knn bsf: inf        ] level  9 - node 149173          LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 25106           LBG dist: 1.795695003428566e-07
[knn bsf: inf        ] level  8 - node 120530          LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 170893          LBG dist: 1.765677561658728e-07
[knn bsf: inf        ] level  7 - node 61280           LBG dist: 2.4315641558358934e-06
[knn bsf: inf        ] level  7 - node 120529          LBG dist: 2.453721322013358e-06
[knn bsf: inf        ] level  7 - node 143518          LBG dist: 0.0
[knn bsf: inf        ] level  6 - node 86030           LBG dist: 0.0
[knn bsf: inf        ] level  6 - node 143517          LBG dist: 1.50276867097708e-06
[knn bsf: inf        ] level  5 - node 13256           LBG dist: 2.6244576711015473e-06
[knn bsf: inf        ] level  5 - node 86029           LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 163040          LBG dist: 2.6179739338393253e-06
[knn bsf: inf        ] level  4 - node 810             LBG dist: 4.522902016268542e-07
[knn bsf: inf        ] level  4 - node 86028           LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 207152          LBG dist: 4.3667315637063347e-07
[knn bsf: inf        ] level  3 - node 457             LBG dist: 0.0
[knn bsf: inf        ] level  3 - node 86027           LBG dist: 7.825428943506582e-07
[knn bsf: inf        ] level  3 - node 221902          LBG dist: 7.024008699847457e-07
[knn bsf: inf        ] level  2 - node 14              LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 456             LBG dist: 1.1185841457901548e-06
[knn bsf: inf        ] level  2 - node 222965          LBG dist: 6.997132874688362e-07
[knn bsf: inf        ] level  1 - node 6               LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 9               LBG dist: 2.2953788473650518e-07
[knn bsf: inf        ] level  1 - node 227024          LBG dist: 3.0835804262542576e-09
[knn bsf: inf        ] level  0 - node 2               LBG dist: 0.02189835768101354
[knn bsf: inf        ] level  0 - node 3               LBG dist: 0.0
[knn bsf: 2.96448e-08] level  0 - node 3               min. DTW: 2.964480660174612e-08
[knn bsf: 2.96448e-08] level  0 - node 4               LBG dist: 4.347839287126069e-09
[knn bsf: 2.96448e-08] level  0 - node 224637          LBG dist: 9.464199988897262e-08
[knn bsf: 2.96448e-08] level  0 - node 4               min. DTW: 1.9423857190330249e-07

LBG calculations:         39
examined inner nodes:     16
examined leaf nodes:      2
LB_Kim calculations:      1191
DTW calculations:         295

 k |  ts_id   |     name      |       distance       | start_index | end_index | stats
---+----------+---------------+----------------------+-------------+-----------+-------
 1 | 96222167 | United States | 2.96448066017461e-08 |           0 |       208 |  ...

Time: 7600.031 ms (00:07.600)
```

**Listing A.6** Nearest neighbors of *the United* for the years 1800 – 2008.

```
# SELECT * FROM knn('the United', start_index => 100, end_index => 130,
#                   k => 2, radius => -1, verbose => True);
get time series...
[knn bsf: inf        ] level 14 - node 119218          LBG dist: 8.062464804207799e-07
[knn bsf: inf        ] level 14 - node 235501          LBG dist: 0.0
[knn bsf: inf        ] level 13 - node 119219          LBG dist: 8.061609508853667e-07
[knn bsf: inf        ] level 13 - node 235499          LBG dist: 0.0
[knn bsf: inf        ] level 12 - node 119217          LBG dist: 8.059970881347681e-07
[knn bsf: inf        ] level 12 - node 235498          LBG dist: 0.0
[knn bsf: inf        ] level 11 - node 119216          LBG dist: 8.058239933261124e-07
[knn bsf: inf        ] level 11 - node 235497          LBG dist: 0.0
[knn bsf: inf        ] level 10 - node 119215          LBG dist: 8.053555862054853e-07
[knn bsf: inf        ] level 10 - node 160225          LBG dist: 0.0
[knn bsf: inf        ] level  9 - node 97259           LBG dist: 8.048468216587114e-07
[knn bsf: inf        ] level  9 - node 149173          LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 25106           LBG dist: 8.037581999942891e-07
[knn bsf: inf        ] level  8 - node 120530          LBG dist: 0.0
[knn bsf: inf        ] level  8 - node 170893          LBG dist: 8.013039300431562e-07
[knn bsf: inf        ] level  7 - node 61280           LBG dist: 8.737239722849553e-07
[knn bsf: inf        ] level  7 - node 120529          LBG dist: 8.809948916855471e-07
[knn bsf: inf        ] level  7 - node 143518          LBG dist: 0.0
[knn bsf: inf        ] level  6 - node 86030           LBG dist: 0.0
[knn bsf: inf        ] level  6 - node 143517          LBG dist: 8.729475382104447e-07
[knn bsf: inf        ] level  5 - node 13256           LBG dist: 8.69579058277419e-07
[knn bsf: inf        ] level  5 - node 86029           LBG dist: 0.0
[knn bsf: inf        ] level  5 - node 163040          LBG dist: 8.66333131044798e-07
[knn bsf: inf        ] level  4 - node 810             LBG dist: 8.647254945265137e-07
[knn bsf: inf        ] level  4 - node 86028           LBG dist: 0.0
[knn bsf: inf        ] level  4 - node 207152          LBG dist: 8.613974605715504e-07
[knn bsf: inf        ] level  3 - node 457             LBG dist: 0.0
[knn bsf: inf        ] level  3 - node 86027           LBG dist: 8.866930945679905e-07
[knn bsf: inf        ] level  3 - node 221902          LBG dist: 8.236557026781433e-07
[knn bsf: inf        ] level  2 - node 14              LBG dist: 0.0
[knn bsf: inf        ] level  2 - node 456             LBG dist: 1.0285315664521846e-06
[knn bsf: inf        ] level  2 - node 222965          LBG dist: 9.223030029750501e-07
[knn bsf: inf        ] level  1 - node 6               LBG dist: 0.0
[knn bsf: inf        ] level  1 - node 9               LBG dist: 5.878268680118461e-07
[knn bsf: inf        ] level  1 - node 227024          LBG dist: 3.0255240544908576e-08
[knn bsf: inf        ] level  0 - node 2               LBG dist: 0.0035411292104014776
[knn bsf: inf        ] level  0 - node 3               LBG dist: 0.0
[knn bsf: 1.81180e-08] level  0 - node 3               min. DTW: 1.0009375719301362e-08

LBG calculations:         37
examined inner nodes:     15
examined leaf nodes:      1
LB_Kim calculations:      325
DTW calculations:         185

 k |  ts_id   |     name      |       distance       | start_index | end_index | stats
---+----------+---------------+----------------------+-------------+-----------+-------
 1 | 60130037 | New York      | 1.00093757193014e-08 |         100 |       130 |  ...
 2 | 96222167 | United States | 1.81180182953422e-08 |         100 |       130 |  ...

Time: 108.803 ms
```

**Listing A.7** Nearest neighbors of *the United* for the years 1900 – 1930. Note that the resulting neighbors differ from the ones in Listing A.6 where another interval is queried.