Massively Distributed Time Series Indexing and Querying

Djamel-Edine Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Themis Palpanas

Massively Distributed Time Series Indexing and Querying

Djamel-Edine Yagoubi[1], $RezaAkbarinia^1$, $FlorentMasseglia^1$, $ThemisPalpanas^2$

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 (or high-dimensional vectors, in general), 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 4 billion time series in less than 5 hours, while the state of the art centralized algorithms do not scale and have their limit on 1 billion time series, where they 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.

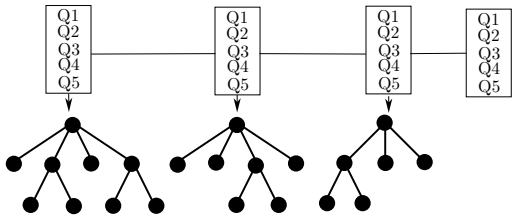
Index Terms—Time Series, Parallel Indexing, Distributed Querying

1 Introduction

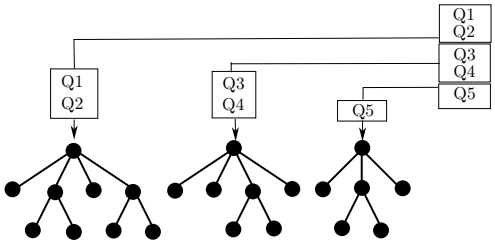
Nowadays individuals are able to monitor various indicators for their personal activities (*e.g.*, through smart-meters or smart-plugs for electricity or water consumption), or professional activities (*e.g.*, through the sensors installed on plants by farmers). Sensors technology is also improving over 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) [9], [12], [17], [20], [21], [25], [26], [27], [29], [34] 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 [19], [25], [39].

In order to improve the performance of such similarity queries, indexing is one of the most popular techniques [6], [7], which has been successfully used in a variety of settings and applications [2], [4], [8], [14], [18], [31], [32], [37]. Although recent studies have shown that in certain cases sequential scans can be very efficient [25], [35], 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 [38] (*e.g.*, in neuroscience, or manufacturing applications [21]), which is the focus of this study. Therefore, indexing is required in order to efficiently support data exploration tasks, which involve ad-hoc queries.

In this work, we focus on the problem of similarity search in such massive sets of time series[1] by means of scalable index construction and use. Unfortunately, making an index over billions of time series by using traditional centralized approaches is highly time consuming [22], [23]. Moreover, a naive construction of the index on the parallel environment may lead to poor querying performances. This is illustrated in Figure 1, where an index is computed and stored on a distributed file system. The index is in the form of a tree, where each leaf contains time series (or id/addresses of time series on the disk). We explain in



(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.*

- *1: Inria - University of Montpellier - Lirmm, France*
  E-mail: Djamel-Edine.Yagoubi@inria.fr, Reza.Akbarinia@inria.fr, Florent.Masseglia@inria.fr
- *2: Paris Descartes University, France*
  E-mail: themis@mi.parisdescartes.fr

1. We note that the proposed techniques are also applicable for high-dimensional vectors.

details in Section 2.1 how to build and query such an index. Now, let us consider that the time series dataset is naively split on the W distributed nodes (Figure 1a). Then, for a new query q, we don't know what split may contain the best answer to that query (*i.e.*, what time series, in the distributed dataset, is the most similar to q). This is not an issue with one query. But when we deal with a batch of queries, then the parallel computing power is just under exploited by such a naive approach. Basically, a batch of queries B has to be duplicated and sequentially processed on each node. However, 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 [4] 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 [36]. We provide dedicated strategies and algorithms for a deep combination of parallelism and indexing techniques, for better query performances.

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 (or high-dimensional vectors, in general).
- We implemented our index construction and query processing algorithms, and evaluated their performance over large volumes of data (up to 4 billion series, for a total volume of 6 Terabytes), using both synthetic and real data with sequences and vectors. Our experiments illustrate the benefits of our algorithm with an indexing time of less than 2 hours for more than 1 billion 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 this paper[2] is organized as follows. In Section 2, we define the problem we address in the paper and present the related background. In Section 3 and Section 4, we describe the details of our parallel index construction and query processing algorithms. In Section 5, we present a detailed experimental evaluation to verify the effectiveness of our approach. In Section 6, we discuss the related work. Finally, we conclude in 7.

2. A preliminary version of this work has appeared elsewhere [33].

2 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.

2.1 iSAX Representation

Given two time series (or vectors) of real numbers, $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_m\}$ such that $n = m$, the Euclidean distance between X and Y is defined as [8]: $ED(X, Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$. 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 [30], [31]. The iSAX representation will be used to represent time series in our index.

The iSAX representation extends the SAX representation [16]. This latter representation is based on the PAA representation [15] 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.

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, $w << |X|$).

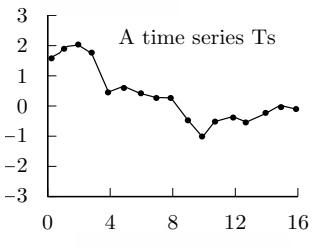
The SAX representation takes as input the reduced time series obtained using PAA. It discretizes this representation into a predefined set of symbols, with a given cardinality, where a symbol is a binary number. Example 2 gives an illustration of the SAX representation.

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].

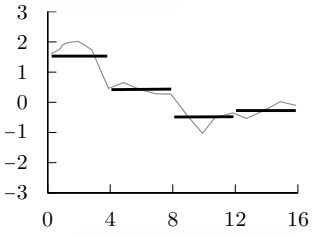
The iSAX representation uses a variable cardinality for each symbol of SAX representation, each symbol is accompanied by a number that denotes its cardinality. We defined the iSAX representation of time series X as $iSAX(X)$ and we call it the iSAX word of the time series X. For example, the iSAX word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

The lower bounding approximation of the Euclidean distance for iSAX representation $iSAX(X) = \{x'_1, ..., x'_w\}$ and $iSAX(Y) = \{y'_1, ..., y'_w\}$ of two time series X and Y is defined as:

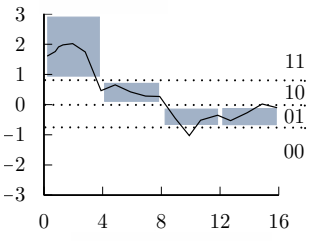
$$MINDIST(iSAX(X), iSAX(Y)) = \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w}(dist(x'_i, y'_i))^2}$$



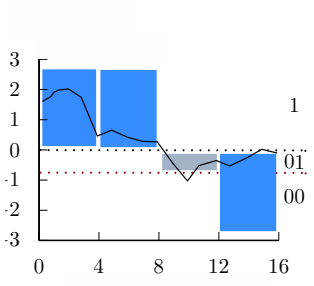
(a) *A time series X of length 16*



(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. Here, the symbols are given in binary notation, where 00 is the first symbol, 01 is the second symbol, etc. The time series of Figure 2a in the representation of Figure 2d is [fourth, third, second, second] (which becomes* $[11, 10, 01, 01]$ *in binary). The representation of that time series in Figure 2c becomes* $[1_2, 1_2, 01_4, 0_2]$*, where* $1_{(2)}$ *means that* 1 *is the selected symbol among 2 possible choices, 01 is the selected symbol among 4 possible choices, etc.*

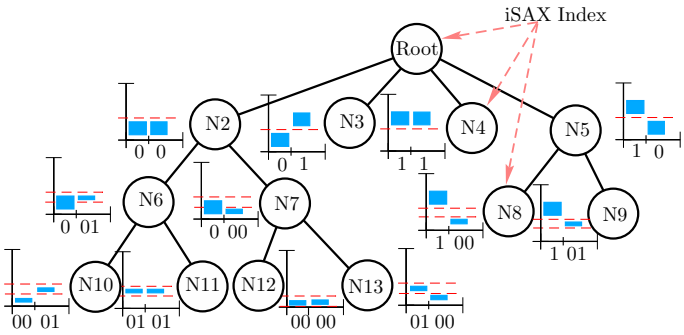


Fig. 3: *Example of iSAX Index*

, where the function $dist(x'_i, y'_i)$ is the distance between two iSAX symbols x'_i and y'_i. The lower bounding condition is formulated as:

$$MINDIST(iSAX(X), iSAX(Y)) \leq ED(X, Y)$$

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.

Example 3. Figure 3 illustrates an example of iSAX index, where each iSAX word has 2 symbols and a maximum cardinality of 4. The root node has 2^2 children while each child node forms a binary sub-tree. There are three types of nodes: *root node*, *internal node* (N2, N5, N6, N7) and *terminal node or leaf node* (N3, N4, N8, N9, N10, N11, N12, N13). Each leaf node links to a disk file that contains the corresponding time series (up to th time series).

Note that previous studies have shown that the iSAX index is robust with respect to the choice of parameters (word length, cardinality, leaf threshold) [4], [31], [38]. 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 [13].

2.2 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 two kinds of k nearest neighbors based queries.

Definition 1. (EXACT k NEAREST NEIGHBORS) Given a query time series Q and a set of time series D, let $R = ExactkNN(Q, D)$ be the set of k nearest neighbors of Q from D. Let $ED(X, Y)$ be the Euclidean distance between the points X and Y, then the set R is defined as follows:

$$(R \subseteq D) \wedge (|R| = k) \wedge$$
$$(\forall a \in R, \forall b \in (D - R), ED(a, Q) \leq ED(b, Q))$$

Definition 2. (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.

2.3 Spark

For implementing our parallel algorithms we use Spark [36], 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. Similar to MapReduce [5], Spark can be deployed on the Hadoop Distributed File System (HDFS) [28]. 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.

To execute a Spark job, we need a master node to coordinate job execution, and some worker nodes to execute a parallel operation. These parallel operations are summarized to two types: (i) Transformations: to create a new RDD from an existing one (*e.g.*, Map, MapToPair, MapPartition, FlatMap); and (ii) Actions: to return a final value to the user (*e.g.*, Reduce, Aggregate or Count).

2.4 Problem Definition

The problem we address is as follows. Given a (potentially huge) set of time series, find the results of exact and approximate k-NN queries as presented in definitions 1 and 2, by means of an index and query processing performed in parallel.

3 Distributed iSAX (DiSAX)

DiSAX, our first 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.

The pseudo-code of this index construction can be seen in Algorithm 1. The input is a data partitions that contains time series in ASCII form. First, the algorithm obtains the iSAX representation of all time series using the highest possible cardinalities (lines 2-4). Then each worker builds an independent iSAX index on its partition (lines 5-9) using the iSAX index insertion function (lines 10-26).

Algorithm 1: DiSAX Index construction

Input: Data partitions $P = \{P_1, P_2, \ldots, P_n\}$ of a database D, w the length of the iSAX word
Output: Index structures

```
1  D.cache(); //cache all the database in the cluster, where each time series has a unique ID
2  MapToPair( ID of Time series: ID ,Time Series: X )
3      Convert time series X to iSAX_word with high cardinalities and size w
4      emit (ID ,iSAX_word)
5  MapPartition( Set of <ID,iSAX_word>: iSAX_words )
6      rootNode = new RootNode
7      foreach <ID,iSAX_word> in iSAX_words do
8          rootNode.insert(ID ,iSAX_word)
9      emit (rootNode)
10 Function insert (ID ,iSAX_word)
11     if the subtree corresponding to iSAX_word exists then
12         node = the node corresponding to iSAX_word
13         if node is leaf node then
14             if node is not full then
15                 node.insert(ID ,iSAX_word)
16             else
17                 newNode = new InternalNode
18                 newNode.insert(ID ,iSAX_word)
19                 foreach iSAX_word in node do
20                     newNode.insert(ID ,iSAX_word)
21                 remove(node)
22         else
23             node.insert(ID ,iSAX_word)
24     else
25         newNode = new TerminalNode
26         newNode.insert(ID ,iSAX_word)
```

3.1 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 definitions 1 and 2. We perform such queries with two search methods: approximate and exact.

3.1.1 Approximate Search

Given a batch B of queries, the master node duplicates B on each worker (node) keeping an index for a subset of the data (i.e, a data split). 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. Each worker w sends all the found time series to the master. Let $|W|$ be the number of workers, the master thus receives $k \times |W|$ nearest neighbors for each

Algorithm 2: DiSAX Approximate Search

Input: iSAX Indexes, where each partition has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of queries time series
Output: k nearest neighbors

1 **MapToPair(** *ID of Time series:* ID *,Time Series:* q **)**
2 Convert time series X to *iSAX_word* with high cardinalities and size w.
3 **emit** (*ID ,iSAX_word*)
4 Duplicate Q on each partition
5 **MapPartition(** *iSAX index, Set of <ID,iSAX_word>:* $iSAX\_words$ **)**
6 get the $rootNode$ from iSAX index
7 **foreach** *<ID,iSAX_word>* ***in*** $iSAX\_words$ **do**
8 *rootNode*.ApprSearch(*ID ,iSAX_word*)
9 **emit** (*ApprSearch results*)
10 **Function** `ApprSearch(`*ID ,*$iSAX\_word$`)`
11 *node* = the node corresponding to $iSAX\_word$
12 **if** *node is a terminal node* **then**
13 Find the k nearest neighbors using Euclidean distance
14 **else**
15 *node*.ApprSearch(*ID ,iSAX_word*);

Algorithm 3: DiSAX Exact Search

Input: iSAX Indexes where each partition has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of queries time series
Output: k nearest neighbors

1 Duplicate Q on each partition
2 **MapPartition(** *iSAX index, Q* **)**
3 get the $rootNode$ from iSAX index
4 **foreach** q ***in*** Q **do**
5 **bsf** = *rootNode*.ApprSearch(*ID ,iSAX_word of q*) *rootNode*.ExactSearch(*ID ,q,bsf*)
6 **emit** (*ExactSearch results*)
7 **Function** `ExactSearch(`*ID ,q*`)`
8 **bsfDist** = Infinite; **queue** = Initialize a priority queue with all the children of the root;
9 **while** ***node*** = *pop next node from* ***queue*** **do**
10 **if** *node is terminal node and MinDist(q,node) < bsfDist* **then**
11 bsf = Find the k nearest neighbors
12 **else if** *MinDist(q, node)* $\geq$ *bsfDist* **then**
13 break;
14 **else**
15 Add the children of the node to priority queue ;

query Q, sorts them by decreasing order of their distance to Q, and selects the k top ones.

The algorithm, described in Algorithm 2, starts by obtain the iSAX representation of all queries time series using the highest possible cardinalities (lines 1-3). Then the master node duplicates the queries on each partition (worker) (line 4), and each worker uses its local index to retrieve time series that correspond to each query (lines 5-9), using the approximate search function (lines 10-15).

3.1.2 Exact Search

The exact search proceeds in two steps. In Step 1, the algorithm firstly uses the approximate search described in Section 3.1.1 to obtain *AKNN*, an approximate k nearest neighbours set. Then each worker creates a priority queue to examine the index nodes that may contain the time series that are probably more similar to Q than those of *AKNN*. Such nodes are identified as in the original iSAX [30], [31], where the lower bound distance used for priority queue ordering is computed using $MINDIST\_PAA\_iSAX$ according to *AKNN*. The difference is that, instead of a sequential scan of the series found in the identified leaf nodes, we emit the IDs of the series. In step 2, the algorithm retrieves all the time series that match the IDs emitted by the workers, and then finds the k nearest neighbors using the Euclidean distance.

The algorithm, described in Algorithm 3. The master node duplicates the queries on each partition (worker) (line 1), and each worker uses its local index and starts by putting all the children of the root in priority queue using their lower distance bound towards the query (line 8), Then the one with the best minimum distance is explored (line 9), if the best lower bound is bigger than the BSF distance (line 12) the algorithm stops. If node is an internal node (line 15) then all children are added into the priority queue.

3.2 Limitations of DiSAX

The parallel index constructed by DiSAX in a distributed environment is effective but calls for improvements. Actually, it leads to query response times that sometimes are high, because the query processing work is not well distributed among the computing nodes. The reason is that each node should examine all queries in the index, even if the index contains no similar result for the query.

Furthermore the index obtained by iSAX2+ would be very different from the union of the local distributed iSAX indexes. This also has an impact on the size of the index. Since merging all the local indexes would call for specific algorithms (if it is even possible) the size of the global index of distributed iSAX is higher than the index of centralized iSAX2+.

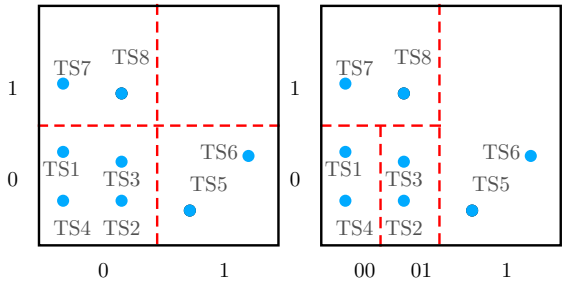
4 Distributed Partitioned iSAX (DBasicPiSAX and DPiSAX)

In this section, we present a novel parallel partitioned index construction algorithms, 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

TABLE 1: *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) *Partitioning according to DbasicPiSAX.* **(b)** *Partitioning according to DPiSAX.*

Fig. 4: *The result of the partitioning algorithms (DPiSAX and DbasicPiSAX) on sample S (from Table 1) into four partitions.*

how to distribute the time series in the splits, according to their iSAX representation. However, deciding the good split criteria calls for careful attention since bad choices may lead to highly imbalanced partitions, as illustrated in this section with i) DbasicPiSAX, a first version of our partitioned indexing technique and ii) DPiSAX, the final version with, to the best of our knowledge, the best load balance and the best querying performances obtained for time series indexing in distributed environments.

4.1 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 two partitioning methods that enable separating the dataset into non-overlapping subsets based on iSAX representations, namely "the basic approach" (or DbasicPiSAX) and "the statistical approach" (or DPiSAX). Both methods proceed with a common simple strategy: successively divide the sample by splitting the biggest partition into two sub-partitions, until the number of partitions is equal to the number of workers. However, at each step, once the biggest partition is identified, the main difference is in the assignment strategy (*i.e.*, how is each time series in the sample assigned to one or the other of the new partitions?).

4.2 Basic Approach: DbasicPiSAX

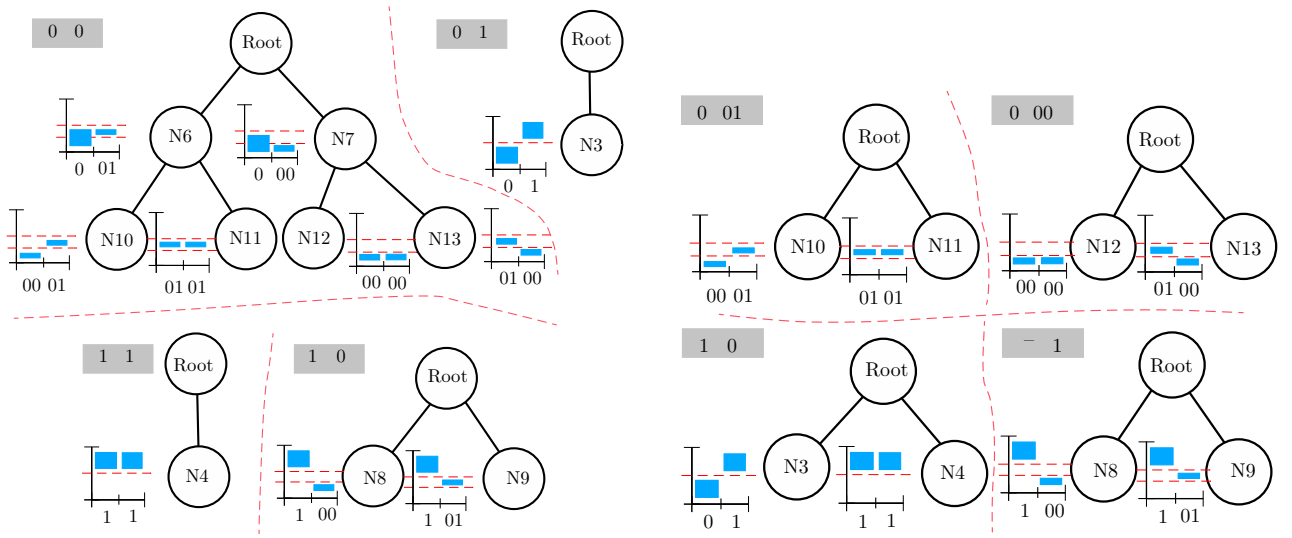
In the basic approach, splitting the biggest partition is done according to the first bit of each symbol in the iSAX words, as we can see in Algorithm 4 (line 1-4). Let us consider the n^{th} splitting step, each time series is assigned to a new partition depending on the first bit of its n^{th} symbol. Of course, when the number of symbols has been reached for a partition (*i.e.*, it cannot be divided anymore because the last symbol has been reached) then we need to consider the remaining partitions for new splits.

Example 4. Let's consider Table 1, 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. First, we use the first bit of the first segment to define two partitions. The first partition contains all the time series having their first iSAX word starting with 1, and the second partition contains the time series having their first iSAX word starting with 0. We obtain two partitions: "0" and "1". The biggest partition is "0" (*i.e.*, containing the time series $TS1$ to $TS6$). This partition is split again, according to the first bit of the second symbol. We now have the following partitions: from the first step, partition "1", and from the second step, partitions "00", and "01". Now, partition "00" is the biggest one. However, it cannot be split anymore since the maximum number of symbols has been reached. We choose the next biggest partition, *i.e.*, "1". After splitting this partition using the first bit of the second segment, we obtain two new partitions:"11" and "10". Partition "10" contains all the time series of the old partition (*i.e.*, partition "1"). Consequently, we have four partitions, where partition "11" is empty. Figure 4a shows the obtained partitions and Figure 5a shows the indexes obtained with these partitions.

The partitioning Algorithm achieves two goals: 1) generating P partitions; and 2) preserving vertical division of the iSAX tree. Notice that the second goal is achieved because our partitioning algorithm uses the first bit of each symbol. Therefore, iSAX words having cardinality 2 are used to produce a set of, at most, 2^w partitions. In the original iSAX index, when the construction starts with a cardinality of 2, a set of 2^w children is produced at the root node. Intuitively, in our running example, when we compare the centralize index (the original iSAX index) in Figure 3, and the parallel indexes in the Figure 5a obtained with the basic partitioning approach, we observe the vertical division of the original iSAX index.

4.3 Limitations of the Basic Approach

Obviously, the partitions obtained with the basic partitioning approach are not balanced. This is due to two main reasons. First, the partitioning algorithm preserves vertical division of the original iSAX index and the iSAX index is not balanced. The second reason is that, the partitioning algorithm does not take into account the data distribution in the partitions. Because of the limits in the number of symbols, it is possible to end up with highly imbalanced partitions, as illustrated by Figure 5a and also by our experiments. Because of this imbalanced distribution of the data,



(a) *DbasicPiSAX indexes after partitioning and indexing. Partition "11" is empty for the sampling step, but may contain data after indexing.*

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

Fig. 5: *The indexes built by DPiSAX and DbasicPiSAX on sample S (from Table 1) on four partitions.*

the basic approach is limited in the size of datasets it can process. If the capacity of a computing node is reached (*i.e.*, the node in charge of the biggest partition cannot handle the data that corresponds to it), then the index building process cannot progress.

Moreover, the maximum number of partitions that can be generated is 2^w (where w is the SAX word length). Since each partition is managed by a computing node for the local index construction, if the number of partitions is lower than the number of available computing nodes, then there will be idle nodes. This is a threat for the speed-up of the approach and calls for better solutions, as presented in the next subsection.

Algorithm 4: DbasicPiSAX Partitioning Function

Input: Sample S of *iSAX_words*, p number of partitions
Output: Partition Table BT
1 **while** *the number of partitions is less than* p **do**
2 *BigPartition* = the biggest partition
3 *//In the first iteration BigPartition = S*
4 Divide *BigPartition* into two partitions

4.4 Statistical Approach: DPiSAX

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) (Algorithm 5 lines 7-11) 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 (lines 4-5), 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 $\mu \pm \sigma$ (line 9). From the segments for which this is true, we choose the one having μ closer to the break-point (line 10).

In order to illustrate this, let us consider the blue boxes of the diagrams in Figure 5a. We choose the biggest blue box that ensures the best splitting by considering the next break-point.

Example 5. Let's consider the same case as described in Example 4. Figure 4b shows the obtained partitions and Figure 5b shows the indexes obtained with these 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 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.
We also illustrate, in Figure 5a, the variability of the distribution of time series for each symbol. For instance, in partition "00", for node $N6$, there is a much higher variability in the first symbol (marked "0" in the diagram, and represented by the blue box) than the second symbol (marked "01", blue box).

Algorithm 5: DPiSAX Partitioning Function

Input: Sample S of *iSAX_words*, p number of partitions
Output: Partition Table BT

```
1  while the number of partitions is less than p do
2      BigPartition = the biggest partition
3      //In the first iteration BigPartition = S
4      mean[] = ComputeSymbolsMean(BigPartition)
5      stdev[] = ComputeSymbolsStDev(BigPartition)
6      segmentToSplit = null
7      foreach segment s in BigPartition do
8          b = getbreak-point(s)
9          if b within mean[s] ± stdev[s] then
10             if mean[s] close to b then segmentToSplit
               then
11                 segmentToSplit = s
12     Divide BigPartition into two partitions in
       segmentToSplit
```

Optimization. Because many time series have the same iSAX representation, we may end up with groups of iSAX words that are the same, even when using the maximum cardinality (as it is our case). Therefore, we turn this data duplication into an advantage. Actually, the index construction is done as in Section 3, but the difference is that in the insertion function, we provide the algorithm with a bulk insertion function. The goal of this function is to better consider iSAX words with the same representation and to improve the index construction cost. This is done by linking all the IDs of time series having the same representation to only one corresponding iSAX word.

The pseudo-code of the parallel index construction by DPiSAX is shown in Algorithm 6. Given a time series dataset, the algorithm firstly creates the iSAX representation of each time series in parallel (lines 2-4). Then, it inserts the representations in parallel to the index by using the bulkInsertion function (lines 5-9). Each time series t is inserted to the index by the worker (*i.e.*, the processor) that is responsible for the partition to which t belongs. If the subtree of the partition does not exist, it will be created (lines 23- 25). Then, the time series t is inserted to its corresponding leaf node in the subtree (lines 14-15). If the node gets full (*i.e.*, its size gets higher than the threshold), then it will be split (lines 16-20).

4.5 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, according to the definitions of approximate k-NN and exact k-NN search as presented in definitions 1 and 2. Approximate and exact search are performed as follows:

- **Approximate search**: searching for the approximate k nearest neighbors of the time series Q is done as in Section 3.1.1. The difference is that just one iSAX index is queried instead of all the parallel indexes.

Algorithm 6: DPiSAX Index construction

Input: Data partitions $P = \{P_1, P_2, \ldots, P_n\}$ of a database D, w the length of the iSAX word, p number of partitions
Output: Index structures

```
1  D.cache(); //cache all the database in the cluster, where
   each time series has a unique ID
2  MapToPair( ID of Time series: ID ,Time Series: X )
3      Convert the time series X to iSAX_word with
       high cardinalities and size w
4      emit (ID ,iSAX_word)
5  MapPartition( Set of Set<ID,iSAX_word>:
   iSAX_words )
6      rootNode = new RootNode
7      foreach Set <ID,iSAX_word> in iSAX_words do
8          rootNode.bulkInsertion(Set<ID,iSAX_word>)
9      emit (rootNode)
10 Function bulkInsertion(Set <ID,iSAX_word>:
   iSAX_words)
11     if the subtree corresponding to iSAX_words exists
       then
12         node = the node corresponding to
           iSAX_words
13         if node is leaf node then
14             if node is not full then
15                 node.bulkInsertion(iSAX_words)
16             else
17                 newNode = new InternalNode
18                 newNode.bulkInsertion(iSAX_words)
19                 newNode.bulkInsertion(all iSAX words
                   of node)
20                 remove(node)
21         else
22             node.bulkInsertion(iSAX_words)
23     else
24         newNode = new TerminalNode
25         newNode.bulkInsertion(iSAX_words)
```

  Actually, we are able to 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. The algorithm, described in Algorithm 7, starts by obtain the iSAX representation of all queries time series using the highest possible cardinalities (lines 1-3). The master sends each query to the partition (worker) that has the same iSAX word as the query (line 4), and each worker uses its local index to retrieve time series that correspond to each query (lines 5-9), using the approximate search function (lines 10-15).
- **Exact search**: for retrieving the exact k nearest neighbors of a given query time series q, we first use the approximate search, described above, in order to obtain an approximate best-so-far k nearest neighbors. Then, each worker performs the exact search algo-

Algorithm 7: DPiSAX Approximate Search

Input: iSAX Indexes, where each partition has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of Query time series
Output: k nearest neighbors

1 **MapToPair(** *ID of Time series:* ID *,Time Series:* q **)**
2 Convert time series X to *iSAX_word* with high cardinalities and size w.
3 **emit** (*ID ,iSAX_word*)
4 Send each query to the partition that has the same iSAX word as the query
5 **MapPartition(** *iSAX index, Set of <ID,iSAX_word>:* $iSAX\_words$ **)**
6 get the the $rootNode$ from iSAX index
7 **foreach** *<ID,iSAX_word>* ***in*** $iSAX\_words$ **do**
8 *rootNode*.ApprSearch(*ID ,iSAX_word*)
9 **emit** (*ApprSearch results*)
10 **Function** `ApprSearch(`*ID ,*$iSAX\_word$`)`
11 *node* = the node corresponding to $iSAX\_word$
12 **if** *node is terminal node* **then**
13 Find the k nearest neighbors using Euclidean distance
14 **else**
15 *node*.ApprSearch(*ID ,iSAX_word*);

Algorithm 8: DPiSAX Exact Search

Input: iSAX Indexes, where each partitions has one index $I = \{I_1, I_2, \ldots, I_n\}$ and a collection Q of queries time series
Output: k nearest neighbors

1 Send each query to the partition that has the same iSAX word as the query
2 **MapPartition(** *iSAX index,* Q **)**
3 get the $rootNode$ from iSAX index
4 **foreach** q ***in*** Q **do**
5 **bsf** = *rootNode*.ApprSearch(*ID ,iSAX_word of q*) *rootNode*.ExactSearch(*ID ,q,bsf*)
6 **emit** (*ExactSearch results*)
7 **Function** `ExactSearch(`*ID ,q*`)`
8 **bsfDist** = Infinite; **queue** = Initialize a priority queue with all the children of the root;
9 **while** ***node*** = *pop next node from* ***queue*** **do**
10 **if** *node is terminal node and MinDist(q,node)* $<$ *bsfDist* **then**
11 bsf = Finds the k nearest neighbors
12 **else if** *MinDist(q, node)* $\geq$ *bsfDist* **then**
13 break;
14 **else**
15 Add the children of the node to priority queue ;

rithm as described in Section 3.1.2. This is described in Algorithm 8. The master sends each query to the partition (worker) that has the same iSAX word as the query (line 1), and each worker uses its local index and starts by putting all the children of the root in priority queue using their lower distance bound towards the query (line 8). Then, the one with the best minimum distance is explored (line 9). If the best lower bound is bigger than the BSF distance (line 12) then the algorithm stops. If the node is an internal node (line 15) then all its children are added to the priority queue.

5 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 evaluate the performance of three versions of our solution: 1) DiSAX is the parallel implementation of iSax as described in Section 3 ; 2) DbasicPiSAX is the sampling-based indexing algorithm with basic partitioning as described in Section 4.2; 3) DPiSAX is our complete solution, with the statistical partitioning described in Section 4.4. Furthermore, we compare our solutions to two state of the art baselines: the most efficient centralized version of iSAX index (*i.e.*, iSAX2+ [4]), 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) [25].

TABLE 2: *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%

Our experiments are divided into two sections. In Section 5.2, we measure the index construction times with different parameters. In Section 5.3, we focus on the query performance of our approach.

The splitting strategy of DPiSAX, described in Section 4.4 is essentially the same to the centralized state of the art one, iSAX2+, described in [4]. Therefore, the results retrieved in our experiments are the same as those retrieved by iSAX2+. In the interest of space, we do not show graphs for retrieval performances, since they would systematically show a 100% agreement to those of iSAX2+.

Reproductibility: we implemented our approaches on top of Apache-Spark [36], using the Java programming language. The iSAX2+ index is also implemented with Java. Our code is available at https://djamelinfo.github.io/test/projects/DPiSAX/.

5.1 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 [10]. After preprocessing, it contains 40 million time series of 256

values, for a total size of 150Gb. The second real world data is the TexMex corpus [11]. It contains 1 Billion 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 [1], [2], [3], [4], [8], [30], [37]. Table 2 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 [3], [4], [30], [31], [37] have shown to work well across data with very different characteristics.

5.2 Index Construction Time

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

Figure 6 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 all parallel approaches, 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, DbasicPiSAX in 76 minutes and DiSAX in 64 minutes, while the iSAX2+ index is built in more than 5 days on a single node.

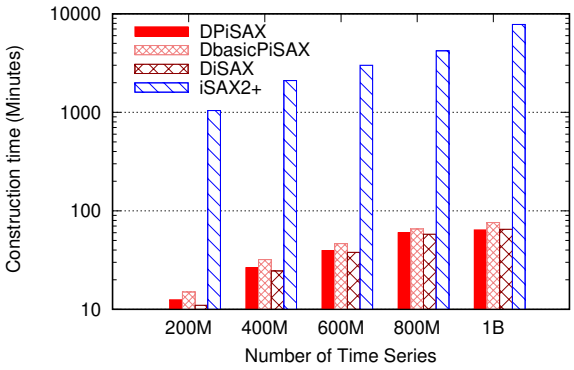


Fig. 6: *Logarithmic scale. Construction time as a function of dataset size. Parallel algorithms (DiSAX and DPiSAX) are run on a cluster of 32 nodes. iSAX2+ is run on a single node. With 1 billion Random Walk TS, iSAX2+ needs 5 days and our distributed algorithms need less than 2 hours.*

Figure 7 shows the same evaluation on the TexMex dataset. We can observe very similar behavior of our parallel approaches. As for the previous experiment, reported in Figure 6, the centralized version of iSAX2+ builds the index on a single machine in up to 4 days. We only report the response time of scalable approaches in Figure 7, for a better visual comparison of their performances.

Figure 8 reports an extended view on the index construction times, only for parallel approaches, and with datasets having size up to 4 billion time series (6.2TB). The running time increases with the number of time series for DPiSAX and DiSAX. DbasicPiSAX does not scale and cannot execute

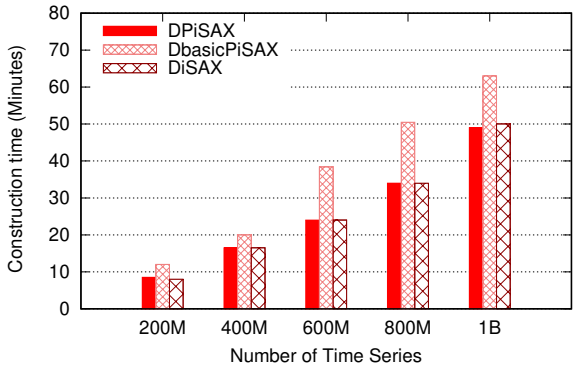


Fig. 7: *Construction time as a function of dataset size. Parallel algorithms (DiSAX and DPiSAX) are run on a cluster of 32 nodes. With 1 billion TS from TexMex dataset.*

on datasets having size above 1Tb. This is due to its imbalanced partitions, where one of the computing node receives so much data that it cannot build the index. This will be better discussed with Figure 14.

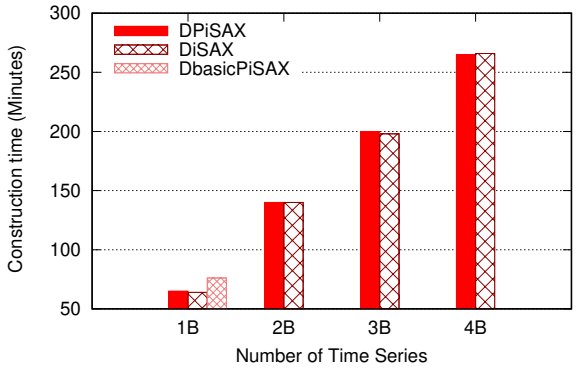


Fig. 8: *Construction time as a function of dataset size, for parallel algorithms on a cluster of 32 nodes, and with datasets up to 4 billion Random Walk time series.*

Figures 9 and 10 illustrate the parallel speed-up of our approach on the Random Walk (Figure 9) and the TexMex (Figure 10) datasets. The results show a near optimal gain for DPiSAX and DiSAX on our dataset. From the figure 9, we observe that the construction time for DbasicPiSAX is the same with 32 nodes and 40 nodes, this is because DbasicPiSAX does not use all the available processors. Actually, the basic partitioning algorithm (as described in Section 4.2) is limited in the number of partitions it can generate. By construction, it is able to generate up to $2^8 = 256$ partitions (more generally, 2^w partitions, where w is the SAX word length). In order to fully exploit the computing of all 320 cores, we need to build 320 partitions. This is over the maximum number of partitions that DbasicPiSAX is able to manage (*i.e.*, in this case, 256).

Figure 14 reports our measures of load balance, on 32 nodes and one billion time series, where partitions are sorted by decreasing order of the measured criteria: number of nodes in the local trees (11), number of time series in the partitions (Figure 12) and index depth (Figure 13). Our results illustrate the near ideal balance of our DPiSAX

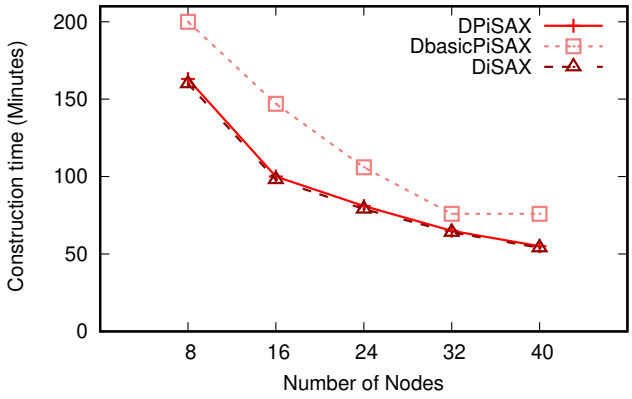


Fig. 9: *Construction time as a function of cluster size. DPiSAX and DiSAX have has a near optimal parallel speed-up. With 1 billion TS from the Random Walk dataset.*

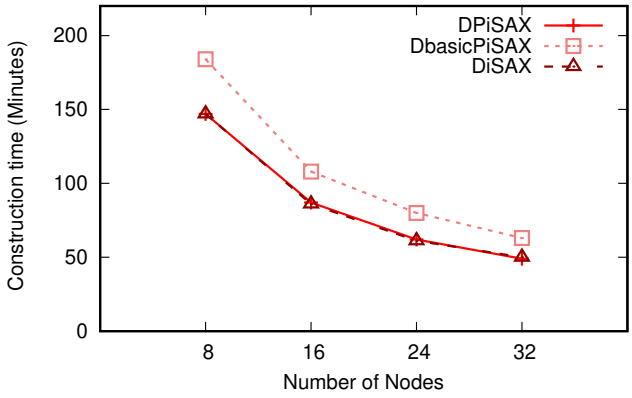


Fig. 10: *Construction time as a function of cluster size. DPiSAX and DiSAX have has a near optimal parallel speed-up. With 1 billion TS from the TexMex dataset.*

approach, while DbasicPiSAX is totally unbalanced. The number of time series, for instance, in the case of DbasicPiSAX, ranges from 0 (which means an empty partition) to 100 millions (*i.e.*, 10% of the data is indexed on one partition out of 320). DiSAX is perfectly balanced in the index construction phase owing to its sequential split of the data in the partitioning phase, but totally imbalanced in querying because it has to send the whole batch of queries to all partitions, leading to poor performances as illustrated in the remaining of our experiments.

Figure 15 reports the performance gains of our parallel approaches on the centralized version of iSAX2+ on our synthetic and real datasets. The results show that DPiSAX is between 40 and 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. As seen, the speedup of DPiSAX compared to the centralized iSAX2+ is higher than the number of cluster nodes, i.e. 32. The reason is that each node of the cluster has 8 cores, and for each core, Spark can create one worker (thread). Thus, the speedup is not higher than the number of cluster cores.

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 time taken by the Spark deployment when compared to index construction, is higher than the much larger Random Walk dataset.

Our experiments with varying leaf capacity show that this parameter has a negligible effect on performance (results omitted for brevity). This is because the RDD implementation used by Spark avoids the performance penalty related to disk I/O, which is heavily affected by the choice of the leaf capacity [4].

5.3 Query Performance

In the following experiments, we evaluate the querying performance of our algorithms, and compare them to the state of the art. In the case of our synthetic data, we generate Random Walk queries with the same distribution as described in Section 5.1. For the seismic data, we obtained seismic time series from the same IRIS Seismic Data Access repository [10] to be used as queries. In the case of the TexMex corpus, similar series correspond to similar images. The corpus contains 10^4 example queries together with information about which image in the corpus is the nearest neighbor. In any dataset, for each time series t in the query batch, the goal is to check if the approach is able to find the k time series that are considered to be the most similar to t in this dataset, both with exact and approximate K-NN search.

Figure 16 compares the search time of approximate k nearest neighbors queries for the parallel approaches proposed in this work. We can observe that the response time increases with the number of queries for all approaches. However, for DPiSAX the search time is lower than DbasicPiSAX (owing to the better partition balancing) and much better than DiSAX (owing to DPiSAX's cability of splitting the query batch and redirect the queries to the adequate partitions). In our experiments, we also compared the search time of parallel approaches to that of iSAX2+ for answering approximate k nearest neighbors queries with a varying size of query batch. We observed that the approximate search time of DPiSAX is better than that of the iSAX2+ by a factor of up to 16 (*e.g.*, the search time for 10 millions queries is 2270 sec for iSAX2+ and 138 sec for DPiSAX).

Figure 17 gives the exact search run time of our parallel approaches on the index constructed over 1 billion time series. We observe that DPiSAX is always faster than DbasicPiSAX and DiSAX, owing to its near ideal load balance.

Figure 18 compares cumulative time (Indexing + Exact 10-NN) of DPiSAX, DbasicPiSAX and DiSAX to PLS. A direct use of PLS is justified under 1K queries. Above that limit, the cumulative time of building the index and querying is much lower for our approaches, which are the clear winners.

Figure 19 illustrates the performance gains of our approaches on the centralized version of iSAX2+ and on PLS on synthetic and real world datasets, with batches of 10K queries (indexing time not included). We observe that DPiSAX and DbasicPiSAX have the best performance, owing to their query redirection mechanisms. However, DbasicPiSAX is not always as efficient as DPiSAX because of a less balanced partitioning. DPiSAX is generally between 19 and 43 times faster than iSAX2+ and PLS.

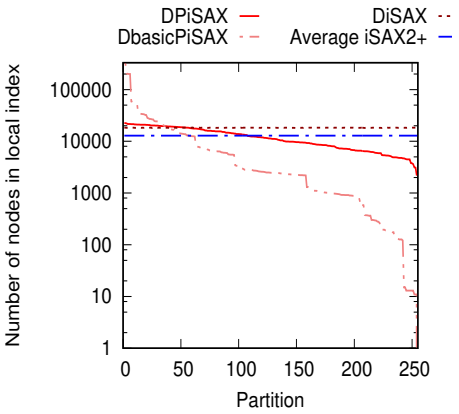


Fig. 11: *number of nodes*

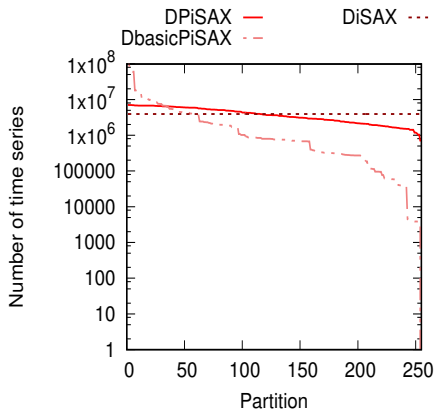


Fig. 12: *number of time series*

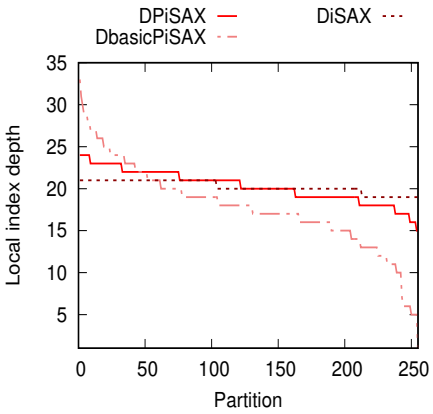


Fig. 13: *Local index depth*

Fig. 14: *Load balance in partitions: distribution of the number of nodes (a), of the number of time series (b), and of index depth (c), sorted by decreasing order in the partitions. The strong imbalance of DbasicPiSAX is the main reason of failure on massive datasets (i.e., above 1 billion TS).*

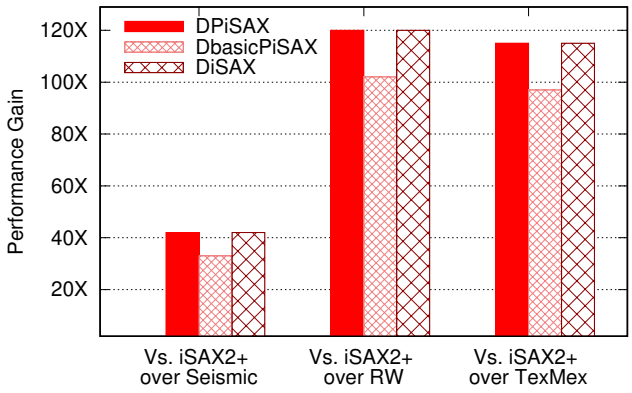


Fig. 15: *Performance gain on iSAX2+ in construction time, over seismic (40 millions TS), Random Walk (RW, 1 billion TS) and TexMex (1 billion TS), with a cluster of 32 nodes.*

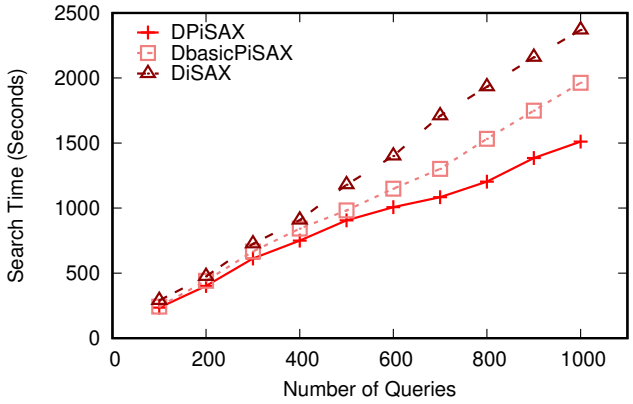


Fig. 17: *Run time of exact 10-NN queries over Random Walk dataset (limited to 1 billion TS because DbasicPiSAX does not scale on bigger datasets), cluster of 32 nodes.*

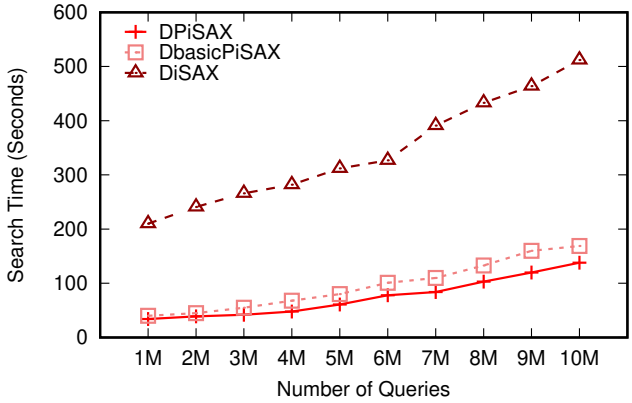


Fig. 16: *Run time of approximate 10-NN queries over Random Walk dataset (limited to 1 billion TS because DbasicPiSAX does not scale on bigger datasets), cluster of 32 nodes,.*

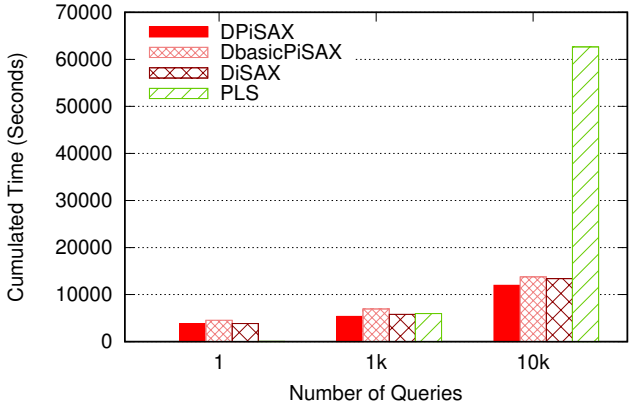


Fig. 18: *Cumulative time (Indexing + Exact 10-NN) over Random Walk dataset (limited to 1 billion TS because DbasicPiSAX does not scale on bigger datasets), cluster of 32 nodes.*

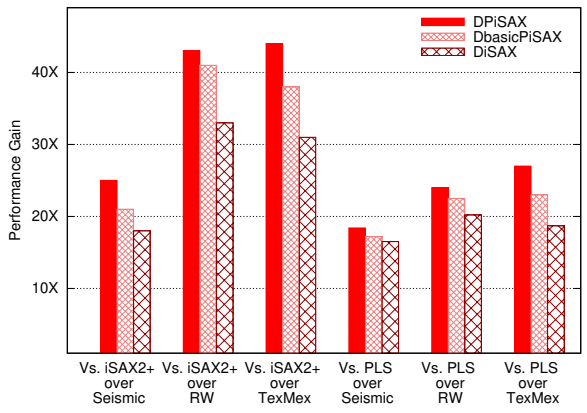


Fig. 19: *Performance gain (query only) of our parallel approaches on iSAX2+ and PLS, for exact 10-NN search time, batches of 10k queries, over seismic, Random Walk (RW) and TexMex datasets.*

6 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, InfluxDB, OpenTSDB, and DalmatinerDB 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) [30], [31], iSAX 2.0 [3], [4], iSAX2+ [4], Adaptive Data Series Index (ADS Index) [37], Dynamic Splitting Tree (DSTree) [32], Compact and Contiguous Sequence Infrastructure (Coconut) [14], Parallel Index for Sequences (ParIS) [24], and Ultra Compact Index for Variable-Length Similarity Search (ULISSE) [18]. A recent study is comparing the performance of several different time series indexes [6].

The iSAX index family (iSAX 2.0, iSAX2+, ADS Index) is based on SAX representation [16] 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 [4], 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 [32] 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.

The ParIS index [24] was recently proposed for taking advantage of the modern hardware parallelization opportunities within a single compute node. ParIS describes techniques that use the Single Instruction Multiple Data (SIMD) instructions, as well as the multi-core and multi-socket architectures, for parallel index creation and query answering. As such, ParIS is complementary to our approach.

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.

7 Conclusions

We proposed DPiSAX, a novel and efficient parallel solution to index and query billions of time series, or high-dimensional vectors, in general. 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 results show that by using our parallel techniques, 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.

In our future work, we intend to combine our approach with techniques that exploit modern hardware parallelism inside each computing node (i.e., SIMD, multi-core, multi-socket, and GPU) [23], [24].

Acknowledgments

The research leading to these results has received funding from the European Union's Horizon 2020, under grant agreement No. 732051.

References

[1] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Int. Conf.on FODO*, 1993.
[2] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: Efficient time series search and retrieval. In *EDBT*, 2008.
[3] 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.
[4] 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.
[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[6] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 2019.

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

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

[9] Pablo Huijse, Pablo A. Estévez, Pavlos Protopapas, Jose C. Principe, and Pablo Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Comp. Int. Mag.*, 9(3):27–39, 2014.

[10] IRIS. Seismic data access. http://ds.iris.edu/data/access/.

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

[12] Kunio Kashino, Gavin Smith, and Hiroshi Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.

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

[14] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut: A scalable bottom-up approach for building data series indexes. *PVLDB*, 11(6), 2018.

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

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

[17] Michele Linardi and Themis Palpanas. ULISSE: ultra compact index for variable-length similarity search in data series. In *ICDE*, 2018.

[18] Michele Linardi and Themis Palpanas. Scalable, variable-length similarity search in data series: The ulisse approach. *PVLDB*, 2019.

[19] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. Matrix profile X: VALMOD - scalable discovery of variable-length motifs in data series. In *SIGMOD*, 2018.

[20] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. VALMOD: A suite for easy and exact detection of variable length motifs in data series. In *SIGMOD*, 2018.

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

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

[23] Themis Palpanas. The parallel and distributed future of data series mining. In *International Conference on High Performance Computing & Simulation, HPCS*, 2017.

[24] Botao Peng, Themis Palpanas, and Panagiota Fatourou. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. *IEEE BigData*, 2018.

[25] 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.

[26] Usman Raza, Alessandro Camerra, Amy L. Murphy, Themis Palpanas, and Gian Pietro Picco. Practical data prediction for real-world wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, accepted for publication, 2015.

[27] W.H.Baumgartner et al. G.Ponti C.R.Shrader P. Lubinski H.A.Krimm F. Mattana J. Tueller S. Soldi, V. Beckmann. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 2014.

[28] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Int. ISPASS*, 2010.

[29] Dennis Shasha. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.*, 22(2):40–46, 1999.

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

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

[32] 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.

[33] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. Dpisax: Massively distributed partitioned isax. In *ICDM*, pages 1135–1140, 2017.

[34] Lexiang Ye and Eamonn J. Keogh. Time series shapelets: a new primitive for data mining. In *KDD*, 2009.

[35] 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*.

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

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

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

[39] Kostas Zoumpatianos and Themis Palpanas. Data series management: Fulfilling the need for big sequence analytics. In *ICDE*, 2018.

Djamel-Edine Yagoubi is a data scientist at StarClay. He did his PhD in Computer Science at Inria, in the Zenith team and the University of Montpellier. He has worked in the field of Big Data analytics, particularly Massively Distributed Time Series Indexing and Querying.

Reza Akbarinia is a research scientist at Inria. He received his Ph.D. degree in Computer Science from the University of Nantes in 2007. His research focuses on data management and analysis in large-scale distributed systems (P2P, grid, Cloud) and data privacy. He has authored and co-authored two books and several technical papers in main DB conferences and journals. He has served as PC member in several conferences, such as SIGMOD, VLDB, ICDE, EDBT, CIKM, etc.

Florent Masseglia is a scientific researcher in computer science at Inria since 2002. He works in Montpellier, in the Zenith team of Inria, on the analysis of very large scientific data. These data, derived from observations, experiments and simulation are indeed complex, often very large, and are at the heart of important issues to better understand the studied domains (agronomy, biology, medicine).
http://www.florent-masseglia.info

Themis Palpanas is Senior Member of the French University Institute (IUF), and Professor of computer science at Paris Descartes University. He is the author of 9 US patents (3 implemented in world-leading commercial data management products). He is the recipient of 3 Best Paper awards, and the IBM Shared University Research (SUR) Award. He is serving as Editor in Chief for BDR Journal, Associate Editor for PVLDB 2019 and TKDE journal, and Editorial Advisory Board member for IS journal.