

Beyond one billion time series: indexing and mining very large time series collections with *i*SAX2+

Alessandro Camerra · Jin Shieh · Themis Palpanas ·
Thanawin Rakthanmanon · Eamonn Keogh

Received: 23 March 2012 / Revised: 23 September 2012 / Accepted: 28 December 2012
© Springer-Verlag London 2013

Abstract There is an increasingly pressing need, by several applications in diverse domains, for developing techniques able to index and mine very large collections of time series. Examples of such applications come from astronomy, biology, the web, and other domains. It is not unusual for these applications to involve numbers of time series in the order of hundreds of millions to billions. However, all relevant techniques that have been proposed in the literature so far have not considered any data collections much larger than one-million time series. In this paper, we describe *i*SAX 2.0 and its improvements, *i*SAX 2.0 Clustered and *i*SAX2+, three methods designed for indexing and mining truly massive collections of time series. We show that the main bottleneck in mining such massive datasets is the time taken to build the index, and we thus introduce a novel bulk loading mechanism, the first of this kind specifically tailored to a time series index. We show how our methods allows mining on datasets that would otherwise be completely untenable, including the first published experiments to index one billion time series, and experiments in mining massive data from domains as diverse as entomology, DNA and web-scale image collections.

Keywords Time series · Data mining · Representations · Indexing · Bulk loading

A. Camerra · T. Palpanas (✉)
University of Trento, Trento, Italy
e-mail: themis@disi.unitn.eu

A. Camerra
e-mail: a.camerra@studenti.unitn.it

J. Shieh · E. Keogh
University of California, Riverside, CA, USA
e-mail: shiehj@cs.ucr.edu

E. Keogh
e-mail: eamonn@cs.ucr.edu

T. Rakthanmanon
Kasetsart University, Bangkok, Thailand
e-mail: fengtwr@ku.ac.th

1 Introduction

The problem of indexing and mining time series has captured the interest of the data mining and database community for almost two decades [5, 7, 16, 21, 23, 26, 38]. Indeed, several data mining techniques (e.g., clustering, classification, outlier detection, pattern identification, motif discovery, and others) have been developed for and applied to time series data. Time series indexing is relevant to all these techniques, because it allows them to execute fast on large collections of time series.

However, there remains a huge gap between the scalability of the methods in the current literature, and the needs of practitioners in many domains, ranging from astronomy [34] to neuroscience [1]. To illustrate this gap, consider the selection of quotes from unsolicited emails sent to the current authors, asking for help in indexing massive time series datasets.

- “... we have about a million samples per minute coming in from 1000 gas turbines around the world... we need to be able to do similarity search for ...” Lane Desborough, General Electric.
- “... an archival rate of 3.6 billion points a day, how can we (do similarity search) in this data?” Josh Patterson, Tennessee Valley Authority.

Our communication with such companies and research institutions has lead us to the perhaps surprising conclusion: For all attempts at large-scale mining of time series, it is the *time complexity of building the index* that remains the most significant bottleneck, for example, a state-of-the-art method [21] needs over 6 days to build an index with 100-million items.

Additionally, there is a pressing need to reduce retrieval times, especially as such data are clearly doomed to be disk resident. Once a dimensionality-reduced representation (i.e., Discrete Fourier Transform—DFT, Discrete Wavelet Transform—DWT, and Symbolic Aggregate Approximation—SAX) has been decided on, the only way to improve retrieval times is by optimizing splitting algorithms for tree-based indexes (i.e., R-trees and M-trees), since a poor splitting policy leads to excessive and useless subdivisions, which create unnecessarily deep subtrees and cause lengthier traversals.

In this work, we solve both of these problems, by proposing the first bulk loading technique for a time series index. We achieve this by introducing significant extensions to the recently introduced multi-resolution symbolic representation *indexable Symbolic Aggregate approximation (iSAX)* [21]. As we will show with the largest (by far) set of time series indexing experiments ever attempted, we can reduce the index building time by 82 % with a novel bulk loading scheme, which is the first bulk loading algorithm for a time series index. Also, our new splitting policy reduces the size of the index by 27 %. The number of disk page accesses is reduced by more than 70 %, while more than 99.5 % of those accesses are sequential.

To push the limits of time series data mining, we consider experiments that index 1,000,000,000 (one billion) time series of length 256. To the best of our knowledge, this is the first time a paper in the literature has reached the one billion mark for similarity search on multimedia objects of any kind. Previous studies on indexing time series have considered datasets of 1,000,000 objects [5], 500,000 objects [16], 100,000 objects [23], 6,480 objects [7], and 27,000 objects [26]. Thus, the 1,000,000,000 objects considered here represent real progress, beyond the inevitable improvements in hardware performance.

We further show that the scalability achieved by our ideas allows us to consider interesting data mining problems in entomology, biology, and the web that would otherwise be untenable. The contributions we make in this paper can be summarized as follows.

- We present mechanisms that allow *iSAX* 2.0, a data structure suitable for indexing and mining time series, to scale to very large datasets.
- We propose a new node splitting algorithm, based on simple statistics that are accurate, yet efficient to compute. This algorithm leads to an average reduction in the size of the index by 27 %.
- We introduce the first bulk loading algorithm, specifically designed to operate in the context of a time series index. The proposed algorithm can dramatically reduce the number of random disk page accesses (as well as the total number of disk accesses), thus reducing the time required to build the index by an order of magnitude.
- Furthermore, we describe *iSAX* 2.0 Clustered and *iSAX*2+, which improve on our bulk loading algorithm by optimizing the way the raw time series data are managed. These new algorithms lead up to a *further* 40 % reduction in the index build time.
- Finally, we present the first approach that is experimentally validated to scale to data collections of time series with up to 1 *billion* objects, giving practitioners in various domains the opportunity for pain-free analysis and mining of their time series collections (even on commodity hardware).

We note that this paper is an extension of our previous work on *iSAX* 2.0 [8]. Apart from including more details and figures related to the description of *iSAX* 2.0, as well as of the experiment based on the DNA data, we also present two new algorithms, namely *iSAX* 2.0 Clustered and *iSAX*2+, along with a new set of experiments that evaluate their performance and show significant improvements over the previous approach.

The rest of the paper is organized as follows. We review some background material in Sect. 2. Section 3 introduces the basic pillars for our scalable index, *iSAX* 2.0. In Sect. 4, we describe *iSAX* 2.0 Clustered and *iSAX*2+, which further improve our bulk loading mechanism. Section 5 discusses the experimental evaluation. Section 6 presents the related work, and Sect. 7 the conclusions.

2 Preliminaries

As noted previously, there are numerous dimensionality reduction techniques available for time series. In this section, we review SAX and its recent extension, *iSAX*, which are at the heart of our proposed ideas. (For a more detailed discussion, refer to [21].) We note that recently others have found *iSAX* useful, for example a group of researchers working on gesture recognition [27], and another group in order to find time series motifs [9] and understand their statistical significance [10].

2.1 The SAX representation

In Fig. 1a, we show a time series T of length $n = 16$. This time series can be represented in w -dimensional space by a vector of real numbers $\vec{C} = \bar{c}_1, \dots, \bar{c}_w$. The i th element of \vec{C} is calculated by:

$$\bar{c}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} T_j$$

Fig. 1b shows T converted into this representation (called Piecewise Aggregate Approximation, or PAA [24]) reducing the dimensionality from 16 to 4.

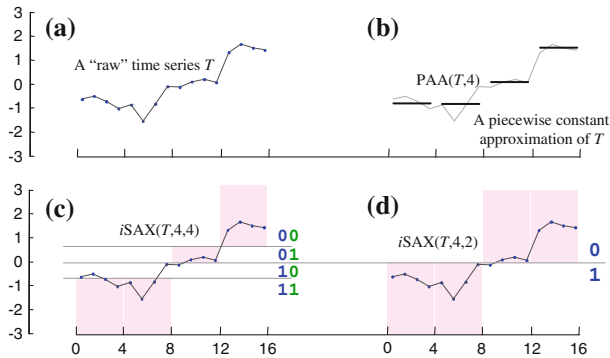


Fig. 1 **a** A time series T , of length 16. **b** A PAA approximation of T , with 4 segments. A time series T converted into SAX words of cardinality 4 **c** and cardinality 2 **d**

Table 1 Converting to a reduced (by half) cardinality SAX word by ignoring trailing bits

$SAX(T, 4, 16) = \mathbf{T}^{16} = \{1100, 1101, 0110, 0001\}$
$SAX(T, 4, 8) = \mathbf{T}^8 = \{110, 110, 011, 000\}$
$SAX(T, 4, 4) = \mathbf{T}^4 = \{11, 11, 01, 00\}$
$SAX(T, 4, 2) = \mathbf{T}^2 = \{1, 1, 0, 0\}$

Note that the PAA coefficients are intrinsically real-valued, and for reasons we will make clear later, it can be advantageous to have *discrete* coefficients. We can achieve this discreteness with SAX. The SAX representation takes the PAA representation as an input and discretizes it into a small alphabet of symbols with cardinality a . The discretization is achieved by creating a series of breakpoints running parallel to the x -axis and labeling each region with a discrete label. Any PAA segment that falls in that region can then be mapped to the appropriate label.

The SAX representation supports arbitrary breakpoints; however, it has been shown that an effective choice is a sorted list of numbers $Breakpoints = \beta_1, \dots, \beta_{a-1}$ such that the area under a $N(0, 1)$ Gaussian curve from β_i to $\beta_{i+1} = 1/a$ produces symbols with approximate equi-probability.

A SAX word is simply a vector of discrete numbers. For example, the SAX word shown in Fig. 1c can be written as $\{3, 3, 1, 0\}$ or in binary form as $\{11, 11, 01, 00\}$.

We denote this word as \mathbf{T}^4 and assume that it is produced by the function $SAX(T, 4, 4)$. The “ \mathbf{T} ” is written in boldface to distinguish it from the raw data from which it was derived, and the superscript of “4” denotes the cardinality of the symbols. Under this notation, the SAX word shown in Fig. 1d can be written as $SAX(T, 4, 2) = \mathbf{T}^2 = \{1, 1, 0, 0\}$. Note that once we have \mathbf{T}^4 , we can derive \mathbf{T}^2 by simply ignoring the trailing bits from each symbol within the SAX word. Naturally, this is a recursive property. If we converted T to SAX with a cardinality of 8, we have $SAX(T, 4, 8) = \mathbf{T}^8 = \{110, 110, 011, 000\}$; from this, we can convert to any lower resolution that differs by a power of two, by ignoring the correct number of bits. Table 1 makes this clearer.

The ability to change cardinalities on the fly is exploitable by our splitting policies, as we will demonstrate in Sect. 3.2.

2.2 The iSAX representation

It is tedious to write out binary strings, so we can use integers to represent SAX symbols. For example:

$$\text{SAX}(T, 4, 8) = \mathbf{T}^8 = \{\mathbf{110}, \mathbf{110}, \mathbf{011}, \mathbf{000}\} = \{6, 6, 3, 0\}$$

However, this can make the SAX word ambiguous, since we cannot be sure what the cardinality is (although we know it is at least 7). We resolve this ambiguity by writing the cardinality as a superscript. From the above example:

$$i\text{SAX}(T, 4, 8) = \mathbf{T}^8 = \{6^8, 6^8, 3^8, 0^8\}$$

One of the key properties of the *iSAX* representation is the ability to compare two *iSAX* words of different cardinalities, as well as *iSAX* words where *each* word has mixed cardinalities (such as $\{\mathbf{111}, \mathbf{11}, \mathbf{101}, \mathbf{0}\} = \{7^8, 3^4, 5^8, 0^2\}$) 0.

iSAX support for mixed cardinalities is a feature which allows an index structure to split along any arbitrary dimension or symbol. It is this flexibility which allows *iSAX* to be indexable (as opposed to classic SAX). As we demonstrate in the follow sections, we can exploit this property to create a novel splitting policy that allows for extremely efficient indexing of massive datasets.

2.3 Indexing iSAX

iSAX's variable granularity allows us to index time series. Using the *iSAX* representation, and by defining values for the cardinality b and wordlength w , we can produce a set of b^w different mutually exclusive *iSAX* words. These can be represented by files on disk, for example the word $\{6^8, 6^8, 3^8, 0^8\}$ can be mapped to `6.8_6.8_3.8_0.8.txt`

A user-defined threshold th defines the maximum number of time series that a file can hold.

Imagine that we are in the process of building an index and have chosen $th = 100$. At some point, there may be exactly 100 time series mapped to the *iSAX* word $\{2^4, 3^4, 3^4, 2^4\}$. If we come across another time series that maps in the same place, we have an overflow, so we need to split the file. The idea is to choose one *iSAX* symbol, examine an additional bit, and use its value to create two new files. In this case, the original file: $\{2^4, 3^4, 3^4, 2^4\}$ splits into $\{4^8, 3^4, 3^4, 2^4\}$ (child file 1), and $\{5^8, 3^4, 3^4, 2^4\}$ (child file 2). For some time series in the file, the extra bit in their first *iSAX* symbol was a 1 and for others, it was a 0. In the former case, they are remapped to child 1, while in the latter to child 2.

The use of the *iSAX* representation has led to the creation of a hierarchical, but unbalanced, index structure that contains non-overlapping regions and has a controlled fan-out rate. The three classes of nodes found in this index structure are described below.

Root Node: The root node is the representative of the complete *iSAX* space and is similar in functionality to an internal node. The root node contains no SAX representation, but only pointers to the children nodes (in the worst case, their number is 2^w).

Leaf Node: This is a leaf level node, which contains a pointer to an index file on disk with the raw time series entries. The node itself stores the highest cardinality *iSAX* word for each time series.

Internal Node: An internal node designates a split in *iSAX* space and is created when the number of time series contained by a leaf node exceeds th . The internal node splits the *iSAX* space by promotion of cardinal values along one or more dimensions as per the iterative doubling policy. *iSAX* employs binary splits along a single dimension, using round robin to

determine the split dimension. Thus, internal nodes store a SAX representation and pointers to their two children.

3 The *i*SAX 2.0 index

As discussed earlier, *i*SAX is a tree structure that is not balanced. In addition, there is no special provision for mechanisms that can facilitate the ingestion of large collections of time series into the index. Through our initial experimentation, we observed that these characteristics can lead to prohibitively long index creation times. For example, indexing a dataset with 500 million time series would need 20 days to complete. Even a modest dataset with 100 million time series requires 2 days in order to be indexed (detailed results are presented in Sect. 5).

Clearly, having to wait for such an extended amount of time before analysis and mining is impractical. This becomes even more pronounced in applications where large numbers of time series are produced on a regular basis and need to be analyzed before proceeding with additional experiments.

Note that the above criticism of *i*SAX refers mainly to index construction and not the utility of the index. Previous work has demonstrated the effectiveness and efficiency of *i*SAX for performing various data analysis and mining tasks [21]. The performance of *i*SAX on these tasks scales sub-linearly as a function of the number of time series indexed. During index creation, the primary bottleneck is hard drive performance and the associated I/O costs. As the amount of indexed data increases, this bottleneck becomes a hard constraint which limits the overall scalability of the index.

In order to overcome the above problems, we propose the following two complementary techniques to improve the scalability of *i*SAX.

- A new algorithm for time series bulk loading that considerably reduces the number of total disk page accesses, while also minimizing the number of random disk page accesses.
- A new splitting policy for the internal nodes of the index resulting in a significantly more compact indexing structure, hence further reducing the I/O cost.

In the following sections, we discuss in more detail these extensions of the *i*SAX index structure that enables it to efficiently operate with data collections orders of magnitude larger than previously tested. We will refer to this *new i*SAX index as *i*SAX 2.0.

3.1 Bulk loading: main algorithm

Inserting a large collection of time series into the index iteratively is a very expensive operation, involving a high number of disk I/O operations. This is because for each time series, we have to store the raw data on disk and insert into the index the corresponding *i*SAX representation. Assuming that the entire index is in main memory, the above procedure translates to one random disk access for every time series in the dataset in the best case (when there is no leaf node split), or more random accesses otherwise.

We now describe an algorithm for bulk loading, which can effectively reduce the number of disk I/O operations. The main idea of the algorithm is that instead of developing the entire index at once, we are focusing our efforts on building the distinct subtrees of the index one at a time. This is beneficial, because by growing a specific subtree of the index, we are effectively minimizing the number of node split operations and streamlining all the disk accesses. Using the proposed algorithm, we can achieve the following.

- Minimize the required disk I/O, since we avoid revisiting leaf nodes in order to split them (which would mean extra disk accesses to read their contents from disk and then writing back the contents of the new leaf nodes). At the same time, we make sure that every time we access the disk for writing the contents of a leaf node, we write on disk all of its contents at once.
- Maximize the number of *sequential* disk page accesses, in the case where the contents of a leaf node do not fit in a single disk page.

We note that the algorithm we propose is novel since the existing approaches on bulk loading are not applicable in our case (we discuss this in detail in Sect. 6).

3.1.1 Algorithm basics

In order to achieve the goals mentioned above, we need to effectively group the time series that will end up in a particular subtree of the index and process them all together. If we could fit all time series in main memory, then it would be possible to create such groups after processing all time series. We could subsequently build each distinct subtree of the index sequentially, creating all necessary leaf nodes one after the other, without needing to revisit any of the leaf nodes already created.

In our case, however, we have to develop a solution under the (realistic) assumption that the entire dataset does not fit in main memory. In the following paragraphs, we discuss the details of the bulk loading algorithm we propose, which operates under the assumption of limited main memory (i.e., less than necessary to fit the index and the entire dataset). The pseudocode of the algorithm is depicted in Fig. 3.

Our algorithm uses two main memory buffer layers, namely *First Buffer Layer (FBL)* and *Leaf Buffer Layer (LBL)*. The FBL corresponds to the first level of *iSAX 2.0* nodes. This correspondence remains stable throughout the creation of the index, because unlike nodes in other indexing structures, *iSAX 2.0* nodes are not subject to shifts in the course of repetitive insertions (since changes in the leaf nodes due to splits are not propagated upwards the *iSAX 2.0* tree). The LBL corresponds to leaf nodes. There are no buffers related to the internal (i.e., other than the first level) *iSAX 2.0* nodes.

These two buffering layers are different in nature. The role of the buffers in FBL is to cluster together time series that will end up in the same *iSAX 2.0* subtree, rooted in one of the direct children of the root. The buffers in FBL do *not* have a restriction in their size and they grow till they occupy all the available main memory. In contrast, the buffers in LBL are used to gather all the time series of leaf nodes and flush them to disk. These buffers have the same size as the size of the leaf nodes (on disk), which in general is more than a single disk page.

3.1.2 Description of the algorithm

The algorithm operates in two phases, which alternate until the entire dataset is processed (i.e., indexed).

Phase 1: The algorithm reads time series and inserts them in the corresponding buffer in FBL (lines 4–16 in Fig. 3). This phase continues until the main memory is almost full. (We need a small amount of extra memory to allocate new nodes during Phase 2. Yet, this is only needed for the beginning of the first iteration of the loop at lines 12–16, since each iteration releases memory.)

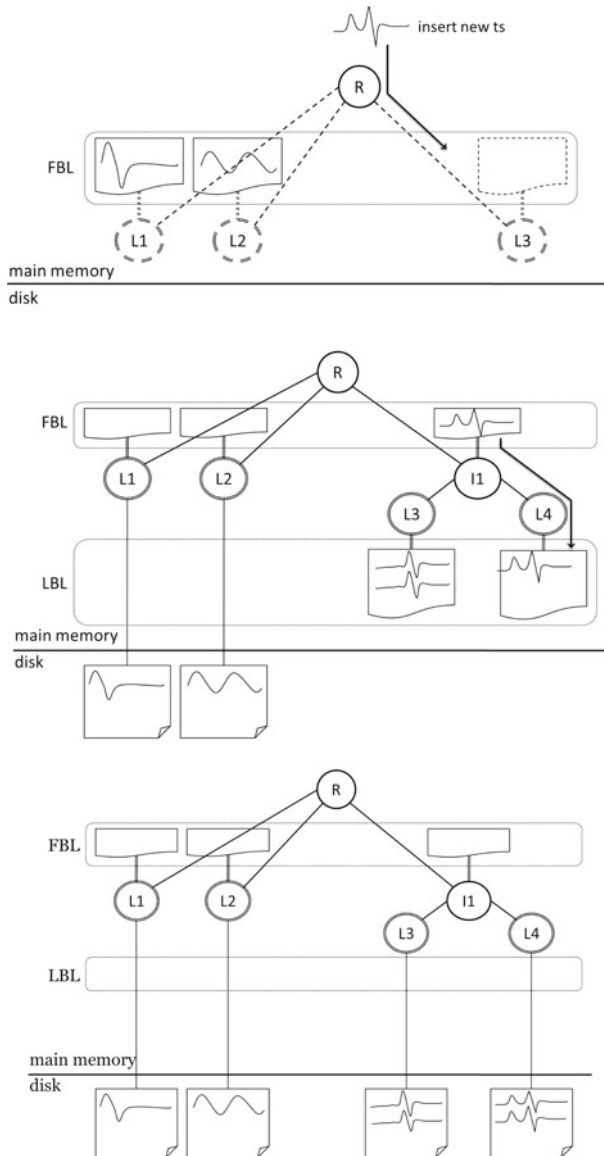


Fig. 2 The bulk loading algorithm. (*top*) Phase 1 fills the FBL buffers with time series until main memory is full. (*middle*) Phase 2, processing subtree rooted at node $I1$ (subtrees rooted at nodes $L1$ and $L2$ have already been flushed to disk). (*bottom*) After phase 2

At the end of Phase 1, we have time series collected in the FBL buffers. This situation is depicted in Figure 2 (*top*). Note that even though we have created some FBL buffers (according to the time series processed so far), the corresponding (leaf) nodes $L1$, $L2$, and $L3$, of the index are not yet created.

Phase 2: The algorithm proceeds by moving the time series contained in each FBL buffer to the appropriate LBL buffers. During this phase, the algorithm processes the buffers in FBL sequentially. For each FBL buffer, the algorithm reads the time series and creates all the

necessary internal (lines 25–33) and leaf (lines 36–39) *i*SAX 2.0 nodes in order to index these time series. It basically creates the entire subtree (or any missing nodes in case a subtree has already been constructed) rooted at the node corresponding to that FBL buffer. For example, in Figure 2 (*middle*), by emptying the right-most FBL buffer, we create the subtree rooted at internal node *I*1. The algorithm also creates for each leaf node a corresponding LBL buffer (line 38). When all time series of a specific FBL buffer have been moved down to the corresponding LBL buffers, the algorithm flushes these LBL buffers to disk (line 15). Notice that in Figure 2 (*bottom*), the LBL buffers for the subtrees rooted at nodes *L*1 and *L*2 have already been flushed to disk, and all the available memory can be dedicated to the LBL buffers of the *I*1 subtree.

At the end of Phase 2 of the algorithm, all the time series from the FBL buffers have moved down the tree to the appropriate leaf nodes (creating new ones if necessary) and LBL buffers, and then from the LBL buffers to the disk [Fig. 2 (*bottom*)]. This means that all buffers (both FBL and LBL) are empty, and we are ready to continue processing the dataset, going back to Phase 1 of the algorithm. This process continues until the entire dataset has been indexed.

Note that the way the algorithm works (Fig. 3), all LBL buffers are flushed to disk at the end of Phase 2. An interesting question is whether we would gain in performance by not flushing the buffers that are almost empty (thus saving disk accesses that do little actual work). This strategy would certainly be beneficial for the first time around. It turns out however that overall it would not lead to better performance. This is because it would reduce the available main memory for the FBL buffers (by reserving memory for the LBL buffers not flushed to disk) and, consequently, result to processing less time series during the subsequent Phase 1. We experimentally validated this argument; therefore, in the rest of this study, we do not report detailed results on this variation of the algorithm.

3.2 Node splitting policy

It is evident that the size of an indexing structure affects index creation time: a more compact structure translates to a smaller number of disk accesses.

Unlike other indexing structures, the *i*SAX index is not balanced. This was a design decision that led to a simple node splitting policy that does not take into account the data contained in the node to be split. In some cases, splitting a node may still result in all the time series ending up in one of the two new nodes, thus necessitating an additional split. This design decision may lead to a poor utilization of the leaf nodes and results in a larger and deeper index structure.

We propose a node splitting policy that makes informed decisions based on the knowledge of the distribution of the data stored in each node. The intuition behind this algorithm is the following. When splitting a node, we wish to distribute the time series in this node equally to the two new nodes. In order to do this exactly, we would have to examine all segments and for each segment, all possible cardinalities. This approach though would be prohibitively expensive. Our algorithm is instead examining for each segment the distributions of the highest cardinality symbols across the relevant time series. Then, it splits the node on the segment for which the distribution of the symbols indicates there is a high probability to divide the time series into the two new nodes, therefore avoiding the problem of useless node splits.

Consider the example depicted in Fig. 4, where we assume an *i*SAX word of length (i.e., number of segments) four, and we would like to split a node whose cardinality is 2 (for all segments). For each segment, we compute the $\mu \pm 3\sigma$ range of the corresponding symbols.

```

1  FBL[] // array of FBL buffers
2  LBL[] // array of LBL buffers
3  Function Bulk_Insert()
4  while ( more time series to index )
5      ts_new = next time series to be indexed
6      iSAX_word = iSAX representation of ts_new
7      if ( main memory still available )
8          if ( no FBL buffer contains iSAX_word )
9              create new FBL buffer corresponding to iSAX_word
10             add ts_new to FBL[]
11         else if ( main memory is full )
12             for each buf in FBL[]
13                 for each ts in buf
14                     call function Insert(ts)
15             flush LBL buffers created during insertion (corresponding to buf)
16             remove from memory those LBL buffers
-----
17  Function Insert(ts_new)
18      iSAX_word = iSAX representation of ts_new
19      if ( subtree corresponding to iSAX_word exists )
20          // current node has a child node to receive ts_new
21          n = destination node of ts_new // route ts_new down the tree
22          if ( n is leaf node )
23              if ( n not full ) // node does not need to be split
24                  add ts_new into LBL[n] // buffer corresponding to n
25              else // node n needs to be split
26                  for each ts in n
27                      // read all time series of n (from disk)
28                      add ts to LBL[n]
29                  n_new = new internal node
30                  for each ts in LBL[n]
31                      n_new.Insert( ts )
32                  n_new.Insert( ts_new )
33                  remove n // all time series moved under n_new
34              else if ( n is internal node )
35                  n.Insert( ts_new )
36          else // current node does not have a child node to receive ts_new
37              n_new_leaf = new leaf node
38              create new LBL buffer corresponding to n_new_leaf
39              add ts_new to this new LBL buffer

```

Fig. 3 Pseudocode for the bulk loading algorithm

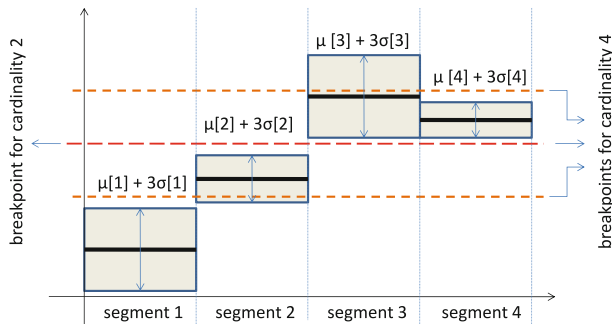


Fig. 4 Node splitting policy example

We observe that this range for segment 1 lies entirely below the lowest breakpoint of cardinality 4 (i.e., the cardinality of the two new nodes after the split). Only the ranges of segments 2 and 3 cross some breakpoint of cardinality 4. Between these two, the algorithm will pick to split on segment 3, because its μ value lies closer to a breakpoint than that of segment 2. This is an indication that with high probability, some of the time series in the node to be split will end up in the new node representing the area above the breakpoint, while the rest will move to the second new node, thus achieving a balanced split.

```

1  Function Split()
2  mean[] = ComputeSymbolMean() // using highest iSAX representation -
3  stdev[] = ComputeSymbolStDev() // already computed during insertions
4  segmentToSplit = none
5  for each segment  $s$  in the iSAX word
6       $b = \text{getBreakPoint}(s)$  // breakpoint of  $s$  with increased cardinality
7      if ( $b$  within  $\text{mean}[s] \pm 3\text{stdev}[s]$ )
8          // segment  $s$  is candidate for splitting
9          if ( $\text{mean}[s]$  closer to  $b$  than  $\text{segmentToSplit}$ )
10              $\text{segmentToSplit} = s$ 
11   $\text{segmentToSplit}.\text{IncreaseCardinality}()$ 

```

Fig. 5 Pseudocode for the node splitting algorithm

The pseudocode for the node splitting algorithm is shown in Fig. 5 (called every time we have to create new internal node: lines 29 and 37 in Fig. 3). The algorithm starts by computing for each segment the first two moments (mean μ and standard deviation σ) of the distribution of symbols over all the time series in the node to be split (lines 2–3). Note that this computation does not incur additional cost. Remember that the highest detail *iSAX* approximation of each time series is already stored along with the time series themselves, and we need to read those in order to materialize the node split.

Subsequently, the algorithm has to choose one of the segments for splitting the node. For each segment, the algorithm examines whether the range generated by $\mu \pm 3\sigma$ crosses any of the *iSAX* breakpoints of the immediately higher cardinality (lines 6–10). Among the segments for which this is true, the algorithm picks the one whose μ value lies closer to a breakpoint (lines 9–10).

4 Efficient handling of raw time series data

In the previous section, we propose mechanisms geared toward the efficient bulk loading of very large collections of time series. We note though that in the algorithms, we described there is no distinction between the approximations of the time series (i.e., their *iSAX* representation) and the raw time series (i.e., the detailed sequence of all the values of the time series).

The above observation is important for the following reason. Every time that we need to split a leaf node, we have to read the raw time series, as well as the approximations, for all the time series contained in this node from the disk. Then, we create the two new children nodes, and we proceed by inserting all these time series in the new nodes, which translates to writing back to disk the same raw time series and their approximations.

Bulk loading the index with a large collection of time series will lead to a large number of node splits (as the index grows in size), and for each one of these splits, we will have to pay the price of reading/writing the raw time series from/to disk multiple times. Evidently, this situation is suboptimal, causing a high number of unnecessary disk I/O.

The main idea for resolving this problem is that the raw time series data are not needed in order to build the *iSAX* index: the highest cardinality *iSAX* approximation of each time series is enough in order to build the index in its final form. Given that this approximation is one to two orders of magnitude smaller in size than the raw time series, the potential for further disk I/O savings is significant. Then, we just need an efficient way to store the raw time series in the correct leaf nodes of the index.

To this effect, we describe two extensions of the basic bulk loading algorithm proposed earlier. These extensions are able to reduce the number of unnecessary disk I/O related to the management of the raw time series data.

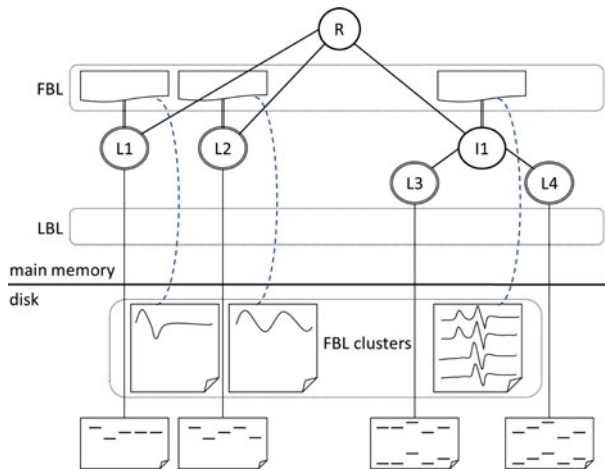


Fig. 6 The *iSAX* 2.0 Clustered index at the end of the first iteration. The FBL Clusters contain the raw time series data corresponding to each one of the FBL buffers, while the leaf node disk pages only contain the approximations of the time series in each leaf node

4.1 The *iSAX* 2.0 clustered index

As mentioned above, even if we just use the approximations in order to build the index, at the end, we still need to move the raw time series data to the disk pages of the corresponding leaf nodes (i.e., the leaf nodes to which these raw time series belong to).

The main challenge in this process is to minimize the random disk accesses, by making sure that the time series that end up in the same leaf node of the index are (temporarily) stored in the same (or contiguous) disk pages.¹

The first approach to address this problem, *iSAX 2.0 Clustered*, is a simple extension of the basic algorithm, which uses an extra set of disk pages for clustering together the raw time series that will potentially end up in the same leaf node. We call these disk pages the *FBL Clusters*.

The *iSAX* 2.0 Clustered approach works as follows. In each iteration, the first phase is the same as in *iSAX* 2.0 (Sect. 3): the raw time series and their approximations are stored in the FBL buffers. During the second phase, we propagate down to the *iSAX* index, to the LBL buffers, and to the leaf node disk pages *only* the approximations of the time series. The raw time series are flushed to disk directly from the FBL buffers to the FBL Clusters. This situation is depicted in Fig. 6.

We note that there is a one-to-one mapping between the FBL buffers and the FBL Clusters. This mapping remains stable throughout the bulk loading process (remember that the *iSAX* index is not balanced, and the children of the root node will never change). Therefore, in the subsequent iterations, the raw time series of the same FBL buffers will be flushed to the same FBL Clusters.

At the end of the procedure, when all the time series have been processed, we need to move the raw time series data from the FBL Clusters to the correct leaf node disk pages.

¹ This temporary storage on disk refers to storing the raw time series data for the period between the time when the time series is processed in order to be indexed, and the time when the raw time series has to be moved to the correct leaf node disk page.

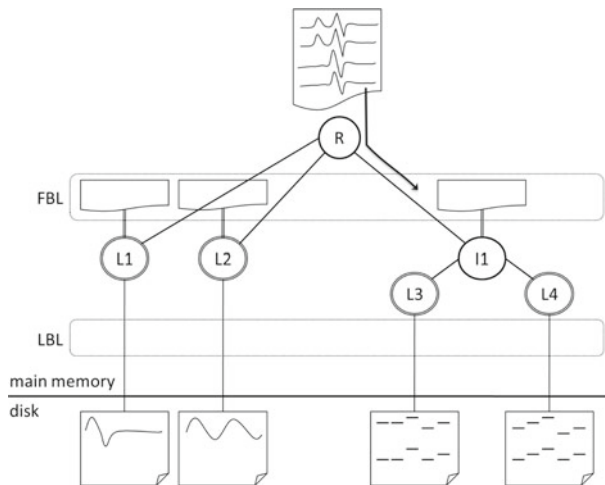


Fig. 7 Moving the raw time series data to their final position for *iSAX* 2.0 Clustered: the last FBL Cluster (corresponding to node *I1*) is being processed. The two leaf node disk pages on the left already contain the raw time series data, as well as their approximations. The two disk pages on the right only contain the approximations

This is done by reading the FBL Clusters one by one (each FBL Cluster may be several disk pages long) and then moving down the raw time series to the corresponding leaf nodes. All the necessary leaf nodes are already in place and no splits are needed: the *iSAX* tree has reached its final form (based on the approximations).

During this process, we once again make use of the LBL buffers in order to ensure that most of the disk I/O involves sequential accesses. Figure 7 shows the situation where we have already processed the FBL Clusters corresponding to the first two FBL buffers, *L1* and *L2*, and we are ready to process the last FBL Cluster, corresponding to node *I1*.

4.2 The *iSax2+* index

The *iSAX* 2.0 Clustered approach that we presented above is expected to lead to a better time performance when compared to *iSAX* 2.0 since it does not require to read from and write to disk the raw time series data every time that a leaf node splits. Nevertheless, we still have to do a fair amount of work for processing the raw time series: we need to read from disk the raw time series for the first time in order to compute the approximation and start the insertion process to the index; we need to store the raw time series data on disk, in one of the FBL Clusters; we need to read the same data from disk after the index-growing phase is complete; and last, we need to write the raw time series data back to disk in the correct leaf node disk page. The question now becomes whether we can avoid some of this work.

The *iSAX2+* approach is giving a positive answer to this question, by taking advantage of the following observation: during the bulk loading process, several time series end up in leaf nodes that will not split till the end of the process. This means that for these time series, there is no need to touch them again (read from, or write to disk), thus saving a considerable number of disk I/O that *iSAX* 2.0 Clustered does not avoid.

Intuitively, the above observation on which *iSAX2+* is based is true, because the *iSAX* index is not balanced and its root has a high fanout. This makes the index grow quite fast in

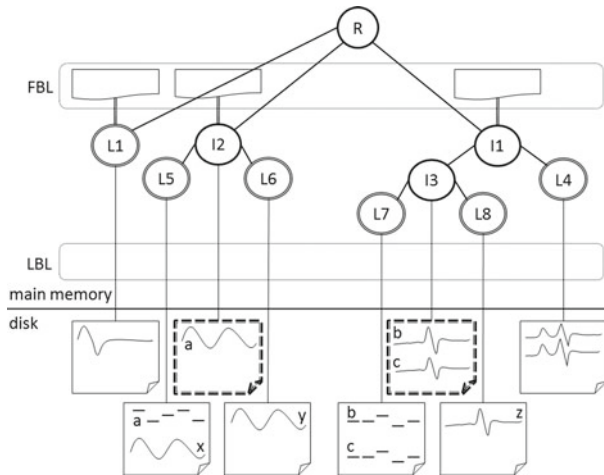


Fig. 8 Result of inserting time series x , y , and z to the index shown in Fig. 2 (bottom), according to *iSAX2+*. In addition to leaf nodes, internal nodes as well point to pages on disk (shown in **bold, dotted lines**) that contain raw time series data

the beginning of the bulk loading process, with its shape stabilizing well before the end of the process. Indicatively, in our experiment with one billion time series, the index became almost stable after having inserted a bit more than 500 million time series: after that point, only a small percentage of the leaf nodes were required to split.

In *iSAX2+*, the first iteration of a bulk loading process is exactly the same as *iSAX 2.0*: raw time series together with their approximations are first inserted in the FBL buffers, then moved to the LBL buffers, and finally flushed to disk. Note that it is at the end of the first iteration that we write to disk for the first time. From the second iteration onwards, *iSAX2+* differentiates itself only in the way that it handles leaf node splits. In particular, it implements the following algorithm. When a leaf node needs to be split, *iSAX2+* reads from the corresponding leaf node disk page only the approximations of the time series contained in that leaf node. It creates the two children nodes and inserts in them the approximations of the time series coming from the parent (i.e., the old leaf that was split), as well as the raw time series data and their approximations coming from the LBL buffer (i.e., the new time series that were processed in the current iteration of the bulk loading process). The raw time series data of the old leaf remain in the same disk page on disk, pointed at by a node that has now become an internal node. This is a major difference of *iSAX2+* to the previous approaches, where both internal nodes and leaf nodes may point to disk pages that contain raw time series data.

Figure 8 depicts the result of inserting (during the second iteration) time series x , y , and z to the state of the index shown in Fig. 2 (bottom). In the latter figure, node $L2$ contains time series a , and node $L3$ contains time series b and c . Assume that when x and y arrive, we need to split $L2$. The result (refer to Fig. 8) is the creation of two new leaf nodes, $L5$ and $L6$, and the conversion of $L2$ into the internal node $I2$. Assume that $L5$ receives two of the time series, a and x : only the approximation of a gets stored in $L5$, while both the raw time series x and its approximation (omitted from the figure for simplicity) are stored in $L5$. Even though $I2$ is an internal node, it points to a page on disk containing the raw time series data for a . Similarly, after inserting time series z , leaf node $L3$ becomes internal node $I3$ with children $L7$ and $L8$. $I3$ points to a disk page containing the raw time series data for b and c (i.e., the two time series that $L3$ used to contain). $L7$ contains the approximations

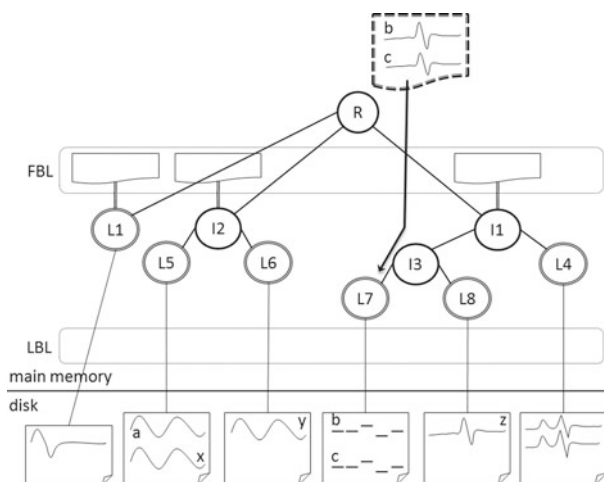


Fig. 9 During the final step of *iSAX2+*, the raw time series in the disk pages pointed to by internal nodes are read in main memory and moved to the correct leaf node in the index

of *b* and *c*, and *L8* contains the raw time series and the approximation (omitted from the figure for simplicity) of *z*.

Following the process described above, we end up with an index, where raw time series data are scattered among leaf nodes, as well as several internal nodes. Therefore, when all the time series have been processed, we need to move the raw time series data from all internal nodes to the correct leaf node disk pages. In order to achieve this, *iSAX2+* traverses the index in depth-first order, reads in memory all the raw time series data pointed to by internal nodes, and flushes them to disk (making use of the LBL buffers) in the corresponding leaf nodes (shown in Fig. 9). This step is fast, because it involves moving data down only a particular subtree of the index. Note that during this step of the algorithm, there is no node splitting; all the required splits have already taken place, using the approximations of the time series.

5 Experimental evaluation

We have designed all experiments such that they are reproducible. To this end, we have built a webpage which contains *all* datasets used in this work, together with spreadsheets that contain the raw numbers displayed in all the figures [14].

Experimental evaluation was conducted on an Intel Xeon E5504 with 24 GB of main memory, 2TB Seagate Barracuda LP hard disk, running Windows Vista Business SP2. All code is in C#.NET 3.5 Framework. For the case study in Sect. 5.2, we used an AMD Athlon 64 X2 5600+ with 3 GB of memory, 400 GB Seagate Barracuda 7200.10 hard disk, and running Windows XP SP2 (with /3 GB switch).

Our experiments are divided into three sections: A) tests that measure the classic metrics of disk accesses, wall clock time, index size, sensitivity to parameters, etc. B) a detailed case study of a deployed use of our system in an important entomology problem, and C) examples of higher level data mining algorithms built using our index as a subroutine.

The algorithms that we evaluate are *iSAX 2.0*, and the original *iSAX*, where all the available main memory is used for disk buffer management (i.e., buffers corresponding to

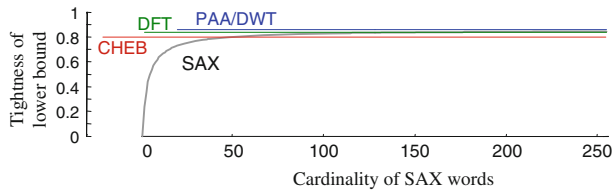


Fig. 10 A comparison of the tightness of lower bound for various time series representations. All approaches except SAX use a constant eight bytes and therefore have a constant tightness of lower bound. The results for SAX show the effect of varying the cardinality from 2 to 256 (and hence the number of bits from 2 to 8)

the leaf level nodes). We also compare to *iSAX-BufferTree*, which is an adaptation of the *Buffered R-Tree* bulk loading algorithm [28]. In this case, instead of having buffers only at the first and leaf levels, we also have some buffers at intermediate levels of the index tree. These buffers are of equal size, which depends on the size of the index (i.e., the buffer size decreases as the index grows). An important distinction of *iSAX* 2.0 is that it is the only bulk loading strategy of the three specifically designed for a non-balanced index tree. It adaptively resizes (FBL) and positions (LBL) its memory buffers according to the needs of the incoming time series. The experiments demonstrate that this choice leads to significant savings in terms of disk page accesses.

We further make the following observations. First, most of the previous comparisons of indexing methods for time series simply reduce to claims about the relative merits of a time series representation method, that is, DWT versus DFT methods. However, there is an increasing understanding that this is a red-herring. It has been forcedly shown that averaged over many datasets, the time series representation makes very little difference [11, 15, 31, 35].

Second, a unique property of *iSAX* is its tiny bit-aware index size. This means that an *iSAX* index is very small compared to the data it indexes, and thus we can fit the entire index in main memory even for the massive datasets we wish to consider. In order to compare to other methods, we have to consider the case of what to do when the index itself is mostly disk resident, and in virtually every case the original authors provide no guidance. For completeness, we show that using the *iSAX* representation, we obtain the same benefit as other methods (in terms of tightness of lower bounds), at a fraction of the space cost.

We can measure the tightness of the lower bounds, which is defined as the lower bounding distance over the true distance. Figure 10 shows this for random walk time series of length 256, with eight PAA/DWT coefficients, eight DFT coefficients (using the complex conjugate property), eight Chebyshev (CHEB) polynomials coefficients, and a SAX representation also of length eight. We varied the cardinality of SAX from 2 to 256, whereas the other methods use a constant eight bytes per coefficient, and thus have a constant value for tightness of lower bounds in this experiment. We averaged results over 1,000 random pairs of time series. The results suggest that there is little reason to choose between PAA/DWT/DFT/CHEB, as has been noted elsewhere [15]. They also show that once the cardinality of *iSAX* is >50 , it is competitive with the other methods, even though it requires only one-eighth of the space (one byte per coefficient versus eight bytes per coefficient for PAA/DWT/DFT/CHEB).

5.1 Scalability of *iSAX* 2.0

In this section, we present experimental results on the scalability of *iSAX* 2.0. In particular, we evaluate the effect of the proposed node splitting and bulk loading algorithms on the time to build and the size of the index.

5.1.1 Splitting policy evaluation

We ran experiments in order to evaluate the new splitting policy implemented in *iSAX* 2.0. In these experiments, we compare our results against those obtained by the use of the *iSAX* splitting policy. We generated datasets of sizes 1–100 million time series, where each time series has length of 256, generated as follows. In order to generate the series, we use a standard normal distribution $N(0,1)$, where each point in the time series is generated as $x_{i+1} = N(x_i, 1)$. We report the averages over these 10 runs (their variance was 5 % or less for all our experiments).

Even if the datasets used in this section are smaller than the other used in this paper, the results follow the same trends. We note that we obtain similar results to the ones presented below when instead of varying the number of time series, we vary the threshold *th*. We omit these results for brevity. (All experiments in this section were run using the proposed bulk loading algorithm, as well. Though, this fact does not affect the interpretation of the results.)

Index Size: In the first set of experiments, we measure the total size of the index (in number of nodes), after having indexed the entire dataset. Figure 11 (bottom) shows that there is a quadratic dependency between the number of nodes in the index and the number of time series indexed.

The results show that the new splitting policy implemented in *iSAX* 2.0 can effectively reduce the number of nodes required by the index. On average, *iSAX* 2.0 needs 34 % less nodes than the *iSAX* index. These results validate our premise that using the first moments of the distributions of the *iSAX* symbols is a simple, yet effective mechanism for identifying suitable split segments.

The results also demonstrate the significant impact that the leaf node capacity has on the index. Evidently, when this capacity is decreased, the index needs to grow many more internal and leaf nodes in order to accommodate the time series to be indexed.

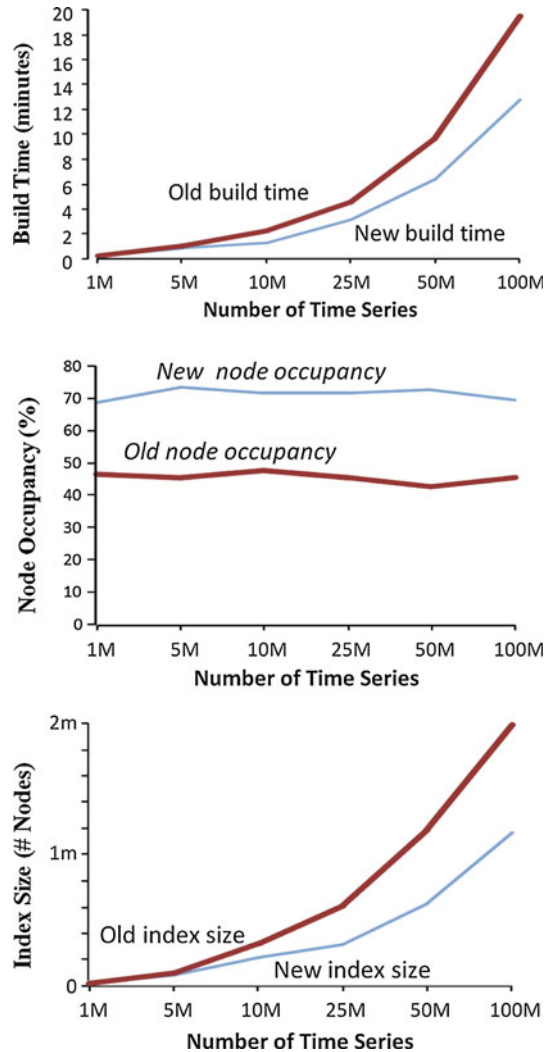
Index Build Time: In the next set of experiments, we measure the time needed to build the index as a function of the number of time series [Fig. 11 (top)]. We observe that the curves in the graph follow the same trends as before, with the time to build the index increasing quadratically.

This result is not surprising, since the build time is strongly correlated with the number of nodes of the index. Once again, we observe the benefit of the proposed node splitting algorithm, which leads to an average reduction of 30 % in the index built time. Therefore, maintaining a small index size is highly desirable.

Leaf Node Utilization: We now investigate the average utilization (or occupancy) of the leaf nodes. A bad splitting policy that does not take into account information on the data contained in the nodes to be split can generate unbalanced splits, leading to low usage of the leaf nodes and to long insertion times. Remember that having many leaf nodes with low utilization translates to the need for an increased number of leaf nodes (in order to accommodate the same number of time series) and, consequently, for an increased number of disk page accesses.

The graph of Fig. 11 (middle) shows that the new splitting algorithm results in leaf nodes with an average of 54 % more occupancy than the old splitting algorithm, underlining the effectiveness of the proposed policy. The experiments also show that there is no variability in the leaf node utilization as we vary the number of time series in the index.

Fig. 11 Splitting policy comparison between *i*SAX (old) and *i*SAX 2.0 (new) when varying the size of the dataset: construction time (*top*), leaf node occupancy (*middle*), and number of nodes (*bottom*)



5.1.2 Bulk loading evaluation

In order to test the proposed approach, we index a set of datasets with sizes from 100 million to 1 billion time series composed by random walks of length 256. Each data point in the time series is produced as $x_{i+1} = N(x_i, 1)$, where $N(0, 1)$ is a standard normal distribution. We use a leaf node threshold $th = 8,000$ and wordlength $w = 8$. We compare the obtained results with the performance of the *i*SAX index.

Index Build Time: The first experiment shows the time needed to build the index for the two different methods [see Fig. 12 (top)]. The results demonstrate the scalability of *i*SAX 2.0 as the dataset size increases, with a trend that is almost linear. In contrast, the time to build the *i*SAX index grows much faster and very quickly becomes prohibitively expensive. It took 12 and 20 days to index the datasets of size 400 and 500 million time series, respectively. At that point, though, we were forced to discontinue the experiments with *i*SAX. We estimated that

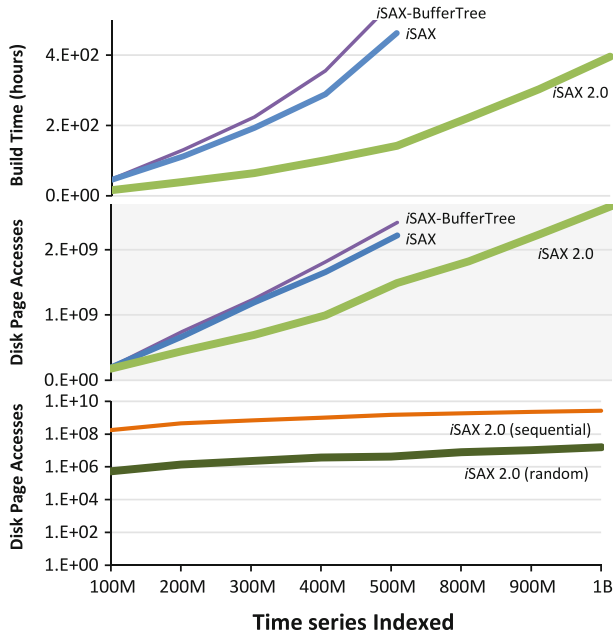


Fig. 12 Index metrics as a function of dataset sizes. (*top*) Time to build an index. (*middle*) Disk page accesses between indexing methods. (*bottom*) Distribution of sequential and random disk page accesses for *iSAX* 2.0

it would take around 56 days to index 1 billion time series. The *iSAX*-BufferTree algorithm initially performs better than *iSAX*, but its performance deteriorates as the size of the dataset increases.

The problem with the above two strategies is that they cannot effectively concentrate the available memory resources in the areas of the index that are most needed. Instead, they allocate memory in a more balanced way across the index, which does not result in the best performance since in our case the index is not a balanced tree.

Using the proposed bulk loading algorithm, *iSAX* 2.0 manages to index a dataset with 100 million time series in just 16 h. The one billion time series dataset is indexed in <400 h (about 16 days), which corresponds to an indexing time of 1 ms/time series.

Disk Page Accesses: In Fig. 12 (middle), we show the number of disk page accesses performed by the three indexing methods during the same experiments.

The graph shows that when using the bulk loading algorithm, we need to access the disk only half the times as before. This is already a significant improvement in the performance of the algorithm. Though, if we take a closer look at the experimental results, we make another very interesting observation [refer to Fig. 12 (bottom)]. More than 99.5 % of the disk page accesses that *iSAX* 2.0 has to perform are sequential accesses, which means that random accesses are consistently two orders of magnitude less than the number of sequential accesses. In contrast, most of the disk accesses that the *iSAX* and *iSAX*-BufferTree strategies perform are much more expensive random accesses (since they involve the flushing of buffers corresponding to different nodes of the index), leading to an index build time that is an order of magnitude larger than that of *iSAX* 2.0.

***iSAX* 2.0 Clustered and *iSAX*2+:** In Fig. 13, we show the same results for *iSAX* 2.0 Clustered and *iSAX*2+. When compared to *iSAX* 2.0, *iSAX* 2.0 Clustered exhibits an improvement of 30 % in the time to build the index and *iSAX*2+ an improvement of 40 %

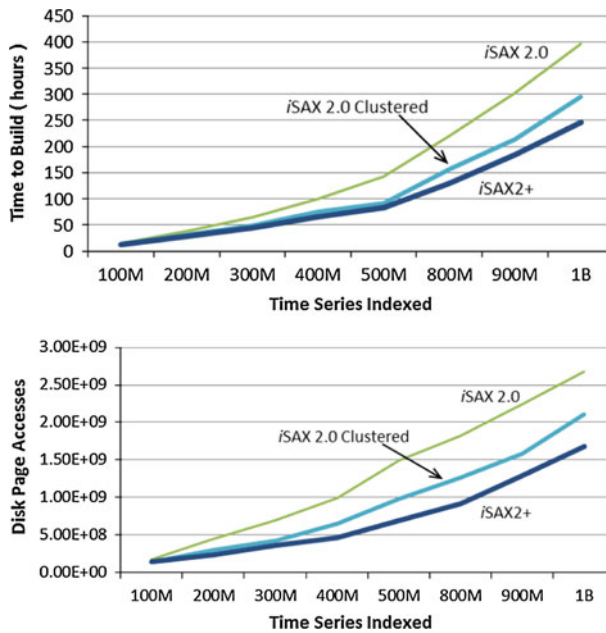


Fig. 13 Index metrics as a function of dataset sizes for *iSAX 2.0 Clustered* and *iSAX2+*. (top) Time to build an index. (bottom) Disk page accesses between indexing methods

[Fig. 13 (top)]. *iSAX2+* is able to index one billion time series in just 10 days, in 82 % less time than *iSAX*. These trends are explained by the number of disk page accesses required by *iSAX 2.0 Clustered* and *iSAX2+*, which are less than the disk page accesses of *iSAX 2.0* by 32 % and 47 % (on the average), respectively. When compared to *iSAX*, *iSAX2+* achieves a reduction of around 74 %.

These results show that the bulk loading algorithm is extremely effective in reducing the I/O cost, thus enabling *iSAX2+* to index 1,000,000,000 time series.

5.2 A case study in entomology

Many insects such as aphids, thrips, and leafhoppers feed on plants by puncturing their membranes and sucking up the contents. This behavior can spread disease from plant to plant causing discoloration, deformities, and reduced marketability of the crop. It is difficult to overstate the damage these insects can do. For example, just one of the many hundreds of species of Cicadellidae (Commonly known as Sharpshooters or Leafhoppers), *Homalodisca coagulata* first appeared in California around 1993 and has since done several billions of dollars of damage and now threatens California's \$34 billion dollar grape industry [4].

In order to understand and ultimately control these harmful behaviors, entomologists glue a thin wire to the insect's back and then measure fluctuations in voltage level to create an Electrical Penetration Graph (EPG). Figure 14 (top) shows the basic setup.

This simple apparatus has allowed entomologists to make significant progress on the problem. As USDA scientist Dr. Elaine Backus recently noted, "Much of what is known today about hemipteran feeding biology ... has been learned via use of EPG technology" [6].

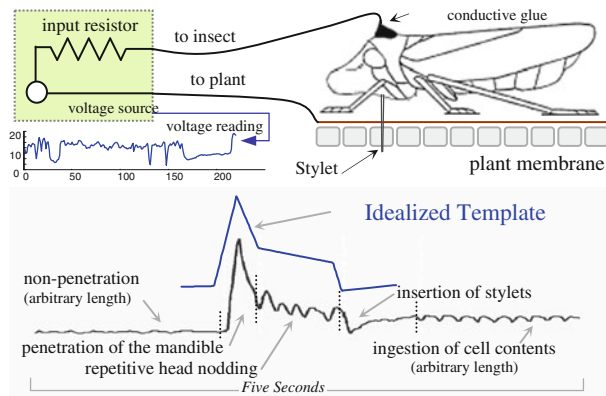


Fig. 14 (top) A schematic diagram showing an EPG apparatus used to record insect behavior. (bottom) An EPG insect behavior derived from subset of Fig. 2 from [19]. An idealized version of the observed behavior created by us is shown with a **bold blue line** (color figure online)

However, in spite of the current successes, there is a bottleneck in progress due to the huge volumes of data produced. For example, a single experiment can last up to 24 h. At 100 Hz that will produce a time series with approximately eight-million data points. Entomologists frequently need to search massive archives for known patterns to confirm/refute hypotheses. For example, a recent paper asks whether the chemical thiamethoxam causes a reduction in xylem² feeding behavior by a Bird Cherry-Oat Aphid (*Rhopalosiphum padi*). The obvious way to test such a hypothesis is to collect EPG data of both a treatment group and a control group and search for occurrences of the (well known) xylem feeding pattern.

Recently, the Entomology Department at UC Riverside asked us to create an efficient tool for mining massive EPG collections [17]. We have used the techniques introduced in this work as a beta version of such a tool, which will eventually be made freely available to the entomological community. Let us consider a typical scenario in which the tool may be used. In Fig. 14 (bottom), we see a copy of Fig. 2 from [19]. This time series shows a behavior observed in a Western Flower Thrip (*Frankliniella occidentalis*), an insect which is a vector for more than 20 plant diseases. The Beet Leafhopper (*Circulifer tenellus*) is not particularly closely related to thrips, but it also feeds on plants by puncturing their membranes and sucking sap. Does the Beet Leafhopper exhibit similar behavior?

To answer this question, we indexed 20,005,622 subsequences of length 176 from the Beet Leafhopper EPG data, which had been collected in 60 individual experiments conducted from 2007 to 2009. We used a th size of 2,000 and w of eight to construct an index on our AMD machine. Even with fewer resources, it took only 5.1 h to build the index, which occupied a total of 26.6 gigabyte of disk space. As shown in Fig. 14 (bottom), we used the simple idealized version as a query to our database. Figure 15 (left) shows the result of an approximate search, which takes <0.5 s to answer.

This result suggests that although the insect species is different (recall we queried a Thrip behavior on Beet Leafhopper database), the behaviors are similar, differing only in the insertion of stylet behavior. As a sanity check, we also queried the database with an idealized version of a Beet Leafhopper behavior, the so-called Waveform A; in this case, Fig. 15 (right) shows that the match is much closer.

² Xylem is plant sap responsible for the transport of water and soluble mineral nutrients from the roots throughout the plant.

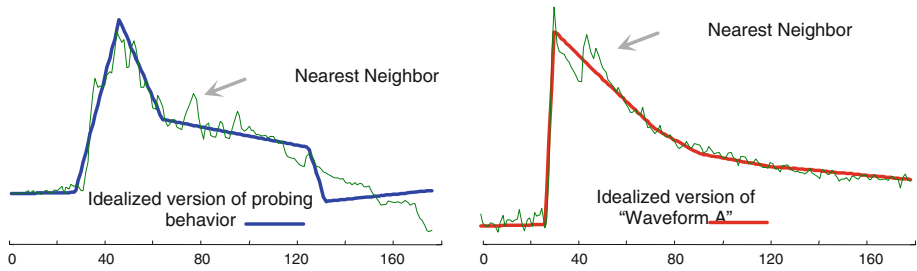


Fig. 15 Query time series and its approximate nearest neighbor

Table 2 An algorithm for converting DNA to time series

```

 $T_1 = 0;$ 
For  $i = 1$  to length (DNAstring)
    If DNAstring $_i = \mathbf{A}$ , then  $T_{i+1} = T_i + 2$ 
    If DNAstring $_i = \mathbf{G}$ , then  $T_{i+1} = T_i + 1$ 
    If DNAstring $_i = \mathbf{C}$ , then  $T_{i+1} = T_i - 1$ 
    If DNAstring $_i = \mathbf{T}$ , then  $T_{i+1} = T_i - 2$ 
End

```

5.3 Mining massive DNA sequences

The DNA of the Rhesus Macaque (*Macaca mulatta*) genome consists of nearly three billion base pairs (approximately 550,000 pages of text if written out in the format of this paper), beginning with **TAACCCTAACCCTAA...** We converted this sequence into a time series using the simple algorithm shown in Table 2.

Figure 16 (left) shows an example of the time series created from the DNA of monkey chromosome 3, together with the human chromosome 21. Note that they are not globally similar, but a *subsection* of each is locally similar if we flip the direction of one sequence. This figure suggests what is already known: the most recent common ancestor of the macaque and humans lived *only* about 25 million years ago, so we expect their DNA to be relatively similar. However, since humans have 23 chromosomes and the monkey has only 21, the mapping of chromosomes cannot be one to one; some chromosomes must be mapped in a jigsaw fashion. But what is the mapping?

To answer this question, we indexed the *entire* time series corresponding to the macaque DNA (non-sex related). We used a subsequence length of 16,000, down-sampled by a factor of 25 to mitigate “noise.” We then used a sliding window with a step size of five to extract a total of 21,612,319 subsequences. To index, we used a *th* size of 1,000 and *w* of 10. In total, it took 6.75 h to build the index.

We obtained queries from the human genome in the same manner and queried with both the original and transposed versions. For each human chromosome, we issued an average of 674 approximate searches (recall that chromosomes have differing lengths) and recorded the 10 nearest neighbors. In Fig. 16 (right), we summarize *where* the top ten neighbors are found, by creating a grid and coloring the cell with an appropriate shade of gray. For example, a pure white cell at location $\{i, j\}$ means that *no* query from human chromosome i mapped to monkey chromosome j and a pure black cell at location $\{i, j\}$ means that *all* 10 queries from

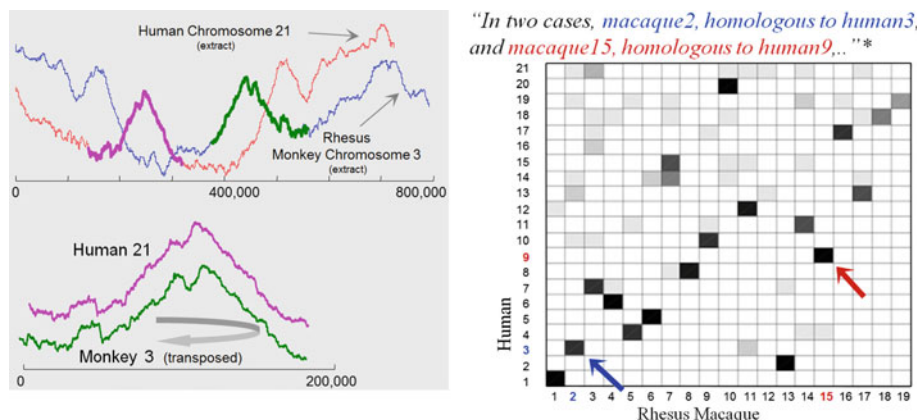


Fig. 16 (left) An example of DNA converted into time series. (right) The cells represent potential mappings between the Macaque and Human Genomes. The darker the cell, the more often the nearest neighbor of a time series taken from a particular human chromosome had a nearest neighbor from a particular Macaque chromosome. *Quote from *An initial genetic linkage map of the rhesus macaque* [20]. The smallest chromosomes including the sex chromosomes are omitted

human chromosome^{*i*} mapped to monkey chromosome^{*j*}. This figure has some unambiguously dark cells, telling us for example that Human 2 is homologous (“equivalent”) to Macaque 3. In addition, in some cases, the cells in the figure suggest that two human chromosomes may match to a single Macaque chromosome. For example, in the column corresponding to Macaque 7, the two darkest cells are rows 14 and 15.

The first paper to publish a genetic linkage map of the two primates tells us “macaque7 is homologous to human14 and human15” [20]. More generally, this correspondence matrix is at least 95 % in agreement with the current agreement on homology between these two primates [20].

Figure 17 is a graphical representation of the chromosome mappings discovered using *iSAX* (left), and those that have been experimentally verified by biologists up to now (right). The figure shows that the true mappings are a subset of those indicated by *iSAX*, which demonstrates the potential utility of the proposed approach.

This experiment demonstrates that we can easily index tens of millions of subsequences in less than a day, answer 13,480 queries in 2.5 h, and produce objectively correct results that can help scientists make use of the data collections they already have.

5.4 Mining massive image collections

While there are hundreds of possible distance measures proposed for images, a recent paper has shown that simple Euclidean distance between color histograms is *very* effective if the training dataset is *very* large [22]. More generally, there is an increasing understanding that having lots of data without a model can often beat smaller datasets, even if they are accompanied by a sophisticated model [3, 18]. Indeed, Peter Norvig, Google’s research director, recently noted that “All models are wrong, and increasingly you can succeed without them.” The ideas introduced in this work offer us a chance to test this theory.

We indexed the color histograms of the famous MIT collection of 80 million low-resolution images [22]. As shown in Fig. 18, these color histograms can be considered pseudo “time series.” At indexing time, we omitted very simple images (e.g., those that are comprised of

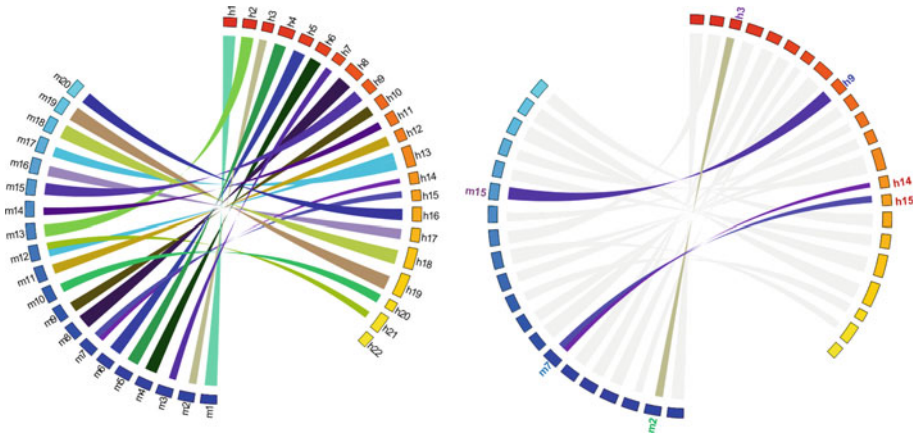


Fig. 17 Illustration of candidate chromosome mappings discovered using our approach (*left*), and mappings verified experimentally by biologists (*right*). The verified mappings are a subset of those discovered using *iSAX*

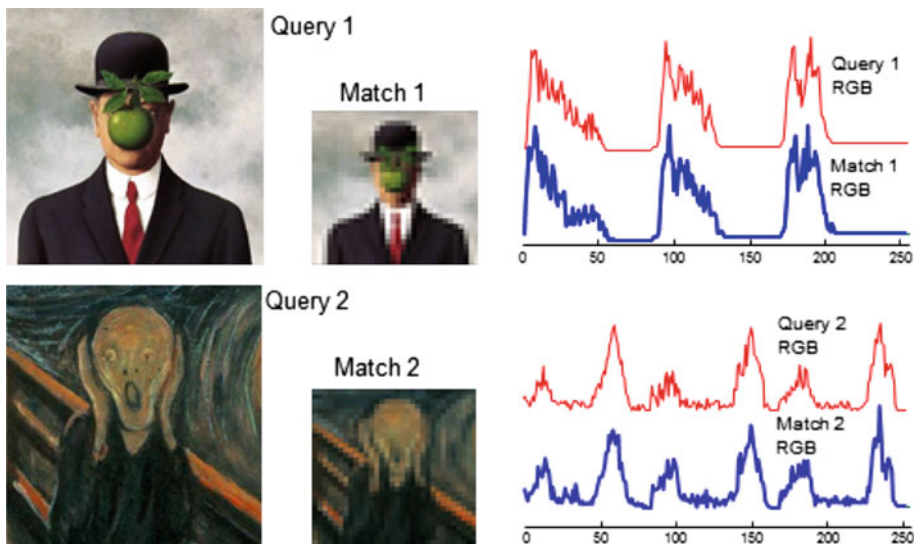


Fig. 18 (*left*) A detail of *The Son of Man* by René Magritte, which we used as a query to our index, finding “Match 1.” (*right*) A detail of *The Scream* by Edvard Munch, which we used as a query that returned “Match 2.” The *insets* show the similarity of the images in RGB histogram space

only one or two colors, etc.). In total, our index contains the color histograms of 69,161,598 images.

We made color histograms of length 256 and used a *th* size of 2,000 and *w* of 8. It took 9.8 h to build the index, which is inconsequential compared to the 9 months of 24 h a day crawling it took to collect it [22]. The data occupy a total of 133 GB of disk space. The latter figure only includes the space for the time series, the images themselves required an extra of 227 GB.

Does this random sampling of the web's images contain examples of iconic art images? To test this, we found examples of two famous images using Google image search and converted the image to color histograms of length 256. We then used these to search our collection with an approximate search. Each search took less than a second, and the results can be seen in Fig. 18. Note that we are not claiming that Euclidean distance between color histograms is the best measure for image similarity. This experiment simply demonstrates the scalability and generality of our ideas, as a side effect of demonstrating the *unreasonable effectiveness of* (massive amounts of) *data 0*.

6 Related work

There is an extensive literature on time series indexing; the interested reader should refer to [5, 15, 21] and the references therein for useful surveys and empirical comparisons. There are at least a dozen well-known methods for approximation (i.e., dimensionality reduction) of time series data, which have been used for a variety of applications, such as indexing motions [25], amnesic representations [31], streaming time series pattern matching [30], uncertain time series matching [13], and others. These approximation methods include the Discrete Fourier Transformation [16], Singular Value Decomposition (SVD), Discrete Cosine Transformation, Discrete Wavelet Transformation [32], Piecewise Aggregate Approximation [24], Adaptive Piecewise Constant Approximation [23], Chebyshev polynomials [7]. However, recent extensive empirical evaluations suggest that on average, there is little to differentiate between these representations in terms of fidelity of approximation, and thus indexing power [15, 31].

The approximation we use in this work is intrinsically different from the techniques listed above; in that, it is discrete [29], rather than real-valued. This discreteness is advantageous; in that, the average byte used by discrete representations carries much more information than its real-valued counterparts. This allows our index to have a much smaller memory footprint, and it allows us to explore novel, simple, and effective splitting strategies that exploit the discrete nature of the representation.

Several works have studied the problem of indexing time series datasets [7, 5, 16, 21, 24, 26]. This is an important problem and serves as the step-stone for enabling the analysis and mining of time series collections at a large scale. In this work, we are dealing with a complementary problem: we are not proposing a new time series index, but rather a novel technique that allows us to build a time series index fast.

The problem of bulk loading has been studied in the context of traditional database indices, such as B-trees and R-trees, and other multi-dimensional index structures [2, 12, 28, 33, 36, 37]. For these structures, two main approaches have been proposed. First, we have the merge-based techniques [12]) that preprocess data into clusters. For each cluster, they proceed with the creation of a small tree that is finally merged into the overall index. It is not clear how such techniques could be applied in our problem setting, since the mere task of clustering datasets of such scale would incur a cost higher than indexing, therefore rendering the merge-based techniques inapplicable.

Second, there are the buffering-based techniques [28, 33, 36] that use main memory buffers to group and route similar time series together down the tree, performing the insertion in a lazy manner. These techniques are not directly applicable in our setting, since they have been designed to improve the bulk loading performance of *balanced* index structures (as shown in our experiments for an adaptation of [28]). Another interesting technique would be the

two-step approach of the path-based method [36]). But this one is not applicable either, because it *requires* the existence of a balanced index tree in order to produce correct results.

Finally, we note that no previous work has explicitly studied the problem of bulk loading in the context of an index for time series data.

7 Conclusions

We describe *i*SAX 2.0, an index structure specifically designed for ultra-large collections of time series, and propose new mechanisms and algorithms for efficient bulk loading and node splitting. Furthermore, we propose two extensions of the above approach, namely *i*SAX 2.0 Clustered and *i*SAX2+. These extensions focus on the efficient handling of the raw time series data during the bulk loading process.

We experimentally validate the proposed algorithms, including the first published experiments to consider datasets of size up to one billion time series, showing that we can deliver a significant improvement in the time required to build the index.

An obvious extension of this work is to consider I/O parallelism (e.g., each subtree of the root node could be processed in parallel). An in-depth study of this direction is part of our future work.

Acknowledgments This research was funded by NSF awards 0803410 and 0808770.

References

- ADHD (2012) http://www.fcon_1000.projects.nitrc.org/indi/adhd200/
- An N, Kanth R, Kothuri V, Ravada S (2003) Improving performance with bulk-inserts in Oracle R-trees. VLDB, pp 948–951
- Anderson C (2008) The end of theory: the data deluge makes the scientific method obsolete. Wired 16(7). http://www.wired.com/science/discoveries/magazine/16-07/pb_theory
- Andersen P, Brodbeck B, Mizell R (2009) Assimilation efficiency of free and protein amino acids by *Homalodisca vitripennis* (Hemiptera: Cicadellidae: Cicadellinae) feeding on *Citrus sinensis* and *Vitis vinifera*. In: Andersen PC, Brodbeck BV, Mizell RF (eds) Florida entomologist, vol. 92, issue 1, pp 116–122.
- Assent I, Krieger R, Afschari F, Seidl T (2008) The TS-tree: efficient time series search and retrieval. In: Proceedings of the 11th international conference on extending database technology: advances in database technology (EDBT '08). ACM, New York, NY, pp 252–263
- Backus E, Bennett W (2009) The AC-DC correlation monitor: new EPG design with flexible input resistors to detect both R and emf components for any piercing-sucking hemipteran. J Insect Physiol 55(10):869–884
- Cai Y, Ng R (2004) Indexing spatio-temporal trajectories with Chebyshev polynomials. In: Proc SIGMOD
- Camerra A, Palpanas T, Shieh J, Keogh EJ (2010) *i*SAX 2.0 (2010). Indexing and mining one billion time series. ICDM, pp 58–67
- Castro N, Azevedo PJ (2010) Multiresolution motif discovery in time series. SDM, pp 665–667
- Castro N, Azevedo PJ (2011) Time series motifs statistical significance. Proceedings of the eleventh SIAM international conference on data mining
- Chakrabarti K, Keogh EJ, Mehrotra S, Pazzani MJ (2002) Locally adaptive dimensionality reduction for indexing large time series databases. ACM Trans Database Syst 27(2):188–228
- Choubey R, Chen L, Rundensteiner EA (1999) GBI: a generalized R-tree bulk-insertion strategy. SSD, pp 91–108
- Dallachiesa M, Nushi B, Mirylenka K, Palpanas T (2012) Uncertain time-series similarity: return to the basics. Proc VLDB Endow (PVLDB) J 5(11):1662–1673
- Data (2012) <http://www.disi.unitn.it/themis/isax2plus/>
- Ding H, Trajcevski G, Scheuermann P, Wang X, Keogh E (2008) Querying and mining of time series data: experimental comparison of representations and distance measures. PVLDB 1(2):1542–1552

16. Faloutsos C, Ranganathan M, Manolopoulos Y (1994) Fast subsequence matching in time-series databases. In: Snodgrass RT, Winslett M (eds) Proceedings of the 1994 ACM SIGMOD international conference on management of data (SIGMOD '94). ACM, New York, NY, pp 419–429
17. Greg W (2009) Personal communication. August 12th
18. Halevy A, Norvig P, Pereira F (2009) The unreasonable effectiveness of data. *IEEE Intell Syst* 24(2):8–12
19. Kindt F, Joosten NN, Peters D, Tjallingii WF (2003) Characterisation of the feeding behaviour of western flower thrips in terms of EPG waveforms. *J Insect Physiol* 49:183–191
20. Rogers J et al (2006) An initial genetic linkage map of the rhesus macaque (*Macaca mulatta*) genome using human microsatellite loci. *Genomics* 87:30–38
21. Shieh J, Keogh E (2008) iSAX: indexing and mining terabyte sized time series. In: ACM SIGKDD
22. Torralba A, Fergus R, Freeman WT (2008) 80 million tiny images: a large data set for nonparametric object and scene recognition. *IEEE PAMI* 30(11):1958–1970
23. Keogh EJ, Chakrabarti K, Mehrotra S, Pazzani MJ (2001a) Locally adaptive dimensionality reduction for indexing large time series databases. In: SIGMOD
24. Keogh EJ, Chakrabarti K, Pazzani MJ, Mehrotra S (2001b) Dimensionality reduction for fast similarity search in large time series databases. *Knowl Inf Syst* 3(3):263–286
25. Keogh E, Palpanas T, Jordan VB, Gunopulos D, Cardle M (2004) Indexing large human-motion databases. VLDB, Toronto, ON, Canada, August
26. Keogh EJ, Smyth P (1997) A probabilistic approach to fast pattern matching in time series databases. In: Proceedings of the third international conference on knowledge discovery and data mining (KDD-97), Newport Beach, California, pp 24–30
27. Kohlsdorf D, Starner T, Ashbrook D (2011) MAGIC 2.0: a web tool for false positive prediction and prevention for gesture recognition systems. In: FG' 11
28. Lars A, Klaus H, Vahrenhold J (2002) Efficient bulk operations on dynamic R-trees. *Algorithmica* 33(1):104–128
29. Lin J, Keogh EJ, Wei L, Lonardi S (2007) Experiencing SAX: a novel symbolic representation of time series. *Data Min Knowl Discov* 15(2):107–144
30. Marascu A, Khan SA, Palpanas T (2012) Scalable similarity matching in streaming time series. In: Pacific-Asia conference on knowledge discovery and data mining (PAKDD), Kuala Lumpur, Malaysia, May
31. Palpanas T, Vlachos M, Keogh EJ, Gunopulos D (2008) Streaming time series summarization using user-defined amnesic functions. *IEEE Trans Knowl Data Eng* 20(7):992–1006
32. Popivanov I, Miller RJ (2002) Similarity search over time-series data using wavelets. In: Proceedings of the 18th international conference on data engineering, pp 212–221
33. Soisalon-Soininen E, Widmayer P (2003) Single and bulk updates in stratified trees: an amortized and worst-case analysis. *Comput Sci Perspect*, pp 278–292
34. TSST (2012) <http://www.usvao.org/science-tools-services/time-series-search-tool/>
35. Wu Y-L, Agrawal D, Abbadi AE (2000) A comparison of DFT and DWT based similarity search in time-series databases. In: Proceedings of the 9th international conference on information and knowledge management (CIKM '00). ACM, New York, NY, pp 488–495
36. Van den Bercken J, Seeger B (2001) An evaluation of generic bulk loading techniques. VLDB, pp 461–470
37. Van den Bercken J, Seeger B, Widmayer P (1997) A generic approach to bulk loading multidimensional index structures. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proceedings of the 23rd international conference on very large data bases (VLDB '97). Morgan Kaufmann Publishers Inc., San Francisco, CA, pp 406–415
38. Zoumpatianos K, Palpanas T, Mylopoulos J (2012) Strategic management for real-time business intelligence. In: International workshop on business intelligence for the real, time enterprise (BIRTE)

Author Biographies



Alessandro Camerra is a technical specialist at IBM System and Technology Group and a student at the Politecnico of Milan, Italy. Before that, he worked as research assistant at the University of Trento, Italy, and he was a visiting researcher at the University of California, Riverside. His publications mainly cover the area of time series indexing and wireless sensor network technologies. He has received a best paper award at PERCOM 2012.



Jin Shieh received a B.S and PhD degree in computer science from the University of California, Riverside (UCR) in 2005 and 2010, respectively. His research interests at UCR involved data mining with emphasis on the analysis and applications of time series. Specifically, this included work on searching and retrieval of time series. He has been with Microsoft Corporation, Redmond, WA, since 2010.



Themis Palpanas is a professor of computer science at the University of Trento, Italy. Before that, he worked at the IBM T.J. Watson Research Center and has also worked for the University of California at Riverside, Microsoft Research and IBM Almaden Research Center. He is the author of eight US patents, three of which are part of commercial products. He has received three Best Paper awards (PERCOM 2012, ICDE 2010, and ADAPTIVE 2009) and is General Chair for VLDB 2013.



Thanawin Rakthanmanon is a faculty member of the Department of Computer Engineering, Kasetsart University, Thailand. He has graduated with a Ph.D. from the Department of Computer Science and Engineering at the University of California, Riverside, in 2012. His research interests are Data Mining and Machine Learning, especially in efficient algorithm developments, motif discovery, time series classification/clustering, and document analysis. During his Ph.D., he has published several papers in top-tier data mining conference and journals, such as KDD, ICDM, SDM, KAIS and TKDD. Moreover, in 2012, his paper won the KDD best paper award.



Eamonn Keogh is a full professor of Computer Science at the University of California, Riverside. His research interests are data mining, machine learning, and information retrieval. Several of his papers have won best paper awards, including papers at SIGKDD, ICDM, and SIGMOD. Dr. Keogh is the recipient of a 5-year NSF Career Award for “Efficient Discovery of Previously Unknown Patterns and Relationships in Massive Time Series Databases.”