Time Series Discord Detection in Medical Data using a Parallel Relational Database

Diane Myung-kyung Woodbridge* Andrew T. Wilson* Mark D. Rintoul* Richard H. Goldstein†

*Sandia National Laboratories, Albuquerque, New Mexico
†RGI Informatics, Boston, MA
{dwoodbr, atwilso, mdrinto}@sandia.gov
rgoldstein@rgi-informatics.com

***Abstract*—Recent advances in sensor technology have made continuous real-time health monitoring available in both hospital and non-hospital settings. Since data collected from high frequency medical sensors includes a huge amount of data, storing and processing continuous medical data is an emerging big data area. Especially detecting anomaly in real time is important for patients' emergency detection and prevention. A time series discord indicates a subsequence that has the maximum difference to the rest of the time series subsequences, meaning that it has abnormal or unusual data trends. In this study, we implemented two versions of time series discord detection algorithms on a high performance parallel database management system (DBMS) and applied them to 240 Hz waveform data collected from 9,723 patients. The initial brute force version of the discord detection algorithm takes each possible subsequence and calculates a distance to the nearest non-self match to find the biggest discords in time series. For the heuristic version of the algorithm, a combination of an array and a trie structure was applied to order time series data for enhancing time efficiency. The study results showed efficient data loading, decoding and discord searches in a large amount of data, benefiting from the time series discord detection algorithm and the architectural characteristics of the parallel DBMS including data compression, data pipe-lining, and task scheduling.**

I. Introduction

Recent advances in sensor technology have made continuous real-time patient monitoring systems available. Health care providers can monitor vital signs, activities and moods of patients who are hospitalized or even discharged using light weight wearable sensors. Many sensors have capability of monitoring signals continuously in order to detect any health-related issues. These high frequency medical sensors include pulse oximeter, sphygmomanometer, accelerometer, pressure sensors, gyroscope, etc.

Loading high frequency data requires data summarization before delivering data to a database system (DBMS), when a single node-based DBMS is not tolerant in the multi-user setting [1]. However, coarsely summarized data can miss important patterns of data and prohibit possible analytics on the original data. This can potentially affect system reliability and efficiency. Especially in health monitoring systems, it is sometimes necessary to load and analyze high frequency raw data with very low false negative rates.

A parallel DBMS enhances speed of data processing and I/O by using multiple processors and disks in parallel. The previous work [2] showed that IBM PureData for Analytics, a parallel DBMS powerd by Netezza [3] performed well in both single and multi user settings for processing large amount of medical data. As the previous study focused on scalability of the system for executing queries rather than data analytics, developing and applying an algorithm to analyze time series anomaly was considered as a next step.

A time series discord indicates a subsequence that has the maximum distance to its neighbor in the given time series data, which means abnormal or unusual data trends. Finding time-series discords is a promising new technology in health monitoring for early detection and prevention of life threatening situations. In this study, we implemented two versions of symbolic aggregate approximation (SAX) algorithm [4] and validated the performance of the searches. Keogh claims that the brute-force SAX algorithm compares all possible pairs yielding quadratic time complexity which is not suitable for most large real-world datasets, while the heuristic algorithm applies heuristics to reorder the search sequences and skip unnecessary comparisons [4].

Since the previous study showed a good scalability and time efficiency of IBM PureData for Analytics, a single-rack hardware-plus software system, we utilized the same DBMS to develop the SAX algorithms in this study. As SAX-based algorithms are known to work well on time-series discords detection in medical data [4][5], this study focuses on the performance of the algorithm on the high performance parallel DBMS.

The rest of the paper is organized as follows. In Section II, we compare distributed DBMS especially focusing on Hadoop MapReduce with parallel DBMS based on existing literature. The system specification of IBM PureData for Analytics is described in Section III. Two versions of SAX algorithms and their experiment results are described in Section IV and Section V accordingly. In Section VI, we conclude the study and provide suggestions for future studies.

II. Background

Utilizing distributed data storage and processing of big data on computer clusters became popular. One of the most

widely used distributed data storage and processing models is Hadoop MapReduce. In this model, Apache Hadoop [6] acts as a distributed storage, divides big data into smaller pieces and distributes to the nodes in the clustered file system, while MapReduce [7] performs data processing algorithms and scheduling algorithms on the cluster.

While many studies investigated and proved the efficiency of distributed systems for medical data processing including medical images and vital signs [8][9], that of parallel databases has not been researched enough. This can be related to the cost issues of parallel databases. While MapReduce systems are mostly open source and free, parallel DBMSs are expensive. Moreover, installing a parallel DBMS takes longer time and sometimes requires expert support from its vendor. Therefore, quick and dirty research experiments might not be suitable for analyzing medical data using a parallel DBMS unless one already has a system which is not feasible in most academic research settings.

Stonebaker compared Hadoop MapReduce with parallel DBMSs including DBMS-X and Vertica [10]. The study showed that parallel DBMSs work better at efficient querying of large amount of data and provide higher level programming languages to users. Additionally, parallel DBMSs outperform in sequential searches, GROUP BY and JOIN queries. DBMS-X and Vertica that the author used performed 36 and 21 times respectively faster than 100-node clusters of Hadoop MapReduce. The author explained that this is because of the common architectural characteristics of parallel DBMSs including data compression, data pipe-lining, and task scheduling.

In this study, we utilized IBM PureData for Analytics, a high performance parallel database system that Sandia National Laboratories owns to detect abnormal data trends in real time.

III. SYSTEM

IBM PureData for Analytics performs high speed data analytics using its parallel processing units called S-Blades which include arrays of multi-core CPUs, multi-engine Field Programmable Gate Arrays (FPGAs) and Random-access memory (RAM) and its software. The commonly accessed data is located in RAM instead of on a disk, and FPGA keeps data only required for high performance processing of analytics. The remaining data is processed on a CPU. The system software optimizes query plan to minimize disk I/O and data movement. Additionally, the software determines schedules to balance workloads and executes them simultaneously (Figure 1). The system that we utilized for this study has 3TB of uncompressed disk capacity and built-in compression that typically results in 80-120TB of effective storage for user data. The system is separated into a storage pool containing 96 disks and a processing pool comprising 12 processing blades (Table I).

In IBM PureData, an external table is an external file that the system treats it as a database table but the actual data exists outside of the database. The ***nzload*** command creates an external table definition, loads data from a file in the local or a remote client and performs all insertion jobs within a single transaction. While ***nzload*** is running, records are sent to the CPUs with the current transaction ID. When an CPU receives the records, it allocates resources and writes the data to the databases or the table on the disk. When the data loading job is completed, ***nzload*** drops the external table. This works a lot faster than using a series of INSERT queries for loading data from an external file.

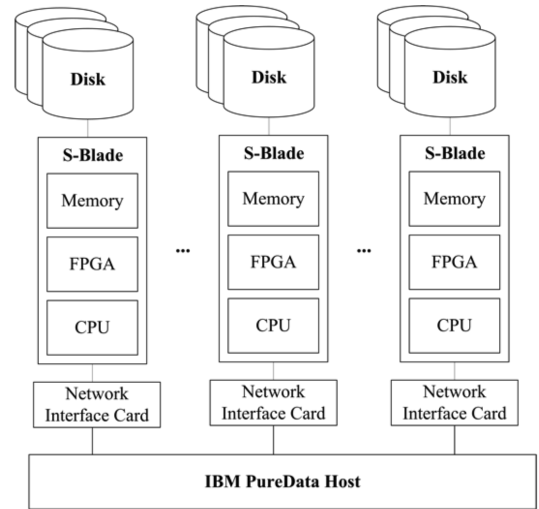

Fig. 1: IBM PureData System Architecture [3]

Component	Specifications
Front End (x2)	2x 4-core Intel Xeon, 2.4 GHz 24 GB memory
Blade (x12)	2x 4-core Intel Xeon, 2.4 GHz 24 GB memory, 4 FPGAs with 2 stream engines apiece
Disk (x96)	1 TB SAS, 7200 RPM Redundancy provided by database
Internal Network	Gigabit Ethernet

TABLE I: IBM PureData Hardware Specifications

IBM PureData for Analytics provides features that users can define own functions in addition to most of the common Structured Query Language (SQL) functions. A user-defined table function (UDTF) can be declared in C++ and is a way for a user to define a new data analytics action which are not available with the built-in SQL functions. A UDTF can return zero or more rows of data which are qualified.

A stored procedure provides an ability to encapsulate jobs of data processing, data transaction and application branching behavior such as IF-ELSE-THEN. IBM PureData supports stored procedures using Postgres' PL/pgSQL. As an application developed using a stored procedure resides on the database host, the application works fast by avoiding the network time and is easy to maintain.

In this study, we utilized ***nzload*** for loading data from external files, and UDTFs and stored procedures for data decoding and time series discord detection. While a stored procedure must be invoked using the CALL, EXECUTE or SELECT commands, an UDTF should be invoked in a FROM clause in a SELECT statement. The implemented UDTFs and stored procedures benefited from data pipe-lining and task

β_i \ α	3	4	5
β_1	-0.43	-0.67	-0.84
β_2	0.43	0	-0.25
β_3		0.67	0.25
β_4			0.84

TABLE II: breakpoints lookup table. Breakpoints $B = \beta_1, ..., \beta_{\alpha-1}$ that β_i to β_{i+1} is $\frac{1}{\alpha}$ under $N(0,1)$.

	a	b	c	d
a	0	0	0.67	1.34
b	0	0	0	0.67
c	0.67	0	0	0
d	1.34	0.67	0	0

TABLE III: $distance$ lookup table for $a = 4$.

scheduling provided by the DBMS for an efficient search.

IV. ALGORITHM

In this study, we applied *i*) the brute-force version of the SAX algorithm and *ii*) heuristically ordered time series using SAX (HOT SAX) to detect time series discords [11] [12]. Both algorithms apply piece-wise aggregate approximation (PAA) to time series data of length n and reduce the length to f ($f \leq n$). When original time series is $C = c_1, ...c_n$, PAA data ($\overline{C}$) is calculated by Equation (1).

$$\overline{C}_i = \frac{f}{n} \sum_{j=\frac{n}{f}(i-1)+1}^{\frac{n}{f}i} c_j \tag{1}$$

Converting original data into PAA reduces the data from n dimensions to f dimensions by dividing data into data frame size of f. The best value for the f parameter is not universal, and depends on the data.

SAX represents PAA data to symbols with alphabet size a and of length f. Assuming that the given time series have high Gaussian distribution, PAA data can be converted to standard normal distribution z which follows $N(0,1)$.

$$z = \frac{x - \mu}{\sigma} \tag{2}$$

Then, the algorithm discretizes values using breakpoints in standard normal cumulative distribution table (Table II) which returns a equal-sized areas. The well-known best value for the a parameter is either 3 or 4 [11][13].

Given two time series data represented in PAA, $\overline{Q}$ and $\overline{C}$, their symbolic representations are given as $\hat{Q}$ and $\hat{C}$. The minimum distance between two symbolic representations can be calculated using Equation (3). The $distance$ function in Table III calculates distance between two alphabet symbols.

$$MINDIST(\hat{Q}, \hat{C}) = \sqrt{\frac{n}{f}} \sqrt{\sum_{i=1}^{f} (distance(\hat{q}_i, \hat{c}_i))^2} \tag{3}$$

In this study, we applied a brute force way of detecting discords with $O(n^2)$ complexity and a heuristic way with $O(n)$ complexity. Both functions are developed in IBM PureData for Analytics using UDTFs.

A. Brute Force Discord Detection

The brute force algorithm looks for time series discords in a naive way. It takes each possible subsequence and calculates a distance to the nearest non-self match which is a subsequence that is not overlapped with the candidate subsequence (Algorithm 1). The brute force algorithm applies nested loops. The outerloop of the algorithm considers each possible candidate subsequence and the innerloop linearly scans to find the candidate's nearest non-self match. Since Algorithm 1 goes through all possible subsequence pairs to find the nearest non-self match, it takes $O(n^2)$ complexity. This is not suitable for most large datasets, the algorithm requires improvements in time efficiency.

```
Data: Symbolic representation of time series data (T),
      sliding window size (w)
Result: worst_discord_distance,
        worst_discord_location
worst_discord_distance = 0;
worst_discord_location = NaN;
for i = 1... | T | -w + 1 do
    nearest_neighbor_distance = infinity;
    for j = 1... | T | -w + 1 do
        if | i - j |≥ w then
            if nearest_neighbor_distance >
            Distance(t_i...t_{i+w-1}, t_j...t_{j+w-1}) then
                nearest_neighbor_distance =
                Distance(t_i...t_{i+w-1}, t_j...t_{j+w-1});
            end
        end
    end
    if nearest_neighbor_distance >
    worst_discord_distance then
        worst_discord_distance =
        nearest_neighbor_distance;
        worst_discord_location = i ;
    end
end
```

Algorithm 1: Brute Force Discord Detection

B. Heuristic Discord Detection

In order to improve the speed of Algorithm 1, we need to order the outer and inner loop effectively. For the inner loop, if we find any subsequence that is same or similar to the current candidate, this would make terminating the loop early as the current candidate could not be a discord. For the outer loop, ordering rare subsequenes first would assign a large value to $worst_discord_distance$ and help inner loop to be terminated early. This heuristic is applied only once for the outer loop. However, the heuristic for the inner loop should be

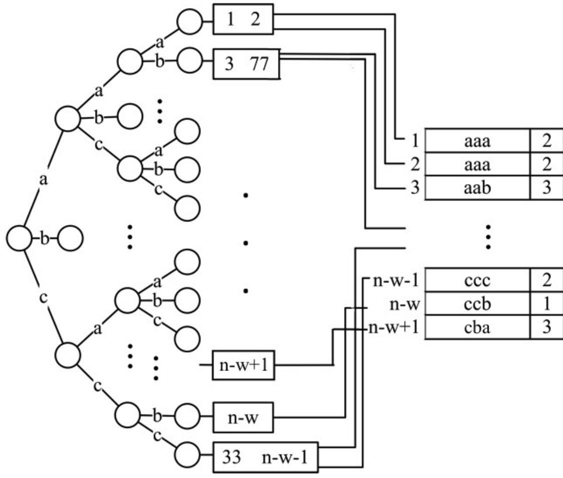

Fig. 2: A combination of an array and a trie structure for improving outer and inner loop sequences.

invoked for every iteration of the outer loop for considering the current candidate subsequence.

In order to design aforementioned outer and inner loop, a combination of an array and a trie structure is applied (Figure 2). The array structure has two fields to store i) a subsequence of the symbolized data and ii) its frequency in the given time series data. After discretizing data, representing signals as symbols and extracting subsequence by applying a sliding window, we stored each subsequence with a length of w in the array. This step is done sequentially in order to represent an index of the array represents the location of the subsequence.

A trie is an ordered tree data which is commonly used for storing and searching strings and a root is an empty string and all descendants of a node share a common prefix. In the trie structure, we stored each strings and extended the structure so that a leaf node includes a list of locations of the word. In this manner, an index of the array and a leaf node in the trie structure can have a connection. By counting the size of a list in the trie structure, a frequency of each word in the array can be updated. For instance, in Figure 2, we can find that the word, **aab** occurred at index 3 and 77 by looking up the trie, and can update the frequency of 3 into the array structure. Also, if we are interested in a particular location such as n-w-1, we can go to the array and find that the word is **ccc** and it also occurred at index 33 by tracking down the trie structure.

Both array and trie structures can be created in time and space linear in length of a time series data [14][15]. The outer loop is sorted to have the least common words by searching in the array structure, while the inner loop is sorted to have a same or similar word that the outer loop is indicating by looking up the trie. Based on the developed structure, we can find words with the smallest frequency by scanning the rightmost column of the array and assign them to the outer loop first. Then we order the items in the inner loop in the order of the elements in the liked list at the terminal nodes of the trie (Algorithm 2). In the inner loop, using the array and trie structure, the rest of the subsequences are visited randomly after visiting candidate subsequence chosen from the outer loop.

Data: Symbolic representation of time series data (T), sliding window size (w), $outer_loop_orders$, $inner_loop_orders$
Result: $worst_discord_distance$, $worst_discord_location$
$worst_discord_distance = 0$;
$worst_discord_location = NaN$;
for i *in* T *ordered by* $outer_loop_orders$ **do**
  $nearest_neighbor_distance = infinity$;
  for j *in* T *ordered by* $inner_loop_orders$ **do**
    if $\mid i - j \mid \geq w$ **then**
      if $Distance(t_i...t_{i+w-1}, t_j...t_{j+w-1}) < worst_discord_distance$ **then**
        $Break$;
      end
      if $nearest_neighbor_distance > Distance(t_i...t_{i+w-1}, t_j...t_{j+w-1})$ **then**
        $nearest_neighbor_distance = Distance(t_i...t_{i+w-1}, t_j...t_{j+w-1})$;
      end
    end
  end
  if $nearest_neighbor_distance > worst_discord_distance$ **then**
    $worst_discord_distance = nearest_neighbor_distance$;
    $worst_discord_location = i$;
  end
end

Algorithm 2: Heuristic Discord Detection

V. Experiment Results

A. Data

We received data of 9,723 patients collected from hospital settings. Patient information were de-identified based on health insurance portability and accountability act (HIPPA) regulations. The total size of data is 413 GB including patient information, vital signs, waveform data and data record information (Figure 3). Vital signs were monitored every 2 seconds and the total number of monitored vital sign types is 83. Each vital sign file includes average of 12.71 types. Examples of recorded vital signs are heart rate, electromyography (EMG), positive end expiratory pressure and central venous pressure. Waveform data were collected every 1/240 seconds. The total number of monitored channels is 35 and each file includes average of 5.32 channels. Monitored waveform data types include oxygen saturation (SpO_2), carbon dioxide concentration (CO_2), intracranial pressure, central venous pressure and others. Both vital signs and waveform data files include missing data, but we did not apply any missing data imputation techniques for this study [16].

B. Data Transformation and Data Loading

Since storing waveform data in double is not efficient, data was compressed in 16-bit binary code which represents the length of the data and binary coded waveform data values (Figure 4).

We encoded total of 441,928 16-bit binary codes which includes 212,125,440 numeric values from 8971 XML files

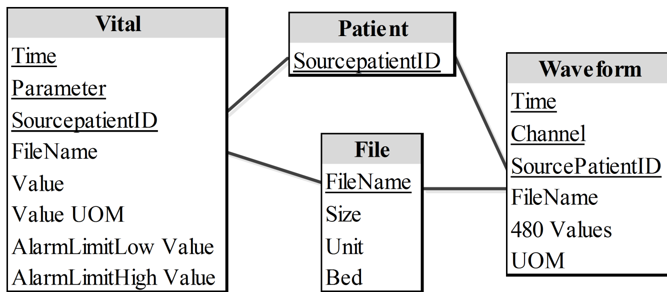

Fig. 3: Input data description.

Data Size	16-bit Binary Coded Data

Fig. 4: Converted Wave Data Format.

using a Python script. Data was inserted into tables using ***nzload*** command, a high performance data loading command provided by the system described in Section III.

In this study, we developed user defined table functions to decode encoded values into the original format.

C. Experiment Results

We loaded data into tables listed in Figure 3 using ***nzload***. Loading 78 KB vital sign data and encoded 16 digit binary values from 8971 files only took 0.1 and 1.08 seconds respectively (Table IV). This experiment result shows that loading big data files using $nzload$ is efficient.

Execution time for decoding 16 digit binary codes within a given time period is described in Table V. This job was done using a UDTF which provided extensive abilities to define a function required for data analytics but not available in built-in SQL functions.

For an optimal case where time series discords exist, we utilized Channel I data of a patient. The original data does not include time series discords during the 60 second time frame. Therefore, we synthesized the data to have 2 time series discords in a symbolized PAA, in order to compare the performance of two different SAX algorithms. In this case, we set f, a and w as 5, 3 and 5 respectively. The brute-force algorithm (Algorithm 1) was executed with the time complexity of $O(n^2)$, while the heuristic algorithm (Algorithm 2) was completed the job within one clock cycle, where the number of clock cycles per second is 100 for the IBM PureData system that we utilized (Figure 5). Compared to [17] where the author utilized Core2Duo 2.2 GHz 1GB RAM PC, this seems as a huge time efficiency improvement. However, as we did not used same data as the author utilized, further investigation is required.

File	Size	Execution Time (sec)
Vital	78 KB	0.1
Waveform	847 MB	1.08

TABLE IV: $nzload$ Execution Time.

Input Data Duration (sec)	Execution Time (sec)
600	0.097
1800	0.166
3000	0.202
4200	0.243
5400	0.304
6600	0.330

TABLE V: Execution Time for Decoding 16 Digit Binary $SpO2$ Data using UDTF.

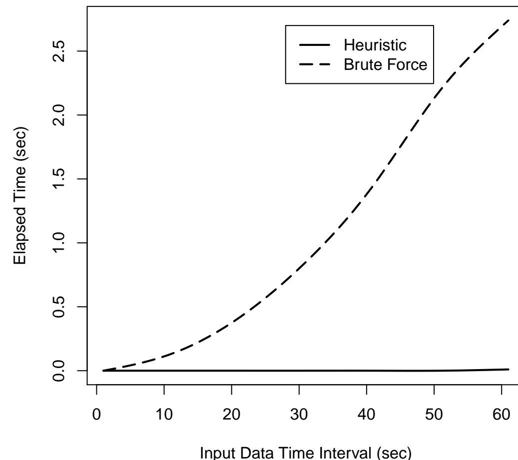

Fig. 5: Execution Time of Brute Force and Heuristic Algorithms with Time Series Discords in Data.

However, if there is no anomaly existing, Algorithm 2 was also executed with the time complexity of $O(n^2)$, since an early termination in the inner loop was not available (Figure 6). In this case, we used $SpO2$ data and set f, a and w as 5, 3 and 5 respectively.

In order to verify the effect of different bin (f) and window (w) sizes, we applied the heuristic algorithm with different bin and window sizes (Figure 7 and Figure 8). We utilized data from the same patient data used for Figure 6. The experiment results show that a case with a smaller bin size and a larger window size works faster.

For comparing time efficiency with different alphabet sizes (a), we utilized the same $SpO2$ data set as above. Compared to when $a = 4$, the heuristic search was completed earlier when a was set to 3 (Figure 9). In this case, we only evaluated when a is either 3 or 4, based on the studies [11][13] which indicate that the best value for a is generally 3 or 4.

VI. Conclusion

In this study, we utilized a parallel relational database, IBM PureData for Analytics to detect time series discords in medical data. We applied two versions of symbolic aggregate

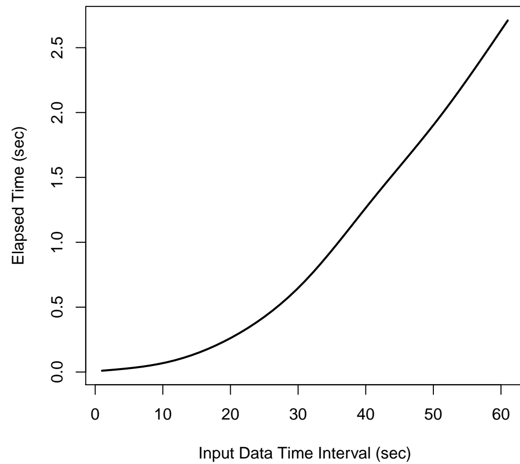

Fig. 6: Heuristic Algorithm Execution Time without Time Series Discords in Data.

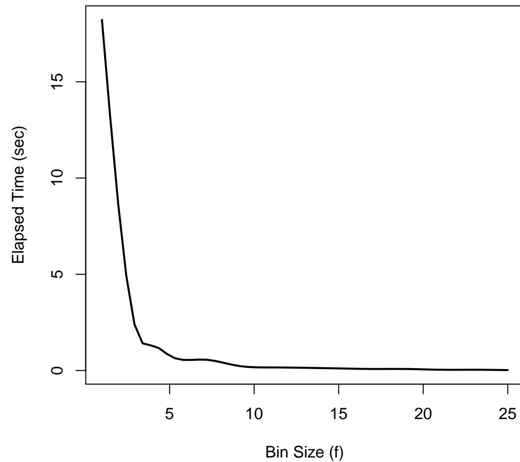

Fig. 7: Heuristic Algorithm Execution Time with Different Bin Sizes (f) (a = 3, w = 5).

approximation algorithms to data collected from patients. The experiment results showed efficient data loading, decoding and time series discord detection using the high performance parallel database. Additionally, the heuristic version of the time series discord detection algorithm works with $O(n)$ when a discord exists in time series data. However, if there is no discord in data, the algorithm works in $O(n^2)$. The experiment results indicate that how to tune the window size (w), alphabet size (a) and bin size (f) parameters determines the efficiency of the algorithm especially for the case without a discord.

It must be worth to investigate other algorithms working better in a case where an anomaly does not exist in data. Especially in the biomedical domain, algorithms should work

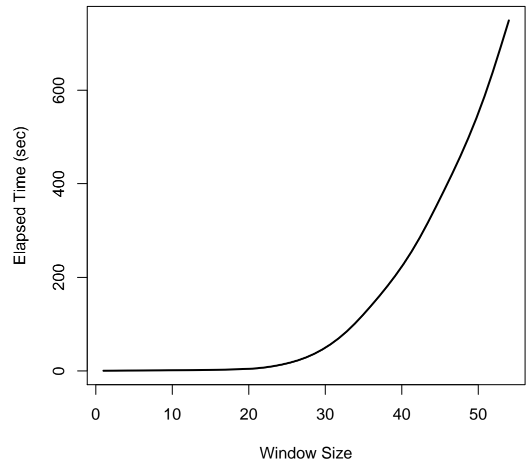

Fig. 8: Heuristic Algorithm Execution Time with Different Window Sizes (w) (f = 5. a = 3).

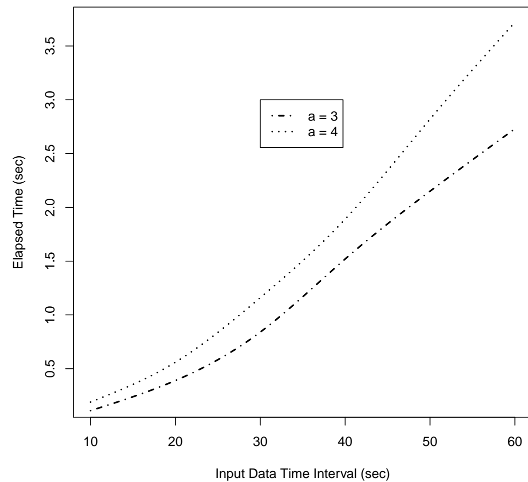

Fig. 9: Heuristic Algorithm Execution Time with Different Alphabet Sizes (a) (f = 5. w = 5).

in real time regardless of existence of an anomaly. Additionally, instead of using a single time series data only, techniques which consider multiple data types contributing to a time series discord should be investigated [18].

MapReduce on distributed systems works well for semi-structured data set without a schema and complex data analytics requiring multiple passes over data [10]. Since Hadoop MapReduce and parallel DBMS have different strengths for complex data analytics with different formats of data, comparing and utilizing both systems to get the best results should be considered as a future work.

ACKNOWLEDGMENT

We are grateful to Bill Goldman and Christopher Beggio at Sandia National Labs for their heroic efforts in arranging for the care and feeding of the Netezza systems. Without their persistence in navigating a procedural maze this project might not have happened at all.

REFERENCES

[1] M.-k. Suh, C.-A. Chen, J. Woodbridge, M. K. Tu, J. I. Kim, A. Nahapetian, L. S. Evangelista, and M. Sarrafzadeh, "A remote patient monitoring system for congestive heart failure," *Journal of medical systems*, vol. 35, no. 5, pp. 1165–1179, 2011.

[2] A. T. Wilson and M. D. Rintoul, "Evaluating parallel relational databases for medical data analysis." Sandia National Laboratories, Tech. Rep., 2012.

[3] P. Francisco, "The netezza data appliance architecture: a platform for high performance data warehousing and analytics," *IBM Redbooks*, 2011.

[4] E. Keogh, J. Lin, S.-H. Lee, and H. Van Herle, "Finding the most unusual time series subsequence: algorithms and applications," *Knowledge and Information Systems*, vol. 11, no. 1, pp. 1–27, 2007.

[5] M. C. Chuah and F. Fu, "Ecg anomaly detection via time series analysis," in *Frontiers of High Performance Computing and Networking ISPA 2007 Workshops*. Springer, 2007, pp. 123–135.

[6] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] Q. Yao, Y. Tian, P.-F. Li, L.-L. Tian, Y.-M. Qian, and J.-S. Li, "Design and development of a medical big data processing system based on hadoop," *Journal of medical systems*, vol. 39, no. 3, pp. 1–11, 2015.

[9] F. Wang, V. Ercegovac, T. Syeda-Mahmood, A. Holder, E. Shekita, D. Beymer, and L. H. Xu, "Large-scale multimodal mining for healthcare with mapreduce," in *Proceedings of the 1st ACM International Health Informatics Symposium*. ACM, 2010, pp. 479–483.

[10] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: friends or foes?" *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, 2010.

[11] E. Keogh, J. Lin, and A. Fu, "Hot sax: Efficiently finding the most unusual time series subsequence," in *Data mining, fifth IEEE international conference on*. IEEE, 2005, pp. 8–pp.

[12] J. Lin, E. Keogh, S. Lonardi, and B. Chiu, "A symbolic representation of time series, with implications for streaming algorithms," in *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*. ACM, 2003, pp. 2–11.

[13] S. Rombo and G. Terracina, "Discovering representative models in large time series databases," in *Flexible Query Answering Systems*. Springer, 2004, pp. 84–97.

[14] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," in *SODA*, vol. 97, 1997, pp. 360–369.

[15] K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array," in *Algorithms and Computation*. Springer, 2000, pp. 410–421.

[16] M.-k. Suh, J. Woodbridge, M. Lan, A. Bui, L. S. Evangelista, and M. Sarrafzadeh, "Missing data imputation for remote chf patient monitoring systems," in *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*. IEEE, 2011, pp. 3184–3187.

[17] M. T. Son and D. T. Anh, "Some novel heuristics for finding the most unusual time series subsequences," in *Advances in Intelligent Information and Database Systems*. Springer, 2010, pp. 229–240.

[18] X. Li and J. Han, "Mining approximate top-k subspace anomalies in multi-dimensional time-series data," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 447–458.