Parallel Time Series Join Using Spark

Chuitian Rong[a,b], Lili Chen[a], Yasin N. Silva[c,*]

[a] *School of Computer Science and Technology, Tianjin Polytechnic University*
[b] *Tianjin Key Laboratory of Autonomous Intelligence Technology and Systems*
[c] *Arizona State University, Tempe, AZ 85281, USA*

Abstract

A time series is a sequence of data points in successive temporal order. Time series data is produced in many applications scenarios and the techniques for its analysis have generated substantial interest. Time series join is a primitive operation that retrieves all pairs of correlated subsequences from two given time series. As the Pearson correlation coefficient, a measure of the correlation between two variables, has multiple beneficial mathematical properties, for example the fact that it is invariant with respect to scale and offset, it is used to measure the correlation between two time series. Considering the need to analyze big time series data, we focus on the study of scalable and distributed techniques to process massive datasets. Specifically, we propose a parallel approach to perform time series joins using Spark, a popular analytics engine for large-scale data processing. Our solution builds on (1) a fast method to compute the Fast Fourier Transform (FFT) on the times series to calculate the correlation between two time series, (2) a loss-less partition method to divide the time series into multiple subsequences and enable a parallel and correct computation of the join result, and (3) optimization techniques to avoid redundant computations. We performed extensive tests and showed that the proposed approach is efficient and scalable across different datasets and test configurations.

Keywords: Time Series Join; Parallel; Partition; Spark

*Yasin N. Silva
Email address: ysilva@asu.edu (Yasin N. Silva)

1. Introduction

A time series is a sequence of data items or observations that are ordered in time. Time series datasets are produced in a variety of applications, including financial data analysis [1], medical and health monitoring [2, 3], and industrial automation applications [4]. In the financial domain, time series can keep track of stock prices, currency rates or commodity prices over time. Using this data, organizations can detect outliers, find patterns and forecast future variations. In the medical field, time series are generated by various monitoring systems, such as ECG and EEG devices. These time series could be used to better diagnose patients by comparing a patient's time series with the ones in a database of known patterns. Moreover, the rapid expansion of IoT (Internet of Things) technologies that produce large amounts of time series data creates the need for efficient mechanisms to process and analyze large time series.

Time series join is a primitive and useful operation in time series data analysis. This operation aims to identify all pairs of correlated subsequences from two given time series. The operation requires a measure to compute the similarity or correlation between two subsequences. The Pearson correlation coefficient is a commonly used similarity measure for time series due to its multiple beneficial mathematical properties, such as the fact that it is invariant with respect to scale and offset. The Pearson coefficient can reveal the true similarity between two sequences using Z-normalization, and represent a fair correlation comparison using length normalization. The correlated subsequences generated using the Pearson coefficient can provide more useful information than other similarity measures, such as ED (Euclidean Distance), DTW (Dynamic Time Warping)[5], and LCSS (Longest Common Subsequence) [6]. Particularly, the generated correlated subsequences are robust enough to perform additional analysis tasks such as data classification, clustering, forecasting, pattern discovery, and outlier detection. In this paper, we adopt the use of the Person correlation coefficient as the similarity measure.

While time series datasets can vary in size, this paper focuses on the study of

highly scalable mechanisms to process very large time series data. These mechanisms are of particular interest considering that many industrial applications generate very large time series and the fact that performing time series analysis on these datasets using a single computer is often inefficient. Specifically, we study time series joins on Apache Spark, a primarily in-memory big data processing framework. Spark uses the RDD (Resilient Distributed Datasets) as its primary data structure and supports a rich set of data operations. Unlike Hadoop, that writes intermediate results into the distributed file system (HDFS), Spark keeps intermediate results in memory and makes it easy to share data among parallel jobs.

Joining two time series based on correlation can be computationally expensive. The naive algorithm takes $O(n^4)$, where n is the length of the time series. The previously proposed Jocor approach requires $O(n^2 \log n)$[7]. In this paper, we propose a parallel algorithm that can find out the correlated segments faster than non-parallel join approaches with the time complexity of $O(n^2 \log n)$. In order to achieve this goal, we devise a parallel FFT algorithm to compute the shifted cross products between two time series. Furthermore, we propose a lossless partition method for time series segmentation and shifted-cross-product matrix partitioning to eliminate data redundancy and unnecessary computations. Our proposed parallel algorithm can efficiently perform joins on very long time series, or many short time series. Also, the proposed parallel algorithm can be easily extended to other similarity functions by changing the similarity measures when computing the similarities of all possible subsequences.

The main contributions of this paper are:

1. We devised a parallel FFT algorithm to enable the efficient computation of Pearson correlation coefficients.
2. We designed new partitioning methods for time series segmentation and shifted-cross-product matrix decomposition.
3. We implemented the proposed parallel time series join algorithm on Spark. In order to improve the efficiency further, we designed and implemented

mechanisms for removing redundant data.

4. We conducted extensive experiments on public real data sets to verify the efficiency and scalability of the proposed algorithms.

The remaining part of the paper is organized as follows. Related Work is discussed in Section 2. In Section 3, we present the problem definition and background concepts. Section 4 discusses the details of the parallel FFT algorithm. Section 5 presents the methods for time series segmentation and shifted-cross-product matrix partitioning. Section 6 presents the proposed parallel time series join algorithm. The experimental results are shown in Section 7 and our conclusion in Section 8.

2. Related Works

Time series join is a basic operation in time series data analysis. It can provide useful information for time series classification, clustering, forecasting, motif discovery and outlier detection[8]. Most of the related work focuses on the subsequence search problem. There is little work focusing on the join problem due to its high computational complexity. The existing work can be organized into several categories.

One such category is composed of methods that use ED (Euclidean Distance)[9], DTW (Dynamic Time Warping)[5], and LCSS (Longest Common Subsequence) [6] as the similarity measures. These methods operate on raw data using normalization techniques [10, 11] and consequently are vulnerable to variations of scale and offset. The algorithms using ED require that the two time series be of equal length. These methods could produce large errors when processing time series with different phases. The algorithms based on DTW can lead to unintuitive alignments, where a single point on one time series maps onto a large subsection of another time series. Also, this approach is sensitive to noise and computationally expensive[12].

Another category of related work is composed of methods that apply Jaccard similarity [13] as the similarity measure. This approach transforms the

time series into a set and applies the Jaccard metric to calculate the similarity between two sequences. Since the Jaccard metric has been proposed to compute similarities of finite sets, this approach is not efficient with very long time series.

Many of the contributions in this area aim to retrieve similar subsequences with length constraints. Some techniques focus on finding the longest correlated sequences between a query and a long time series. In[14], the authors propose an approach that uses an index and an α-skip method to find the longest segment with a correlation (with respect to the query) above a given threshold.

In recent years, several algorithms, such as Jocor[7] and LCS-Jocor[15], have addressed the problem of time series join. The time complexity of Jocor is $O(n^2 \log n)$, where n is the length of the time series. LCS-Jocor transforms the time series into a string using PAA and SAX[16], finds the longest common substrings (LCS), and computes the correlations of corresponding subsequences. In[17], the authors introduce Matrix Profile, a data structure for time series data analysis and related work exploiting this data structure [18]. The Matrix Profile contains, in fact, the pre-computed similarities of all possible subsequences in the time series. Based on Matrix Profile, all similar sub time series can be retrieved. The computational complexity of computing the Matrix Profile is $O(n^2 logn)$. The authors also extended their work by using a GPU to accelerate the time series join, and the time complexity of computing Matrix Profile is reduced to $O(n^2)$. [19, 20]. In [21], the authors proposed a new algorithm SCRIMP++ running on single machine and accelerated by GPU. While, SCRIMP++ applied the ED as the similarity measure and cannot support range queries.

Recently, distributed and parallel platforms have been widely used for large scale data analysis. Some of works are focusing on performing join operations using distributed and parallel platforms. The works in [22, 23, 24, 25] have explored the efficient way to perform set similarity joins. The parallel theta join using MapReduce that to join two data sets on attributes like in relational databases is explored in [26, 27]. The similarity joins on high dimensional data using Spark is studied based on data representation and vertical partition techniques [28]. However, our approaches aims to find correlated segments from

Table 1: Symbols And Definition

Symbol	Definition
T	Time Series T
$\|T\|$	The length of time series T
n_1	The length of time series T_r
n_2	The length of time series T_s
L	The length of correlated subsequence
L_{min}	The smallest length of correlated subsequence
l	The length of the subsequence in each segment
$C(T_r^{'}, T_s^{'})$	Correlation coefficient between subsequences $T_r^{'}$ and $T_s^{'}$
$W_(m/2)^{nk}$	Butterfly coefficient of FFT
$X(k)$	The result of FFT
I	The index of a data point in the time series
k	The segment index during time series segmentation
$\mathcal{Z}$	The Shift-Cross-Product Matrix of two time series

two time series. In [29], it introduced predictive analytics approaches based on time series and neural network using MapReduce framework. These methods are mainly applicable to precision agriculture. The work in [30] proposed different methods for predicting big time series combined Spark's MLlib library for machine learning. These methods are different from our methods, so we do not go into details.

FFT algorithm is widely used in many industries for data tranformation. Also, there are many related works on parallel FFT, such as BLAS [31]. In BLAS, the parallel FFT is implemented by MPI. In this work, we proposed a implementation for parallel FFT algorithm that is different from that in BLAS. In this work, the parallel FFT algorithm is implemented on Spark cluster by utilizing the relationships of butterfly coefficient with the data index. The butterfly coefficient for each element is computed paralleled only based on the data index in the original data sequence. We will discuss the details in Section 4.

In summary, existing algorithms for time series joins have high time complexity and most produce inaccurate results. In this paper, we propose an efficient parallel time series join method on Spark. Our approach achieves better performance by relying on a parallel FFT algorithm and effectively integrating data partitioning techniques.

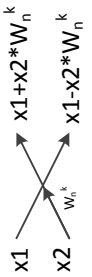


Figure 1: Butterfly Operation

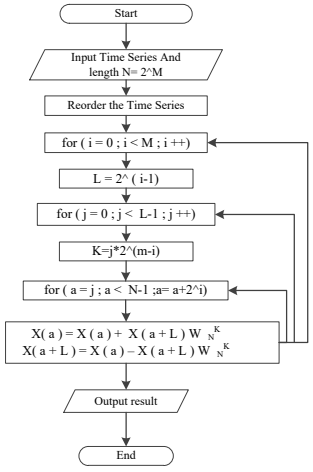


Figure 2: Traditional FFT

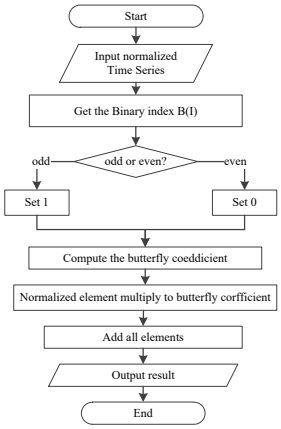


Figure 3: Parallel FFT

3. Problem Definition and Background

In this section, we present the problem definition and introduce the required background concepts. Table 1 describes the symbols used in this paper.

Definition 1. *(Time Series) A Time Series T is a sequence of data points in temporal order. $T = [(t_1, d_1), (t_2, d_2), ..., (t_n, d_n)]$, where d is the data item at time t, and n is the length of the time series.*

Definition 2. *(Subsequence) A Subsequence $T[j : j+m] = [d_j, d_{j+1}, ..., d_{j+m-1}]$ is a set of continuous points in T starting at position j with length m. In this paper, we just consider the data values of the time series.*

Definition 3. *(Pearson Correlation Coefficient) The Pearson correlation coefficient is a measure of the correlation between two variables. It measures the degree of correlation between two time series. Given two time series T_r and T_s, the Pearson correlation coefficient is computed using Formula 1.*

$$C(T_r, T_s) = \frac{(E[(T_r - E(T_r))(T_s - E(T_s))])}{(\sigma_{T_r}\sigma_{T_s})} \quad (1)$$

The Pearson correlation coefficient is not a metric and it range is [-1, 1]. When the coefficient value is closer to 1, the two time series are more positively correlated. When the coefficient is closer to -1, the two time series are more negatively correlated. The positive correlation can better reflect the similarity between two time series. In this paper, unless explicitly stated, we only consider

positively correlated subsequences. Observe that the correlation coefficient can be computed using Formula 2, where u_{T_r}, μ_{T_s} and σ_{T_r}, σ_{T_s} are the mean and Standard deviation of T_r and T_s respectively, and the $\sum T_r T_s$ can be calculated from the dot product between the subsequences of the two sequences.

$$C(T_r, T_s) = \frac{(\sum T_r T_s - m\mu_{T_r}\mu_{T_s})}{(m\sigma_{T_r}\sigma_{T_s})} \tag{2}$$

In order to guarantee scale and offset invariance, we normalize the time series using *Z-Normalization* before computing the correlation coefficient. The *Z-Normalization* is computed using Formula 3.

$$Norm(T) = \frac{(T - \mu_T)}{\sigma_T} \tag{3}$$

Definition 4. *(Time Series Join) Given two time series T_r and T_s with the same resolution, a correlation coefficient threshold θ and the minimum length L_{min}, T'_r and T'_s are subsequences of T_r and T_s, respectively, the Time Series Join on T_r and T_s aims to retrieve all possible subsequence pairs that satisfy $C(T'_r, T'_s) \geq \theta$, $|T'_r| \geq L_{min}$ and $|T'_s| \geq L_{min}$.*

Definition 5. *(Shift-Cross-Product Matrix) Given two time series T_r and T_s, the Shift-Cross-Product Matrix $\mathcal{Z}$ stores the cross products of arbitrary subsequences in T_r and T_s. The size of $\mathcal{Z}$ is $n_2 * 2n_1$, where n_2 is the length of T_s and n_1 is the length of T_r. The purpose is to reuses the computation when compute the correlation between two time series, use a matrix caching the shift cross product between two time series[7].*

Definition 6. *(Fast Fourier Transform) FFT is an efficient algorithm to compute the DFT (Discrete Fourier Transform) of a sequence, and the earliest radix-2 FFT was introduced in [32]. FFT is used to compute the cross products of arbitrary subsequences of two time series. By doing this, the computational time complexity of $\sum T_r T_s$ in Formula 2 can be reduced to $O(n^2)$. FFT can be computed as follows:*

$$X(k) = \sum_{n=1}^{m/2} x(2n) W_{m/2}^{nk} + W_m^k \sum_{n=1}^{m/2} x(2n+1) W_{m/2}^{nk} \tag{4}$$

$$X(k + \frac{m}{2}) = \sum_{n=1}^{m/2} x(2n) W_{m/2}^{nk} - W_m^k \sum_{n=1}^{m/2} x(2n+1) W_{m/2}^{nk} \tag{5}$$

Where $W_{m/2}^{nk} = \exp^{(-i * \frac{2\pi * (nk)}{m/2})}$ (i denotes the complex unit). $x(2n)$ is an element in the original sequence and $X(k)$ is the element after transformation. The range of k is $k = 0, 1, ..., m/2$. In this paper, we devise a parallel FFT algorithm to improve the efficiency of cross product computation.

4. Parallel FFT

FFT algorithm can reduce the computational time complexity of DFT from $O(n^2)$ to $O(n \log n)$. In this paper, the use of the Pearson correlation coefficient as the similarity measure, creates the need to compute the cross products of subsequences. In order to perform this task efficiently, we devised a parallel FFT algorithm.

4.1. Traditional FFT Algorithm

The basic idea of FFT is to divide a sequence into two equal parts iteratively. The odd part $x_1(n)$ contains the data items with odd index and the even part $x_2(n)$ the items with even index. $x_1(n)$ and $x_2(n)$ are computed using equations 4 and 5 [33], respectively. Formulas 4 and 5 are known as the butterfly operation. The $W_{m/2}^{nk}$ component in both equations is known as the butterfly coefficient. Figure 1 shows the butterfly operation between two elements. The traditional FFT algorithm processes the sequence by generating each possible pair of odd and even parts iteratively using formulas 4 and 5, as shown in Figure 2.

The algorithmic steps of FFT are given in Figure 2. Using the time series T = [3,5,9,11,14,1,7,10] as an example, the algorithm first divides the sequence into the odd and even parts iteratively. This steps enables the generation of a new ordered sequence T' = [3,14,9,7,5,1,11,10]. Finally, the butterfly operation is performed on T' and this generates a new sequence, as shown in Figure 4.

4.2. Parallel FFT Algorithm

In the FFT algorithm, we can observe that the butterfly coefficient for each element is related to its original index in the time series. Considering this, for each $\mathcal{I}^{th}$ element d_I, we can generate a binary code $\mathcal{B}(\mathcal{I})$ by iterative computing $\mathcal{I} = \frac{\mathcal{I}}{2^i}$, where i is the number of the iterative computation (which is determined by the length of the time series). In the i^{th} iterative computation, the i^{th} bit in $\mathcal{B}(\mathcal{I})$ is set to 0 when $\mathcal{I}$ is even, and 1 otherwise. This produces a binary code $\mathcal{B}(\mathcal{I})$ for each element. In fact, this binary code contains the position of the element in each iterative computation using the traditional FFT. Using this

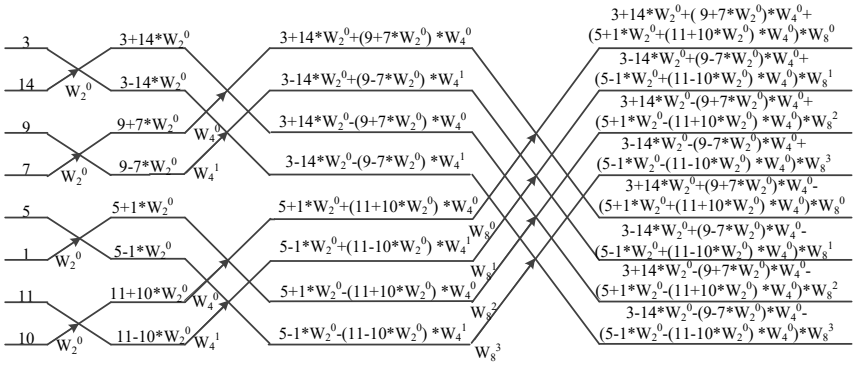


Figure 4: An example of traditional FFT

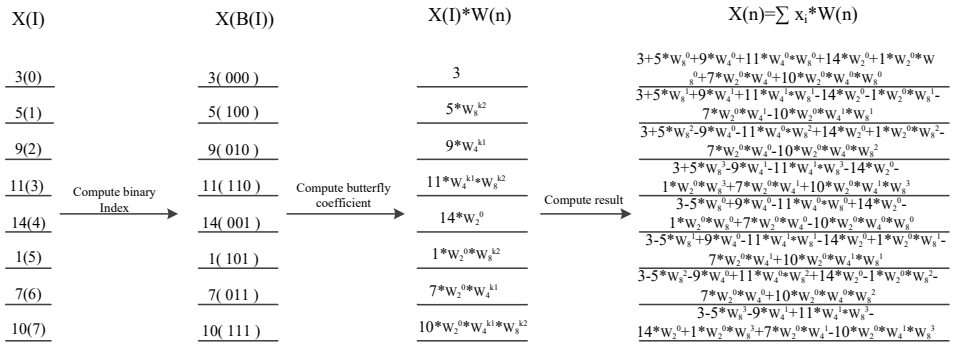


Figure 5: An example of Parallel FFT

information, we can compute the butterfly coefficient for each element independently. For each element d_I and its butterfly coefficient, the generic computing formulas can be reduced to Formula 6 and Formula 7. In Formula 6, j represents the bit positions set to 1 in the binary code $\mathcal{B}(\mathcal{I})$.

$$W(n) = \prod W_{2^j}^k * (-1)^t, k \in [0, 2^{j-1}) \qquad t = \begin{cases} 0 & \text{n=k} \\ 1 & \text{n=k+}2^{j-1} \end{cases} \tag{6}$$

$$X(n) = \sum_{i=0}^{n-1} x(i) * W(n) \tag{7}$$

In the above generic formulas, the only parameter is the element's index $\mathcal{I}$. Based on these formulas, the transformed value of each element can be computed independently. Thus, we can parallelize the FFT computation by removing the iterative computations in the traditional FFT approach.

The parallel FFT algorithm has four steps. First, it gets the original index $\mathcal{I}$ of each element. Second, it transforms the value of index $\mathcal{I}$ into binary code $\mathcal{B}(\mathcal{I})$ and gets all the bit positions that are set to 1 in $\mathcal{B}(\mathcal{I})$. Third, it computes the butterfly coefficient using Formula 6 based on the value of bit positions obtained in the previous step. Finally, it computes the transformed value of each element using Formula 7. As the process for each element is independent, all the steps are implemented in a parallel manner. The steps of parallel FFT are shown in Figure 3.

Using the sample time series T = [3,5,9,11,14,1,7,10], according to Figure 3, we get the original indices for each element in T as [0,1,2,3,4,5,6,7] and their binary codes. Specifically, for element 11 with index $\mathcal{I}$=3, the binary code $\mathcal{B}(3)$ is '110'. Its butterfly coefficient is $W(n) = \prod(\exp^{(-i(\frac{2\pi}{2^j})*k)} * (-1)^t)$, and the range of k and n are [0, 2^{j-1}) and [0, k+2^j/2], respectively. From $\mathcal{B}(3)$ = '110', we can identify that the 3^{rd} and 2^{nd} bits are set to '1'. So, $j \in \{3, 2\}$ and $W(n) = (\exp^{(-i(\frac{2\pi}{2^2})*k)} * (-1)^t) * (\exp^{(-i(\frac{2\pi}{2^3})*k)} * (-1)^t)$. In Figure 5, we use the abbreviation $W(n) = W_4^{k1} * W_8^{k2}$ for simplicity. All the elements can be transformed in a similar way. Finally, all the transformed results are aggregated to obtain the new sequence. Figure 5 shows the details of this example.

Parallel FFT vastly improves the efficiency of computing the shifted-cross-product of arbitrary subsequences of two time series in Formula 2. The pseudo code of the parallel FFT algorithm is given in Algorithm 1. This approach replaces the iterative computations in traditional FFT by the simultaneously computation of the butterfly coefficient for each element. The approach then aggregates all transformed elements using a linear scan.

5. Time Series Segmentation and Matrix Decomposition

In this section, we describe key techniques that enable a parallel implementation of the time series join. An initial challenge is to partition a time series into multiple segments and properly use the capabilities of Spark's RDDs.

Algorithm 1: $map - FFT(T)$

Input : T: the normalized time series
Output: X: the transformed time series

```
1 foreach d_I ∈ T do
2     d_I ← getComplex(d_I);
3     B(I) ← getBinaryCode(I);
4     for n = 0 → |T| do
5         W(n) ← ∏ W_{2^j}^k * (−1)^t;
6         X_n ← ∑ x_i * W(n);
```

5.1. Time Series Segmentation

As previously stated, RDDs are the main data structure in Spark. The system provides a high level of fault tolerance in the presence of nodes failures. In Spark, intermediate results are stored in memory reducing the disk I/O overhead. The data of an RDD is divided into different partitions and distributed across different nodes. Considering Spark's properties, we divide the two joined time series into a number of equal sized segments. The length and number of generated segments for each time series do not need to be the same.

In order to guarantee the accuracy of the result, we require that adjacent segments should have an overlap of at least L_{min} elements. The index range of elements in a segment can be calculated by Formulas 8 and 9. I_{min} and I_{max} are the minimum and maximum indices of one segment. If the length of the time series $n \leq I_{max}$, the index range in the last segment is $[I_{min},\ n]$. k_{max}, the number of segments generated for one time series, can be computed using formula 10. Figure 6 visually represents the segmentation of a time series.

In Spark, we use the *map*, *partitionBy* and *groupByKey* operators to implement the data partitioning process as shown in Algorithm 2. All the key/value pairs are stored as RDD elements. As presented in Algorithm 2, the first step transforms the time series elements into $\langle key, value \rangle$ pairs using the *map* operator (Lines 1-2). Here, the *value* is the time series element and the *key* is its index. And assign partition number k for each $\langle key, value \rangle$ according to Formulas 8 and 9, and transform it to $\langle k, \langle key, value \rangle \rangle$ (Lines 3-5). Then, *partitionBy* operator is used to partition time series according to k (Line 6). Finally, the

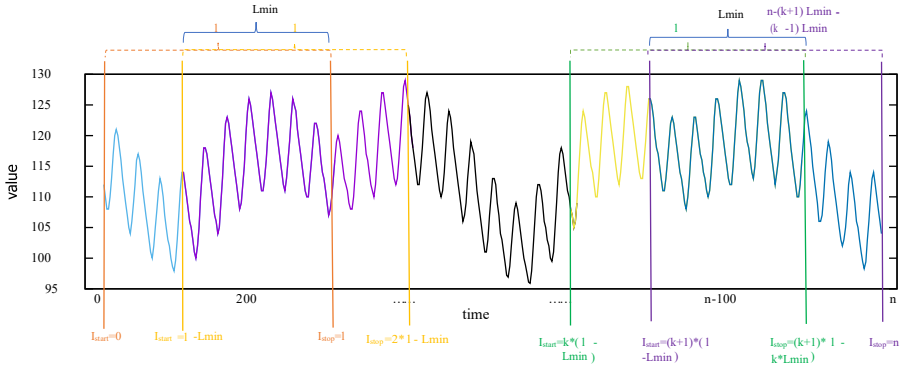


Figure 6: Time series segmentation

Algorithm 2: $Segmentation(T_r, k_{max}, l, L_{min})$

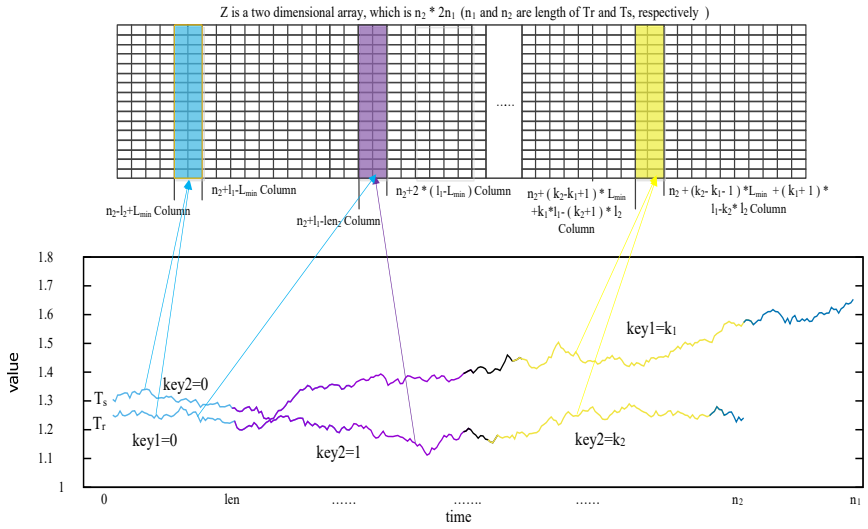
Input : T_r : the normalized time series
k_{max} : the number of partitions
l : the length in each partition
L_{min} : the minimum length of correlated segments

1 **foreach** $T_r(I) \in T_r$ **do**
2 $\quad T_r(I) \leftarrow < key, T_r(I) >$;
3 **for** $k = 0 \rightarrow k_{max}$ **do**
4 $\quad$ **if** $key \geq k * (l - L_{min}) \&\& key \leq (k+1) * l - k * L_{min}$ **then**
5 $\quad\quad midresult \leftarrow \langle k, \langle key, T_r(I) \rangle \rangle$;
6 $result \leftarrow midresult.partitionBy(k_{max})$;
7 $result \leftarrow result.gorupByKey()$;

$groupByKey$ operator is used to aggregate the new key/value pairs and keep the elements ordered (as in the original time series) by their k (Line 7). Each time series is processed in a similar way. After this, we aggregate the subsequences using the $join$ operator. This last step generates records of the form $\langle (k_1, k_2), (Iterable[T'_r], Iterable[T'_s]) \rangle$, where k_1 and k_2 are the indices of two subsequences.

$$I_{min} \geq k * (l - L_{min}) \tag{8}$$

$$I_{max} \leq (k+1) * l - k * L_{min} \tag{9}$$



T_r and T_s are two time series and assume $|T_r| > |T_s|$. T_r and T_s are divided into k subsequences.

Figure 7: Decomposition of the product matrix

$$k_{max} = \frac{n - L_{min}}{l - L_{min}} + 1 \tag{10}$$

5.2. Shift Cross Product Matrix Decomposition

The Shift Cross Product Matrix is generated by computing the shift cross products of arbitrary subsequences of two time series. The matrix $\mathcal{Z}$ is used to store the shift cross products of two subsequences that could have any starting positions and lengths. We can observe that when computing the Pearson correlation coefficient of two subsequences using Formula 2, only a small fraction of the matrix is used. To address this, we partition the matrix into multiple blocks to enable effective access to the matrix parts that are used during the coefficient computation. Figure 7 shows an example of matrix decomposition. For any two subsequences that start at indices i and j with length len in both time series, the corresponding part in the shift-cross-product is located in the range of $\mathcal{Z}[i][n_2 - i + j] - \mathcal{Z}[i + len][n_2 - i + j]$, where n_2 is the length of the smaller time series. According to this formula, we can get the corresponding block from $\mathcal{Z}$ for each segment pair. Without losing generality,

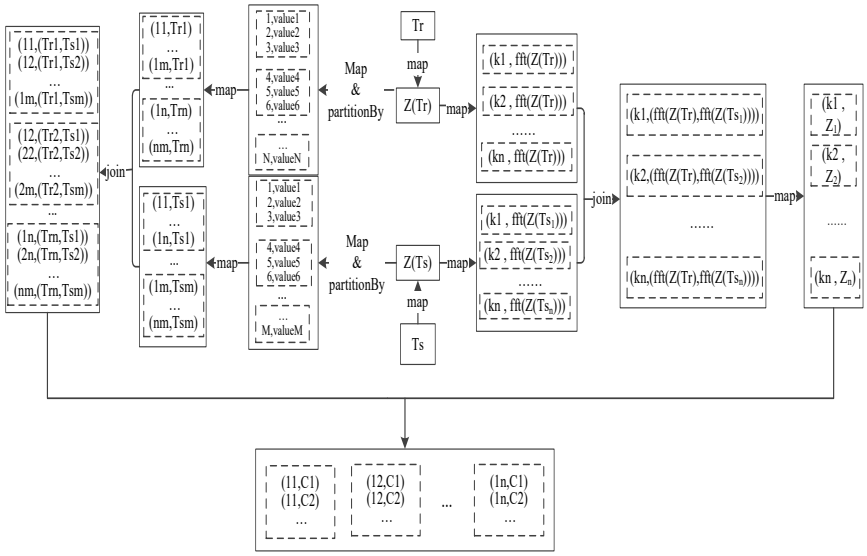


Figure 8: Data Flow in the Parallel Time Series Join

we assume that the indices of two segments are k_1 and k_2, and their lengths are l_1 and l_2, respectively. Then, their corresponding shift-cross-product $\mathcal{Z}$ is located between columns $n_2 + (k_2 - k_1 + 1)L_{min} + k_1 * l_1 - (k_2 + 1) * l_2$ and $n_2 + (k_2 - k_1 - 1)L_{min} + (k_1 + 1) * l_1 - k_2 * l_2$. There are three special cases. First, when the segment is the last subsequence in the longer time series, their corresponding block in $\mathcal{Z}$ is between columns $n_2 + (k_2 - k_1)L_{min} + k_1 * l_1 - (k_2 + 1) * l_2$ and $n_2 + (k_2 - 1)L_{min} + n_1 - k_2 * l2$. Second, if one of the segment is the last subsequence in the shorter time series, their corresponding block is between columns $(1 - k_1)L_{min} + k_1 * l_1$ and $n_2 + (k_2 - k_1)L_{min} + (k_1 + 1) * l_1 - k_2 * l_2$. Third, if two segments are the last subsequences in two time series, the corresponding block is between columns $(1 - k_1)L_{min} + k_1 * l_1$ and $n_2 + n_1 + (k_2 - 1)L_{min} - k_2 * l_2$. Figure 7 shows these three cases.

6. Parallel Time Series Join

In this section, we present the implementation details of the parallel time series join in Spark. Our approach builds on the proposed parallel FFT and data partition techniques with time complexity $O(n^2 \log n)$, where n is the length of

Algorithm 3: $map - Product(T_r, T_s)$

```
   Input  : T_r, T_s: the normalized time series
   Output: z: the dot product matrix
1  N ← T_r.length, m ← T_s.length;
2  for i = 0 → m do
3  |  T_s ← T_s.takeRight(m − i);
4  for j = 0 → 2*N do
5  |  T_r ← Array[T_r, 0];
6  |  T_s ← Array[T_s, 0];
7  T_R ← FFT(T_r), T_S ← FFT(T_s);
8  Z ← T_R * T_S;
9  z ← iFFT(Z);
```

time series. The data flow under our algorithm is represented in Figure 8. In this figure, T_r and T_s are the two time series with lengths N and M ($N \geq M$), respectively, T_r^i and T_s^j are the i^{th} and j^{th} segments in the time series, and C_{ij} is the correlation coefficient of the subsequences. The Parallel Time Series Join algorithm has the following main steps.

1. Preprocessing. The time series is normalized (Z-Normalization) in order to eliminate the effects of scale and offset.
2. Cross product computation. The cross product of arbitrary subsequences in the time series is computed using the parallel FFT algorithm.
3. Data partitioning. Each time series is split into multiple segments.
4. Parallel processing of the segment pairs to identify correlated subsequences with $C_{ij} > \alpha$.
5. Aggregation of the previous step's results.

Computing the cross product of arbitrary subsequences. This step, described in Algorithm 3, was inspired by [7] and uses a parallel FFT algorithm to reduce its complexity. The algorithm gets first all subsequences from T_s and then extends all sequences to have twice the length of the longer time series (Lines 2-6). Next, it transforms each time series using parallel FFT (Line 7). Finally, it computes the cross products using the transformed time series and produces the shift cross matrix $\mathcal{Z}$ (Line 8).

Algorithm 4: $map - comCorrelation(list, L_{min}, \mathcal{Z})$

Input : $list$: $\langle (Iterable[T'_r], Iterable[T'_s]) \rangle$
L_{min} : the minimum length of correlated segments
$\mathcal{Z}$: two dimensional array computed using Algorithm 3
Output: *result*: the correlated segments and their correlation coefficients

```
1  n ← list._1.length, m ← list._2.length;
2  for i = 0 → m do
3      for j = 0 → n do
4          maxLength ← min(m − i + 1, n − j + 1);
5          len ← L_min;
6          while len < maxLength do
7              SumT'_r T'_s ← Z(i)(m − i + j) − Z(i + len)(m − i + j);
8              mean ← getMean(list._1, list._2);
9              stdv ← getStdv(list._1, list._2);
10             C ← (SumT'_r T'_s − len ∗ mean) / (len ∗ stdv);
11             if C > α then
12                 f ← (len/(1+len) + len/(1+len)^2 ∗ maxV^2)^{-1};
13                 stepSize ← (log (1−α)/(1−C)) ÷ (log f − 1/len);
14             len ← len + stepSize + 1;
15         result ← (i, j, len, C);
```

Computing correlation coefficients. This step generates all possible time series segments (using the time series segmentation approach from Section 5.1) and identifies the highly correlated subsequence pairs (one from each time series) using Algorithm 4. Algorithm 4 first gets the number of subsequences in each partition(Line 1). Then, it calculates the length range of segment pairs (Lines 4-5) and uses equation 2 and the two-dimensional array produced by Algorithm 3 to compute the correlation coefficients for every length at location of the two subsequences (Lines 7-10). Finally, it records the location, length and correlation value for subsequence pairs where $C_{ij} > \alpha$ (Lines 11-15).

Optimization. When computing the Pearson correlation coefficients, we use the shift cross matrix $\mathcal{Z}$ produced by Algorithm 3. We observed, however, that for each segments pair only a small block in the Shift Cross Product Matrix $\mathcal{Z}$ is needed. Considering this, we partition the matrix $\mathcal{Z}$ into blocks such that for a given segment pair we can access only the relevant matrix blocks and avoid data and computational redundancy. The method for shift cross matrix

partitioning was introduced in section 5.2. Regarding the join algorithm, only a minor change in Algorithm 4 is needed: parameter $\mathcal{Z}$ should be changed to the corresponding $\mathcal{Z}$ block associated with the segments pair.

7. Experimental Evaluation

In order to verify the efficiency and scalability of the proposed methods, we conducted extensive experiments on three public datasets. We varied experimental parameters such as time series length, number of computing nodes and minimum length of subsequences (L_{min}). We evaluated the initial method (PTSJ) and the optimized approach (B-PTSJ) integrating data partitioning techniques. The tests used two time series with different lengths.

7.1. Experimental Settings and Datasets

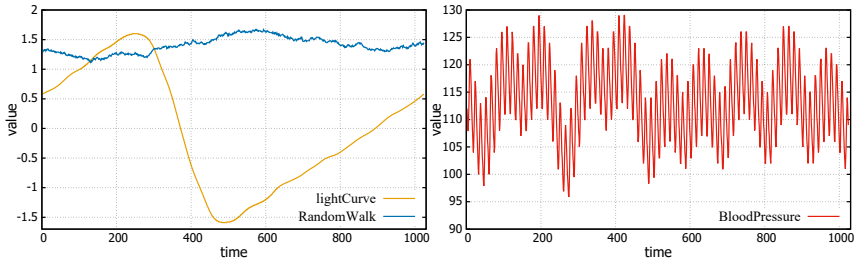
Our experiments were conducted on Spark-2.1.0 using the Scala programming language. The cluster had one master node and 8 worker nodes. The configuration of each node was: Linux CentOS 6.2, 4 cores, 6 GB of memory and 500 GB of disk space. The datasets were originally stored in HDFS and the final results were stored back in HDFS. The three datasets used in the experiments are public datasets site[1] that contain many small time series. We created two long time series using these short time series. The dataset values and statistics are shown in Figure 9 and Table 2, respectively.

Light Curve: It contains the light curve time series of 8000 stars [34]. Each has 1000 observations associated with a given star.

Blood Pressure: It contains 100 blood pressure sequences collected using the salt sensitivity test [35]. Each time series has 2000 observations.

Random Walk: This data set was synthesized from Random Walk dataset. It contains 4 time series, each has 8000 observations.

[1] https://files.secureserver.net/0fzoieonFsQcsM



(a)LightCurve and RandWalk (b)BloodPressure

Figure 9: Dataset Values

Table 2: Dataset Statistics

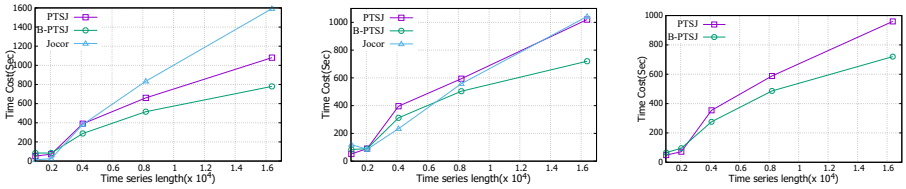
Data Set	Length	Number	Mean	Std
Light Curve	1000	1000	-2.25e-06	0.99
Blood Pressure	2000	100	113.35	6.79
Random Walk	8000	4	1.40	0.14

7.2. Effect of time series length

In this experiment, we changed the length of the time series from 1000 to 16000 by concatenating multiple time series from the same dataset. We set the minimum subsequence length L_{min}=100, the segment length $L = 300$ and the correlation threshold $\alpha = 0.9$. To compare the speed of our methods and the non-parallel approach Jocor proposed in[7]. Figure 10 shows that the execution time of Jocor, PTSJ and B-PTSJ increase when the length increases. PTSJ and B-PTSJ are more efficient than Jocor on Light Curve when the length exceeds 8000 and on Blood Pressure datasets, our B-PTSJ method is more efficient than Jocor when the length exceeds 8000. The Jocor can't compute the correlation when the length exceeds 2000 on Random Walk dataset. For methods PTSJ and B-PTSJ, when the length is 1000, the time of storing, transferring and scanning the Shift Cross Matrix $\mathcal{Z}$ is smaller than the cost of $\mathcal{Z}$ decomposition. As the length increases, the cost of processing the entire $\mathcal{Z}$ matrix also increases and this favors the optimized B-PTSJ approach.

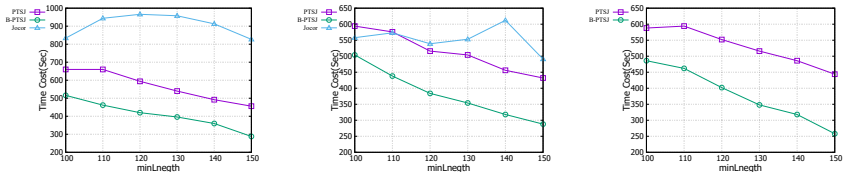
7.3. Effect of the minimum subsequence length (L_{min})

In this test, we performed joins on two time series with different lengths (8192 and 2048). The segment length is 300 and the correlation α=0.9. L_{min}



(a)Light Curve (b) Blood Pressure (c) Random Walk

Figure 10: Effect of length for different datasets



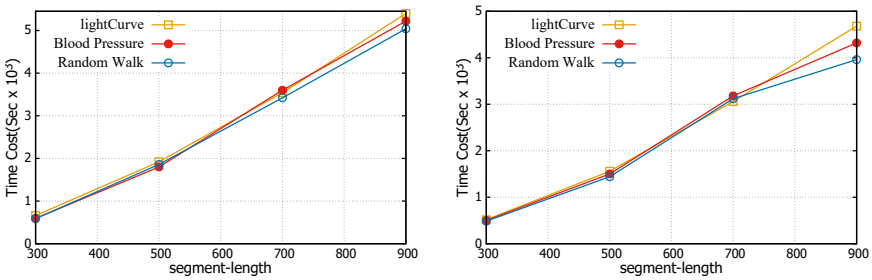
(a)Light Curve (b) Blood Pressure (c) Random Walk

Figure 11: The effect on min-length L_{min}

varies from 100 to 150. Figure 11 shows that the efficiency of the two methods improves as L_{min} increases. Based on the algorithmic descriptions presented in previous sections, we can observe that varying L_{min} will affect the produced segmentation. As L_{min} increases, the number of segments and concurrency also increase. Thus, the proposed methods are more efficient with larger L_{min} values. And form Figure 11 we can see that due to the data characteristics, the three data sets have some different in our proposed methods.

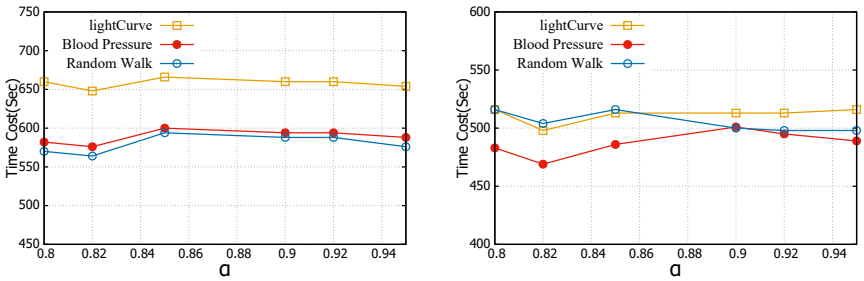
7.4. Effect of the segment length

We tested the effect of segment length in this experiment, and we vary the subsegment length of two time series experimented on our proposed methods. We tested on two time series with length are 8192 and 2048, respectively. In this experiment, we set the correlation α=0.9, L_{min}=100, and collect the results by varying the segment length from 300 to 900 with the interval 200. The results are shown in Figure 12. We can observed from Figure 12, as we increase the length of segment in each partition, the execution time is a linearly increase on three different datasets. It's due to that the two time series are partitioned into multiple segments when computing parallel FFT and performing join operations. If the length of segments is larger, the number of segments is smaller and the



(a) The PTSJ Approach (b)The B-PTSJ Approach

Figure 12: The effect on segment length



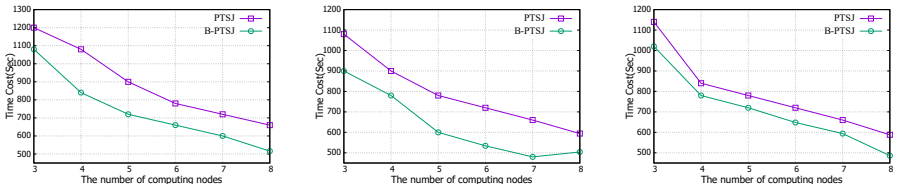
(a) The PTSJ Approach (b)The B-PTSJ Approach

Figure 13: The influence of threshold α

concurrency is lower. So, when increasing the length of segment, the concurrency is lower and the execution time increased. It's to be noted that there is a restriction when partitioning the time series. Each segment cannot less than the L_{min} and the adjacent segments should have an overlap with the length of L_{min}. So, the segments length cannot be decreased arbitrarily and the concurrency can be improved to some extent.

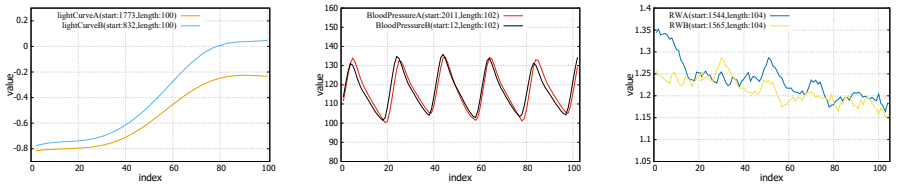
7.5. Effect of the correlation threshold (α)

In this experiment, we varied the correlation threshold from 0.8 to 0.95. The length of the time series are 8192 and 2048, the segment length is 300, and L_{min} = 100. Figure 13 shows that the variation of α does not have a significant effect on the execution time. This is the case since α does not influence the core steps of our solution including parallel FFT and partitioning.



(a) Light Curve (b) Blood Pressure (c) Random Walk

Figure 14: Effect of the number of computing nodes



(a) Light Curve (b) Blood Pressure (c) Random Walk

Figure 15: The interesting join results

7.6. Increasing the number of computing nodes

In this experiment, we join two time series with lengths 8192 and 2048, the segment length is 300, L_{min} = 100 and α=0.9. We vary the number of computing nodes in the cluster from 3 to 8. Figure 14 shows that the execution time of the proposed methods decrease linearly as the number of nodes increases. B-PTSJ outperforms PTSJ in all the tests.

To certify if our methods actually find out effective join results on real life datasets, In Figure 15, we give the interesting join segments on three datasets, and they have a correlation coefficient of 0.95, 0.98 and 0.91, respectively. For Blood Pressure dataset (Figure 15 (b)), if we observed a correlated segment, it can be used to predict of cardiac tamponade, which can cause death.

8. Conclusion

Time series join is a primitive operation for large time series analysis and is widely used in many applications. In this paper, we propose a parallel time series join on Spark that faster than the non-parallel method Jocor. To this end, we devised a parallel FFT algorithm and proposed novel partitioning mechanisms for time series segmentation and shift cross product matrix decomposition. A

thorough performance evaluation show that the proposed methods are efficient and scalable under different experimental settings. In the future, we plan to investigate parallel joins using different similarity measures.

9. Acknowledgements

This work was supported by the National Natural Science Foundation of China(No.61402329 and No.61972456) and the Tianjin Natural Science Foundation (No.19JCYBJC15400).

References

[1] R. Flanagan, L. Lacasa, Irreversibility of financial time series: A graph-theoretical approach, Physics Letters A 380 (20) (2016) 1689–1697.

[2] W. DF, C. LS, Z. X, V. MI, L. HL, Time series for blind biosignal classification model., Computers in Biology and Medicine 54 (C) (2014) 32–36.

[3] K. Samiee, c. P. Kovã, M. Gabbouj, Epileptic seizure classification of eeg time-series using rational discrete short-time fourier transform, IEEE Transactions on Biomedical Engineering 62 (2) (2015) 541–552.

[4] X. Wang, J. Lin, N. Patel, M. Braun, A self-learning and online algorithm for time series anomaly detection, with application in cpu manufacturing, in: CIKM, 2016, pp. 1823–1832.

[5] T. Rakthanmanon, B. Campana, e. Mueen, Searching and mining trillions of time series subsequences under dynamic time warping, in: SIGKDD, 2012, pp. 262–270.

[6] Abdulla-Al-Maruf, H. H. Huang, K. Kawagoe, Time series classification method based on longest common subsequence and textual approximation, in: ICDIM, 2012, pp. 130–137.

[7] A. Mueen, H. Hamooni, T. Estrada, Time series join on subsequence correlation, in: ICDM, 2015, pp. 450–459.

[8] S. K. Jensen, T. B. Pedersen, C. Thomsen, Time series management systems: A survey, TKDE 29 (11) (2017) 2581–2600.

[9] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases, SIGMOD 23 (2) (1994) 419–429.

[10] D. Yankov, E. J. Keogh, S. Lonardi, A. W. Fu, Dot plots for time series analysis, in: ICTAI, 2005, pp. 159–168.

[11] Y. Chen, G. Chen, K. Chen, B. C. Ooi, Efficient processing of warping time series join of motion capture data, in: ICDE, 2009, pp. 1048–1059.

[12] T. Fu, A review on time series data mining, Eng. Appl. of AI 24 (1) (2011) 164–181.

[13] J. Peng, H. Wang, J. Li, H. Gao, Set-based similarity search for time series, in: SIGMOD, 2016, pp. 2039–2052.

[14] Y. Li, H. U. Leong, L. Y. Man, Z. Gong, Efficient discovery of longest-lasting correlation in sequence databases, VLDB Journal 25 (6) (2016) 767–790.

[15] V. D. Vinh, N. P. Chau, D. T. Anh, An efficient method for time series join on subsequence correlation using longest common substring algorithm (2016) 121–131.

[16] A. Camerra, T. Palpanas, J. Shieh, E. J. Keogh, isax 2.0: Indexing and mining one billion time series, in: ICDM, 2010, pp. 58–67.

[17] C. C. M. Yeh, Y. Zhu, U. etc., Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile, Data Mining & Knowledge Discovery 32 (1) (2018) 1–41.

[18] C. M. Yeh, Y. Zhu, e. Liudmila Ulanova, Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets, in: ICDM, 2016, pp. 1317–1322.

[19] Y. Zhu, Z. Zimmerman, e. Nader Shakibay Senobari, Exploiting a novel algorithm and gpus to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins, Knowl. Inf. Syst. 54 (1) (2018) 203–236.

[20] Y. Zhu, Z. Z. etc., Matrix profile II: exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins, in: ICDM, 2016, pp. 739–748.

[21] Y. Zhu, e. Chin-Chia Michael Yeh, Matrix profile XI: SCRIMP++: time series motif discovery at interactive speeds, in: ICDM, 2018, pp. 837–846.

[22] R. Vernica, M. Carey, C. Li, Efficient parallel set-similarity joins using MapReduce, in: SIGMOD, ACM, 2010, pp. 495–506.

[23] C. Rong, W. Lu, X. Du, X. Zhang, Efficient and exact duplicate detection on cloud, Concurrency and Computation: Practice and Experience 25 (15) (2013) 2187–2206.

[24] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, A. K. Tung, Efficient and scalable processing of string similarity join, TKDE 25 (10) (2013) 2217–2230.

[25] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, X. Du, Fast and scalable distributed set similarity joins for big data analytics, in: ICDE 2017, 2017, pp. 1059–1070.

[26] A. Okcan, M. Riedewald, Processing theta-joins using mapreduce, in: SIGMOD 2011, 2011, pp. 949–960.

[27] X. Zhang, L. Chen, M. Wang, Efficient multi-way theta-join processing using mapreduce, PVLDB 5 (11) (2012) 1184–1195.

[28] C. Rong, X. Cheng, Z. Chen, N. Huo, Similarity joins for high-dimensional data using spark, Concurrency and Computation: Practice and Experience 31 (20) (2019).

[29] M. Bendre, R. Manthalkar, Time series decomposition and predictive analytics using mapreduce framework, Expert Syst. Appl. 116 (2019) 108–120.

[30] A. Galicia, J. F. Torres, F. Martínez-Álvarez, A. T. Lora, A novel spark-based multi-step forecasting algorithm for big data time series, Inf. Sci. 467 (2018) 800–818.

[31] Blas(basic linear algebra subprograms), http://www.netlib.org/blas/.

[32] J. Cooley, An algorithm for the machine of complex fourier series, Mathematics of Computation 19 (90) (1965) 297–297.

[33] P. Duhamel, M. Vetterli, Fast fourier transforms: A tutorial review and a state of the art, Signal Processing 19 (4) (1990) 259–299.

[34] U. Rebbapragada, P. Protopapas, C. E. Brodley, C. Alcock, Finding anomalous periodic time series, Machine Learning 74 (3) (2009) 281–313.

[35] S. M. Bugenhagen, C. A. Jr, D. A. Beard, Identifying physiological origins of baroreflex dysfunction in salt-sensitive hypertension in the dahl ss rat, Physiological Genomics 42 (1) (2010) 23.