Distributed Discord Discovery: Spark Based Anomaly Detection in Time Series

*Abstract*—Since the concept of time series discord has been put forward, many promising methods were proposed to resolve this compute-intensive problem. However, most of these efficient methods only adapted to single computing element, which means they are just single-threaded algorithms. In the background of big data technology, these single-threaded methods can’t fit well in the big dataset from real-world problems, both in computing and storing the dataset itself. In this work, we introduce a distributed discord discovery method, derived from the basic discord discovery algorithm and we make it scalable to as many computing resources as we have. Our distributed method base on the Spark computing framework and we store data in HDFS, both computing and data are distributed. We evaluated the efficiency of our method both in real-world and synthetic dataset, especially in Terabyte sized dataset.

1. INTRODUCTION

The problem of time series discord discovery is finding the most unusual or most anomalous one in a time series dataset. Since it has been put forward in [5], many promising methods were proposed to resolve this compute-intensive problem. Despite the simplicity of its definition, time series discord has a wild range of application, for example, fault diagnostics, intrusion detection, data cleaning, etc. Apparently it’s a promising technology in data mining community, and it has attracted a lot of attention.

Although the era of Big Data is coming, the method for finding the most unusual time series still focused on accelerating the single-threaded algorithm. Most of these methods can’t fully utilize the computing resources such as multi-core CPU or cluster-computing, thus these methods may not be able to handle the real-world dataset today. For example, Facebook manages more than 25 terabytes of log data daily in 2009 [2], and today maybe a medium-sized business website will have such a volume of data. Both storing and processing such large dataset will be the bottleneck of legacy method for discord discovery, such as HOTSAX [1]. In this paper we will present a distributed solution for discord discovery, which resolve the problem of storing and processing by HDFS [3] and Spark [4] respectively. With this distributed method, we will be capable of handling the terabyte or even petabyte sized of dataset provided enough computing resources. Our method outperforms any state-of–the-art discord discovery algorithm as far as we known.

Our work makes three distinguished contribution in time series discord discovery. Firstly, data independent and result combinable are two fundamental requirement of distributed computing. We make the non-combinable discord result combinable by some tiny but useful changes, which make our method fit well in the Spark computing framework. Secondly, we eliminate the memory limitation of legacy algorithm without losing performance, because our dataset now is distributed on the cluster and each computing node in cluster can handle each block of dataset in memory efficiently. Thirdly, as we have the capability of processing larger dataset, we can extend our method to multi-dimensional time series discord discovery which apparently has larger data size.

The rest of the paper is organized as follows. Section II reviews related work and gives out some notation before we introduce the whole problem. In Section III, we will show the detail of our method and the implementation of our framework. Section IV will present some evaluation from our experiment on both real-world and synthetic dataset. Finally, Section V offers some conclusions and suggestions for future work.

1. RELATED WORKS
2. *Related Works*

The definition of discord was first proposed in [5] for electrocardiogram application, and so is the legacy algorithm for discord discovery. This legacy algorithm is named as HOTSAX in [1] by the same authors. HOTSAX is an acceleration version of the basic brutal force algorithm. As the definition describes, discord is one of the time series in dataset who has the maximum nearest neighbor distance. So the brutal force algorithm just computing the pair-wise distances among all the time series. This brutal force algorithm can be accelerated several times by early abandon, and what HOTSAX does is making early abandon happen more frequently by adjusting the iteration order of both inner and outer loops. The iteration order is determined by the similarity of each time series. So HOTSAX uses a Trie [14] to store the similar time series in same branch. The acceleration of HOTSAX against brutal force algorithm is about three orders of magnitude as claimed in [1].

Since then, various enhanced algorithms based on HOTSAX have been proposed to speed up it. In [6], the authors introduced a word-size-free algorithm by using Haar wavelets instead of SAX [7] which used in HOTSAX for dimension reduction. A parameter-free discord search algorithm is proposed in [8], which is called direct discord search, and it can find the top-K discord in quasi-periodic time series. Another novel discord discovery algorithm called BitClusterDiscord is raised in [9] and it uses PAA (Piecewise Aggregate Approximation) bit serialization to segment time series and then clusters them by K-Medoids clustering algorithm which also achieves the goal as HOTSAX does by using Trie. The extension of [9] introduced GDS (general direct search) [13] to eliminate the quasi-periodicity assumption in direct search and make it another parameter-free search. Among all of the methods for speeding up HOTSAX, HOTiSAX [10] is the most direct enhancement of HOTSAX as its name indicated. Indexed SAX [11] is introduced in HOTiSAX to further accelerate the legacy one.

Apart from speeding up HOTSAX, others augment it from another point of view. For example, the authors of [12] pointed out that HOTSAX had assumed that dataset could be fit in main memory which is not true in real-world. So all the augmentation version of HOTSAX is also has that kind of limitation, which means these HOTSAX-like algorithms only perform well in small dataset. And the disk aware algorithm proposed in [12] is aimed at large dataset, however, by scanning the time series resident in disk into memory one by one, this disk aware algorithm results in extra I/O time which occupies about 50% of the total time. This is some kind of tradeoff between performance and memory limitation. Even though this disk aware algorithm is able to handle large dataset in terabyte sized, it is inefficient as it’s still single-threaded. So we can hardly speed up it when data size increases, because this disk aware algorithm is not scalable to computing resources.

In order to make the discord discovery algorithm scalable, the first thing we need to do is splitting the whole dataset into blocks and then distributing them to multiple computing nodes. After the computing nodes finish their work, the sub-result of each node should be aggregated to get the whole-result. This is the core idea of MapReduce [15]. Reducing I/O time introduced by disk aware algorithm is another optimization we need to do. So we should cache block of time series in memory instead of scanning them into memory from disk when we are processing them. Fortunately, we can obtain this capability of caching from Spark, which enhances MapReuce by introducing the feature of caching data block in memory.

1. *Notation*

First we will give some definition of our notation, which helps to define our problem clearly in later. Most of these definitions are quoted from [1] and [12].

*Time Series*: A time series *T = t1,…,tm* is an ordered set of *m* real-valued variables.

*Subsequence*: Given a time series T of length *m*, a subsequence *C* of *T* is a sampling of length *n≤m* of contiguous position from *p*, that is, *C = tp,…tp+n-1* for *1 ≤ p ≤ m-n+1*.

From [12] we know there are usually two kinds of time series dataset. In one of them the time series are generated from short distinct events, which mean there are numerous short time series. The second one simply consists of all possible subsequences extracted from the time series of a long ongoing process. The main difference between them is whether they have trivial matches [16]. The former one is generated from distinct event, so it doesn’t need to consider the trivial matches. The latter one’s subsequences are extracted from the same long series, so if two subsequences have close position *p1* and *p2* are very likely to be similar each other. From now on, we will focus on the first kind of dataset for simplicity, and our method can easily be extended to the second kind of dataset with some additional minor bookkeeping to discount trivial matches.

Now we can present the formal definition of time series discords:

*Time Series Discord*: Given a dataset *S*, the time series *C* *S* is called the most significant discord in *S* if the distance to its nearest neighbor (or its nearest non-trivial match in case of subsequence dataset) is largest, which means for any time series *M* *S* the following holds: *min(Dist(C,Q))≥ min(Dist(M,P))*, where *Q, P S* (and *Q, P* are not-trivial matches of *C* and *M* in case of subsequence datasets).

The Dist function above usually refers to Euclidean distance, the formal definition is: Given two time series Q and C of length n, the Euclidean distance between them is defined as:

*Kth Time Series Discord*: Given a dataset *S*, the time series *C S* is called the most significant *k-th* discord in S if the distance to its *k-th* nearest neighbor (or its *k-th* nearest non-trivial match in case of subsequence dataset) is largest.

1. *Problem Definition*

Given the above notation, now we can define our problem clearly.

*Input*: a large dataset consists of numerous distinct time series, and the dataset is too large to load into memory as a whole. Besides, we don’t want to scan the time series into memory one by one which will introduce much more extra I/O time.

*Output*: Top-k discords of the input dataset, which means output from the largest discord to the Kth discord.

1. Distributed Discord Discovery
2. Result-combinable discord discovery

For a distributed discord discovery, the first thing we need to do is make sub-result combinable. As mentioned above, HOTSAX’s results are non-combinable, so a divide and conquer is unlikely to efficiently find discords. Thus we need some change to make it result-combinable. What we do is changing the output from *top-K* to *range-r*. A *range-r* discord has distance to its nearest neighbor at least *r*. A simple algorithm can be used to get range-r discord:

|  |
| --- |
| function range\_r\_discord(S, r)  C=[]  for s1 in S:  nn\_dist=MAX\_FLOAT  for s2 in S:  if(s1 == s2):  continue  d=Dist(s1,s2)  if(d<r):  break and continue to next s1  if(d<nn\_dist):  nn\_dist=d  if(nn\_dist>r):  C.add(s1)  return C |

For a *range-r* problem, the results of sub-problems are definitely combinable.

If we have a large dataset *S*, we split it into *T* subset, and we can apply our *range\_r\_discord* function on each subset, thus we get *T* sub-results. These sub-results are just candidates of the final result. We need another function to verify their certificate to be discord. Before verifying, we combine the sub-results first. The verify algorithm is also simple:

|  |
| --- |
| function verify\_candidates(C, S, r)  KC=[]  for c in C:  for s in S:  if(c==s):  continue  d=Dist(c, s)  if(d<r):  KC.add(c)  return KC |

C is the combined sub-results, and we again apply this *verify\_candidate* function on each subset, each subset returns a set contains the candidates who failed in this *verify\_candidates* function. The union of all these returned results from each subset will be removed from C, and the remaining part of C becomes the *range-r* discords.

Before going on, we’ll analysis the complexity of this distributed *range-r* discord algorithm first. The *range\_r\_discord* function is obviously quadratic complexity to the size of dataset. Fortunately, the input subset *S* does not have a very large size, so this brutal force implementation wouldn’t lose too much performance against other HOTSAX-like algorithms. If better performance is really needed, we still have a chance to replace this brutal force one with HOTSAX. It seems that the *verify\_candidates* function is also quadratic complexity. However, for a well-chosen *r*, the size of candidates *C* should be as small as constant. So this function is approximately linear to the size of dataset.

Now we illustrate the data flow of the distributed discord discovery algorithm in Figure 1. The number of subset in this figure is three, but actually it depends on the size of the whole dataset. Usually we limit the size of subset, and scale the number of subset up when data size increased, as the *range\_r\_discord* function performs worse when input data size increased. With this figure of data flow, now we can present our implementation on Spark.

1. Distributed discord discovery on Spark

Spark is a distributed computing framework, and it becomes more popular than Hadoop in big data recently. Spark is capable of anything that MapReduce (computing framework in Hadoop) can do, and even much more efficient. The main idea of Spark computing is splitting large dataset into small blocks, and processing these small blocks (called RDD, Resilient Distributed Dataset) in a distributed and parallel way. After some of these small tasks have finished, Spark starts aggregating the results and reducing them into one. So our distributed discord discovery method shown in Figure1 fits well in this Spark computing framework. Before detailed description of implementation, we give out the architecture of our platform in Figure 2. Our cluster consists of 4 VMs, one of them as the master node, and the other three as slaves. The Spark driver component in master node is a submit service help to deliver our program to the Spark framework.

We implement this program as we describe above on Spark in Scala which is a preferred language on Spark. Our input dataset is resident on HDFS, so all data is already distributed across the cluster on *datanode* as shown in Figure 2. Besides, the *Spark worker* can fetch data from *datanode* locally which saves a lot of time for data transmission between each node by network. With HDFS, our Spark program doesn’t need to split the dataset itself. The only thing we need to do is give it an HDFS address referring to our input dataset. After that our Spark program can handle all other things like applying *range\_r\_discord* on each RDD, and then collecting all candidates together. On the next stage we continue to apply *verify\_candidate* on each RDD after broadcasting the collected candidate set from last stage across the cluster, finally we union the failed candidates that should be kicked out, and then we can get the true discords.

1. Empirical Evaluation
2. Performance of the Distributed Discord Algorithm

We test the performance of our distributed algorithm on large-sized dataset which is generated by inserting several anomalous time series into a random walk dataset. To compare performance with the disk aware algorithm, we are going to use dataset as large as theirs. First we list out experimental result of disk aware algorithm on large-sized dataset in Table 1.

Table Time efficiency of the disk aware algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| Examples | Dataset size | I/O time | Total time |
| 1 million | 3.57GB | 27min | 41min |
| 10 million | 35.7Gb | 4h30min | 7h52min |
| 100 million | 0.35Tb | 45h | 90h33min |

Each example is 512-length of float numbers. And as we mentioned before, I/O time becomes the bottleneck of this algorithm as it occupies half of the total time.



Figure Data flow of the distributed discord discovery algorithm.



Figure Platform architecture

Before listing out our result, we first describe our experimental environment. We have 4 VMs in cluster, and each of them has 1 CPU core of 2.13GHz and 1g memory. Ubuntu 12.04 is installed in every VM and version of Spark is 1.2.0, and version of Hadoop is 2.6.0. Replication of HDFS is 3 and block size is 64MB, and other configuration of Spark and HDFS is set as default.

In Figure 3 we show you the 2 discords we inserted in the dataset and their nearest neighbor respectively.

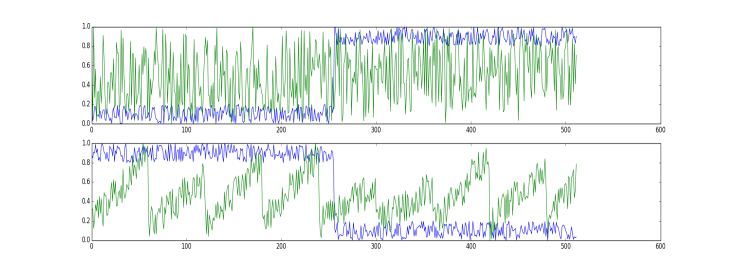


Figure Discord and its nearest neighbor

Table 2 gives out our experiment results,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Examples | Disk size | Job 1 time | Job 2 time | Total time |
| 1 million | 3.86GB | 8m56s | 2m1s | 10m57s |
| 10 million | 38.6GB | 71m19s | 18m33s | 89m42s |
| 100 million | 0.38TB |  |  |  |

1. Multi-dimensional discord discovery

REFERENCES

1. E. Keogh, J. Lin, A. Fu, “Hot SAX: Efficiently finding the most unusual time series subsequence,” IEEE International Conference on Data Mining, pp. 226-233, 2005.
2. <http://www.datacenterknowledge.com/archives/2009/10/13/facebook-now-has-30000-servers/>
3. S. Ghemawat, H. Gobioff, S. Leung, “The Google file system,” ACM Symposium on Operating Systems Principles, pp. 29-43, 2003.
4. M. Zaharia, M.Chowdhury, M. Franklin, S. Shenker, I. Stoica, “Spark: cluster computing with working sets,” USENIX conference on Hot topics in cloud computing, 2010.
5. J. Lin, E. Keogh, A. Fu, H. Van Herle, “Approximations to magic: Finding unusual medical time series,” Computer-Based Medical Systems, pp. 329-334, 2005.
6. A. Fu, O. Leung, E. Keogh, J. Lin, “Finding time series discords based on haar transform,” Advanced Data Mining and Applications, pp. 31-41, 2006.
7. J. Lin, E. Keogh, S. Lonardi, B. Chiu, “A symbolic representation of time series, with implications for streaming algorithms,” ACM SIGMOD workshop on Research issues in DMKD, pp. 2-11, 2003.
8. W. Luo, M. Gallagher, “Faster and parameter-free discord search in quasi-periodic time series,” In Advances in Knowledge Discovery and Data Mining, pp. 135-148, 2011.
9. G. Lia, O. Bräysy, L. Jiang, Z.Wud, Y. Wang, “Finding time series discord based on bit representation clustering,” Knowledge Based System, pp. 243-254, 2013.
10. H. T. Q. Buu, D. T. Anh. “Time Series Discord Discovery Based on iSAX Symbolic Representation,” Knowledge and Systems Engineering, vol. 11, no. 18, pp. 14-17, 2011.
11. J. Shieh, E. Keogh, “iSAX: indexing and mining terabyte sized time series,” ACM SIGKDD international conference on KDD, pp. 623-631, 2008.
12. D. Yankov, E. Keogh, U. Rebbapragada, “Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets,” International Conference on Data Mining, pp. 381-390, 2007
13. W. Luo, M. Gallagher, J. Wiles, “Parameter-free search of time-series discord,” Journal of computer science and technology, 28(2):300-310, 2013.
14. <http://en.wikipedia.org/wiki/Trie>
15. J. Dean, S. Ghemawat, “MapReduce: simplified data processing on large clusters,” Communications of The ACM, vol. 51, no. 1, pp. 107-113, 2008.
16. B. Chiu, E. Keogh, S. Lonardi, “Probabilistic discovery of time series motifs,” In Proc. of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD’03), pp. 493–498, 2003.