


Summer 7-15-2017

Distributed Knowledge Discovery for Diverse Data

Hossein Hamooni
University of New Mexico

Follow this and additional works at: https://digitalrepository.unm.edu/cs_etds

 Part of the [Numerical Analysis and Scientific Computing Commons](#), [Other Computer Engineering Commons](#), and the [Other Computer Sciences Commons](#)

Recommended Citation

Hamooni, Hossein. "Distributed Knowledge Discovery for Diverse Data." (2017). https://digitalrepository.unm.edu/cs_etds/86

This Dissertation is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Computer Science ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Hossein Hamooni

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Abdullah Mueen

, Chairperson

Shuang Luan

Trilce Estrada

Amy Neel

Distributed Knowledge Discovery for Diverse Data

by

Hossein Hamooni

B.S., Computer Engineering, Amirkabir University of Technology, 2010

M.S., Computer Networks, Amirkabir University Technology, 2013

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2017

Dedication

To Nikan, my amazing wife, who has supported me endlessly throughout the process. I would not be who I am today if not for her support. She is the most caring person I know in my life and I wish her years of success to come. I also dedicate this dissertation to my parents, Abbas and Fatemeh, for all they have done for me. They are my true heroes who taught me to believe in myself and in my dreams. Finally, I dedicate this work to my wonderful sisters, Hanieh and Saeedeh, who have always been there for me.

Acknowledgments

I would like to thank my advisor, Abdullah Mueen, for all I have learned from him. He taught me how to find interesting problems and do original research. I enjoyed working with such a young and vibrant advisor. I would not have a successful Ph.D. without his support.

I would like to give my sincere thanks to other members on my Ph.D. committee, Shuang Luan, Trilce Estrada, and Amy Neel for serving on my committee and for giving critical feedback on this dissertation.

I am also grateful to my collaborators from other institutes. I would like to thank Biplob Debnath of NEC Labs, Jianwu Xu of NEC Labs, Hui Zhang of NEC Labs, and Hao Yang of Visa Research.

Finally, I would like to give special thanks to my wife, Nikan, who has been a great collaborator in different projects. I have the honor of co-authoring several research papers with her.

Distributed Knowledge Discovery for Diverse Data

by

Hossein Hamooni

B.S., Computer Engineering, Amirkabir University of Technology, 2010

M.S., Computer Networks, Amirkabir University Technology, 2013

Ph.D., Computer Science, University of New Mexico, 2017

Abstract

In the era of new technologies, computer scientists deal with massive data of size hundreds of terabytes. Smart cities, social networks, health care systems, large sensor networks, etc. are constantly generating new data. It is non-trivial to extract knowledge from big datasets because traditional data mining algorithms run impractically on such big datasets. However, distributed systems have come to aid this problem while introducing new challenges in designing scalable algorithms. The transition from traditional algorithms to the ones that can be run on a distributed platform should be done carefully. Researchers should design the modern distributed algorithms based on the problem domain. The main goal of this dissertation is to demonstrate the importance of domain specific knowledge in developing scalable knowledge discovery algorithms on distributed systems. Data properties such as origin, type, context and size play important roles to achieve speed, efficiency and scalability. In this dissertation, I describe three domain specific knowledge discovery systems on three diverse domains: a distributed algorithm to extract patterns

from log messages generated by computers, a distributed algorithm to find abnormal behavior in social media, and a scalable algorithm for matching patterns in streaming time series data. I explain how to exploit the data properties in a distributed knowledge discovery system to achieve scalability and speed. The algorithms achieve horizontal scalability for any data size, and the systems are currently deployed at the University of New Mexico.

Contents

List of Figures	xii
List of Tables	xvii
1 Introduction	1
1.1 Pattern recognition for computer log messages	2
1.2 Anomaly detection in social media	3
1.3 Online pattern recognition for streaming time series	4
2 Pattern recognition for computer log messages	5
2.1 Introduction	5
2.2 Related Work and Background	8
2.2.1 Motivating Examples	8
2.2.2 Pattern Recognition Framework	9
2.2.3 Challenges	10
2.3 Fast Log-Pattern Mining	11

<i>Contents</i>	viii
2.3.1 Tokenization and Type Detection	11
2.3.2 Fast and Memory Efficient Clustering	12
2.3.3 Log Pattern Recognition	17
2.3.4 Hierarchy of Patterns	19
2.4 Evaluation	22
2.4.1 HLAer: The Baseline	23
2.4.2 Datasets	24
2.4.3 Accuracy	25
2.4.4 Memory Usage	27
2.4.5 Running Time	28
2.4.6 Map-Reduce vs. Sequential	28
2.4.7 Parameter Sensitivity	30
2.5 Case Study	31
2.6 Conclusion	33
3 Anomaly detection in social media	34
3.1 Introduction	34
3.2 Background	37
3.2.1 Data Collection	38
3.2.2 Complex Handovers	39
3.3 Detecting URL Changes and Handovers	40

<i>Contents</i>	ix
3.3.1 Active Detection (Probing)	41
3.3.2 Data-driven Detection	41
3.4 Handover Analysis	47
3.4.1 Temporal Profile	47
3.4.2 Content Profile	51
3.4.3 URL Change Analysis	53
3.4.4 Connectivity Profile	54
3.4.5 Lag Profile	56
3.4.6 Handover Lag Distribution	58
3.5 Why Handovers?	59
3.5.1 Mentions and External URLs	59
3.5.2 Suspension	60
3.6 Experiments	61
3.6.1 Scalability	61
3.7 Related Work	63
3.8 Conclusion	64
4 Distributed Pattern Matching over Streaming Time Series	65
4.1 Introduction	65
4.2 Background and Definition	67
4.2.1 Problem Definition	67

4.2.2	Distributed Stream Processing Systems	69
4.3	General Framework	70
4.3.1	Windowing	71
4.3.2	Distribution	72
4.3.3	Computation	73
4.4	DisPatch in Map-Reduce	74
4.4.1	Map-Reduce Framework	74
4.4.2	Analysis of the map-reduce framework	77
4.4.3	Supported Distance Measures	77
4.5	Exploiting The Overlap	81
4.5.1	Euclidean Distance	82
4.5.2	Correlation Coefficient	84
4.5.3	Manhattan Distance	84
4.5.4	DTW Distance	85
4.6	Empirical Evaluation	85
4.6.1	Datasets	86
4.6.2	Data Rate and Dictionary Size	88
4.6.3	Speed up by Exploiting the Overlap	89
4.6.4	Scalability	90
4.6.5	Delay Sensitivity	90
4.7	Applications	91

<i>Contents</i>	xi
4.7.1 Patient Monitoring	92
4.7.2 Power Consumption Monitoring	93
4.8 Related Work	94
4.9 Conclusion	95
5 Conclusion and Future Work	96
References	98

List of Figures

2.1	Extracting log patterns for a given set of logs. Hierarchy of patterns gives user the flexibility to choose a level based on his needs.	6
2.2	Creating hierarchy of patterns by using a fast clustering algorithm and a fast pattern recognition algorithm.	9
2.3	Two examples of how pre-processing works on the input log messages.	12
2.4	An example of how Algorithm 2 works.	18
2.5	Pattern selection illustration with $a_1 = 1$, $a_2 = 0$, and $a_3 = 0$ in the cost function.	22
2.6	Comparison of running times of the sequential and the map-reduce clustering. (right) Comparing the running time of LogMine with <i>HLAer</i> . The y-axis is in log scale.	26
2.7	Sensitivity of the algorithm to the parameters $MaxDist_0$ and α	29
3.1	A user (Twitter id: 2664619086) with ten URL changes on 7 November 2015. Some URLs are used more than once (dotted connections), which form a loop of URLs.	35

3.2 The URL `paradisecameron` was handed over among four users nine times. User 468 appears in the handover chain exactly every other time. The dashed lines connecting the same users show *loops* in this chain. 36

3.3 $User_1$ handed over the URL `Tom` to $User_2$. The release time is t_2 and the claim time is t_3 , and the real handover lag is $t_3 - t_2$. We calculate an upper bound, $t_4 - t_1$, for the handover lag based on the last tweet of $User_1$ at t_1 before the handover, and the first tweet from $User_2$ at t_4 after the handover. 38

3.4 The process of detecting URL handovers in the Twitter network. 2 URL handovers are detected from 9 tweet objects in this example. 46

3.5 An example of a user with daily periodicity and a strong association with handover. 49

3.6 Comparing the normal activity of a user with its activity near the handovers. 97.4% of the users have higher activity-per-hour around handovers. Both x-axis and y-axis are in log scale. 49

3.7 Three accounts with almost identical activity profile and correlated handovers. Handovers initiate change in activity patterns. 50

3.8 The plot shows how the content of the tweets changes with URL changes. 52

3.9 (left) Percentage of users (out of 231,800 users) based on the frequency of changing URL. (right) The probability of a user doing a URL handover given the URL change frequency. The probability reaches 1 for a user with frequency higher than 68 (almost one URL change every day). 54

3.10 The bipartite graph that shows the connections between users and URLs. The dashed nodes form a connected component of size 5. User U_4 has not been involved in any handovers. 56

3.11 The distribution of handovers based on their lags. (left) The lags that are calculated using our probing technique. (right) The lags that are calculated using the data-driven approach. 50th percentiles are shown in both distributions. 57

3.12 This figure shows the number of URLs for a given handover lag and a given number of handovers on that URL. The X axis is in log scale. As the number of handovers on a URL increases, the average of handover lags on that URL goes down drastically. We can see many different lags for URLs that are handed over just once. 58

3.13 The percentage of URLs based on the number of mentions for random URLs and handover URLs. 60

3.14 Twitter suspension rate of handover users. 61

3.15 (left) How running time changes as we add more tweets. (right) How adding more working nodes helps us to detect the handover faster. We get an almost linear speedup by adding more nodes. 63

4.1 A dictionary of five patterns and a stream moving from right to left. Streaming pattern matching systems identifies any occurrence of any of the patterns in the stream. Pattern C and E appear in the stream with significant correlation. 66

4.2 An overview of the inputs, outputs, and how they are connected to our system. Each source (stream) has its own dictionary of patterns. 69

4.3 The general framework of our system. Step 1: How we create batches out of the streaming time series (red segments). Step 2: How We create partitions for each batch (green segments). Two parameters w_s (window size) and w_o (window overlap size) are also shown in the figure. 72

4.4 A tiny example that shows how the DisPatch framework works in map-reduce fashion. We use coloring to show the correspondence between the patterns and the partitions we generate in the *Distribution* step. In this example, the DisPatch framework has found a match with patterns 3 starting from time 5. Note that this is the default *Distribution* and *Computation* strategy where we slide the partition window one point at a time. In the optimized version of our system, we may slide the partition window multiple points, and exploit the overlap between successive distance calculations. 77

4.5 A toy example of convolution operation being used to calculate sliding dot products for time series data. Note the reverse and append operation on X and Y in the input. Fifteen dot products are calculated for every slide. The cells m ($= 2$) to n ($= 4$) from left (green/bold arrows) contain valid products. 83

4.6 The results from the power consumption data. We use 16 workers and set $max_D = 1$ in all these experiments. As long as the points are below the red dashed line, the DisPatch framework can handle the stream in real time. It would get behind the stream for all the points above the dashed line. 87

4.7 The results from the EURO/USD exchange rate data. We use 16 workers and set $max_D = 1$ in all these experiments. As long as the points are below the red dashed line, the DisPatch can handle the stream in real time. It would get behind the stream for all the points above the dashed line. 88

4.8 A set of experiments on power consumption data. (left) Scalability of DisPatch by adding more workers to the system. The minimum number of workers to not get behind the input stream is 4 in this example. (middle) The effect of the delay parameter on the processing time. Higher delays let DisPatch work under less pressure. Both number of workers and delay (max_D) have the same effect for all distance measures. (right) The comparison between the performance of DisPatch and the early abandoning technique to find matches under Pearson’s correlation. DisPatch can process 20 patterns at 4000 Hz while the early abandoning technique can do the same number of patterns at 1000 Hz. 89

4.9 The comparison between performance of DisPatch and early abandoning technique on EEG data at 256 Hz. DisPatch is able to process data up to 14 times faster than the early abandoning technique. DisPatch can process 30 patterns with 1 second delay in real time. The total length of these 30 pattern is 522K points (34 minutes). 90

4.10 EEG recording of an epileptic patient for 21 hours at 256 Hz. We create an enhanced dictionary in an unsupervised fashion in the first 3.5 hours, and predict 10 out of 11 seizures well ahead of time in the remaining 18 hours. Red arrows show the index of the matched patterns. 92

4.11 Whole house power consumption data [49] in the top. We build a set of rules using [70]. The antecedents of the rules form a dictionary, in this case a washer dictionary. DisPatch matches the dictionary to the subsequent data to predict dryer usages. 93

List of Tables

2.1	A summary of all datasets.	24
2.2	The accuracy of pattern recognition and the agreement score for different datasets. We do not report these measurements on D3 because HLAer cannot handle it.	28
3.1	The tweets from the same user with 2 URLs. The user change its URL from <code>zflexins</code> to <code>loveyorslf</code> on 11 December 2015. All the tweets of the left column are about Justin Bieber, and the ones in the right column are about Harry Styles (both are famous singers). The average in-URL similarity is 0.35, while the across-URL similarity is 0.03.	53
3.2	Running time of the data-driven algorithms to detect URL Handovers and Changes.	62
4.1	List of supported distance (or similarity) measures and the corresponding matching functions.	78
4.2	How each parameter is related to the complexity of the dictionary matching problem.	89

Chapter 1

Introduction

Data mining researchers have to come up with new knowledge discovery algorithms because the size of the data that should be processed by computers is constantly increasing. Smart cities, social networks, health care systems, large sensor networks, etc. are constantly generating huge amount of data. For example, Facebook has to process 600 TB of new data every day [15]. Twitter also receives 500 million tweets daily [21]. It is **not** a trivial task to make traditional data mining algorithms scalable for these huge datasets. We have to design the algorithms completely from scratch in many cases. In order to have a distributed, fast and efficient knowledge discovery algorithm for a given huge dataset, we should consider the specifications and the properties of that dataset. In other words, we should exploit the natural properties of the dataset to make the algorithm efficient. For the sake of clarity, let us discuss an examples.

Dynamic Time Warping (DTW) is a distance measure defined for two given time series and widely used in time series mining tasks like classification and clustering. The two given time series are warped non-linearly in the time dimension to determine a measure of their similarity independent of certain non-linear variations in the time dimension. DTW, in its original formulation, is an expensive measure to calculate. It takes $O(n^2)$ to cal-

culates the DTW distance for time series of length n . [58] introduces an exact method to calculate the DTW distance for sparse binary time series which mostly contains zeros. The authors have taken advantage of the time series sparsity and improved the time complexity of DTW calculation to $O(m^2)$ where m is the number of non-zero values. Since $m \ll n$ for a sparse time series, this translates to a huge speed up. This method has also the same amount of improvement in space complexity compared to the original DTW. In one application, authors have shown that they could cluster 4,170 Twitter users based on their temporal activities by using their method in 18 minutes. The same task takes 180 hours by using the original DTW formulation. This means that they got the exact same results 557 times faster just by customizing the algorithm for the sparse binary time series. This example clearly shows that we can exploit the properties of data (sparsity in this case), to design more efficient data mining algorithms for specific data.

In this thesis, we will discuss scalable distributed knowledge discovery algorithms for three different data types: computer log messages, social media, and streaming time series. For each data type, we discuss the properties of the data, and how we can design an efficient knowledge discovery algorithm based on those properties. All the problems we are going to discuss have real world applications.

1.1 Pattern recognition for computer log messages

Modern engineering incorporates smart technologies in all aspects of our lives. Smart technologies are generating terabytes of log messages every day to report their status. It is crucial to analyze these log messages and present usable information (e.g. patterns) to administrators, so that they can manage and monitor these technologies. Patterns minimally represent large groups of log messages and enable the administrators to do further analysis, such as anomaly detection and event prediction. Although patterns exist commonly in automated log messages, recognizing them in massive set of log messages from heterogeneous sources without any prior information is a significant undertaking. We pro-

pose a method, named LogMine, that extracts high quality patterns for a given set of log messages. Our method is fast, memory efficient, accurate, and scalable. LogMine is implemented in map-reduce framework for distributed platforms to process millions of log messages in seconds. LogMine is a robust method that works for heterogeneous log messages generated in a wide variety of systems. Our method exploits algorithmic techniques to minimize the computational overhead based on the fact that log messages are always automatically generated. We evaluate the performance of LogMine on massive sets of log messages generated in industrial applications. LogMine has successfully generated patterns which are as good as the patterns generated by exact and unscalable method, while achieving a $500\times$ speedup. This work is published in CIKM 2016. We discuss the details in Chapter 2.

1.2 Anomaly detection in social media

Social media sites (e.g. Twitter and Pinterest) allow users to change the name of their accounts. A change in the account name results in a change in the URL of the user's homepage. We develop an algorithm that discovers a large number of social media accounts performing *synchronous* and *collaborative* URL changes. We identify various types of URL changes such as handover, exchange, serial handover and loop exchange. All such behaviors are likely to be automated behavior and may indicate accounts either already involved in malicious activities or being prepared to do so. We focus on *URL handovers* where a URL is released by a user and claimed by another user.

In this work, we analyze URL changes and handovers in social media, and find interesting association between handovers and temporal, textual and network behaviors of users. We show several anomalous behaviors from suspicious users for each of these associations. We identify that URL handovers are instantaneous automated operations. We further investigate to understand the benefits of URL handovers, and identify a strong association with misleading internal links and avoiding suspension mechanisms of the hosting

sites. Our handover detection algorithm, which makes such analysis possible, is scalable to process millions of posts (e.g. tweets, pins) and shared publicly online. This work is published in SocInfo 2016, and we will discuss the details in Chapter 3.

1.3 Online pattern recognition for streaming time series

Matching a dictionary of patterns to a streaming time series is a primary component of real-time monitoring systems such as earthquake monitoring, industrial process monitoring and patient monitoring. Until now, the problem has been solved independently with smart pruning strategies, efficient approximation and pattern indexing among many others. With advanced sensing and storing capabilities, the complexity of the dictionary matching problem is fast growing with larger dictionary size and faster data streams warranting a distributed matching process robust enough for arbitrary complexity. In this work, we develop and evaluate novel distribution strategies combining state-of-the-art algorithmic optimization techniques with the Streaming Spark framework. Our distribution strategies covers the entire problem space for uniformly sampled streams, and are arbitrarily scalable. We will discuss the details in Chapter 4.

Chapter 2

Pattern recognition for computer log messages

2.1 Introduction

The Internet of Things (IoT) enables advanced connectivity of computing and embedded devices through internet infrastructure. Although computers and smartphones are the most common devices in IoT, the number of “things” is expected to grow to 50 billion by 2020 [9]. IoT involves machine-to-machine communications (M2M), where it is important to continuously monitor connected machines to detect any anomaly or bug, and resolve them quickly to minimize the downtime. Logging is a commonly used mechanism to record machines’ behaviors and various states for maintenance and troubleshooting. An acceptable logging standard is yet to be developed for IoT, most commonly due to the enormous varieties of “things” and their fast evolution over time. Thus, it is extremely challenging to parse and analyze log messages from systems like IoT.

An automated log analyzer must have one component to *recognize* patterns from log messages, and another component to *match* these patterns with the inflow of log messages

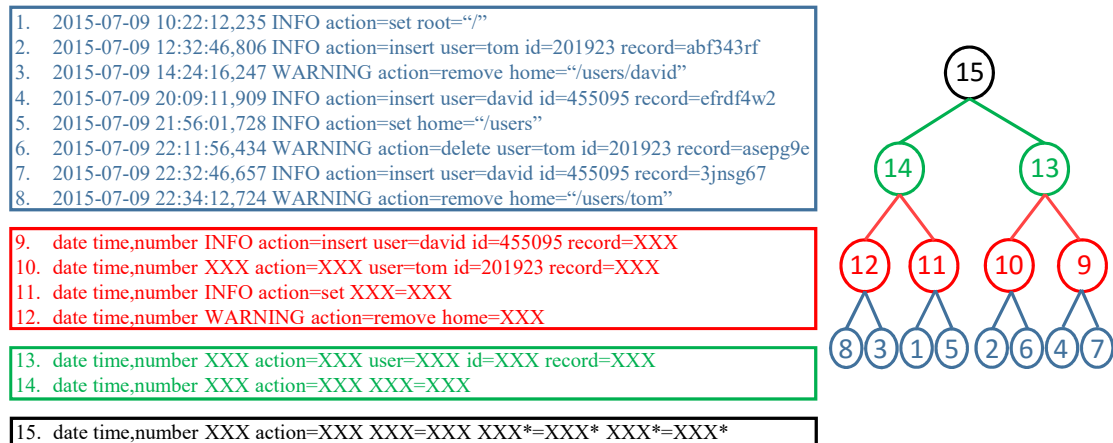


Figure 2.1: Extracting log patterns for a given set of logs. Hierarchy of patterns gives user the flexibility to choose a level based on his needs.

to identify events and anomalies. Such a log message analyzer must have the following desirable properties:

- **No-supervision:** The pattern recognizer needs to be working from the scratch without any prior knowledge or human supervision. For a new log message format, the pattern recognizer should not require an input from the administrator.
- **Heterogeneity:** There can be log messages generated from different applications and systems. Each system may generate log messages in multiple formats. An automated recognizer must find all formats of the log messages irrespective of their origins.
- **Efficiency:** IoT-like systems generate millions of log messages every day. The log processing should be done so efficiently that the processing rate is always faster than the log generation rate.
- **Scalability:** Pattern recognizer must be able to process massive batches of log messages to maintain a current set of patterns without incurring CPU and memory bottlenecks.

Many companies such as Splunk[17], Sumo Logic[18], Loggly[10], LogEntries[11], etc. offer log analysis tools. Open source packages such as Elasticsearch[6], Graylog[8]

and OSSIM[13] have also been developed to analyze logs. Most of these tools and packages use regular expressions (regex) to match with log messages. These tools assume that the administrators know how to work with regex, and there are plenty of tools and libraries that support regex. However, these tools do not have the desirable properties mentioned earlier. By definition, these tools support only supervised matching. Human involvement is clearly non-scalable for heterogeneous and continuously evolving log message formats in systems such as IoT, and it is humanly impossible to parse the sheer number of log entries generated in an hour, let alone days and weeks. On top of that, writing regex rules is long, frustrating, error-prone, and regex rules may conflict with each other especially for IoT-like systems. Even if a set of regex rules is written, the rate of processing log messages can be slow due to overgeneralized regexes.

A recent work on automated pattern recognition has shown a methodology, called *HLAer*, for automatically parsing heterogeneous log messages [59]. Although *HLAer* is unsupervised and robust to heterogeneity, it is not efficient and scalable because of massive memory requirement and communication overhead in parallel implementation.

In this work, we present an *end-to-end* framework, **LogMine**, that addresses all of the discussed problems with the existing tools and packages. LogMine is an unsupervised framework that scans log messages only *once* and, therefore, can quickly process hundreds of millions of log messages with a very *small amount of memory*. LogMine works in iterative manner that generates a hierarchy of patterns (regexes), one level at every iteration. The hierarchy provides flexibility to the users to select the right set of patterns that satisfies their specific needs. We implement a map-reduce version of LogMine to deploy in a massively parallel data processing system, and achieve an impressive 500× speedup over a naive exact method.

We discuss related works in Section 2.1, and background in Section 2.2. We describe our proposed method in Section 2.3. Section 2.4 and 2.5 discuss the experimental findings and case studies on the related problems respectively. Finally, we conclude in Section 2.6.

2.2 Related Work and Background

Authors of [83] have proposed a method to cluster the web logs without any need to user-defined parameters. Their method is not scalable to large datasets because the time complexity is $O(n^3)$ where n is the number of the logs. [35] introduces a method to create the search index of a website based on the users' search logs. [63] discusses a pre-processing algorithm that extracts a set of fields such as IP, date, URL, etc. from a given dataset of web logs. Authors of [36] have proposed a method to help website admins by extracting useful information from users' navigation logs.

In [29], the authors have clearly reasoned why map-reduce is the choice for log processing rather than RDBMS. Authors have showed various join processing techniques for log data in map-reduce framework. This work, along with [45], greatly inspired us to attempt clustering on massive log data. In [79], the authors have described a log-structured database system that is optimized towards write-heavy data such as logs. In [43], the authors describe a unified logging infrastructure for heterogeneous applications. Our framework is well suited to work on top of both of these infrastructures with minimal modification.

In HPC (High Performance Computing), logs have been used to identify failures, and troubleshoot the failures in large scale systems [60][51]. Such tools majorly focus on categorizing *archived* log messages into sequence of failure events, and use the sequence to identify root cause of a problem.

2.2.1 Motivating Examples

In Figure 2.1, we show examples of log patterns that our method generates. Our method clusters the messages into coherent groups, and identifies the most specific log pattern to represent each clusters. In Figure 2.1, the input log segment has four different clusters. The messages within each cluster can be merged to produce the log patterns shown in red.

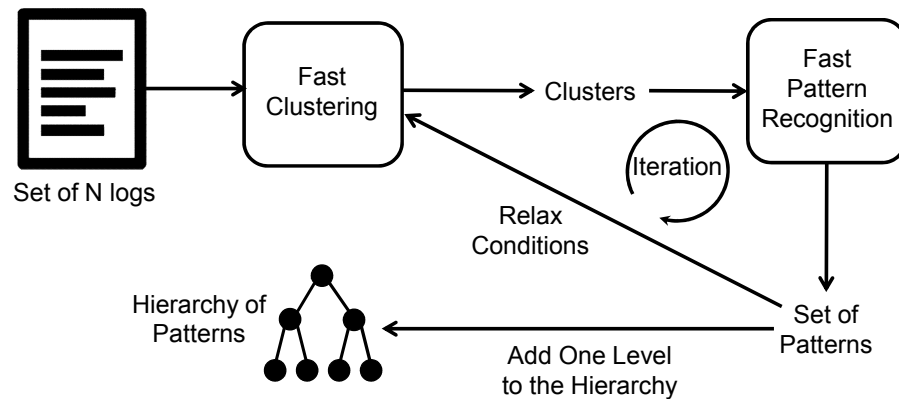


Figure 2.2: Creating hierarchy of patterns by using a fast clustering algorithm and a fast pattern recognition algorithm.

These four patterns can again be clustered and merged to form two more general patterns (in green). Same process can be repeated till we get to the root of the hierarchy which contains the most general pattern. Note that, there are three basic types of fields in each pattern: **Fixed value**, **Variable** and **Wildcard**. A fixed value field is matched just with one unique value like *www*, *httpd* and *INFO*. A variable field is matched with any value of a certain type such as *number*, *IP address* and *date*. Wildcards are matched with values of all types.

Clearly the most generic pattern is a set of wildcards, and the most specific pattern is a set of fixed attributes. None of these patterns are useful for the administrators. Our method produces a hierarchy of patterns: specific patterns are children of general patterns. Such hierarchy is useful for the system administrators to pick the right level of detail they want to track in the log messages as opposed to write regular expression manually.

2.2.2 Pattern Recognition Framework

With the above motivation, we design a novel framework for LogMine as shown in the Figure 2.2. LogMine takes a large batch of logs as input and clusters them very quickly

with a restrictive constraints using the clustering module. A pattern is then generated for each cluster by our pattern recognition module. The sets of patterns form the leaf level of the pattern hierarchy. The method will continue to generate clusters with relaxed constraints in subsequent iterations and merge the clusters to form more general patterns, which will constitute a new parent level in the hierarchy. LogMine continues to iterate until the most general pattern has been achieved and/or the hierarchy of patterns is completely formed.

The framework meets all the criteria of a good log analytics tool. It is an unsupervised framework that does not assume any input from the administrator. The framework produces a hierarchy of patterns which is interpretable to the administrator. The framework does not assume any property in sources of the log messages. The recognizer can work on daily or hourly batches if the following challenges can be tackled.

2.2.3 Challenges

This framework for log pattern recognition is unsupervised and suitable for heterogeneous logs. However, the scalability of the framework depends on the two major parts of the framework: *log clustering* and *pattern recognition*. Standard clustering and recognition methods do not scale well, and it is non-trivial to design scalable versions of the clustering and recognition modules for massive sets of log messages. Since the two modules work in a closed loop, we must speedup both of them to scale the framework for large datasets.

To quantify the significance of the challenge, let us imagine an average website that receives about 2 million visitors a day (much less compared to 500 million tweets a day in Twitter, or 3 billion searches a day in Google). Even if we assume that each visit results in only one log message, 2 million log messages per day is a reasonable number. Clustering such a large number of log messages in only one iteration is extremely time consuming. A standard DBSCAN [82] algorithm takes about two days to process a dataset of this size with state-of-the-art optimization techniques [4]. Similar amount of time would be needed

by k-means algorithm although no one would know the best value for the parameter k . Clearly, the framework cannot work on daily batches using standard clustering algorithms and, therefore, we need an optimized clustering algorithm developed for log analysis. A similar argument can be made for the recognition module where a standard multiple sequence alignment operation on a reasonable sized cluster may need more than a day to recognize the patterns.

2.3 Fast Log-Pattern Mining

The key intuition behind our log mining approach is that *logs are automatically generated messages* unlike sentences from a story book. There are some specific lines in an application source code that produce the logs, therefore, all log messages from the same application are generated by a finite set of formats. Standard clustering and merging methods do not consider any dependency among the objects in the dataset. In logs, the dependency among the messages is natural, and as we show in this work, the dependency is useful to speedup the clustering and the merging process.

2.3.1 Tokenization and Type Detection

We assume all the logs are stored in a text file and each line contains a log message. We do a simple pre-processing on each log. We tokenize every log message by using white-space separation. We then detect a set of pre-defined types such as date, time, IP and number, and replace the real value of each field with the name of the field. For instance, we replace *2015-07-09* with *date*, or *192.168.10.15* with *IP*. This set of pre-defined types can be configured by the user based on his interest in the content over the type of a field. Figure 2.3 shows an example of tokenization and type replacements in a log message. Tokenization and type detection is embedded in the clustering algorithm without adding any overhead due to the one-pass nature of the clustering algorithm described in the next

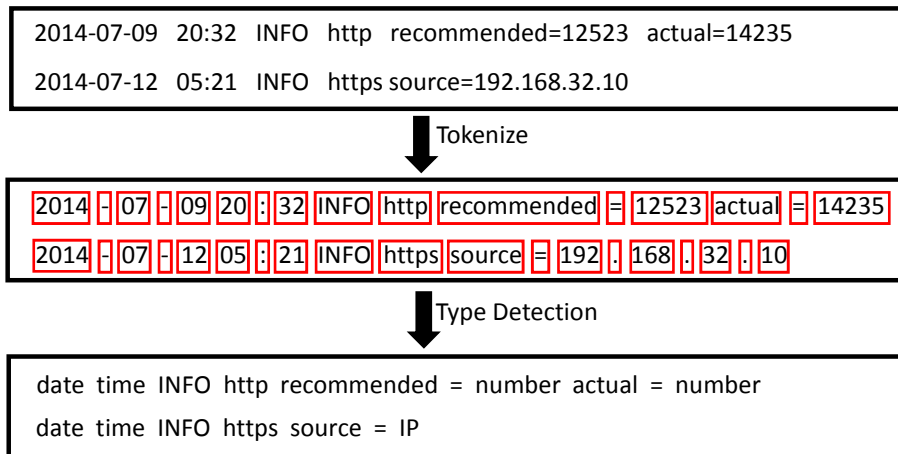


Figure 2.3: Two examples of how pre-processing works on the input log messages.

section. Although this step is not mandatory, we use it to make the similarity between two logs meaningful. If no type detection is done, two logs generated by the same pattern can have a low similarity, just because they have different values for the same field. Therefore, we may end up generating huge number of unnecessary patterns if we do not tokenize.

2.3.2 Fast and Memory Efficient Clustering

Our clustering algorithm is simply a one-pass version of the friends-of-friend clustering for the log messages. Our algorithm exploits several optimization techniques to improve the clustering performance.

Distance Function: We first define the distance between two log messages by the following equations:

$$Dist(P, Q) = 1 - \sum_{i=1}^{Min(len(P), len(Q))} \frac{Score(P_i, Q_i)}{Max(len(P), len(Q))} \quad (2.1)$$

$$Score(x, y) = \begin{cases} k_1 & \text{if } x=y \\ 0 & \text{otherwise} \end{cases} \quad (2.2)$$

P_i is the i th field of log P , and $len(P)$ is the number of fields of log P . k_1 is a tunable parameters. We set $k_1 = 1$ in our default log distance function, but this parameter can be changed to put more or less weight on the matched fields in two log messages.

Since we want to cluster patterns in the subsequent iterations of our framework, we also need a distance function for patterns. The distance between two patterns is defined very similarly as the distance between two log messages, just with a new score function.

$$Score(x, y) = \begin{cases} k_1 & \text{if } x=y \text{ and both are fixed value} \\ k_2 & \text{if } x=y \text{ and both are variable} \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

We again set $k_1 = k_2 = 1$ in our default pattern distance function. Our distance function is non-negative, symmetric, reflexive, and it satisfies the triangular inequality. Therefore, it is a metric. Log messages generated by the same format have a very small distance (zero in most cases), and log messages generated by different formats have larger distances. This is a desirable property for fast and memory-efficient log clustering algorithm. In the high dimensional space, the log messages form completely separated and highly dense regions. Therefore, finding the clusters using the above distance function is a straightforward task.

Finding Clusters: In this section, we explain how to find the dense clusters out of the input logs in a sequential fashion. The same approach will also be used when we create the hierarchy of log patterns by several iterations of clustering. First, we define an internal parameter named *MaxDist*, which represents the maximum distance between any log entry/message in a cluster and the cluster representative. Therefore, the maximum

distance between any two logs in a cluster is $2 \times \text{MaxDist}$. We start from the first log message and process all the log messages one by one until we reach the last message. Each cluster has a representative log message, which is also the first member of the cluster. For any new log message, we insert the message in one of the existing clusters only if the distance between the log and the representative is less than the MaxDist . Otherwise, when the message is not similar to any representative, we create a new cluster and put the log message as the representative of that new cluster.

The above process can be implemented in a very small memory footprint. We need to keep just one representative log for each cluster in the memory, and output the subsequent log messages without saving in the memory. This allows our algorithm to process massive number of logs with a small amount of memory. In fact, the memory usage of our clustering algorithm is $O(\text{number of clusters})$. Ignoring large number of log messages when deciding about cluster membership and using only one representative log message do not reduce the quality of the clusters. The major reason is that all the log messages in any given cluster are almost identical because they are most likely generated by the same code segment of the same application. Therefore, the above one-pass clustering algorithm with a very small MaxDist in the beginning can generate highly dense (i.e., consistent) clusters of log messages, where keeping one representative message is both sufficient and efficient.

We finally have a set of dense clusters with one representative each. As mentioned before, this algorithm is also used to cluster and merge patterns. In case of clustering the patterns, unlike the above approach, we keep all the patterns in each cluster because we will use them in the pattern recognition component. In most systems, the set of patterns generated after the first iteration fits in the memory. The speed and efficiency of this algorithms comes from the fact that the number of dense clusters does not scale with the number of log messages, because it is not possible for an application to generate huge number of unique patterns. In other words, finding a million unique patterns is impossible even in a dataset of hundreds of millions of log messages.

The one-pass clustering algorithm has a strong dependency on the order of the mes-

sages, which is typically the temporal order. The pathological worst case happens when one message from every pattern in every cluster appears very early in the log, and all of the remaining messages will have to be compared with all of the unique representatives. In practice, log messages from the same application show temporal co-location which makes them more favorable for the clustering algorithm.

Early Abandoning Technique: Early abandoning is a useful technique to speedup similarity search under Euclidean distance. Some of the initial mentions of early abandoning were in [38][23]. It has been extensively used later by many researchers for problems such as time series motif discovery [55], and similarity search under dynamic time warping (DTW) [61]. We adopt the technique for log analytics.

When comparing a new log message with a cluster representative, if the distance is smaller than *MaxDist*, we can add the new log to that cluster. Since the distance between two logs is calculated in one scan of the logs by summing up only non-negative numbers, early abandoning techniques can be applied. As we compare two logs field by field, we may discover that the accumulated distance has already exceeded the threshold, *MaxDist*, even though many of the fields are yet to be compared. In this case, we don't need to continue calculating the distance completely, because we are certain that these two logs are not in *MaxDist* radius of each other. Since the number of fields in a log can be large, this technique helps us skip a significant amount of calculation, especially when *MaxDist* is small.

Scaling via Map-Reduce Implementation: We mentioned earlier that the memory usage of our one-pass clustering algorithm is $O(\text{number of clusters})$. The one-pass clustering algorithm is very amenable to parallel execution via map-reduce approach. For each log in our dataset, we create a key-value pair. The key is a fixed number (in our case 1), and the value is a singleton list containing the given log. We also add the length based index to the value of each tuple. In the reduce function, we can merge every pair of lists. Specifically, we always keep the bigger list as the base list, and update this base list by adding all

elements of the smaller list to it (if needed). This makes the merging process faster. While adding the elements of the smaller list to the base set, we add only the elements which do not have any similar representative in the base set. If a very close representative already exists in the base list, we ignore the log. We also update the length based index of the base list meanwhile. Finally, the base list will be the result of the merging of two given lists. The pseudo code of the reduce function can be found in Algorithm 1.

Algorithm 1 Reduce

Input: Two tuples $A = (1, List_1), B = (1, List_2)$

Output: A tuple

if $size(List_1) \geq size(List_2)$ **then**

$Base_list \leftarrow List_1$

$Small_list \leftarrow List_2$

else if $size(List_1) < size(List_2)$ **then**

$Base_list \leftarrow List_2$

$Small_list \leftarrow List_1$

for $i = 1, \dots, size(Small_list)$ **do**

$Found = False$

for $j = 1, \dots, size(Base_list)$ **do**

if $d(Small_list(i), Base_list(j)) \leq MaxDist$ **then**

$Found = True$

break

if $\neg Found$ **then**

Append $Small_list(i)$ in the $Base_list$

return $(1, Base_list)$

Since we use the same key for all the logs, we will get one tuple as the final output which contains all the log representative (dense clusters). As we need to create a key-value tuple for each log, the memory usage of the map-reduce implementation is no longer $O(\text{number of dense clusters})$, in fact it is $O(\text{number of log entries})$. This is not a problem

because each worker in map-reduce platform loads a chunk of the logs. Even if a chunk of the data does not fit in memory, new map-reduce frameworks like *Spark*[86] can handle that with a small overhead. We compare the running time of both sequential and parallel implementations in Section 2.4.

2.3.3 Log Pattern Recognition

After we cluster the logs, we need to find a pattern for each cluster. Since we keep one representative for each dense cluster in the first round, the representative itself is the pattern of its cluster, but in the subsequent rounds, after we cluster the patterns, we need an algorithm that can generate one pattern for a set of logs/patterns in a cluster. We start with the pattern generation process for a pair of patterns and then generalize to a set of patterns.

Pattern Generation from Pairs: Irrespective of the pattern recognition algorithm, we always need to merge two logs at some point in the algorithm. Therefore, we shortly discuss our *Merge* algorithm here. Given two logs to be merged, we first need to find their best alignment. The best alignment of two logs is the one that generates the minimum number of wildcards and variables after merging. In the alignment process, some gaps may be inserted between the fields of each log. The alignment algorithm ensures that the length of two logs are equal after inserting the gaps. Once we have two logs with the same length, we process them field by field and generate the output. An example is shown in Figure 2.4. Note that the align step introduces gaps in the second message. The field detection step requires a straightforward scan of the two logs. A detailed pseudocode can be found in Algorithm 2. There are different algorithms for aligning two sequences. We use *Smith-Waterman* algorithm which can align two sequences of length l_1 and l_2 in $O(l_1.l_2)$ time steps [71]. Therefore, the time complexity of our *Merge* function is also $O(l_1.l_2)$. We use the same score function as in [59] for the Smith-Waterman algorithm.

Sequential Pattern Generation: To generate the pattern for a set of patterns, we start

Algorithm 2 Merge

Input: Two logs (Log_a, Log_b)

Output: A merged log

$Log'_a, Log'_b \leftarrow Align(Log_a, Log_b)$

for $i, i = 2, 3, \dots, |Log'_a|$ **do**

$x \leftarrow Field_i(Log'_a)$ and $y \leftarrow Field_i(Log'_b)$

if $x = y$ **then**

$Field_i(Log_{new}) \leftarrow x$

else if $Type(x) = Type(y)$ **then**

$Field_i(Log_{new}) \leftarrow Variable_{Type(x)}$

else

$Field_i(Log_{new}) \leftarrow Wildcard$

return Log_{new}

from the first log message, merge it with the second log, then merge the result with the third log and we go on until we get to the last one. Clearly, the success of this approach largely depends on the ordering of the patterns in the set. However, as described before, the logs inside each of the dense clusters are almost identical. This is why, in practice, the merge ordering does not associate with the quality of the final pattern. In other words, we will get the same results if we do the merging in reverse or any arbitrary order. If the logs to be merged are not similar, sequential merging may end up producing a pattern

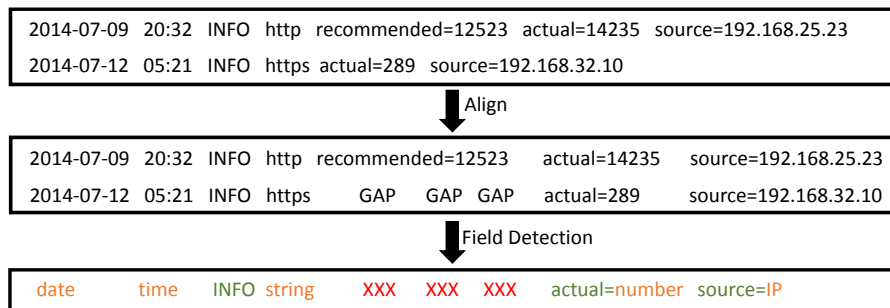


Figure 2.4: An example of how Algorithm 2 works.

with many wildcards which is not desirable. There exists techniques to find the optimal merge ordering for a set of patterns. We provide detailed experiments in the Section 2.4 to empirically show that sequential merging does not lose quality.

Scaling via Map-Reduce Implementation: As discussed before, the order of merging the logs in a cluster to create the final pattern has no effect on the output. Such sequential pattern generation can be parallelize very easily. An efficient way to implement sequential merging is using map-reduce framework. This framework can be useful whenever the order of the operation does not matter, and that is true for our case. Since the pattern recognition is done after clustering the logs, we know the exact cluster for each log. In the map function, we create a key-value pair for each log. The key is the cluster number of the log and the value is the log itself. The map-reduce framework will reduce all the key-value pairs with the same key. In the reduce function, two logs from the same cluster are merged. The final output of the reduce phase is one pattern for each cluster which is exactly what we want. If we ignore the map-reduce framework overhead, in a full parallel running of this algorithm on m machines, its time complexity is $O(\frac{n}{m}.l^2)$, where n is the number of the logs, and l is the average number of fields in each log.

2.3.4 Hierarchy of Patterns

In Sections 2.3.2 and 2.3.3, we explain how to find dense clusters of logs, and how to find one pattern that covers all the log messages in each cluster. These two modules constitute an iteration in our pattern recognition framework. We also motivate that one set of patterns generated in one of the iterations can be too specific or general, and may not satisfy the administrator. In contrast, a hierarchy of patterns can provide an holistic view of the log messages, and the administrator can pick a level with the right specificity in the hierarchy to monitor for anomalies.

In order to create the hierarchy, we use both clustering and pattern recognition algo-

rithms iteratively as shown in Figure 2.2, and produce the hierarchy in bottom-up manner. In the first iteration, we run the clustering algorithm with a very small *MaxDist* on the given set of logs. This is our most restrictive clustering condition. The output of the clustering is a (possibly large) set of dense clusters each of which has a representative log. The representative log is trivially assigned as the pattern for a dense cluster without calling the pattern recognition module. We treat these patterns as the leaves (lowest level) of the pattern hierarchy. To generate the other levels of the hierarchy, we increase the *MaxDist* parameter of the clustering algorithm by a factor of α ($MaxDist_{new} = \alpha MaxDist_{old}$) and run the clustering algorithm on the generated patterns. In other words, we run a more relaxed version of the clustering algorithm on the patterns which will produce new clusters. We then run the pattern recognition module on all the patterns that are clustered together to find more general patterns. These set of new patterns will be added to the hierarchy as a new level. In each iteration of this method, a new level is added to the hierarchy. As we go higher in the hierarchy, we add less number of patterns, which are more general than the patterns in the lower levels. This structure gives us the flexibility to choose whatever level of the hierarchy as the desired set of patterns.

Hybrid Pattern Recognition: When we explain our sequential pattern recognition, we assume that the logs/patterns inside a cluster are very close together. In order to generate the hierarchy of patterns, we start from the leaf level which has the most specific patterns, and the clusters are also dense. As we go up in the hierarchy and merge patterns, the clusters become less dense. Since the *MaxDist* parameter has been relaxed, we allow patterns with larger distances to group together. This creates a chance of being incorrect at the higher levels of the hierarchy if we use the sequential pattern recognition algorithm. Fortunately, the number of patterns inside each cluster at the higher levels of the hierarchy is much less than the lower levels, and we can use a selective merge order, instead of the sequential order, found by the classic UPGMA (Unweighted Pair Group Method with Arithmetic Mean) method. UPGMA, which is simply the hierarchical clustering with average linkage [72], is optimal when the clusters are spherical in shape in the high di-

mensional space, and we conjecture that this is asymptotically true for log patterns when clusters contain large number of messages.

Our final pattern recognition algorithm is a hybrid of UPGMA, the sequential recognition and the map-reduce implementation. Algorithm 3 shows how We pick the best choice for each cluster of logs. th_1 and th_2 are the thresholds for density and the number of patterns in a cluster respectively.

Algorithm 3 HybridPatternRecognition

Input: A cluster of logs ($Logs$)
Output: A pattern
if $density(Logs) \leq th_1$ **then**
 $pattern \leftarrow UPGMA(Logs)$
else
 if $Size(Logs) \leq th_2$ **then**
 $pattern \leftarrow SequentialPatternGeneration(Logs)$
 else
 $pattern \leftarrow MapReducePatternGeneration(Logs)$
return $pattern$

Cost of a Level: Given a hierarchy of patterns for a set of logs, the user may be interested in a level with specific properties. Some users may prefer to get the minimum number of patterns while the others may be interested to get very specific patterns and not care about the number of patterns. There are many different criteria one can say a level is satisfying or not. We introduce an intuitive cost function to pick the best descriptive level of the hierarchy. We also propose a cost function that suits our datasets. This cost function can easily serve as a general template to calculate the cost of a level of the hierarchy.

$$Cost = \sum_{i=1}^{\text{\# of clusters}} Size_i \times (a_1 WC_i + a_2 Var_i + a_3 FV_i) \quad (2.4)$$

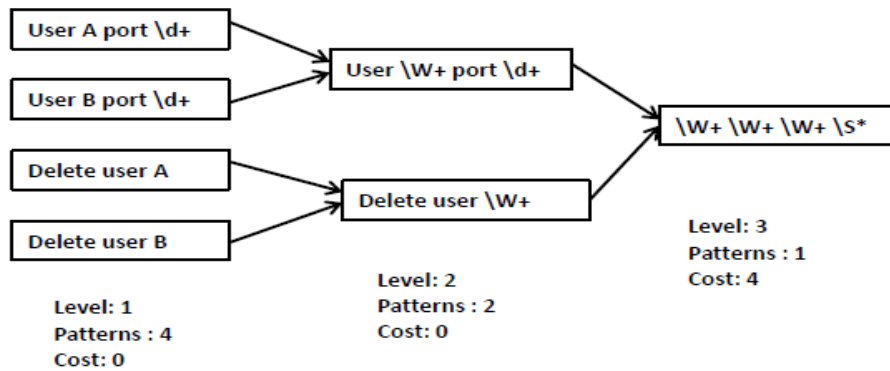


Figure 2.5: Pattern selection illustration with $a_1 = 1$, $a_2 = 0$, and $a_3 = 0$ in the cost function.

where $Size_i$ is the number of logs in cluster i and WC_i , Var_i and FV_i are the number of wildcards, variable fields and fixed value fields in the pattern of cluster i respectively. a_1 , a_2 and a_3 are tunable parameters that can be set in such a way that satisfies user's requirements.

If a user has no preference, we can set $a_1 = 1$, $a_2 = 0$, and $a_3 = 0$ in the cost function, and select the level having no wildcards with minimum number of patterns as the final set of patterns. For example, in Figure 2.5 we find that Level 2 generates two patterns with no wildcards, so we select these two patterns from Level 2 as the final set of patterns. In our experiments, we assume that a user will not provide any preferences. A user can also provide preferences by specifying the maximum number of expected patterns. For example, a user may want to generate at most 4 patterns. In this case, we select two patterns from the Level 2 in Figure 2.5 because it will generate minimum number of wildcards while not exceeding the user given maximum of 4 patterns.

2.4 Evaluation

We describe the baseline method that we compare LogMine against first. We then discuss our datasets and provide detailed experimental results.

2.4.1 HLAer: The Baseline

We pick the method *HLAer* [59] as a baseline algorithm, that is similar to our method in being unsupervised and supporting heterogeneous logs. *HLAer* finds very good sets of patterns, if not the optimal patterns, under reasonable assumptions. *HLAer* uses a highly accurate and robust clustering algorithm, OPTICS (Ordering Points To Identify the Clustering Structure), and the average linkage technique (UPGMA) instead of sequential merging for pattern recognition. We describe these two modules next.

Clustering using OPTICS: OPTICS (Ordering Points To Identify the Clustering Structure) is a famous hierarchical density-based clustering algorithm [25] used in *HLAer*. In OPTICS, a priority queue of the objects (e.g. using an indexed heap) is generated [82]. The priority queue can be used to cluster the objects at many different levels without redoing the bulk of the computation. OPTICS has two parameters: ϵ and *MinPts*. It ensures that the final clusters have at least *MinPts* objects, and the maximum distance between any two objects in a cluster is less than or equal to ϵ .

OPTICS is an expensive clustering algorithm for large datasets because it needs to calculate the *MinPts*-nearest neighbors for each object, which requires $O(n^2)$ pair-wise distance calculations for n objects. Typical improvement strategies include parallel computation and indexing techniques. However, these techniques become invalid in the iterative framework and for near-online batch processing. One may also think of pre-computing the pairwise distances, which requires loading all the computed distances in memory in $O(n^2)$ space for random accesses during clustering. Irrespective of non-scalability, OPTICS is a good baseline to compare accuracy with. The algorithm can find clusters of arbitrary shapes and densities in the high dimensional space. The algorithm is easy to reconfigure without recalculating the pair-wise distances.

Log pattern recognition via UPGMA: Once the *HLAer* finds all the clusters, it needs to find a pattern for each cluster that covers all the log entries in it. *HLAer* considers

Table 2.1: A summary of all datasets.

Dataset	# Logs	# Fields	Availability
D1	2,000	65	proprietary
D2	8,028	200	proprietary
D3	10,000,000	90	proprietary
D4	10,000	37	public
D5	10,000	45	public
D6	10,000	25	public

a log entry as a sequence of fields. It finds the best way to align multiple log entries together based on a cost function. After alignment, the algorithm merges each field by selecting a representative field to output. Unweighted Pair Group Method with Arithmetic Mean (UPGMA) is one of the most commonly used multiple sequence alignment (MSA) algorithms. It is a simple bottom-up hierarchical clustering algorithm [72] which can produce a tree of input objects on the basis of their pairwise similarities. *HLAer* runs UPGMA on the set of logs and finds the best order of merging the log entries. The *Merge* function that *HLAer* uses is identical to the one in Section 2.3.3.

If there are n log entries with l fields (on average), the time complexity of running UPGMA is $O(n^2l^2)$ which takes 300 years for 10 million logs. However, UPGMA can be used for smaller number of log entries, and generate valuable ground truth to evaluate the performance of our pattern recognition algorithms.

2.4.2 Datasets

For evaluation, we use six different datasets as summarized in Table 2.1:

- **D1** and **D2** are small proprietary datasets.
- **D3** is an industrial proprietary dataset of size 10,000,000 generated by an application during 30 days. The size of D3 on disk is 10 GB.

- **D4** and **D5** are random log entries from two traces which contain a day’s worth of all HTTP requests to the EPA webserver located at Research Triangle Park, North Carolina [7], and the San Diego Supercomputer Center in San Diego, California [16] respectively.
- **D6** is a synthetic dataset generated from 10 pre-defined log patterns. We fix the type of each field in the patterns, and generate random values for that field. This dataset is used as a ground truth to evaluate the accuracy of our method.

We set $MaxDist = 0.01$ and $\alpha = 1.3$ in all the experiments unless otherwise stated. Since *HLAer* takes ϵ and *MinPts* as the input parameters, an expert set them for each dataset. As *HLAer* is a single CPU algorithm, for fair comparison, we run the sequential version of our algorithm on a single machine for all experiments unless otherwise stated.

2.4.3 Accuracy

Since our algorithm is made up of two main components, we test the accuracy of each component separately.

Accuracy of Clustering: We use the clusters generated by OPTICS as a baseline. We define an *agreement* metric to calculate how close the output of our fast clustering algorithm is to the output of OPTICS. Given a set of n log entries $S = \{l_1, l_2, \dots, l_n\}$, we run OPTICS to get the set of clusters $X = \{X_1, X_2, \dots, X_r\}$ and we run our clustering algorithm to get the set of clusters $Y = \{Y_1, Y_2, \dots, Y_s\}$. The *agreement* score is $\frac{a}{b}$, where a is the number of pairs of logs in S that are in the same set in Y and in the same set in X , and b is the number of pairs of logs in S that are in the same set in Y . This metric takes a value between 0 (the worst) and 1 (the best). Note that splitting a cluster of OPTICS into multiple clusters does not harm us because it leads to more accurate patterns and we can merge the sub-clusters in higher levels of the hierarchy. On the other hand, having a

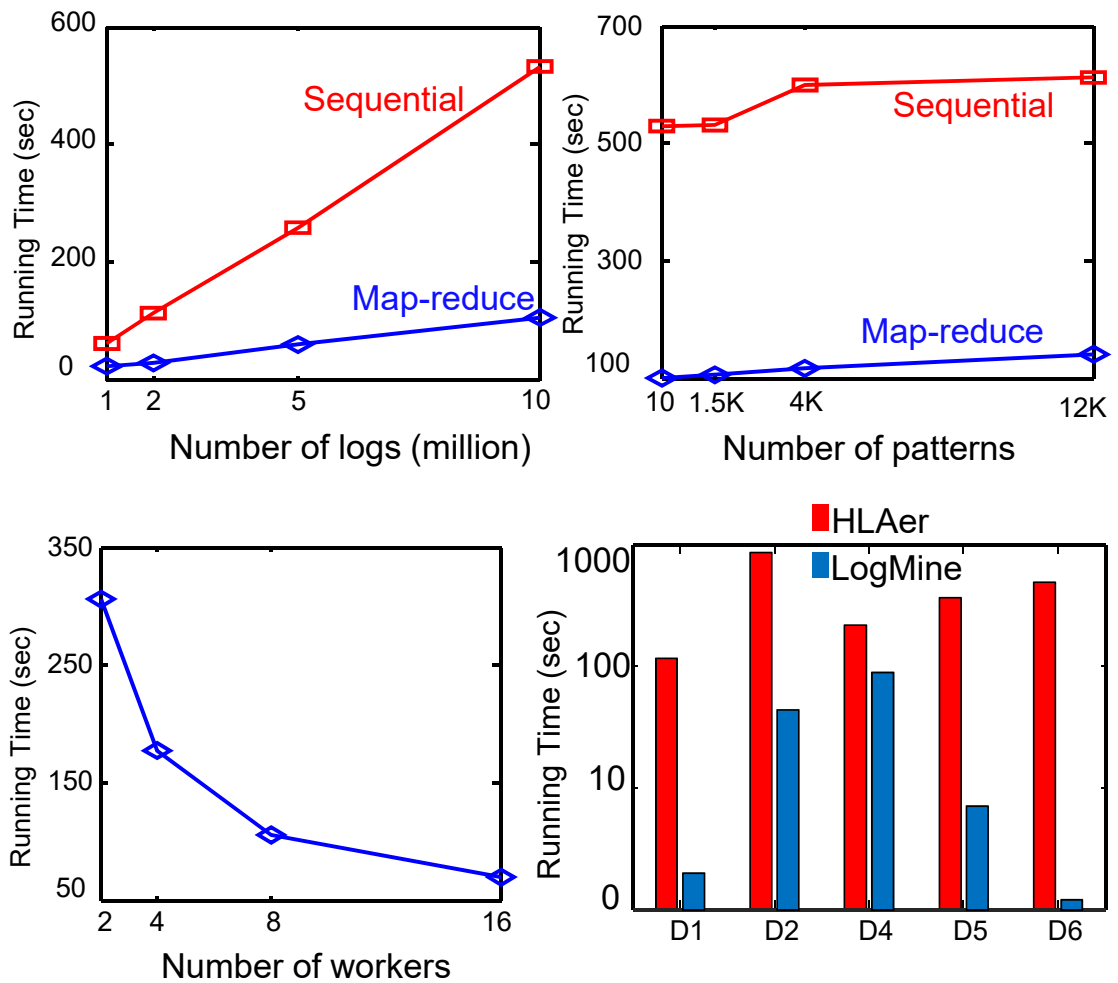


Figure 2.6: Comparison of running times of the sequential and the map-reduce clustering. (right) Comparing the running time of LogMine with *HLAer*. The y-axis is in log scale.

cluster which has log entries from multiple OPTICS's clusters can generate a meaningless pattern.

As shown in Table 2.2, in all datasets except D2, we capture the OPTICS's clusters. The problem with D2 is that the log entries do not have a clear underlying structure. Logs in D2 have many strings and commas, and they are very similar to each other. In addition, OPTICS throws away 38% of the entries as outliers because they do not fall in a cluster with at least MinPts entries. Therefore, a set of separable clusters may not exist in this

dataset failing both LogMine and OPTICS.

Accuracy of Pattern Recognition: As discussed before, UPGMA finds the best order to merge the log entries, and it produces the best possible pattern for a given cluster. We use the results of UPGMA as a ground truth to evaluate the accuracy of our pattern recognition algorithm. We cluster each dataset by both OPTICS and our clustering algorithm, and then give each cluster to both UPGMA and our pattern recognition algorithm to produce one pattern. We compare the patterns generated by the two algorithms, field by field. The accuracy of a given pattern compared to the ground truth is simply the number of matched fields over the number of all fields. The accuracy of pattern recognition for each dataset is calculated as below.

$$\text{Total Accuracy} = \frac{\sum_{i=1}^{\# \text{ of clusters}} (Acc_i \times Size_i)}{\sum_{i=1}^{\# \text{ of clusters}} Size_i} \quad (2.5)$$

where Acc_i is the accuracy of pattern recognition on cluster i and $Size_i$ is the number of log entries in cluster i . As Table 2.2 shows we can get almost same patterns as UPGMA except in D2. Since the quality of clustering on D2 is low, the logs inside each cluster are not very similar, and the order of merging can change the structure of the final pattern. The fact that our patterns for D2 are 73% similar to UPGMA patterns does not mean that ours are not accurate, because the patterns generated by UPGMA are also low quality. All the other results support the fact that the order of merging has no effect on the final pattern in a cluster with similar logs.

2.4.4 Memory Usage

HLAer calculates all the logs pairwise distances and use them while running OPTICS. This needs either $O(n^2)$ memory space or multiple disk accesses in case all the distances are stored back in the disk for n logs. Conversely, LogMine is very memory efficient with a space complexity of $O(\text{number of clusters})$ in sequential fashion. We run our sequential implementation and measure the amount of memory used by both *HLAer* and LogMine.

Table 2.2: The accuracy of pattern recognition and the agreement score for different datasets. We do not report these measurements on D3 because HLAer cannot handle it.

	Accuracy	Agreement Score	HLAer Memory (MB)	LogMine Memory (MB)
D1	100%	86%	32	2
D2	73%	48%	520	11
D4	96%	95%	801	15
D5	98%	100%	802	14
D6	100%	100%	795	13

Results are shown in Table 2.2.

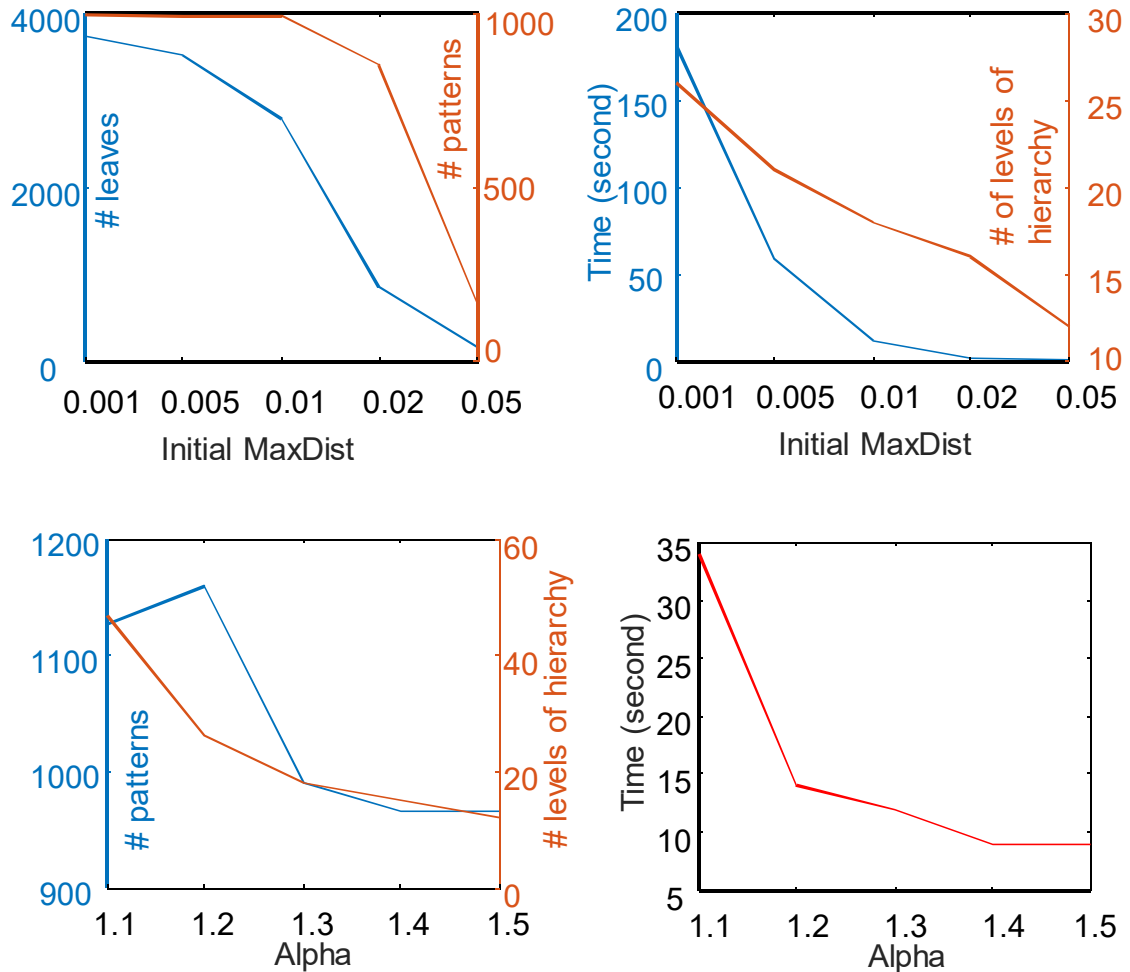
2.4.5 Running Time

HLAer has maximum processing capacity of 10,000 log entries because of the quadratic memory requirement. For the rest of the datasets, results are shown in Figure 2.6(right). **LogMine is up to $500\times$ faster than *HLAer*.** It takes 1,524 seconds for LogMine to cluster the logs, and find all the patterns in dataset D3.

It is worth to mention that LogMine has an advantage over *HLAer* in terms of running time of pattern recognition. Pattern recognition component of LogMine has a fixed running time for datasets of the same size, because it just scans the data once no matter how many cluster exists in the dataset. In contrast, *HLAer* depends on domain and data properties

2.4.6 Map-Reduce vs. Sequential

We discussed the way we find the dense clusters in Section 2.3.2 both in sequential and map-reduce fashion. In this experiment we compare them. We generate synthetic data by changing number of log entries (10 million default) and number of patterns (1500 default). We change the number of map-reduce workers (8 default) to understand scalability. Each

Figure 2.7: Sensitivity of the algorithm to the parameters $MaxDist_0$ and α .

worker has 1 GB of memory and a single-core CPU.

As shown in Figure 2.6(left), the execution time of the map-reduce implementation grows slowly compared to the growth of the sequential implementation. Map-reduce implementation reaches up to $5\times$ speed-up by using 8 workers compared to the sequential implementation. Note that we have a fixed number of patterns in this experiment. Our map-reduce implementation can handle millions of logs in few minutes, because the number of patterns does not grow at the same rate as the number of logs grows in real world applications. Figure 2.6(second-left) shows that with increasing number of patterns, the

growth in the execution time of both sequential and map-reduce implementation consistently grows. In Figure 2.6(second-right), we show that doubling the number of workers reduces the running time by 40%. The reason is that as we add more workers, the algorithm needs to perform more merges (see Algorithm 1), and this adds more overhead to the algorithm.

2.4.7 Parameter Sensitivity

We have two parameters in our algorithm, and both of them are used in the clustering phase. The first one is the $MaxDist_0$ which is the value we use to run the clustering algorithm at the lowest level of the hierarchy. After we find the leaves, we relax the clustering algorithm condition by increasing the $MaxDist$ by a factor of α . We run an experiment 10 times on a dataset of 10,000 log entries picked randomly from dataset D3, and report the average of different measurements change by increasing $MaxDist_0$ (0.01 default) and α (1.3 default). Results are shown in Figure 2.7. We make the following observations:

- As we increase $MaxDist_0$ we find fewer number of leaves in shorter time. Since the leaves are the basis of our hierarchy, we don't want to lose many of them due to a large $MaxDist_0$. On the other hand, small $MaxDist_0$ usually (not always) yields to longer running time. Although the best value for $MaxDist_0$ is to some extent depends on the dataset, we recommend to set it to 0.01.
- As the $MaxDist_0$ grows, we may end up extracting fewer number of patterns, but the plot shows that the algorithm can capture almost the same set of patterns even if we change $MaxDist_0$ in a wide range [0.001,0.02]. Thus, our algorithm is not very sensitive to this parameter in terms of the final set of patterns.
- Obviously if we pick smaller values for $MaxDist_0$ and α , the final hierarchy will have more levels and we have more options to choose a satisfactory level in the

hierarchy. However, it takes longer to produce such a hierarchy.

- Number of patterns does not change drastically with changes in α . We vary this parameter from 1.1 to 1.5 and the number of final patterns stays within the range [966,1127]. We find $\alpha = 1.3$ is the best performing value.

2.5 Case Study

We use LogMine to analyze logs collected from the OpenStack framework, which an open-source platform for cloud computing [12]. We have collected logs from the execution of `nova-boot` commands to demonstrate how LogMine can be used to build a log analytics solution. LogMine enable us to reveal various insights on logs that helps us to quickly diagnose the cause of failures.

Dataset: We have collected logs generated by 200 successful execution of the `nova-boot` commands. This is our training dataset. Using LogMine we have found 28 patterns which can correctly identify all training logs. These patterns serve as a basis for analyzing OpenStack logs. Next, we have collected logs from six failed executions (i.e., abnormal) of `nova-boot` commands. This is our testing dataset.

Detecting New Logs: To identify the cause of a failure, we need to detect logs which are not seen during the successful (i.e., normal) executions of the `nova-boot` commands. We use the 28 patterns generated by LogMine to detect those unseen logs. If a log in the testing dataset does not match with any of the 28 patterns, then we conclude that it is a new log. Using LogMine we have correctly identify those new logs, and report them to the system administrators for further analysis. Typically, administrators run adhoc keyword-based search on these new logs using their domain knowledge. In this case, searching few keywords, they have identified a subset of new logs, which help them to quickly localize the cause of the failed executions.

Detecting Logs with New Content: To analyze a failure, we are interested to find out whether or not there is any content-wise anomaly among the failure logs. To this end, we have structured 28 patterns generated by LogMine into various fields, and built a content-profile map for each field using the training dataset. During testing, if an incoming log matches with any of the 28 patterns, we identify its fields from the content. Now, if the content of any field value is not present in our training content-profile map, then we report corresponding log to the system administrators for detailed analysis. Using this content analysis, they have correctly identified new contents in the testing logs, which help them to diagnose failure scenarios quickly.

Detecting Logs with Abnormal Execution Sequence: The execution of OpenStack could result abnormal log pattern sequence if any failure happens. Therefore, we can detect system failure by discovery of anomalous log sequences. In order to achieve this functionality, we built log sequence order for any pair of 28 patterns and modeled their statistics such as the maximal elapse time, maximal concurrency of log pattern during training stage. During testing, we detect if the incoming log violates any of the following rules: log sequence reversal, exceeding the maximal elapse time or concurrency number, or missing the matching log for the pair. System administrators find these violations very useful to debug failures.

Detecting Log Rate Fluctuations: In order to analyze logs, it is helpful to find out whether or not there is any fluctuation in the log rates compared to the normal working scenarios. To detect fluctuations, we keep track of the range (i.e., minimum and maximum counts) that we have observed in a fixed interval of the training dataset for all 28 patterns generated by LogMine. During testing, if the matched logs count of any pattern falls out its training range in an interval, we report corresponding time range and pattern information to the system administrators for the further analysis, and they find it very useful to diagnose failure scenarios.

2.6 Conclusion

We have proposed an end-to-end framework, LogMine, to identify patterns in massive heterogeneous logs. LogMine is the first such framework that is ① unsupervised, ② scalable and ③ robust to heterogeneity. LogMine can process millions of logs in a matter of seconds on a distributed platform. It is a one-pass framework with very low memory footprint, which is useful to scale the framework up to hundreds of millions of logs.

Chapter 3

Anomaly detection in social media

3.1 Introduction

Social media sites, such as Twitter, Pinterest, Tumblr and Instagram allow people to broadcast messages and content (URLs, images, videos) publicly to their followers. Many of these sites allow users to change their homepage URLs by changing their account names. Users may need to change their URLs for many reasons, such as marriage, re-branding, business acquisition and closing, and so on. Such events are relatively rare for any human or business user in social media sites. However, we observe abnormally high URL change frequencies for a subset of Twitter users.

For example, we identify a user changing its URL 283 times in 78 days, equivalent to roughly one change every six hours. Some of the URLs, released and claimed in the same day, are shown in Figure 3.1. We identify an even more abnormal scenario where a URL, `twitter.com/MalumaOficial`, belonged to ten users in three months. Each user *handed over* the URL to another user collaboratively. In Figure 3.2, we show the sequence of handovers where nodes are user accounts and an arrow represents the direction of a handover. Such abnormal URL *handovers* are highly unlikely to be performed by a group

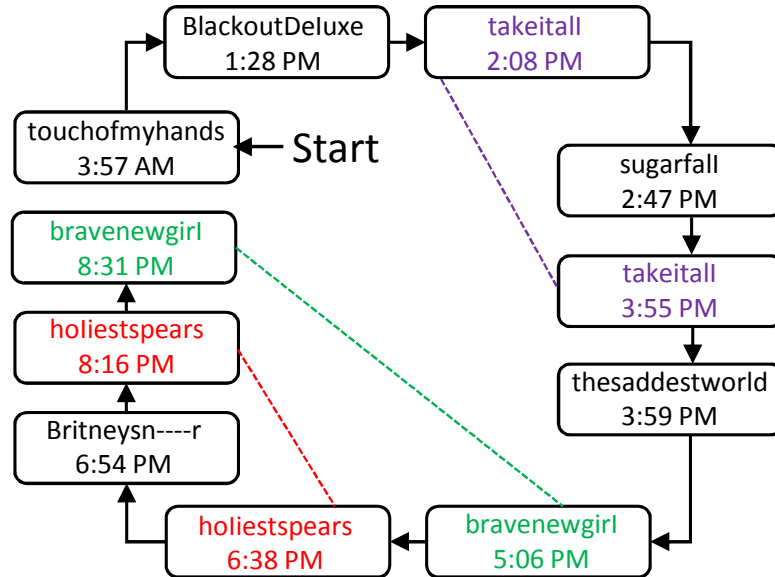


Figure 3.1: A user (Twitter id: 2664619086) with ten URL changes on 7 November 2015. Some URLs are used more than once (dotted connections), which form a loop of URLs.

of normal users, and most likely are generated by automated bots. As we see more such examples in Twitter, we focus only on Twitter bots in this work. Our work is generalizable to other social media sites with streaming data sources.

There have been dozens of papers on mining Twitter data [52][47] [74][28][62]. However, URL changes have not been studied with due diligence. An estimated 8.5% accounts in Twitter are bot accounts [73]. Our work shows that bot accounts carry out automated URL changes on a regular basis. URL changes waste resources on Twitter, create many broken URLs, and mislead Twitter users to spam account pages. These negative consequences motivate this work.

In this work, we investigate such abnormal URL changes and handovers to discover *why* and *how* users make such changes. We develop a parallel algorithm using the map-reduce framework to identify URL changes in streaming data. Our algorithm is incremental and scalable to support social media similar to Twitter in size and traffic. We create a set of 231K URL changes in Twitter over a period of three months (10/15-01/16). We perform temporal, textual, and graph-based analyses on this data and discover several interesting

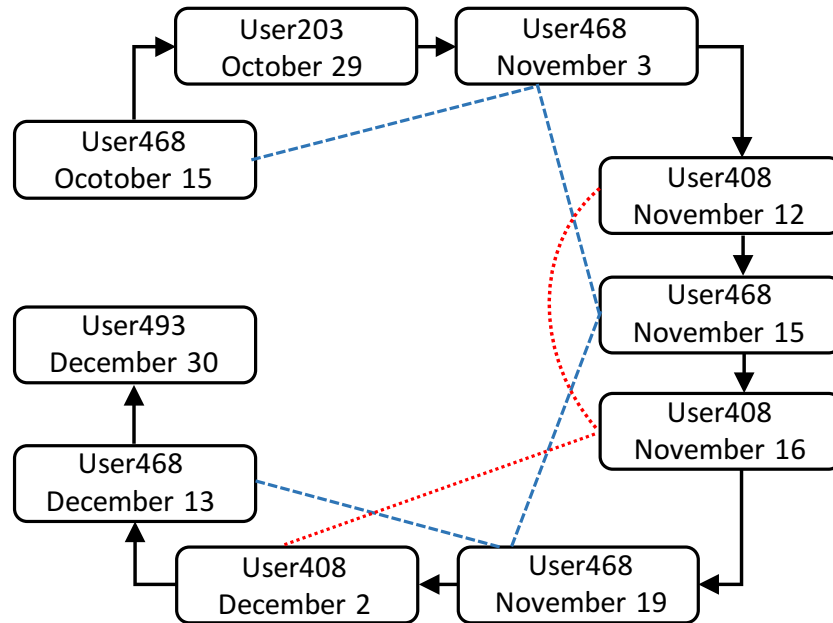


Figure 3.2: The URL `paradisecameron` was handed over among four users nine times. User 468 appears in the handover chain exactly every other time. The dashed lines connecting the same users show *loops* in this chain.

anomalies about URL changes. Our findings are summarized below.

- Both URL changes and URL handovers are atomic operations.
- URLs that are handed over are more frequently mentioned by other users.
- URL handovers are associated with changes in content after the handover.
- URL handovers can be temporally correlated.
- URL changes are done in an organized and collaborative way by large groups of users.

The rest of this chapter is organized as follows. We begin with a background section that provides examples of various types of URL changes and handovers. We describe our algorithm to discover URL changes and handovers in Section 3. We provide association analysis with temporal, textual and graph-based features in Section 4. We investigate *why* and *how* frequent handovers are performed in Section 5. We discuss scalability experiments in Section 6, related work in Section 7, and conclusion in Section 8.

3.2 Background

We start with sufficient background information so readers are more familiar with URL changes and handovers.

URL Changes: Imagine a Twitter user with the name `tom_hanks`. The URL to the profile page of this user would be `twitter.com/tom_hanks`. If this user changes the screen name to `thanks`, the URL of its profile page will change to `twitter.com/thanks`. Such a change in the URL does not affect the social connections of `tom_hanks` in the Twitter network. All of the followers and followings of the account before and after the change remain the same. However, the URL change invalidates the old URL, which will no longer be accessible from other places on the Internet. URL changes also invalidate all of the old *mentions*¹ within Twitter, since mentions are the short form of URLs. In some social media, such as Pinterest, the old URL still functions because the site automatically redirects visitors to the new URL unless the old URL is taken by some other account.

URL Handovers: A URL handover consists of two URL changes, in which one user releases a URL and another user claims that URL. Let us consider the example in Figure 3.3 to describe URL handovers in reality. A user (`user1`) changes its screen name (URL) from `Tom` to `John`. The name `Tom` is then free on the network and can be claimed by any other user. If another user (`user2`) claims the name `Tom` by releasing its previous name `Bill`, a handover happens. We say the URL `twitter.com/Tom` has been handed over from `user1` to `user2`. Here the `user1` is the *from-account* and the `user2` is the *to-account*. We also define the *handover lag* as the time duration between `user1` releasing the URL `twitter.com/Tom` and the `user2` claiming it.

In sites like Pinterest, the old URLs are redirected to the new ones; therefore a single user gains no direct benefit from handing over a URL. Conversely in Twitter, where redi-

¹Twitter users can mention other users by using the '@' symbol which creates a link to the profile page of the mentioned user. For example `@thanks` is a link to the address `twitter.com/thanks`

rection is not automatic, normal users often create new accounts to keep the old URLs. These are handovers but are also a valid behavior. In such scenarios, one of the from or the to accounts should be inactive (e.g. no tweeting) after the handover. The chance that a handover happens randomly between two *unconnected* users is very low, and we identify a *suspicious handover* if either of the following statements are true for a handover.

- Both of the from-account and to-accounts continue posting after the handover.
- OR
- Both of the from-account and to-accounts have a history of activity before the handover.

In the remainder of the work, we will refer to a suspicious handover as simply a handover unless otherwise specified.

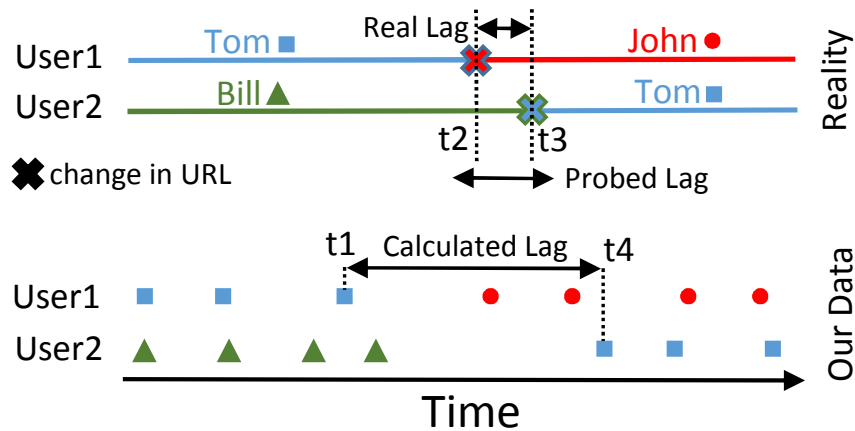


Figure 3.3: $User_1$ handed over the URL Tom to $User_2$. The release time is t_2 and the claim time is t_3 , and the real handover lag is $t_3 - t_2$. We calculate an upper bound, $t_4 - t_1$, for the handover lag based on the last tweet of $User_1$ at t_1 before the handover, and the first tweet from $User_2$ at t_4 after the handover.

3.2.1 Data Collection

We use the Twitter streaming API to collect data and produce a set of suspicious handovers. The Twitter streaming API caps the number of tweets sent to the client to a small fraction

of the total volume of Tweets at any given moment [22]. We have never exceeded 48 Hz in practice. Our data collection module receives the tweets which contain the timestamp of the tweet, the URL, the user ID, the follower count, and some other information about the author of the tweet. The Twitter API provides tweets that satisfy a given condition such as matching a given keyword, being about a topic, being tweeted from a geo-location, or being about a specific set of users. We consider each tweet as a singleton object with a set of predefined features including timestamp, user ID, URL, geo-location, number of followers, number of accounts the user is following, and tweet content. In order to detect handovers, we consider three relevant features: the timestamp, the user ID, and the URL. Although a user can change the URL of the account, the user ID is fixed for an account throughout the lifetime of the account.

We use different keyword filters to collect tweets for a week. We sort users based on their number of tweets and pick the top 40,000 as the seed for the rest of the data collection from the Twitter streaming API. We make the data collection process parallel on eight computers, each of which listens to 5,000 users continuously. This parallelization maximizes the number of tweets that can be collected from the streaming source.

We have started collecting data from Twitter on 15 October 2015 and continued until 31 December 2015. We have collected 130 million tweets with 5.7 million unique users² and 6 million unique URLs.

3.2.2 Complex Handovers

One URL change involves two URLs and one user account. One handover involves three URLs and two user accounts (Figure 3.3). However, URL changes and handovers can produce much more complex scenarios that are extremely unlikely to happen in a network that is built for independent social entities. A few complex scenarios are given below.

²Although our data collection seed contained 40,000 users, in total we collected tweets from 5.7 million different users.

- A user changes the URL multiple times and forms a *chain* of URLs. An example is shown in Figure 3.1. Long chains are more frequent in Twitter than what is expected in a network of independent users.
- Some chains of URLs create a *loop* when the user reclaims an old URL (i.e. $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$).
- A URL can be handed over in a chain from user A to user B, and then from user B to user C. This is a suspicious behavior because it shows that multiple accounts are interested in having the same URL. It gets more suspicious if each of these accounts own the URL for a short time.
- The handovers on a URL can also create a loop of users. A set of users can claim and release a URL and form a loop, which indicates that they either have a signaling mechanism to let each other know when the URL is free and ready to claim, or they are controlled by the same entity (Figure 3.2).

Although a single URL change may not be an abnormal behavior, the chance of all of the abnormal scenarios described above happening inadvertently is very low. We find a multitude of evidence showing that users are performing such changes and handovers automatically using computer programs.

3.3 Detecting URL Changes and Handovers

The phenomenon illustrated by Figure 3.1 clearly shows evidence of suspicious behavior and motivates further examination. However, detecting URL changes and handovers is non-trivial. We describe two approaches to detect URL changes and handovers: *active* and *data-driven*. Although we describe our technique focusing on Twitter API, the methods should work for any streaming API.

3.3.1 Active Detection (Probing)

The high-level idea of active detection is to probe Twitter to check the status of an account. We use the `users/show` function of the Twitter API which returns all the information for a given user ID or a user screen name (URL). We start with a list of user IDs and check their statuses in a round robin fashion. Whenever we see that the URL for a user is different from what we have seen the last time, we take this as a URL change. Upon detecting a change, we add the released URL to a URL-list that we track in order to see if any of the URLs are reclaimed by some user. Once we identify that a URL has been claimed, we detect a handover and register the *lag* between release and claim. We also add the to-user to the user-list to start tracking future changes or handovers.

Active detection has some serious limitations. First, we do not have a list of suspicious users to start the probing with. Second, Twitter provides very little bandwidth for such status checks (`users/show`) for users and URLs. We can query Twitter at most 180 times every 15 minutes. This is a much lower rate compared to the Twitter streaming API, which provides tweets at the rate up to 48Hz. If we start the probing with a set of less than 180 users, the probing technique allows us to observe a handover no later than 45 minutes after claiming. However, these two limitations restrict us from identifying a large number of handovers.

3.3.2 Data-driven Detection

Since we can just query Twitter to check statuses of users 180 times in each 15 minutes, we take an alternative approach where we identify handovers using tweets. We collect a massive set of tweets at the maximum bandwidth for 11 weeks, and are able to detect a large number of URL handovers based on the analysis of these tweets.

Key Idea of the Detection Process: Since Twitter does not provide an event flag representing a URL change, we devise an algorithm to identify handovers based on the last

tweet from the from-account and the first tweet from the to-account before and after the handover respectively. Figure 3.3 shows a toy example of handover detection using the streaming data provided by Twitter. Note that the handover lag can be calculated as the time between the last and the first tweets from the from-account and to-account, respectively.

Although our calculated lag is always larger than the real lag, we observe a wide range of lags in our dataset, including surprisingly short ones. For example, the URL `santacurIy` was being used by an account with zero followers for 6 days. The account then released the URL, and a more popular account with 5,000 followers took it over. The calculated lag in this scenario is only 110 seconds. In other words, 4 consecutive events happened just in 110 seconds: 1) The from-account tweets for the last time using a URL. 2) From-account releases the URL 3) The to-account claims the URL 4) The to-account tweets with the new URL.

Computationally, handover detection is very similar to the *group-by order-by* queries for relational databases. We require grouping the tweets from the same URL and sorting the tweets for the same URL based in order of timestamps. We need to compare successive pairs of tweets from the same URL to detect change in their user IDs. Each such change in user ID corresponds to a handover. The process is further complicated by the scale of the data. A single processor cannot manage millions of tweets in reasonable time, guiding us to develop parallel solutions. We discuss our algorithms to solve the problem in this section.

Our algorithm have detected a total of 13,831 URL handovers on 12,326 unique URLs handed over by 21,257 users in the 78 days of data collection. We also detect 231,800 users who changed their URL at least once in this time period.

Handover Detection: A handover can be detected when a URL is associated with more than one user accounts in a specific period of time. A naive approach to find handover is to use a hash map on all the URLs where the key is the URL and the value is the set of all user

IDs associated with that URL in any tweet. After grouping all of the tweets, we need to go through all of the hash keys and values and find those URLs that are associated with more than one user IDs. This naive approach is computationally expensive, as the intermediate data needs to be read multiple times, and the naive algorithm does not exploit parallelism. We retain the idea behind this naive approach and develop a completely parallel algorithm to detect handovers in a map-reduce fashion. We want not only to detect the handovers but also to know the time period each user account owned that URL in the past. This will help us to analyze the impact of handovers on an account during the time the account owned the URL.

Every map-reduce algorithm has two key components: a map function (mapper), and a reduce function (reducer). There can be other useful functions such as *filters* in a map-reduce framework. We discuss each of these components in this section. For clarity we define the input and the output of our map-reduce framework. The input is a set of tweets $T = \{tw_1, tw_2, \dots, tw_n\}$ and the output is a set of URLs $U = \{url_1, url_2, \dots, url_m\}$ where:

- $url_i = \{(user_1, t_1, t_2), (user_2, t_3, t_4), \dots, (user_k, t_{2k-1}, t_{2k})\}$
- $k \geq 2$
- $t_j \leq t_{j+1}, \forall j \ 1 \leq j \leq 2k - 1.$

Step1: Mapper The map function in our framework converts a tweet object to an object that can be used by the reducers. It produces a set of key-value tuples where the key is the URL of the tweet and the value is the user ID plus two timestamps. Initially both timestamps are equal to the tweet timestamp, but they will be converted to a start timestamp and an end timestamp in the next steps, which reflect the period of time in which the URL was associated with each account. In other words, initially:

$$mapper(tweet_i) = \langle url_i, \{(user_i, t_i, t_i)\} \rangle$$

Step2: Reducer The reduce function plays the key role in our map-reduce framework.

The map-reduce framework guarantees that all objects with the same URL will be reduced together and that they produce one last merged object. A merged object in our case is $\langle url_i, \{(user_a, t_1, t_1), (user_b, t_2, t_2), \dots, (user_z, t_k, t_k)\} \rangle$ where $t_i \leq t_{i+1}$. The reducer function takes two key-value tuples as the input and produces one merged tuple as the output. As mentioned earlier, the value part is made up of a set of user IDs, each of which has a starting and an end time. The reducer function takes these two lists and creates a sorted output based on the start times of each object³. Since all the lists have just one element at the beginning of the reduce task, they are trivially sorted. As the reducer combines them, the merged lists are also sorted. In other words, the input to the reducer function is two sorted list of length m and n , and it just takes $O(m + n)$ steps for the reducer to sort them by using the merge sort approach.

Step3: Mapping Values After the reducer produces a sorted list of user IDs with timestamps for each URL, we need to merge all the consecutive tweets with the same user ID to create a shorter list for each URL where there is a start and an end time for each user ID. For example, if the output of the reducer for a URL is

$$\{(A, 1, 1), (A, 4, 4), (B, 7, 7), (B, 9, 9), (A, 15, 15), (A, 20, 20)\}$$

then the output of running the map value function would be:

$$\{(A, 1, 4), (B, 7, 9), (A, 15, 20)\}$$

Step4: Filter We explained how we create the list of all users associated with the same URL in our map-reduce framework. However, we are not interested in detecting URLs that are only associated with one user ID. These are usually regular accounts and are not of interest, but our map value function will output them too. To filter out these URLs from the output of the map-reduce framework, we use a simple *filter* function. This function checks the length of the list of the users after step 3 and outputs only the lists that have

³At this stage of the algorithm, the start time and the end time of the objects are still the same

Algorithm 4 Reducer

Input: Two lists of sorted tweets with start and end times**Output:** Sorted list of tweets based on their start timestamps $i_1 \leftarrow 1$ $i_2 \leftarrow 1$ $L_{out} = \text{empty list}$ **while** $i_1 \leq |L_1|$ and $i_2 \leq |L_2|$ **do** **if** $L_1[i_1] \leq L_2[i_2]$ **then** $L_{out} \leftarrow \text{concat}(L_{out}, L_1[i_1])$ $i_1 \leftarrow i_1 + 1$ **else** $L_{out} \leftarrow \text{concat}(L_{out}, L_2[i_2])$ $i_2 \leftarrow i_2 + 1$ **while** $i_1 \leq |L_1|$ **do** $L_{out} \leftarrow \text{concat}(L_{out}, L_1[i_1])$ $i_1 \leftarrow i_1 + 1$ **while** $i_2 \leq |L_2|$ **do** $L_{out} \leftarrow \text{concat}(L_{out}, L_2[i_2])$ $i_2 \leftarrow i_2 + 1$ **return** L_{out}

more than one user ID.

Incremental Handover Detection: The map-reduce framework we discussed in this section is in batch mode. In other words, it gets a set of tweets as the input, and produces the set of all handovers. While running the map-reduce framework, if we store the intermediate data right before the *filter* function, we can use it to detect the handovers in the future in an incremental fashion. For example, if we collect tweets in November, detect all handovers from that month, and store the intermediate data on disk, once we have the

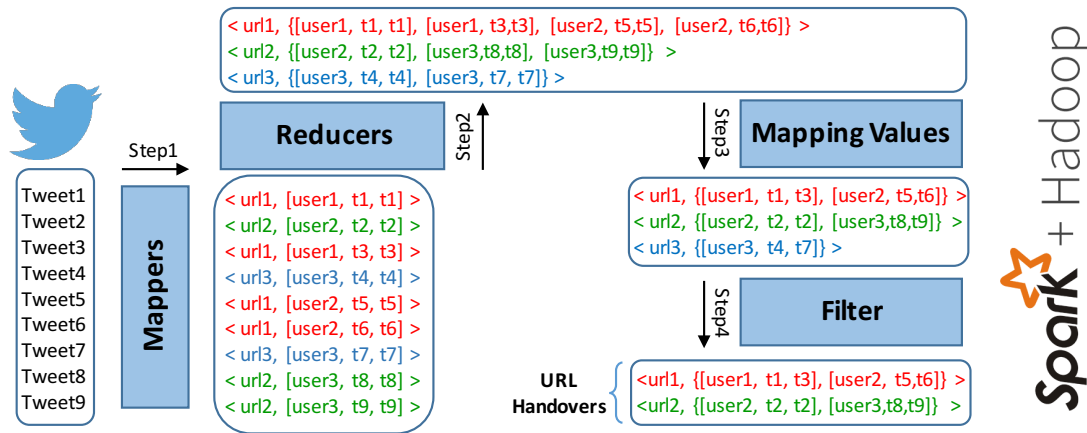


Figure 3.4: The process of detecting URL handovers in the Twitter network. 2 URL handovers are detected from 9 tweet objects in this example.

tweets from December we can use that intermediate data to find new handovers that have occurred from the beginning of November to the end of December. This processing will be faster than running batch mode on all of the November and December data. The reducer, the map value function, and the filter of the incremental version are exactly the same as before, but the mapper is faster due to using the map-reduce built-in functions to produce key-value pairs from the intermediate data, and also because the data gets smaller after step 3.

Limitations of Map-Reduce framework: We discussed how to find URL handovers by listening to the Twitter API and using the map-reduce framework. Detecting a URL change plays a key role in URL handover detection. We detect URL changes by looking at the metadata provided by Twitter for each tweet object. We can detect a URL change if we receive at least two tweets with different URLs from that user. Therefore, if the user does not have any tweet activity, or we do not receive tweets of the user due to an API limitation, we may miss some of the handovers. In addition, we may not be able to calculate a tight upper bound for URL handover for the same reasons. Although the probing technique addresses the latter issue, we may still miss some handovers due to limited access to the Twitter data.

Implementation using Spark: Apache Spark is a fast and general-purpose engine for large-scale data processing. It was introduced in 2013 [87]. We implement all of our map-reduce-based handover detection code using Java in Spark.

URL Change Detection: We have explained how to detect handovers in a given set of tweets. The other item of interest is URL changes, i.e. when a user changes its URL in Twitter. The same map-reduce framework can be used to detect URL changes for all users. We just need to modify the way we create key-value pairs. In this case the key is the user and the value is the associated URL with its timestamps. The final result we get for each user would be in the following format:

$\langle user_i, \{(url_1, t_1, t_2), (url_2, t_3, t_4), \dots, (url_k, t_{2k-1}, t_{2k})\} \rangle$ which shows $k - 1$ URL changes for this user.

3.4 Handover Analysis

In this section we analyze the handovers detected by our method to observe several suspicious behaviors related to the the user's temporal profile, tweet content, and the frequency of URL changes. We also discuss how multiple users can be connected through handovers. We finally analyze the handover lags to show that the handovers are automated.

3.4.1 Temporal Profile

We investigate questions related to the temporal profile of a user involved in a handover. We extract hourly time series of every user in every handover for 78 days. As mentioned in Section 3.2.1, each tweet object contains the timestamp of that tweet in millisecond resolution, and the number of followers of the user at that time. We construct hourly

activity time series of every user by aggregating the total number of activities the user performs in each hour. Note that Twitter does not guarantee to provide all of the tweets of a user; therefore we achieve a lower bound time series on user activity. As we shall see, such partial data is enough to reveal abusive behaviors on Twitter.

Similarly, we create the follower time series of a user which shows the changes in the popularity of that user. We receive follower information embedded in the tweets, yielding some unevenly spaced measurements of the follower counts of a user. We interpolate the in-between follower counts by the last received count with an assumption that follower counts change very slowly (particularly for non-popular and old accounts).

Activity Association: We first consider the distribution of handovers over 11 weeks. We only consider handovers that have less than a day of calculated lag. This ensures that the real lag is at most 24 hours, a reasonable value. In Figure 3.5(top) we show the frequency distribution of the hourly aggregates of handover counts over 1890 hours. We use the method in [77] and identify three sharp peaks pointing to weekly, daily and 12-hourly periodicity. Figure 3.5(bottom) shows an example activity sequence of a user with daily and weekly periodicity.

We investigate if the handovers are related to a change in activity patterns. We check if the average activity levels of a user in the 6-hour windows before and after a handover are significantly different. 91% of the times the difference is less than 1 tweet an hour. Therefore, we conclude there is no significant change in the activity level before and after the handovers. However, exceptions are possible. Figure 3.5(bottom) shows an example where the activity starts and stops with handovers.

Next, we consider the association between handovers and the activity level around them. We calculate the average activity per hour of every from-account in the 6-hour window just before release the URL, and the same of every to-account in the 6-hour window just after claiming the URL. We compare them with the average of activity-per-hour of the user, calculated over the duration of data collection. We identify a significant difference

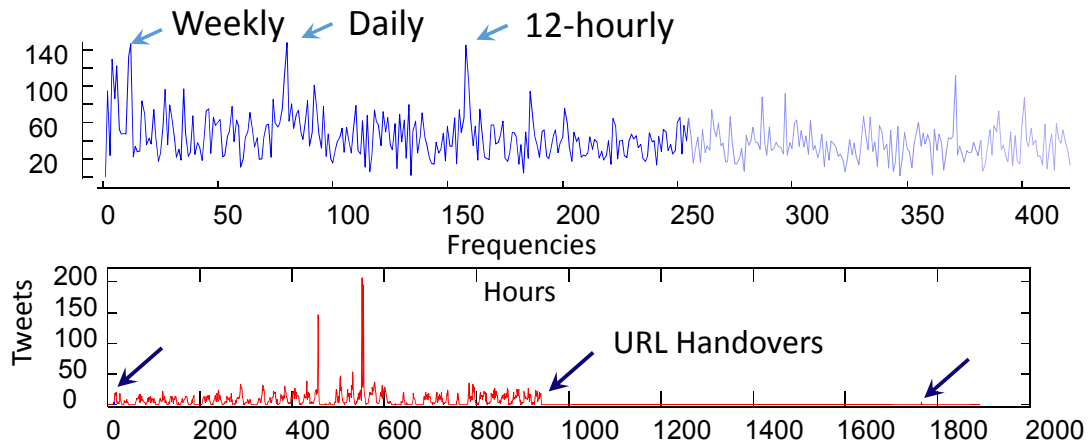


Figure 3.5: An example of a user with daily periodicity and a strong association with handover.

in activity level before and after release and claim. Quantitatively, 97.4% of the users are more active than usual when performing handover (Figure 3.6).

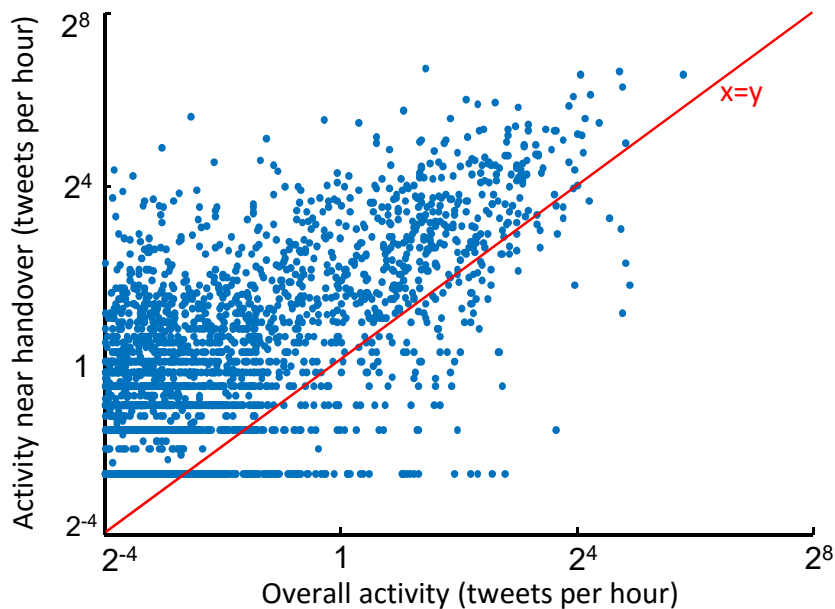


Figure 3.6: Comparing the normal activity of a user with its activity near the handovers. 97.4% of the users have higher activity-per-hour around handovers. Both x-axis and y-axis are in log scale.

Cross-user Association: We further consider cross-user associations in temporal profiles

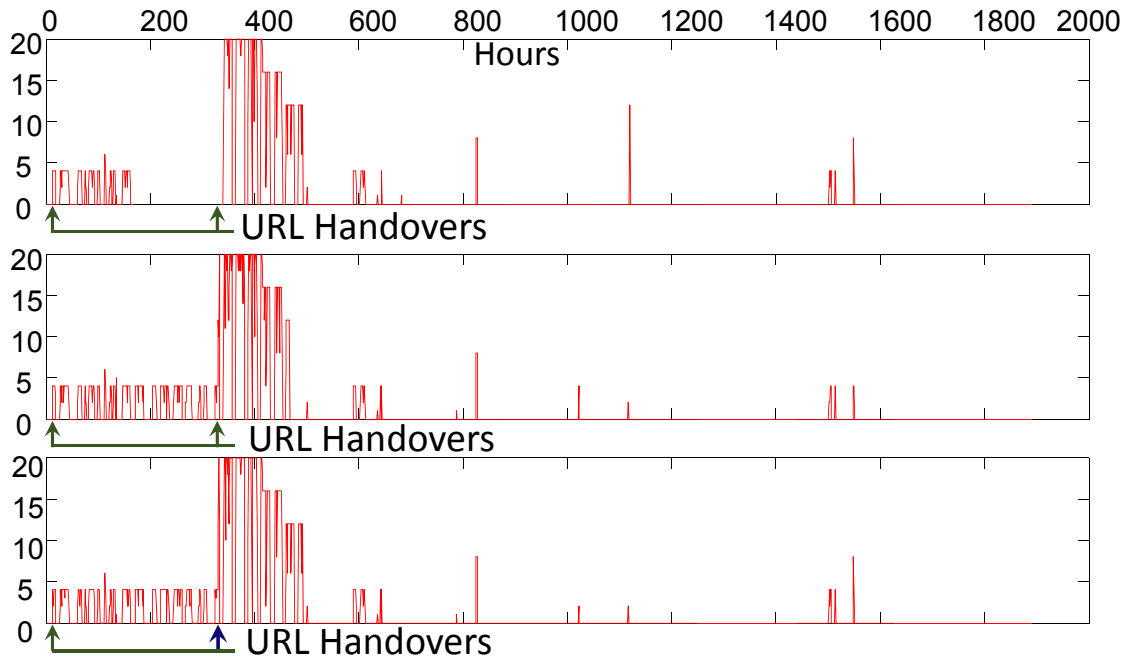


Figure 3.7: Three accounts with almost identical activity profile and correlated handovers. Handovers initiate change in activity patterns.

of handovers. We use standard time series motif discovery tools [53] to identify the most frequent activity time series. Note that the expected similarity in activity time series between two users for 11 weeks is almost zero. Interestingly, We identify a motif of three users who have almost identical activity patterns with an average correlation coefficient of 0.96. Furthermore, the accounts perform URL handovers within the same hour in the same manner (e.g. to-from-to). The motifs are shown in Figure 3.7. The URLs that were handed over by these accounts are all related to celebrities such as MacMiller, Rihanna, Drake, Megan Fox and Lil Wayne.

We consider the motif as a significant discovery because it reveals that offenders work in correlation, possibly using the same codebase, and that they hand over at the same time to swap or pass URLs that they do not want to lose. In the future, we will investigate how to scale handover detection in real time so we can track the interest areas of the offenders to take countermeasures.

3.4.2 Content Profile

Users tweet about various topics. The topic of a tweet can be determined by analyzing the keywords in it. We first remove all useless words like *is, are, the, to, from, RT*⁴, . . . , and then process the tweet content. We use the content of the tweets to determine the similarity between two tweets, and also two sets of tweets. We use the Jaccard similarity coefficient [81] as our metric. If the set of keywords in *tweet*₁ and *tweet*₂ are T_1 and T_2 respectively, the Jaccard similarity can be calculated by Equation 3.1.

$$Jaccard_sim(tweet1, tweet2) = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} \quad (3.1)$$

The similarity between two sets of tweets X and Y can be defined as the average (*avg_{sim}*) similarity of all pairs of tweets in them.

$$Sim(X, Y) = average\left(\sum_{i=1}^{|X|} \sum_{j=1}^{|Y|} Jaccard_sim(X_i, Y_j)\right) \quad (3.2)$$

We use this measure to calculate the similarity between two Twitter users, or between two different periods of time (before and after the handover) for the same user to profile content changes around handovers.

Content Association: We consider the content of tweets for a user before and after the URL change⁵ to see *if the content changes with the change in URL*. Let T_1 and T_2 be the sets of tweets of a user from its first and second URL respectively. We calculate the in-URL similarity as the weighted average of $Sim(T_1, T_1)$ and $Sim(T_2, T_2)$, and the across-URL similarity as $Sim(T_1, T_2)$. For example, Table 3.1 shows the tweets of a user with two different URLs: `zflexins` and `loveyorslf`. The user tweeted 98 times with

⁴The word “RT” appears at the beginning of all retweets and has nothing to do with the content.

⁵We specifically are interested in URL changes that were part of a handover.

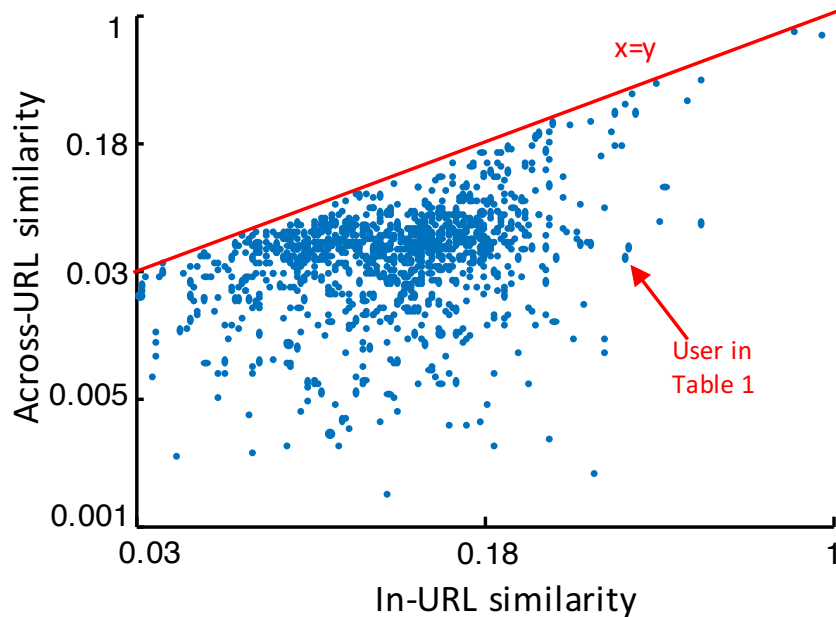


Figure 3.8: The plot shows how the content of the tweets changes with URL changes.

the first URL about Justin Bieber, and 94 times with the second URL about Harry Styles. These are two of the most popular celebrities in Twitter with millions of followers. There is a clear change in the topic of the tweets after the URL change. The average in-URL similarity for this user is 0.35 while the across-URL similarity is 0.03. It is humorous that the content of the first tweet after URL change is: `RIP zflexins`. Both of these URLs are now associated with some other accounts.

In order to check this hypothesis for other users, we select a random set of handover users that have exactly two URLs associated with them in our dataset. We filter out the users for which $|T_1| < 5$ or $|T_2| < 5$, and finally come up with 1,051 users. Figure 3.8 shows the comparison of in-URL with across-URL similarity for each of these users. We have perfectly 100% of the users with higher in-URL similarity than the across-URL similarity. It means that the overall topic of the tweets changes when a user changes its URL, especially if that URL change is a part of URL handover.

Table 3.1: The tweets from the same user with 2 URLs. The user change its URL from `zflexins` to `loveyorslf` on 11 December 2015. All the tweets of the left column are about Justin Bieber, and the ones in the right column are about Harry Styles (both are famous singers). The average in-URL similarity is 0.35, while the across-URL similarity is 0.03.

www.twitter.com/zflexins	www.twitter.com/loveyorslf
RT @justinbieber:UK! Tonight on @CapitalOfficial from 7pm 'Justin Bieber's Capital Album Party Replay'. Hear the tracks from #Purpose	harry styles coisa mais linda gente!!!
RT @JBCrewdotcom: Another photo of Justin Bieber with a fan at the M&G in Tokyo, Japan yesterday. (December 4) https://t.co/ofAYAjzP1M	harry s,tao precioso gente como vcs nao gostam dele????????? https://t.co/o0x2DG38JI
RT @JBCrewdotcom: Another video of Justin Bieber singing at a restaurant in Japan today. (December 5) https://t.co/jZqaMaezrO	vou tweetar video de harry stylesN
RT @favjarbara: interviewer: what do you think about justin bieber's relationships?bp: hahaha he's mine	harry w kendall eu to gRITANDO AQUI, OPSSS https://t.co/MURzVWnc0Q
RT @NME: Justin Bieber announces UK Arena tour dates for 2016 https://t.co/ECsRUqEPxk	@KendallJBrasil: 31/12- Mais fotos de Kendall e Harry Styles em St. Barts, Frana. https://t.co/CytM8Hixk

3.4.3 URL Change Analysis

In this experiment, we check if the frequency of URL changes (average number of URL changes per day) of a user has any relation with the probability of that user being involved in a URL handover, since we believe both a high number of URL changes and being involved in a handover are suspicious behaviors. There are 231,800 users in our dataset which changed their URL at least once during our data collection. Figure 3.9(left) shows the percentage of these users for different frequencies of URL changing. The probability of a user being involved in a handover given the frequency it has changed its URL is shown in Figure 3.9(right). The higher the change frequency, the larger the probability of performing handovers. The reason why we do not show users with more than 9 URL

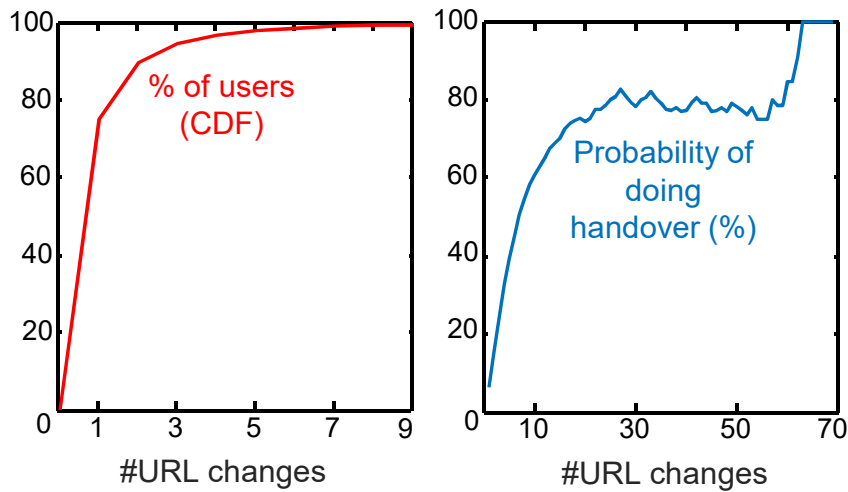


Figure 3.9: (left) Percentage of users (out of 231,800 users) based on the frequency of changing URL. (right) The probability of a user doing a URL handover given the URL change frequency. The probability reaches 1 for a user with frequency higher than 68 (almost one URL change every day).

change frequency on the left side is that they comprise less than 1% of our dataset. However, we can say almost all of this 1% have done a URL handover by looking at the right hand side of the figure.

3.4.4 Connectivity Profile

In addition to temporal and textual profiling, and URL change analysis, we look at the connectivity profiles of the users that perform handovers.

We create a bipartite graph where the left side is the set of all users and the right side is the set of all URLs, and a link between a user u_i and a URL v_j exists if u_i owned v_j at some point in our dataset, and the URL was used for a handover. A handover is defined as a subgraph with three nodes and two edges in which two nodes from the left side (users) have a link to the same node on the right side.

An example of this graph is shown in Figure 3.10. In this example, the URL V_1 has

been handed over by users U_1 and U_2 . The URL V_2 has also been handed over by users U_2 and U_3 . These two handovers together form a cluster of size 5 in which three users and two URLs are connected. On the other hand, the user U_4 has owned the URL V_3 the whole time which is a normal behavior.

We use the classic co-clustering approach to identify clusters in the user-URL bipartite graph [34]. Any balanced cluster with more than three members points to organized teamwork behavior between accounts. It is very unlikely that a large balanced cluster was created in this bipartite graph by accident.

We find a cluster of size 2,273 (1,205 users + 1,068 URLs) which has 2,399 edges. The average degree of each node in this cluster is 2.11. It is highly unlikely that such a cluster is formed randomly, and thus this cluster supports our original hypothesis that correlated and frequent handovers are signatures of automated accounts managed by the same entity. If we had more data, we could have identified more handovers, and the cluster could have been much larger. About 6% of the users that perform suspicious behavior are in this particular cluster, proving our suspicion correct beyond a doubt.

If we consider all of the clusters with more than three members (non-trivial handover clusters), they cover 31% of all users who have been involved in handovers. Although any URL handover is a suspicious behavior, this provides us with additional evidence of misbehavior from this 31%. We believe that the majority of the other 69% also belongs to a non-trivial cluster, but we are not able to catch them due to lack of data. As we show in the next section, social media sites are slow in suspending such offenders. We have detected thousands of automated spammers, even without the complete dataset, and yet Twitter has suspended only a fourth of them in six weeks.

These users who are doing URL handovers usually change their URLs more than once. Not all of their URLs are included in the discussed bipartite graph since we just add the URLs which have been handed over. If we include all of the URLs of the users who have done a handovers (even the URLs that have not been used in any handover so far)

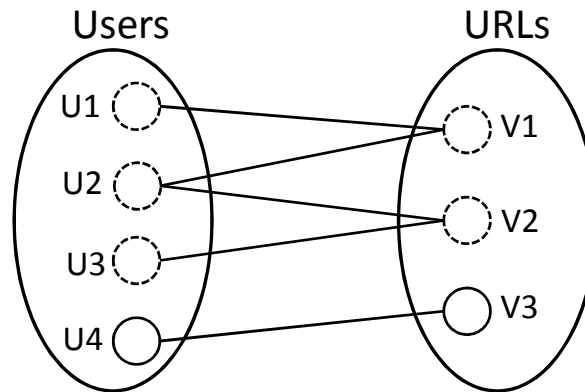


Figure 3.10: The bipartite graph that shows the connections between users and URLs. The dashed nodes form a connected component of size 5. User U_4 has not been involved in any handovers.

in our graph and re-cluster, the biggest cluster would have 1,205 users and 6,040 URLs. These newly added URLs are good candidates for our active probing technique since they belonged to a suspicious user at some point in the past.

3.4.5 Lag Profile

To examine whether these handovers are organized from a central source, we perform an analysis on handover time-lag. The real lag between releasing a URL and claiming it back is not detectable from the publicly available tweets. Our active probing tool, which is not scalable because of a capacity limit set by Twitter, estimates handover lag at most an hour longer than the real lag. We can only probe 180 users and/or URLs every 15 minutes. We start probing every day with a list of 100 users who we know have high numbers of URL changes (based on our dataset). We have done this experiment for 8 consecutive days and observed 210 handovers. Figure 3.11(left) shows the CDF of the percentage of handovers for different lags. The sharp increases at minutes 15, 30, and 45 are the result of the discontinuities in the probing algorithm caused by Twitter API limitations imposed on our algorithm. The approximately linear CDF illustrates the remarkable fact that handovers are instantaneous operations. We can verify this claim by simulating a set of instantaneous

handovers spread uniformly over time and applying our probing algorithm to calculate an estimated CDF. The estimated CDF is indeed a line and the slope of the line is very similar to what we have observed.

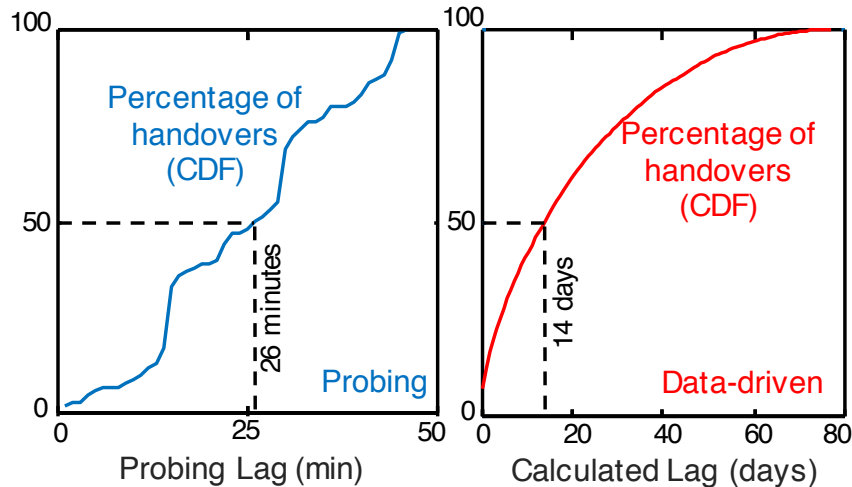


Figure 3.11: The distribution of handovers based on their lags. (left) The lags that are calculated using our probing technique. (right) The lags that are calculated using the data-driven approach. 50th percentiles are shown in both distributions.

This analysis formed the basis of our data-driven detection process. In the data-driven detection process, we can only detect a handover if the pair of accounts tweet something before and after the handover, and Twitter provides us the tweets. Under such stringent condition, the lags we calculate are weak upper bounds of the real lags.

We show the CDF of the handover lags detected by the data-driven technique in Figure 3.11(right). Although the data-driven process detects larger lags compared to the real lag, since we know from this analysis that the handovers are mostly instantaneous operations performed by automated programs, we trust that the handovers detected using the data-driven technique are highly suspicious. Also note that the lag for half of the handovers we find is less than 14 days. For higher lags, the probability of decreases. In other words, if a URL is not claimed after few days of releasing, it (probably) will not ever be claimed.

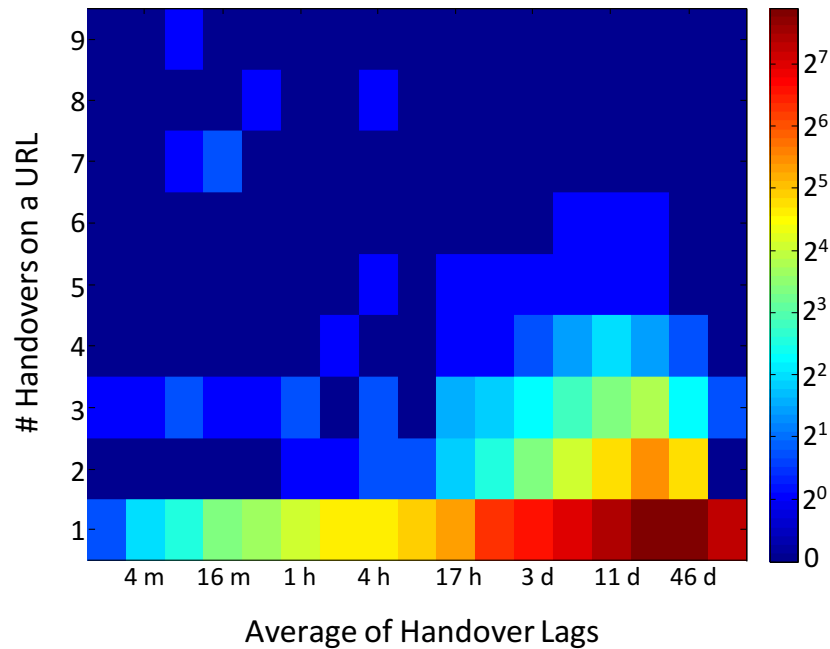


Figure 3.12: This figure shows the number of URLs for a given handover lag and a given number of handovers on that URL. The X axis is in log scale. As the number of handovers on a URL increases, the average of handover lags on that URL goes down drastically. We can see many different lags for URLs that are handed over just once.

3.4.6 Handover Lag Distribution

Earlier we showed the cumulative distributions of handover lags. As the number of times a URL is handed over between users increases, it gets more suspicious. We believe that it is not possible that a URL gets handed over multiple times incidentally. The other sign of a suspicious URL handover is having a short lag. In this experiment, we want to see if there is any relation between these two parameters: the number of times a URL is handed over and the average handover lag for all the handovers being done on that URL. We show a heat map in Figure 3.12. The X axis is in log scale. As it is shown, all the URLs that are handed over more than 7 times have an average lag of 1 hour or less, in some cases even less than 10 minutes. For example, the URL in the top row was handed over 9 times, and the average of all its 9 handover lags is 7 minutes. This clearly implies that this URL is valuable for its owner(s), and thus handovers are done quickly on this URL so as not

to lose it. On the other hand, some URLs that were handed over once or twice have an average lag of several days. In these cases either the owner was not worried about losing the URL or the calculated lag is much higher than the real lag due to lack of activity.

3.5 Why Handovers?

Such a magnitude of automated URL changes must have good reasons behind. In this section, we discuss association of handovers with potential benefits such as obtaining human followers and avoiding suspension, and thus attempt to answer the question *why handovers are so frequent?*.

3.5.1 Mentions and External URLs

Although URL changes do not impact the internal connectivity among users (who follows whom), it has a direct impact on URLs linked from external web pages. It also affects the links created by *mentions* within Twitter. For example, when user1 mentions user2 as @DavidW (whose screen name is *DavidW*) in a tweet, Twitter creates the URL `twitter.com/DavidW` and embeds it in the tweet content. If user2 hands over this URL to user3, the mention *DavidW* would point to user3's profile page. Thus, thousands of mentions within Twitter are being abused by URL changes.

Our hypothesis is that the miscreants change URLs frequently to fool users in visiting different pages every time they follow the same mention. The motivation is to increase the chance of getting a human visitor or follower in the process.

To test this hypothesis, we use the Twitter Advanced Search page in which one can search for the mentions of a certain screen name (i.e. URL). We count the number of mentions a URL receives in the first fifteen pages of the search result. Figure 3.13 shows the percentage of URLs based on the number of mentions for 1000 random URLs and 1000

handover URLs. The URLs that have been handed over have a higher number of mentions compared to random URLs. The average number of mentions for handover URLs is 80 compared to 22 for random URLs.

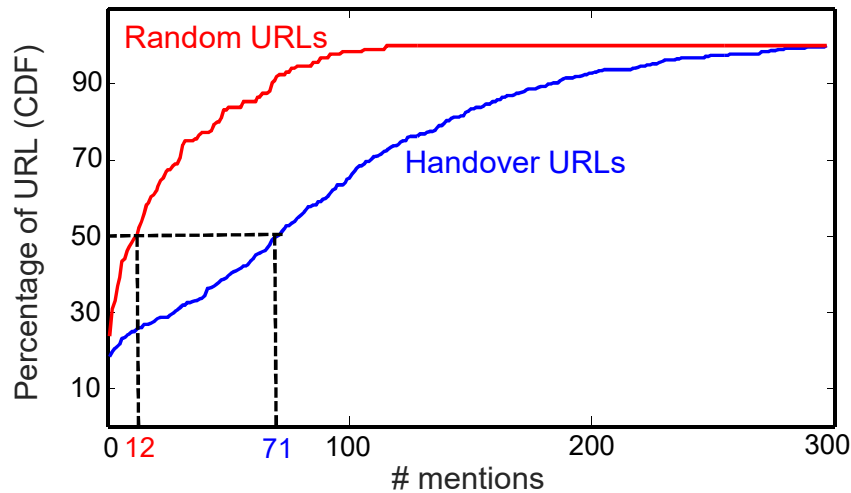


Figure 3.13: The percentage of URLs based on the number of mentions for random URLs and handover URLs.

3.5.2 Suspension

Twitter suspends accounts that violate its rules [20]. We have detected URL changes and handovers until 31 December 2015. In order to see whether or not doing the handover has any impact on the suspension of the involved users, we check the status of all handover users almost every week from 1 January 2016 to 8 February 8 2016. Each point in Figure 3.14 shows the percentage of the handover accounts being suspended by Twitter until that day. The interesting point is that although these users had done many URL handovers in our data collection period, just 0.6% of them were suspended by 1 January 2016. In Section 3.4.4, we mentioned that we have additional strong evidence of 31% of the handover users being suspicious, and still just 7% of these accounts are suspended by Twitter at the time of writing, while We had found these suspicious users weeks earlier.

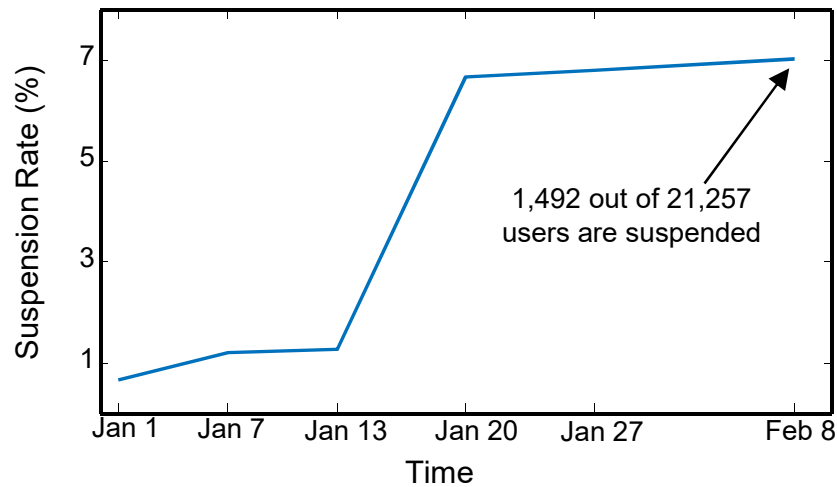


Figure 3.14: Twitter suspension rate of handover users.

3.6 Experiments

We provide detailed experimental results on execution time, distributed speedup, and parameter sensitivity. We use 16 worker nodes each of which has 1 GB memory for running the map-reduce code. All the other experiments are done on an Intel Corei7 machine running Ubuntu 14.04 with 32 GB of memory. We share the entire set of handovers in the supporting page of this project [19]. The supporting page also contains presentation slides, Excel sheets, and the source code to detect handovers.

3.6.1 Scalability

We present a map-reduce framework to detect all of the URL handovers and URL changes in a set of tweets. In order to analyze the running time of our algorithm, we design three experiments as follow:

Comparison with Naive approach

In order to calculate the speedup of our URL handover and URL change detection

method, we implement a single core version of our method in Python. We use a hash map to keep track of all the users and timestamps associated with each URL in our dataset. We compare the running time of our map-reduce code with this naive implementation in Table 3.2. The input to both codes is a set of 130 million tweets. The numbers reported in Table 3.2 include pre-processing too. We use 16 worker nodes to run the map-reduce code for this experiment.

Our Spark-based implementation is extremely fast compared to the naive implementation. Although we use a hash-map for URLs, there is no indexing mechanism to quickly find the hash bucket for a certain URL. In other words, the naive algorithm grows quadratically on number of URLs we have. In contrast, the Spark framework smartly distributes the data to virtually create an indexing structure on the URLs and thus, the algorithm grows much more slowly in Spark [87]. Therefore an estimated speedup from the naive implementation would be 16^2 for 16 workers. In practice, we achieve half of the estimated gain due to constant overhead in distributing and merging computation.

Table 3.2: Running time of the data-driven algorithms to detect URL Handovers and Changes.

	Spark Implementation	Single-core	Speedup
<i>Handovers</i>	7.9 minutes	15.5 hours	120×
<i>Changes</i>	7.3 minutes	13 hours	106×

On Large Scale Data

Finding handovers requires a large scale group-by order-by on URLs. Although some users change URLs very frequently, total number of URLs is much smaller than the number of tweets. Therefore, the detection process essentially scales linearly with respect to the number of tweets. Figure 3.15 shows the linear trend that we obtain experimentally on our dataset of 130 million tweets.

We check the effect of adding more computing workers on the running time. We increase the number of workers from 1 to 16. As it is illustrated in Figure 3.15, doubling

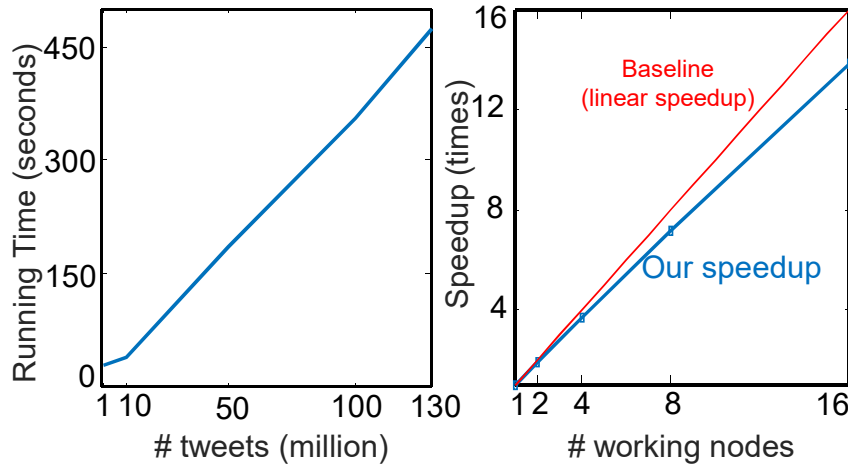


Figure 3.15: (left) How running time changes as we add more tweets. (right) How adding more working nodes helps us to detect the handover faster. We get an almost linear speedup by adding more nodes.

the number of workers almost halves the running time because the length of the list for each URL is much smaller than the number of URLs. We claim that this decreasing trend will not stop until we deploy as many workers as the number of unique URLs in our dataset, since each worker can work independently on a URL in that case. We get 14X speedup by using 16 worker nodes, which we consider to be a very effective implementation.

3.7 Related Work

URL changes and handovers have been actively performed by users in social media. To the best of our knowledge, our work is one of the first ones to investigate the association between these activities and abuse in social media. Authors of [50] has also been studying other aspects of reusing screen names in a parallel research project. Research has been done on other various aspects of abuse in social media including account hijacking [75], trolling [30], faking [24] and trafficking fraudulent accounts [76]. All of these works provide an important perspective on how fraudsters, merchants and abusers are manipulating social media for their own benefit. Our work considers URL handovers in the same man-

ner. There are several works on bot and automated user account detection in social media using data mining techniques. In [33], the authors have modeled the inter-arrival time between tweets to understand bot behavior. In [31] and [44], supervised techniques are used to detect bots at registration time.

3.8 Conclusion

We develop methods to detect URL handovers between accounts in social media using publicly available data. We perform an in-depth analysis on the users who perform URL changes and handovers and identify several interesting characteristics. Collaborative abusers exploit this ability to change their URLs in social media to trick regular human users into following spam accounts. Our data analysis discovers automated and collaborative handovers in temporal and connectivity profiles of these users, and provides useful insights into how the abusers are operating. In future work we will develop active prevention based on these insights.

Chapter 4

Distributed Pattern Matching over Streaming Time Series

4.1 Introduction

Given a set of time series patterns, we consider the problem of matching the patterns to the most recently observed samples (i.e. sliding window) of a streaming time series in order to identify occurrences of the patterns. Matching previously observed and annotated patterns over a streaming time series can rapidly identify critical events and provide valuable lead time to manage disasters in diverse applications such as patient monitoring, seismic activity monitoring and process monitoring. As simple the problem may appear, recent developments in the data collection, distribution [1] and mining [89][84], and wide adoption of real-time systems [46][85] have exposed the shortcomings of existing techniques. Modern sensing and transmission systems enable streaming data collection at an *unprecedented rate*; data mining techniques extract *thousands of meaningful patterns*, and time critical applications demand *rapid response*. Existing techniques fail at each of these advancements separately.

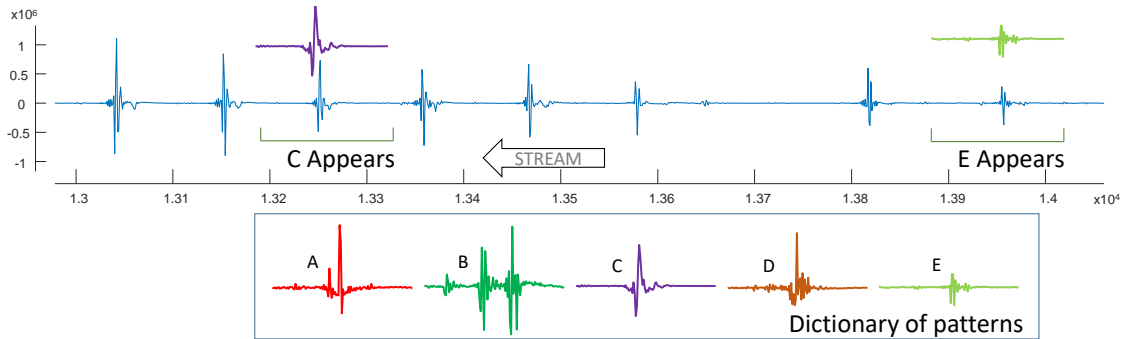


Figure 4.1: A dictionary of five patterns and a stream moving from right to left. Streaming pattern matching systems identifies any occurrence of any of the patterns in the stream. Pattern C and E appear in the stream with significant correlation.

Existing methods on streaming pattern matching are developed as single CPU systems that start that fall behind the stream, incrementally delay producing outputs, and eventually overflow allotted memory [64][61]. For example, a typical EEG headset produces 256Hz signals at 60 electrodes, creating a stream of 15,000 observations per second. If we have a small dictionary of only *one* eight-second (2,048 samples) long pattern, single CPU systems fail. Even if the systems worked, their unrealistic assumptions such as no-normalization [64] and equal length patterns [48], would limit their applicability to monitoring applications.

We develop a distributed pattern matching system, *DisPatch*, for streaming time series data that is *exact*, *scalable* and guarantees to produce the outputs within a *bounded delay*. In the scenario above, *DisPatch* can match a pattern with two workers and guarantees detection *within a second*. *DisPatch* scales almost linearly to match more patterns with more workers within a second. *DisPatch* performs overlapping windowing operations and distributes the windows and the patterns to all available workers. The computation in each worker is optimized with state-of-the-art techniques for pruning, abandoning and reusing computation. *DisPatch* is flexible enough to work with various distance measures including Euclidean distance, Pearson’s correlation coefficient, Manhattan distance and Dynamic Time Warping (DTW) distance. *DisPatch* assumes no structure on the pattern dictionary, which can have variable length patterns in any order.

Our contribution in this work is summarized below:

- We develop one of the first *distributed systems* to match a dictionary of patterns to a stream of time series.
- DisPatch guarantees *exactness* and *bounded detection delay* at the same time.
- Given enough resources, DisPatch comprehensively covers several independent dimensions: data rate, number of patterns, length of patterns, and maximum delay.
- DisPatch uses a novel distribution strategy to maximally utilize available compute resources and adopts state-of-the-art optimization techniques for pruning, abandoning, and reusing computation.
- We show novel applications of dictionary matching in critical monitoring tasks.

We discuss the background in Section 4.2 and the general framework in Section 4.3. We explain our map-reduce implementation in Section 4.4. An optimized version of the map-reduce framework is presented in Section 4.5. We describe the empirical evaluation in Section 4.6, the applications in Section 4.7, and the related work in Section 4.8. Finally, we conclude in Section 4.9.

4.2 Background and Definition

4.2.1 Problem Definition

We define a streaming time series, T , as a sequence of (t_i, v_i) pairs representing the timestamp (t_i) and value (v_i) of the i th observation. The timestamps are integers in any valid time unit (e.g. milliseconds, minutes, etc.) The values are real numbers. We define the maximum sampling frequency (i.e. data rate) of a stream as R . Our system assumes a constant data rate at R , accommodating a non-uniform data rate up to R is discussed as a future work in Section 4.9. In most application domains, the data rate is set by many other

economic, environmental and operational constraints; therefore we assume no bound on the data rate.

A *subsequence* of a streaming time series is a contiguous sequence of observations (i.e. (t, v) pairs) over a period of time. The length of a subsequence is the number of observations in a subsequence (assuming constant data rate). A *pattern* is a subsequence of a specific length. A *pattern dictionary* is a set of patterns $P = \{p_1, p_2, \dots, p_N\}$ of arbitrary lengths represented by $|p_i|$. Here, N is the number of patterns in the dictionary, and L is the length of the longest pattern in the dictionary (in seconds).

We define a *matching function* $M(d(x, y), \tau_d)$, where $d(x, y)$ is a distance function and τ_d is a threshold representing the maximum allowable distance to match two time series x and y of any length. $M = true$ only if $d(x, y) < \tau_d$. An analogous definition of the matching function, $M(s(x, y), \tau_s)$ can be produced with a similarity measure $s(x, y)$ and a minimum similarity threshold (τ_s). Our system supports multiple distance and similarity measures such as correlation coefficient, Euclidean distance, Manhattan distance and Dynamic Time Warping (DTW) distance.

Problem Definition: Given a streaming time series $T = \{(t_i, v_i)\}$, a set of patterns $P = \{p_1, p_2, \dots, p_n\}$, a matching function $M(d(x, y), \tau_d)$, and a maximum time delay max_D , identify all subsequences in the stream matching to a pattern in the dictionary no later than max_D seconds after the occurrences of the subsequences. DisPatch evaluates the input parameters and the available resources and starts processing only if it can guarantee the exactness and the maximum delay, otherwise it halts at the beginning.

The above problem seeks an *exact* report of *all* matching subsequences. The need for exactness has been relaxed to gain processing speed in the past [39]. However, our target domains are critical enough to care about all matches, as opposed to 90% of the matches. In addition, we can always gain speed using more resources in a distributed setting; sacrificing exactness is no longer necessary for tractability.

The generic problem assumes one streaming time series. Ideally, for multiple indepen-

dent streams, our proposed system can simply be replicated.

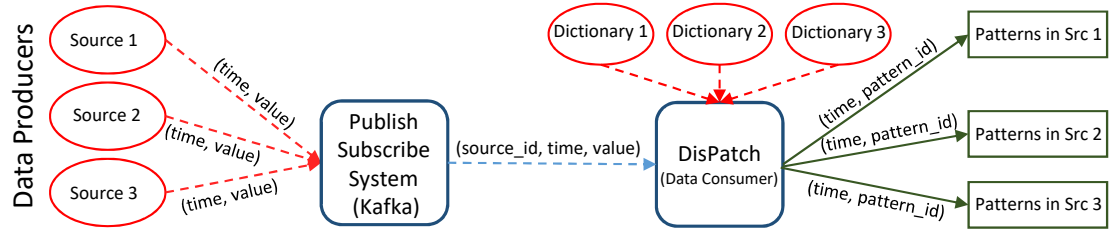


Figure 4.2: An overview of the inputs, outputs, and how they are connected to our system. Each source (stream) has its own dictionary of patterns.

4.2.2 Distributed Stream Processing Systems

We propose, DisPatch, a distributed method to solve the generic dictionary matching problem with delay guarantee.

Apache Kafka: In order to have a standard streaming protocol for our system, we use Apache Kafka, which is a high-throughput distributed messaging system [1]. Kafka is a publish-subscribe system which maintains a set of topics to which users subscribes to. *Data producers* publish the messages to a *topic*, and a copy of the message is sent to all *data consumers* who have subscribed to that topic. We create a topic per data stream, and our method acts as the consumer of all topics. This makes it possible for different data sources to publish their messages (data objects) to the Kafka server as producers, and our method subscribes to all topics to receive and process data from different sources in the same system. Our system linearly scales to multiple sources by just adding more hardware resources for processing (See Figure 4.2).

The Kafka server combines multiple streams in one common stream in an arbitrary order. Consider a stream of data objects (s, t, v) where s is the stream ID, t is the time, and v is the value observed at that time. Assume we have two streams of data, one for the temperature and one for the humidity of a station. The object $(1, 20, 78)$ denotes the 78°F temperature at the station at time 20, and $(2, 22, 0.8)$ denotes the 80% relative humidity at

time 22. We may receive (2,22,0.8) before (1,20,78). We adopt this flexibility and do not assume any specific order of data arrival.

Spark Streaming: Spark Streaming [3] is a system that supports running map-reduce jobs on data streams. Spark Streaming is an extension of the core Spark API [2] that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Spark Streaming collects stream data into small batches and processes each batch in distributed manner. Batches can be as small as 1 second long to as large as multiple hours long. Although we use Spark Streaming for implementing our method, any framework which splits the data in to batches and can run map-reduce jobs on these batches can be used to implement our methods.

4.3 General Framework

The dictionary matching problem has previously been solved on single-CPU without any performance guarantee; this motivates us to consider two novel research goals:

- **Scalability:** The system should scale to a large number of combinations (if not *all* combinations) of input parameters.
- **Performance:** The system should *always* (as opposed to on-average) produce output within the given delay requirement.

We propose a distributed pattern matching system, called DisPatch, shown in Figure 4.2. The DisPatch framework inputs a multiplexed stream of data at an arbitrary rate and a dictionary of patterns of arbitrary lengths, and outputs the occurrences of the patterns in the stream. Each input data stream can have its own dictionary of patterns. The system consists of three components:

- **Windowing:** DisPatch batches the streaming data using overlapping windows. Windowing is a not a trivial task because of several free parameters including slide

amount, overlap amount, and window size.

- **Distribution:** DisPatch distributes the workload across a set of workers. We use map-reduce style distribution and implement an optimal distribution strategy.
- **Computation:** DisPatch uses single-CPU pattern matching algorithms with state-of-the-art optimizations to prune off, abandon and/or reuse overlapping computation.

4.3.1 Windowing

The input to the DisPatch framework is a stream of time-value pairs: $(t_1, v_1), (t_2, v_2), (t_3, v_3), \dots$. In order to break down the stream in small batches, we have to set two parameters: the length of the window in seconds (w_s) and the length of the overlap between two consecutive windows in seconds (w_o). We introduce a novel approach to specify these two parameters such that our system can guarantee *bounded delay* and *exactness* with *optimum* overhead. The two parameters are defined as below.

$$w_s = w_o + max_D \quad (4.1)$$

$$w_o = \max(1, \lceil L \rceil) \quad (4.2)$$

w_s, w_o , and max_D must be positive integers with a unit of second. $\lceil L \rceil$ selects the nearest larger second of L . In all of our application domains, a delay of one second is more than sufficient, while smaller units of time can be easily attained for ultra fast data rate (e.g. MHz) using more resources. State-of-the-art stream processing framework (e.g. Spark Streaming) use the same windowing unit. Considering the way we define the windowing parameters, one can easily derive the following relationship.

$$w_s \geq \max(2, L) \quad (4.3)$$

In other words, $w_s \geq 2$ and $w_o \geq 1$. Our system distributes and computes each window independently and slides the window $w_s - w_o = max_D$ second forward to create the next

batch, and finish computation for current batch. The system guarantees bounded delay assuming that we successfully scale the computation to finish within max_D seconds. If DisPatch cannot finish the computation in $w_s - w_o$, it alerts the user to provide more worker nodes and/or relax the problem conditions. It halts otherwise.

The above windowing guarantees the exactness of the DisPatch framework. The overlapping segment ensures that we never miss a pattern on the border of two successive windows because w_o is longer than the longest pattern. Note that, for shorter w_o , the system may miss some of the occurrences of patterns having larger lengths (i.e. $|p_i| > w_o$)¹.

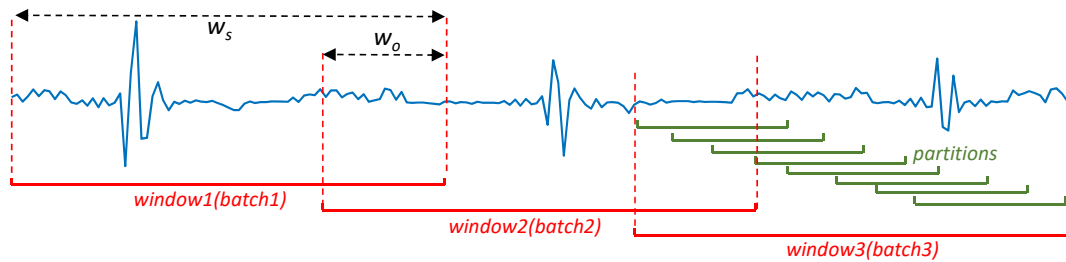


Figure 4.3: The general framework of our system. Step 1: How we create batches out of the streaming time series (red segments). Step 2: How We create partitions for each batch (green segments). Two parameters w_s (window size) and w_o (window overlap size) are also shown in the figure.

4.3.2 Distribution

After windowing the input stream, the data points that gather on the same window create a batch. At this stage, we have a time series of size w_s , and a set of patterns P to match with. We need to slide each pattern in the set along the time series and check if it matches with any subsequence by using the matching function M .

¹We have defined all the length parameters (L , w_o , and w_s) in second unit. We refer to these parameter in the rest of the document either in seconds or in points interchangeably which would be clear from the context. To convert a parameter from the second unit to the point unit, we multiply it by R . Obviously, the reverse is done by dividing the length by R .

To exploit distributed computing platform, we distribute the matching process across many workers. Since each pattern in the dictionary is matched independently of the others, it is reasonable to assign each worker a subset of the patterns (or exactly one pattern) to match. The time series in a batch can be distributed as well. In the naive extreme, we can distribute each distance or similarity calculation between a pattern and a subsequence of identical length to an independent worker. On the smart extreme, we can distribute the batch to minimal number of workers to exploit the overlap between successive distance calculation. To ensure bounded delay the system must finish computation in max_D seconds, and a sweetspot may lie somewhere in-between. We create overlapping *partitions* of the batched time series and have each worker compare a partition with a subset of the patterns (See Figure 4.3). The size and the number of partitions are dynamically selected based on the problem complexity and the number of workers. We slide the partition window one point at a time for the default distribution strategy (Section 4.4). In the optimized distribution strategy (Section 4.5), we either slide it multiple points or create one partition per batch (no sliding).

4.3.3 Computation

Each worker receives a partition of the the time series and a subset of patterns. The worker has to find all the subsequences of the times series that are matched with any of the patterns and report them. In case the size of the partition and the size of the pattern is equal, the worker just has one subsequence to check. The naive approach is to sequentially match each pattern along the time series. The naive approach has a total computational cost of $O(w_s L)$ for linear distance functions (e.g. Euclidean distance) and $O(w_s L^2)$ for quadratic distance functions (e.g. DTW). Based on the distance function chosen by the user, we optimize the way the worker is searching the patterns to make it faster. Since this part is the bottleneck of the whole pipeline from the execution time perspective, making it faster will speed up the whole system. Note that a new batch is ready to process every $w_s - w_o$ seconds. This means that all the calculations on a single batch should be done in $w_s - w_o$

seconds or the system will fall behind the stream and can not guarantee the maximum delay anymore.

4.4 DisPatch in Map-Reduce

Once the windowing is done, we need to have a distributed matching algorithm to find all the matched subsequences. We now provide a map-reduce based algorithm for DisPatch framework which covers *Distribution* and *Computation* steps discussed in Sections 4.3.2 and 4.3.3.

4.4.1 Map-Reduce Framework

Map Operation

For each unique pattern length in the dictionary, we create a sliding window of the same size, slide the window along the time series, and send a set of key-value pairs to the reducer. This set of key-value pairs can later be consumed by the reducers to create the partitions for a given batch of data points. For any data point like (t_i, v_i) in the input stream, we generate multiple key-value pairs each of them playing the role of that data point in a different partition.

Figure 4.4 shows an example of how our map-reduce framework works. There are three patterns, two of them have 4 points, and one of them has 5 points. We have to create all possible subsequences of lengths 4 and 5 in our data batch. A data point like $(5,0)$ is a part of $[1:5], [2:6], \dots, [5:9]$ subsequences of length 5, and a part of $[2:5], [3:6], \dots, [5:8]$ subsequences of length 4. Therefore, we create 9 key-value pairs shown in Figure 4.4 for this data point. Note that although we have two patterns of length 4 in the dictionary, we create subsequences of length 4 once. This reduces the amount of memory needed by the workers.

Algorithm 5 Map function in DisPatch

Input: A data point (t, v)
Output: A set of key-value pairs
 S =empty set
 Q =set of unique patterns' lengths
for $i = 1, \dots, |Q|$ **do**
 $l = Q[i]$
 for $j = t - l, \dots, t$ **do**
 $tuple = ((j : j + l), [(t, v)])$
 $S.append(tuple)$
return S

In Algorithm 5, we generate a set of tuples $((j : j + l), [(t, v)])$ for each data point (t, v) . j is the first index of the partition, and $j + l$ is the last index. $(j : j + l)$ is used as the key for the reduce function. All data points with identical $(j : j + l)$ values are sent to the same reducer where they are sorted based on their indices, and later (not in this step) compared against the patterns of length l in the dictionary.

Reduce Operation

Once all the key-value pairs are generated in the mapping step, the reducer puts together and sorts all the key-value pairs based on the time index. The output of the reducer is all the subsequences sorted and ready to be matched with the patterns in the dictionary. Algorithms 5 and 6 shows how we create and distribute the partitions through mappers and reducers. These two operations together form the *Distribution* step.

We use the merge sort to merge and sort the time-value pairs that are given to the reduce function as the input. Note that the running time of Algorithm 6 is $O(n + m)$ since both input lists are already sorted.

Algorithm 6 Reduce function in DisPatch

Input: Two sorted lists of key-value pairs A and B **Output:** A constructed partitionlet $A = ((j : j + l), [(t_1^1, v_1^1), \dots, (t_n^1, v_n^1)])$ let $B = ((j : j + l), [(t_1^2, v_1^2), \dots, (t_m^2, v_m^2)])$ $S =$ Do a merge sort on the set of time-value pairs in A and B based on time**return** S

Flat Map Operation

This is the *Computation* step where each worker's job is to compare a subset of the patterns in the dictionary (which have identical lengths) with a subsequence of exactly the same size by using the matching function M . The pattern and the subsequence at this step are of identical length because of the way we create the partitions. The worker gets a subsequence of length l and compares that against all the patterns of length l in the dictionary by using the matching function.

Algorithm 7 Flat map function in DisPatch

Input: Subsequence X , Dictionary P **Output:** Matched patterns with X if any $S = \emptyset$ **for** $i = 1, \dots, |P|$ **do** **if** $|X| == |P[i]|$ **then** **if** $M(d(X, P[i]), \tau_d) == True$ **then** $S.append(i)$ **return** S

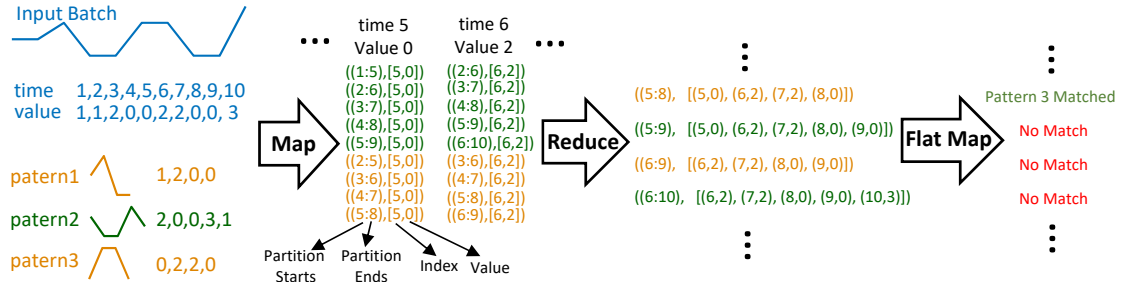


Figure 4.4: A tiny example that shows how the DisPatch framework works in map-reduce fashion. We use coloring to show the correspondence between the patterns and the partitions we generate in the *Distribution* step. In this example, the DisPatch framework has found a match with patterns 3 starting from time 5. Note that this is the default *Distribution* and *Computation* strategy where we slide the partition window one point at a time. In the optimized version of our system, we may slide the partition window multiple points, and exploit the overlap between successive distance calculations.

4.4.2 Analysis of the map-reduce framework

The map function of the DisPatch framework (Algorithm 5) produces $(w_s N l)$ tuples where N is the number of patterns in the dictionary and l is the average length of the patterns in the dictionary in points. This number can get really big for real world applications. Since all the computations on a batch need to be done in $w_s - w_o$ seconds, the baseline method may fall behind the stream if the mappers and reducers have to do lots of jobs. However, based on the matching function chosen by the user, we can optimize the *Distribution* and *Computation* steps in such a way that the system does a lot less computation. We will discuss the optimization techniques in Section 4.5.

4.4.3 Supported Distance Measures

Our system supports different distance/similarity measures to be used in the matching function. The user has the freedom to choose the measure based on his needs. He can also define his own distance function. Since every measure may require its own optimization techniques, we discuss each of them separately. We first explain how each distance

Table 4.1: List of supported distance (or similarity) measures and the corresponding matching functions.

Dist./Sim. Measure	Matching Function
Correlation Coefficient	$\text{Cor}(x,y) \geq \tau_s$
Euclidean Distance	$\text{Euclidean}(x,y) \leq \tau_d$
Manhattan Distance	$\text{Manhattan}(x,y) \leq \tau_d$
DTW Distance	$\text{DTW}(x,y) \leq \tau_d$

measure can be calculated and its time complexity, and then we discuss the techniques to optimize it. Given two time series P and Q of length n , we define 4 distance/similarity measures that are supported in our system. See Table 4.1 for a list of supported matching functions.

There is no specific range for any of these measures (except correlation coefficient). Since the distance between the subsequence and the pattern can be higher for longer patterns, the user would have to give a separate matching threshold for each pattern in the dictionary. To provide more flexibility and consistency for the user, we normalize all the distance measures in the range $[-1,1]$. In this case, the user can provide one threshold for the matching function, and we use that to find the matches regardless of the length of the patterns. We will discuss the normalization process for each distance measure separately.

We have implemented 4 most popular distance/similarity measures for our system. We explain two of them in this section (Manhattan and DTW) along with techniques we use to speed them up. We explain the other two distance/similarity measures (Euclidean distance and correlation coefficient) in Section 4.5 where we show how to exploit the overlap between successive distance calculations to increase the processing capacity of the system.

Manhattan Distance

Manhattan distance can be calculated by using Equation 4.4. The time complexity to calculate this distance is $O(n)$ where n is the length of the input time series. We define the

matching function as $Manhattan(x, y) \leq \tau_d$.

$$Manhattan(P, Q) = \sum_{i=1}^n |P_i - Q_i| \quad (4.4)$$

To optimize the pattern matching process under Manhattan distance, we use the early abandoning technique in the *Computation* step. Since Manhattan distance is a sum of some non-negative numbers, the early abandoning technique is applicable. We basically abandon the distance calculation whenever we pass the distance threshold provided by the user.

DTW Distance

DisPatch allows matching patterns under an elastic distance measure such as Dynamic Time Warping (DTW). Dynamic time warping allows signals to *warp* against one another. Constrained DTW allows warping within a window of w samples and is defined as follows.

$$DTW(x, y) = D(m, m)$$

$$D(i, j) = (x_i - y_j)^2 + \min \begin{cases} D(i-1, j) \\ D(i, j-1) \\ D(i-1, j-1) \end{cases} \quad (4.5)$$

$$D(0, 0) = 0$$

$$\forall_{i>0, j>0} D(i, 0) = D(0, j) = \infty$$

$$\forall_{|i-j|>w} D(i, j) = \infty$$

We normalize the DTW distance by the length of the warping path. This scales the DTW distance in the range $[-1, 1]$.

$$Norm_DTW(P, Q) = \frac{DTW(P, Q)^2}{2 \times PL(P, Q)} - 1 \quad (4.6)$$

Here $PL(P, Q)$ is the warping path length of the solution in the dynamic programming table of DTW. Given a time series X and a query Y , where $|X| = n$, $|Y| = m$, and $n \geq m$,

calculating the DTW distance between Y and all subsequences of X of length m takes $O(nm^2)$. This is an order of magnitude more computation compared to the non-elastic measures, requiring more workers to guarantee the same max_D .

There are dozens of papers that describe speeding up techniques for DTW based similarity search. Lower bounding technique [41], early abandoning technique [61], hardware acceleration technique [66], summarization technique [80], anticipatory pruning [26], etc. We use the method in [41], called *LB_Keogh*, to speed up the process of finding the matches. For completeness, we describe *LB_Keogh* here. *LB_Keogh* generates two envelopes (U and L) of a candidate and compares them with the query to produce the lower bound. *LB_Keogh* takes linear time for computation and insignificant off-line preprocessing for the envelopes. The formula to compute U , L and *LB_Keogh* where w is the constraint window is given below.

$$\begin{aligned}
 \forall i U_i &= \max(x(i-w : i+w)) \\
 \forall i L_i &= \min(x(i-w : i+w)) \\
 LB_Keogh &= \sum_{i=1}^n \begin{cases} (y_i - U_i)^2 & \text{if } y_i > U_i \\ (y_i - L_i)^2 & \text{if } y_i < L_i \\ 0 & \text{otherwise} \end{cases} \quad (4.7)
 \end{aligned}$$

To optimize the pattern matching under DTW distance, we use *LB_Keogh* in the *Computation* step. The worker receives an equi-length subset of the patterns and a subsequence. For each pattern, the worker calculates the *LB_Keogh* in linear time. If the calculated *LB_Keogh* is larger than the maximum DTW distance (τ_d) specified in the matching function, we do not calculate the real DTW distance, since the real DTW distance is definitely larger than the maximum allowed. Otherwise, we compute the expensive DTW matrix.

Euclidean distance

Euclidean distance can be calculated by using Equation 4.8. The time complexity of calculating the Euclidean distance is $O(n)$. In this case, we define the matching functions as $Euclidean(x, y) \leq \tau_d$. We describe how to exploit the overlap between successive Euclidean distance calculations and optimize the framework for this distance measure in Section 4.5.

$$Euclidean(P, Q) = \sqrt{\sum_{i=1}^n (P_i - Q_i)^2} \quad (4.8)$$

Correlation Coefficient

Pearson's correlation is a measure of the linear dependence between two time series P and Q , giving a value between +1 and -1 inclusive, where 1 is total positive correlation, 0 is no correlation, and -1 is total negative correlation. We explain how to speed up matching under correlation coefficient in Section 4.5.

$$Cor(P, Q) = \frac{\sum_{i=1}^n (P_i - \bar{P})(Q_i - \bar{Q})}{\sqrt{\sum_{i=1}^n (P_i - \bar{P})^2} \sqrt{\sum_{i=1}^n (Q_i - \bar{Q})^2}} \quad (4.9)$$

4.5 Exploiting The Overlap

In Section 4.4, we discuss our map-reduce based framework. In this section we explain how we can re-design the *Distribution* and *Computation* steps by exploiting the overlap between successive distance calculations to increase the processing capacity of the system tremendously. We explain and implement the *Distribution* and *Computation* strategies

for Euclidean distance and correlation coefficient. We also explain how this idea can be adopted for Manhattan distance and DTW as well.

4.5.1 Euclidean Distance

Given a time series X and a query Y , where $|X| = n$, $|Y| = m$, and $n \geq m$, calculating the Euclidean distance between Y and all X 's subsequences of length m takes $O(nm)$. We explain how this process can be done faster in this section.

Pattern matching under a non-elastic linear distance measure (e.g. correlation, Euclidean distance, Manhattan distance) requires calculating the dot products between the pattern and all subsequences of a given time series. DisPatch uses our recent proposed algorithm, MASS (Mueen's Algorithm for Similarity Search) [84] that finds all distances between the query pattern and subsequences of the time series in $O(n \log n)$ operation. Algorithm 8 and Figure 4.5 describe the convolution based process of calculating all sliding dot products. Sliding dot products are sufficient to calculate the Euclidean distances with or without z-normalization. Using MASS, we can calculate the Euclidean distance between Y and all X 's subsequences of length m in $O(n \log(n))$. We refer readers to MASS website for the complete algorithm [57].

Algorithm 8 SlidingDotProducts

Input: A time series X , a pattern Y , $|X| > |Y|$

Output: All sliding dot products between Y and subsequences of X

$n \leftarrow \text{length}(X)$, $m \leftarrow \text{length}(Y)$

$X_a \leftarrow \text{Append } X \text{ with } n \text{ zeros}$

$Y_r \leftarrow \text{Reverse } Y$

$Y_{ra} \leftarrow \text{Append } Y_r \text{ with } 2n - m \text{ zeros}$

$Y_{raf} \leftarrow \text{FFT}(Y_{ra})$, $X_{rf} \leftarrow \text{FFT}(X_r)$

$XY \leftarrow \text{InverseFFT}(Y_{raf} * X_{rf})$

return XY

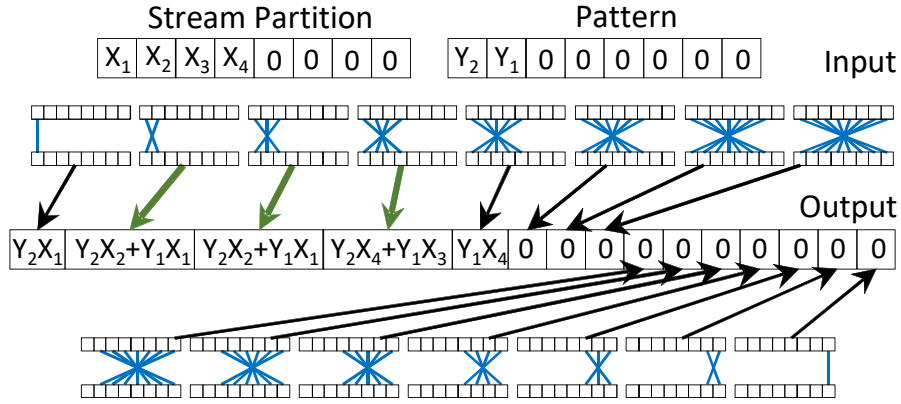


Figure 4.5: A toy example of convolution operation being used to calculate sliding dot products for time series data. Note the reverse and append operation on X and Y in the input. Fifteen dot products are calculated for every slide. The cells $m (= 2)$ to $n (= 4)$ from left (green/bold arrows) contain valid products.

The way we specify w_s guarantees that $w_s \geq L$. This means that even the longest pattern in the dictionary fits in one batch. The main purpose of the *Distribution* step is to distribute matching tasks to the workers to gain speed up. The default distribution strategy creates all overlapping partitions for each pattern length in the dictionary so that the partition and the patterns have identical lengths. Considering the improved distance calculation method we discussed in this section (MASS), we create longer partitions (if possible), and the workers in the *Computation* step check a subset of the pattern dictionary against all subsequences in the partition. The partition size can be equal to w_s in some cases meaning that we create just one partition for a given batch. If $N \geq \#workers$, we create one partition per batch and distribute the patterns between the workers. This gives us the maximum speed. If $N \leq \#workers$, it means that we do not have enough patterns to keep all the workers busy. Therefore, it is better to create more than one overlapping partitions in each batch to utilize all the workers. Ideally, we want to create $\frac{\#workers}{N}$ overlapping partitions. In order to not miss any pattern on the border of two partitions, we make sure that the length of the overlap is no less than L . We basically use the same idea as Section 4.3.1 to overlap the partitions. We set the overlap size to L . Therefore, the length of each partition would be $L + \frac{(w_s-L) \cdot N}{\#workers}$. In this case, we send a subsequence of the

batch and a pattern to a worker. This utilizes all the workers maximally and produces the results as fast as possible.

Note that the new approach is much faster than the default *Distribution* and *Computation* strategies, and it also requires less memory since it produces less key-value pairs in the *Map* operation. The reason is that it always generates less number of partitions for a given data batch and dictionary compared to the default strategy.

4.5.2 Correlation Coefficient

Pearson's correlation coefficient and normalized Euclidean distance are related as below [56].

$$\text{corr}(P, Q) = 1 - \frac{d^2(\hat{P}, \hat{Q})}{2m} \quad (4.10)$$

Here \hat{P} is the z-normalized form of P and m is the length of the two time series. The time complexity to calculate the Pearson's correlation is $O(n)$. We define the matching function as $\text{corr}(x, y) \geq \tau_s$. We can use the same optimization technique that we use for Euclidean distance with a little tweak since these two measures are directly related by the above equation.

4.5.3 Manhattan Distance

Manhattan distance is always larger than Euclidean distance because of the triangle inequality. We use this fact to calculate lower bound distance using the convolution based technique described above. Subsequences for which the lower bounds are larger than the given threshold (τ_d) do not require further validation by exact distance computation.

The goodness of this method depends on two factors: tightness of the lower bounds that largely depends on data and the allowable overhead that corresponds to the computing platform the system is running on.

4.5.4 DTW Distance

Exploiting overlaps while matching a pattern against a long time series has been discussed in the literature [64]. The method costs $O(nm)$ time and space to match a pattern of length m with a long time series of length n . However, the algorithm in [64] does not z-normalize the subsequences before calculating the distances, which limits the applicability of the method in most of our application domains.

A naive exact method to match patterns with z-normalization under DTW distance requires $O(nm^2)$ time and $O(n)$ space. The methods in [61] reduces the computation time with pruning and early abandoning techniques while z-normalizing each subsequence independently. To prioritize z-normalized matching, we resort to pruning and abandoning DTW calculations in the workers (default *Distribution* strategy).

4.6 Empirical Evaluation

We use Apache Spark on a cluster with 16 workers for our experiments. Each worker has a single core CPU with 4 GB of memory. They all run Ubuntu 14.04. In the Apache Kafka server, we create a topic with 16 threads² so that each worker can read the data from one thread. There are 3 parameters for each experiment: data rate, dictionary size, and the delay.

In order to be able to compare the performance of our system across different experiments, we normalize the processing time of DisPatch. In each experiment, DisPatch has max_D seconds to process w_s seconds worth of input data. For example, if $max_D = 6$ and $w_s = 10$, we have 6 seconds to process 10 seconds worth of data. If DisPatch finishes the job in less than 6 seconds, it can move to the next batch on time. Otherwise, it cannot guarantee the 6 seconds maximum delay and will get behind the stream. In this case, if it

²The created topic has 16 partitions in Kafka server. We use the word *thread* instead of partition so that the reader does not confuse Kafka partition with the partition we explain in our method.

takes 4.8 seconds to finish the job for each batch, the *normalized processing time* that we report is $4.8/6 = 0.8$. Since this number is less than 1, DisPatch can finish processing the batch before the next batch is ready.

In each experiment, we report the normalized processing time after running the experiment for 30 minutes. This ensures that the reported number is representative and valid. As long as the normalized processing time is not greater than 1, the system can process the input stream in real time, otherwise it will get behind the input stream. The red dashed line corresponding to normalized processing time of 1 can be found in all figures.

Although DisPatch is an exact system and it finds *all* the subsequences which satisfy the matching function, we use [61] to find the ground truth to make sure that the system is 100% accurate. The max_D parameter can be any integer between 1 and $\lceil L \rceil$ seconds. We set $max_D = 1$ second in all experiments unless otherwise stated. We share the code and the data of all experiments on the project page [14]. We use the default *Distribution* strategy (Section 4.4) for DTW and Manhattan distances, and the optimized *Distribution* strategy (Section 4.5) for Euclidean distance and correlation coefficient.

4.6.1 Datasets

We use three datasets to run extensive experiments on our system using different distance measures and parameters. For all datasets, we either have the pattern dictionary or generate the dictionary by getting help from the experts in that field. Although each dataset has an original sampling frequency, we artificially replay the data to DisPatch at different rates to test the whole system with different parameters. The pattern matching process and the index of matched subsequences are independent of the sampling frequency. Higher frequencies just put more pressure on the system since it has to process more data points in one second.

Power Consumption Data This is the power consumption data collected from a dish-

washer in a smart house. The data was originally recorded at 0.15 Hz from 09 Oct 2013 to 10 Jul 2015. There are 100 patterns in the dictionary ranging from 21 to 717 points in length. The total length of all patterns is 8986 points.

EURO/USD Exchange Rate This dataset includes a month worth of EURO to USD exchange rate in September 2016. The data is recorded at 10 Hz. The pattern dictionary consists of 167 patterns ranging from 62 to 302 points in length having a total length of 35,070 points.

EEG Data This dataset consists of EEG³ recordings from a pediatric subject (patient number 24 in [69]) with intractable seizures. Subject was monitored for several days following withdrawal of anti-seizure medication in order to characterize his seizures and assess his candidacy for surgical intervention. All signals were sampled at 256 samples per second with 16-bit resolution. There are 16 seizure patterns in the dictionary ranging from 4,353 to 18,177 points in length having a total length of 134,928 points all together. This dataset is available at [5].

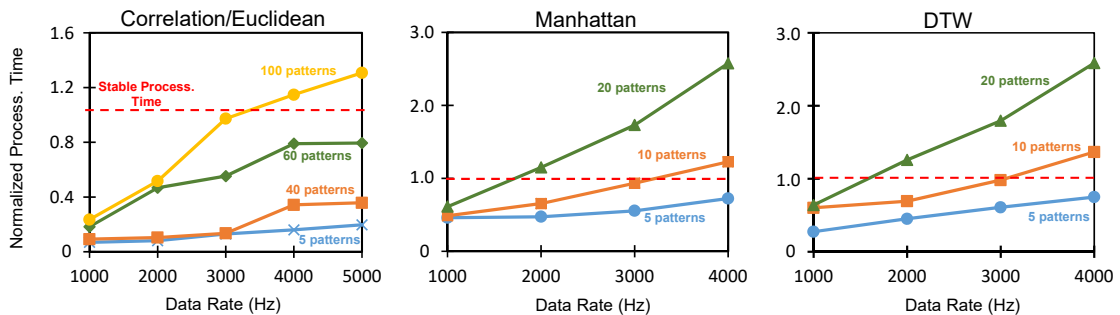


Figure 4.6: The results from the power consumption data. We use 16 workers and set $max_D = 1$ in all these experiments. As long as the points are below the red dashed line, the DisPatch framework can handle the stream in real time. It would get behind the stream for all the points above the dashed line.

³Electroencephalography (EEG) is an electrophysiological monitoring method to record electrical activity of the brain.

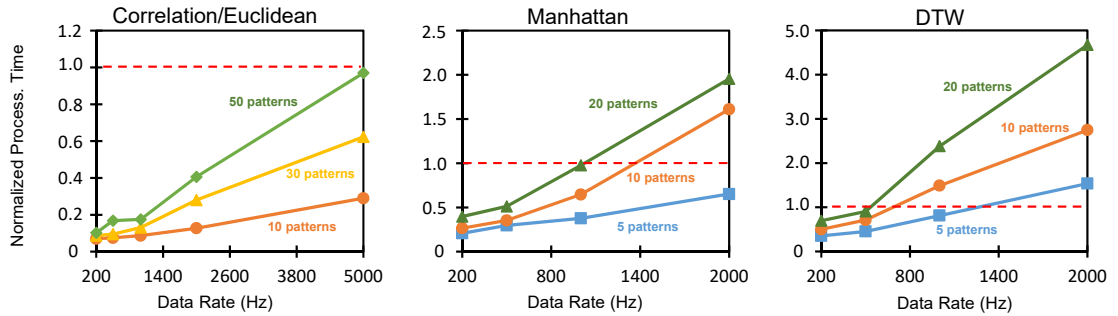


Figure 4.7: The results from the EURO/USD exchange rate data. We use 16 workers and set $max_D = 1$ in all these experiments. As long as the points are below the red dashed line, the DisPatch can handle the stream in real time. It would get behind the stream for all the points above the dashed line.

4.6.2 Data Rate and Dictionary Size

We test our system on different datasets which have various dictionary properties and domain parameters. Table 4.2 shows the effect of each parameter on the complexity of the dictionary matching problem. Figures 4.6 and 4.7 show the results. For each dataset, we replay the data at various rates and use different numbers of patterns in the dictionary. As long as the plots are under the dashed red line, the DisPatch framework can process the stream. We have the same plot for correlation coefficient and Euclidean distance since the underlying matching function for both of them is identical. These two measures can be directly translated to each other. Since DTW is the most expensive distance measure, the processing capacity of our system on DTW is less compared to other distance measures. The Manhattan distance in many cases has the same normalized processing time as DTW since we use the same *Distribution* strategy for both of them, and also the LB_Keogh lower bound for DTW distance is linear in time complexity. However, the DisPatch framework performs much faster on correlation coefficient (and Euclidean distance) because we exploit overlap between successive distance calculations and optimize both *Distribution* and *Computation* steps. For example, DisPatch can process 10 patterns on EURO/USD exchange rate data at 500 Hz under DTW distance while it can handle the same number of patterns at 5000 Hz under Euclidean distance.

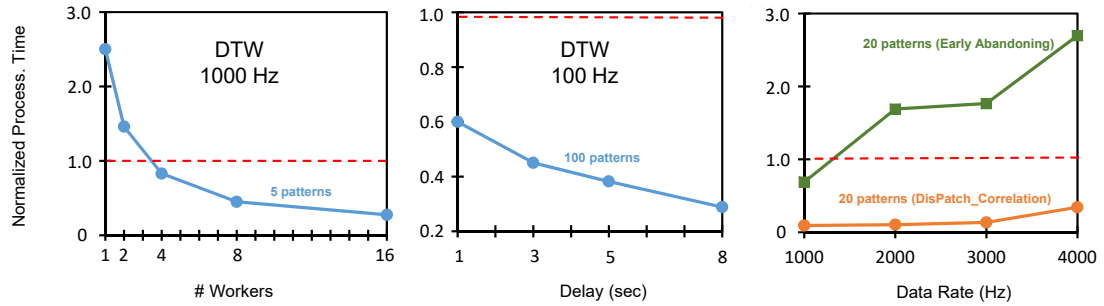


Figure 4.8: A set of experiments on power consumption data. (left) Scalability of DisPatch by adding more workers to the system. The minimum number of workers to not get behind the input stream is 4 in this example. (middle) The effect of the delay parameter on the processing time. Higher delays let DisPatch work under less pressure. Both number of workers and delay (max_D) have the same effect for all distance measures. (right) The comparison between the performance of DisPatch and the early abandoning technique to find matches under Pearson’s correlation. DisPatch can process 20 patterns at 4000 Hz while the early abandoning technique can do the same number of patterns at 1000 Hz.

Table 4.2: How each parameter is related to the complexity of the dictionary matching problem.

Parameter	Change	Effect on Complexity
R	↑	↑
L	↑	↑
max_D	↑	↓
\sum length of patterns	↑	↑

4.6.3 Speed up by Exploiting the Overlap

To have a fair comparison, we compare the performance of DisPatch on correlation coefficient with the early abandoning technique on the same similarity measure. For the early abandoning technique, we use the default *Distribution* and *Computation* strategies. Results are shown in Figure 4.9 and Figure 4.8 (right). The way we exploit overlaps (Section 4.5) and optimize the system makes it much faster. In case of power consumption data, DisPatch can process 20 patterns at 4000 Hz while the early abandoning technique can just handle the same number of patterns at 1000 Hz. For EEG data, DisPatch could process data up to 14 times faster than the early abandoning technique.

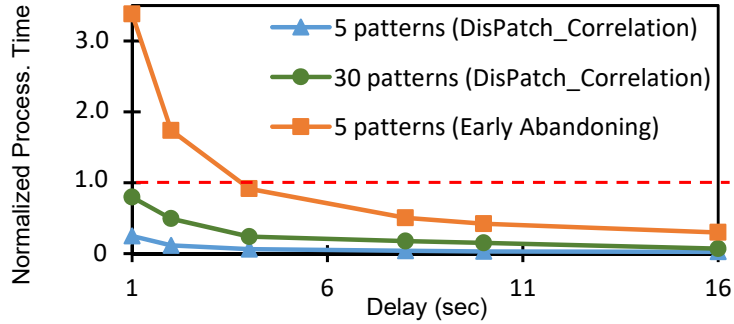


Figure 4.9: The comparison between performance of DisPatch and early abandoning technique on EEG data at 256 Hz. DisPatch is able to process data up to 14 times faster than the early abandoning technique. DisPatch can process 30 patterns with 1 second delay in real time. The total length of these 30 pattern is 522K points (34 minutes).

4.6.4 Scalability

In order to test the scalability of our system, we start from a cluster with 1 worker and double the number of workers in every step. The results are shown in Figure 4.8 (left). The processing time decreases between 40% and 46% in each step which is very close to linear speedup. Adding more workers always adds some overhead to the system. Note that the maximum speedup happens when the number of Spark workers are equal to number of threads created for the topic in the Apache Kafka server. In that case, each worker reads the data from one thread which would lead to maximum parallelism. We do not recommend having more workers than Kafka threads since the workers would be locked waiting for the data occasionally.

4.6.5 Delay Sensitivity

In this experiment we investigate the effect of max_D parameter on the performance of DisPatch. As shown in Figure 4.8 (middle), as the user lets the system find the patterns after higher delays, the processing time per second goes down. The reason is that DisPatch has to process w_s seconds worth of data each max_D seconds. Therefore, the load of the system is $\frac{w_s}{max_D} = 1 + \frac{w_o}{max_D}$. As we increase max_D , the load decreases and the system

can perform more efficiently.

Theoretically, the user can increase the delay with no limit and expect the system to work more and more efficiently, but there are some limitations in practice. Firstly, if the user picks a very large delay value, the system has to buffer more data for each window and it may run out of memory. Secondly, since we use Apache Spark to implement DisPatch, both w_s and w_o should be a multiple of the atomic batch unit⁴. To minimize the load of the system, we set the size of this atomic batch unit to max_D which also means that the w_o should be a multiple of max_D . Considering that the load is $1 + \frac{w_o}{max_D}$ and $w_o = max(1, \lceil L \rceil)$, we conclude that load of the system goes down as we increase max_D and this trend stops once max_D reaches $\lceil L \rceil$. To sum up, max_D can be any integer between 1 and $\lceil L \rceil$. After that, there is no point having higher delays since we can not take advantage of that due to practical limitations.

When using high data rate, we have to set $max_D = 1$ second because L gets smaller than 1 second, and as we mentioned before max_D should not be greater than $\lceil L \rceil$. The reason that we do the delay experiment at 100 Hz is to be able to increase max_D and see the performance of DisPatch. We would not be able to do that at high frequencies.

Since the scalability of DisPatch and the effect of delay on the performance are independent of the distance measure and other parameters, we report the scalability and delay results just for power consumption dataset. They would look similar for other datasets.

4.7 Applications

Matching a dictionary of patterns with a streaming time series enables novel monitoring, with use cases in critical application domains including patient monitoring and power usage monitoring. We show two unique use cases in this section.

⁴The word *batch* has a different meaning in Apache Spark compared to how we use it to explain our method.

4.7.1 Patient Monitoring

Numerous measurement devices exist that are used in various combinations to monitor patients in hospitals. For example, EEG devices measure brain activity, Oximeters measure the level of Oxygen in blood and so on. Such devices can stream uniformly sampled measurements, which can be analyzed to produce personalized dictionary of patterns and subsequently be matched to identify recurrence of events. To elaborate, consider the EEG recording of an epileptic patient in Figure 4.10 for 21 hours at 256 Hz. The patient had several seizure during data collection. We replay the data at 256 Hz to demonstrate a personalized monitoring scenario.

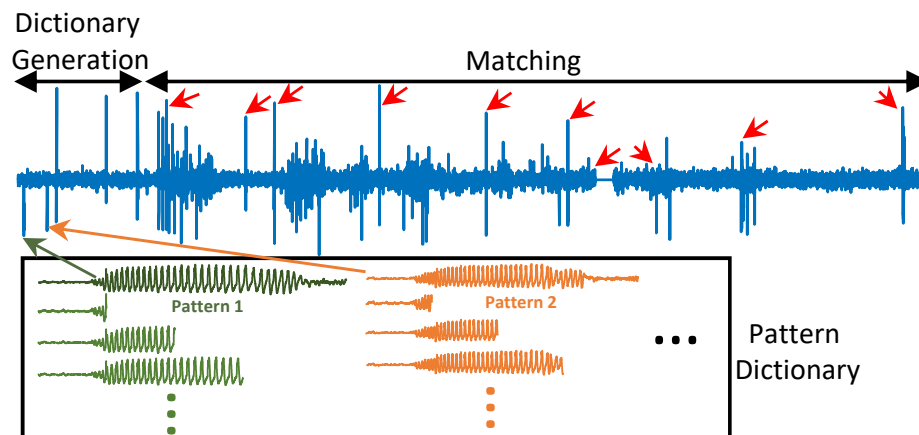


Figure 4.10: EEG recording of an epileptic patient for 21 hours at 256 Hz. We create an enhanced dictionary in an unsupervised fashion in the first 3.5 hours, and predict 10 out of 11 seizures well ahead of time in the remaining 18 hours. Red arrows show the index of the matched patterns.

We use an online unsupervised motif discovery tool [54] to create a dictionary of five repeating patterns by observing the first 200 minutes of the data (dictionary generation step). All of the five patterns correspond to seizures and show a clear signature shape. We enhance the dictionary by taking several prefixes so that we can detect the seizure as early as possible. Next we channel the stream through DisPatch with the enhanced dictionary and a minimum correlation coefficient of 0.6 as the matching function. The patient had eleven more seizures, and our algorithm could detect almost *all* of them (10 out of 11)

with up to twenty seconds of lead time. Thus, DisPatch along with the motif mining can potentially enable highly specialized monitoring with no previously labeled data.

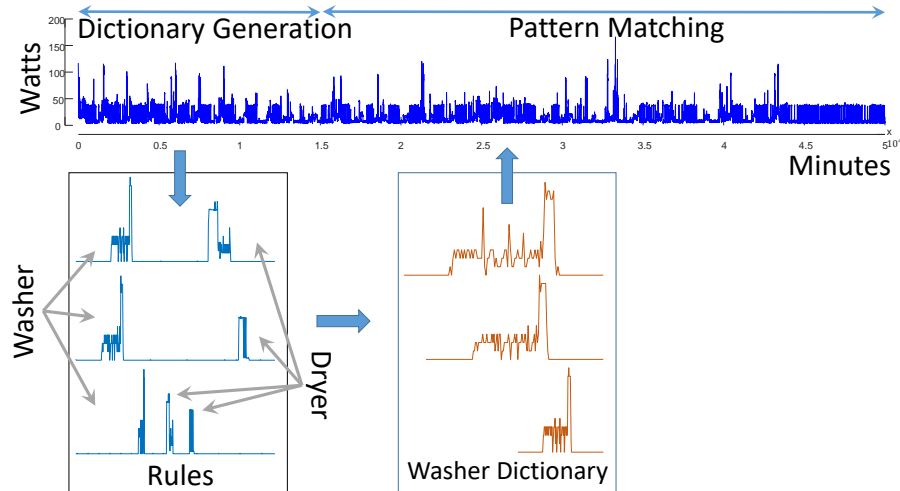


Figure 4.11: Whole house power consumption data [49] in the top. We build a set of rules using [70]. The antecedents of the rules form a dictionary, in this case a washer dictionary. DisPatch matches the dictionary to the subsequent data to predict dryer usages.

4.7.2 Power Consumption Monitoring

Monitoring and predicting power consumption is an important objective for both power companies and consumers. We demonstrate that DisPatch is useful in power consumption monitoring. Controllers of most home appliances and power tools run very simple forms of deterministic state machines which create signature patterns. In addition, home owners often develop a usage pattern with identical daily/weekly frequency, duration and sequence of appliance/tool usages. For example, a common usage pattern is that the *dryer starts very shortly after the washer stops*. Unsupervised rule discovery algorithms [70] can find such patterns in consumption history and create a set of rules followed by a dictionary of antecedent patterns. Note, in Figure 4.11, the gap between the washer and dryer, and the number of times an appliance is used are variable. We replay a stream of whole house power consumption data from a smart home and successfully predict dryer usage based on

washer usage with 65% precision.

4.8 Related Work

Time series similarity search [37] is the flipped version of pattern matching over streams. Numerous algorithmic optimization techniques have been discussed in the literature as part of time series similarity search that exploit indexing techniques for independent time series objects [67][27][42][40][78]. Indexing techniques are efficient for offline and random accesses, while completely unsuitable for streaming data. Other approaches include single-scan technique to support arbitrary query length [61] and similarity search for uncertain time series [65] among many others.

Matching a dictionary of patterns on a stream of data is studied in several contexts: streaming classification [68], statistical monitoring [88], dictionary matching [48] and monitoring under warping [64]. DisPatch is the first (to the best of our knowledge) to solve the problem on distributed systems with horizontal scalability. Existing dictionary matching algorithms [48] do not consider variable length patterns, DisPatch can match an arbitrary number of patterns of arbitrary lengths. The authors of [88] use a similar windowing technique to ours to compute aggregated functions for the data streams. Unlike their method, we adopt overlapping windows to guarantee exactness. Linear scan techniques to match a query pattern exploit *pruning* and *early abandoning* techniques [61]. However, DisPatch exploits overlaps between successive subsequences of a streaming time series. Although we use *Spark Streaming* to implement DisPatch, other similar systems [32] can also be used.

4.9 Conclusion

We describe a distributed pattern matching system for streaming time series data. Our system, DisPatch, is exact and, it scales to guarantee bounded delay. DisPatch has unique windowing and distribution strategy to employ state-of-the-art optimization techniques. DisPatch uses a convolution based approach to utilize overlapping subsequences. We show two compelling application scenarios where DisPatch can potentially make significant difference. Our grand future goal is to augment DisPatch with online unsupervised mining algorithms to develop a comprehensive distributed stream mining system. Handling a data stream with variable data rate can also be done in the future.

Chapter 5

Conclusion and Future Work

In this thesis, we described how to design and customize distributed data mining and knowledge discovery algorithms by taking in to consideration the properties of the data. We showed that traditional data mining algorithms are not sufficient for the new huge datasets. We discussed challenges in designing distributed algorithms and how to tackle them by designing data specific algorithms. The research presented in this thesis can be continued in many directions. We showed 3 problems in different domains. Since each domain that we discussed has its own properties and challenges, we discuss the future work separately.

1. **Pattern recognition for computer log messages:** Pattern recognition is a crucial component of log analysis tools. We presented an efficient and linearly scalable system to extract patterns from log messages. One future direction in this field is to find a clustering algorithm for log messages which is efficient and at the same time robust to noise. Although we showed that the order of logs does not play an important role in the quality of the clusters, there are corner cases in which we could end up getting low quality clusters for log messages. Another future direction to make our LogMine system better is to design an algorithm to dynamically adjust the rate of relaxing the condition as we build the hierarchy of patterns.

2. **Anomaly detection in social media:** The number of bots is constantly increasing in social media sites like Twitter. We presented a distributed algorithm that could detect URL handover in Twitter. Our work can be used to detect and suspend harmful bots and make social media sites safer. A future direction for this work is to train a model based on the URL handovers that have happened so far, and predict which account and when is going to hand over a URL to another account. If predicted accurately, we can claim the URL and stop the handover cycle. This will make the operational cost higher for bot owners and reduce their harm.

3. **Distributed Pattern Matching over Streaming Time Series:** Matching a dictionary of patterns against a streaming time series is a challenging task. We presented an exact and distributed algorithm that can find all the matches and guarantee a bounded delay. Our algorithm assumes a constant data rate of the input. A useful direction for future work is to make our system work for variable rate streams. [58] presents a method to calculate the DTW distance between two sparse time series faster than $O(n^2)$. Another future work direction is to incorporate this method in to DisPatch. One may also extend our work by letting the user pick a maximum delay value longer than the longest pattern in the dictionary.

References

- [1] Apache Kafka. <http://kafka.apache.org>.
- [2] Apache Spark. <http://spark.apache.org>.
- [3] Apache Spark Streaming. <http://spark.apache.org/streaming>.
- [4] Benchmarking for DBSCAN and OPTICS. <http://elki.dbs.ifi.lmu.de/wiki/Benchmarking>.
- [5] CHB-MIT Scalp EEG Database. <https://physionet.org/pn6/chbmit>.
- [6] Elasticsearch: Store, Search, and Analyze. <https://www.elastic.co/guide/index.html>.
- [7] EPA dataset. <http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html>.
- [8] GrayLog. <https://www.graylog.org>.
- [9] Internet of Things (IoT). <http://www.cisco.com/web/solutions/trends/iot/overview.html>.
- [10] Log Management Explained. <https://www.loggly.com/log-management-explained/>.
- [11] LogEntries. <https://logentries.com/doc/>.
- [12] OpenStack. <https://en.wikipedia.org/wiki/OpenStack>.
- [13] OSSIM (Open Source Security Information Management). <https://en.wikipedia.org/wiki/OSSIM>.
- [14] Project's Repository. http://cs.unm.edu/~hamooni/papers/stream_pat.

- [15] Scaling the Facebook data warehouse. <https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb>.
- [16] SDSC dataset. <http://ita.ee.lbl.gov/html/contrib/SDSC-HTTP.html>.
- [17] Splunk. http://www.splunk.com/en_us/solutions/solution-areas/internet-of-things.html.
- [18] Sumo Logic. <https://www.sumologic.com/>.
- [19] Supporting web page. <http://cs.unm.edu/~hamooni/papers/handover>.
- [20] The twitter rules. <https://support.twitter.com/articles/18311>.
- [21] Twitter Statistics. <http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/10>.
- [22] Twitter streaming api. <https://dev.twitter.com/tags/streaming-api>.
- [23] E. and S. Kasetty. On the need for time series data mining benchmarks: a survey and empirical demonstration. *Data Mining and knowledge discovery*, 7(4):349–371, 2003.
- [24] L. Akoglu, R. Chandy, and C. Faloutsos. Opinion fraud detection in online reviews by network effects. In *Icwsn*, pages 2–11, 2013.
- [25] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod Record*, volume 28, pages 49–60. ACM, 1999.
- [26] I. Assent, I. Assent, M. Wichterich, M. Wichterich, R. Krieger, R. Krieger, H. Kremer, H. Kremer, T. Seidl, and T. Seidl. Anticipatory dtw for efficient similarity search in time series databases. *Work*, 2(1):826–837, 2009.
- [27] I. Assent, M. Wichterich, R. Krieger, H. Kremer, and T. Seidl. Anticipatory dtw for efficient similarity search in time series databases. *Journal Proceedings of the VLDB Endowment*, 2(1):826–837, 8 2009.
- [28] S. Asur and B. A. Huberman. Predicting the future with social media. In *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 492–499. IEEE, 8 2010.

- [29] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986. ACM, 2010.
- [30] J. Cheng, C. Danescu-Niculescu-Mizil, and J. Leskovec. Antisocial behavior in on-line discussion communities. In *Proceedings of ICWSM*, 2015.
- [31] Z. Chu, S. Gianvecchio, H. Wang, and S. Jajodia. Detecting automation of twitter accounts: Are you a human, bot, or cyborg? *IEEE Transactions on Dependable and Secure Computing*, 9(6):811–824, 11 2012.
- [32] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI’10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.
- [33] A. F. Costa, Y. Yamaguchi, A. J. M. Traina, C. Traina Jr., and C. Faloutsos. Rsc: Mining and modeling temporal activity in social media. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’15. ACM, 2015.
- [34] I. S. Dhillon. Co-clustering documents and words using bipartite co-clustering documents and words using bipartite spectral graph partitioning. In *Proc of 7th ACM SIGKDD Conf*, pages 269–274, 2001.
- [35] C. Ding and J. Zhou. Log-based indexing to improve web site search. In *SAC*, pages 829–833. ACM, 2007.
- [36] M. Eltahir and A. Dafa-Alla. Extracting knowledge from web server logs using web usage mining. In *Computing, Electrical and Electronics Engineering (ICCEEE)*, pages 413–417, Aug 2013.
- [37] C. Faloutsos and C. Faloutsos. Fast subsequence matching in time-series databases. *ACM SIGMOD Record*, 23(2):419–429, 1994.
- [38] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.
- [39] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.
- [40] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems*, 30(2):364–397, 2005.

- [41] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. In *Knowledge and Information Systems*, volume 7 of *VLDB '02*, pages 358–386, 2005.
- [42] E. J. Keogh, L. Wei, X. Xi, S.-h. Lee, and M. Vlachos. Lb_keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures. In *Vldb*, pages 882–893, 2006.
- [43] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012.
- [44] K. Lee, J. Caverlee, and S. Webb. Uncovering social spammers. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '10*, page 435. ACM Press, 7 2010.
- [45] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.
- [46] L.-w. H. Lehman, R. P. Adams, L. Mayaud, G. B. Moody, A. Malhotra, R. G. Mark, and S. Nemati. A Physiological Time Series Dynamics-Based Approach to Patient Monitoring and Outcome Prediction. *IEEE Journal of Biomedical and Health Informatics*, 19(3):1068–1076, 5 2015.
- [47] H. Li, A. Mukherjee, B. Liu, R. Kornfield, and S. Emery. Detecting campaign promoters on twitter using markov random fields. In *Data Mining (ICDM), 2014 IEEE International Conference on*, pages 290–299, 2014.
- [48] Li Wei, E. Keogh, H. Van Herle, and A. Mafra-Neto. Atomic Wedgie: Efficient Query Filtering for Streaming Times Series. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 490–497. IEEE.
- [49] S. Makonin, F. Popowich, L. Bartram, B. Gill, and I. V. Baji. Ampds: A public dataset for load disaggregation and eco-feedback research. In *2013 IEEE Electrical Power and Energy Conference, EPEC 2013*, pages 1–6, 2013.
- [50] E. Mariconti, J. Onaolapo, S. S. Ahmad, N. Nikiforou, M. Egele, N. Nikiforakis, and G. Stringhini. What's in a name? understanding profile name reuse on twitter. In *International World Wide Web Conference (WWW)*, 2017.
- [51] C. D. Martino, S. Jha, W. Kramer, Z. Kalbarczyk, and R. K. Iyer. Logdiver: A tool for measuring resilience of extreme-scale systems and applications. In *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, pages 11–18, 2015.

- [52] Y. Matsubara, Y. Sakurai, N. Ueda, and M. Yoshikawa. Fast and exact monitoring of co-evolving data streams. In *2014 IEEE International Conference on Data Mining*, pages 390–399. IEEE, 12 2014.
- [53] A. Mueen and N. Chavoshi. Enumeration of time series motifs of all lengths. *Knowl. Inf. Syst.*, 45(1):105–132, 2015.
- [54] A. Mueen and E. Keogh. 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*, number C in KDD '10, page 1089. ACM Press, 2010.
- [55] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, pages 473–484. SIAM, 2009.
- [56] A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 171–182, 2010.
- [57] A. Mueen, K. Viswanathan, C. Gupta, and E. Keogh. The fastest similarity search algorithm for time series subsequences under euclidean distance, 8 2015.
- [58] C. N. Mueen Abdullah, N. Abu-El-Rub, H. Hamooni, and A. Minnich. Fast Warping Distance for Sparse Time Series. In *Proceedings of the IEEE International Conference on Data Mining*, ICDM '16, 2016.
- [59] X. Ning and G. Jiang. HLAer: A system for heterogeneous log analysis, 2014. *SDM Workshop on Heterogeneous Learning*.
- [60] R. Rajachandrasekar, X. Besseron, and D. K. Panda. Monitoring and predicting hardware failures in hpc clusters with ftb-ipmi. In *IPDPSW Workshops*, pages 1136–1143. IEEE, 2012.
- [61] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 262–270, New York, NY, USA, 2012. ACM.
- [62] J. Ratkiewicz, M. Conover, M. Meiss, B. Goncalves, A. Flammini, and F. Menczer. Detecting and tracking political abuse in social media, 2011.
- [63] K. S. Reddy, G. P. S. Varma, and I. R. Babu. Preprocessing the web server logs: An illustrative approach for effective usage mining. *SIGSOFT Softw. Eng. Notes*, 37(3):1–5, May 2012.

- [64] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. Braid: Stream mining through group lag correlations. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, page 610, 2005.
- [65] S. R. Sarangi and K. Murthy. Dust: A generalized notion of similarity between uncertain time series smruti. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '10*, page 383. ACM Press, 7 2010.
- [66] D. Sart, A. Mueen, W. Najjar, V. Niennattrakul, and E. Keogh. Accelerating dynamic time warping subsequence search with gpus and fpgas. icdm 2010. In *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 1001–1006, 2010.
- [67] J. Shieh and E. Keogh. Isax: Disk-aware mining and indexing of massive time series datasets. *Data Mining and Knowledge Discovery*, 19(1):24–57, 2009.
- [68] J. Shieh and E. Keogh. Polishing the right apple: Anytime classification also benefits data streams with constant arrival times. In *Proceedings - IEEE International Conference on Data Mining, ICDM*, pages 461–470, 2010.
- [69] A. H. Shoeb and J. V. Guttag. Application of machine learning to epileptic seizure detection. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 975–982, 2010.
- [70] M. Shokoohi-Yekta, Y. Chen, B. Campana, B. Hu, J. Zakaria, and E. Keogh. Discovery of meaningful rules in time series. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*, pages 1085–1094. ACM Press, 8 2015.
- [71] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.
- [72] P. Sneath and R. Sokal. Unweighted pair group method with arithmetic mean. *Numerical Taxonomy*, pages 230–234, 1973.
- [73] V. Subrahmanian, A. Azaria, S. Durst, V. Kagan, A. Galstyan, K. Lerman, L. Zhu, E. Ferrara, A. Flammini, F. Menczer, R. Waltzman, A. Stevens, A. Dekhtyar, S. Gao, T. Hogg, F. Kooti, Y. Liu, O. Varol, P. Shiralkar, V. Vydiswaran, Q. Mei, and T. Huang. The darpa twitter bot challenge. *IEEE Computer (In press)*, 2016.
- [74] K. Thomas, C. Grier, D. Song, and V. Paxson. Suspended accounts in retrospect: an analysis of twitter spam. In *Proceedings of the 2011 ACM , IMC '11*, pages 243–258, 2011.

- [75] K. Thomas, F. Li, C. Grier, and V. Paxson. Consequences of connectivity: Characterizing account hijacking on twitter. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pages 489–500. ACM Press, 11 2014.
- [76] K. Thomas, V. Paxson, D. Mccoy, and C. Grier. Trafficking fraudulent accounts : The role of the underground market in twitter spam and abuse trafficking fraudulent accounts :. In *USENIX Security Symposium, SEC'13*, pages 195–210, 2013.
- [77] M. Vlachos, D. Gunopulos, and G. Das. Rotation invariant distance measures for trajectories. In *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '04*, page 707. ACM Press, 8 2004.
- [78] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multi-dimensional time-series with support for multiple distance measures. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03*, pages 216–225. ACM, 2003.
- [79] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow.*, 5(10):1004–1015, 2012.
- [80] D. H. Wang, A. Ketterlin, and P. Ganc. A global averaging method for dynamic time warping , with applications to clustering. *Pattern Recognition*, 44(3):678–693, 2010.
- [81] Wikipedia. Jaccard index — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Jaccard_index&oldid=688763411.
- [82] Wikipedia. DbSCAN — wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=DBSCAN&oldid=672504091>, 2015.
- [83] C. Xu, S. Chen, and J. Cheng. Network user interest pattern mining based on entropy clustering algorithm. In *Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 200–204, Sept 2015.
- [84] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *ICDM*, 2016.
- [85] C. E. Yoon, O. O'Reilly, K. J. Bergen, and G. C. Beroza. Earthquake detection through computationally efficient similarity search. *Science advances*, 1(11):e1501057, 12 2015.

- [86] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the HotCloud'10*.
- [87] M. A. Zaharia. An architecture for and fast and general data processing on large clusters. 2013.
- [88] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, volume 54 of *VLDB '02*, pages 358–369, 2002.
- [89] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh. 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.