RadiusSketch: Massively Distributed Indexing of Time Series
Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Dennis Shasha

To cite this version:

HAL Id: lirmm-01620154
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01620154
Submitted on 20 Oct 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
RadiusSketch: Massively Distributed Indexing of Time Series

Djamel-Edine Yagoubi1,2,
Florent Masseglia1

1Inria & LIRMM, Montpellier, France
2Dep.of Computer Sc., NYU

Djamel.Edine.Yagoubi@inria.fr
Florent.Masseglia@inria.fr

Abstract—Performing similarity queries on hundreds of millions of time series is a challenge requiring both efficient indexing techniques and parallelization. We propose a sketch/random projection-based approach that scales nearly linearly in parallel environments, and provides high quality answers. We illustrate the performance of our approach, called RadiusSketch, on real and synthetic datasets of up to 1 Terabyte and 500 million time series. The sketch method, as we have implemented, is superior in both quality and response time compared with the state of the art approach, iSAX2+. Already, in the sequential case it improves recall and precision by a factor of two, while giving shorter response times. In a parallel environment with 32 processors, on both real and synthetic data, our parallel approach improves by a factor of up to 100 in index time construction and up to 15 in query answering time. Finally, our data structure makes use of idle computing time to improve the recall and precision yet further.

I. INTRODUCTION

Time series arise in many application domains such as finance, agronomy, health, earth monitoring, weather forecasting, to name a few. Because of advances in sensor technology, such applications may produce millions to trillions of time series per day, requiring fast analytical and summarization techniques.

Usually, indexing is at the core of major time series management solutions, as well as analytical tasks (e.g., classification, clustering, pattern discovery, visual analytics, and others) because indexes enable fast execution of similarity queries, which constitute the core operation of the domain. That core operation is what we want to solve very fast, viz. given a time series, find similar time series (e.g., all those having a correlation above a threshold).

Unfortunately, creating an index over billions of time series by using traditional centralized approaches is highly time consuming. Our experiments show that iSAX2+ [5], a state of the art index, may take at least one day with one billion time series, or more, in a centralized environment (see the building time reported in [5]). Most state of the art indices have focused on: i) data representation, with dimensionality reduction techniques such as Discrete Wavelet Transform, Discrete Fourier Transform, or more recently Symbolic Aggregate Approximation; and ii) index building techniques, considering the index as a tree that calls for optimal navigation among sub-trees and shorter traversals.

An appealing opportunity for improving the index construction time is to take advantage of the computing power of distributed systems and parallel frameworks such as MapReduce[11] or Spark [26].

In centralized systems, one of the most efficient ways to index time series for the purpose of similarity search is to combine a sketch approach [9] with grid structures. Random projection is based on the idea of taking the inner product of each time series, considered as a vector, with a set of random vectors whose entries are +1 or -1 [9]. The resulting sequence of inner products is called a sketch vector (or sketch for short). The goal is to reduce the problem of comparing pairs of time series to the problem of comparing their sketches, which are normally much shorter.

To avoid comparing the sketch of each time series of the database with that of the searched time series, [9] uses grid structures on pairs of sketch entries (e.g., the first and second entry in one grid, the third and fourth in the second grid, and so on) to reduce the complexity of search. Given the sketches \( s \) and \( s' \) of two time series \( t \) and \( t' \), the more grids in which \( s \) and \( s' \) coincide, the greater the likelihood that \( t \) and \( t' \) are similar. In time series data mining, sketch-based approaches have also been used to identify representative trends [10], maintain histograms [25], and to compute approximate wavelet coefficients [13], etc. All aspects of the sketch-based approach are parallelizable: the computation of sketches, the creation of multiple grid structures, and the computation of pairwise similarity. However, a straight parallel implementation of existing techniques would under-exploit the available computing power.

In this paper, we propose a parallel solution to construct a sketch-based index over billions of time series. Our solution makes the most of the parallel environment by exploiting each available core. Our contributions are as follows:

- We propose a parallel index construction algorithm that takes advantage of distributed environments to efficiently build sketch-based indices over very large volumes of time series. In our approach, we provide a greedy technique that uses idle processors of the system to increase query precision.
- We propose a parallel query processing algorithm, which given a query, exploits the available processors of the
distributed system to answer the query in parallel by using the constructed index which has already been distributed among the nodes of the system at construction time.

- We implemented our index construction and query processing algorithms, and evaluated their performance over large volumes of data, i.e., 500 million time series for a total size of 1 Terabytes. The results illustrate the efficiency of index construction in massively distributed environments. We compare our index construction algorithm to the current state of the art index building algorithm and show how parallelism enables important gains. We also illustrate how efficient our index is for search queries, and show how to improve precision and recall.

The rest of the paper is organized as follows. In Section II we discuss closely related background material on time series indexing. In Section III we describe the details of our parallel index construction and query processing algorithms. In Section IV we present a detailed experimental evaluation to verify the effectiveness of Sketch Approach compared to iSAX2+. Finally, we conclude in Section V.

II. RELATED WORK

Indexes often make the response time of lookup operations sublinear in the database size. Relational systems have mostly been supported by hash structures, B-trees, and multidimensional structures such as R-trees, with bit vectors playing a supporting role. Such structures work well for lookups, but only adequately for similarity queries.

The problem of indexing time series using centralized solutions has been widely studied in the literature, e.g., [3], [4], [12], [24], [5]. For instance, in [3], Assent et al. propose the TS-tree (time series tree), an index structure for efficient retrieval and similarity search over time series. The TS-tree provides compact summaries of subtrees, thus reducing the search space very effectively. To ensure high fanout, which in turn results in small and efficient trees, index entries are quantized and dimensionally reduced.

In [4], Cai et al. use Chebyshev polynomials as a basis for dealing with the problem of approximating and indexing d-dimensional trajectories and time series. They show that the Euclidean distance between two d-dimensional trajectories is lower bounded by the weighted Euclidean distance between the two vectors of Chebyshev coefficients, and use this fact to create their index.

In [12], Faloutsos et al. use R*-trees to locate multi dimensional sequences in a collection of time series. The idea is to map a large time series sequence into a set of multi-dimensional rectangles, and then index the rectangles using an R*-tree. Our work is able to use the simplest possible multi-dimensional structure, the grid structure, because our problem is simpler as we will see.

A. Discrete values

Several techniques have been used to reduce the dimensionality of time series, before creating the index. Examples of such techniques that can significantly reduce the time and space required for the index are: singular value decomposition (SVD) [12], the discrete Fourier transformation (DFT) [2], discrete wavelets transformation (DWT) [7], piecewise aggregate approximation (PAA) [19], adaptive piecewise constant approximation (APCA) [6], random sketches [9], and symbolic aggregate approximation (SAX) [21].

In [24], Shieh et Keogh propose a multiresolution symbolic representation called indexable Symbolic Aggregate Approximation (iSAX) which is based on the SAX representation. The advantage of iSAX over SAX is that it allows the comparison of words with different cardinalities, and even different cardinalities within a single word. iSAX can be used to create efficient indices over very large databases. In [5], an improved version of iSAX, called iSAX2+, has been used to index more than one billion time series. It uses two different buffers for storing parts of the index and time series in memory before flushing them into the disk. It also uses an efficient technique for splitting a leaf node when its size is higher than a threshold. In [27], instead of building the complete iSAX2+ index over the complete dataset and querying only later, Zoumpatianos et al. propose to adaptively build parts of the index, only for the parts of the data on which the users issue queries. We view the state-of-the-art iSAX2+ as the primary competition to our own indexing method.

B. Sketches

Our method is based on the use of random vectors. The basic idea is to multiply each time series (or in a sliding window context, each window of a time series) with a set of random vectors. The result of that operation is a "sketch" of the time series consisting of the distance (or similarity) of the time series to each random vector. Then two time series can be compared by comparing sketches.

The theoretical underpinning of the use of sketches is given by the Johnson-Lindenstrauss lemma [17].

Lemma 1: Given a collection $C$ of $m$ time series with length $n$, for any two time series $\vec{x}, \vec{y} \in C$, if $\epsilon < 1/2$ and $k = \frac{9\log m}{\epsilon^2}$, then

$$(1 - \epsilon) \leq \frac{\| s(\vec{x}) - s(\vec{y}) \|}{\| \vec{x} - \vec{y} \|} \leq (1 + \epsilon)$$

holds with probability 1/2, where $s(\vec{x})$ is the Gaussian sketch of $\vec{x}$.

Note that the sketch approach we advocate is a kind of Locality Sensitive Hashing [14], by which similar items are hashed to the same buckets with high probability. In particular, the sketch approach is similar in spirit to SimHash [8], in which the vectors of data items are hashed based on their angles with random vectors.

Our goal (and contribution) is to construct a parallel index by exploiting the sketch approach in a distributed environment for both:

- better performance (fast index building, compared to approaches from the state of the art, like iSAX2+)
• high quality (precision and recall) for similarity search.

To the best of our knowledge, the literature contains no solution for parallel construction of indices over very large time series datasets (e.g., billions of time series, with hundreds or thousands of values each). The fact is that parallelization of existing methods is not straightforward. iSAX2+ [3], for instance, creates a tree on a single machine. A straightforward implementation in a parallel environment would be to distribute the nodes at the first level on different machines. In this case, the communication cost might overwhelm the benefits. As explained in the introduction, we parallelize the sketch approach both at index creation time and at query processing time. Experiments show excellent and nearly linear gains in performance.

III. PARALLEL SKETCH APPROACH

This section reviews our algorithm for sketches, discusses the index structure required, and then shows how to parallelize the construction both to increase speed and improve quality.

A. The Sketch Approach

The sketch approach, as developed by Kushilevitz et al. [20], Indyk et al. [15], and Achlioptas [1], provides a very nice guarantee: with high probability a random mapping taking \( b \) points in \( R^m \) to points in \( (R^d)^{2b+1} \) (the \((2b+1)\)-fold cross-product of \( R^d \) with itself) approximately preserves distances (with higher fidelity the larger \( b \) is).

In our version of this idea, given a point (a time series or a window of a time series) \( t \in R^m \), we compute its dot product with \( N \) random vectors \( r_i \in \{1, -1\}^m \). This results in \( N \) inner products called the sketch (or random projection) of \( t \). Specifically, sketch function of \( t \) is \( sketch(t_i) = (t_i \cdot r_1, t_i \cdot r_2, ..., t_i \cdot r_N) \).

We compute sketches for \( t_1, ..., t_b \) using the same random vectors \( r_1, ..., r_N \). By the Johnson-Lindenstrauss lemma [17], the distance \( ||\text{sketch}(t_i) - \text{sketch}(t_j)|| \) is a good approximation of \( ||t_i - t_j|| \). Specifically, if \( ||\text{sketch}(t_i) - \text{sketch}(t_j)|| < ||\text{sketch}(t_k) - \text{sketch}(t_m)|| \), then it's very likely that \( ||t_i - t_j|| < ||t_k - t_m|| \).

Figure 1 gives an illustration of the Symbolic Aggregate approXimation distance (SAX distance) and sketch distance, compared to the actual Euclidean distance. This is done for 1,000 couples of random time series from a seismic dataset. We observe that \( \text{Sketch} < \text{SAX} < \text{Euclidean distance} \) on the first two dimensions (vectors). Example 1: Let’s consider two time series \( t = (2, 2, 5, 2, 6, 5) \) and \( t' = (2, 1, 6, 5, 5, 6) \). Suppose that we have generated four random vectors as follows: \( r_1 = (-1, 1, -1, 1, 1, 1), r_2 = (1, 1, 1, -1, 1, 1) \), \( r_3 = (-1, 1, 1, 1, -1, 1) \), and \( r_4 = (1, 1, -1, 1, 1, 1) \). Then the sketches of \( t \) and \( t' \), i.e., the inner products computed as described above, are respectively \( s_1 = (14, 6, 18) \) and \( s_2 = (13, 5, 11, 15) \). In this example, we create two grids, Grid 1 and Grid 2, as depicted in figure 2. Grid 1 is built according to the sketches calculated with respect to vectors \( r_1 \) and \( r_2 \) (where \( t \) has sketch values 14 and 6 and \( t' \) has sketch values 13 and 5). In other words, Grid 1 captures the values of the sketches of \( t \) and \( t' \) on the first two dimensions (vectors). Grid 2 is built according to vectors \( r_3 \) and \( r_4 \) (where \( t \) has sketch values 5 and 18 and \( t' \) has sketch values 11 and 15). Thus, Grid 2 captures the values of the sketches on the last two dimensions. We observe that \( t \) and \( t' \) are close to one another in Grid 1. On the other hand, \( t \) and \( t' \) are far apart in Grid 2.
B. Partitioning Sketch Vectors

In the following, we use correlation and distance more or less interchangeably because one can be computed from the other once the data is normalized. Specifically, the Pearson correlation is related to the Euclidean distance as follows: Here \( \hat{x} \) and \( \hat{y} \) are obtained from the raw time series by computing
\[
\hat{x} = \frac{x - \text{avg}(x)}{\sigma_x}, \quad \text{where} \quad \sigma_x = \sqrt{\sum_{i=1}^{n}(x_i - \text{avg}(x))^2}.
\]

Multi-dimensional search structures don’t work well for more than four dimensions in practice [23]. For this reason, as indicated in Example[1], we adopt a first algorithmic framework that partitions each sketch vector into subvectors and builds grid structures for the subvectors as follows:

- **Partition each sketch vector** \( s \) of size \( N \) into groups of some size \( |g| \).
- **The \( i \)th group of each sketch vector** \( s \) is placed in the \( i \)th grid structure (of dimension \( |g| \)).
- **If two sketch vectors** \( s \) and \( s' \) are within distance \( c \times d \) in more than a given fraction \( f \) of the groups, then the corresponding time series are candidate highly correlated time series and should be checked exactly.

For example, if each sketch vector is of length \( N = 40 \), we might partition each one into ten groups of size \( |g| = 4 \). This would yield 10 grid structures. Suppose that the fraction \( f \) is 90%, then a time series \( t \) is considered similar to a searched time series \( t' \), if they are similar in at least nine grids.

C. Distributed Framework

Spark [26] is a parallel programming framework that aims at efficiently processing large datasets. This programming model can perform analytics with in-memory techniques to overcome disk bottlenecks. Similar to MapReduce [11], Spark can be deployed on the Hadoop Distributed File System (HDFS) [23] as well as standalone. Unlike traditional in-memory systems, the main feature of Spark is its distributed memory abstraction, called resilient distributed datasets (RDD). RDD is an efficient and fault-tolerant abstraction for memory abstraction, called resilient distributed datasets (HDFS) [22] as well as standalone. Unlike traditional in-

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).

D. Massively Distributed Index construction

Our approach for sketch construction in massively distributed environments proceeds in two steps: 1) **Local construction of sketch vectors and groups**, on the distributed computing nodes; 2) **Global construction of grids**, with one computing node per grid.

1) **Local construction of sketch vectors and groups**: Before distributing the construction of sketch vectors, the master node creates a set of \( N \) random vectors of size \( n \), such that each vector \( r_i = (r_{i,1}, r_{i,2}, \cdots, r_{i,n}) \), contains \( n \) elements. Each element \( r_{i,j} \in r_i \) is a random variable in \( \{1, -1\} \) with probability \( 1/2 \) for each value. Let \( R \) be the set of random vectors. \( R \) is duplicated on all workers (i.e., processors of the distributed system), so they all share the same random vectors.

Let \( D \) be the input dataset involving \( t \) times series. Each time series \( t \in D \) is of length \( n: t = (t_{1,1}, t_{2,1}, \cdots, t_{n,n}) \). \( D \) can be represented as a matrix as follows:

\[
D = \begin{bmatrix}
    t_1 &=& <t_{1,1} & \cdots & t_{1,n}> \\
        &\vdots& &\vdots& \\
    t_l &=& <t_{l,1} & \cdots & t_{l,n}>
\end{bmatrix}
\]

During the first step of sketch construction, each mapper takes a set of time series \( P \subseteq D \), and projects them to the random vectors of \( R \), in order to create their sketches. Let \( s_j = (s_{j,1}, s_{j,2}, \cdots, s_{j,N}) \), be the sketch of a time series \( t_j \), then \( s_{j,i} \in s \) is the inner product between \( t_j \) and \( r_i \). The sketch of \( t_j \) can be written as \( s_j = t_j \cdot R \).

Let \( p \) be the number of time series involved in \( P \), then the result of random projection in a mapper is a collection \( X \) of \( p \) sketches, each corresponding to one times series of \( P \):

\[
X_{N \times t} = \begin{bmatrix}
    s_1 &=& <s_{1,1} & \cdots & s_{1,N}> \\
        &\vdots& &\vdots& \\
    s_p &=& <s_{p,1} & \cdots & s_{p,k}>
\end{bmatrix}
\]

The sketches are partitioned into equal subvectors, or groups, according to the size of sketches and vectors. If, for instance, sketch vectors have length 40, and groups have size four, we partition each vector into ten groups (of size four). For distribution needs, the mapper assigns to each group an ID in \([1..\text{NumberOfGroups}]\).

With a sequential construction of groups, where groups are contiguous, the mappers simply have to emit \(<\text{key}, \text{value}>\) pairs where \( \text{key} \) is the unique ID of a group and \( \text{value} \) is a tuple made of the data values of the sketch for these dimensions, and the time series ID.

**Example 2**: Let us consider \( s_j \) the sketch of series \( t_j \), such that \( s_j = (<2,4,5,9>, \{g_1, g_2\}) \) the set of two contiguous groups of size two that can be built on \( s_j \) (i.e., \( g_1 = (s_{j,1}, s_{j,2}), g_2 = (s_{j,3}, s_{j,4}) \)). In the simple version of our approach, this information is communicated to reducers (in charge of building the corresponding grids) by emitting two \(<\text{key}, \text{value}>\) pairs: \(<\text{key} = g_1, \text{value} = ((2,4),1)\>\) for the information about \( g_1 \) and \(<\text{key} = g_2, \text{value} = ((5,9),1)\>\) for the information about \( g_2 \).

2) **Optimized shuffling for massive distribution**: In cases when a dimension may be involved in multiple groups, a mapper emits each dimension ID (rather than the group ID) as the key, while the value embeds a couple, made of the data value of the sketch for this dimension and the series ID. The goal is to avoid sending redundant information that is repeated from one group to another. This is even more important when...
Algorithm 1: Index construction

Input: Data partitions $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ of a database $\mathcal{D}$ and a collection $\mathcal{R}$ of random vectors

Output: Grid structures

// Map Task
1 flatMapToPair( Time Series: $T$ )
2 - Project $T$ to $\mathcal{R}$
3 - Partition sketch into equal groups
4 forall groups do
5     emit (key: ID of group, value:(Grid Structure, list(values)))
6 // Reduce Task
7 reduceByKey( key: ID of group, list(values) )
8 - Use the list of values to build a d-dimensional grid structure
9     emit (key:ID of group, values: grid structure)

Algorithm 2: Query processing

Input: Grids Structures, a collection $\mathcal{R}$ of random vectors and a collection $\mathcal{Q}$ of Query time series

Output: List of time series

// Map Task 1
1 flatMapToPair( Time Series: $T_q$ )
2 - Project $T_q$ to $\mathcal{R}$
3 - Partition sketch into equal groups
4 forall groups do
5     emit (key: ID of group, value: $T_q$ ID and group data)
6 - Combine the values of the previous job result with the result of previous task, where key: ID of group and value:(Grid Structure, list(values))

// Map Task 2
7 flatMapToPair( key: ID of group, value:(Grid Structure, list(values)) )
8 foreach group in values do
9     if group $\in$ Grid Structure then
10        emit (key: $T_q$ ID, value: IDs of the found time series)

// Reduce Task
11 reduceByKey( key: $T_q$ ID, value: list(values) )
12 foreach found time series do
13     - Computes the number of occurrence
14     if the found time series has the greatest count value then
15        emit (key: $T_q$ ID, value: ID of the found time series)

3) Global construction of grids: Reducers receive local information from mappers, from which they construct grids. More precisely, in the reduce phase, each reducer receives a group ID, and the list of all generated values (group data and sketch ID). It uses the list to build a d-dimensional grid structure. Each grid is stored in a d-dimensional array in the mappers main memory or in HDFS (Hadoop Distributed File System), where each group is mapped to a cell according to its values. The pseudo-code of our index construction in Spark is shown in algorithm [1]

E. F-RadiusSketch

The above framework circumvents the curse of dimensionality by making the groups small enough that grid structures can be used. Our goal is to achieve extremely high recall (above 0.8) and reasonable precision (above 0.58). Increasing the size of the sketch vector improves the accuracy of the distance estimate but increases the search time. In our experiments, accuracy improved noticeably as the sizes increased to about 256. Beyond that, accuracy does not improve much and performance suffers. Because the dimensions used for comparison in the grids do not have to be disjoint, we build grids based on random and possibly overlapping combinations of dimensions. Our strategy is to choose the same number of combinations as the available processors.

Let the similarity $sim(t, t')$ of two series be the fraction of grids where $t$ and $t'$ fall in the same cell, for all possible grids. We show that by increasing the number of grids, the standard error in the computed similarity of $t$ and $q$ decreases. Let $G$ be
the set of all grids which can be generated for the sketches. Suppose $p$ is the percentage of the grids of $G$, in which $q$ and $t$ are similar. Let $G_k$ be the set of $n$ grids randomly selected from $G$, and $m(G_k)$ be the fraction of the grids of $G_k$ in which $t$ and $q$ are similar. Let $\Delta_m$ be the standard error in $m(G_k)$, i.e., $\Delta_m = |p - m(G_k)|$. The selection of $G_k$ grids from $G$ can be considered as a sampling process. We know from statistics [13] that by increasing the number of samples, the standard error of the mean of samples decreases. Thus, increasing the number of random grids decreases the standard error of $m(G_k)$. When the samples are independent, the standard error of the samples mean is computed as: $\Delta_m = \frac{\delta}{\sqrt{m}}$ where $\delta$ is the standard deviation of samples’ distribution. The samples aren’t independent in our case because the grids may overlap, but this is a suggestive approximation. Our goal in random combinations is to get as close as possible to the best possible results of sketches, in order to lower the error. This goal can be achieved by adjusting the number of random groups where, according to the discussion above, the error decreases with the number of random groups.

F. 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 $D$. We perform such a search in three steps, as follows.

Step 1: map. Each mapper receives a subset of the searched time series $Q' \subset Q$ and the same collection $R$ of random vectors that was used for constructing the index (see Section III-D1). The mappers generate in parallel the sketch vector for each given time series $t$ in their subset of queries, and partition the sketch vectors into groups (the same dividing principle into groups used for constructing the grid structures is applied). Each mapper emits the ID of groups as the key, and the sketch ID (i.e., query ID) coupled with group’s sketch data as value.

Step 2: map. Each mapper takes one or several grid structures of the index and the emitted groups of step 1 that correspond to the chosen grids. For each sketch of a searched time series $t$ in a group, the mapper checks in the corresponding grid, the cell that contains data points similar to $t$, where each data point contains the ID of the corresponding times series in the grid structure as depicted in Figure 3a. For each time series $t \in Q$, and for each times series $t'$ that belongs to the same cell as $t$ in the grid structure, the mapper emits a key-value pair, where the key is the ID of $t$, and the value is the ID of $t'$.

Step 3: reduce. In the reduce phase, each reducer computes for each given key (i.e., the ID of the searched time series) the count of each emitted value, i.e., the IDs of the found time series in different grids. Then, for the searched time series, the reducer emits to HDFS the ID of the time series that has the greatest count value.

The pseudo-code of query processing for Spark is given by Algorithm 2.

Searching for the $k$-nearest neighbours (k-NN) of the time series $Q$ is done as in the previous section III-F in three steps steps. The difference is that in Step 2, each mapper returns for each searched time series, $k$ candidate times series from the grid. In addition, in Step 3, for each query $t$, $k$ candidates that have the highest counts are returned as the answer to the query $t$. Sometimes, in Step 2, the mapper does not find enough data points in cell $c$, leading to a lack of information for time series retrieval on the third step. In such cases, all the neighbors of $c$ will be visited until $k$ points are found, as depicted in figure 3b.

IV. Experiments

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

We evaluate the performance of two versions of our solution: 1) RadiusSketch that is the basic version of our parallel indexing approach with partitioning; 2) F-RadiusSketch, as described in Section III-F We compare our solutions with the most efficient version of the iSAX index (i.e., iSAX2+2) proposed in [5]. We implemented two versions of our approach, one for centralized environments and the other version on top of Apache-Spark [26] for a distributed environment, using the Java programming language. The iSAX2+ index [5] is also implemented with Java, in a centralized version only. The source code of our approaches, our version of iSAX2+ and the documentation for reproducible experiments, are available at: https://radiussketch.github.io/RADIUSketch/

The parallel experimental evaluation was conducted on a cluster of 32 machines, each operated by Linux, with 64 Gigabytes of main memory, and Intel Xeon X5670 CPU and 250 Gigabytes hard disk. The centralized versions of sketches and iSAX2+ were executed on a single machine with the same characteristics.

Our experiments are divided into two sections. In Section IV-B, we measure the grid construction times with different parameters. In Section IV-C, we focus on the query performance of the sketch approach, both in terms of response time and accuracy.

A. Datasets and Setting

1) Datasets: We carried out our experiments on both real-word and synthetic datasets. The first one is a seismic dataset of 40 million time series, where each time series has a length of 256. It has a total size of 491 Gigabytes. For the second one, we generated a dataset of 500 million time series using a random walk data series generator, 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 (which may be negative) of the last number to the new number. The total size of our synthetic dataset is 1 Terabytes.

2) Parameters: Table I shows the default parameters (unless otherwise specified in the text) used for each approach. For Sketch and RadiusSketch, the number of groups is given by $\frac{\text{SketchSize}}{\text{GroupSize}}$. For F-RadiusSketch, the number
Fig. 3: Query processing in RadiusSketch. Each mapper identifies the cell that correspond to a query and emits the IDs of the corresponding time series (a). If there is not enough information the mapper does a broader search in the adjacent cells (b).

Fig. 4: Construction time as a function of dataset size (random walk dataset). Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

Fig. 5: Construction time as a function of dataset size (seismic dataset). Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-RadiusSketch</td>
<td>Group size = 2</td>
<td>iSAX2+</td>
<td>Threshold = 8,000</td>
</tr>
<tr>
<td></td>
<td>Sketch size = 60</td>
<td></td>
<td>Word length w = 8</td>
</tr>
<tr>
<td></td>
<td>Number of groups = 256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RadiusSketch</td>
<td>Group size = 2</td>
<td>Sketch</td>
<td>Group size = 2</td>
</tr>
<tr>
<td></td>
<td>Sketch size = 120</td>
<td></td>
<td>Sketch size = 60</td>
</tr>
</tbody>
</table>

TABLE I: Default parameters

B. Grid Construction Time

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

Figures 4 and 5 report the index construction times for both of the tested datasets. The index construction time increases with the number of time series for all approaches. In our distributed testbed, the index construction time is lower than it is in a centralized environment, with time reduced almost linearly. Figure 4 reports the construction time of centralized approaches (iSAX2+ and sketches) in days, while the scale unit is in minutes for RadiusSketch (60 groups of size 2) and F-RadiusSketch (256 groups of size 2). For 500 million time series, on the random walk dataset, the RadiusSketch index is built in 35 minutes on 32 machines, while the iSAX2+ index is built in more than 3 days on a single node.

To illustrate the parallel speed-up of our approach, Figures 6a and 6b show the relationship between the execution time and the number of nodes. For both of our approaches, we report the total construction time with and without I/O cost (e.g., RadiusSketch-I/O is without I/O cost). The results illustrate a near optimal gain for F-RadiusSketch on the random walk dataset. For instance, the construction time is almost 60 minutes without I/O cost with 8 nodes, and drops down to 30 minutes without I/O cost for 16 nodes (i.e., the in-memory construction time is reduced by a factor of two when the number of nodes is doubled).

Figures 7a and 7b show the shuffling cost for a varying number of time series. We observe that the shuffling cost
Fig. 6: Construction time as a function of cluster size. F-RadiusSketch has a near optimal parallel speed-up on the random walk dataset.

Fig. 7: Shuffling as a function of dataset size. The shuffling costs of RadiusSketch and F-RadiusSketch increase linearly.

Fig. 8: Search time of sketch versions and iSAX2+. Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.
Fig. 9: The effect of the number of combinations on recall is roughly logarithmic and monotonically increasing (Avg. value for 1M queries).

Fig. 10: Recall of sketches and iSAX2+ (Avg value for 1M queries). Increasing the number of grids with F-RadiusSketch gives higher recall. Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.

Fig. 11: Precision of sketches and iSAX2+ (Avg value for 1M queries). Increasing the number of grids with F-RadiusSketch gives higher precision. Parallel algorithms (RadiusSketch and F-RadiusSketch) are run on a cluster of 32 nodes. Sequential algorithms (iSAX2+ and Sketch) are run on a single node.
increases linearly, which illustrates that our approaches are able to scale on massively distributed environments. We also observe a very similar shuffling cost between F-RadiusSketch and RadiusSketch. This result is mainly due to the shuffling optimization presented in Section III-D2.

C. Query Performance

Given a query \( q \), let \( TP, TN, FP \) and \( FN \) be the true positive/negative and false positive/negative results of an index, respectively. To evaluate the retrieval capacity of an index, we consider two measures:

- **Recall**: we search for the 20 most similar series to \( q \) according to the index. Then, we compare the result to a linear search with \( q \) on the whole dataset, where the top 10 similar series are returned. The number of true positive candidate series returned by the index is counted among the top 20 series given by the index.

- **Precision**: here, the same principle is applied but restricted to the top 10 series returned by the index.

In both cases, we set \( \text{precision} = V P / (V P + F P) \) and \( \text{recall} = V P / (V P + F N) \). In the following experiments, for the seismic dataset the queries are time series randomly picked from the dataset. For the random walk dataset, we generate random queries with the same distribution. For each time series \( t \) in the query, the goal is: i) to check if the approach is able to retrieve \( t \) (if it exists in the case of random walk); and ii) to find the nine other time series that are considered to be the most similar to \( t \) in the dataset.

Figures 8a and 8b compare the search time of the sketch approaches to that of iSAX2+ for answering queries with a varying size of query batch. These figures show that, in our experiments, the search time of RadiusSketch and F-RadiusSketch is better than that of the iSAX2+ by a factor of 13, e.g., the search time for 10 million queries is 2700s for iSAX2+ and 200s for F-RadiusSketch.

Figures 9a and 9b illustrate the impact of the number of combinations (i.e. number of groups) on the recall of F-RadiusSketch. We observe that the recall increases with the number of groups. For instance, when the number of groups is 256, we observe a recall of 0.80 for the random walk dataset and 0.55 for the seismic dataset. This shows the trend of the recall as a function of the number of combinations. The effect is roughly logarithmic and monotonically increasing.

Figures 10a and 10b illustrate the recall of different tested approaches, with varying parameters for the sketch approaches. For all the settings, the recall performance of sketches is higher than iSAX2+. We observe that F-RadiusSketch outperforms all the other approaches when the number of combinations is maximum. For instance, with 256 groups, the recall of F-RadiusSketch is up to 80%, while that of iSAX2+ is 26%.

The same experiment has been done to study precision, with very similar results as reported in Figures 11a and 11b.

V. Conclusion

RadiusSketch is a simple-to-implement high performance method to perform similarity search at scale. It achieves better runtime performance and better quality than its state-of-the-art competitor iSAX2+ in a sequential environment. Further, RadiusSketch parallelizes naturally and nearly linearly. In future work, we will extend this work to data streams.

REFERENCES


