DPiSAX: Massively Distributed Partitioned iSAX

Djamel-Edine Yagoubi[1,*] Reza Akbarinia[1] Florent Masseglia[1] Themis Palpanas
[1]Inria & LIRMM, Montpellier, France Paris Descartes University
Djamel-Edine.Yagoubi@inria.fr Reza.Akbarinia@inria.fr Florent.Masseglia@inria.fr themis@mi.parisdescartes.fr

***Abstract*—Indexing is crucial for many data mining tasks that rely on efficient and effective similarity query processing. Consequently, indexing large volumes of time series, along with high performance similarity query processing, have became topics of high interest. For many applications across diverse domains though, the amount of data to be processed might be intractable for a single machine, making existing centralized indexing solutions inefficient. We propose a parallel indexing solution that gracefully scales to billions of time series, and a parallel query processing strategy that, given a batch of queries, efficiently exploits the index. Our experiments, on both synthetic and real world data, illustrate that our index creation algorithm works on 1 billion time series in less than 2 hours , while the state of the art centralized algorithms need more than 5 days. Also, our distributed querying algorithm is able to efficiently process millions of queries over collections of billions of time series, thanks to an effective load balancing mechanism.**

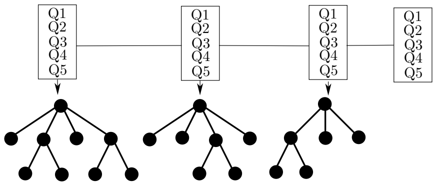
I. Introduction

Nowadays, individuals are able to monitor various indicators for their personal activities (*e.g.*, through smart-meters or smart-plugs for electricity or water consumption), or professional activities (*e.g.*, through the sensors installed on plants by farmers). Sensors technology is also improving over time and the number of sensors is increasing, *e.g.*, in finance and seismic studies. This results in the production of large and complex data, usually in the form of time series (or *TS* in short) [14], [13], [12], that challenge knowledge discovery. With such complex and massive sets of time series, fast and accurate similarity search is a key to perform many data mining tasks like Shapelets, Motifs Discovery, Classification or Clustering [14].

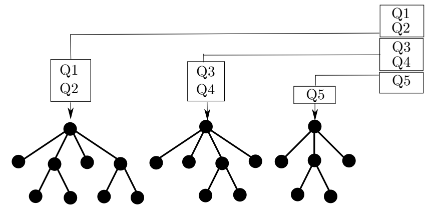
In order to improve the performance of such similarity queries, indexing is one of the most popular techniques [6], which has been successfully used in a variety of settings and applications [7], [16], [3], [17], [5], [20]. Although recent studies have shown that in certain cases sequential scans can be very efficient [14], [18], such techniques are only advantageous when the database consists of a single, long time series, and query answers are small subsequences of this long time series. Such approaches, however, are not beneficial in the general case of querying a mixed database of many small time series [21] (*e.g.*, in neuroscience, or manufacturing applications [12]), which is the focus of this study. Therefore, indexing is required in order to efficiently support data exploration tasks, which involve ad-hoc queries.

*The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732051.

This work has been performed in the context of the Computational Biology Institute (www.ibc-montpellier.fr)



(a) *Straightforward implementation: the batch of queries is duplicated on all the computing nodes.*



(b) *Ideal distribution of time series in the index nodes: each query is sent only to the relevant partition.*

Fig. 1: *Straightforward Vs. partitioned strategies for TS indexing and querying. Load balancing is a major lever.*

In this work, we focus on the problem of similarity search in such massive sets of time series by means of scalable index construction and use. Unfortunately, making an index over billions of time series by using traditional centralized approached is highly time consuming. Moreover, a naive construction of the index on the parallel environment may lead to poor querying performances. This is illustrated in Figure 1 where the time series dataset is naively split on the W distributed nodes (Figure 1a). In this case, a batch of queries B has to be duplicated and sequentially processed on each node. By means of a dedicated strategy where each query in B could be oriented to the right partition (*i.e.*, the partition that must correspond to the query) the querying work load can be significantly reduced (Figure 1b shows an ideal case where B is split in W subsets and really processed in parallel). Our goal is to reach such an ideal distribution of index construction and query processing in massively distributed environments.

We propose a parallel solution to construct the state of the art iSAX-based index [5] over billions of time series by making the most of the parallel environment by carefully distributing the work load. Our solution takes advantage of the computing power of distributed systems by using parallel frameworks such as MapReduce or Spark [19]. Our contributions are as follows:

- We propose a parallel index construction algorithm that takes advantage of distributed environments to efficiently build iSAX-based indices over very large volumes of time series. We implemented our index construction and query processing algorithm, and evaluated their performance over large volumes of data (up to 4 billion time series of length 256, for a total volume of 6 Terabytes). Our experiments illustrate the performance of our algorithm with an indexing time of less than 2 hours for more than 1 billion time series, while the state of the art centralized algorithm needs more than 5 days.
- We also propose a parallel query processing algorithm that, given a query, exploits the available processors of the distributed system to answer the query in parallel by using the constructed parallel index. As illustrated by our experiments, and owing to our distributed querying strategy, our approach is able to process 10M queries in less than 140 seconds, while the state of the art centralized algorithm needs almost 2300 seconds.

The rest of the paper is organized as follows. In Section II, we define the problem we address in the paper and present the related background. In Section III, we describe the details of our parallel index construction and query processing algorithm. In Section IV, we present a detailed experimental evaluation to verify the effectiveness of our approach. In Section V, we discuss the related work. We conclude in VI.

II. Problem Definition and Background

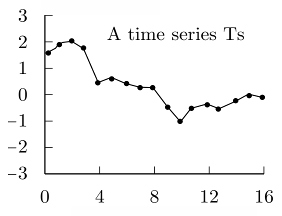
A time series X is a sequence of values $X = \{x_1, ..., x_n\}$. We assume that every time series has a value at every timestamp $t = 1, 2, ..., n$. The length of X is denoted by $|X|$. Figure 2a shows a time series of length 16, which will be used as running example throughout this paper.

A. iSAX Representation

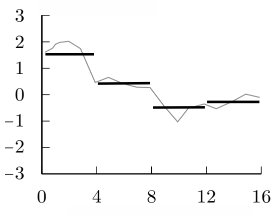
Given two time series $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_m\}$ such that $n = m$, the Euclidean distance between X and Y is defined as [7]: $ED(X,Y) = \sqrt{\sum_{n}^{i=1}(x_i - y_i)^2}$. The Euclidean distance is one of the most straightforward similarity measurement methods used in time series analysis. In this work, we use it as the distance measure.

For very large time series databases, it is important to estimate the distance between two time series very quickly. There are several techniques, providing lower bounds by segmenting time series. Here, we use a popular method, called indexable Symbolic Aggregate approXimation (iSAX) representation [15], [16]. The iSAX representation will be used to represent time series in our index.

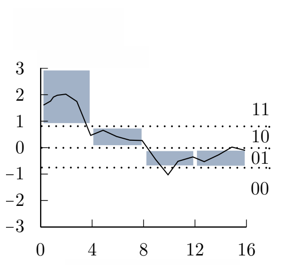
The iSAX representation extends the SAX representation [11]. This latter representation is based on the PAA representation [10] which allows for dimensionality reduction while providing the important lower bounding property as we will show later. The idea of PAA is to have a fixed segment size, and minimize dimensionality by using the mean values on each segment. Example 1 gives an illustration of PAA.



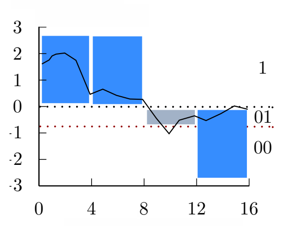
(a) *A time series X of length 16*



(b) *A PAA representation of X, with 4 segments*



(c) *A SAX representation of X, with 4 segments and cardinality 4,* $[11, 10, 01, 01]$.



(d) *An iSAX representation of X, with 4 segments and different cardinalities* $[1_2, 1_2, 01_4, 0_2]$.

Fig. 2: *A time series X is discretized by obtaining a PAA representation and then using predetermined break-points to map the PAA coefficients into SAX symbols*

Example 1: Figure 2b shows the PAA representation of X, the time series of Figure 2a. The representation is composed of $w = |X|/l$ values, where l is the segment size. For each segment, the set of values is replaced with their mean. The length of the final representation w is the number of segments (and, usually, $w << |X|$).

The SAX representation takes as input the reduced time series obtained using PAA. It discretizes this representation into a predefined set of symbols, with a given cardinality, where a symbol is a binary number. Example 2 gives an illustration of the SAX representation.

Example 2: In Figure 2c, we have converted the time series X to SAX representation with size 4, and cardinality 4 using the PAA representation shown in Figure 2b. We denote SAX(X) = [11, 10, 01, 01].

The iSAX representation uses a variable cardinality for each symbol of SAX representation, each symbol is accompanied by a number that denotes its cardinality. We defined the iSAX representation of time series X as $iSAX(X)$ and we call it the iSAX word of the time series X. For example, the iSAX word shown in Figure 2d can be written as $iSAX(X) = [1_2, 1_2, 01_4, 0_2]$.

Using a variable cardinality allows the iSAX representation to be indexable. We can build a tree index as follows. Given a cardinality b, an iSAX word length w and leaf capacity th, we produce a set of b^w children for the root node, insert the time series to their corresponding leaf, and gradually split the leaves by increasing the cardinality by one character if the number of time series in a leaf node rises above the given threshold th.

Note that previous studies have shown that the iSAX index is robust with respect to the choice of parameters (word length, cardinality, leaf threshold) [16], [5], [21]. Moreover, it can also be used to answer queries with the Dynamic Time Warping (DTW) distance, through the use of the corresponding lower bounding envelope [9].

B. Similarity Queries

The problem of similarity queries is one of the main problems in time series analysis and mining. In information retrieval, finding the k nearest neighbors (k-NN) of a query is a fundamental problem. In this section, we define k nearest neighbors based queries.

Definition 1: (APPROXIMATE k NEAREST NEIGHBORS) Given a set of time series D, a query time series Q, and $\epsilon > 0$. We say that $R = AppkNN(Q, D)$ is the approximate k nearest neighbors of Q from D, if $ED(a, Q) \leq (1 + \epsilon)ED(b, Q)$. Where a is the k^{th} nearest neighbor from R and b is the true k^{th} nearest neighbor.

C. Spark

For implementing our parallel algorithm we use Spark [19], which is a parallel programming framework aiming to efficiently process large datasets. This programming model can perform analytics with in-memory techniques to overcome disk bottlenecks. Unlike traditional in-memory systems, the main feature of Spark is its distributed memory abstraction, called resilient distributed datasets (*RDD*), that is an efficient and fault-tolerant abstraction for distributing data in a cluster. With RDD, the data can be easily persisted in main memory as well as on the hard drive. Spark is designed to support the execution of iterative algorithms.

D. Problem Definition

The problem we address is as follows. Given a (potentially huge) set of time series, find the results of approximate k-NN queries as presented in definition 1, by means of an index and query processing performed in parallel.

III. DISTRIBUTED PARTITIONED iSAX

In this section, we present a novel parallel partitioned index construction algorithm, called *DPiSAX*, along with very fast parallel query processing techniques.

Our approach is based on a sampling phase that allows anticipating the distribution of time series among the computing nodes. Such anticipation is mandatory for an efficient query processing, since it will allow, later on, to decide what partition contains the time series that actually correspond to the query. To do so, we first extract a sample from the time series dataset, and analyze it in order to decide how to distribute the time series in the splits, according to their iSAX representation.

TABLE I: *A sample S of 8 time series converted to iSAX representations with iSAX words of length 2*

Time series	iSAX words	Time series	iSAX words
TS_1	$\{01, 00\}$	TS_5	$\{00, 10\}$
TS_2	$\{00, 01\}$	TS_6	$\{01, 11\}$
TS_3	$\{01, 01\}$	TS_7	$\{10, 00\}$
TS_4	$\{00, 00\}$	TS_8	$\{10, 01\}$

A. Sampling

In Distributed Partitioned iSAX, our index construction combines two main phases which are executed one after the other. First, the algorithm starts by sampling the time series dataset and creates a partitioning table. Then, the time series are partitioned into groups using the partitioning table. Finally, each group is processed to create an iSAX index for each partition.

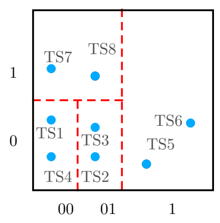
More formally, our sampling is done as follows. Given a number of partitions P and a time series dataset D, the algorithm takes S sample time series of size L from D using stratified sampling, and distributes them among the W available workers. Each worker takes S/W time series and emits its iSAX words $SWs = \{iSAX(ts_i), i = 1, ..., L\}$. The master collects all the workers' iSAX words and performs the partitioning algorithm accordingly. In the following, we describe the partitioning method that enable separating the dataset into non-overlapping subsets based on iSAX representations.

B. Partitioning Algorithm

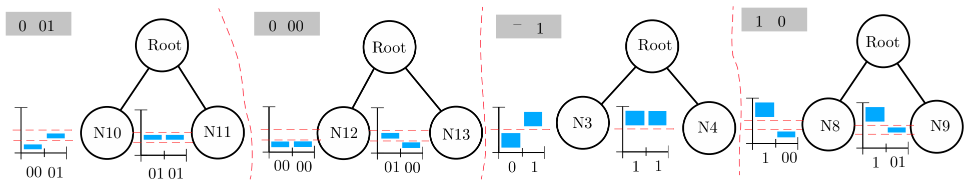
Here, our partitioning paradigm considers the splitting power of each bit in the iSAX symbols, before actually splitting the partition. As in the basic approach, the biggest partition is considered for splitting at each step of the partitioning process. The main difference is that we don't use the first bit of the n^{th} symbol for splitting the partition. Instead, we look for all bits (whatever the symbol) with the highest probability to equally distribute the time series of the partition among the two new sub-partitions that will be created. To this effect, we compute for each segment the $\mu \pm \sigma$ interval, where μ is the mean and σ is the standard deviation, and we examine for each segment if the break-point of the additional bit (*i.e.*, the bit used to generate the two new partitions) lies within the interval $\mu \pm \sigma$. From the segments for which this is true, we choose the one having μ closer to the break-point.

In order to illustrate this, let us consider the blue boxes of the diagrams in Figure 3b. We choose the biggest blue box that ensures the best splitting by considering the next break-point.

Example 3: Let's consider Table I, where we use iSAX words of length two to represent the time series of a sample S. Suppose that we need to generate four partitions. To generate four partitions, we compute the $\mu \pm \sigma$ interval for the first segment and the second segment, and choose the first bit of the second segment to define two partitions. The first partition contains all the time series having their second segment in iSAX word starting with 0, and the second partition contains the time series having their second segment in iSAX word



(a) *Partitioning according to DPiSAX.*



(b) *DPiSAX indexes after partitioning and indexing. The partitioning principle of DPiSAX allows better balance.*

Fig. 3: *The result of the partitioning algorithm on sample S (from Table I) into four partitions.*

starting with 1. We obtain two partitions: "0" and "1". The biggest partition is "0" (*i.e.*, the one containing time series $TS1$ to $TS4$, $TS7$ and $TS8$). We compute the $\mu \pm \sigma$ interval for all segment over all the time series in this partition. Then, the partition is split again, according to the first bit of the first symbol. We now have the following partitions: from the first step, partition "1", and from the second step, partitions "00", and "10". Now, partition "00" is the biggest one. This partition is split for the third time, according to the second bit of the first symbol and we obtain four partitions. Figure 3a shows the obtained partitions and Figure 3b shows the indexes obtained with these partitions.

C. Index Construction

DPiSAX, our parallel index construction, sequentially splits the dataset for distribution into partitions. Then each worker builds an independent iSAX index on its partition, with the iSAX representations having the highest possible cardinalities. Representing each time series with iSAX words of high cardinalities allows us to decide later what cardinality is really needed, by navigating "on the fly" between cardinalities. The word of lower cardinality being obtained by removing the trailing bits of each symbol in the word of higher cardinality. The output of this phase, with a cluster of W nodes, is a set of W iSAX indexes built on each split.

D. Query Processing

Given a collection of queries Q, in the form of time series, and the index constructed in the previous section for a database D, we consider the problem of finding time series that are similar to Q in D, as presented in definition 1. (Due to lack of space, we omit the discussion of exact query answering to future work.)

Given a batch B of queries, the master node identify the right partition where the index is stored and send the corresponding query by using its iSAX words. Then, we send each query to the partition that has the same iSAX word as the query. Each worker uses its local index to retrieve time series that correspond to each query $Q \in B$, according to the approximate k-NN criteria. On each local index, the approximate search is done by traversing the local index to the terminal node that has the same iSAX representation as the query. The target terminal node contains at least one and at most th iSAX words, where th is the leaf threshold. A main memory sequential scan over these iSAX words is performed in order to obtain the k nearest neighbors using the Euclidean distance.

TABLE II: *Default parameters*

Parameters	Value	Parameters	Value
iSAX word length	8	Leaf capacity	1,000
Basic cardinality	2	Number of machines	32
Maximum cardinality	512	Sampling fraction	10%

IV. Performance Evaluation

In this section, we report experimental results that show the quality and the performance of DPiSAX for indexing time series.

The parallel experimental evaluation was conducted on a cluster of 32 machines, each operated by Linux, with 64 Gigabytes of main memory, Intel Xeon CPU with 8 cores and 250 Gigabytes hard disk. The iSAX2+ approach was executed on a single machine with the same characteristics.

We compare our solution to two state of the art baselines: the most efficient centralized version of iSAX index (*i.e.*, iSAX2+ [5]), and Parallel Linear Search (PLS), which is a parallel version of the UCR Suite fast sequential search (with all applicable optimizations in our context: no computation of square root, and early abandoning) [14].

Our experiments are divided into two sections. In Section IV-B, we measure the index construction times with different parameters. In Section IV-C, we focus on the query performance of our approach.

Reproductibility: we implemented our approach on top of Apache-Spark [19], using the Java programming language. The iSAX2+ index is also implemented with Java. Our code is available at http://djameledine-yagoubi.info/projects/DPiSAX/.

A. Datasets and Settings

We carried out our experiments on two real world and synthetic datasets, up to 6 Terabytes and 4 billion series. The first real world data represents seismic time series collected from the IRIS Seismic Data Access repository [1]. After preprocessing, it contains 40 millions time series of 256 values, for a total size of 150Gb. The second real world data is the TexMex corpus [8]. It contains 1 Billion time series (SIFT feature vectors) of 128 points each (derived from 1 Billion images). Our synthetic datasets are generated using a Random Walk principle, each data series consisting of 256 points. At each time point the generator draws a random number from a Gaussian distribution N(0,1), then adds the value of the last number to the new number. This type of generator has been widely used in the past. [2], [7], [3], [15], [4], [5], [20]. Table II shows the default parameters (unless otherwise specified in the text) used for each approach. The iSAX word length, PAA size, leaf capacity, basic cardinality, and maximum cardinality were chosen to be optimal for iSAX, which previous works [15], [16], [4], [5], [20] have shown to work well across data with very different characteristics.

B. Index Construction Time

In this section, we measure the index construction time in DPiSAX and compare it to the construction time of the iSAX2+ index.

Figure 4 reports the index construction times for all approaches on our Random Walk dataset. The index construction time increases with the number of time series for all approaches. This time is much lower in the case of DPiSAX, than that of the centralized iSAX2+. On 32 machines, and for a dataset of one billion time series, DPiSAX builds the index in 65 minutes , while the iSAX2+ index is built in more than 5 days on a single node.

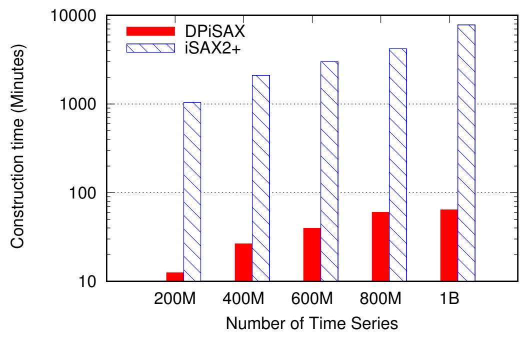


Fig. 4: *Logarithmic scale. Construction time as a function of dataset size. DPiSAX is run on a cluster of 32 nodes. iSAX2+ is run on a single node. With 1 billion Random Walk TS, iSAX2+ needs 5 days and our distributed algorithm needs less than 2 hours.*

Figure 5 illustrates the parallel speed-up of our approach on the Random Walk dataset. The results show a near optimal gain for DPiSAX.

Figure 6 reports the performance gains of our parallel approach when compared to the centralized version of iSAX2+ on our synthetic and real datasets. The results show that DPiSAX is 40-120 times faster than iSAX2+. We observe that the performance gain depends on the dataset size in relation to the number of Spark nodes used in the deployment. Note that the time Spark needs to deploy on 32 nodes is accounted for in our measurements. Thus, given the very short time needed to construct the DPiSAX index on the seismic dataset (420 seconds), the proportion of the time taken by the Spark

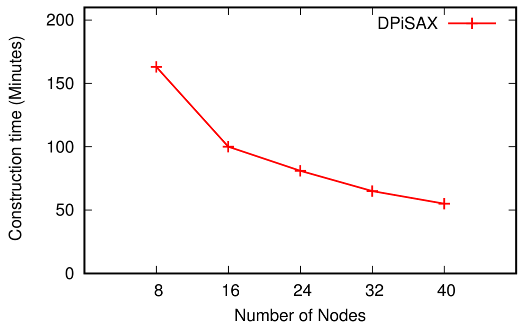


Fig. 5: *Construction time as a function of cluster size. DPiSAX has a near optimal parallel speed-up. With 1 billion TS from the Random Walk dataset.*

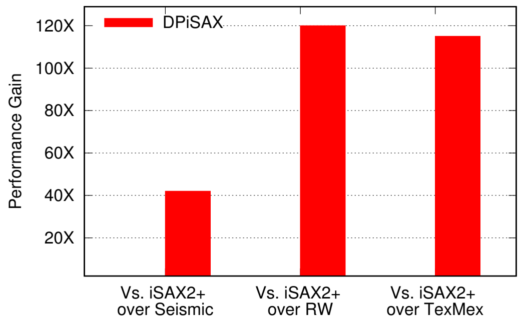


Fig. 6: *Performance gain on iSAX2+ in construction time, over seismic (40 millions TS), Random Walk (RW, 1 billion TS) and TexMex (1 billion TS), with a cluster of 32 nodes.*

deployment, when compared to index construction, is higher than for the much larger Random Walk dataset.

C. Query Performance

We evaluate the querying performance of our algorithm, and compare it to that of iSAX2+. We use our synthetic data, and generate Random Walk queries with the same distribution as described in Section IV-A.

Figure 7 compares the cumulative query answering time of our parallel approach to that of iSAX2+, for answering approximate k nearest neighbors queries with a varying size of

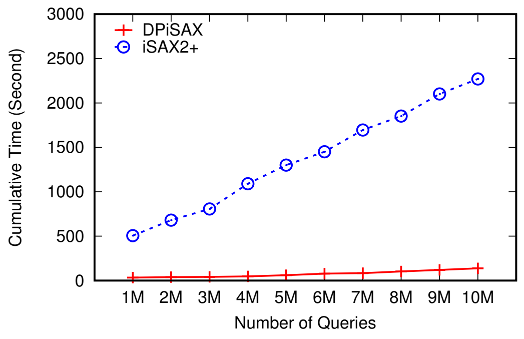


Fig. 7: *Cumulative query answering time (Approximate 10-NN). DPiSAX on a cluster of 32 nodes, iSAX2+ on a single node.*

query batch. We observe that the time performance of DPiSAX is better than that of the iSAX2+ by a factor of up to 16. Note that the total time to answer 10 millions queries is 2270 sec for iSAX2+ and only 138 sec for DPiSAX.

V. Related Work

In the context of time series data mining, several techniques have been developed and applied to time series data, *e.g.*, clustering, classification, outlier detection, pattern identification, motif discovery, and others. The idea of indexing time series is relevant to all these techniques. Note that, even though several databases have been developed for the management of time series (such as Informix Time Series[1], InfluxDB[2], OpenTSDB[3], and DalmatinerDB[4] based on RIAK), they do not include similarity search indexes, focusing on (temporal) SQL-like query workloads. Thus, they cannot efficiently support similarity search queries, which is the focus of our study.

In order to speed up similarity search, different works have studied the problem of indexing time series datasets, such as Indexable Symbolic Aggregate approXimation (iSAX) [15], [16], iSAX 2.0 [4], [5], iSAX2+ [5], Adaptive Data Series Index (ADS Index) [20] and Dynamic Splitting Tree (DSTree) [17]. The iSAX index family (iSAX 2.0, iSAX2+, ADS Index) is based on SAX representation [11] of time series, which is a symbolic representation for time series that segments all time series into equi-length segments and symbolizes the mean value of each segment. As an index structure specifically designed for ultra-large collections of time series, iSAX 2.0 proposes a new mechanism and also algorithms for efficient bulk loading and node splitting policy, wich is not supported by iSAX index. In [5], the authors propose two extensions of iSAX 2.0, namely iSAX 2.0 Clustered and iSAX2+. These extensions focus on the efficient handling of the raw time series data during the bulk loading process, by using a technique that uses main memory buffers to group and route similar time series together down the tree, performing the insertion in a lazy manner. In addition to that, DSTree based on extension of APCA representation, called EAPCA [17] segments time series into variable length segment. Unlike iSAX which only supports horizontal splitting, and only the mean values can be used in splitting, the DSTree uses multiple splitting strategies. All these indexes have been developed for a centralized environment, and cannot scale up to very high volumes of time series.

In this paper, we propose a parallel solution that takes advantage of distributed environments to efficiently build iSAX-based indices over billions of time series, and to query them in parallel with very small running times. To the best of our knowledge, this is the first paper that proposes such a solution.

[1]https://www.ibm.com/developerworks/topics/timeseries
[2]https://influxdata.com/
[3]http://opentsdb.net/
[4]https://dalmatiner.io/

VI. Conclusions

We proposed DPiSAX, a novel and efficient parallel solution to index and query billions of time series. We evaluated the performance of our solution over large volumes of real world and synthetic datasets (up to 4 billion time series, for a total volume of 6TBs). The experimental results illustrate the excellent performance of DPiSAX (*e.g.*, an indexing time of less than 2 hours for more than one billion time series, while the state of the art centralized algorithm needs several days). The results also show that the distributed querying algorithm of DPiSAX is able to process millions of similarity queries over collections of billions of time series with very fast execution times (*e.g.*, 140s for 10M queries), thanks to our load balancing mechanism. Overall, the experimental results show that by using our parallel technique, the indexing and mining of very large volumes of time series can now be done in very small execution times, which are impossible to achieve using traditional centralized approaches.

References

[1] Iris, seismic data access. http://ds.iris.edu/data/access/.

[2] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Int. Conf.on FODO*, 1993.

[3] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: Efficient time series search and retrieval. In *EDBT*, 2008.

[4] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. isax 2.0: Indexing and mining one billion time series. In *ICDM Conf.*, pages 58–67, 2010.

[5] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with i SAX2+. *Knowl. Inf. Syst.*, 2014.

[6] Philippe Esling and Carlos Agon. Time-series data mining. *ACM Comput. Surv.*, 45(1):12:1–12:34, December 2012.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SigRec*, 23(2):419–429, 1994.

[8] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *ICASSP* , 2011.

[9] Eamonn J. Keogh. Exact indexing of dynamic time warping. In *VLDB*, 2002.

[10] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *SIGMOD*, 2003.

[11] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing sax: A novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 2007.

[12] Themis Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Record*, 44(2):47–52, 2015.

[13] Themis Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM*, 2016.

[14] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, 2012.

[15] J. Shieh and E. Keogh. isax: Indexing and mining terabyte sized time series. In *KDD Conf.*, pages 623–631, 2008.

[16] J. Shieh and E. Keogh. isax: Disk-aware mining and indexing of massive time series datasets. *DMKD*, 19(1):24–57, 2009.

[17] Yang W., Peng W., Jian P., Wei W., and Sheng H. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 2013.

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

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[20] K Zoumpatianos, S Idreos, and T Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD Conf.*, 2014.

[21] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. ADS: the adaptive data series index. *VLDB J.*, 25(6):843–866, 2016.