*i*SAX 2.0: Indexing and Mining One Billion Time Series

Alessandro Camerra Themis Palpanas
University of Trento
a.camerra@studenti.unitn.it, themis@disi.unitn.eu

Jin Shieh Eamonn Keogh
University of California, Riverside
{shiehj, eamonn}@cs.ucr.edu

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, a data structure 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 method 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

I. INTRODUCTION

The problem of indexing and mining time series has captured the interest of the data mining and database community for almost two decades. However, there remains a huge gap between the scalability of the methods in the current literature, and the needs of practitioners in many domains. 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, GE.
- "*…an archival rate of 3.6 billion points a day, how can we (do similarity search) in this data?*" Josh Patterson, TVA.

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: e.g., a state-of-the-art method [3] 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 is clearly doomed to be disk resident. Once a dimensionality-reduced representation (i.e DFT, DWT, SAX, etc.) has been decided on, the only way to improve retrieval times is by optimizing splitting algorithms for tree-based indexes (i.e., R-trees, M-trees, etc.), since a poor splitting policy leads to excessive and useless subdivisions, which create unnecessarily deep sub-trees and causing lengthier traversals.

In this work we solve both of these problems with significant extensions to the recently introduced multi-resolution symbolic representation ***i**ndexable **S**ymbolic **A**ggregate appro**X**imation* (*i*SAX) [3]. 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 72% 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 50%, 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. On four occasions the best paper winners at SIGKDD/SIGMOD have looked at the problem of indexing time series, with the largest dataset considered by each paper being 500,000 objects [20], 100,000 objects [21], 6,480 objects [1], and 27,000 objects [23]. 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 *i*SAX 2.0, a data structure suitable for indexing and mining time series, to scale to very large datasets.
- 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.
- We also 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 present the first approach that is experimentally validated to scale to data collections of time series with up to 1 *billion* objects.

The rest of the paper is organized as follows. We review some background material in Section II. Section III introduces the basic pillars for our scalable index, *i*SAX 2.0.

Section IV discusses the experimental evaluation. Section V presents the related work, and Section VI the conclusions.

II. Preliminaries

As noted previously, there are numerous dimensionality reduction techniques available for time series. In this section, we review SAX, and its recent extension, *i*SAX, which are at the heart of our proposed ideas. (For a more detailed discussion, refer to [3].)

A. *The SAX Representation*

In Figure 1(i) 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 $\overline{C} = \bar{c}_1, \ldots, \bar{c}_w$. The ith element of $\overline{C}$ is calculated by:

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

Figure 1(ii) shows T converted into this representation (called PAA [22]) reducing the dimensionality from 16 to 4.

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.

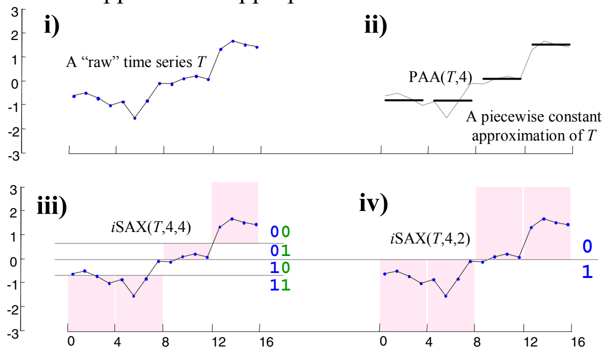

Figure 1. i) A time series T, of length 16. ii) A PAA approximation of T, with 4 segments. A time series T converted into SAX words of cardinality 4 (iii), and cardinality 2 (iv).

The SAX representation supports arbitrary breakpoints, however it has been shown that an effective choice is a sorted list of numbers $Breakpoints = \beta_1, \ldots, \beta_{a-1}$ 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 Figure 1(iii) 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 "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 Figure 1(iv) 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 I makes this clearer.

TABLE I. 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}	

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

B. *The iSAX Representation*

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

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

However, this can make the SAX word ambiguous. If we see *just* the SAX word {6,6,3,0} 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*SAX(T,4,8) = $\mathbf{T}^8$ = $\{6^8,6^8,3^8,0^8\}$

One of the key properties of the *i*SAX representation is the ability to compare two *i*SAX words of different cardinalities. Suppose we have two time series, T and S, which have been converted into *i*SAX words:

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

*i*SAX(S,4,2) = $\mathbf{S}^2$ = {**0** ,**0** ,**1** ,**1** } = $\{0^2,0^2,1^2,1^2\}$

We can find the lower bound between T and S, even though the *i*SAX words that represent them are of different cardinalities. The trick is to *promote* the lower cardinality representation into the cardinality of the larger before giving it to the MINDIST function. We can think of the tentatively promoted $\mathbf{S}^2$ word as $\mathbf{S}^8$ = {**0****$_1$,**0****$_2$,**1****$_3$,**1****$_4$}, then the question is simply what are correct values of the missing **$_i$ bits. Note that both cardinalities can be expressed as the power of some integer, guaranteeing an overlap in the breakpoints used during SAX computation. Concretely, if we have an *i*SAX cardinality of X, and an *i*SAX cardinality of 2X, then the breakpoints of the former are a proper subset of the latter. This is shown in Figure 1(iii) and Figure 1(iv).

Using this insight, we can obtain the missing bit values in $\mathbf{S}^8$ by examining each position and computing the bit values at the higher cardinality which are enclosed by the known bits at the current (lower) cardinality and returning the one which is closest in SAX space to the corresponding value in $\mathbf{T}^8$. This method obtains the $\mathbf{S}^8$ representation usable for MINDIST calculations: S^8 = {**011**,**011**,**100**,**100**}.

Note that this may not be the same *i*SAX word we would have gotten if we had converted the original time series S. We cannot undo a lossy compression. However, using this *i*SAX word *does* give us an admissible lower bound.

Finally, note that in addition to comparing between *i*SAX words of different cardinalities, the promotion trick described above can be used to compare *i*SAX words where *each* word has mixed cardinalities (such as {**111**, **11**, **101**, **0**} = {7^8,3^4,5^8,0^2}).

*i*SAX 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 *i*SAX 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.

C. *Indexing iSAX*

*i*SAX's variable granularity allows us to index time series. Using the *i*SAX representation, and by defining values for the cardinality b and wordlength w, we can produce a set of b^w different mutually exclusive *i*SAX 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 *i*SAX 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 *i*SAX 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 *i*SAX 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 *i*SAX 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 representative of the complete *i*SAX space and is similar in functionality to an internal node. The root node contains no SAX representation, but only pointers to the children nodes.

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 *i*SAX word for each time series.

Internal Node: An internal node designates a split in *i*SAX space, and is created when the number of time series contained by a leaf node exceeds *th*. The internal node splits the *i*SAX space by promotion of cardinal values along one or more dimensions as per the iterative doubling policy. *i*SAX 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.

III. 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 Section IV).

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 [3]. 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 enable it to efficiently operate with data collections orders of magnitude larger than previously tested. We will refer to this *improved* *i*SAX index as *i*SAX 2.0.

A. *Bulk Loading*

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 Section V).

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 Figure 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 *i*SAX 2.0 nodes. This correspondence remains stable throughout the creation of the index, because unlike nodes in other indexing structures, *i*SAX 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 *i*SAX 2.0 tree). The LBL corresponds to leaf nodes. There are no buffers related to the internal (i.e., other than the first level) *i*SAX 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 *i*SAX 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.

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 Figure 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.)

At the end of Phase 1, we have time series collected in the FBL buffers. This situation is depicted in Figure 2(*left*). 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.

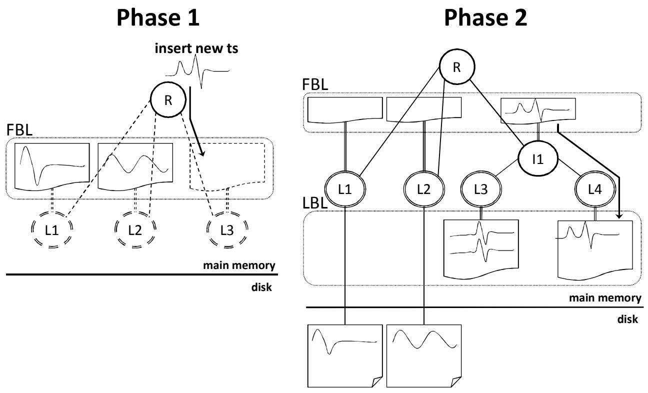

Figure 2. The bulk loading algorithm. *left*) Phase 1 fills the FBL buffers with time series until main memory is full. *right*) Phase 2, processing subtree rooted at node *I1* (subtrees rooted at nodes *L1* and *L2* have already been flushed to disk).

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(*right*), by emptying the right-most FBL buffer, we create the subtree rooted at internal node *I1*. 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(*right*), the LBL buffers for the subtrees rooted at nodes *L1* and *L2* have already been flushed to disk, and all the available memory can be dedicated to the LBL buffers of the *I1* 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. 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, 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, and in the interest of space do not report detailed results on this variation of the algorithm.

B. 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 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.

```
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
```

Figure 3. Pseudocode for the bulk loading algorithm.

Consider the example depicted in Figure 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 μ ± 3σ range of the corresponding symbols. 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.

The pseudocode for the node splitting algorithm is shown in Figure 5 (called every time we have to create new internal node: lines 29 and 37 in Figure 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 *i*SAX 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 μ ± 3σ crosses any of the *i*SAX 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).

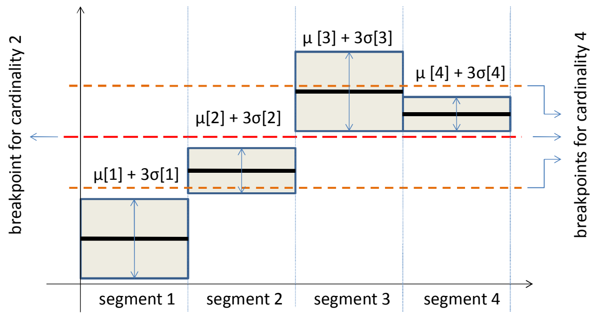

Figure 4. Node splitting policy example.

```
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 = getBreakPoint( s )  // breakpoint of s with increased cardinality
7       if ( b within mean[ s ] ± 3stdev[ s ] )
8           // segment s is candidate for splitting
9           if  ( mean[ s ] closer to b than  segmentToSplit )
10              segmentToSplit = s
11  segmentToSplit.IncreaseCardinality()
```

Figure 5. Pseudocode for the node splitting algorithm.

IV. Experimental Evaluation

We have designed all experiments such that they are reproducible. To this end, we have built a webpage which contains *all* datasets and code used in this work, together with spreadsheets that contain the raw numbers displayed in all the figures [24].

Experimental evaluation was conducted on an Intel Xeon E5504 with 24GB 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 Section IV.B, we used an AMD Athlon 64 X2 5600+ with 3GB of memory, 400 GB Seagate Barracuda 7200.10 hard disk, and running Windows XP SP2 (with /3GB 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, *C*) examples of higher level data mining algorithms built using our index as a subroutine.

The algorithms that we evaluate are *i*SAX 2.0, and the original *i*SAX, where all the available main memory is used for disk buffer management (i.e., buffers corresponding to the leaf level nodes). We also compare to *i*SAX-BufferTree, which is an adaptation of the *Buffered R-Tree* bulk loading algorithm [17]. 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 *i*SAX 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 do not additionally compare our ideas to other time series indices. This would normally be an untenable position for a paper, but we note the following three points.

First, the largest time series dataset indexed by a rival technique is one million objects [2]. In contrast, in this work we consider a dataset which is one thousand times larger. Even our "*small*" datasets are two orders of magnitude larger. No papers that we are aware of claim to scale to the truly massive datasets we wish to consider, and forcing them to do so may misrepresent their contributions (in indexing moderate sized datasets).

Second, 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, i.e., DWT vs. 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 [14].

Finally, a unique property of *i*SAX is its tiny bit-aware index size. This means that an *i*SAX 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 *i*SAX 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 6 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 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 8 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 [14]. They also show that once the cardinality of *i*SAX is greater than 50, it is competitive with the other methods, even though it requires only one eighth the space (one byte per coefficient vs. eight bytes per coefficient for PAA/DWT/DFT/CHEB).

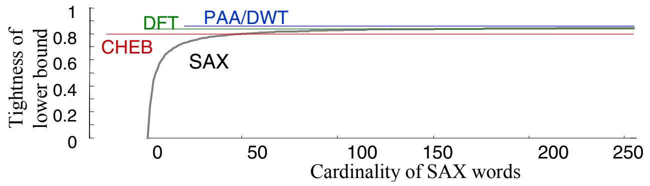

Figure 6. A comparison of the tightness of lower bound for various time series representations. All approaches except SAX use a constant 8 bytes, and therefore have a constant tightness of lower bound. The results for SAX show the effect of varying the cardinality from 2 to 256 (and hence the number of bits from 2 to 8).

In summary, this work is only *apparently* tackling a problem that has been worked on before. In fact, indexing a billion time series is effectively a new problem, considered here for the first time.

A. Scalability of iSAX 2.0

In this section, we present experimental results on the scalability of *i*SAX 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.

1) Splitting Policy Evaluation

We ran experiments in order to evaluate the new splitting policy implemented in *i*SAX 2.0. In these experiments, we compare our results against those obtained by the use of the *i*SAX 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 7(*right*) 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 *i*SAX 2.0 can effectively reduce the number of nodes required by the index. On average, *i*SAX 2.0 needs 34% less nodes than the *i*SAX index. These results validate our premise that using the first moments of the distributions of the *i*SAX 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 (Figure 7(*left*)). 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 to 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 Figure 7(*right*) 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.

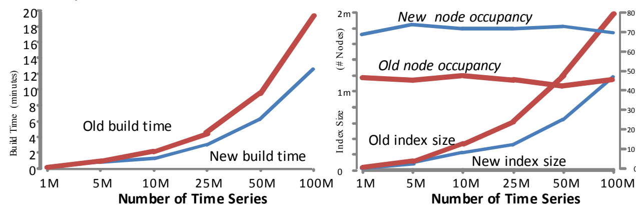

Figure 7. Splitting policy comparison between *i*SAX (old) and *i*SAX 2.0 (new) when varying the size of the dataset: construction time (*left*), number of nodes and leaf node occupancy (*right*).

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*=8000 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 Figure 8(*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 it would take around 56 days to index 1 billion time series. The *i*SAX-BufferTree algorithm initially performs better than *i*SAX, 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, *i*SAX 2.0 manages to index a dataset with 100 million time series in just 16 hours. The 1 billion time series dataset is indexed in less than 400 hours (about 16 days), which corresponds to an indexing time of 1ms/time series.

Disk Page Accesses: In Figure 8(*middle*), we show the number of disk page accesses performed by the three indexing methods during the same experiments.

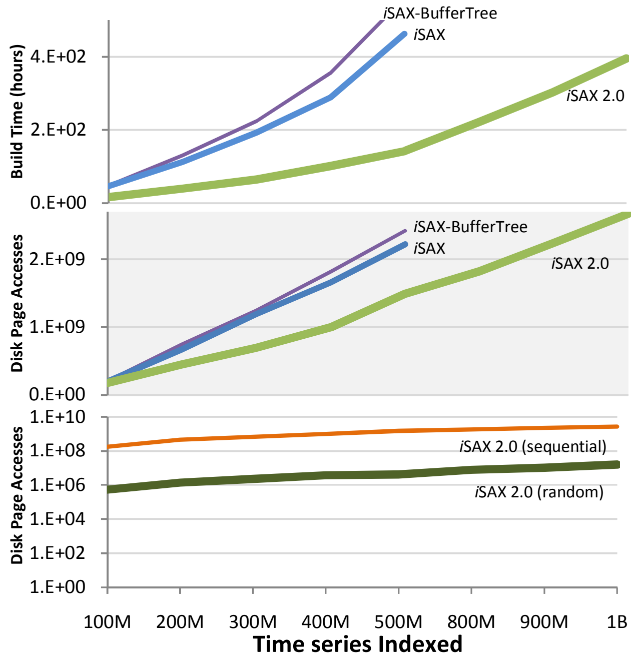

Figure 8. Index metrics as a function of dataset sizes. *top*) Time to build an index. *middle*) Disk page accesses between indexing methods. *bottom*) Distribution of sequential and random disk page accesses for *i*SAX 2.0.

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 Figure 8(*bottom*)). More than 99.5% of the disk page accesses that *i*SAX 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 *i*SAX and *i*SAX-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 *i*SAX 2.0.

These results show that the bulk loading algorithm is extremely effective in reducing the I/O cost, thus, enabling *i*SAX 2.0 to index 1,000,000,000 time series.

B. *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 coagulate* 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 [5].

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 9(*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]. 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 hours. 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 if the chemical thiamethoxam causes a reduction in xylem[1] 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 Dept. at UC Riverside asked us to create an efficient tool for mining massive EPG collections [7]. 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 Figure 9(*bottom*) we see a copy of Fig. 2 from [4]. 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?

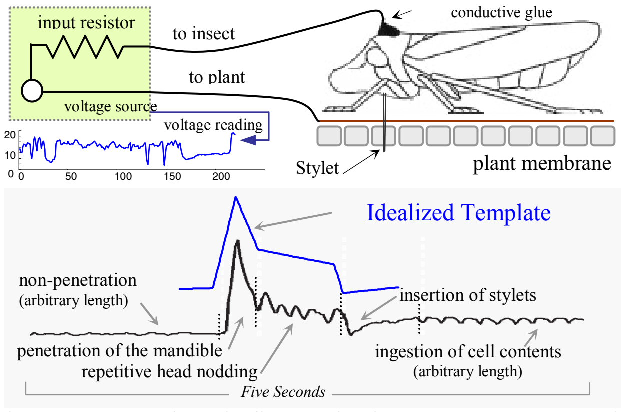

Figure 9. *top*) A schematic diagram showing an EPG apparatus used to record insect behavior. *bottom*) An EPG insect behavior derived from subset of Fig. 2 from [4]. An idealized version of the observed behavior created by us is shown with a bold blue line.

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 2000 and *w* of 8 to construct an index on our AMD machine. Even with fewer resources, it took only 6.25 hours to build the index, which occupied a total of 26.6 gigabyte on disk space. As shown in Figure 9(*bottom*), we used the simple idealized version as a query to our database. Figure 10(*left*) shows the result of an approximate search, which takes less than 0.5 seconds to answer.

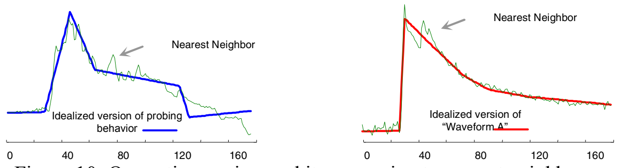

Figure 10. Query time series, and its approximate nearest neighbor.

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, Figure 10(*right*) shows that the match is much closer.

C. *Mining Massive DNA Sequences*

The DNA of the Rhesus Macaque (*Macaca mulatta*) genome consists of nearly 3 billion base pairs (approximately 550,000 pages of text if written out in the format of this

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

paper), beginning with **TAACCCTAACCCTAA**… We converted this sequence into a time series using the simple algorithm shown in TABLE II.

TABLE II. AN ALGORITHM FOR CONVERTING DNA TO TIME SERIES.

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

Figure 11(*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?

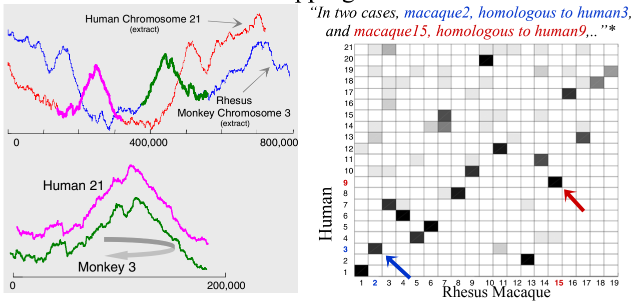

Figure 11. *left*) An example of DNA converted into time series. *right*) The cells represent potential mappings between the Macaque and Human Genomes. The darker the cell, the more often the nearest neighbor of a time series taken from a particular human chromosome had a nearest neighbor from a particular Macaque chromosome. *Quote from *An initial genetic linkage map of the rhesus macaque*. Rogers et al.[2].

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 5 to extract a total of 21,612,319 subsequences. To index, we used a *th* size of 1000 and *w* of 10. In total, it took 9 hours 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 ten nearest neighbors. In Figure 11(*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* ten queries from 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*" [12]. More generally, this correspondence matrix is at least 95% in agreement with the current agreement on homology between these two primates [12]. 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 hours, and produce objectively correct results.

[2] The smallest chromosomes including the sex chromosomes are omitted

D. *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 [8]. 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 [9][10]. 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 [8]. As shown in Figure 12, these color histograms can be considered pseudo "time series". At indexing time we omitted very simple images (e.g. those that are comprised of 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 2000 and *w* of 8. It took 12.3 hours to build the index, which is inconsequential compared to the nine months of twenty-four hours a day crawling it took to collect it [8]. The data occupies a total of 133 gigabytes of disk space. The latter figure only includes the space for the time series, the images themselves required an extra of 227 gigabytes.

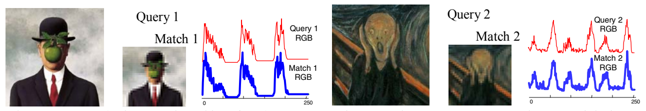

Figure 12. *left*) A detail of *The Son of Man* by René Magritte, which we used as a query to our index, finding "Match 1". *right*) A detail of *The Scream* by Edvard Munch, which we used as a query that returned "Match 2". The insets show the similarity of the images in RGB histogram space.

Does this random sampling of the webs 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 Figure 12. 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* [9].

V. Related Work

The literature on time series indexing is vast; see [14][2][3] 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, including Discrete Fourier Transformation [20], Singular Value Decomposition (SVD), Discrete Cosine Transformation, Discrete Wavelet Transformation [26], Piecewise Aggregate Approximation [22], Adaptive Piecewise Constant Approximation [21], Chebyshev polynomials [1]. 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 [14].

The approximation we use in this work is intrinsically different from the techniques listed above in that it is discrete [25], 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.

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 [15][16][17][18][19][27]. For these structures two main approaches have been proposed. First, we have the merge-based techniques [15] 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 clustering datasets of such scale could incur a cost higher than indexing. Second, there are the buffering-based techniques [17][18][27] 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 [17]). Another interesting technique would be the two step approach of the Path-Based method [18]. 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.

VI. 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. 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.

Acknowledgment

This research was funded by NSF awards 0803410 and 0808770, and by FP7 EU IP project KAP (grant agreement no. 260111).

References

[1] Cai, Y. and Ng, R. 2004. Indexing spatio-temporal trajectories with Chebyshev polynomials. In Proc. SIGMOD 2004. 599-610.

[2] Assent I., Krieger R., Afschari F., Seidl T. (2008). The TS-Tree: Efficient Time Series Search and Retrieval. EDBT 2008

[3] J. Shieh and E. Keogh. iSAX: indexing and mining terabyte sized time series. In ACM SIGKDD, 2008

[4] 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: 183–91

[5] Andersen, P., Brodbeck, B., Mizell, R. (2009) Assimilation efficiency of free and protein amino acids by H. vitripennis feeding on C.sinensis and V. vinifera. Florida Entomologist . March 1, 2009.

[6] 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. Journal of Insect Physiology. Vol 55, Issue 10, October 2009, Pages 869-884

[7] Greg Walker (2009) Personal Communication. August 12th.

[8] A. Torralba, R. Fergus, W. T. Freeman (2008). 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. IEEE PAMI Vol. 30, No. 11. (2008), pp. 1958-1970.

[9] A. Halevy, P. Norvig, and F. Pereira (2009). The Unreasonable Effectiveness of Data," IEEE Intell. Syst., vol. 24, no. 2, pp. 8–12.

[10] C. Anderson, "The End of Theory: The Data Deluge Makes the Scientific Method Obsolete," Wired, vol. 16, no. 7, June 23, 2008.

[11] Reeves, R.H. and Cabin, D.E. (1999) Mouse chromosome 16. Mamm. Genome, 10, 957.

[12] Rogers, J. et al. An initial genetic linkage map of the rhesus macaque (Macaca mulatta) genome using human microsatellite loci. Genomics 87, 30–38 (2006).

[13] Asif, T.C., et al. 2002. Initial sequencing and comparative analysis of the mouse genome. Nature 420: 520–562

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

[15] Rupesh Choubey, Li Chen, Elke A. Rundensteiner: GBI: A Generalized R-Tree Bulk-Insertion Strategy. SSD 1999: 91-108

[16] An, N., Venkata, K. Kanth, R., Ravada, S. Improving Performance with Bulk-Inserts in Oracle R-Trees. VLDB 2003: 948-951

[17] Lars Arge, Klaus Hinrichs, Jan Vahrenhold, Jeffrey Scott Vitter: Efficient Bulk Operations on Dynamic R-Trees. Algorithmica 33(1):104-128 (2002)

[18] Jochen Van den Bercken, Bernhard Seeger: An Evaluation of Generic Bulk Loading Techniques. VLDB 2001: 461-470

[19] Jochen Van den Bercken, Bernhard Seeger, Peter Widmayer: A Generic Approach to Bulk Loading Multidimensional Index Structures. VLDB 1997: 406-415

[20] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast Subsequence Matching in Time-Series Databases. SIGMOD , 1994.

[21] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In SIGMOD Conference, 2001.

[22] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. Knowl. Inf. Syst., 3(3), 2001.

[23] E. J. Keogh, Smyth P. A Probabilistic Approach to Fast Pattern Matching in Time Series Databases. KDD 1997: 24-30

[24] http://www.cs.ucr.edu/~eamonn/iSAX/iSAX.html

[25] J. Lin, E. J. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. Data Min. Knowl. Discov., 15(2), 2007.

[26] I. Popivanov and R. J. Miller. Similarity Search Over Time-Series Data Using Wavelets. In ICDE, 2002.

[27] Eljas Soisalon-Soininen, Peter Widmayer: Single and Bulk Updates in Stratified Trees: An Amortized and Worst-Case Analysis. Computer Science in Perspective 2003: 278-292.