Time Series Similarity Search for Streaming Data in Distributed Systems

Ariane Ziehn
DFKI GmbH, Germany
ariane.ziehn@dfki.de

Marcela Charfuelan
DFKI GmbH, Germany
marcela.charfuelan@dfki.de

Holmer Hemsen
DFKI GmbH, Germany
holmer.hemsen@dfki.de

Volker Markl
DFKI GmbH and TU Berlin, Germany
volker.markl@dfki.de

ABSTRACT

In this paper we propose a practical study and demonstration of time series similarity search in modern distributed data processing platforms for stream data. After an intensive literature review, we implement a flexible similarity search application in Apache Flink, which includes the most commonly used distance measurements: Euclidean distance and Dynamic Time Warping. For efficient and accurate similarity search we evaluate normalization and pruning techniques developed for single machine processing and demonstrate that they can be adapted and leveraged for those distributed platforms. Our final implementation is capable of monitoring many time series in real-time and parallel. Further, we demonstrate that the number of required parameters can be reduced and optimally derived from data properties. We evaluate our application by comparing its performance with electrocardiogram data on a cluster with several nodes. We reach average response times of less than 0,1 ms for windows of 2 s of data, which allow fast reactions on matching sequences.

1 INTRODUCTION

Rakthanmanon et al. [18] demonstrate that they can find a given pattern in a day-long electrocardiogram (ECG) tracing in 35 s under Dynamic Time Warping (*DTW*). Their implementation runs on a single machine and the data is a finite batch set that is completely stored. Batch data sets and single machines enable the programmer to apply traditional optimization methods to the algorithms, but nowadays batch data is often replaced by data streams. In order to handle the increasing amount and complex properties of data streams, open source frameworks for distributed stream data processing (DSPS) have appeared. Yet, many traditional optimization techniques can not be directly applied to process data streams in distributed systems. Can we still efficiently find frequent patterns in new real-life situations, where the data is not stored but streamed? And how can we leverage traditional optimization methods in modern DSPS?

In this paper we answer these questions and demonstrate that we can match ECG patterns with high accuracy in data streams with an average response time of less than 0,1 ms per window by using modern DSPS. In particular, we employ the DSPS Apache Flink [4], which can handle massive amounts of continuous data and process them in real-time with low latency. Besides the additional monitoring effort of distributed systems, which is covered by the frameworks, two other aspects need special attention when developing streaming applications on distributed systems: *ordering* and *partitioning*. These aspects are of particular importance for time series processing of the stream data in parallel jobs. Current approaches on distance calculations between two time series assume that the data is always ordered, according to the time it is produced [1, 12]. This can not be assumed in distributed systems, where data can reach the system with delays or in the worst case samples might be lost. Additionally, the usage of a distributed stream processing approach implies partitioning of the task into independent subtasks for distributed and parallel computing.

In the remainder of this paper, we present first of all the state-of-the-art by exploring related work in Section 2. Afterwards, the basic concepts for similarity search are summarized in Section 3. The necessary edits to the traditional *DTW* algorithm for its usage in distributed systems are explained in Section 4, which is followed by the experiments conducted in this work in Section 5. The conclusion in Section 6 sums up our achievements.

2 RELATED WORK

As for many modern big data analytics applications, the underlying techniques used in this paper concern several fields. The fields of main interest for our application are time series analysis, similarity search, data streams and distributed systems. We rarely find related work, which covers all of the four fields. One close example is [7], where time series correlation analysis is performed in the DSPS Apache Storm [5]. [7] faces similar problems as we do, due to the usage of DSPS for time series analysis, but considers a different analysis task. A second example is a naive distributed implementation [19] that duplicates the data to all cluster nodes. On each node, the data is matched to one of the multiple patterns. The implementation uses no DSPS and multiple shuffles of the same data to several workers causes heavy network traffic, if also multiple streams are analyzed. Thus, the techniques of [19] can not be considered for our approach.

In fact, several works already attempt time series processing and/or similarity search in modern DSPS, but most of them only deal with batch instead of stream processing [1, 8]. Still, these works need to handle distribution and present adapted techniques for time series processing we can use for our application.

Regarding similarity search in streaming on single machines, the exact matching of patterns is first tried on-line in a streaming environment in [14]. The main focus of this approach is not only the pattern matching but also its maintenance (as the pattern might change over time). The conclusion of this work is that both tasks, matching and maintenance, are feasible and can be implemented efficiently in streaming environments and thus, are considered for our approach.
Another common practice is the development and optimization of batch processing algorithms for single machines [12, 15, 16].

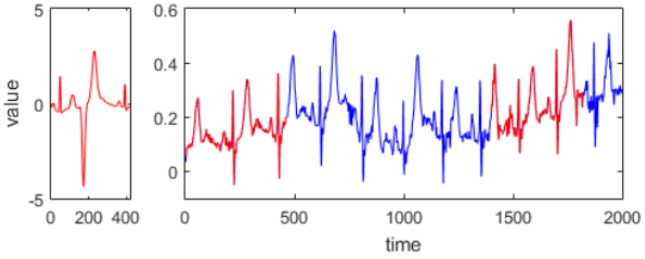


Figure 1: Search for Q (red) in an *ECG* time series (blue)

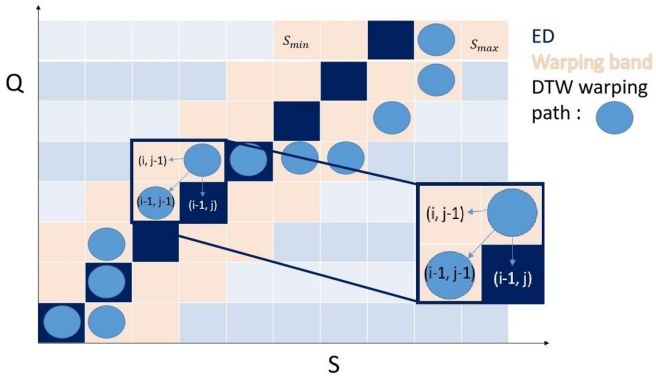


Figure 2: Distance matrix of *DTW*

But, single machines have limited scalability in terms of memory and execution power, which make them cumbersome for real-life, large scale streaming applications. Thus, these approaches either not require low latency responses [18] or make use of specialized hardware and/or hardware optimization to guarantee high performance [10, 11, 13, 20]. Example optimization that can be applied to *DTW* are independent matrix cell [20] or combined normalization and distance calculations [18]. Many proposed optimization made for single machines can not be efficiently implemented or lead to processing delays in distributed systems e.g. due to missing shared memory. Therefore, only a few selected techniques can be considered for our application.

A detailed overview of single machine approaches and implementations for similarity search in time series analysis is presented in [16]. This work offers a deeper inside of common search and matching methods combined with different optimization proposed until 2011.

Similarities are matched and weighted by the usage of distance measurements, which rank, how similar the compared sequences are. Besides the classic Euclidean distance, *DTW* is an often recommended measurement for similarity search [3, 15, 16, 18]. Already in 1994, [2] illustrated the usage of *DTW* for time series, which was previously used for speech recognition. Berndt and Clifford [2] explain the algorithm and suggests further optimization techniques. The most powerful pruning approach for *DTW* is implemented in [18], where Rakthanmanon et al. make intensive usage of a hierarchical lower bound pruning approach with increasing computational expenses to prevent unnecessary expensive full *DTW* calculations. This pruning approach is mainly considered for our implementation.

3 SIMILARITY SEARCH

Similarity search is a basic subtask for time series analytics such as classification and clustering. Let us consider the ECG time series example [17] in Figure 1. In this figure, the red pattern (Q) is matched twice with sequences of the blue time series. To search for patterns in long time series, it is usual to compare it against subsequences of the series with similar length. From now on we will refer to these subsequences as S. Similarity search can refer either to the comparison of sequences in their shape or values, where in time series processing, shape comparison is commonly focused [1]. Due to the fact that the pattern is not necessarily taken from the series, the scales of both might be different (see value range in Figure 1). The scale difference needs to be handled by normalization of each individual sequence, before attempting any meaningful shape comparison.

Distances play an important role in similarity search. A classic one-to-one point distance is the Euclidean distance. In comparison to the one-to-many point distance *DTW*, the Euclidean distance is inflexible in handling shifts in time or amplitude. The different behavior is visualized in Figure 2, where dark squares indicate the point-to-point comparison of the Euclidean distance and circles symbolize the *DTW* warping path. Further, a distance value is denoted as $D(Q, S)$ for the cumulative distance between the pattern Q and sequence S. For the cumulative distance over all ancestors until the two points q_i and s_j, where i and j indicate the index of each point in Q and S, the notation $\hat{d}(q_i, s_j)$ is used. The ancestor dependencies and the distance path through the matrix can also be seen in Figure 2. The final outputs are the remaining *minimal* $DTW(Q, S)$. Keeping in mind that the Euclidean distance (*ED*) is nothing else than a special case of *DTW* with $|Q| = |S|$, both distances can be related with the triangle inequality, which states that their final result has the following property, proven in [15]:

$$DTW(S, Q) \leq ED(S, Q)$$

The *DTW* distance matrix (Figure 2) is derived incrementally row-by-row because of the dependencies within the cells. Its calculation is the most expensive task of *DTW* [20], which is why heavy pruning and path restriction strategies are applied to reduce computation. In order to reduce the size of our *DTW* distance matrix, we use the *Sakoe-Chiba Band* [2, 18]. This warping band restricts the warping path so that only cells, which are a certain number of cells away from the diagonals, are considered. Similar to the experiments made in [3], the size of our band is set, by default, to 10% of the query length $|Q|$ and is further denoted as w_{warp}.

Discovery Methods: Similarity search considers two discovery methods. The first one is *Best Match Discovery*. This method searches for the most similar sequence compared to the given pattern, the so-called best match [20]. For monitoring tasks in stream processing, Best Match Discovery is insufficient as once a global minima is derived, all other sequences are pruned. The second discovery method is *Motif Discovery*. Motif Discovery is the mining of a repeated pattern in time series. According to [6, 15, 18] searching for repetitions of patterns in time series streams is more meaningful than Best Match Discovery, as streams are infinite and many similar patterns are expected to occur. In contrast to Best Match Discovery, Motif Discovery does not only deliver the best match, but also several similar sequences. Thus, it gives a deeper inside of the data [12], meaning that the stream is continuously searched and monitored.

Windows and Streams: When performing time series similarity search on infinite streams, it is common to consider sliding windows to analyze the data. Each window contains a sequence of the data. By sliding over the series, overlaps of the data are considered to prevent that a pattern is cut. Modern DSPS handle streams in the exact same way. Thus, the parameters for partitioning the data into sequences, can be related. First of all, from similarity search we know, that the pattern and the sequences should be of similar length. As *DTW* allows stretching and compression of the sequence [2], a minimal ($|S_{min}|$) and maximal sequence length ($|S_{max}|$) can be defined, considering that:

$$|S_{min}| \leq |Q| \leq |S_{max}|$$

Thus, in order to calculate the entire *DTW* distance matrix in one window, the sliding window size should be equal to $|S_{max}|$. In the window, we can create several subsequences (S_i) which fulfill the following equation:

$$|S_{max} - S_i| \geq |S_{min}|$$

Therefore, $|S_{min}|$ is the perfect overlap size, which still ensures an high accurate result. To slide over the time series, it needs to be divided in *consecutive* sequences. DSPS allow the user to collect data of the a certain time range in one window. Nevertheless, the window content is not guarantied to be in order. Thus, in cases where order is required, sorting by time needs to be applied to ensure a meaningful comparison.

4 DISTRIBUTED DYNAMIC TIME WARPING

Usually, pattern mining is a data discovery task, made at a stage, where users have little knowledge about the data. Therefore, our implementation allows the user to start the search without setting DSPS or similarity search parameters. The initial pruning parameters are derived when the processing starts and are continuously updated whenever a new sequence, more similar to the pattern, is detected. If further knowledge about the data exist, parameters can also be set to narrow the search space. One feature of DSPS is *keying*, which automatically groups the data by assigned labels, e.g. each monitored patient gets his own key. Another powerful feature for similarity search is the global state, which can be accessed to get and update non-static parameters such as maximal distance. Indeed, this global state allows faster pruning on all workers by communicating the *best-so-far* distance result and thus, avoiding unnecessary calculations of unpromising subsequences. A further advantage of this state is, that it is *key-based* and thus able to develop different thresholds for multiple time series [4], e.g. a specific threshold for each monitored patient.

4.1 Normalization

As mentioned before, patterns and sequences might appear in different scales (see Figure 1). Therefore, normalization is required. In streaming we can apply normalization per sliding window. The most accurate solution would be if the slide size is equal to one and the window size equal to the pattern size. Such an accurate normalization is optimized in single machines by leveraging previous calculations. As workers in distributed systems do not have easy access to previous results, our strategy is to extend the window to S_{max} and trigger a new one after an optimal slide size. This strategy reduces not only the number of windows but also allows the implementation of several pruning techniques (see Figure 3). Besides the overall computation reduction this might affect accuracy. The reason for the accuracy loss is that, due to the window extension, additional data points are considered. If those data points are outliers, they influence the transformation of all window observations.

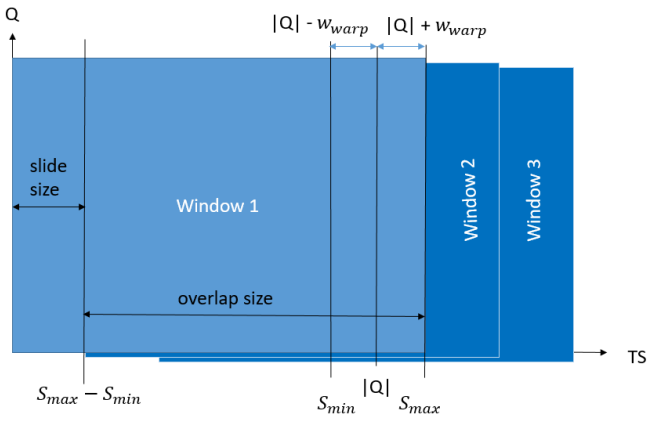


Figure 3: Overview of DTW Parameters

4.2 Window Parameters

So far, we have derived a relationship between window and similarity search parameters, the query length $|Q|$ and the warping band restriction w_{warp}, but no optimal or recommended size for S_{max} and S_{min} have been defined.

We extend the window size from $|Q|$ to $|S_{max}|$ to allow the stretching of our sequence. Still, there are two main reason why the window size should be kept small: Its negative impact on the normalization accuracy (Section 4.1) and that in fact *DTW* is a cumulative distance. This leads to the assumption, that too long sequences can hardly give better results than shorter ones. *DTW* is able to overcome large differences by leaving the diagonal path by maximal w_{warp} cells. Still, longer sequences with lower accumulated distances will not be a common case [16]. Restricting $|S_{max}|$ is a performance optimization as for each additional point a further row is added to the *DTW* matrix. For all those reasons, S_{max} is set by default to the following length and due to the warping band relation we can define S_{min} in a similar way :

$$S_{max} = |Q| + w_{warp} \quad \text{and} \quad S_{min} = |Q| - w_{warp}$$

On the basis of $|Q|$ and w_{warp}, all parameters of our implementation can be derived automatically for *DTW* and its processing in a distributed system. They can also be set manually to adjust the search according to special requirements. The derived window parameters are in accordance with the suggestions made in [16]:

$$|S_{max}| \leq 2 * |Q| \quad \text{and} \quad S_{min} \geq \frac{|Q|}{2}$$

One of the advantages of distributed systems is, that the sliding windows can be distributed among several workers and therefore takes advantage of parallelization. Once a worker has received its window to be processed, we can apply traditional optimization techniques as it is done in single machine processing.

4.3 Pruning

Fast pruning techniques for *DTW* have been considered to reduce execution time. As no comparable distributed stream processing implementation exists, we studied and modified single machine techniques [18] for the usage in distributed systems. Not all of them can be meaningfully applied to our implementation, for example running sums or previous result heuristics with

dependencies between two consecutive subsequences [18] can not be guaranteed in distributed systems. These techniques are simply too expensive to be applied on independent subtasks and workers with only few observations instead of the whole time series. Since pruning often comes at the cost of accuracy, only the following techniques, with good approximate results [6, 18] and applicable to single sequences have been implemented:

(1) **Warping band size and global maximal distance:** We use the *Sakoe-Chiba Band* [18] for reduction of the *DTW* distance matrix search space and further prune all paths as soon as their latest cell reaches the maximal distance. The entire distance matrix is pruned, whenever for a certain q_i, no s_j can be found, which means for all possible paths $\hat{d}(q_i, s_j)$ is above the maximal distance. Due to the usage of a global key-based state for the maximal distance, each time series has its own distance threshold.
(2) **Lower Bounds (LB):** Early pruning of *DTW* matrix calculations is possible with low cost calculations of LB. LB_Kim, LB_KimFL and LB_Keogh variations [18], which consider specific indexes and conditions for (q_i, s_j) combinations, are implemented in our approach. Further, we can prune an entire window using the triangle inequality of *DTW* and Euclidean distance by calculating the Euclidean distance for the first and the last subsequence of the window. If both are above the *best-so-far* Euclidean distance (kept in our global state), the window is pruned. It is worth to note, that this is not an exact search.
(3) **Square root operations** on distance calculations are omitted as the extra computational effort makes no difference to the distance ranking [18].

5 EXPERIMENTS

We have designed three experiments using the ECG_{oneday} [17] dataset and the query pattern (Q) of the experiments from Rakthanmanon et al. [18]. The first experiment is a baseline test that allows us to compare our results against the ones reported in [18]. In the second experiment we simulate a real-life scenario e.g. a hospital where patients stay for a couple of days and their ECG signals are continuously monitored. Finally, in the third experiment the average response time is measured under various loading conditions in order to determine how long it takes the system to match the pattern in a continuous time series stream in average.

5.1 Data and Processing Machines

The ECG_{oneday} dataset contains 20.140.000 data points and represents one day, in particular 22 hours and 23 minutes, of ECG data for a single patient. Thus, we receive 250 data points per second. Using the default settings of our implementation and the length of the given pattern, one Flink window contains 463 data points or 1.9 s of data. As sliding windows are used in our application, another window is triggered by default every 336 ms.

In order to facilitate result comparisons and analyze the processing overhead of parallelization in a distributed system we run our experiments on the following machines:

- *Laptop*, an 8-core Intel with 2.90 Ghz, 31,1 GB RAM, running Ubuntu 18.04
- Another single machine, further called *Server*, with 48 CPUs, 2.90 Ghz, 126 GB of RAM, running Ubuntu 16.04
- Finally, we use a *Cluster* with 6 nodes, each with 48 CPUs, 2.90 Ghz and 48 GB of RAM, running Linux 4.18

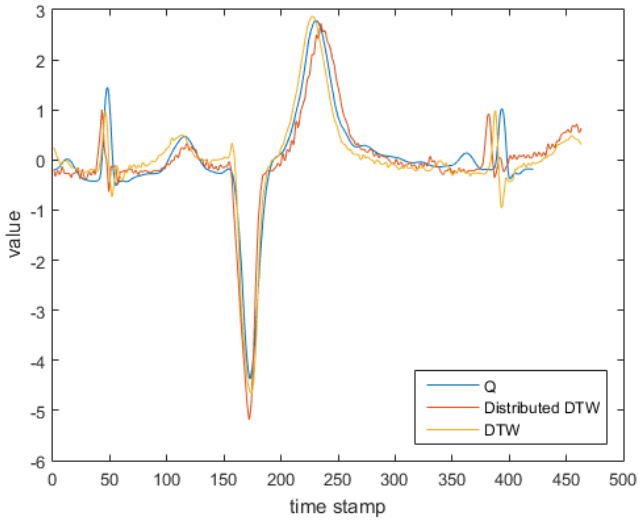


Figure 4: Baseline Test: Accuracy Comparison

Table 1: Baseline Test: Execution Time Comparison for various Machines with parallelization 8

Data	Baseline [18]	Laptop	Server	Cluster
1 patient, 1 day Best Match Discovery	35 s	37 s	145 s	118 s
1 patient, 10 days Best Match Discovery	-	330 s	1.312 s	977 s
1 patient, 10 days Motif Discovery	-	967 s	4.066 s	2.176 s

In our implementation in Apache Flink (version 1.4.2), we handle several patients' data in parallel by utilizing multiple input sources. Additionally, each operator in Flink, like our window function, can split its operations into several subtasks, which can be executed distributed and in parallel. The number of subtasks is defined by the degree of *parallelism*. These subtasks are chained together into so-called tasks. All operators working on the same degree of parallelism can be chained to one task.

In order to execute our experiments we have to define another important parameter on the machines: *Taskslots*. This parameter defines the number of tasks a single worker can accept, which is recommended to be set equal to the number of available CPUs per machine [4]. Thus, *Laptop* is set to 8, while the other two machines are set to 48. Important to notice is that the number of *Taskslots* influences the parallelization degree. If more tasks than available task slots are created, these tasks need to wait, which cause delays or timeout errors. For all tests, five repetitions of each experiment are performed and the average processing times are reported.

5.2 Baseline Test

Our initial test is a comparison between the implementation of Rakthanmanon et al. [18] and ours. Thus, we run our application on the three machines for the ECG_{oneday} dataset. For this test, the search mode of our application was set to Best Match Discovery, which is also used in the experiments of Rakthanmanon et al. We expected a computational overhead for all machines

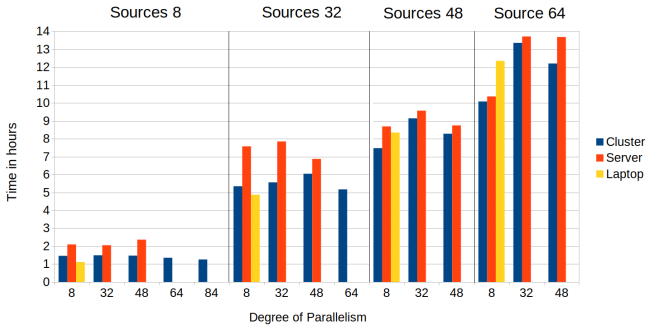


Figure 5: Stream Simulation Test: Total Execution Time for ten Days, several Patients (Sources) and various levels of parallelism executed on *Laptop*, *Server* and the *Cluster*

due to the usage of Apache Flink. Both, *Cluster* and *Server* need to deal with CPU distribution and the *Cluster* further with data shuffles and communication effort to overcome the shared memory disadvantage. These costs only become negligible in case of large data sets or streams, which in fact ECG_{oneday} is not.

We set all machines to a maximal parallelization degree of 8 for a better comparison with the *Laptop* achievements. The results presented in Table 1 prove that our implementation is very close to Rakthanmanon et al. [18] achievements, with only 2 s difference on *Laptop*. Here, the application profits from caching due to shared memory and little monitoring costs of the cores.

On the other hand, shared memory and/or monitoring tasks cause a massive overhead in both, the *Cluster* and the *Server*, as the processing time for these machines are much higher. As mentioned before, these results are not surprising as stream processing as well as distributed systems are intended to process large amounts of data. Thus, we re-run this experiment with an increased number of days to 10. *Laptop* still can handle this amount of data and beats the other two machines in both modes, Best Match and Motif Discovery. Here, we can already notice that the difference between both modes is an execution time increase of up to three times in all machines.

This experiment was also used to double check accuracy, in the sense of verifying whether we obtain the same best match results as Rakthanmanon et al. [18]. In our implementation, due to the distributed normalization used, we obtained two best matches, one of them corresponding to the one found by Rakthanmanon et al. We show the best matches obtained with our distributed *DTW* in comparison to the one of *DTW* [18] in Figure 4.

5.3 Stream Simulation Test

In this experiment we stress the processing machines even more by not only adding more days of data but increasing the number of patients up to 64. The data is processed in parallel and the search mode is set to Motif Discovery, which is a more realistic scenario than Best Match Discovery. This search mode has much higher computational demand, as seen in the first experiment, since more calculations need to be executed and the output rate increases due to a positive tolerance interval of the maximal distance bound. Stream processing is intended to run over a long period of time, thus we fix the number of days to 10 per patient, based on the idea of simulating a real-life scenario e.g. of a hospital, where patients stay for a couple of days. We run this test with an increasing number of patients (sources in the Flink environment) starting with 8 and up to 64 patients. Also, several parallelization levels, from 8 to 84 are applied. The execution time comparison of various settings is visualized in Figure 5.

A first observation in Figure 5 is that the *Laptop* can only handle parallelization 8 and although, it is faster with 8 sources as soon as we increase the parallelization level, *Laptop* is not even able to start the process. Remember that every machine has a limitation on the number of *Taskslots*, for *Laptop* it is 8.

We can also observe that the best results are obtained with the *Cluster* for parallelization 8, irrespective of the number of sources. This result can be explained again by the available number of task slots in the *Cluster*. Moreover, the first experiment with 8 sources on parallelism 8 creates only 64 subtasks excluding the 8 source tasks in the beginning. The *Cluster* can execute all of them in parallel due to its workers capacity (48 Taskslots $*$ 6 nodes), while *Server* as well as *Laptop* need to queue tasks, what increases their response time.

But, how far can we increase the parallelism on the *Cluster*? We observed, that whenever the following inequality is true, we do not profit any more from the distributed environment.

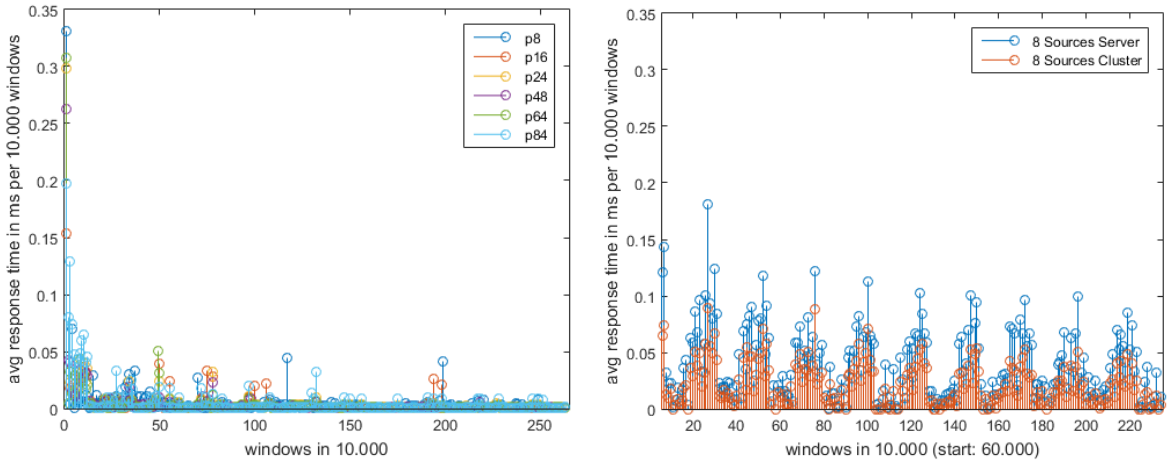
$$\text{Sources} * \text{Degree of Parallelism} \geq 2 * \text{Available task slots}$$

5.4 Average Response Time Test

The final experiment has been conducted to examine, how long the system takes to decide, if a window is similar to the given pattern, or not. For real-life applications the response time is a critical issue, since the faster the matching is done the faster a human can react, e.g. attend a patient in a hospital. In a real-time scenario a worker processing a window needs to wait until its end time is reached, before it is triggered. In other words, as long as it has not seen all the required data, no decision can be made. Thus, this time is not considered as technical caused delay. Inspired by the benchmark experiments performed in [9], we define the average response time by calculating the difference of the system time between the arrival of the final data point of the window and the output of the window result. Figure 6(a) presents the results of our application, aggregated over 10.000 consecutive windows for 10 days data of a single patient. The plot shows, that initializing the application is an expensive process. Here, several pruning parameters are established and permanently updated with improved threshold values between the workers. After the first 10.000 windows, the average response time drops from up to 0,34 ms to less than 0,05 ms in average per window as the application is now able to quickly prune unpromising sequences.

This response time is stable for the next 100.000 windows, which corresponds in real-time to the first 16 hours of the first day. Afterwards, it drops again to responses truly close to 0 and thus to real-time response. Further, we can observe some non-regular peaks in Figure 6(a). They indicate delays, which are caused by the network, e.g. due to the data throughput, communication, result writing, etc. Still, the time variants of the peaks are tiny and appear rarely, thus, they can be ignored for the overall result.

The average response time increases if multi-sources are processed. Figure 6(b) is the resulting plot of the average response time test runs with 8 patients and 10 days of data on *Server* and *Cluster*. In the first 100.000 windows the response time is up to 120 ms with an average of 65 ms for the *Server* and up to 70 ms for the *Cluster* with an average of 36 ms. Only after learning the pruning parameters both machines prove fast responses of 0,022 ms for the *Cluster* and 0,039 ms for the *Server*.



(a) Average Response Time for a single source

(b) Average Response Time, 8 sources, after first 100.000 windows

Figure 6: Average Response Time for *DTW* on *Cluster*: Aggregated results over 10.000 processed windows for 10 days

6 CONCLUSIONS

In this paper we present time series similarity search with *DTW*, which has been reported as computational expensive and difficult to implement for real-time processing. We benefit from Apache Flink features such as smart partitioning and keying and are thus able to process several sources and sliding windows in parallel for independent distance computation. The overall result of this paper is, that efficient similarity search by the usage of modern DSPS is feasible and gives sufficiently fast response for immediate reactions on matching sequences. Besides, and to the best of our knowledge, none of the reviewed literature can handle several maximal distance thresholds for searching patterns in multiple time series in parallel. Future work is required though, especially, to extend the efficiency and generality of the application. Possible future work areas are: handle multiple patterns at the same time, allow processing of multi-dimensional time series and the maintenance of patterns over time, for example by using a *ValueState* to allow on-line learning. For *DTW*, the usage of a tree structure to maintain the pattern would be necessary to react on matches with different length.

ACKNOWLEDGMENTS

This work was partly supported by the German Federal Ministry of Transport and Digital Infrastructure (BMVI) through the Daystream project (grant no. 19F2031A), the German Ministry for Education and Research (BMBF) as BBDC (grant no. 01IS14013A) and the EU H2020-ICT-2017-2018 program through the BigMedilytics project (grant no. 780495).

REFERENCES

[1] Alice Berard and Georges Hebrail. 2013. Searching Time Series with Hadoop in an Electric Power Company. In *Proceedings of the 2Nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine '13)*. ACM, New York, NY, USA, 15–22. https://doi.org/10.1145/2501221.2501224

[2] Donald J Berndt and James Clifford. 1994. Using dynamic time warping to find patterns in time series.. In *KDD workshop*, Vol. 10. Seattle, WA, 359–370.

[3] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. 2008. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1542–1552.

[4] Apache Flink. 2018. Apache Flink: Fast and reliable large-scale data processing engine. http://flink.apache.org

[5] Apache Foundation. 2018. Apache Storm. http://storm.apache.org/

[6] Alborz Geramifard, Finale Doshi, Joshua Redding, Nicholas Roy, and Jonathan How. 2011. Online discovery of feature dependencies. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 881–888.

[7] Tian Guo, Saket Sathe, and Karl Aberer. 2015. Fast distributed correlation discovery over streaming time-series data. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 1161–1170.

[8] Mirko Kämpf and Jan W. Kantelhardt. 2013. Hadoop.TS: Large-Scale Time-Series Processing. *International Journal of Computer Applications* 74, 17 (July 2013), 1–8. https://doi.org/10.5120/12974-0233

[9] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Processing Engines. *arXiv preprint arXiv:1802.08496* (2018).

[10] Maria Kontaki, Apostolos N Papadopoulos, and Yannis Manolopoulos. 2007. Adaptive similarity search in streaming time series with sliding windows. *Data & Knowledge Engineering* 63, 2 (2007), 478–502.

[11] Xiang Lian, Lei Chen, Jeffrey Xu Yu, Guoren Wang, and Ge Yu. 2007. Similarity match over high speed time-series streams. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 1086–1095.

[12] Bo Liu, Jianqiang Li, Cheng Chen, Wei Tan, Qiang Chen, and MengChu Zhou. 2015. Efficient motif discovery for large-scale time series in healthcare. *IEEE Transactions on Industrial Informatics* 11, 3 (2015), 583–590.

[13] Alice Marascu, Suleiman A Khan, and Themis Palpanas. 2012. Scalable similarity matching in streaming time series. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 218–230.

[14] Abdullah Mueen and Eamonn Keogh. 2010. Online Discovery and Maintenance of Time Series Motifs. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*. ACM, New York, NY, USA, 1089–1098. https://doi.org/10.1145/1835804.1835941

[15] Rodica Neamtu, Ramoza Ahsan, Elke Rundensteiner, and Gabor Sarkozy. 2016. Interactive Time Series Exploration Powered by the Marriage of Similarity Distances. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 169–180. https://doi.org/10.14778/3021924.3021933

[16] Panagiotis Papapetrou, Vassilis Athitsos, Michalis Potamias, George Kollios, and Dimitrios Gunopulos. 2011. Embedding-based subsequence matching in time-series databases. *ACM Transactions on Database Systems (TODS)* 36, 3 (2011), 17.

[17] Mueen Rakthanmanon, Campana and Batista. 2012. The UCR Suite: Funded by NSF IIS - 1161997 II. http://www.cs.ucr.edu/~eamonn/UCRsuite.html

[18] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2013. Addressing Big Data Time Series: Mining Trillions of Time Series Subsequences Under Dynamic Time Warping. *ACM Trans. Knowl. Discov. Data* 7, 3 (Sept. 2013), 10:1–10:31. https://doi.org/10.1145/2500489

[19] Norihiro Takahashi, Tomoki Yoshihisa, Yasushi Sakurai, and Masanori Kanazawa. 2009. A parallelized data stream processing system using dynamic time warping distance. In *Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on*. IEEE, 1100–1105.

[20] Limin Xiao, Yao Zheng, Wenqi Tang, Guangchao Yao, and Li Ruan. 2013. Parallelizing dynamic time warping algorithm using prefix computations on GPU. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 294–299.