*i*SAX: Indexing and Mining Terabyte Sized Time Series

Jin Shieh
Dept. of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
shiehj@cs.ucr.edu

Eamonn Keogh
Dept. of Computer Science & Engineering
University of California, Riverside
Riverside, CA 92521
eamonn@cs.ucr.edu

ABSTRACT
Current research in indexing and mining time series data has produced many interesting algorithms and representations. However, it has not led to algorithms that can scale to the increasingly massive datasets encountered in science, engineering, and business domains. In this work, we show how a novel multi-resolution symbolic representation can be used to index datasets which are several orders of magnitude larger than anything else considered in the literature. Our approach allows both fast exact search and ultra fast approximate search. We show how to exploit the combination of both types of search as sub-routines in data mining algorithms, allowing for the exact mining of truly massive real world datasets, containing millions of time series.

Keywords
Time Series, Data Mining, Representations, Indexing

1. INTRODUCTION
The increasing level of interest in indexing and mining time series data has produced many algorithms and representations. However, with few exceptions, the size of datasets considered, indexed, and mined seems to have stalled at the megabyte level. At the same time, improvements in our ability to capture and store data have lead to the proliferation of terabyte-plus time series datasets. In this work, we show how a novel multiresolution symbolic representation called ***i**ndexable **S**ymbolic **A**ggregate approX**i**mation* (*i*SAX) can be used to index datasets which are several orders of magnitude larger than anything else considered in current literature.

The *i*SAX approach allows for both fast exact search and ultra fast approximate search. Beyond mere similarity search, we show how to exploit the combination of both types of search as sub-routines in data mining algorithms, permitting the exact mining of truly massive datasets, with millions of time series, occupying up to a terabyte of disk space.

Our approach is based on a modification of the SAX representation to allow extensible hashing [12]. In essence, we show how we can modify SAX to be a multiresolution representation, similar in spirit to wavelets. It is this multiresolution property that allows us to index time series with zero overlap at leaf nodes [2], unlike R-trees and other spatial access methods.

As we shall show, our indexing technique is fast and scalable due to intrinsic properties of the *i*SAX representation. Because of this, we do not require the use of specialized databases or file managers. Our results, conducted on massive datasets, are all achieved using a simple tree structure which simply uses the standard Windows XP NTFS file system for disk access. While it might have been possible to achieve faster times with a sophisticated DBMS, we feel that the simplicity of this approach is a great strength, and will allow easy adoption, replication, and extension of our work.

A further advantage of our representation is that, being symbolic, it allows the use of data structures and algorithms that are not well defined for real-valued data, including suffix trees, hashing, Markov models etc [12]. Furthermore, given that *i*SAX is a superset of classic SAX, the several dozen research groups that use SAX will be able to adopt *i*SAX to improve scalability [11].

The rest of the paper is organized as follows. In Section 2 we review related work and background material. Section 3 introduces the *i*SAX representation, and Section 4 shows how it can be used for approximate and exact indexing. In Section 5 we perform a comprehensive set of experiments on both indexing and data mining problems. Finally, in Section 6 we offer conclusions and suggest directions for future work.

2. BACKGROUND AND RELATED WORK

2.1 Time Series Distance Measures
It is increasingly understood that Dynamic Time Warping (DTW) is better than Euclidean Distance (ED) for most data mining tasks in most domains [17]. It is therefore natural to ask why we are planning to consider Euclidean distance in this work. The well documented superiority of DTW over ED is due to the fact that in small datasets it *might* be necessary to warp a little to match the nearest neighbor. However, in larger datasets one is more likely to find a close match without the need to warp. As DTW warps less and less, it degenerates to simple ED. This was first noted in [14] and later confirmed in [17] and elsewhere. For completeness, we will show a demonstration of this effect. We measured the classification accuracy of both DTW and ED on increasingly large datasets containing the CBF and Two-Pat problems, two classic time series benchmarks. Both datasets allow features to warp up to 1/8 the length of the sequence, so they may be regarded as highly warped datasets. Figure 1 shows the result.

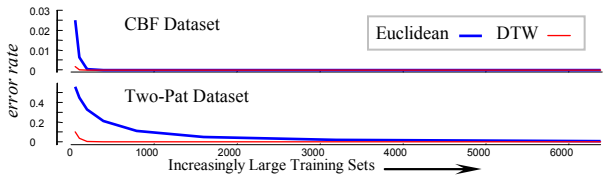

Figure 1: The error rate of DTW and ED on increasingly large instantiations of the CBF and Two-Pat problems. For even moderately large datasets, there is no difference in accuracy

As we can see, for small datasets, DTW is significantly more accurate than ED. However, as the datasets get larger, the

difference diminishes, and by the time there are mere thousands of objects, there is no measurable difference. In spite of this, and for completeness, we explain in an offline Appendix [10] that we can index under DTW with *i*SAX with only trivial modifications.

2.2 Time Series Representations

There is a plethora of time series representations proposed to support similarity search and data mining. Table 1 show the major techniques arranged in a hierarchy.

Table 1: A Hierarchy of Time Series Representations

- **Model Based**
  - Markov Models
  - Statistical Models
  - Time Series Bitmaps
- **Data Adaptive**
  - Piecewise Polynomials
    - Interpolation*
    - Regression
  - Adaptive Piecewise Constant Approximation*
  - Singular Value Decomposition*
  - Symbolic
    - Natural Language
    - Strings
      - Non-Lower Bounding [1][7][13]
      - **SAX*** [12], ***i*SAX***
  - Trees
- **Non-Data Adaptive**
  - Wavelets*
  - Random Mappings
  - Spectral
    - DFT* [6]
    - DCT*
    - Chebyshev Polynomials* [4]
  - Piecewise Aggregate Approximation* [9]
- **Data Dictated**
  - Clipped Data*

Those representations annotated with an asterisk have the very desirable property of allowing lower bounding. That is to say, we can define a distance measurement on the reduced abstraction that is guaranteed to be less than or equal to the true distance measured on the raw data. It is this lower bounding property that allows us to use a representation to index the data with a guarantee of no false dismissals [6]. The list of such representations includes (in approximate order of introduction) the discrete Fourier transform (DFT) [6], the discrete Cosine transform (DCT), the discrete Wavelet transform (DWT), Piecewise Aggregate Approximation (PAA) [8], Adaptive Piecewise Constant Approximation (APCA), Chebyshev Polynomials (CHEB) [4] and Indexable Piecewise Linear Approximation (IPLA). We will provide the first empirical comparison of all these techniques in Section 5.

The only lower bounding omissions from our experiments are the eigenvalue analysis techniques such as SVD and PCA. While such techniques give optimal linear dimensionality reduction, we believe they are untenable for massive datasets. For example, while [16] notes that they can transform 70,000 time series in under 10 minutes, this assumes the data can fit in main memory. However, to transform all the out-of-core (disk resident) datasets we consider in this work, SVD would require several months.

There have been several dozen research efforts that propose to facilitate time series search by first symbolizing the raw data [1][7][13]. However, in every case, the authors introduced a distance measure defined on the newly derived symbols. This allows false dismissals with respect to the original data. In contrast, the proposed work uses the symbolic words to internally organize and index the data, but retrieves objects with respect to the Euclidean distance on the original raw data.

2.3 Review of Classic SAX

For concreteness, we begin with a review of SAX [12]. In Figure 2.*left* we illustrate a short time series T, which we will use as a running example throughout this paper.

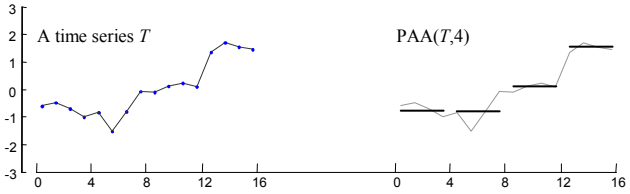

Figure 2: *left*) A time series *T*, of length 16. *right*) A PAA approximation of *T*, with 4 segments

A time series T of length n can be represented in a w-dimensional space by a vector of real numbers $\overline{T} = \bar{t}_1,\ldots,\bar{t}_w$. The i^{th} element of $\overline{T}$ is calculated by the equation to the left:

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

Figure 2.*right* shows our sample time series converted into a representation called PAA [9]. The PAA representation reduces the dimensionality of a time series, in this case from 16 to 4. The SAX representation takes the PAA representation as an input and discretizes it into a small alphabet of symbols with a cardinality of size **a**. The discretization is achieved by imagining a series of breakpoints running parallel to the x-axis and labeling each region between the breakpoints with a discrete label. Any PAA value that falls within that region can then be mapped to the appropriate discrete value. Figure 3 illustrates the idea.

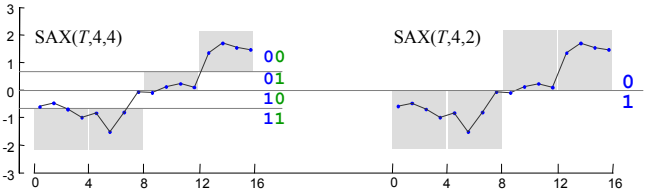

Figure 3: A time series *T* converted into SAX words of cardinality 4 {11, 11, 01, 00} (*left*), and cardinality 2 {1, 1, 0, 0} (*right*)

While the SAX representation supports arbitrary breakpoints, we can ensure almost equiprobable symbols within a SAX word if we use 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 $\beta_{i+1} = 1/a$ (β_0 and β_a are defined as $-\infty$ and ∞, respectively). Table 2 shows a table for such breakpoints for cardinalities from 2 to 8.

Table 2: SAX breakpoints

βi \ a	2	3	4	5	6	7	8
β_1	0.00	-0.43	-0.67	-0.84	-0.97	-1.07	-1.15
β_2		0.43	0.00	-0.25	-0.43	-0.57	-0.67
β_3			0.67	0.25	0.00	-0.18	-0.32
β_4				0.84	0.43	0.18	0.00
β_5					0.97	0.57	0.32
β_6						1.07	0.67
β_7							1.15

A SAX word is simply a vector of discrete symbols. We use a boldface letter to differentiate between a raw time series and its SAX version, and we denote the cardinality of the SAX word with a superscript:

$$\text{SAX}(T,w,a) = \mathbf{T}^{\mathbf{a}} = \{t_1,t_2,..,t_{w-1},t_w\}$$

In previous work, we represented each SAX symbol as a letter or integer. Here however, we will use binary numbers for reasons

that will become apparent later. For example, in Figure 3 we have converted a time series T of length 16 to SAX words. Both examples have a word length of 4, but one has a cardinality of 4 and the other has a cardinality of 2. We therefore have SAX(T,4,4) = $\mathbf{T^4}$ = {11,11,01,00} and SAX(T,4,2) = $\mathbf{T^2}$ = {1,1,0,0}.

The astute reader will have noted that once we have $\mathbf{T^4}$ we can derive $\mathbf{T^2}$ simply by ignoring the trailing bits in each of the four symbols in the SAX word. As one can readily imagine, this is a recursive property. For example, if we convert 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, simply by ignoring the correct number of bits. Table 3 makes this clearer.

Table 3: It is possible to obtain a reduced (by half) cardinality SAX word simply 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 }

As we shall see later, this ability to change cardinalities on the fly is a useful and exploitable property.

Given two time series T and S, their Euclidean distance is:

$$D(T,S) \equiv \sqrt{\sum_{i=1}^{n}(T_i - S_i)^2}$$

If we have a SAX representation of these two time series, we can define a lower bounding approximation to the Euclidean distance as:

$$MINDIST(\mathbf{T^2},\mathbf{S^2}) \equiv \sqrt{\frac{n}{w}}\sqrt{\sum_{i=1}^{w}(dist(t_i,s_i))^2}$$

This function requires calculating the distance between two SAX symbols and can be achieved with a lookup table, as in Table 4.

Table 4: A SAX *dist* lookup table for a = 4

	00	01	10	11
00	0	0	0.67	1.34
01	0	0	0	0.67
10	0.67	0	0	0
11	1.34	0.67	0	0

The distance between two symbols can be read off by examining the corresponding row and column. For example, $dist$(00,01) = 0 and $dist$(00,10) = 0.67.

For clarity, we will give a concrete example of how to compute this lower bound. Recall our running example time series T which appears in Figure 2. If we create a time series S that is simply T's mirror image, then the Euclidean distance between them is D(T,S) = 46.06.

As we have already seen, SAX(T,4,4) = $\mathbf{T^4}$ = {11,11,01,00}, and therefore SAX(S,4,4) = $\mathbf{S^4}$ = {00,01,11,11}. The invocation of the *MINDIST* function will make calls to the lookup table shown in Table 4 to find:

$$dist(t_1,s_1) = dist(11,00) = 1.34$$
$$dist(t_2,s_2) = dist(11,01) = 0.67$$
$$dist(t_3,s_3) = dist(01,11) = 0.67$$
$$dist(t_4,s_4) = dist(00,11) = 1.34$$

Which, when plugged into the *MINDIST* function, gives:

$$MINDIST(\mathbf{T^2},\mathbf{S^2}) = \sqrt{\frac{16}{4}}\sqrt{1.34^2 + 0.67^2 + 0.67^2 + 1.34^2}$$

…to produce a lower bound value of 4.237. In this case, the lower bound is quite loose; however, having either more SAX symbols or a higher cardinality will produce a tighter lower bound. It is natural to ask how tight this lower bounding function can be, relative to natural competitors like PAA or DWT. This depends on the data itself and the cardinality of the SAX words, but coefficient for coefficient, it is surprisingly competitive with the other approaches. To see this, we can measure the *tightness of the lower bounds*, which is defined as the lower bounding distance over the true distance [9]. Figure 4 shows this for random walk time series of length 256, with eight PAA or DWT coefficients and SAX words also of length eight. We varied the cardinality of SAX from 2 to 256, whereas PAA/DWT used a constant 4 bytes per coefficient. The results have been averaged over 10,000 random walk time series comparisons.

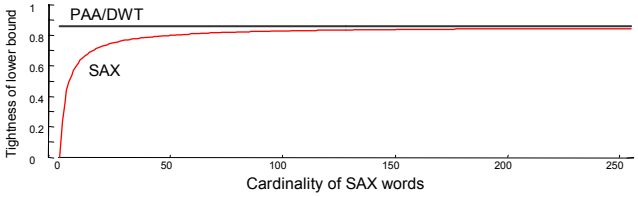

Figure 4: The tightness of lower bounds for increasing SAX cardinalities, compared to a PAA/DWT benchmark

The results show that for small cardinalities the SAX lower bound is quite weak, but for larger cardinalities it rapidly approaches that of PAA/DWT. At the cardinality of 256, which take 8 bits, the lower bound of SAX is 98.5% that of PAA/DWT, but the latter requires 32 bits. This tells us that if we compare representations coefficient for coefficient, there is little to choose between them, but if we do bit-for-bit comparisons (cf. Section 5), SAX allows for much tighter lower bounds. This is one of the properties of SAX that can be exploited to allow ultra scalable indexing.

3. THE *i*SAX REPRESENTATION

Because it is tedious to write out binary strings, previous uses of SAX had integers or alphanumeric characters representing SAX symbols [12]. 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). Since all previous uses of SAX always used a single "hard-coded" cardinality, this has not been an issue. However, the fundamental contribution of this work is to show that SAX allows the comparison of words with different cardinalities, and even different cardinalities *within* a single word. We therefore must resolve this ambiguity. We do this by writing the cardinality as a superscript. For example, in the example above:

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

Because the individual symbols are ordinal, exponentiation is not defined for them, so there is no confusion in using superscripts in this context. Note that we are now using *i*SAX instead of SAX for reasons that will become apparent in a moment.

We are now ready to introduce a novel idea that will allow us to greatly expand the utility of *i*SAX.

3.1.1 *Comparing different cardinality iSAX words*

It is possible 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:

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

$$iSAX(S,4,2) = \mathbf{S^2} = \{0\ \ ,0\ \ ,1\ \ ,1\ \ \} = \{0^2,0^2,1^2,1^2\}$$

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. This guarantees an overlap in the breakpoints used during SAX computation. More 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 3.

Using this insight, we can obtain the missing bit values by examining for each position *i*, the known bits of $\mathbf{S^8}$, defined as $\mathbf{S^k}_i$ and its corresponding bits in $\mathbf{T^8}_i$:

```
IF S^k_i forms a prefix for T^8_i THEN,
        *_i = T^8_i for all unknown bits.
ELSE IF S^k_i is lexicographically smaller than
corresponding bits in T^8_i THEN,
        *_i = 1 for all unknown bits.
ELSE,
        *_i = 0 for all unknown bits.
```

This method obtains the $\mathbf{S^8}$ representation usable for MINDIST calculations:

$$\mathbf{S^8} = \{011,011,100,100\}$$

It is important to note that this is *not* necessarily 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 *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. For example, we can allow *i*SAX words such as $\{111,11,101,0\} = \{7^8,3^4,5^8,0^2\}$. If such words exist, we can simply align the two words in question, scan across each pair of corresponding symbols, and promote the symbol with lower cardinality to the same cardinality as the larger cardinality symbol. In the next section, we explain why it is useful to allow *i*SAX words with different cardinalities.

4. *i*SAX INDEXING

4.1 The Intuition behind *i*SAX Indexing

As it stands, it may appear that the classic SAX representation offers the potential to be indexed. We could choose a fixed cardinality of, say, 8 and a word length of 4, and thus have 8^4 separate labels for files on disk. For instance, our running example *T* maps to $\{6^8,6^8,3^8,0^8\}$ under this scheme, and would be inserted into a file that has this information encoded in its name, such as `6.8_6.8_3.8_0.8.txt`. The query answering strategy would be very simple. We could convert the query into a SAX word with the same parameters, and then retrieve the file with that label from disk. The time series in that file are likely to be very good approximate matches to the query. In order to find the exact match, we could measure the distance to the best approximate match, then retrieve all files from disk whose label has a MINDIST value less than the value of the best-so-far match. Such a methodology clearly guarantees no false dismissals.

This scheme has a fatal flaw, however. Suppose we have a million time series to index. With 4,096 possible labels, the average file would have 244 time series in it, a reasonable number. However, this is the *average*. For all but the most contrived datasets we find a huge skew in the distribution, with more than half the files being empty, and the largest file containing perhaps 20% of the entire dataset. Either situation is undesirable for indexing, in the former case, if our query maps to an empty file, we would have to do some ad-hoc trick (perhaps trying "misspellings" of the query label) in order to get the first approximate answer back. In the latter case, if 20% of the data must be retrieved from disk, then we can be at most five times faster than sequential scan. Ideally, we would like to have a user defined threshold *th*, which is the maximum number of time series in a file, and a mapping technique that ensures each file has at least one and at most *th* time series in it. As we shall now see, *i*SAX allows us to guarantee exactly this.

*i*SAX offers a bit aware, quantized, reduced representation with *variable* granularity. It is this variable granularity that allows us to solve the problem above. Imagine that we are in the process of building the 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, as we continue to build the index, we find another time series maps here, we have an overflow, so we 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:

Original File: $\{2^4,3^4,3^4,2^4\}$ splits into…

Child file 1: $\{4^8,3^4,3^4,2^4\}$

Child file 2: $\{5^8,3^4,3^4,2^4\}$

Note that in this example we split on the first symbol, promoting the cardinality from 4 to 8. 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, and in the latter, remapped to Child 2. The child files can be named with some protocol that indicates their variable cardinality, for example **`5.8`**`_3.4_3.4_2.4.txt` and **`4.8`**`_3.4_3.4_2.4.txt`.

The astute reader will have noticed that the intuition here is very similar to the classic idea of extensible hashing. This in essence is the intuition behind building an *i*SAX index, although we have not explained *how* we decide which symbol is chosen for promotion and some additional details. In the next sections, we formalize this intuition and provide details on algorithms for approximately and exactly searching an *i*SAX index.

4.2 *i*SAX Index Construction

As noted above, a set of time series represented by an *i*SAX word can be split into two mutually exclusive subsets by increasing the cardinality along one or more dimensions. The number of dimensions *d* and word length, *w*, $1 \le d \le w$, provide an upper bound on the fan-out rate. If each increase in cardinality per dimension follows the assumption of iterative doubling, then the alignment of breakpoints contains overlaps in such a way that hierarchical containment is preserved between the common *i*SAX word and the set of *i*SAX words at the finer granularity. Specifically, in iterative doubling, the cardinality to be used after the i^{th} increase in granularity is in accordance with the following

sequence, given base cardinality b: $b*2^i$. The maximum fan-out rate under such an assumption is 2^d.

The use of *i*SAX allows for the creation of index structures that are hierarchical, containing non-overlapping regions [2] (unlike R-trees etc [6]), and a controlled fan-out rate. For concreteness, we depict in Figure 5 a simple tree-based index structure which illustrates the efficacy and scalability of indexing using *i*SAX.

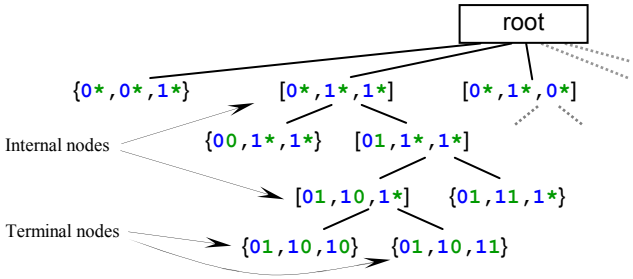

Figure 5: An illustration of an *i*SAX index

The index is constructed given base cardinality b, word length w, and threshold th. The index structure hierarchically subdivides the SAX space, resulting in differentiation between time series entries until the number of entries in each subspace falls below th. Such a construct is implemented using a tree, where each node represents a subset of the SAX space such that this space is a superset of the SAX space formed by the union of its descendents. A node's representative SAX space is congruent with an *i*SAX word and evaluation between nodes or time series is done through comparison of *i*SAX words. The three classes of nodes found in a tree and their respective functionality are described below:

Terminal Node: A terminal node is a leaf node which contains a pointer to an index file on disk with raw time series entries. All time series in the corresponding index file are characterized by the terminal node's representative *i*SAX word. A terminal node represents the coarsest granularity necessary in SAX space to enclose the set of contained time series entries. In the event that an insertion causes the number of time series to exceed th, the SAX space (and node) is split to provide additional differentiation.

Internal Node: An internal node designates a split in SAX space and is created when the number of time series contained by a terminal node exceeds th. The internal node splits the SAX space by promotion of cardinal values along one or more dimensions as per the iterative doubling policy. A hash from *i*SAX words (representing subdivisions of the SAX space) to nodes is maintained to distinguish differentiation between entries. Time series from the terminal node which triggered the split are inserted into the newly created internal node and hashed to their respective locations. If the hash does not contain a matching *i*SAX entry, a new terminal node is created prior to insertion, and the hash is updated accordingly. For simplicity, we employ binary splits along a single dimension, using round robin to determine the split dimension.

Root Node: The root node is representative of the complete SAX space and is similar in functionality to an internal node. The root node evaluates time series at base cardinality, that is, the granularity of each dimension in the reduced representation is b. Encountered *i*SAX words correspond to some terminal or internal node and are used to direct index functions accordingly. Un-encountered *i*SAX words during inserts result in the creation of a terminal node and a corresponding update to the hash table.

Pseudo-code of the insert function used for index construction is shown in Table 5. Given a time series to insert, we first obtain the *i*SAX word representation using the respective *i*SAX parameters at the current node (line 2). If the hash table does not yet contain an entry for the *i*SAX word, a terminal node is created to represent the relevant SAX space, and the time series is inserted accordingly (lines 22-24). Otherwise, there is an entry in the hash table, and the corresponding node is fetched. If this node is an internal node, we call its insert function recursively (line 19). If the node is a terminal node, occupancy is evaluated to determine if an additional insert warrants a split (line 7). If so, a new internal node is created, and all entries enclosed by the overfilled terminal node are inserted (lines 10-16). Otherwise, there is sufficient space and the entry is simply added to the terminal node (line 8).

Table 5: *i*SAX index insertion function

```
1  Function Insert(ts)
2  iSAX_word = iSAX(ts, this.parameters)
3
4  if Hash.ContainsKey(iSAX_word)
5      node = Hash.ReturnNode(iSAX_word)
6      if node is terminal
7          if SplitNode() == false
8              node.Insert(ts)
9          else
10             newnode = new internal
11             newnode.Insert(ts)
12             foreach ts in node
13                 newnode.Insert(ts)
14             end
15             Hash.Remove(iSAX_word, node)
16             Hash.Add(iSAX_word, newnode)
17         endif
18     elseif node is internal
19         node.Insert(ts)
20     endif
21 else
22     newnode = new terminal
23     newnode.Insert(ts)
24     Hash.Add(iSAX_word, newnode)
25 endif
```

The deletion function is obvious and omitted for brevity.

4.3 Approximate Search

For many data mining applications, an approximate search may be all that is required. An *i*SAX index is able to support very fast approximate searches; in particular, they only require a single disk access. The method of approximation is derived from the intuition that two similar time series are often represented by the same *i*SAX word. Given this assumption, the approximate result is obtained by attempting to find a terminal node in the index with the same *i*SAX representation as the query. This is done by traversing the index in accordance with split policies and matching *i*SAX representations at each internal node. Because the index is hierarchical and without overlap, if such a terminal node exists, it is promptly identified. Upon reaching this terminal node, the index file pointed to by the node is fetched and returned. This file will contain at least 1 and at most th time series in it. A main memory sequential scan over these time series gives the approximate search result.

In the (very) rare case that a matching terminal node does not exist, such a traversal will fail at an internal node. We mitigate the effects of non-matches by proceeding down the tree, selecting nodes whose last split dimension has a matching *i*SAX value with the query time series. If no such node exists at a given junction, we simply select the first, and continue the descent.

4.4 Exact Search

Obtaining the exact nearest neighbor to a query is both computationally and I/O intensive. To improve search speed, we use a combination of approximate search and lower bounding distance functions to reduce the search space. The algorithm for obtaining the nearest neighbor is presented as pseudo-code in Table 6.

The algorithm begins by obtaining an approximate best-so-far (BSF) answer, using approximate search as described in Section 4.3 (lines 2-3). The intuition is that by quickly obtaining an entry which is a close approximation and with small distance to the nearest neighbor, large sections of the search space can be pruned. Once a baseline BSF is obtained, a priority queue is created to examine nodes whose distance is potentially less than the BSF. This priority queue is first initialized with the root node (line 6).

Because the query time series is available to us, we are free to use its PAA representation to obtain a tighter bound than the MINDIST between two *i*SAX words. More concretely, the distance used for priority queue ordering of nodes is computed using MINDIST_PAA_*i*SAX, between the PAA representation of the query time series and the *i*SAX representation of the SAX space occupied by a node.

Given the PAA representation, T_{PAA} of a time series T and the *i*SAX representation, S_{iSAX} of a time series S, such that $|T_{PAA}| = |S_{iSAX}| = w$, $|T| = |S| = n$, and recalling that the j^{th} cardinal value of S_{iSAX} derives from a PAA value, v between two breakpoints β_L, β_U, $\beta_L < v \leq \beta_U$, $1 \leq j \leq w$ we define the lower bounding distance as:

$$\text{MINDIST\_PAA\_}i\text{SAX}(T_{PAA}, S_{iSAX}) = \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} \begin{cases} (\beta_{Li} - T_{PAAi})^2 \text{ if } \beta_{Li} > T_{PAAi} \\ (\beta_{Ui} - T_{PAAi})^2 \text{ if } \beta_{Ui} < T_{PAAi} \\ 0 \quad \text{otherwise} \end{cases}}$$

Recall that we use distance functions that lower bound the true Euclidean distance. That is, if the BSF distance is less than or equal to the minimum distance from the query to a node, we can discard the node and all descendants from the search space without examining their contents or introducing any false dismissals.

The algorithm then repeatedly extracts the node with the smallest distance value from the priority queue, terminating when either the priority queue becomes empty or an early termination condition is met. Early termination occurs when the lower bound distance we compute equals or exceeds the distance of the BSF. This implies that the remaining entries in the queue cannot qualify as the nearest neighbor and can be discarded.

If the early termination condition is not met (line 10), the node is further evaluated. In the case that the node is a terminal node, we fetch the index file from disk and compute the distance from the query to each entry in the index file, recording the minimum distance (line 14). If this distance is less than our BSF, we update the BSF (lines 16-17).

In the case that the node is an internal node or the root node, its immediate descendents are inserted into the priority queue (lines 20-23). The algorithm then repeats by extracting the next minimum node from the priority queue.

Before leaving this section, we note that we have only discussed 1NN queries. Extensions to KNN and range queries are trivial and obvious, and are omitted for brevity.

Table 6: Expediting exact search using approximate search and lower bounding distance functions

```
1   Function [IndexFile] = ExactSearch(ts)
2   BSF.IndexFile = ApproximateSearch(ts)
3   BSF.dist = IndexFileDist(ts, BSF.IndexFile)
4
5   PriorityQueue pq
6   pq.Add(root)
7
8   while !pq.IsEmpty
9       min = pq.ExtractMin()
10      if min.dist >= BSF.dist
11          break
12      endif
13      if min is terminal
14          tmp = IndexFileDist(ts, min.IndexFile)
15          if BSF.dist > tmp
16              BSF.dist = tmp
17              BSF.IndexFile = min.IndexFile
18          endif
19      elseif min is internal or root
20          foreach node in min.children
21              node.dist = MINDIST_PAA_iSAX(ts,node.iSAX)
22              pq.Add(node)
23          end
24      endif
25  end
26  return BSF.IndexFile
```

5. EXPERIMENTS

We begin by discussing our experimental philosophy. We have designed all experiments such that they are not only reproducible, but *easily* reproducible. To this end, we have built a webpage which contains all datasets and code used in this work, together with spreadsheets which contain the raw numbers displayed in all the figures [10]. In addition, the webpage contains many additional experiments which we could not fit into this work; however, we note that this paper is completely self-contained.

Experiments are conducted on an AMD Athlon 64 X2 5600+ with 3GB of memory, Windows XP SP2 with /3GB switch enabled, and using version 2.0 of the .NET Framework. All experiments used a 400GB Seagate Barracuda 7200.10 hard disk drive with the exception of the 100M random walk experiment, which required more space, there we used a 750GB Hitachi Deskstar 7K10000.

5.1 Tightness of Lower Bounds

It is important to note that the rivals to *i*SAX are other time series representations, not indexing structures such as R-Trees, VP-Trees etc. We therefore begin with a simple experiment to compare the tightness of lower bounds of *i*SAX with the other lower bounding time series representations, including DFT, DWT, DCT, PAA, CHEB, APCA and IPLA. We measure TLB, the tightness of lower bounds [9]. This is calculated as:

$$\text{TLB} = \text{LowerBoundDist}(T,S) \,/\, \text{TrueEuclideanDist}(T,S)$$

Because DWT and PAA have exactly the same TLB [9] we show one graphic for both. We randomly sample T and S (with replacement) 1,000 times for each combination of parameters. We vary the time series length [480, 960, 1440, 1920] and the number of bytes per time series available to the dimensionality reduction approach [16, 24, 32, 40]. We assume that each real valued representation requires 4 bytes per coefficient, thus they use [4, 6, 8, 10] coefficients. For *i*SAX, we hard code the cardinality to 256, resulting in [16, 24, 32, 40] symbols per word.

Recall that, for TLB, larger values are better. If the value of TLB is zero, then any indexing technique is condemned to retrieving every object from the disk. If the value of TLB is one, then there is no search, we could simply retrieve one object from disk and guarantee that we had the true nearest neighbor. Figure 6 shows the result of one such experiment with an ECG dataset.

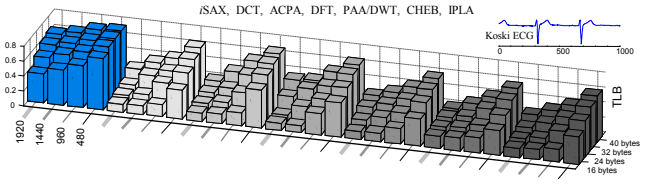

Figure 6: The tightness of lower bounds for various time series representations on the Koski ECG dataset. Similar graphs for thirty additional datasets can be found at [10]

Note that the speedup obtained is generally non-linear in TLB, that is to say if one representation has a lower bound that is twice as large as another, we can usually expect a *much* greater than two-fold decrease in disk accesses.

In a sense, it may be obvious before doing this experiment that *i*SAX will have a smaller reconstruction error, thus a tighter lower bound, and greater indexing efficiency than the real valued competitors. This is because *i*SAX is taking advantage of every bit given to it. In contrast, for the real valued approaches it is clear that the less significant bits contribute *much* less information than the significant bits. If the raw time series is represented with 4 bytes per data point, then each real valued coefficient must also have 4 bytes (recall that orthonormal transforms are merely rotations in space). This begs the question, why not quantize or truncate the real valued coefficients to save space? In fact, this is a very common idea in compression of time series data. For example, in the medical domain it is frequently done for both the wavelet [5] and cosine [3] representations. However, recall that we are not interested in compression per se. Our interest is in dimensionality reduction that allows indexing with no false dismissals. If, for the other approaches, we save space by truncating the less significant bits, then at least under the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) default policy for rounding (RoundtoNearest) it is possible the distance between two objects can *increase*, thus violating the no false dismissals guarantee. We have no doubt that an indexable bit-adjustable version of the real valued representations could be made to work, however, none exists to date.

Even if we naively coded each *i*SAX word with the same precision as the real valued approaches (thus wasting 75% of the main memory space), *i*SAX is still competitive with the other approaches; this is shown in Figure 7. Before leaving this section, we note that we have repeated these experiments with thirty additional datasets from very diverse domains with essentially the same results [10].

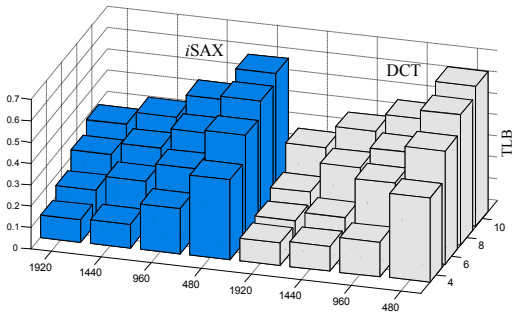

Figure 7: The experiment in the previous figure redone with the *i*SAX word length equal to the dimensionality of the real valued applications (just DCT is shown to allow a "zoom in")

5.2 Indexing Massive Datasets

We tested the accuracy of approximate search for increasingly large random walk databases of sequence length 256, containing [one, two, four, eight] million time series. We used $b = 4$, $w = 8$, and $th = 100$. This created [39,255, 57,365, 92,209, 162,340] files on disk. We generated 1,000 queries, did an approximate search, and then compared the results with the true ranking which we later obtained with a sequential scan. Figure 8 shows the results.

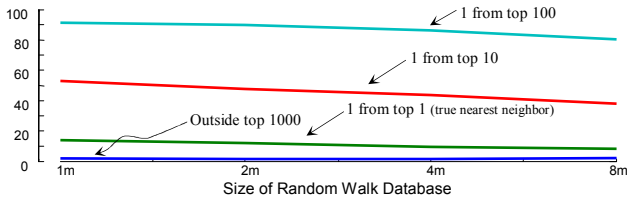

Figure 8: The percentage of cutoffs for various rankings, for increasingly large databases with approximate search

The figure tells us that when searching one million time series, 91.5% of the time approximate search returns an answer that would rank in the top 100 of the true nearest neighbor list. Furthermore, that percentage only slightly decreases as we scale to eight million time series. Likewise, again, for one million objects, more than half the time the approximate searches return an object that would rank in the top 10, and 14% of the time it returns the *true* nearest neighbor. Recall that these searches require exactly one disk access and at most 100 Euclidean distance calculations, so the average time for a query was less than a second.

We also conducted exact search experiments on 10% of the queries. For our four different sized datasets, exact search required an average of [2115.3, 3172.5, 4925.3, 7719.1] disk accesses and finished in an average time of [3.8, 5.8, 9.0, 14.1] minutes, in contrast to sequential scan, which took [71.5, 104.8, 168.8, 297.6] minutes.

To push the limits of indexing, we considered indexing 100,000,000 random walk time series of length 256. To the best of our knowledge, this is as least two orders of magnitude larger that any other dataset considered in the literature [2][4][6][13]. Since the publication of *Don Quixote de la Mancha* in the 17th century, the idiom, "*a needle in a haystack*" has been used to signify a near impossible search. If each time series in this experiment was represented by a piece of hay the size of a drinking straw, they would form a cube shaped haystack with 262 meter sides.

Because of the larger size of data, we increased *th* to 2,000, and used *w* of 16. This created 151,902 files occupying a half terabyte of disk space. The average occupancy of index files is approximately 658.

We issued ten new random walk approximate search queries. Each query was answered in an average of 1.15 seconds. To find out how good each answer was, we did a linear scan of the data to find the true rankings of the answers. Three of the queries did actually discover their true nearest neighbor, the average rank was 8, and the worst query "only" managed to retrieve its 25th nearest neighbor. In retrospect, these results are extraordinarily impressive. Faced with one hundred million objects on disk, we can retrieve only 0.0013895% of the data and find an object that is ranked the top 0.0001%. As we shall see in Sections 5.3/5.4, the extraordinary precision and speed of approximate search combined with fast exact search allows us to consider mining datasets with millions of objects.

We also conducted exact searches on this dataset; each search took an average of 90 minutes to complete, in contrast to a linear scan taking 1,800 minutes.

5.3 Time Series Set Difference

In this section, we give an example of a data mining algorithm that can be built on top of our existing indexing algorithms. The algorithm is interesting in that it uses both approximate search and exact search to compute the ultimate (exact) answer to a problem.

Suppose we are interested in contrasting two collections of time series data. For example, we may be interested in contrasting telemetry from the last Shuttle launch with telemetry from all previous launches, or we may wish to contrast the ten minutes of electrocardiograms just before a patient wakes up with the preceding seven hours of sleep.

To do this, we define the Time Series Set Difference (TSSD):

> **Definition**: Time Series Set Difference(*A*,*B*). Given two collections of time series *A* and *B*, the time series set difference is the subsequence in *A* whose distance from its nearest neighbor in *B* is maximal.

Note that we are not claiming that this is the best way to contrast two time series; it is merely a sensible definition we can use as a starting point.

We tested this definition on an electrocardiogram dataset. The data is an overnight polysomnogram with simultaneous three-channel Holter ECG from a 45 year old male subject with suspected sleep-disordered breathing. We used the first 7.2 hours of the data as the reference set *B*, and the next 8 minutes 39 seconds as the "novel" set *A*. The set *A* corresponds to the period in which the subject woke up. After indexing the data with an *i*SAX word length of 9 and a maximum threshold value of 100, we had 1,000,000 time series subsequences in 31,196 files on disk, occupying approximately 4.91GB of secondary storage. Figure 9 show the TSSD discovered.

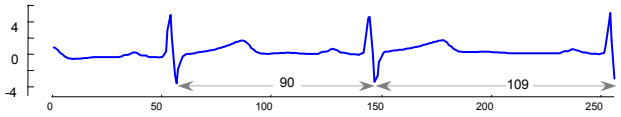

Figure 9: The Time Series Set Difference discovered between ECGs recorded during a waking cycle and the previous 7.2 hours

We showed the result to UCLA cardiologist Helga Van Herle. She noted that the p-waves in each of the full heartbeats look the same, but there is a 21.1% increase in the length of the second one. This indicated to her that this is almost certainly an example of sinus arrhythmia, where the R-R intervals are changing with the patients breathing pattern. This is likely due to slowing of the heart rate with expiration and increase of the heart rate with inspiration, given that it is well known that respiration patterns change in conjunction with changes in sleep stages [15].

An obvious naive algorithm to find the TSSD is to do 20,000 exact searches, one for each object in *A*. This requires ("only") 325,604,200 Euclidean distance calculations, but it requires approximately 5,676,400 disk accesses, for 1.04 days of wall clock time. This is clearly untenable.

We propose a simple algorithm to find the TSSD that exploits the fact that we can do both ultra fast approximate search and fast exact search. We assume that set *B* is indexed and that set *A* is in main memory. The algorithm is sketched out in Table 7.

Table 7: An outline of an algorithm to find the TSSD

```
For each time series in A, find its approximate NN in B
Place each time series in a priority queue sorted by
its NN distance in descending order.
BEGIN
  Remove the time series at the head of the queue.
  Begin an exact search for its nearest neighbor.

  If at any point during this search the best-so-far
  value becomes less than the value at the top of the
  priority queue, reinsert the time series back into
  the priority queue with the best-so-far value as the
  key, then goto BEGIN.

  Otherwise we must have found the true nearest
  neighbor and its distance is greater than the top of
  the priority queue, so we have the true TSSD. Report
  success.
```

To find the discordant heartbeats shown in Figure 9, our algorithm did 43,779 disk accesses (20,000 in the first approximate stage, and the remainder during the refinement search phase), and performed 2,365,553 Euclidean distance calculations. The number of disk accesses for a sequential scan algorithm is somewhat better; it requires only 31,196 disk reads, about 71% of what our algorithm required. However, sequential scan requires 20,000,000,000 Euclidean distance calculations, which is 8,454 times greater than our approach and would require an estimated 6.25 days to complete. In contrast, our algorithm takes only 34 minutes.

Our algorithm is much faster because it exploits the fact that that most candidates in set *A* can be quickly eliminated by very fast approximate searches. In fact, of the 20,000 objects in set *A* for this experiment, only two of them (obviously including the eventual answer) had their true nearest neighbor calculated. Of the remainder, 17,772 were eliminated based only on the single disk access made in phase one of the algorithm, and 2,226 required more than one disk access, but less than a compete nearest neighbor search.

5.4 Batch Nearest Neighbor Search

We consider another problem which can be exactly solved with a combination of approximate and exact search. The problem is that of batch nearest neighbor search. We begin with a concrete example of the problem before showing our *i*SAX-based solution.

It has long been known that all the great apes except humans have 24 chromosomes. Humans, having 23, are quite literally the odd man out. This is widely accepted to be a result of an end-to-end fusion of two ancestral chromosomes. Suppose we do not know which of the ancestral chromosomes were involved in the fusion, we could attempt to efficiently discover this with *i*SAX.

We begin by converting DNA into time series. There are several ways to do this; here we use the simple approach shown in Table 8.

Table 8: An algorithm for converting DNA to time series

```
T1 = 0;
For i = 1 to length(DNAstring)
        If  DNAstringi = A, then Ti+1 = Ti + 2
        If  DNAstringi = G, then Ti+1 = Ti + 1
        If  DNAstringi = C, then Ti+1 = Ti - 1
        If  DNAstringi = T, then Ti+1 = Ti - 2
End
```

We converted Contig NT_005334.15 of the human chromosome 2 to time series in this manner, and then indexed all subsequences of length 1024 using a sliding window. There are a total of 11,246,491 base pairs (approximately 2,100 pages of DNA text

written in this paper's format) and a total of 5,622,734 time series subsequences written to disk.

We converted 43 randomly chosen subsequences of length 1024 of chimpanzee's (Pan troglodytes) DNA in the same manner. We made sure that the 43 samples included at least one sample from each of the chimps 24 chromosomes.

We performed a search to find the chimp subsequence that had the nearest nearest-neighbor in the human reference set. Figure 10 shows the two subsequences plotted together. Note that while the original DNA strings are very similar, they are not identical.

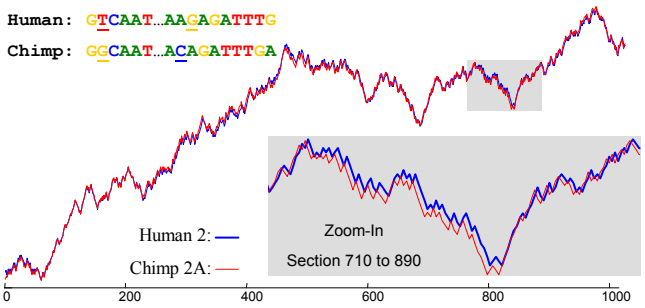

Figure 10: Corresponding sections of human and chimpanzee DNA

Once again, this is a problem where a combination of exact and approximate search can be useful. To speed up the search we use *batch nearest neighbor search*. We define this as the search for the object O in a (relatively small) set *A*, which has the smallest nearest neighbor distance to an object in a larger set *B*. Note that to solve this problem, we really only need *one* exact distance, for object O, to be known. For the remaining objects in *A*, it suffices to know that a lower bound on their nearest neighbors is greater than the distance from O to its nearest neighbor. With this in mind, we can define an algorithm which is generally much faster than performing exact search for each of the objects in *A*. Table 9 outlines the algorithm.

Table 9: Batch Nearest Neighbor Algorithm

```
For each time series in A, find its approximate NN in B
Record the object O, with the smallest approximate distance.

Do an exact search for its nearest neighbor using
ExactSearch(O) (cf Table 6). Record the distance as O.dist
-----
For all remaining objects in A
    Do ExactSearch(ts), as in Table 6 but change lines
2-3 such that BSF.dist is initialized to O.dist

If ExactSearch terminates with null, then this object
could not have been closer to its nearest neighbor than O.

Else This object Onew was closer to its nearest neighbor
than O. So O = Onew
End
```

We can see this algorithm as an anytime algorithm [17]. After the first phase, our algorithm has an approximate answer that we can examine. As the algorithm continues working in the background to confirm or adjust that answer, we can evaluate the current answer and make a determination of whether to terminate or allow the algorithm to persist.

In this particular experiment, the first phase of the algorithm returns an answer (which we later confirm to be the exact solution) in just 12.8 seconds, finding that the randomly chosen substring of chimp chromosome 2A, beginning at 7,582 of Contig NW_001231255 is a stunningly close match to the substring beginning at 999,645 of the Contig NT_005334.15 of human chromosome 2. The full algorithm terminates in 21.8 minutes. In contrast, a naive sequential scan takes 13.54 hours.

6. CONCLUSIONS

We introduced *i*SAX, a representation that supports indexing of massive datasets, and have shown it can index up to one hundred million time series. We have also provided examples of algorithms that use a combination of approximate and exact search to ultimately produce exact results on massive datasets. Other time series data mining algorithms such as motif discovery, density estimation, discord discovery, and clustering can similarly take advantage of combining both types of search. We plan to consider such problems in future work.

7. REFERENCES

[1] André-Jönsson, H. and Badal, D. Z. 1997. Using Signature Files for Querying Time-Series Data. In *Proc 1st PKDD*. p. 211-220.

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

[3] Batista, L. V., Melcher, E. U. K., Carvalho, L.C. 2001. Compression of ECG signals by optimized quantization of discrete cosine transform coefficients. *Medical Engineering & Physics*. 23, 2, 127-134.

[4] Cai, Y. and Ng, R. 2004. Indexing spatio-temporal trajectories with Chebyshev polynomials. In *Proc of the ACM SIGMOD*. p. 599-610.

[5] Chen, J. and Itoh, S. 1998. A wavelet transform-based ECG compression method guaranteeing desired signal quality. *IEEE Transactions on Bio Engineering*. 45, 12, 1414-1419.

[6] Faloutsos, C., Ranganathan, M., & Manolopoulos, Y. (1994). Fast subsequence matching in time-series databases. In Proc. ACM SIGMOD Conf., Minneapolis.

[7] Huang, Y. and Yu, P. S. (1999). Adaptive query processing for time-series data. In *Proc of the 5th ACM SIGKDD*. p. 282-286.

[8] Jeffery, C. (2005). Synthetic Lightning EMP Data. *http://public.lanl.gov/eads/datasets/emp/index.html*

[9] Keogh, E., Chakrabarti, K., Pazzani, M.J, and S. Mehrotra. (2001) Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. KAIS 3(3), 263–286.

[10] Keogh, E., and Shieh, J. (2008). *i*SAX home page *www.cs.ucr.edu/~eamonn/iSAX/iSAX.html*

[11] Keogh, E. (2008). *www.cs.ucr.edu/~eamonn/SAX.htm*

[12] Lin, J., Keogh, E., Wei, L. and Lonardi, S. (2007). Experiencing SAX: a novel symbolic representation of time series. Data Min. Knowl. Discov. 15(2): 107-144.

[13] Megalooikonomou, V., Wang, Q., Li, G., and Faloutsos, C. 2005. A Multiresolution Symbolic Representation of Time Series. In *Proceedings of the 21st ICDE*. p 668-679.

[14] Ratanamahatana, C. A. and Keogh, E. Three Myths about Dynamic Time Warping. In *Proc of SIAM International Conference on Data Mining* (SDM '05), pp 506-510, 2005.

[15] Scholle, S. and Schäfer. T. (1999) Atlas of states of sleep and wakefulness in infants and children. *Somnologie - Schlafforschung und Schlafmedizin* 3:4, 163.

[16] Steinbach, M., Tan, P., Kumar, V., Klooster, S., and Potter, C. 2003. Discovery of climate indices using clustering. *In Proc of the Ninth ACM SIGKDD*. p. 446-455.

[17] Xi, X., Keogh, E., Shelton, C., Wei, L., and Ratanamahatana, C. A. 2006. Fast time series classification using numerosity reduction. In *Proc of the 23rd ICLM*. p. 1033-1040.