Automated Runtime Data Analysis for System Reliability Management

HE, Pinjia

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
February 2018

Thesis Assessment Committee

Professor LUI Chi Shing, John (Chair)

Professor LYU Rung Tsong, Michael (Thesis Supervisor)

Professor LEUNG Kwong Sak (Committee Member)

Professor Cheung S.C. (External Examiner)

Abstract of thesis entitled:

Automated Runtime Data Analysis for System Reliability Management

Submitted by HE, Pinjia

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in February 2018

Runtime data are data generated by systems or programs during their execution. Typical runtime data include system logs and Quality-of-Service (QoS) values, which are widely employed by developers in various system reliability management tasks, such as anomaly detection, operational issues handling, performance prediction, etc. However, traditional reliability management methods become inefficient and error-prone because of the increase of modern system complexity and the rapid growth of runtime data volume. In this thesis, we propose automated data analysis methods to effectively utilize runtime data in reliability management tasks.

Firstly, we conduct an evaluation study on existing data-driven log parsing methods. Log parsing is the first step of many log based reliability management methods. In log parsing, the unstructured raw log messages are transformed into structured event sequences. Although log parsing has been widely studied, a comprehensive benchmarking and an open-source toolkit are lacking. We implement four representative log parsing methods and evaluate their performance in terms of accuracy, efficiency, and effectiveness on reliability management tasks. We obtain six insightful findings, and make these parsing methods open-source for reuse.

Secondly, we propose a parallel log parsing method for large-scale log data analysis. When system logs grow to a large scale, ex-

isting log parsing methods fail to complete in reasonable time, which makes log parsing the bottleneck of reliability management tasks. Because timely reliability management is important, an efficient log parsing method that can accurately parse large-scale log data is highly demanded. Our proposed parallel log parser POP employs specially designed heuristic rules and clustering algorithm. It is optimized on top of Spark, a large-scale data processing platform. Thus, POP can employ the computing power of computer clusters and handle large-scale logs efficiently.

Thirdly, we propose an online log parsing method to parse raw log messages in a streaming manner. Most of existing log parsing methods focus on offline, batch processing of logs. However, typical log collection process in modern systems is online, which make an online log parser more eligible than the offline ones. Besides, an online log parsing methods can keep updating the parsing model by newly collected log messages. By designing a fixed depth parse tree, our proposed online log parsing method can efficiently parse log messages in a streaming manner.

Fourthly, we propose an operational issues prioritization method based on hierarchical log clustering. Modern system developers handle issues reported by their users daily. To gain insights into the issues and find out the solutions, they often need to inspect tons of logs generated during system runtime. Our proposed method largely facilitates the operational issues handling process by clustering similar issues to the same group based on their corresponding log sequences, and recommending the largest issue groups to developers. Specifically, our method includes a coarse-grained clustering based on the event appearance matrix and a fine-grained clustering based on the event count matrix.

Lastly, we propose a QoS prediction method for Web service recommendation. A typical modern system based on Web services need to regularly switch its service components based on their QoS values (e.g., response time) to avoid potential system failure and

maintain system performance. However, it is difficult for service users to monitor the QoS values of all candidate services. To predict these QoS values accurately, our proposed QoS prediction method utilizes matrix factorization on existing sparse QoS values. The location of service providers and users is encoded in the matrix factorization model to improve prediction accuracy.

In summary, this thesis targets at the design of data-driven techniques on system runtime data to automate labor-intensive reliability management tasks. Extensive experiments on real-world datasets determine the effectiveness of our proposed methods.

論文題目：系統可靠性管理的運行時數據自動化分析方法

作者　　：贺品嘉

學校　　：香港中文大學

學系　　：計算機科學與工程學系

修讀學位：哲學博士

摘要　　：

運行時數據由系統或程序在執行過程中生成。典型的運行時數據是日誌和服務質量數據。它們被開發者廣泛采用於各種系統可靠性管理的任務，如異常檢測、業務問題處理、性能預測等。然而，因為現代系統越來越復雜，且運行時數據的數據量快速增長，傳統的可靠性管理方法效率低且容易出錯。在本論文中，我們提出了自動化數據分析方法，以有效地利用運行時數據來完成可靠性管理的任務。

首先，我們對以數據作為驅動的日誌解析方法進行了評價研究。日誌解析是許多基於日誌的可靠性管理方法的第壹步，其作用是將非結構化的原始日誌轉換為結構化的日誌事件序列。雖然日誌解析被廣泛研究，但學界缺乏對不同日誌解析方法的全面基準評測和日誌解析的開源工具包。我們實現了四種有代表性的日誌解析方法，並在準確性、效率和對可靠

性管理任務的有效性方面評估了它們的性能。我們得出了六個有價值的結論，並將這四種日誌解析方法開源以方便其被重復使用。

其次，我們提出了壹種用於大規模日誌分析的並行化日誌解析方法。當系統日誌規模較大時，現有的日誌解析方法無法在合理的時間內完成，這使得日誌解析成為整個可靠性管理任務的性能瓶頸。因為快速及時的可靠性管理非常重要，我們需要壹種高效的日誌解析方法，用於準確地解析大規模日誌數據。我們提出的並行日誌解析算法 POP 包含了專門設計的啟發式規則和聚類算法。在實現上，POP 針對大型數據處理平臺 Spark 作了特殊的優化。因此，POP 可以利用計算機集群的計算能力，有效地處理大規模日誌。

再次，我們提出了壹種在線日誌解析方法，以流處理的方式解析原始日誌。大多數現有的日誌解析方法都采用離線、批處理的方式。然而，現代系統中典型的日誌收集過程是在線的，這使得在線日誌解析器比離線日誌解析器更能融入現代系統中。且在線日誌解析器可以通過新收集的日誌來更新解析模型。我們提出了壹種在線日誌解析方法，通過壹個特殊設計的固定深度的解析樹，其能以流處理的方式高效地解析系統日誌。

然後，我們提出了壹種基於分層日誌聚類的業務問題優先級計算的方法。現代系統操作員每天處理用戶上報的業務問題。為了深入了解問題並找出解決方案，他們常常需要檢查大量系統運行時生成的日誌。我們提出的方法分析與業務問題相關聯的日誌序列。日誌序列相似的業務問題將被聚到同壹類中。統計每壹類所含業務問題的數量後，我們並將業務問題數量最多的幾個類推薦給操作員。這大大加速了業務問題處理的速度。具體而言，我們的方法包括基於事件發生矩陣的粗粒度聚類和基於事件頻率矩陣的細粒度聚類。

最後，我們提出了壹種服務質量的預測方法來作 Web 服務推薦。典型的基於 Web 服務的現代系統需要根據其組件的服務質量（如響應時間）定期切換其服務組件，以避免潛在的系統故障和維護系統性能。然而，用戶很難監控所有備選服務的服務質量。為了準確預測這些服務質量，我們提出的服務質量預測方法在現有的稀疏的服務質量記錄上使用矩陣分解模型。我們在矩陣分解模型中對服務提供者和使用者的位置進行了編碼，並以此提高了預測精度。

綜上所述，本論文的目標是設計以數據作為驅動的技術，對系統運行時數據進行分析，並自動化勞動密集型的系統可靠性管理任務。大量基於真實數據的實驗驗證了我們提出的方法的有效性。

Acknowledgement

I feel highly privileged to take this opportunity to express my sincere gratitude to the people who have been instrumental and helpful on my way to pursuing my PhD degree.

First and foremost, I would like to thank my supervisor, Prof. Michael R. Lyu, for his kind supervision of my PhD study at CUHK. He has provided inspiring guidance and incredible help on every aspect of my research. From choosing a research topic to working on a project, from technical writing to paper presentation, I have learnt so much from him not only on knowledge but also on attitude in doing research. Besides, he gives me a lot of freedom to select my research topic and study necessary techniques. I will always be grateful to his advice, encouragement and support at all levels.

I am grateful to my thesis assessment committee members, Prof. Chi Shing Lui and Prof. Kwong Sak Leung, for their constructive comments and valuable suggestions to this thesis and all my term reports. Great thanks to Prof. Shing-Chi Cheung from The Hong Kong University of Science and Technology who kindly served as the external examiner for this thesis.

I would like to thank my oversea supervisor, Prof. Tao Xie, for his support of my visit to University of Illinois Urbana-Champaign. During this visit, Prof. Xie has provided insightful ideas and constructive feedback to my research. I also thank Dengfeng Li, Chiao Hsieh, Mingming Zhang, Wing Lam, Siwakorn Srisakaokul, Wei Yang, and Zhengkai Wu for the short but wonderful memories in UIUC.

I would like to thank Prof. Hongyu Zhang, my mentor during the internship at Microsoft Research Asia. I also thank friends met in MSRA, Youshan Miao, Yuanwei Lu, Shaowei Wang, Heng Lin, Dongpo Zhao, and Luwei Cheng, for their kindness and help.

I would like to thank my life-long friends, Shiqian Chen, Kunhong Xu, Rong Yuan, and Yiwei Liu, for their trust and support.

I thank Zibin Zheng, Jieming Zhu, Shilin He, Jian Li, and Jianlong Xu, for their valuable guidance and contribution to the research work in this thesis. I am also thankful to my other groupmates, Yangfan Zhou, Haiqin Yang, Yilei Zhang, Guang Ling, Chen Cheng, Yu Kang, Tong Zhao, Junjie Hu, Hongyi Zhang, Shenglin Zhao, Xixian Chen, Yuxin Su, Cuiyun Gao, Hui Xu, Jichuan Zeng, Xiaotian Yu, Pengpeng Liu, and Yue Wang, who gave me encouragement and kind help.

Last but not least, I would like to thank my parents. Without their deep love and constant support, this thesis would never have been completed.

To my family.

Contents

Abstract **i**

Acknowledgement **vii**

1 Introduction **1**
1.1 Overview 1
1.2 Thesis Contributions 5
1.3 Thesis Organization 9

2 Background Review **12**
2.1 System Runtime Data 12
2.2 Log Parsing 14
2.2.1 Problem Description 14
2.2.2 Literature Review 16
2.3 Operational Issues Prioritization via Log Events . . . 25
2.3.1 Problem Description 26
2.3.2 Literature Review 27
2.4 QoS Prediction via Limited QoS Values in Logs . . . 29
2.4.1 Problem Description 30
2.4.2 Literature Review 30

3 Evaluation Study of Log Parsing and Its Use in Log Mining **33**
3.1 Introduction 33
3.2 Log Parsing Overview 37

3.2.1 Overview of Log Parsing 37
3.2.2 Existing Log Parsing Methods 38
3.2.3 Tool Implementation 40
3.3 Log Mining 41
3.3.1 Overview of Log Mining 41
3.3.2 System Anomaly Detection 42
3.4 Evaluation Study 44
3.4.1 Study Methodology 44
3.4.2 RQ1: Accuracy of Log Parsing Methods . . . 46
3.4.3 RQ2: Efficiency of Log Parsing Methods . . 48
3.4.4 RQ3: Effectiveness of Log Parsing Methods on Log Mining 51
3.5 Discussions 53
3.6 Summary 54

4 Parallel Log Parsing for Large-Scale Log Data 55
4.1 Introduction 55
4.2 Parallel Log Parsing (POP) 59
4.2.1 Step 1: Preprocess by Domain Knowledge . 61
4.2.2 Step 2: Partition by Log Message Length . . 61
4.2.3 Step 3: Recursively Partition by Token Position 62
4.2.4 Step 4: Generate Log Events 65
4.2.5 Step 5: Merge Groups by Log Event 65
4.2.6 Implementation 66
4.3 Evaluation 69
4.3.1 Study Methodology 69
4.3.2 Accuracy of POP 72
4.3.3 Efficiency of POP 75
4.3.4 Effectiveness of POP on Log Mining: A Case Study 83
4.3.5 Parameter Sensitivity 85
4.3.6 Observations 91

4.4 Discussions 92
4.5 Summary 93

5 Online Log Parsing via Fixed Depth Tree 94
5.1 Introduction 94
5.2 Methodology 98
5.2.1 Overall Tree Structure 99
5.2.2 Step 1: Preprocess by Domain Knowledge . 100
5.2.3 Step 2: Search by Log Message Length . . . 100
5.2.4 Step 3: Search by Preceding Tokens 101
5.2.5 Step 4: Search by Token Similarity 102
5.2.6 Step 5: Update the Parse Tree 102
5.3 Evaluation 104
5.3.1 Experimental Settings 104
5.3.2 Accuracy of Drain 106
5.3.3 Efficiency of Drain 108
5.3.4 Effectiveness of Drain on Real-World Anomaly Detection Task 110
5.4 Summary 114

6 Prioritizing Operational Issues via Hierarchical Log Clustering 115
6.1 Introduction 115
6.2 POI Framework 118
6.2.1 Raw Logs 118
6.2.2 Log Parsing 118
6.2.3 Vector Generating 119
6.2.4 Log Clustering 120
6.2.5 Issue Prioritization 120
6.3 Methodology 121
6.3.1 Inverse Cardinality 121
6.3.2 Clustering by Log Event Appearance 122
6.3.3 Clustering by Log Event Count 124

6.4 Evaluation 127
6.4.1 Experiment Settings 127
6.4.2 Evaluation of Clustering Algorithm 128
6.4.3 Evaluation of Coverage Ability 130
6.5 Summary 133

7 Location-Based Web Services QoS Prediction via Historical QoS Logs 134
7.1 Introduction 134
7.2 Framework of Web Service Recommendation 138
7.3 Hierarchical Matrix Factorization 140
7.3.1 Overview 140
7.3.2 Users and Services Clustering 141
7.3.3 Local Matrix Factorization 143
7.3.4 Global Matrix Factorization 144
7.4 Experiments 146
7.4.1 Dataset Description 146
7.4.2 Metrics 146
7.4.3 Comparison 147
7.4.4 Impact of α 150
7.4.5 Impact of Dimensionality 152
7.4.6 Impact of Matrix Density 155
7.5 Summary 155

8 Conclusion and Future Work 156
8.1 Conclusion 156
8.2 Future Work 158

A List of Publications 162

Bibliography 165

List of Figures

1.1 Hadoop Ecosystem (modified from figure in [16]) . . . 3
1.2 An Overview of Automated Runtime Data Analysis for System Reliability Management 4

2.1 An Illustrated Example of Log Parsing 15
2.2 An Illustrated Example of User-Service Matrix . . . 30

3.1 Overview of Log Parsing 38
3.2 Running Time of Log Parsing Methods on Datasets in Different Size 49
3.3 Parsing Accuracy on Datasets in Different Size . . . 50

4.1 Overview of Log Analysis 56
4.2 Overview of Log Parsing 60
4.3 Proxifier Log Samples 61
4.4 An Example of *AT*, *RT* Calculation 63
4.5 Overview of POP Implementation 68
4.6 Parsing Accuracy on Datasets in Different Size . . . 77
4.7 Running Time of Log Parsing Methods on Datasets in Different Size 78
4.8 Running Time on Synthetic Datasets 82
4.9 Example Log Group 86
4.10 Impact of GS 87
4.11 Impact of splitRel 89
4.12 Impact of splitRel (splitAbs=0) 90
4.13 Impact of maxDistance 91

5.1 Structure of Parse Tree in Drain ($depth$ = 3) 98
5.2 Parse Tree Update Example ($depth$ = 4) 103
5.3 Running Time of Log Parsing Methods on Data Sets in Different Size 111

6.1 Overview of Our Approach 119
6.2 A Raw Log Message of Hadoop File System (HDFS) 119
6.3 An Example of Clustering by Log Event Appearance 122
6.4 An Example of Clustering by Log Event Count . . . 125
6.5 Evalution of Clustering Ability 129
6.6 Coverage Ability Curve of Different Methods 131

7.1 Web Services Invocation Scenario 136
7.2 Framework of Hierarchical Web Service Recommendation System 138
7.3 An Example of QoS Prediction by Hierarchical Matrix Factorization ($\alpha = 0.8$) 139
7.4 Impact of α on MAE 150
7.5 Impact of α on NMAE 151
7.6 Impact of Dimensionality on MAE 152
7.7 Impact of Dimensionality on NMAE 153
7.8 Impact of Matrix Density 154

List of Tables

3.1 Summary of Our System Log Datasets 45
3.2 Parsing Accuracy of Log Parsing Methods (Raw / Preprocessed) 46
3.3 Anomaly Detection with Different Log Parsing Methods (16,838 Anomalies) 51

4.1 Summary of Our System Log Datasets 71
4.2 Parsing Accuracy of Log Parsing Methods (Raw / Preprocessed) 74
4.3 Log Size of Sample Datasets 75
4.4 Parsing Accuracy of POP on Sample Datasets in Table 4.3 with parameters tuned on 2k datasets . . . 76
4.5 Running Time of POP (Sec) on Sample Datasets in Table 4.3 81
4.6 Anomaly Detection with Different Log Parsing Methods (16,838 Anomalies) 83

5.1 Summary of Log Data Sets 104
5.2 Parameter Setting of Drain 106
5.3 Parsing Accuracy of Log Parsing Methods 107
5.4 Running Time (Sec) of Log Parsing Methods 108
5.5 Log Size of Sample Datasets for Efficiency Experiments 109
5.6 Anomaly Detection with Different Log Parsing Methods (16,838 True Anomalies) 113

7.1 Parameters 148

7.2 Value of α 148
7.3 Performance Comparison (MAE) 149
7.4 Performance Comparison (NMAE) 149

Chapter 1

Introduction

This thesis presents our research towards automated runtime data analysis for system reliability management, which is currently an important field of study and practice in software operation and maintenance. We provide a brief overview of the research problems under study in Section 1.1, and highlight the main contributions of this thesis in Section 1.2. Section 1.3 outlines the thesis structure.

1.1 Overview

Modern systems play an important role in our daily life. For example, distributed systems, which are typical modern systems, have become the core building block of the IT industry, powering various applications such as e-commerce platforms, instant messaging software, online banking systems, etc. These distributed systems usually work in a 24×7 manner serving millions of users all over the world. In practice, a startup company can rent servers from distributed system providers to host its own application or service (e.g., online notebook). Thus, any non-trivial downtime of these systems can lead to enormous revenue loss for both service providers and service users. In February 2017, Amazon Web Services (AWS) has encountered an outage that took down a bunch of large online services (e.g., Trello, Quora, IFTTT) for several hours [5], which

impacted both service providers (i.e., AWS) and service users (e.g., Quora). Therefore, a system reliability management framework is highly in demand.

To manage system reliability, developers mainly rely on system runtime data, which are generated during system execution. Typical runtime data include user information, system logs, Quality-of-Service (QoS) values, etc. Specifically, user information describes the profile of users, which can be employed to search historical user behaviors or issues. System logs are free-form text printed by logging statements in source codes, which are widely utilized in reliability management tasks, such as anomaly detection [35, 42, 94], problem identification [56, 81], problem diagnosis [72, 98], etc. QoS values, which can be extracted from logs, are used in various online service systems to evaluate system reliability.

Although system runtime data are of great use in system reliability management tasks, traditional methods that mainly rely on manual analysis become inefficient and error-prone for modern systems, because modern systems are getting much more complex in structure and larger in scale. For example, as illustrated in Fig. 1.1, Hadoop ecosystem contains various modern systems, including Hadoop Distributed File System (HDFS), Spark, Zookeeper, etc. Different systems provide different functionalities, such as data storage, data intelligence, monitoring, etc. It is therefore typical that these systems often need to coordinate with each other to accomplish a task, which leads to the complexity of system reliability management. Besides, each of these systems can employ various existing open-source components, so the structure of these systems themselves can be very complex. Usually, modern systems are also large-scale, which are widely used by big companies (e.g., Amazon EC2 [1], Google Cloud [4], and Microsoft Azure [6]). These large-scale systems generate tons of runtime data every day. For example, cloud system in Alibaba produces about 30-50 gigabytes (around 120-200 million lines) of system logs per hour.

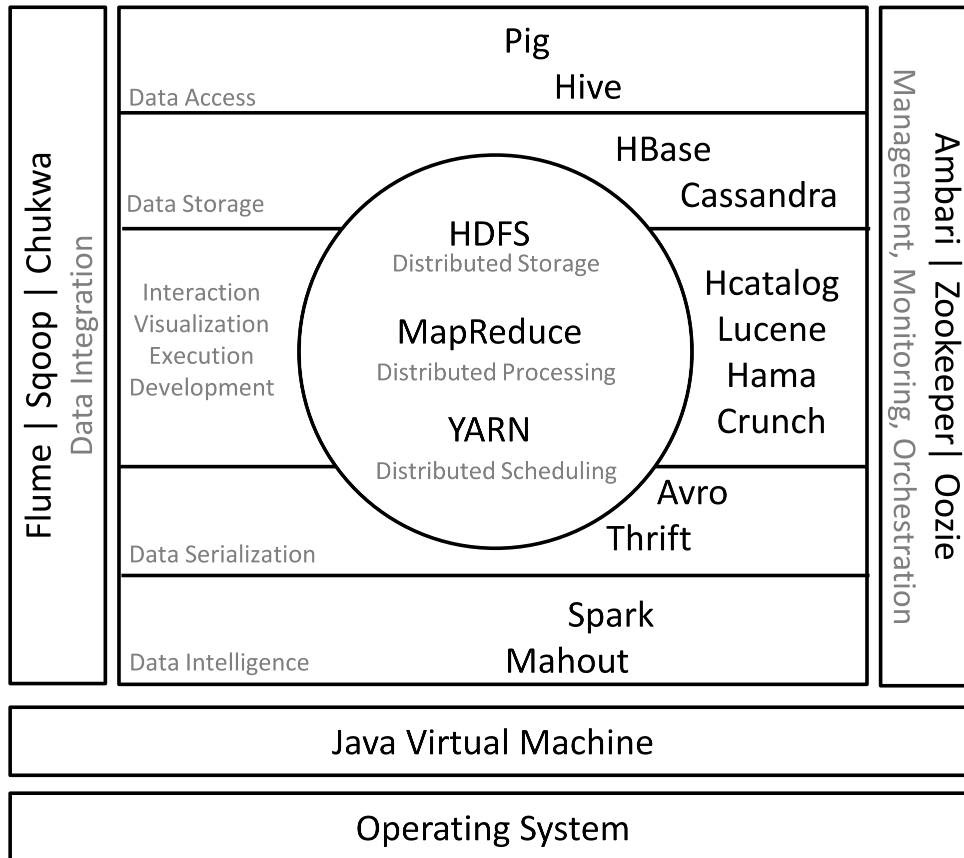


Figure 1.1: Hadoop Ecosystem (modified from figure in [16])

Thus, it is challenging to employ these system runtime data effectively and efficiently. First, since modern systems are complex and often employ various existing open-source components, the runtime data generated are often in various formats. For example, logs are printed by logging statements written by different developers or even different logging frameworks, so logs are often free-form text messages. Manually inspecting these free-form logs are prohibitive. Moreover, these free-form logs cannot be directly input to most of automated log analysis frameworks, which require structured data (e.g., a matrix) as input. Second, due to the large amount of users and complexity of the systems, developers of modern systems receive tons of operational issues. To address these

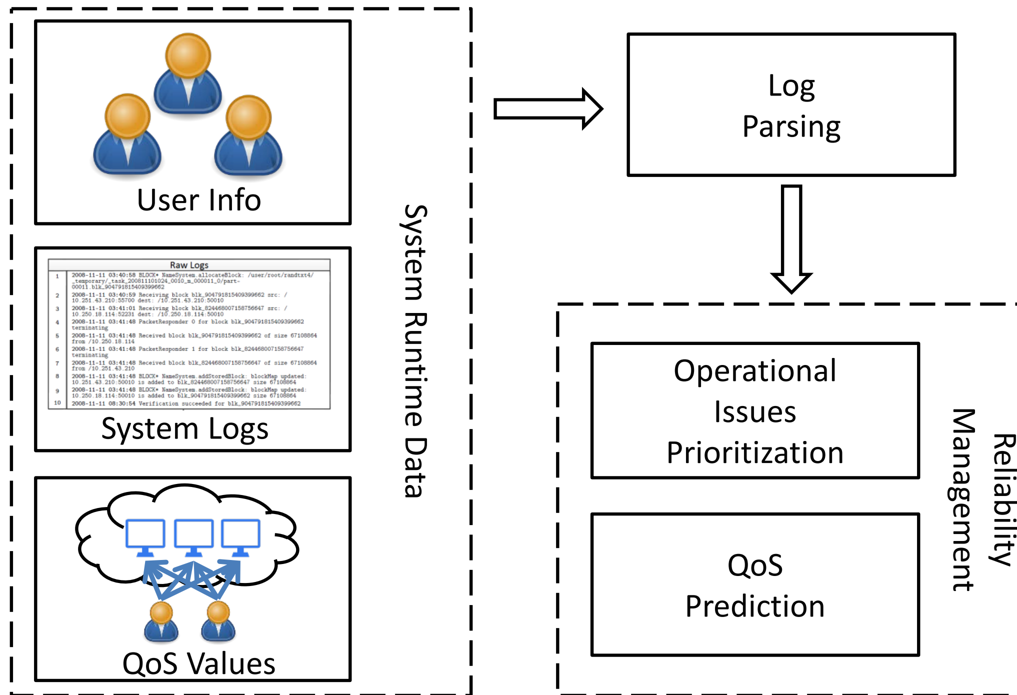


Figure 1.2: An Overview of Automated Runtime Data Analysis for System Reliability Management

issues, developers need to inspect the corresponding runtime data. Given that many operational issues in modern systems are redundant [56, 81], traditional methods that rely on manual analysis becomes inefficient and error-prone. Third, modern systems employ QoS values to evaluate the reliability of their provided services. A service often provides different QoS (e.g., response time) to different service because of the underlying network. To provide the most suitable services to users, developers need to monitor various QoS values. However, as the number of users rapidly grows, it is impossible to monitor the QoS values of services received by all users. Thus, it is highly desired to predict QoS values based on limited QoS value data.

In this context, automated runtime data analysis has recently emerged as a promising solution for modern system reliability management. With the help of data mining techniques (e.g., clustering

algorithms) and large-scale data processing platforms (e.g., Apache Spark [11]), traditional reliability management methods that are labor-intensive and error-prone, can be automatically handled by data analysis techniques. Fig. 1.2 illustrates the overview of the automated runtime data analysis framework. System runtime data has a wealth of information that can be utilized by developers. After log parsing, which transforms runtime data to structured data, data mining techniques can be employed in various system reliability management tasks, such as operational issues handling and QoS prediction. As a result, the objective of this thesis is to build a system reliability management framework, where runtime data are analyzed automatically to obtain insights and assist developers in various reliability management tasks. The research of this thesis comprises five parts. In the first three parts, we focus on the study of log parsing. Specifically, in the first part, we conduct an evaluation study on existing log parsers and publicly release our implementation as a reusable toolkit. In the second part, we propose a parallel log parser built on top of Apache Spark. In the third part, we design an online log parser to parse logs in a streaming manner. In the fourth part, we propose an operational issues prioritization method based on hierarchical log clustering. Finally, in the fifth part, we design a QoS prediction method, which employs limited QoS values extracted from logs.

1.2 Thesis Contributions

In this thesis, we make contributions to system reliability management in the following ways:

1. **Evaluation study of log parsing**

   Logs, which record runtime information of modern systems, are widely utilized by developers (and operators) in system development and maintenance. Due to the ever-increasing

size of logs, data mining models are often adopted to help developers extract system behavior information. However, before feeding logs into data mining models, logs need to be parsed by a log parser because of their unstructured format. Although log parsing has been widely studied in recent years, users are still unaware of the advantages of different log parsers nor the impact of them on subsequent log mining tasks. Thus they often re-implement or even re-design a new log parser, which would be time-consuming yet redundant. To address this issue, we study four log parsers and package them into a toolkit to allow their reuse [39]. In addition, we obtain six insightful findings by evaluating the performance of the log parsers on five datasets with over ten million raw log messages, while their effectiveness on a real-world log mining task has been thoroughly examined.

2. **Parallel log parsing for large-scale log data**

   Although the overall accuracy of existing parsers is high, they are not robust across all datasets. When logs grow to a large scale (e.g., 200 million log messages), which is common in practice, these parsers are not efficient enough to handle such data on a single computer. To address the above limitations, we design and implement a parallel log parser (namely POP) [40] on top of Spark, a large-scale data processing platform. POP employs specially designed heuristic rules and hierarchical clustering algorithm. Comprehensive experiments have been conducted to evaluate POP on both synthetic and real-world datasets. The evaluation results demonstrate the capability of POP in terms of accuracy, efficiency, and effectiveness on subsequent log mining tasks. Specifically, POP achieves the highest parsing accuracy on all real-world datasets compared with the existing methods. Besides, POP can parse our synthetic HDFS (Hadoop Distributed File System) dataset, which

contains 200 million lines of raw log messages, in 7 mins.

3. **Online log parsing**

   Most of the existing log parsing methods (e.g., parsers in Chapter 3 and Chapter 4) focus on offline, batch processing of logs. However, as the volume of logs increases rapidly, model training of offline log parsing methods, which employs all existing logs after log collection, becomes time-consuming. To address this problem, we propose an online log parsing method, namely Drain [41], that can parse logs in a streaming and timely manner. To accelerate the parsing process, Drain uses a fixed depth parse tree, which encodes specially designed rules for parsing. We evaluate Drain on five real-world log datasets with more than 10 million raw log messages. The experimental results show that Drain has the highest accuracy on four datasets, and comparable accuracy on the remaining one. Besides, Drain obtains 51.85%$\sim$81.47% improvement in running time compared with the state-of-the-art online parser. We also conduct a case study on an anomaly detection task using Drain in the parsing step, which determines the effectiveness of Drain in log analysis.

4. **Prioritizing operational issues via hierarchical log clustering**

   As modern systems become complex and large-scale, operators may need to handle tons of operational issue every day. An operational issue is a system problem reported by users at runtime, accompanied with user ID, configuration information, log sequences, etc. Typically, operators need to first inspect the accompanied log sequences and then identify the problem encountered. However, it is prohibitive and error-prone for operators to manually handle a large number of issues. To address this problem, we propose POI to facilitate the issue handling process. POI clusters similar operational

issues into groups based on the corresponding log sequences. Then POI prioritizes the issue groups according to the number of issues they contain. With POI, operators can focus on the representative issue clusters instead of the large-scale log sequences. Extensive experiments have been conducted on a real-world dataset with 16,838 issues. Compared with the existing methods, POP achieves the highest F-measure and the best issue coverage.

5. **Location-based Web services QoS prediction via historical QoS logs**

   Developers, who design systems based on service-oriented architecture (SOA), combine several Web services for system development. As the number of functionally-equivalent services increases rapidly, developers need to select the most suitable Web services based on their QoS values (e.g., response time). However, in reality, the QoS values are not easy to obtain, because developers only know historical invocations, whose QoS values are recorded in service logs. To tackle this challenge, we design a location-based hierarchical matrix factorization (HMF) method to perform personalized QoS prediction. We cluster developers (i.e., users) and services into several user-service groups based on their location. To better characterize the QoS data, our HMF model is trained in a hierarchical way by using the global QoS matrix as well as several location-based local QoS matrices generated from user-service groups. Then the missing QoS values are predicted by compactly combining the results from local matrix factorization and global matrix factorization. Comprehensive experiments are conducted on a real-world Web service QoS dataset with 1,974,675 Web service invocation records. The experimental results show that our HMF method achieves higher prediction accuracy than the state-of-the-art methods.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2**

  In this chapter, we review some background knowledge and related work on automated runtime data analysis for system reliability management. First, we briefly introduce system runtime data, with focus on its usage in modern system reliability management, typical runtime data, system logs, and log management. Then we review representative log parsing methods, including both offline log parsers and online log parsers. After that, we introduce typical log mining methods that enhance system reliability, with focus on representative operational issues handling techniques. Finally, we review QoS prediction approaches that predict missing QoS values based on limited QoS values from QoS logs.

- **Chapter 3**

  This chapter presents an evaluation study on four representative log parsing methods and their use in log mining, with the aim to provide researchers and developers with a benchmark and a reusable parsing toolkit. We conclude with six insightful findings with respect to their accuracy, efficiency, and effectiveness on the subsequent log mining tasks. The source codes of the four evaluated log parser have been released for reproduction. More specifically, in Section 3.1, we introduce log parsing and the motivation of our evaluation study. Section 3.2 reviews four representative log parsing methods (i.e., SLCT [88], IPLoM [64], LKE [35], LogSig [85]). Section 3.3 reviews recent studies on log mining with a detailed example of anomaly detection. The evaluation study results are reported in Section 3.4. We discuss some limitations in Section 3.5. Finally, we conclude this chapter in Section 3.6.

- **Chapter 4**

  In this chapter, we present a parallel log parsing framework, namely POP. POP employs specially designed heuristic rules and hierarchical clustering algorithm. To achieve parallelization, we optimize POP on top of Spark, a large-scale data processing platform. To the best of our knowledge, POP is the first parallel log parsing framework. More specifically, Section 4.1 introduces log parsing and our motivation for parallel log parsing. Section 4.2 introduces our parallel log parsing method, which includes five steps, and its implementation details on Spark. The evaluation results are reported in Section 4.3. We discuss practical usage of POP in Section 4.4. We summarize this chapter in Section 4.5.

- **Chapter 5**

  This chapter presents an online log parsing framework, namely Drain, which can parse logs in a streaming manner. The core technique of Drain is a fixed depth tree, which encodes different specially designed heuristic parsing rules. When a log message arrives, Drain employs the fixed depth tree to search the most suitable log group for parsing, and update the rules accordingly. More specifically, Section 5.1 presents the overview of log parsing and explains the motivation of designing an online log parser. Section 5.2 describes the five steps of our online log parsing method, Drain. The first four steps search a correct log group for the current log message, while the fifth step updates the fixed depth tree accordingly. We evaluate the performance of Drain in Section 5.3. Finally, we conclude this chapter in Section 5.4.

- **Chapter 6**

  In this chapter, we present an operational issue prioritizing framework, namely POI. The core idea of POI is to cluster

similar issues into groups based on the corresponding log sequences, and then prioritize these issue groups according to the number of issues inside. More specifically, Section 6.1 introduces the motivation of operational issues prioritization. Section 6.2 presents some background knowledge and the prioritizing framework. Then Section 6.3 introduces our hierarchical log clustering algorithm in detail. Experimental results are explained in Section 6.4, and finally, we conclude this chapter in Section 6.5.

- **Chapter 7**

  This chapter presents a Web service QoS values prediction method, namely HMF, based on historical QoS logs. A developer usually has employed only a few Web services, so only limited QoS values are available in historical QoS logs. The core idea of HMF is to predict QoS values by learning from similar developers and services. More specifically, Section 7.1 explains the motivation of QoS prediction based on historical QoS logs. Section 7.2 describes the framework of our QoS prediction. Our proposed hierarchical matrix factorization model is explained in detail in Section 7.3. Section 7.4 presents experiments and discusses the experiment results. Finally, we conclude this chapter in Section 7.5.

- **Chapter 8**

  The last chapter summarizes this thesis and provides some future directions that deserve for further exploration.

To make each chapter self-contained, we may briefly reiterate the critical contents, such as model definitions and motivations, in some chapters.

□ **End of chapter.**

Chapter 2

Background Review

This chapter briefly reviews some background knowledge and related work of our research. First, we provide background knowledge about system runtime data, especially system logs. Then in the following three subsections, we explain the three main problems studied in this thesis, including log parsing in Section 2.2, log mining based on log events in Section 2.3, and log mining based on logged variables Section 2.4. In each subsection, we first introduce the research field. Then we present the problem description, including the concrete problem studied, problem input, problem output, and its usage in system reliability management. Finally, we review related literature.

2.1 System Runtime Data

Runtime data are data generated by systems or programs during their execution. Compared with traditional systems, modern systems are more complex and large-scale, especially online service systems. These systems often work in a 24×7 manner to serve millions of users all over the world. For example, a startup company may put part of its services on an online service system. Any non-trivial downtime of these systems can lead to revenue loss of both service providers and services users. However, due to the complexity and

large scale, modern systems inevitably encounter failures (e.g., node failures). Thus, system reliability management methods are highly in demand. Specifically, developers need to monitor the system status at runtime, spot potential anomalies, and handle operational issues efficiently. To achieve these goals, typically, developers rely on system runtime data, because they are often the only data available for analysis.

Modern systems have a wide range of runtime data, including user information, system logs, QoS values, etc. User information profiles a user, for example, the historical user behaviors of a specific user. System logs are free-form text generated by the system at runtime to record system operations. QoS values represent Quality-of-Service values, which measure the nonfunctional aspect of the system. Among all these runtime data, system logs are the most common and often the most important one, because when a user reports an operational issue, the corresponding system logs will be sent to the developers accordingly. Besides, QoS values are often extracted from the variable part of QoS logs.

A log message is printed by a logging statement written by developers. A typical log message contains two parts: constant part and variable part. Constant part contains tokens that describe the system behavior, while variable part records runtime variables in the program. Example logs are illustrated in Fig. 2.1. The raw logs in this figure are extracted from logs generated by Hadoop Distributed File System (HDFS). In this thesis, we call the constant part of a log message "a log event", and different log messages has their own "log events". We also use "log event" and "log event type" interchangeable in this thesis. For example, Fig. 2.1 demonstrates six log event types.

With the prevalence of distributed systems and cloud computing, log management becomes a challenging problem because of security assurance requirements and the huge volume of log data. Hong et al. [43] design a framework to sanitize search logs with strong

privacy guarantee and sufficiently retained utility. Zawoad et al. [100] propose a scheme to reveal cloud users' logs for forensics investigation while preserving their confidentiality. Meanwhile, to assist log analysts in searching, filtering, analyzing, and visualizing a mountain of logs, some promising solutions, such as commercial Splunk [19], and open-source Logstash[18], Kibana [17], have been provided. These solutions provide many plugins/tools for monitoring and analyzing popular system logs (e.g., TCP/UDP, Apache Kafka) and present stunning visualization effects.

2.2 Log Parsing

In this section, we introduce the background knowledge of log parsing. Log parsing aims at transforming unstructured log messages into structured log events. System runtime log messages are unstructured, because these log messages are free-form texts printed by logging statements written by developers to describe system operations and runtime system status. However, many log mining algorithms that automatically mine insights from system logs require structured input (e.g., a matrix). Thus, we need a log parser to automatically generate structured log events based on the unstructured system runtime logs. Then structured input, such as a matrix, can be easily calculated based on log events.

2.2.1 Problem Description

Essentially, log parsing is a clustering problem, where log messages with the same log event type are clustered into one log group. After clustering, we can obtain a number of log groups, say N groups. Then log parser will generate N log event types by extracting the log event of each log group. With the generated log events, log parser can match each log message with a log event type according to its corresponding log group.

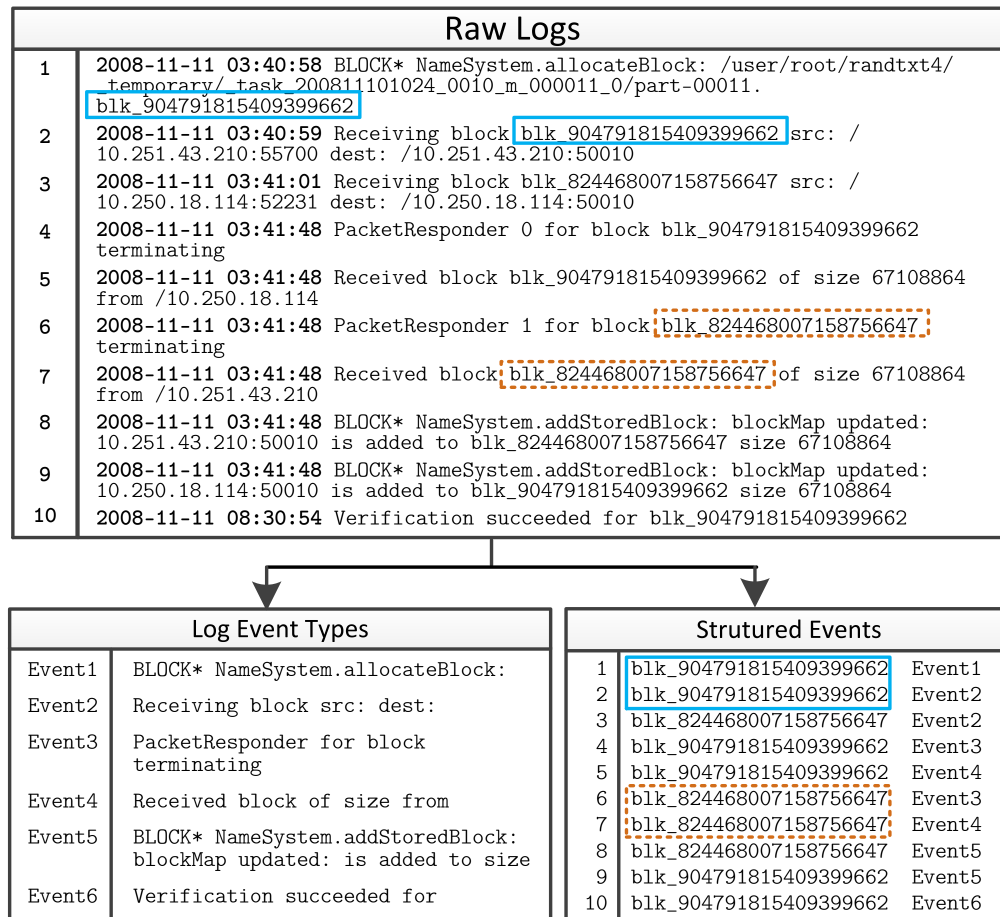


Figure 2.1: An Illustrated Example of Log Parsing

Technically, log parsing aims at distinguishing the constants and variables in every log message. The input of a log parser is a sequence of log messages collected in system runtime. Typically, the log messages contain timestamp, verbosity level, and raw message content. The output of a log parser is a sequence of log events. For example, as illustrated in Figure 2.1, there are 10 unstructured log messages. These log messages are collected from Hadoop Distributed File System. They are the input of a log parser. After log parsing, the parser generates two parts. The first part contains log event types, which are all the system operations mined from the unstructured logs. The second part contains a list of log events,

which are represented by event IDs. In practice, developers can also extract fields of interest and present them in the parsing output, such as the block ID in Figure 2.1. Developers can generate vectors for different blocks based on the block IDs and log events. Note that all the log parsing methods proposed in this thesis only rely on the log messages as input, and does not require any source code information. With project source code, we can find out all possible log event type by statically analyzing the source code. However, source codes are often inaccessible in practice, for example, systems that use third-party libraries or online Web services.

2.2.2 Literature Review

Recently, log parsing has been widely studied. Typical log parsing methods can be either offline or online. Offline log parsers mine the log event types from historical log data in batch mode, while online log parsers work in a streaming manner. For both offline and online log parsers, we can employ heuristic rules based technique or clustering algorithms. In the next subsections, we will introduce both heuristic rules based offline log parsers and clustering based offline log parsers. For online log parser, we focus on heuristic rules based online methods, because currently there is a lack of clustering based online log parsers. Finally, we introduce some other log parsing methods, such as source code based log parsing methods.

Offline Log Parsing Based on Heuristic Rules

Offline log parsing methods parse log messages in batch mode. Specifically, developers collect historical system logs, and input a batch of logs into the offline log parser. Then the structured events can be directly input to subsequent log mining models. In practices, after obtaining all the log event types from historical logs, developers can employ those log event types to parse log messages. Developers can periodically rerun the log parser to mine new log

event types, and update the log event types accordingly.

Heuristic rule-based log parsing methods rely on manually defined rules. Developers mainly learn these rules by studying the characteristics of system logs. Then, they hardcode these rules and cluster the unstructured log messages into different log groups by these rules. In the following, we will introduce two heuristic rules based log parsing methods: SLCT [88] and IPLoM [64].

SLCT. To the best of our knowledge, SLCT [88] is the first log parser, which is short for Simple Logfile Clustering Tool. SLCT is proposed by Vaarandi [88] in 2003. Inspired by association rule, SLCT parse logs in four steps.

- Step 1: Word vocabulary construction. SLCT makes a pass over the words in all the logs and counts the occurrence of them. In this step, the position of the word is also considered. For example, “send” as the 1st word of a log and “send” as the 2nd word of a log are considered different. Words occur more than support threshold, say N, are defined as frequent words.

- Step 2: Cluster candidates construction. In this step, SLCT makes the second pass over all the logs, while at this time it focuses on frequent words. All the frequent words in a log will be extracted be the log template of itself. The number of logs that match a certain log template is counted, and each log template represents a cluster candidate.

- Step 3: Log template extraction. SLCT goes through all cluster candidates and log templates whose corresponding cluster contains more than N logs are selected as the output templates. The logs of clusters which are not selected are placed into outliers class.

- Step 4: Cluster combination. This step is optional. SLCT could make a pass through all selected clusters and combine two clusters if one of them is the subcluster of the other. For

example, cluster “PacketResponder 1 for block * terminating” is the subcluster of “PacketResponder * for block * terminating”. Therefore, these two clusters will be combined in this step.

IPLoM. IPLoM is a heuristic rules-based log parser proposed by Makanju et al. [64]. It conducts log parsing in a 4-step process based on heuristic rules.

- Step 1: Partition by event size. Logs are partitioned into different clusters according to its length. In real world logs, it is possible that logs belong to one template are of variable length. In this case, the result of IPLoM should be postprocessed manually.
- Step 2: Partition by token position. At this point, each cluster contains logs with the same length. Assuming there are m logs whose length are n in a cluster, this cluster can be regarded as an m-by-n matrix. This step based on the assumption that the column with least number of unique words (split word position) is the one contains constants. Thus, the split word position is used to partition each cluster, i.e. each generated cluster has the same word in the split word position.
- Step 3: Partition by search for mapping. In this step, two columns of the logs are selected for further partitioning based on the mapping relation between them. To determine the two columns, the number of unique words in each column is counted (**i.e.** word count) and the two columns with the most frequently appearing word count are selected. There are four mapping relations: 1-1, 1-M, M-1, M-M. In the case of 1-1 relations, logs contain the same 1-1 relations in the two selected columns are partitioned into the same cluster. For 1-M and M-1 relations, we should first decide whether the M side column contains constants or variables. If the M side contains

constants, the M side column is used partition logs in 1-M/M-1 relations. Otherwise, the 1 side column is used. Finally, logs in M-M relations are partitioned into one cluster.

- Step 4: Log template extraction. IPLoM processes through all the clusters generated in previous steps and generates one log template for each of them. For each column in a cluster, the number of unique words is counted. If there is only one unique word in a column, the word is regarded as constant. Otherwise, the words in the column are variables and will be replaced by a wildcard in the output.

Three important parameters of IPLoM are Partition Support Threshold (PST), Cluster Goodness Threshold (CGT) and Lower Bound (LB). PST decides whether a newly generated cluster in the first 3 steps should be regarded as an outlier. CGT indicates whether a cluster is good enough and could skip step 3. Last but not least, LB is used in step 3 to decide whether words in the M side are constants or variables.

The author of SLCT has released its source code written in *C* online [7]. IPLoM has not been released. In this thesis, we implement both SLCT and IPLoM in *Python* and released them as a toolkit.

Offline Log Parsing Based on Clustering Algorithms

LKE. Log Key Extraction (LKE) is used in [35, 60], which utilize both clustering algorithms and heuristic rules.

- Step 1: Log clustering. Weighted edit distance is designed to evaluate the similarity between two logs, $WED = \sum_{i=1}^{n} \frac{1}{1+e^{x_i - v}}$. n is the number of edit operations to make two logs the same, x_i is the column index of the word which is edited by the i-th operation, v is a parameter to control weight. LKE links two logs if the WED between them is less than a threshold σ. After

going through all pairs of logs, each connected component is regarded as a cluster. Threshold σ is automatically calculated by utilizing K-means clustering to separate all WED between all pair of logs into 2 groups, and the largest distance from the group containing smaller WED is selected as the value of σ.

- Step 2: Cluster splitting. In this step, some clusters are further partitioned. LKE firstly finds out the longest common sequence (LCS) of all the logs in the same cluster, such as "Receiving block src: dest:" in log 2 and log 3 in Figure 2.1. The rests of the logs are dynamic parts separated by common words, such as "/10.251.43.210:55700" or "blk_904791815409399662". The number of unique words in each dynamic part column, which is denoted as $|DP|$, is counted. For example, $|DP| = 2$ for the dynamic part column between "src:" and "dest:" in log 2 and log 3. If the smallest $|DP|$ is less than threshold ϕ, LKE will use this dynamic part column to partition the cluster.

- Step 3: Log template extraction. This step is similar to the step 4 of IPLoM. The only difference is that LKE removes all variables when they generate log templates, instead of representing them with wildcards.

LogSig. LogSig [85] is a clustering algorithm which parses logs in three steps.

- Step 1: Word pair generation. In this step, each log is converted to a set of word pairs. For example, the 10th raw log in Figure 2.1 is converted to the following word pairs: *(Verification, succeeded), (Verification, for), (Verification, blk_904791815409399662), (succeeded, for), (succeeded, blk_904791815409399662), (for, blk_904791815409399662)*. Each word pair preserves the order information of the original log.

- Step 2: Clustering. LogSig requires users to determine the number of clusters, say k, which leads to k randomly partitioned clusters of logs at the beginning of clustering. In each iteration of clustering, LogSig goes through all the logs and move them to other clusters if needed. For each log, potential value, which is based on word pairs generated in step 1, is calculated to decide to which cluster the log should be moved. The potential value is explained in detail in Section 4.2.2 in [85]. LogSig keeps clustering until no log is decided to move in one iteration.

- Step 3: Log template extraction. At this point, there are k clusters of logs. For each cluster, words in more than half of the logs are selected as candidate words of the template. To figure out the order of candidate words, LogSig goes through all the logs in the cluster and count how many times each permutation appears. The most frequent one is the log template of the cluster.

LKE and LogSig have not been released by their authors. In this thesis, we implement both LKE and LogSig in *Python* and released them as a toolkit.

Online Log Parsing

Different from offline log parsers, online log parsers parse log messages in a streaming manner. Online log parsers are useful for systems whose codes update frequently. In practice, online log parsers maintain parsing rules and update them dynamically based on the incoming logs. Existing online log parsers employ a tree to encode heuristic rules. The main differences between existing online log parsers are the tree structure and the tree update mechanism. In the following, we will introduce SHISO [69] and Spell [31], two state-of-the-art online log parsers. For both log parsers, we introduce their tree structure, search phase, and update phase.

SHISO. SHISO is proposed by Mizutani [69]. Its core idea is to use all the nodes in the tree as index nodes, where each node in the tree corresponds to a log event type except the root node.

- Tree structure: SHISO's tree starts with a root node, which links to several child nodes. The number of child nodes is controlled by a parameter. For other nodes, we call them index nodes. Each index node links to its child node, and contains a log event type. Thus, each index node employed by SHISO represents a log group.

- Search phase: For an incoming log message, SHISO starts the search process from the root node. It compares the log message with the log events stored in all its child node, and find out the child node whose log event type shares the most similarity with the incoming log message. If the similarity is larger than a threshold, the log message belongs to log group represented by the child node. Otherwise, SHISO checks whether the number of child nodes equals the maximum number of child node, which is a model parameter. If not, the incoming log message will form a new log group, so a new index node will be added as a child node to the current node. If the number of child nodes already equals the maximum number, SHISO will go to the child node that has the largest similarity and iteratively search its child node. If the lengths (i.e., number of tokens) of the log message and that of the log event are different, SHISO thinks the similarity between them is 0. Otherwise, SHISO calculates the similarity by adding up the similarity values between tokens in the same position. For example, the similarity between the first tokens in the log message and the log event. Then the added up value is divided by $2L$, where L is the length of the log message. To calculate the similarity between two tokens, SHISO first map each token to a 4-dimension vector. Each dimension of this vector counts the

number of 4 author-defined chars, including lowercase chars, uppercase chars, digits, and others. For example, "A123ab" will be map to vector $(2, 1, 3, 0)$. After mapping, SHISO calculates the similarity between two vectors based on their Euclidean Distance.

- Update phase: When a new log message arrives, SHISO will go through the search phase. However, the update phase is not called unless a new log event type has been generated. Thus, it will update either the format of an existing index node, or generate a new index node. SHISO maintains a log event table, which contains all existing log events. SHISO calculate the similarity between the newly generated log event and all the existing log events, and find out the existing log event that has the largest similarity. If the largest similarity is larger than a threshold, the two log events are merged.

Spell. Spell is designed by Du et al. [31]. Spell uses a prefix tree to guide the parsing process.

- Tree structure: Spell designs a prefix tree to guide the parsing process. The root node acts as the starting node for the search process. The root node is linked to child nodes, where each child node has a token and links to its own child nodes. The tokens in the tree are employed to accelerate the search process. During the parsing process, some nodes will be connected to log groups, while others only provide the index function.

- Search phase: For an incoming log message. Spell provides two search strategy: prefix tree approach and simple loop approach. Spell first uses prefix tree approach to find the most suitable log group. If it does not find any suitable log group, Spell uses simple loop approach. Prefix tree approach searches the most suitable log group by comparing the tokens in the incoming log message and the tokens prefix tree. Firstly, Spell

checks whether a token stored in its child node equals the first token of the incoming log message. If yes, Spell traverses to that child node and keep checking for the subsequent tokens in the log message. The main goal of this process is to find a path in the prefix tree, where all the tokens along the path form a subsequence of the incoming log message. If the ratio between the number of the tokens along the path and the length of the log message is larger than a threshold, the corresponding log group (if any) will be selected as the search result. If prefix tree approach cannot find a suitable log group, Spell employs the simple loop approach, which linearly compares the similarity between the log message and all existing log events. The similarity is calculated by the length of the longest common subsequence (LCS).

- Update phase: When a log message matches an existing log group, Spell will update the log event stored in the log group. Specifically, Spell compares the tokens in the same token position of log messages and the log event. If the tokens are different, Spell updates the token in that token position to a wildcard. When a log message does not match any existing log group, a new log group is generated with the log message as the log event. Then, a new path is added to the prefix tree, where the last node along the path points to the newly generated log group.

Other Log Parsing Methods

The log parsers introduced above are all data-driven methods that parse unstructured logs without other project materials, such as source codes. There are also some other log parsing methods. Xu et al. [94] propose a log parser based on source code analysis to extract log events from logging statements. Specifically, they statically analyze the project source codes, and find out all the

logging statements. Then they extract the constants from the logging statements and generate parsing rules (i.e., regular expressions) accordingly. However, source codes are often unavailable or incomplete to access, especially when third-party components are employed. Thus, we do not compare our proposed log parsers with source code based methods. Jiang et al. [48] design a data-driven method to parse log messages. The core idea is employing clone detection techniques to cluster similar log messages into log groups. However, their method requires developers to manually construct heuristics (e.g., regular expressions) to find out parameters at the beginning. Different from their work, the log parsers studied in this thesis aims at distinguishing the constants and variables in the logs automatically. Hamooni et al. [38] design automated and parallel log parsing method based on MapReduce. They maintain a hierarchy of patterns in a tree and give users the flexibility to choose a level based on their needs. However, this method mainly focuses on semi-structured logs, which do not contain much free-form text descriptions about system operations. Thus, in summary, these log parsers either have different input (e.g, [94, 38]), or requires extra manual effort (e.g., [48]). The log parsers studied in this thesis employ only system logs as input, and require limited manual effort.

Many researchers (e.g., [23, 72, 38, 22]) and practitioners (as revealed in StackOverflow questions [8, 10]) in this field have to implement their own log parsers to deal with their log data. Our work (Chapter 3, Chapter 4, and Chapter 5) not only provides valuable insights on log parsing, but also releases open-source tool implementations on the proposed log parsers and four representative log parsers.

2.3 Operational Issues Prioritization via Log Events

Logs, as an important data source, are in widespread use to ensure system dependability. Typical examples include anomaly detection

[94, 35, 42], program verification [23, 81], problem diagnosis [98, 72], and security assurance [74, 37]. Most of these log mining tasks employ the parsing results of log parsers as input. Specifically, they need to know the log event type of each log message. For example, some anomaly detection methods [94, 42] generate an event count matrix based on the list of log events output by the log parser. Then the occurrence frequency of each event type can be calculated easily.

Similarly, operational issues prioritization requires the structured log events as input. In the following, we will first introduce the operational issues prioritization process, problem input, and problem output. Then we review two state-of-the-art log-based operational issues prioritization technique in detail and other related log mining tasks.

2.3.1 Problem Description

Essentially, operational issues prioritization is a clustering problem, where similar issues will be clustered into the same issue group. In system runtime, developers may receive issues reported by users because of anomalous system behavior, for example, the response time rapidly increases. The issue often contains related system information, including user information, system configuration, system logs, etc. To address user reported issues, developers mainly inspect the related system logs and try to identify the potential problem. Operational issues prioritization techniques aim at accelerating this process. Specifically, the input is the system logs of all pending issues, where logs belong to an issue form a log sequence. The output will be an ordered list of issue groups. Each issue group contains issues caused by the same potential problem. With this output, developers can handle the high ranking issue groups first, instead of randomly selecting issues. By inspecting an entire issue group, the developer can probably resolve the whole issue group by inspecting only a few issues insides. Thus operational issues

prioritization techniques can largely facilitate the issue handling process.

2.3.2 Literature Review

Operational issue prioritization is a new problem that attracts researchers in recent years. The two most closely related methods are proposed by Shang et al. [81] and Lin et al. [56].

Shang et al. [81] focus on helping developers handle the issues reported in the deployment phase of big data analytics applications. They first collect the logs generated by the applications on several machines with small data. After generating log sequences based on the task IDs inside, they apply the proposed algorithm to cluster the log sequences into issue groups. When the system is deployed on large-scale cloud, the collected logs will go through the same process, including log sequence generation and clustering. Then they will compare the issue groups generated from the test data logs and the large cloud data logs. Only the differences will be reported to developers for further inspection. To cluster the log sequences, they only focus on the appearance of different log events without considering the occurrence frequency of the events. Besides, they do not consider the order of different events. Essentially, they transform each log sequence into an event appearance vector that only contains 0 or 1 for its elements. Finally, the log sequences with the exact same vectors will be clustered into the same group.

Lin et al. [56], which improves the clustering performance of [81], aims at identifying similar problems (i.e., issues) for online service systems. With their method, developers can inspect the issue groups instead of manually inspecting the whole system logs. Specifically, different from the previous method [81], they count the occurrence frequency of different log events. Besides, they employ TF-IDF [80] recalibrate the weights of different log events. Intuitively, by using TF-IDF, log events that appear in more instances

will be assigned a lower weight. Then, instead of clustering the issue with exactly the same vectors, this paper employs the agglomerative hierarchical clustering algorithm to cluster similar vectors based on cosine similarity. The cosine similarity is defined as follows:

$$\begin{aligned} Similarity(S_i, S_j) &= \frac{S_i \cdot S_j}{||S_i||||S_j||} \\ &= \frac{\sum_{k=1}^{n} S_iE_k \times S_jE_k}{\sqrt{\sum_{k=1}^{n}(S_iE_k)^2}\sqrt{\sum_{k=1}^{n}(S_jE_k)^2}} \end{aligned} \tag{2.1}$$

where S_iE_k represents the kth event in the jth sequence vector. After clustering the log sequences into different issue groups, for each issue group, they will find out a representative log sequence by choosing the centroid of the cluster. Then when a new issue arrives, this method can calculate the cosine distance between the new issue vector and all existing representative log sequence vector, and recommend to developers the issue group with the nearest representative vector.

Both methods [81, 56] introduce employ clustering algorithms to cluster log sequences into different groups. With these issue group, developers only need to inspect a number of issue groups instead of manually handling all the reported issues. Similar to these two methods, in Chapter 6, we propose a method to accelerate the issue handling process by clustering. However, we design a novel clustering framework containing a coarse-grained clustering and a fine-grained clustering. Besides, we also propose a novel weighting strategy to recalibrate the weights of different log events.

Besides operational issue handling, there are various reliability assurance tasks that employ log mining techniques, which will be briefly introduced in the following. For anomaly detection, Xu et al. [94] propose a PCA-based model, which is trained by system logs, to detect runtime anomalies. Fu et al. [35] generate a Finite State Automaton (FSA) based on log messages to detect anomalies.

Kc et al. [51] detect anomalies by using both coarse-grained and fine-grained log features. As for program verification, Beschastnikh et al. [23] propose Synoptic to construct a finite state machine from logs as system model. Shang et al. [81] analyze logs from both pseudo and cloud environment to detect deployment bugs for big data analytics applications. Log analysis also facilitates system security assurance. Gu et al. [37] leverage system logs to build an attack detection system for cyber infrastructures. Oprea et al. [74] employ log analysis to detect early-stage enterprise infection. Besides, Pattabiraman et al. [75] design an assertion generator based on execution logs to detect application runtime errors. Log analysis is also employed in structured comparative analysis for performance problem diagnosis [72] and time coalescence assessment for failure reconstruction [29]. As shown in our experiments, the accuracy and efficiency of log parsing could have a great impact on the whole log analysis tasks. Thus, we believe our parallel log parsing approach could benefit future studies on dependability assurance with log analysis.

2.4 QoS Prediction via Limited QoS Values in Logs

In recent years, Service Oriented Architecture (SOA) has become popular in software engineering. Developers can combine a number of existing Web services online to realize a complex system. In this process, developers first select Web services by functions. However, the number of Web services that provide similar functions grows rapidly. Thus, after selecting Web services by functions, developers need to select the most suitable ones based on their non-functional properties: Quality-of-Service (QoS) values. Typical QoS values include response time, throughput, failure probability, etc. For example, developers can select the Web service with the shortest response time. However, in practice, a developer usually only called a few Web services, so she does not know the QoS values of most

	s_1	s_2	s_3	s_4	s_5
u_1	?	0.8	?	?	0.9
u_2	1.0	?	1.0	?	?
u_3	?	?	0.3	0.1	?
u_4	?	0.7	?	0.8	?

Figure 2.2: An Illustrated Example of User-Service Matrix

of the service candidates. Thus, we need QoS prediction method to predict the QoS values based on the existing values. These QoS values can be extracted from the logs that record QoS values. In the following, we introduce the QoS prediction problem, its input, and output.Then we review literature about QoS prediction.

2.4.1 Problem Description

Based on the existing QoS values, we can form a user-service matrix, where each row represents a user and each column represents a service. Thus, each cell in the matrix represents the QoS value when a user calls a service. As illustrated by Fig. 2.2, in this matrix, there are four users and five services, and only limited QoS values are available. This matrix will be the input of QoS prediction. The output of QoS prediction will be a predicted matrix. Based on the predicted matrix, developers can select the service with the best QoS value.

2.4.2 Literature Review

In recent years, service computing [102, 92, 104, 83, 107] has attracted more and more attention from industry communities as well as academic circles. Among all topics in service computing, QoS-aware service selection and service composition are studied in a large number of literature [21, 33, 44, 50], whose goal is to decide

which candidate services to be used as components in complex systems. However, most of the research work has a necessary precondition: QoS values of all candidate services for corresponding users are already known, which is always not satisfied in real-world cases. Thus, many researchers begin to focus on QoS prediction issues, which aims at analyzing existing user-services invocation records and then predict those unobserved ones.

Collaborative filtering was applied to this problem by Shao et al. [82] first. In their paper, a user-based collaborative filtering method was proposed, which predicts a specific user based on similar users. Zheng et al. [103] designed a hybrid approach to leverage both user-based and item-based collaborative filtering. Chen et al. [26] built a region model before collaborative filtering step, their method can be tuned to trade off speed and recommendation accuracy. Tang et al. [87] raised a hierarchical method to predict QoS values on the user side and service side separately. Different from previous methods that directly dig out neighbors on all historical records, they seek similar users in the same Autonomous System (AS) and country first. These collaborative filtering methods are classified as memory-based collaborative filtering techniques. Although these methods have a good performance when there are enough user-service invocation records, they do not perform well when the matrix becomes bigger and sparser.

Compared with memory-based collaborative filtering methods, model-based collaborative filtering methods can provide us with more precise prediction result. The overall idea of model-based methods is to train a model according to existing data and use that trained model to predict missing QoS values. Probabilistic matrix factorization (i.e. matrix factorization in this paper) was proposed by Salakhutdinov et al. [70]. Lo et al. [59] raised an extended matrix factorization approach, whose main contribution is two novel relational regularization terms that can improve prediction accuracy. They also proposed a location-aware matrix factorization

model [58]. In that work, they designed two user-location-aware matrix factorization models, each of which extended by a location regularization term. However, location information of services is neglected, which is actually helpful in improving prediction accuracy. In this thesis, the model proposed in Chapter 7 utilizes geographical information of both users and services simultaneously. Besides, our model is run in a hierarchical way, which means it can make use of not only global context, but also local information.

Besides, recent studies focus on enhancing the reliability of Web service systems. Cubo et al. [28] use dynamic software product lines to reconfigure service failures dynamically. Service selection and recommendation are also widely studied [46, 67]. These studies usually employ QoS (quality of service) values to characterize the reliability of different Web services. Jurca et al. [49] propose a reliable QoS monitoring technique based on client feedback. Yao et al. [95] develop a model with accountability for business and QoS compliance. Besides, Chen et al. [25] propose a performance prediction method for component-based applications. Our proposed online log parser is critical for log analysis techniques, which can complement with these methods in reliability enhancement for Web service systems. The log analysis methods can also improve the reliability of many existing service systems [101, 32].

□ **End of chapter.**

Chapter 3

Evaluation Study of Log Parsing and Its Use in Log Mining

In log analysis, before feeding logs into data mining models, logs need to be parsed by a log parser because of their unstructured format. This chapter presents an evaluation study on four representative log parsers. A key finding is that log parsing is important, because 4% errors in parsing could even cause on order of magnitude performance degradation in log mining. The main points of this chapter are as follows. (1) It reviews and evaluates four representative log parsers in terms of accuracy and efficiency (2) It conducts a case study on system anomaly detection to evaluate the effectiveness of log parsers on log mining. (3) It obtains six insightful findings and release the studied parsers as an open-source toolkit.

3.1 Introduction

Logs are widely used to record runtime information of software systems, such as the timestamp of an event, the unique ID of a user request, and the state of a task execution. The rich information of logs enables system developers (and operators) to monitor the runtime behaviors of their systems and further track down system problems in production settings.

With the ever-increasing scale and complexity of modern sys-

tems, the volume of logs is rapidly growing, for example, at a rate of about 50 gigabytes (around 120∼200 million lines) per hour [68]. Therefore, the traditional way of log analysis that largely relies on manual inspection has become a labor-intensive and error-prone task. To address this challenge, many efforts have recently been made to automate log analysis by the use of data mining techniques. Typical examples of log mining include anomaly detection [94, 35, 63], program verification [23, 81], problem diagnosis [98, 72], and security assurance [74, 37]. However, raw log messages are usually unstructured, because developers are allowed to record a log message using free text for convenience and flexibility. To enable automated mining of unstructured logs, the first step is to perform log parsing, whereby unstructured raw log messages can be transformed into a sequence of structured events.

Typically, a log message, as illustrated in the following example, records a specific system event with a set of fields: *timestamp* (recording the occurring time of the event), *verbosity level* (indicating the severity level of the event, e.g., INFO), and *raw message content* (recording what has happened during system operation).

```
2008-11-09 20:35:32,146 INFO dfs.DataNode$DataXceive
r: Receiving block blk_-1608999687919862906 src: /10
.251.31.5:42506 dest: /10.251.31.5:50010
```

As observed in the example, the raw message content can be divided into two parts: *constant* part and *variable* part. The constant part constitutes the fixed plain text and remains the same for every event occurrence, which can reveal the event type of the log message. The variable part carries the runtime information of interest, such as the values of states and parameters (e.g., the IP address and port: 10.251.31.5:50010), which may vary among different event occurrences. The goal of log parsing is to extract the event by automatically separating the constant part and variable part of a raw log message, and further transform each log message into a specific event (usually denoted by its constant part). In this

example, the event can be denoted as "*Receiving block * src: * dest: **", where the variable part is identified and masked using asterisks. We will use "event" and "template" interchangeably in this paper.

Log parsing is essential for log mining. Traditionally, log parsing relies heavily on regular expressions to extract the specific log event (e.g., SEC [54]). However, modern software systems, with increasing size and complexity, tend to produce a huge volume of logs with diverse log events. It requires non-trivial efforts for manual creation and maintenance of regular expression rules. Especially, when a system constantly evolves, the rules of log parsing will most likely become outdated very often. For example, Google's systems, as studied in [93], have been introduced with up to thousands of new log printing statements every month. As a result, there is a high demand for automated log parsing methods, capable of evolving with the system.

To achieve this goal, recent studies have proposed a number of data-driven approaches for automated log parsing (e.g., SLCT [88], IPLoM [62], LKE [35], LogSig [85]), in which historical log messages are leveraged to train statistical models for event extraction. Despite the importance of log parsing, we found that, to date, there is a lack of systematic evaluations on the effectiveness and efficiency of the automated log parsing methods available. Meanwhile, except SLCT [88] that was released more than 10 years ago, there are no other ready-to-use tool implementations of log parsers. Even with commercial log management solutions, such as Splunk [19] and Logstash [18], users need to provide complex configurations with customized rules to parse their logs. In this context, engineers and researchers have to implement their own log parsers when performing log mining tasks (e.g., [23, 72, 22]), which would be a time-consuming yet redundant effort. Besides, they are likely unaware of the effectiveness of their implementations compared to other competitive methods, nor do they notice the impact of log parsing on subsequent log mining tasks.

To fill this significant gap, in this paper, we perform a systematic evaluation study on the state-of-the-art log parsing methods and their employment in log mining. In particular, we intend to investigate the following three research questions:

RQ1: *What is the accuracy of the state-of-the-art log parsing methods?*

RQ2: *How do these log parsing methods scale with the volume of logs?*

RQ3: *How do different log parsers affect the results of log mining?*

Towards this end, we have implemented four widely-employed log parsers: SLCT [88], IPLoM [62], LKE [35], LogSig [85]. They are currently available on our Github[1] as an open-source toolkit, which can be easily re-used by practitioners and researchers for future study. For evaluation, we have also collected five large log datasets (with a total of over 10 million raw log messages) produced by production software systems. The evaluation is performed in terms of both accuracy and efficiency in log parsing. Furthermore, we evaluate the impact of different log parsers on subsequent log mining tasks, with a case study on system anomaly detection (proposed in [94]).

Through this comprehensive evaluation, we have obtained a number of insightful findings: Current log parsing methods could obtain high overall accuracy (*Finding 1*), especially when log messages are preprocessed with some domain knowledge based rules (*Finding 2*). Clustering-based log parsing methods could not scale well with the volume of logs (*Finding 3*), and the tuning of parameters (e.g., number of clusters) is time-consuming (*Finding 4*). Log mining is effective only when the parsing accuracy is high enough (*Finding 5*). Because log mining can be sensitive to some critical events. 4% parsing errors on critical events can cause an order of magnitude

[1]https://github.com/cuhk-cse/logparser

performance degradation in log mining (*Finding 6*). These findings as well as our toolkit portray a picture about the current situation of log parsing methods and their effectiveness on log mining, which we believe could provide valuable guidance for future research in this field.

3.2 Log Parsing Overview

This section first provides an overview of log parsing and then describes four existing log parsing methods. These methods are widely employed and thus become the main subjects of our study.

3.2.1 Overview of Log Parsing

Fig. 3.1 illustrates an overview of log parsing. The raw log messages, as shown in the figure, contain ten log messages extracted from HDFS log data on Amazon EC2 platform [94]. The log messages are unstructured data, with timestamps and raw message contents (some fields are omitted for simplicity of presentation). In real-world cases, a log file may contain millions of such log messages. The goal of log parsing is to distinguish between *constant* part (fixed plain text) and *variable* part (e.g., blk_ID in the figure) from the log message contents. Then, all the constant message templates can be clustered into a list of *log events*, and *structured logs* can be generated with each log message corresponding to a specific event. For instance, the log message 2 is transformed to "*Event2*" with a log template "*Receiving block * src: * dest: **". The output of a log parser involves two files with *log events* and *structured logs*. Log events record the extracted templates of log messages, while structured logs contain a sequence of events with their occurring times. Finally, the structured logs after parsing can be easily processed by log mining methods, such as anomaly detection [94] and deployment verification [81].

Figure 3.1: Overview of Log Parsing

3.2.2 Existing Log Parsing Methods

Log parsing has been widely studied in recent years. Among all the approaches proposed, we choose four representative ones, which are in widespread use for log mining tasks. With the main focus on evaluations of these log parsing methods, we only provide brief reviews of them; the details can be found in the corresponding references.

1) SLCT

SLCT (Simple Logfile Clustering Tool) [88] is, to the best of our knowledge, the first work on automated log parsing. The work also released an open-source log parsing tool, which has been widely employed in log mining tasks, such as event log mining [89], symptom-based problem determination [45] and network alert classification [90].

Inspired by association rule mining, SLCT works as a three-step procedure with two passes over log messages: *1) Word vocabulary construction.* It makes a pass over the data and builds a vocabulary of word frequency and position. *2) Cluster candidates construction*. It makes another pass to construct cluster candidates using the word vocabulary. *3) Log template generation.* Clusters with enough log messages are selected from candidates. Then, the log messages in each cluster can be combined to generate a log template, while remaining log messages are placed into an outlier cluster.

2) IPLoM

IPLoM (Iterative Partitioning Log Mining) [64] is a log parsing method based on heuristics specially designed according to the characteristics of log messages. This method has also been used by a set of log mining studies (e.g., alert detection [63], event log analysis [65] and event summarization [47]).

Specifically, IPLoM performs log parsing through a three-step hierarchical partitioning process before template generation: *1) Partition by event size.* Log messages are partitioned into different clusters according to different lengths. *2) Partition by token position.* For each partition, words at different positions are counted. Then the position with the least number of unique words is used to split the log messages. *3) Partition by search for mapping.* Further partition is performed on clusters by searching for mapping relationships between the set of unique tokens in two token positions selected

using a heuristic criterion. *4) Log template generation.* Similar to SLCT, the final step is to generate log templates from every cluster.

3) LKE

LKE (Log Key Extraction) [35] is a log parsing method developed by Microsoft, and has been applied in a set of tasks on unstructured log analysis [35, 60].

LKE utilizes both clustering algorithms and heuristic rules for log parsing: *1) Log clustering.* Raw log messages are first clustered by using hierarchical clustering algorithms with a customized weighted edit distance metric. *2) Cluster splitting.* A splitting step based on heuristic rules is performed to further split the clusters. *3) Log template generation.* The final step is to generate log templates from every cluster, similar to SLCT and IPLoM.

4) LogSig

LogSig [85] is a more recent log parsing method, which has been validated in [86].

LogSig works in three steps: *1) Word pair generation.* Each log message is converted to a set of word pairs to encode both the word and its position information. *2) Log Clustering.* Based on the word pairs, a potential value is calculated for each log message to decide which cluster the log message potentially belongs to. After a number of iterations, the log messages can be clustered. *3) Log template generation.* In each cluster, the log messages are leveraged to generate a log template.

3.2.3 Tool Implementation

Among these log parsing methods, we only found an open-source implementation on SLCT in C language. To enable our evaluations, we have implemented the other three log parsing methods in Python

and also wrapped up SLCT as a Python package. For ease of use, we define standard input/output formats for these log parsers. As shown in Fig. 3.1, the input is a file with raw log messages, while the output contains both a file with log events and a file with structured logs. The output can be easily fed into subsequent log mining tasks. Currently, all our implementations have been open source on Github, which can be used as a toolkit for log parsing. We believe our toolkit could benefit other researchers and practitioners as well.

It is also worth noting that our current implementation targets at exactly reproducing the log parsing methods (as described in original work) for our evaluation purposes. As we will show in Section 3.4.3, LKE and LogSig do not scale well on large datasets. Although we plan to improve their efficiency in our future work, users may need to pay more attention when using our current toolkit.

3.3 Log Mining

In this section, we briefly introduce three representative log mining tasks and explain how the adopted log parsing step can affect the performance of these tasks. Further, we describe the details of a specific log mining task, system anomaly detection, which will be used for our evaluations.

3.3.1 Overview of Log Mining

Anomaly detection: Logs of Hadoop File System (HDFS) are used by Xu et al. [94] to detect anomalies in a 203-nodes HDFS. In this case, they employ source code based log parsers (not evaluated because it is beyond the scope of this paper) to find out the log events associated with each block ID, which are further interpreted with a block ID-by-event count matrix. This matrix is fed into a machine learning model to detect anomalies of the system. If the log parser

adopted does not work well, some block IDs will match wrong log events, which could ruin the generated matrix and lead to failure of the anomaly detection approach.

Deployment verification: Big data application is usually developed in pseudo-cloud environment (with several PC nodes) and finally deployed in a large-scale cloud environment. Runtime analysis and debugging of such applications in deployment phase is a challenge tackled by Shang et al. in [81]. To reduce the amount of log messages which needs to be checked by developers, they compare the log event sequences generated in pseudo-cloud and large-scale cloud. Only the different log event sequences are reported to the developers, which greatly alleviates their workload. In this task, a bad log parser may produce wrong log event sequences. This could largely degrade the reduction effect because their method is based on the comparison of log event sequences.

***System model construction*:** Computer systems are difficult to debug and understand. To help developers gain insight into system behaviors, Beschastnikh et al. [23] propose a tool called Synoptic to build an accurate system model based on logs. Synoptic requires parsed log events as input and generates a finite state machine as the output system model. If an unsuitable log parser is used, both initial model building step and model refinement step will be affected. These may result in extra branches or even totally different layout of the model.

3.3.2 System Anomaly Detection

To better study the impact of log parsing approaches on the subsequent log mining task, we reproduce the anomaly detection method proposed in [94] on its original HDFS logs while using different log parsing approaches discussed in Section. 3.2.2. The anomaly detection method contains three steps: log parsing, event matrix generation, and anomaly detection.

Log Parsing

The input of the anomaly detection task is a text file, each line of which is a raw log message recording an event occurring on a block in HDFS. In this step, log parsing method is adopted to figure out two things. One is all the event types appearing in the input file. The other is the events associated with each block, which distinguished by block ID. These two are exactly in the two output files of our log parser modules. We emphasize that the parsing output is not specific to anomaly detection, but also suitable for other log mining tasks.

Matrix Generation

Parsed results are used to generate an event count matrix Y, which will be fed into the anomaly detection model. In the event count matrix, each row represents a block, while each column indicates one event type. The value in cell $Y_{i,j}$ records how many times event j occurs on block i. We could generate Y with one pass through the parsed results. Instead of directly detecting anomaly on Y, TF-IDF [80], which is a well-established heuristic in information retrieval, is adopted to preprocess this matrix. Intuitively, TF-IDF is to give lower weights to common event types, which are less likely to contribute to the anomaly detection process.

Anomaly Detection

In this case, anomaly detection is to find out suspicious blocks that may indicate problems (e.g., HDFS namenode not updated after deleting a block). The model used is Principle Component Analysis (PCA) [94], which is a statistical model that captures patterns in high-dimensional data by selecting representative coordinates (principle components). PCA is used in this problem because principle components can represent most frequent patterns of events associated with blocks, which is called normal space S_d. Specifically, the first k principle components are selected to form

S_d, while the remaining $n-k$ dimensions form S_a (anomaly space), where n is the number of columns (total number of event type) of the matrix. In this task, each row in the event count matrix is a vector y associated with a block. The intuition of anomaly is the vector whose end point is far away from normal space. The "distance" could be formalized by squared prediction error $SPE \equiv ||y_a||^2$, where y_a is the projection of y on S_a. y_a is calculated by $y_a = (I-PP^T)y$, where $P = [v_{1,}v_{2,}..., v_k]$. A block is marked as anomaly if its corresponding y satisfies:

$$SPE = ||y_a||^2 > Q_\alpha,$$

where Q_α is a threshold providing $(1-\alpha)$ confidence level. For Q_α, we choose $\alpha = 0.001$ as in the original paper [94].

3.4 Evaluation Study

This section presents our study methodology and reports on the detailed results for the proposed research questions.

3.4.1 Study Methodology

Log Datasets: To facilitate systematic evaluations on the state-of-the-art log parsing methods, we have used five large log datasets ranging from supercomputers (BGL and HPC) to distributed systems (HDFS and Zookeeper) to standalone software (Proxifier), with a total of 16,441,570 lines of log messages. Table 3.1 provides a basic summarization of these datasets. Logs are scarce data for research, because companies are often reluctant to release their production logs due to confidentiality issue. We obtained three log datasets, with the generous support from their authors. Specifically, BGL is an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL), with 131,072 processors and 32,768GB memory [73]. HPC is also

Table 3.1: Summary of Our System Log Datasets

System	Description	#Logs	Length	#Events
BGL	BlueGene/L Supercomputer	4,747,963	10∼102	376
HPC	High Performance Cluster (Los Alamos)	433,490	6∼104	105
Proxifier	Proxy Client	10,108	10∼27	8
HDFS	Hadoop File System	11,175,629	8∼29	29
Zookeeper	Distributed System Coordinator	74,380	8∼27	80

an open dataset with logs collected from a high performance cluster at Los Alamos National Laboratory, which has 49 nodes with 6,152 cores and 128GB memory per node [57]. HDFS logs are collected in [94] by using a 203-node cluster on Amazon EC2 platform. To enrich the log data for evaluation purpose, we further collected two datasets: one from a desktop software Proxifier, and the other from a Zookeeper installation on a 32-node cluster in our lab.

In particular, the HDFS logs from [94] have well-established anomaly labels, each of which indicates whether or not a request for a data block operation is an anomaly. The labels are made based on domain knowledge, which are suitable for our evaluations on anomaly detection with different log parsers. Specifically, the dataset with over 11 million log messages records 575,061 operation requests with a total of 29 event types. Among all the 575,061 requests, 16,838 are marked as anomalies, which we use as ground truth in our evaluation.

***Experimental Setup*:** All our experiments were run on a Linux server with Intel Xeon E5-2670v2 CPU and 128GB DDR3 1600 RAM, running 64-bit Ubuntu 14.04.2 with Linux kernel 3.16.0. We use F-measure [66, 9], a commonly-used evaluation metric for clustering algorithms, to evaluate the parsing accuracy of log parsing methods. To calculate F-measure, we manually obtain the

Table 3.2: Parsing Accuracy of Log Parsing Methods (Raw / Preprocessed)

	BGL	HPC	HDFS	Zookeeer	Proxifier
SLCT	**0.61/0.94**	0.81/0.86	0.86/0.93	0.92/0.92	0.89/-
IPLoM	0.99/0.99	0.64/0.64	0.99/1.00	0.99/0.99	0.84/-
LKE	0.67/0.70	0.17/0.17	**0.57/0.96**	0.78/0.82	0.81/-
LogSig	**0.26/0.98**	0.77/0.87	0.91/0.93	0.96/0.99	0.84/-

ground truths for all logs of these dataset. It is possible because we iteratively filter out logs with confirmed event using regular expression. Experiments about LKE and LogSig are run 10 times to avoid bias of clustering algorithms, while others are run once because they are deterministic. We note here that only the parts of free-text log message contents are used in evaluating the log parsing methods.

3.4.2 RQ1: Accuracy of Log Parsing Methods

To study the accuracy of different log parsing methods, we use them to parse our collected real logs. As with the existing work [85], we randomly sample 2k log messages from each dataset in our evaluation, because the running time of LKE and LogSig is too long on large log datasets (e.g., LogSig requies 1 day to parse entire BGL data). The average results of 10 runs are reported in Table 3.2. We can observe that the overall accuracy of these log parsing methods is high (larger than 0.8 in most cases). Meanwhile, the overall accuracy on HDFS, Zookeeper and Proxifier datasets is higher than that obtained on the others. We found that this is mainly because BGL and HPC logs involve much more event types, each of which has a longer length than other datasets.

Especially, we found that LKE takes an aggressive clustering strategy, which groups two clusters if any two log messages between them has a distance smaller than a specified threshold. This is why LKE has an accuracy drop on HPC dataset, in which it clusters almost all the log messages into one single cluster in the first step.

BGL contains a lot of log messages whose event is "*generating core.**", such as "generating core.2275" and "generating core.852". Intuitively, the similarity of these two log messages are 50%, because half of the words are different. LogSig tends to separate these log messages into different clusters, which causes its low accuracy on BGL. Particularly, IPLoM leverages some heuristic rules developed on the characteristics of log messages, while other log parsing methods rely on well-studied data mining models. However, we found that IPLoM obtains the superior overall accuracy (0.88) against other log parsing methods. This further implies the particular importance of exploiting the unique characteristics of log data in log parsing, which would shed light on future design and improvement of a log parser.

Finding 1: Current log parsing methods achieve modest overall parsing accuracy (F-measure).

Instead of running log parsing methods directly on raw log messages, developers usually preprocess log data with domain knowledge. In this experiment, we study the impact of preprocessing on parsing accuracy. Specifically, we remove obvious numerical parameters in log messages (i.e., IP addresses in HPC & Zookeeper & HDFS, core IDs in BGL, and block IDs in HDFS). Proxifier does not contain words that could be preprocessed based on domain knowledge. Preprocessing is mentioned in LKE and LogSig; however, its importance has not been studied.

In Table 3.2, the numbers on the left/right side represent the accuracy of log parsing methods on raw/preprocessed log data. In most cases, accuracy of parsing is improved. Preprocessing greatly increases the accuracy of SLCT/LKE/LogSig on one dataset (in bold). However, preprocessing could not improve the accuracy of IPLoM. This is mainly because IPLoM considers preprocessing internally in its four-step process.

Finding 2: Simple log preprocessing using domain knowledge (e.g. removal of IP address) can further improve log parsing

accuracy.

3.4.3 RQ2: Efficiency of Log Parsing Methods

In Fig. 3.2, we evaluate the running time of the log parsing methods on all datasets by varying the number of raw log messages. Notice that as the number of raw log messages increases, the number of events becomes larger as well (e.g., 60 events in BGL400 while 206 events in BGL40k). SLCT and IPLoM, which are based on heuristic rules, scale linearly with the number of log messages (note that Fig. 3.2 is in logarithmic scale). Both of them could parse 10 million HDFS log messages within five minutes. For the other two clustering-based parsing methods, LogSig also scales linearly with the number of log messages. However, its running time also increases linearly with the number of events, which leads to relatively longer parsing time (e.g, 2+ hours for 10m HDFS log messages). The time complexity of LKE is $O(n^2)$, which makes it unable to handle large-scale log data, such as BGL4m and HDFS10m. Some running time of LKE is not plotted because LKE could not parse some scales in a reasonable time (may cause days or even weeks). To reduce the running time of clustering-based log parsing method, parallelization is a promising direction.

Finding 3. Clustering-based log parsing methods could not scale well on large log data, which implies the demand for parallelization.

The accuracy of log parser is affected by parameters. For example, the number of clusters of LogSig decides the number of events, which should be set beforehand. For large-scale log data, it is difficult to select the most suitable parameters by trying different values, because each run will cause a lot of time. A normal solution is to tune the parameters in a sample dataset and directly apply them on large-scale data. To evaluate the feasibility of this approach, we tune parameters for log parsing methods on 2k sample log messages, which are used in our parsing accuracy experiment. In Fig. 3.3,

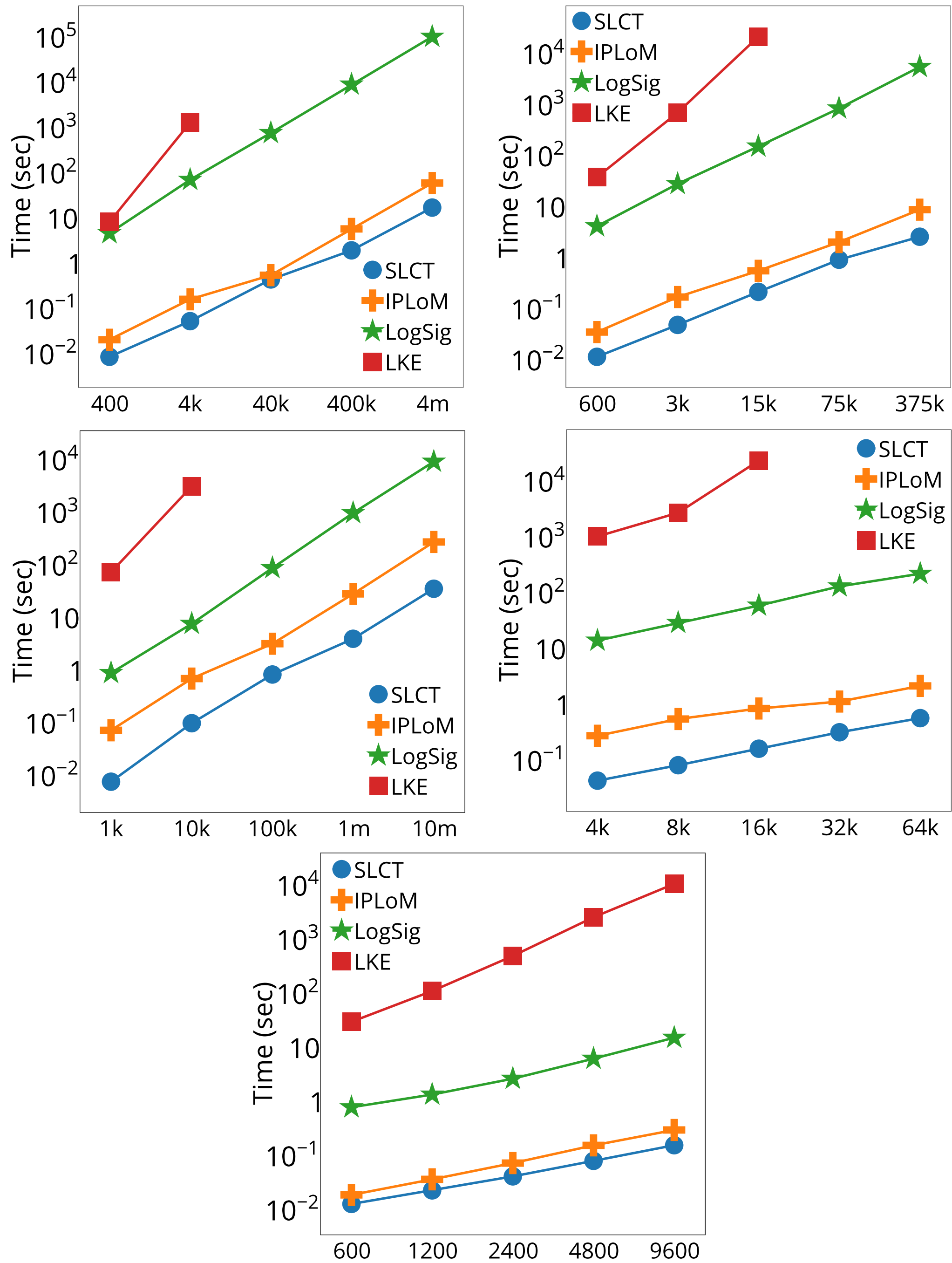


Figure 3.2: Running Time of Log Parsing Methods on Datasets in Different Size

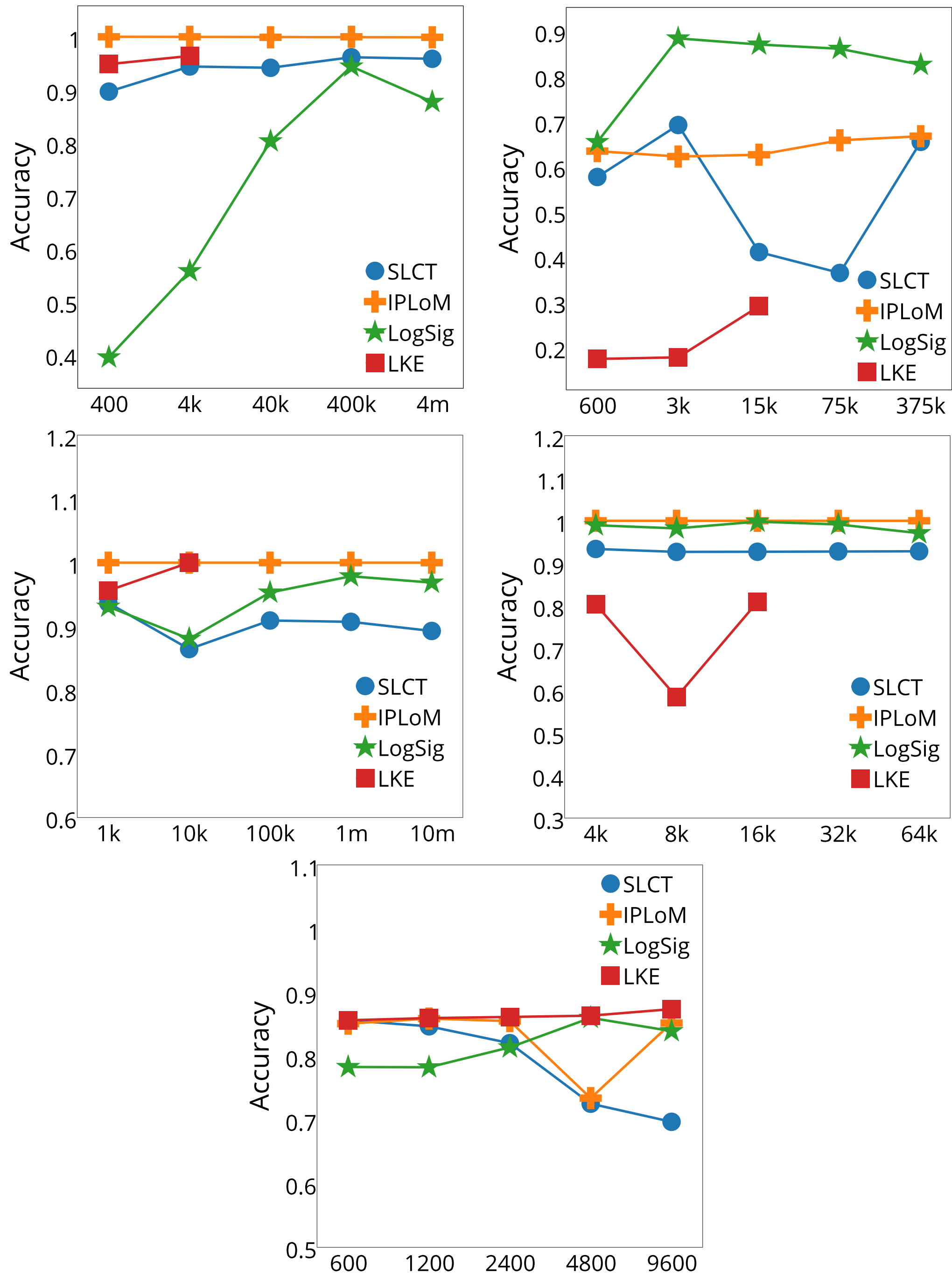


Figure 3.3: Parsing Accuracy on Datasets in Different Size

Table 3.3: Anomaly Detection with Different Log Parsing Methods (16,838 Anomalies)

	Parsing Accuracy	Reported Anomaly	Detected Anomaly	False Alarm
SLCT	0.83	18,450	10,935 (64%)	7,515 (40%)
LogSig	0.87	11,091	10,678 (63%)	413 (3.7%)
IPLoM	0.99	10,998	10,720 (63%)	278 (2.5%)
Ground truth	1.00	11,473	11,195 (66%)	278 (2.4%)

we vary the size of the dataset and evaluate the accuracy of the log parsing method using these parameters. The results show that the IPLoM performs consistently in most cases. SLCT is also consistent in most cases except HPC. The accuracy of LKE is volatile because of the weakness of its clustering algorithm discussed in Section 3.4.2. LogSig performs consistently on datasets with limited types of events, but its accuracy varies a lot on datasets with many events (i.e., BGL and HPC). Thus, for LKE and LogSig, directly using parameters tuned on sample dataset is not practical, which makes parameter tuning on large-scale logs time-consuming.

Finding 4. Parameter tuning for clustering-based log parsing methods is a time-consuming task, especially on large log datasets.

3.4.4 RQ3: Effectiveness of Log Parsing Methods on Log Mining

To evaluate the effectiveness of log parsing methods on log mining, we use three log parsers to tackle the parsing challenge of a real-world anomaly detection task described in Section 3.3.2. In this task, there are totally 16,838 anomalies, which are found manually in [94]. The parameters of SLCT and LogSig are re-tuned to provide good Parsing Accuracy. LKE is not employed because it could not handle this large amount of data (10m+ lines) in reasonable time. The evaluation results are illustrated in Table 3.3. Reported Anomaly is the number of anomalies reported by PCA, while adopt-

ing different log parsers in the log parsing step. Detected Anomaly is the number of true anomalies detected by PCA. False Alarm means the number of wrongly detected anomalies. Ground truth is the experiment using exactly correct parsed results in anomaly detection. Notice that even the Ground truth could not detect all anomalies because of the boundary of the PCA anomaly detection model.

From Table 3.3, we observe that the parsing accuracy of these parsing methods are high (0.83 at least). LogSig and IPLoM lead to nearly optimal results on the anomaly detection task. However, not all parsing methods lead to optimal results. SLCT presents high Parsing Accuracy (0.83), but it brings about 7,515 False Alarms in anomaly detection, which introduces extensive unnecessary human effort on inspection.

Finding 5. Log parsing is important because log mining is effective only when the parsing accuracy is high enough.

From Table 3.3, we observe that the parsing accuracy of SLCT (0.83) and LogSig (0.87) is comparable. However, the performance of log mining using LogSig as parser is an order of magnitude better than that using SLCT. Log mining task using SLCT presents 7,515 False Alarms, introducing much more human inspection effort than that using LogSig, which only leads to 413 False Alarms. Besides, the log mining tasks using LogSig and IPLoM as parsers produce comparable results. However, LogSig presents 12% more parsing errors than IPLoM. These reveal that log mining results are sensitive to some critical events, which could cause an order of magnitude performance degradation. These also indicate that f-measure, despite pervasively used in clustering algorithm evaluation, may not be suitable to evaluate the effectiveness of log parsing methods on log mining.

Finding 6. Log mining is sensitive to some critical events. 4% *errors in parsing could even cause an order of magnitude performance degradation in log mining.*

3.5 Discussions

Diversity of dataset. Not all datasets (two out of five) used in our evaluation are production data, and the results may be limited by the representativeness of our datasets. This is mainly because public log data is lacking. As a result, we cannot claim that our results are broadly representative. However, Zookeeper and HDFS are systems widely adopted by companies for their distributed computing jobs. We believe these logs could reflect the logs from industrial companies to some extent. We also mitigate this issue by generating many sample datasets from the original ones, where each sample dataset has different properties, such as log size and the number of log events. The proposed parser POP at least has consistent accuracy and efficiency on all these datasets, which demonstrates its robustness. Besides, we thank those who release log data [94, 57, 73], which greatly facilitates our research.

Diversity of log mining tasks. Results of effectiveness of log parsing methods are evaluated on anomaly detection, which may not generalize to other log mining tasks. This is mainly because public real-world log mining data with labels is scarce. However, the anomaly detection task evaluated is an important log mining task widely studied [53, 77], which is presented in a paper [94] enjoying more than 300 citations. Besides, even conducting evaluation on one log event mining task, the result reveals that an accurate log parser is of great importance for obtaining optimal log mining performance. We will consider to extend our methodology on more log parsing data and different log mining tasks in our future work.

Distributed Log Parsing. Our experiments show that current log parsing methods cost a lot of time on big data input. The amount of log message in industrial companies could be much larger. Log parsing methods based on heuristic rules are fast but their parsing result is not good enough to fulfill the need of log mining task. Thus, to accelerate the parsing process and further improve its

accuracy, log parsing methods which run in a distributed manner are in demand. Clustering algorithms which could be parallelized should be considered.

Logging of Event ID. We could also improve log parsing process by recording event ID in logs in the first place. This approach is feasible because developer writing log knows exactly which event a log message statement match. Thus, adding event ID to log message is a good logging practice [79] from the perspective of log mining. Tools that could automatically add event ID into source code may greatly facilitate the log parsing process.

3.6 Summary

Log parsing is employed pervasively in log mining. However, due to the lack of studies on performance of log parsing methods, users often re-design a specialized log parser, which is time-consuming. In this chapter, we study the performance of four state-of-the-art log parsing methods through extensive experiments. We also analyze the effectiveness of the log parsing methods on a real-world log mining task with 10 million log messages. We provide six valuable findings on the parsing accuracy of the log parsers, efficiency of the log parsers, and their effectiveness on log mining. In addition, the source code of these log parsing methods is released for reuse and further study.

□ **End of chapter.**

Chapter 4

Parallel Log Parsing for Large-Scale Log Data

This chapter presents POP, a parallel log parsing framework that can utilize the computing power of multiple machines simultaneously. The key notion is that each machine could mine parsing statistics locally, and communicate with the center node for the final parsing. The main points of this chapter are as follows. (1) It presents the design and implementation of the first parallel log parsing framework POP. (2) POP employs both specially designed heuristic rules and hierarchical clustering algorithm. (3) It evaluates POP on synthetic and real-world datasets and releases the source code of POP for repeatable research.

4.1 Introduction

Large-scale distributed systems are becoming the core components of the IT industry, supporting daily use software of various types, including online banking, e-commerce, and instant messaging. In contrast to traditional standalone systems, most of such distributed systems run on a 24×7 basis to serve millions of users globally. Any non-trivial downtime of such systems can lead to significant revenue loss [13, 12], and this thus highlights the need to ensure system dependability.

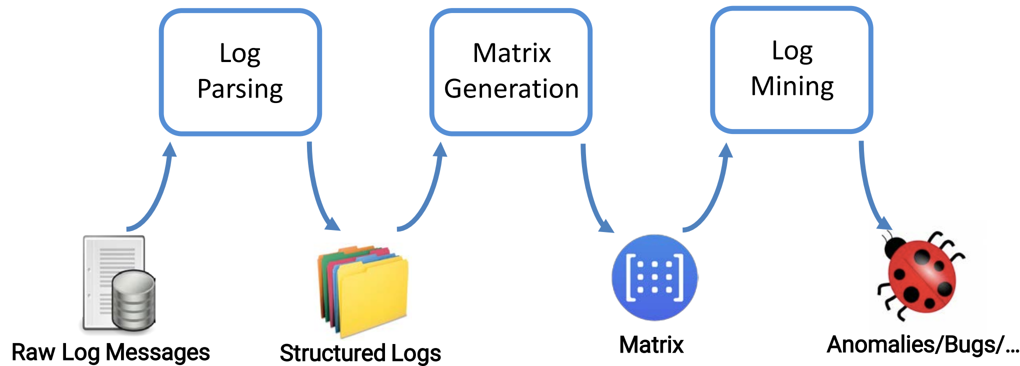


Figure 4.1: Overview of Log Analysis

System logs are widely utilized by developers (and operators) to ensure system dependability, because they are often the only data available that record detailed system runtime information in production environment. In general, logs are unstructured text generated by logging statements (e.g., *printf()*, *Console.Writeline()*) in system source code. Logs contain various forms system runtime information, which enables developers to monitor the runtime behaviors of their systems and to further assure system dependability.

With the prevalence of data mining, the traditional method of log analysis, which largely relies on manual inspection and is labor-intensive and error-prone, has been complemented by automated log analysis techniques. Typical examples of log analysis techniques on dependability assurance include anomaly detection [94, 35, 42], program verification [23, 81], problem diagnosis [98, 72], and security assurance [74, 37]. Most of these log analysis techniques comprise three steps: log parsing, matrix generation, and log mining (Fig. 4.1). The performance of log parsing plays an important role in various log analysis frameworks in terms of both accuracy and efficiency. The log mining step usually accepts structured data (e.g., a matrix) as input and reports mining results to developers. However, raw log messages are usually unstructured because they are natural language designed by developers. Typically, a raw log message, as illustrated in the following example, records a specific system event

with a set of fields: timestamp, verbosity level, and raw message content.

```
2008-11-09 20:46:55,556 INFO dfs.DataNode$PacketResp
onder: Received block blk_3587508140051953248 of siz
e 67108864 from /10.251.42.84
```

Log parsing is usually the first step of automated log analysis, whereby raw log messages can be transformed into a sequence of structured events. A raw log message, as illustrated in the example, consists of two parts, namely a *constant* part and a *variable* part. The constant part constitutes the fixed plain text and represents the corresponding event type, which remains the same for every event occurrence. The variable part records the runtime information, such as the values of states and parameters (e.g., the block ID: blk_3587508140051953248), which may vary among different event occurrences. The goal of log parsing is to automatically separate the constant part and variable part of a raw log message (also known as log de-parametrization), and to further match each log message with a specific event type (usually denoted by its constant part). In the example, the event type can be denoted as "*Received block * of size * from **", where the variable part is identified and masked using asterisks.

Traditional log parsing approaches rely heavily on manually customized regular expressions to extract the specific log events (e.g., SEC [54]). However, this method becomes inefficient and error-prone for modern systems for the following reasons. First, the volume of log grows rapidly; for example, it grows at a rate of approximately 50 GB/h (120∼200 million lines) [68]. Manually constructing regular expressions from such a large number of logs is prohibitive. Furthermore, modern systems often integrate open-source software components written by hundreds of developers [94]. Thus, the developers who maintain the systems are usually unaware of the original logging purpose, which increases the difficulty of the manual method. This problem is compounded by the fact that the

log printing statements in modern systems update frequently [106] (e.g., hundreds of new logging statements every month [93]); consequently, developers must regularly review the updated log printing statements of various system components for the maintenance of regular expressions.

In Chapter 3, we conduct a comprehensive evaluation of four representative log parsers (i.e., SLCT [88], IPLoM [62], LKE [35], LogSig [85]). We do not consider source code-based log parsing [94], because, in many cases, the source code is inaccessible (e.g., in third party libraries). By using five real-world log datasets with over 10 million raw log messages, we evaluate the log parsers' performance in terms of accuracy (i.e., F-measure [66, 9]), efficiency (i.e., execution time), and effectiveness on a log mining task (i.e., anomaly detection [94] evaluated by detected anomaly and false alarm). We determine that, although the overall accuracy of these log parsing methods is high, they are not robust across all datasets. When logs grow to a large scale (e.g., 200 million log messages), these parsers fail to complete in reasonable time (e.g., one hour), and most cannot handle such data on a single computer. We also find that parameter tuning costs considerable time for these methods, because parameters tuned on a sample dataset of small size cannot be directly employed on a large dataset.

To address these problems, in this chapter, we propose a parallel log parsing method, called POP, that can accurately and efficiently parse large-scale log data. Similar to previous papers [88, 62, 35, 85], POP assumes the input is single-line logs, which is the common case in practice. To improve accuracy in log parsing, we employ iterative partitioning rules for candidate event generation and hierarchical clustering for event type refinement. To improve efficiency in processing large-scale data, we design POP with linear time complexity in terms of log size, and we further parallelize its computation on top of Spark, a large-scale data processing platform.

We evaluate POP on both real-world datasets and large-scale

synthetic datasets with 200 million lines of raw log messages. The evaluation results show the capability of POP in achieving accuracy and efficiency. Specifically, POP can parse all the real-world datasets with the highest accuracy compared with the existing methods. Moreover, POP can parse our synthetic HDFS (Hadoop Distributed File System) dataset in 7 min, whereas SLCT requires 30 min, and IPLoM, LKE, and LogSig fail to terminate in reasonable time. Moreover, parameter tuning is easy in POP because the parameters tuned on small sample datasets can be directly applied to large datasets while preserving high parsing accuracy.

- It presents the design and implementation of a parallel log parsing method (POP), which can parse large-scale log data accurately and efficiently.
- We evaluate POP on both synthetic and real-world datasets, which determines its superiority in accuracy, efficiency, and effectiveness in log mining tasks.
- The source code of POP has been publicly released [14], allowing for easy use by practitioners and researchers for future study.

4.2 Parallel Log Parsing (POP)

From the implementation and systematic study of log parsers introduced in Section 2, we observe that a good log parsing method should fulfill the following requirements: (1) Accuracy. The parsing accuracy (i.e., F-measure) should be high. (2) Efficiency. The running time of a log parser should be as short as possible. (3) Robustness. A log parsing method needs to be consistently accurate and efficient on logs from different systems.

Thus, we design a ***p****arallel l****o****g* ***p****arsing* method, namely **POP**, to fulfill the above requirements. POP preprocesses logs with simple

Figure 4.2: Overview of Log Parsing

domain knowledge (step 1). It then hierarchically partitions the logs into different groups based on two heuristic rules (step 2 and 3). For each group, the constant parts are extracted to construct the log event (step 4). Finally, POP merges similar groups according to the result of hierarchical clustering on log events (step 5). We design POP on top of Spark [11, 99], a large-scale data processing platform using the parallelization power of computer clusters, and all computation-intensive parts of POP are designed to be highly parallelizable.

[18:03:38] chrome.exe, 4381 bytes sent, 6044 bytes received, lifetime **09:14**
[16:49:08] chrome.exe, 464 bytes sent, 1101 bytes received, lifetime **<1 sec**

Figure 4.3: Proxifier Log Samples

4.2.1 Step 1: Preprocess by Domain Knowledge

According to our study on the existing log parsers, simple preprocessing using domain knowledge can improve parsing accuracy, so raw logs are preprocessed in this step. POP provides two preprocessing functions. First, POP prunes variable parts according to simple regular expression rules provided by developers, for example, removing block ID in Fig. 4.2 by "blk_[0-9]+". For all datasets used in our experiment, at most two rules are defined on a dataset. This function can delete variable parts that can be easily identified with domain knowledge. Second, POP allows developers to manually specify log events based on regular expression rules. This is useful because developers intend to put logs with certain properties into the same partition in some cases. For example, Fig. 4.3 contains two log messages from Proxifier dataset. The two logs will be put into the same partition by most of the log parsing methods. However, developers may want to count the session with less than 1 second lifetime separately. In this case, POP can easily extract the corresponding logs based on the regular expression ".*<1 sec.*". Note that the simple regular expressions used in this step require much less human effort than those complex ones used by traditional methods to match the whole log messages.

4.2.2 Step 2: Partition by Log Message Length

In this step, POP partitions the remaining logs into nonoverlapping groups of logs. POP puts logs with the same log message length into the same group. By log message length, we mean the number of tokens in a log message. This heuristic, which is also used by IPLoM [64], is based on the assumption that logs with the same log

event will likely have the same log message length. For example, log event "Verification succeeded for *" from HDFS dataset contains 4 tokens. It is intuitive that logs having this log event share the same log message length, such as "Verification succeeded for blk_1" and "Verification succeeded for blk_2". This heuristic rule is considered coarse-grained, so it is possible that log messages in the same group have different log events. "Serve block * to *" and "Deleting block * file *" will be put into the same group in step 2, because they both contain 5 tokens. This issue is addressed by a fine-grained heuristic partition rule described in step 3. Besides, it is possible that one or more variable parts in the log event contain variable length, which invalidates the assumption of step 2. This will be addressed by hierarchical clustering in step 5.

4.2.3 Step 3: Recursively Partition by Token Position

In step 3, each group is recursively partitioned into subgroups, where each subgroup contains logs with the same log event (i.e., same constant parts). This step assumes that if the logs in a group having the same log event, the tokens in some token positions should be the same. For example, if all the logs in a group have log event "Open file *", then the tokens in the first token position of all logs should be "Open". We define *complete token position* to guide the partitioning process.

Notations: Given a group containing logs with log message length n, there are n token positions. All tokens in token position i form a token set TS_i, which is a collection of distinct tokens. The cardinality of TS_i is defined as $|TS_i|$. A token position is *complete* if and only if $|TS_i| = 1$, and it is defined as a *complete token position*. Otherwise, it is defined as an *incomplete token position*.

Our heuristic rule is to recursively partition each group until all the resulting groups have enough complete token positions. To evaluate whether complete token positions are enough, we define

Group Goodness (GG) as following.

$$GG = \frac{\#CompleteTokenPositions}{n}. \tag{4.1}$$

A group is a *complete group* if $GG > GS$, where GS stands for *Group Support*, a threshold assigned by developers. Otherwise, the group is an incomplete group. In this step, POP recursively partitions the groups if the current group is not a complete group.

Algorithm 1 provides the pseudo code of step 3. POP regards all groups from step 2 as incomplete groups (line 1). Incomplete groups are recursively partitioned by POP to generate a list of complete groups (lines 4∼24). For each incomplete group, if it already contains enough complete token positions, it is moved to the complete group list (lines 6∼8). Otherwise, POP finds the split token position, which is the token position with the lowest cardinality among all incomplete token positions. Because of its lowest cardinality, tokens in the split token position are most likely to be constants. Then POP calculates *Absolute Threshold (AT)* and *Relative Threshold (RT)* (line 11∼12). A token position with smaller *AT* and *RT* is more likely to contain constants. For example, in Fig. 4.4, column (i.e., token position) 1 and 2 have smaller *AT* (2) and *RT* (0.5), so they are more likely to contain constants compared with column 3, whose *AT* is 4 and *RT* is 1. Note that we only need to calculate *AT* and *RT* for the split token position. We demonstrate *AT* and *RT* for all the columns in Fig. 4.4 for better explanation. Thus, POP regards the tokens as variables only when both *AT* and *RT* are

An Incomplete Group		
1. Send	to	10.10.35.01
2. Receive	from	10.10.35.02
3. Receive	from	10.10.35.03
4. Send	to	10.10.35.04

Calculate AT, RT →

Column	AT	RT
1	2	0.5
2	2	0.5
3	4	1

Figure 4.4: An Example of *AT*, *RT* Calculation

Algorithm 1 POP Step 3: Recursively partition each group to complete groups.

Input: a list of log groups from step 2: $logGroupL$; and algorithm parameters: $GS, splitRel, splitAbs$

Output: a list of complete groups: $completeL$

1: $incompleteL \leftarrow logGroupL$
2: $completeL \leftarrow List(\,)$ ▷ Initialize with empty list
3: $curGroup \leftarrow$ first group in $incompleteL$
4: **repeat**
5: Set $n \leftarrow |curGroup|$
6: **if** ISCOMPLETE($curGroup, GS$)$= true$ **then**
7: Add $curGroup$ to $completeL$
8: Remove $curGroup$ from $incompleteL$
9: **else**
10: Find the split token position s
11: Compute $AT \leftarrow |TS_s|$
12: Compute $RT \leftarrow |TS_s|/n$
13: **if** $AT > splitAbs$ *and* $RT > splitRel$ **then**
14: Add $curGroup$ to $completeL$
15: Remove $curGroup$ from $incompleteL$
16: **else**
17: Partition $curGroup$ to several $resultGroup$ based on the token value in split token position
18: **for all** $resultGroup$ **do**
19: **if** ISCOMPLETE($resultGroup, GS$)$= true$ **then**
20: Add $resultGroup$ to $completeL$
21: **else**
22: Add $resultGroup$ to $incompleteL$
23: $curGroup \leftarrow$ next group in $incompleteL$
24: **until** $incompleteL$ is empty

25: **function** ISCOMPLETE($group, gs$)
26: Compute token sets for token positions in $group$
27: Compute GG ▷ by Equation 4.1
28: **if** $GG > gs$ **then**
29: **return** $true$
30: **else**
31: **return** $false$

larger than manually defined thresholds (i.e., $splitAbs$ and $splitRel$ respectively). If all tokens in the split token position are variables, POP moves the current group to the complete group list, because it could not be further partitioned (line 13∼15). Otherwise, POP partitions the current group into $|TS_s|$ resulting groups based on the token value in the split token position (line 17). Among all the result groups, the complete groups are added into the complete group list, while the incomplete ones are added to the incomplete group list for further partitioning (line 18∼22). Finally, the complete group list is returned, where logs in each group share the same log event type.

4.2.4 Step 4: Generate Log Events

At this point, the logs have been partitioned into nonoverlapping groups by two heuristic rules. In this step, POP scans all the logs in each group and generates the corresponding log event, which is a line of text containing constant parts and variable parts. The constant parts are represented by tokens and the variable parts are represented by wildcards. To decide whether a token is a constant or a variable, POP counts the number of distinct tokens (i.e., $|TS|$) in the corresponding token position. If the number of distinct tokens in a token position is one, the token is constant and will be outputted to the corresponding token position in a log event. Otherwise, a wildcard is outputted.

4.2.5 Step 5: Merge Groups by Log Event

To this end, logs have been partitioned into nonoverlapping complete groups, and each log message is matched with a log event. Most of the groups contain logs that share the same log event. However, some groups may be over-parsed because of suboptimal parameter setting, which causes false negatives. Besides, it is possible that some variable parts in a log event have variable length,

which invalidates the assumption in step 2. This also brings false negatives.

To address over-parsing and further improve parsing accuracy, in this step, POP employs hierarchical clustering [36] to cluster similar groups based on their log events. The groups in the same cluster will be merged, and a new log event will be generated by calculating the Longest Common Subsequence (LCS) [61] of the original log events. This step is based on the assumption that if logs from different groups have the same log event type, the generated log event texts from these groups should be similar. POP calculates Manhattan distance [52] between two log event text to evaluate their similarity. Specifically,

$$d(a,b) = \sum_{i=1}^{N} |a_i - b_i|, \tag{4.2}$$

where a and b are two log events, N is the number of all constant token values in a and b, and a_i means the occurrence number of the i-th constant token in a. We use Manhattan distance because it assigns equal weight to each dimension (i.e., constant). This aligns with our observation that all constants are of equal importance in log parsing. Besides, Manhattan distance is intuitive, which makes parameter tuning easier. POP employs complete linkage [34] to evaluate the distance between two clusters, because the resulted clusters will be compact, which avoids clustering dissimilar groups together. The only parameter in this step is $maxDistance$, which is the maximum distance allowed when the clustering algorithm attempts to combine two clusters. The algorithm stops when the minimum distance among all cluster pairs is larger than $maxDistance$.

4.2.6 Implementation

To make POP efficient in large-scale log analysis, we build it on top of Spark [11, 99], a large-scale data processing platform [55].

Specifically, Spark runs iterative analysis programs with orders of magnitude faster than Hadoop Mapreduce [2]. The core abstraction in Spark is Resilient Distributed Datasets ($RDDs$), which are fault-tolerant and parallel data structures representing datasets. Users can manipulate RDDs with a rich set of Spark operations called $transformations$ (e.g., map, filter) and $actions$ (e.g, reduce, aggregate). Calling transformations on an RDD generates a new RDD, while calling actions on an RDD reports calculation result to users. Spark employs lazy evaluation, so that transformations on RDDs will not be executed until an action is called. At that time, all preceding transformations are executed to generate the RDD, where the action is then evaluated. We build POP on top of Spark because it is good at parallelizing identical computation logic on each element of a dataset, and it directly uses the output of one step in memory as the input to another. In our case, an RDD can represent a log dataset, where each element is a log message. POP can be parallelized by transformations and actions, because each POP step requires computation-intensive tasks that cast identical computation logic to every log message. To parallelize these tasks, we invoke Spark operations with specially designed functions describing the computation logic. In the following, we will introduce the Spark operations we applied for the five POP steps.

The implementation of POP on Spark is illustrated in Fig. 4.5. The five rounded rectangles at the bottom represent the five steps of POP, where the numbered arrows represent the interactions between the main program and the $Spark\ cluster$. The main program is running in $Spark\ driver$, which is responsible for allocating Spark tasks to $workers$ in the Spark cluster. For a POP Spark application, in step 1, we use $textFile$ to load the log dataset from a distributed file system (e.g., HDFS) to Spark cluster as an RDD (arrow 1). Then, we use map to preprocess all log messages with a function as input describing the preprocessing logic on single log message (arrow 2). After preprocessing, we $cache$ the preprocessed log

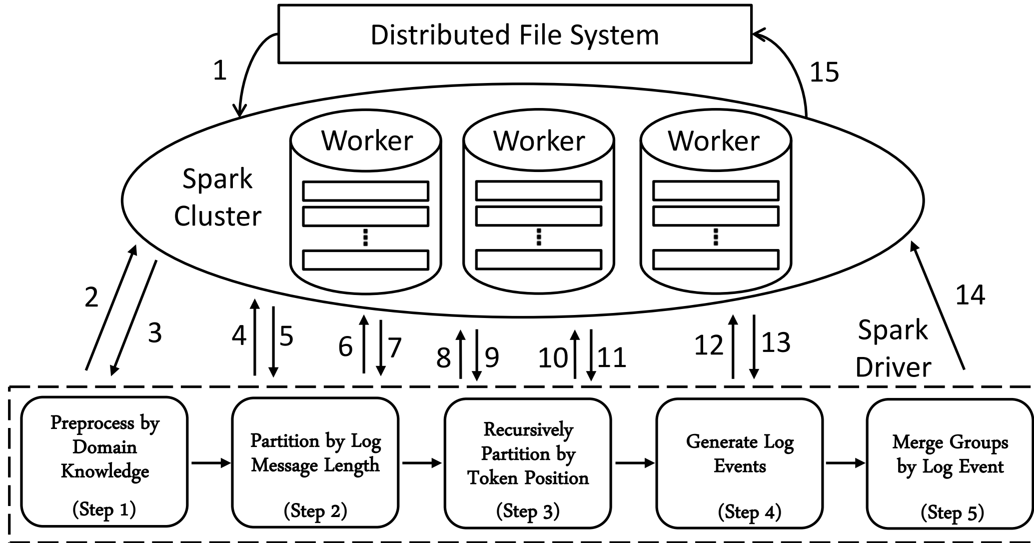


Figure 4.5: Overview of POP Implementation

messages in memory and return an RDD as the reference (arrow 3). In step 2, we use $aggregate$ to calculate all possible log message length values (arrow 4) and return them as a list (arrow 5). Then for each value in the list, we use $filter$ to extract log messages with the same log message length (arrow 6), which is returned as an RDD (arrow 7). Now we have a list of RDDs. In step 3, for each RDD, we employ $aggregate$ to form the token sets for all token positions (arrow 8∼9) as described in Section 4.2.3. Based on the token sets and pre-defined thresholds, the driver program decides whether current RDD could be further partitioned or not. If yes, we use $filter$ to generate new RDDs and add them into the RDD list (arrow 10∼11). Otherwise, we remove it from the list and pass the RDD to step 4. In step 4, we use $reduce$ to generate log events for all RDDs (arrow 12∼13). When all log events have been extracted, POP runs hierarchical clustering on them in main program. We use $union$ to merge RDDs based on the clustering result (arrow 14). Finally, merged RDDs are outputted to the distributed file system by $saveAsTextFile$ (arrow 15).

The implementation of this specialized POP is non-trivial. First,

Spark provides more than 80 operations and this number is increasing quickly due to its active community. We need to select the most suitable operations to avoid unnecessary performance degradation. For example, if we use $aggregateByKey$ in step 2 and step 3 instead of $aggregate$, the running time will be one order of magnitude longer. Second, we need to design tailored functions as input for Spark operations, such as $aggregate$ and $reduce$. Though we use $aggregate$ in both step 2 and step 3, different functions have been designed. The source code of POP has been release [14] for reuse. Note that the existing log parser can also be parallelized, but they require non-trivial efforts.

4.3 Evaluation

This section presents our evaluation methodology first. Then we evaluate the performance of POP in terms of its accuracy, efficiency, and effectiveness on subsequent log mining tasks in different subsections. For each of these three evaluation items, we first briefly review the evaluation study result obtained in Chapter 3; then we analyze the evaluation results of POP. Finally, we present parameter sensitivity analysis and conclude with experimental observations.

4.3.1 Study Methodology

Log Datasets. We used five real-world log datasets, including supercomputer logs (BGL and HPC), distributed system logs (HDFS and Zookeeper), and standalone software logs (Proxifier). Table 4.1 provides the basic information of these datasets. *Log Size* column describes the number of raw log messages, while *#Events* column is the number of log event types. Since companies are often reluctant to release their system logs due to confidentiality, logs are scarce data for research. Among the five real-world log datasets in Table 4.1, three log datasets are obtained from their

authors. Specifically, BGL is an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs (LLNL), with 131,072 processors and 32,768GB memory [73]. HPC is also an open dataset with logs collected from a high performance cluster at Los Alamos National Laboratory, which has 49 nodes with 6,152 cores and 128GB memory per node [57]. HDFS logs are collected in [94] by engaging a 203-node cluster on Amazon EC2 platform. To enrich the log data for evaluation purpose, we further collected two datasets: one from a desktop software Proxifier, and the other from a Zookeeper installation on a 32-node cluster in our lab. In particular, the HDFS logs from [94] have well-established anomaly labels, each of which indicates whether a block has anomaly operations. Specifically, the dataset with over 10 million log messages records operations on 575,061 blocks, among which 16,838 are anomalies.

Log Parser Implementation. Among the four studied log parsing methods, we only find an open-source implementation on SLCT in C language. To enable our evaluations, we have implemented the other three log parsing methods in Python and also wrapped up SLCT as a Python package. Currently, all our implementations have been open-source as a toolkit on Github [14].

Evaluation Metric. We use F-measure [66, 9], a commonly-used evaluation metric for clustering algorithms, to evaluate the parsing accuracy of log parsing methods. The definition of parsing accuracy is as the following.

$$Parsing\ Accuracy = \frac{2 * Precision * Recall}{Precision + Recall}, \tag{4.3}$$

where $Precision$ and $Recall$ are defined as follows:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}, \tag{4.4}$$

where a true positive (TP) decision assigns two log messages with the same log event to the same log group; a false positive (FP)

Table 4.1: Summary of Our System Log Datasets

System	Log Size	Length	#Events	Description
BGL	4,747,963	10∼102	376	BlueGene/L Supercomputer
HPC	433,490	6∼104	105	High Performance Cluster (Los Alamos)
HDFS	11,175,629	8∼29	29	Hadoop Distributed File System
Zookeeper	74,380	8∼27	80	Distributed System Coordinator
Proxifier	10,108	10∼27	8	Proxy Client

decision assigns two log messages with different log events to the same log group; and a false negative (FN) decision assigns two log messages with the same log event to different log groups. If the logs are under-partitioning, the precision will be low because it leads to more false positives. If a log parsing method over-partitions the logs, its recall will decrease because it has more false negatives. Thus, we use F-measure, which is the harmonic mean of precision and recall, to represent parsing accuracy. To obtain the ground truth for the parsing accuracy evaluation, we split the raw log messages into different groups with the help of manually-customized regular expressions.

Experimental Setting. The experiments of systematic evaluation on existing log parsers are run on a Linux server with Intel Xeon E5-2670v2 CPU and 128GB DDR3 1600 RAM, running 64-bit Ubuntu 14.04.2 with Linux kernel 3.16.0. Experiments of POP

are run on Spark 1.6.0 with YARN as the cluster controller on 32 physical machines. The cluster has 4TB memory and 668 executors in total. All 32 physical machines are inter-connected with 10Gbps network switch. In our experiment, unless otherwise specified, we use 16 executors, each of which has 25G memory and 5 executor cores. We set Kryo as the Spark serializer because it is significantly faster and more compact than the default one [15]. The parameter setting follows the experience of Cloudera [78], a leading software company that provides big data software, services and supports. To avoid bias, each experiment is run 10 times and the averaged result is reported.

4.3.2 Accuracy of POP

In this section, we first evaluate the accuracy of POP. Then we study whether POP can consistently obtain high accuracy on large datasets if parameters tuned on small datasets are employed.

Parsing Accuracy

In this section, we first evaluate the parsing accuracy of existing parsers with and without preprocessing. Then the parsing accuracy of POP is explained.

Similar to the existing work [85], we randomly sample 2k log messages from each dataset in our evaluation, LKE and LogSig cannot parse large log datasets in reasonable time (e.g., LogSig requires 1 day to parse the entire BGL data). For each experiment, we use the same 2k log messages for all 10 executions. These 2k datasets have been released on our project website [14]. The results are reported in Table 4.2 (i.e., the first number in each cell). As shown in the table, the accuracy of existing log parsers is larger than 0.8 in many cases. Besides, the overall accuracy on HDFS, Zookeeper and Proxifier datasets is higher than that on BGL and HPC. We find that this is mainly because BGL and HPC logs involve

much more event types, and they have more various log length range compared with HDFS, Zookeeper and Proxifier. For the preprocessing step, obvious numerical parameters in log messages (i.e., IP addresses in HPC&Zookeeper&HDFS, core IDs in BGL, and block IDs in HDFS) are removed. Preprocessing is mentioned in LKE and LogSig, but its effectiveness has not been studied.

In Table 4.2, the second number in each cell represents the accuracy of log parsing methods on preprocessed log data. In most cases, accuracy of parsing is improved. Preprocessing greatly increases the accuracy SLCT on BGL, LKE on HDFS, and LogSig on BGL (in bold). However, preprocessing could not improve the accuracy of IPLoM. This is mainly because IPLoM considers preprocessing internally in its four-step process. The parsing accuracy of existing parsers is summarized as follows.

Simple preprocessing using domain knowledge (e.g., removal of IP address) improves log parsing accuracy. With preprocessing, existing log parsers can achieve high overall accuracy. But none of them consistently generates accurate parsing results on all datasets.

To evaluate the accuracy of POP, we employ it to parse the same 2k datasets. For dataset BGL, HPC, HDFS and Zookeeper, we set GS to 0.6, $splitAbs$ to 10, $splitRel$ to 0.1, $maxDistance$ to 0. Parameter tuning is intuitive because all these parameters have physical meanings. Developer can easily find the suitable parameter setting with basic experience on datasets. For dataset Proxifier, we set GS to 0.3, $splitAbs$ to 5, $splitRel$ to 0.1, $maxDistance$ to 10. The parameter setting of Proxifier is different because it contains much fewer log events (i.e., 8 as described in Table 4.1) compared with others. Besides, we extract log messages containing text "$<$1 sec" in step 1 of POP, which simulates the practical condition described in Section 4.2.1.

The results are presented in the last line of Table 4.2. We observe that POP delivers the best parsing accuracy for all these datasets. For datasets that has relatively few log events (e.g., HDFS and Proxifier),

Table 4.2: Parsing Accuracy of Log Parsing Methods (Raw / Preprocessed)

	BGL	HPC	HDFS	Zookeeper	Proxifier
SLCT	**0.61/0.94**	0.81/0.86	0.86/0.93	0.92/0.92	0.89/-
IPLoM	0.99/0.99	0.64/0.64	0.99/1.00	0.99/0.99	0.84/-
LKE	0.67/0.70	0.17/0.17	**0.57/0.96**	0.78/0.82	0.81/-
LogSig	**0.26/0.98**	0.77/0.87	0.91/0.93	0.96/0.99	0.84/-
POP	**0.99**	**0.95**	**1.00**	**0.99**	**1.00**

its parsing accuracy is 1.00, which means all the logs can be parsed correctly. For datasets that has relatively more log events, POP still delivers very high parsing accuracy (0.95 for HPC). POP has the best parsing accuracy because of three reasons. First, POP will recursively partition each log group into several groups until they become complete groups. Compared with other log parsers based on heuristic rules (e.g., SLCT), POP provides more fine-grained partitioning. Second, POP merges similar log groups based on the extracted log event, which amends over-partitioning. Third, POP allows developers to manually extract logs with certain properties, which reduces noise for the partitioning process.

Parameter Tuning

The accuracy of log parsers is affected by parameters. For large-scale log data, it is difficult to select the most suitable parameters by trying different values, because each run will cause a lot of time. Typically, developers will tune the parameters on a small sample dataset and directly apply them on large-scale data.

To evaluate the feasibility of this approach, we sampled 25 datasets from the original real-world datasets. Table 4.3 shows the number of raw log messages in these 25 sample datasets, where each row presents 5 sample datasets generated from a real-world dataset.

We apply parameters tuned on 2k datasets. In Fig. 4.6, we evaluate the accuracy of the log parsers on the datasets presented in Table 4.3 employing these parameters. The results show that IPLoM

Table 4.3: Log Size of Sample Datasets

BGL	400	4k	40k	400k	4m
HPC	600	3k	15k	75k	375k
HDFS	1k	10k	100k	1m	10m
Zookeeper	4k	8k	16k	32k	64k
Proxifier	600	1200	2400	4800	9600

performs consistently in most cases except a 0.15 drop on Proxifier. SLCT varies a lot on HPC and Proxifier. The accuracy of LKE is volatile in Zookeeper because of its aggressive clustering strategy. LogSig obtains consistent accuracy on datasets with limited types of events, but its accuracy fluctuates severely on datasets with many log events (i.e., BGL and HPC). The results of existing parsers in this section are summarized as follows.

Parameter tuning is time-consuming for existing log parsing methods except IPLoM, because they could not directly use parameters tuned on small sampled data for large datasets.

The experimental results of POP are shown in Table 4.4 and Fig. 4.6. We observe that the accuracy of POP is very consistent for all datasets. The accuracy on Zookeeper is 0.99 for all 5 sampling levels, which indicates the parameters tuned on 2k sample dataset lead to nearly the same parsing results. For HPC, HDFS and Proxifier, the fluctuation of the accuracy is at most 0.02, while the accuracy is at least 0.95. For BGL, the accuracy has a 0.1 drop for the last two sampling levels. But POP can still obtain 0.89 accuracy in these two levels, while 0.1 is not a large drop compared with existing parsers in Fig. 4.6. Compared with existing methods, POP is the only parser that obtains high accuracy consistently on all datasets using the parameters tuned on small sampled data.

4.3.3 Efficiency of POP

In this section, we evaluate the efficiency of POP. Specifically, we first measure the running time of these log parsers on 25 sampled

Table 4.4: Parsing Accuracy of POP on Sample Datasets in Table 4.3 with parameters tuned on 2k datasets

BGL	0.98	0.99	0.99	0.89	0.89
HPC	0.95	0.97	0.96	0.96	0.97
HDFS	1.00	0.99	0.99	0.99	0.99
Zookeeper	0.99	0.99	0.99	0.99	0.99
Proxifier	1.00	1.00	1.00	0.99	0.99

datasets with varying number of log messages (i.e., log size) in Table 4.3. Second, we evaluate the running time of these log parsers on synthetic datasets containing over 200 million log messages, which is comparable to large-scale modern production systems[68].

Note that running time in this paper means the time used to run log parsers (i.e., training time). In addition to training time, we measure the efficiency for parsing a new log message, which is 173μs for BGL, 108μs for HPC, 36μs for HDFS, 29μs for Zookeeper, and 20μs for Proxifier. The matching process relies on regular expressions, thus its time depends on the number of log events and their lengths. The matching time is similar for different log parsers.

Running Time on Real-World Datasets

In Fig. 4.7, we evaluate the running time of the log parsing methods on all datasets by varying the number of raw log messages (i.e., log size). Notice that as the number of raw log messages increases, the number of events becomes larger as well (e.g., 60 events in BGL400 while 206 events in BGL40k). Fig. 4.7 is in logarithmic scale, so we can observe the time complexity of these log parsers from the slope of the lines. As show in the figure, the running time of SLCT and IPLoM scale linearly with the number of log messages. They both could parse 10 million HDFS log messages within five minutes. However, as the slopes show, their running time increases fast as the log size becomes larger, because they are limited by the

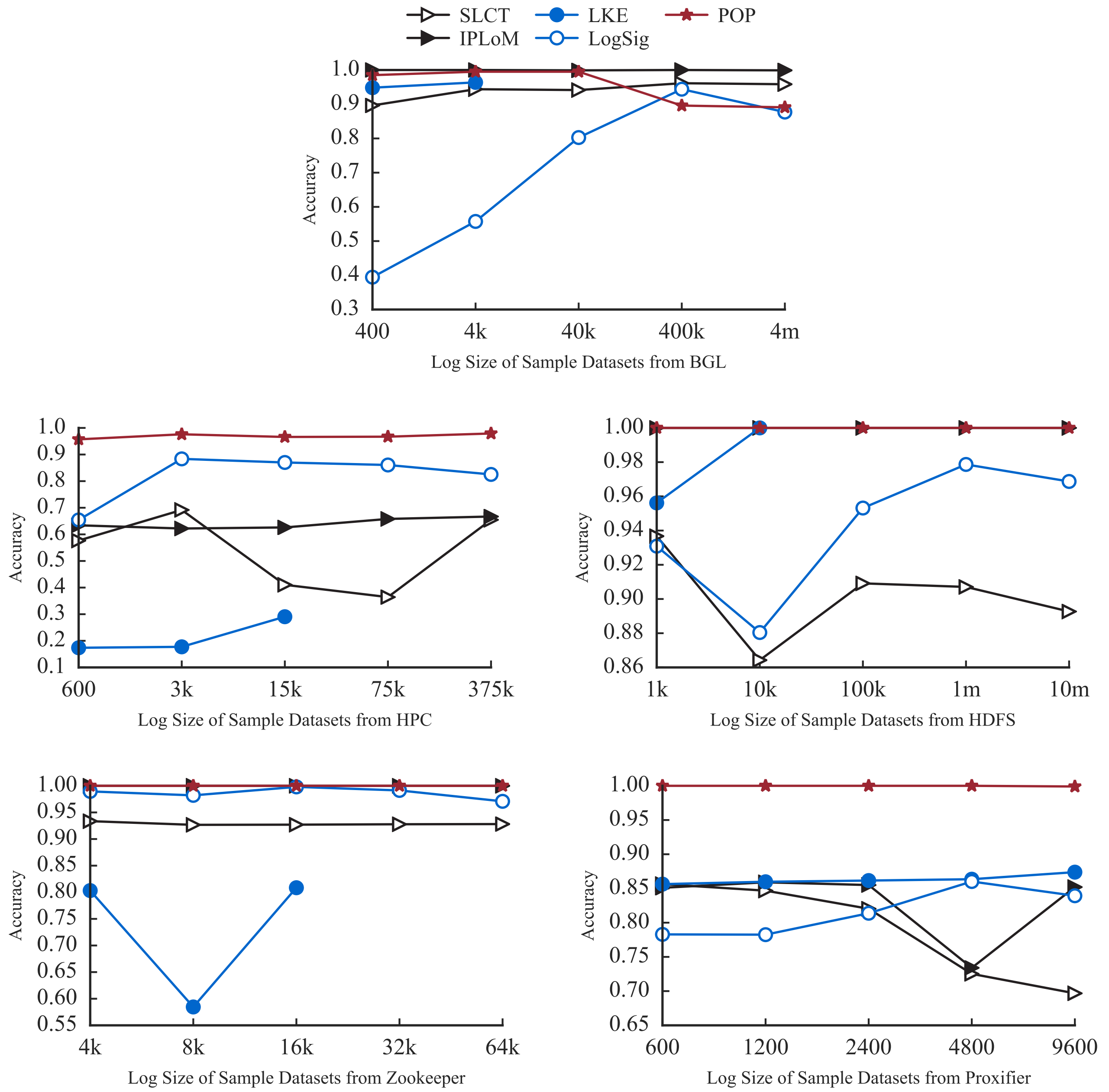


Figure 4.6: Parsing Accuracy on Datasets in Different Size

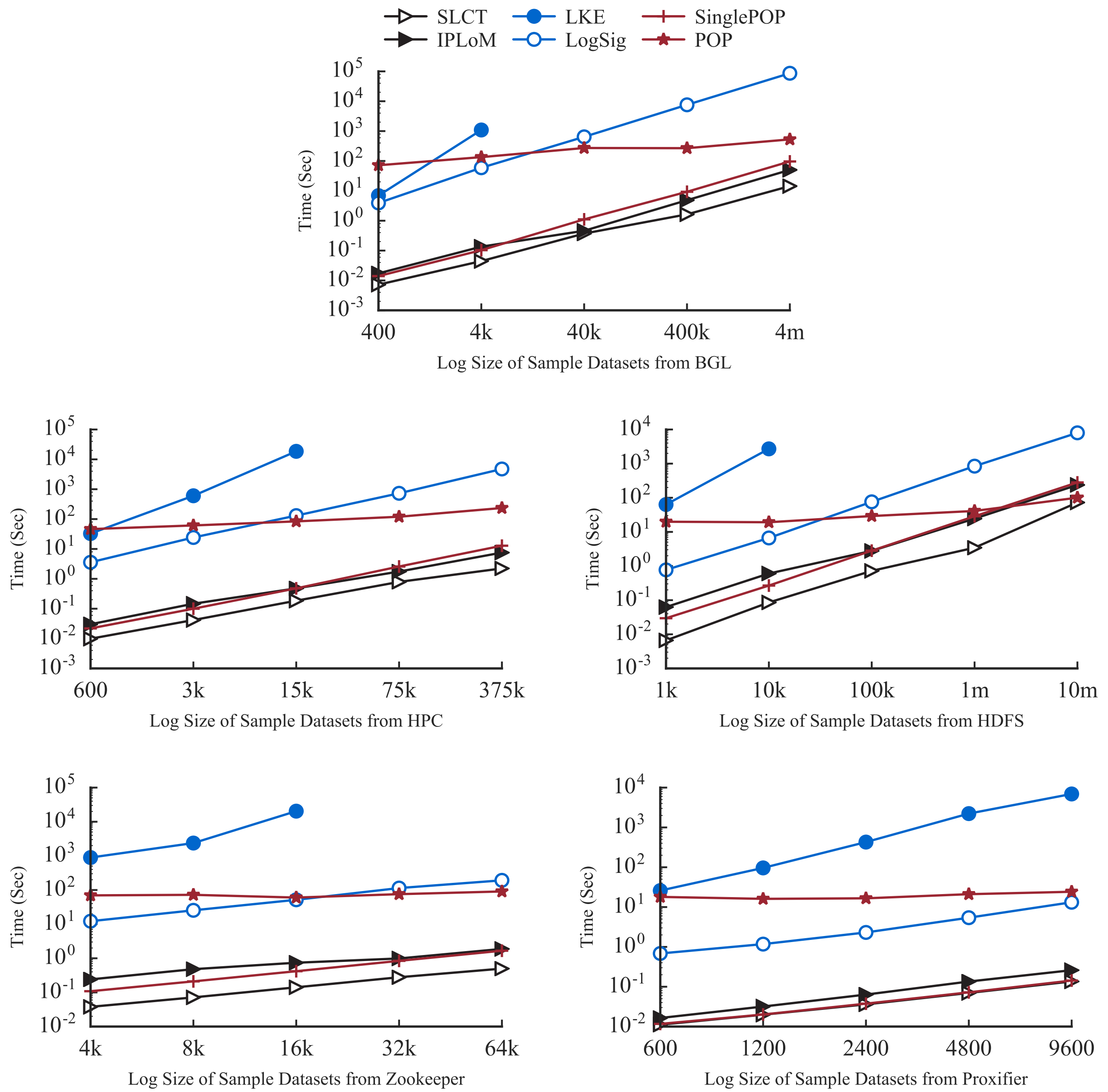


Figure 4.7: Running Time of Log Parsing Methods on Datasets in Different Size

computing power of a single computer. The fast increasing speed can lead to inefficient parsing on production level log data (e.g., 200 million log messages). The running time of LogSig also scales linearly with the number of log messages. However, it requires much running time (e.g, 2+ hours for 10m HDFS log messages), because its clustering iterations are computation-intensive and its word pair generation step is time-consuming. The time complexity of LKE is $O(n^2)$, where n is the number of raw log messages, which makes it unable to handle some real-world log data, such as BGL4m and HDFS10m. Running time of some LKE experiments is not plotted because LKE could not terminate in reasonable time (i.e., days or even weeks). The running time of existing parsers on real-world datasets is summarized as follows.

Clustering-based log parsers require much running time on real-world datasets. Heuristic rule-based log parsers are more efficient, but their running time increases fast as the log size becomes larger. These imply the demand for parallelization.

The time complexity of POP is $O(n)$, where n is the number of raw log messages. In step 1, step 2 and step 4, POP traverses all log messages once so the time complexity for these steps are all $O(n)$. In step 3, POP may scan some log messages more than once due to recursion. However, in the case of log parsing, the recursion depth can be regarded as a constant because it will not increase as the number of log messages, which remains small in all our datasets. Thus, the time complexity of step 3 is also $O(n)$. Finally, the time complexity of step 5 is $O(m^2 \log m)$, where m is the number of log events. We do not consider it in the time complexity of POP, because m is far less than n. So the time complexity of POP is $O(n + n + n + n + m^2 \log m) = O(n)$.

The "SinglePOP" lines represent the running time of the nonparallel implementation of POP on different datasets. We can observe that the running time of SinglePOP is even shorter than the parallel implementation of POP. Because the nonparallel implementation of

POP does not require any data transportation between nodes, which is required by parallel programs. Besides, the parallel implementation needs to deploy the runtime environment (e.g., set up the nodes that will be used) at the beginning, though automatically, will cost some constant time.

The experimental results of POP are presented in Table 4.5 and Fig. 4.7. Fig. 4.7 shows that POP has the slowest increasing speed of running time as the log size becomes larger. Its increasing speed is even much better (i.e., slower) than linear parsers (i.e., SLCT, IPLoM, LogSig). For a few cases, the running time of POP even decreases when the log size becomes larger. This is mainly caused by two reasons. First, a larger dataset could benefit more from parallelization than a smaller one. Second, it is possible that a smaller dataset requires deeper recursion in step 3 of POP, which increases its running time. Compared with the existing methods, POP enjoys the slowest running time increase because of its $O(n)$ time complexity and its parallelization mechanism. It can parse a large amount of log messages very fast (e.g., 100 seconds for 10 million HDFS log messages). Although its running time is slower than IPLoM and SLCT in some cases, POP turns out to be more efficient for two reasons. First, as we can observe from Fig. 4.7, the running time increase of POP is the slowest, so POP will be faster than other log parsers when log size is larger. For example, POP is faster than IPLoM on 10m HDFS dataset. Second, the efficiency of IPLoM and SLCT is limited by computing power or/and memory of single computer, while POP is able to utilize multiple computers. Note that with parallelization, the running time will be reduced as the number of computers increases. Thus our parser may obtain even better performance with more computers. However, this increase will also slow down finally because of the trade-off between reduced computation time and extra data transmission time.

Table 4.5: Running Time of POP (Sec) on Sample Datasets in Table 4.3

BGL	71.87	134.48	271.98	268.12	527.63
HPC	46.24	61.29	83.81	119.81	234.92
HDFS	19.82	19.17	29.14	41.03	100.58
Zookeeper	69.62	72.22	60.07	75.56	90.69
Proxifier	18.00	16.08	16.60	21.07	24.22

Running Time on Large-Scale Synthetic Datasets

In this section, we evaluate the running time of log parsers on very large synthetic datasets, which are randomly generated from BGL and HDFS. These two datasets are representative because they include log datasets with a lot and a few log events respectively. BGL has more than 300 log events, while HDFS has 29. The synthetic datasets are generated from the real-world datasets. For example, to generate a 200m synthetic dataset from HDFS dataset, we randomly select a log message from the dataset each time, and repeat this random selection process 200 million times. Fig. 4.8 presents the experimental results in linear scale.

In this figure, a result is neglected if its running time is larger than one hour, because we want to evaluate the effectiveness of these log parsers in production environment (e.g., 120∼200 million log messages per hour [68]). Thus, experimental results of SLCT, IPLoM and POP are plotted, while LKE and LogSig require more than one hour on these datasets. The running time increase of IPLoM is the fastest among the plotted three. It requires more than an hour for two datasets generated from HDFS; therefore, they are not plotted. Besides, IPLoM requires more than 16G memory when the synthetic dataset contains 30m or more log messages for both BGL and HDFS. Because IPLoM needs to load the whole dataset into memory, and it creates extra data of comparable size in runtime. SLCT is more efficient than IPLoM, and it requires the least time on BGL datasets. SLCT only requires two passes across all log data, and it is implemented in C instead of Python. However, its

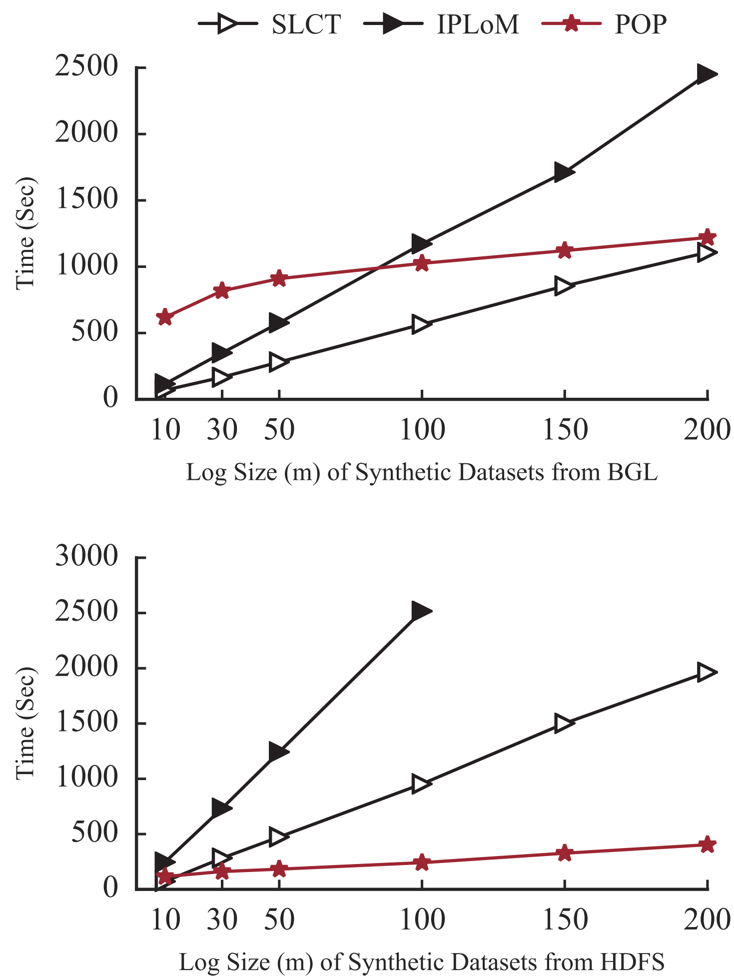


Figure 4.8: Running Time on Synthetic Datasets

running time increases fast as the log size becomes larger, because SLCT is limited by the computing power of single computer. The running time of existing parsers on large-scale synthetic datasets is summarized as follows.

Clustering-based log parsers cannot handle large-scale log data. Heuristic rule-based log parsers are efficient, but they are limited by the computing power or/and memory of a single computer.

For POP, we use 64 executors on BGL datasets and 16 executors on HDFS datasets, each of which has 16G memory and 5 executor

Table 4.6: Anomaly Detection with Different Log Parsing Methods (16,838 Anomalies)

	Parsing Accuracy	Reported Anomaly	Detected Anomaly	False Alarm
SLCT	0.83	18,450	10,935 (64%)	7,515 (40%)
LogSig	0.87	11,091	10,678 (63%)	413 (3.7%)
IPLoM	0.99	10,998	10,720 (63%)	278 (2.5%)
POP	0.99	10,998	10,720 (63%)	278 (2.5%)
Ground truth	1.00	11,473	11,195 (66%)	278 (2.4%)

cores. We use more executors on BGL datasets because they require more recursive partitioning in step 3. We set 16G memory because this is a typical memory setting for a single computer. We observe that POP has the slowest growth speed among all three methods. Besides, POP requires the least running time for HDFS datasets. Though SLCT requires less time for BGL datasets, its running time increases faster than POP, which is shown by their comparable results on 200m log message dataset generated from BGL. Thus, POP is the most suitable log parser for large-scale log analysis, given that the size of logs will become even larger in the future.

4.3.4 Effectiveness of POP on Log Mining: A Case Study

Log mining tasks usually accept structured data (e.g., matrix) as input and report mining results to developers, as described in Fig. 4.1. If a log parser is inaccurate, the generated structured logs will contain errors, which can further ruin the input matrix of subsequent log mining tasks. A log mining task with erroneous input tends to report biased results. Thus, log parsing should be accurate enough to ensure the high performance of subsequent log mining tasks.

To evaluate the effectiveness of POP on log mining, we apply different log parsers to tackle the parsing challenge of a real-world anomaly detection task. This task employs Principal Component Analysis (PCA) to detect anomalies. Due to the space limit, the

technical details of this anomaly detection task is described in our supplementary report [20]. There are totally 16,838 anomalies in this task, which are found manually in [94]. We re-tune the parameters of the parsers for better $parsing\ accuracy$. LKE is not employed because it could not handle this large amount of data (10m+ lines) in reasonable time. Table 4.6 demonstrates the evaluation results. $Reported\ anomaly$ is the number of anomalies reported by log mining model (i.e., PCA) while adopting different log parsers in the log parsing step. $Detected\ anomaly$ means the number of true anomalies detected by PCA. $False\ alarm$ is the number of wrongly detected anomalies. $Ground\ truth$ is an anomaly detection task with exactly correct parsed results. Notice that even the ground truth could not detect all anomalies because of the boundary of the PCA anomaly detection model.

From Table 4.6, we observe that LogSig and IPLoM lead to nearly optimal results on the anomaly detection task. However, SLCT does not perform well in anomaly detection with its acceptable parsing accuracy (0.83). It reports 7,515 false alarms in anomaly detection, which introduces extensive human effort on inspection. Furthermore, the parsing accuracy of SLCT (0.83) and LogSig (0.87) is comparable, but the performance of anomaly detection using LogSig as parser is one order of magnitude better than that using SLCT. Anomaly detection task using LogSig only reports 413 false alarms. These reveal that anomaly detection results are sensitive to some critical events, which are generated by log parsers. It is also possible that F-measure, despite pervasively used in clustering algorithm evaluation, may not be suitable to evaluate the effectiveness of log parsing methods on log mining. The effectiveness of existing parsers on the anomaly detection task is summarized as follows.

Log parsing is important because log mining is effective only when the parsing result is accurate enough. Log mining is sensitive to some critical events. 4% *errors in parsing could even cause one*

order of magnitude performance degradation in anomaly detection.

The parameters of POP in this experiment are the same as those tuned for 2k HDFS datasets. We observe that the accurate parsed results of POP are effective from the perspective of this anomaly detection task. Although there are still 37% non-detected anomalies, we think this is the limitation of the anomaly detection model PCA. Because the anomaly detection task with ground truth as input provides comparable performance, where 34% anomalies are not detected. Note that although the performance of the anomaly detection task with POP as input is the same as that of IPLoM in Table 4.6, their parsing results are different.

4.3.5 Parameter Sensitivity

POP has four parameters: GS, splitRel, splitAbs, maxDistance. We will explain these parameters one by one. All the sensitivity experiments are run on the 2k datasets, which are the datasets used to evaluate the accuracy of the parsers in Section 4.3.2. Similar to the parameter setting in our accuracy experiments, for dataset BGL, HPC, HDFS, and Zookeeper, we set GS to 0.6, splitRel to 0.1, splitAbs to 10, maxDistance to 0; for Proxifier, we set GS to 0.3, splitRel to 0.1, splitAbs to 5, maxDistance to 10. To study the impact of different parameters, we evaluate the accuracy of POP while varying the value of the studied parameter.

Impact of GS

According to our definition, GS decides whether a log group is complete in step 3. If a log group is complete, it will be sent to step 4 without further partitioning. Intuitively, GS is the threshold about the percentage of columns where all the tokens are the same in a log group. For example, if we set GS to 0.6, POP will regard log groups with more than 60% columns having the same tokens respectively as complete groups. For the log group in Fig. 4.9, two (i.e., the

first and third) columns have the same tokens in that token position (i.e., "Send" and "file") respectively. Its GG (Group Goodness) is 2/3. GG (2/3) is larger than GS (0.6), so this log group is a complete group.

1.	Send	configuration	file
2.	Send	network	file
3.	Send	bootstrap	file
4.	Send	video	file

Figure 4.9: Example Log Group

The sensitivity analysis for GS is demonstrated by Fig. 4.10. We can observe that the accuracy of POP is high for all datasets if we set GS in range [0.5, 0.6]. Intuitively, this means a log group is complete if more than half or 60% of its columns have the same tokens for the whole column. When GS is smaller, a log group is easier to get sent to step 4 without further partitioning, which may lower the accuracy because we may put log messages with different log events into the same log group. Thus we can observe the relatively lower accuracy for range [0, 0.3] on HPC and range [0, 0.2] on HDFS. When GS is larger, a log group has higher probability to go through partitioning process in step 3, which may lower the accuracy because we may put log messages with the same log event into different log groups. Thus we can observe the relatively lower accuracy in range [0.8, 1.0] on HPC, range [0.8, 1.0] on HDFS and range [0.9, 1.0] on Zookeeper. To tune GS for a new dataset, we could first set GS to 0.5 or 0.6. Then we evaluate the accuracy of POP on a sample dataset (e.g., 2k lines), which is often much smaller than the whole dataset. If the accuracy is not satisfactory, we can increase or decrease the GS by 0.1, and evaluate POP's accuracy on the sample dataset again. We can repeat this process until the accuracy becomes lower. Then we select the GS that achieves the highest accuracy on the sample dataset. According to our parameter tuning experiment (Section

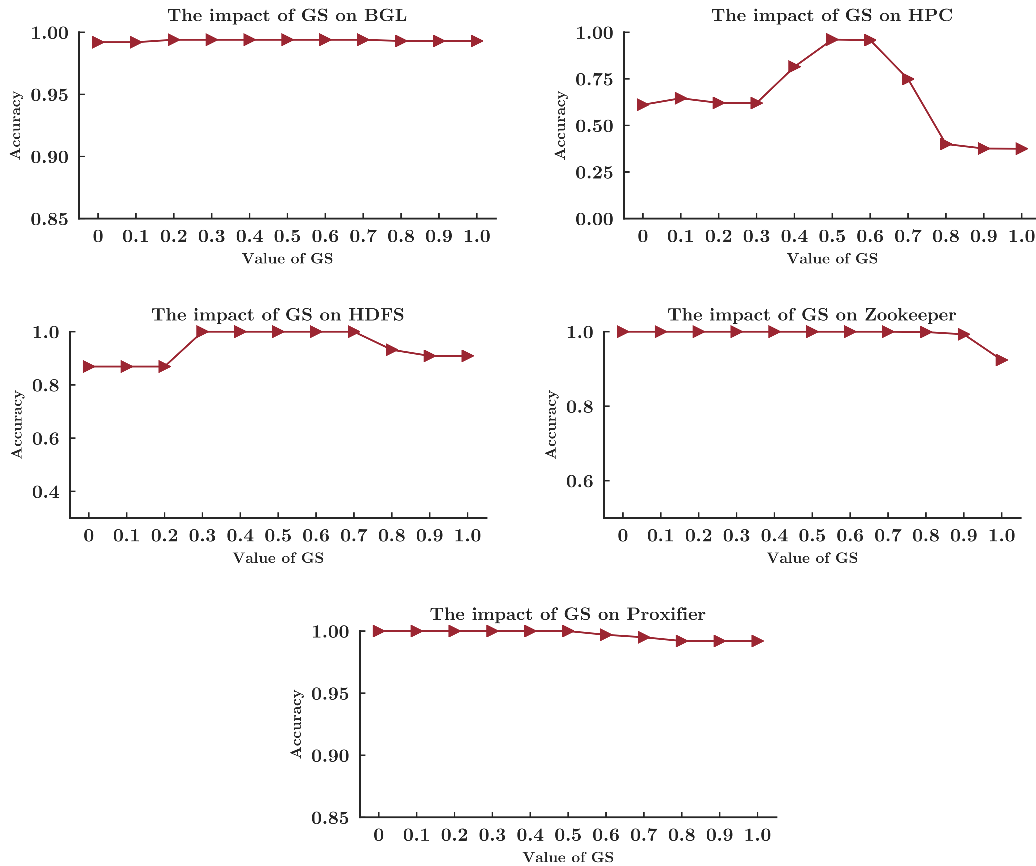


Figure 4.10: Impact of GS

4.2.2 in the revised manuscript), POP can achieve high accuracy when using the parameters tuned on small sampled data.

Impact of splitRel and splitAbs

Parameter splitRel and splitAbs are used to control the partitioning of the incomplete log groups in step 3 of POP. For each incomplete group, POP will try to split the log group based on the tokens in the split token position. If the tokens in the split token position are constants, the current group will be partitioned into several groups. Otherwise, the log group will be sent to step 4. splitAbs is the threshold about the number of unique tokens (i.e., AT) in a column (i.e., token position). For example, in Fig. 4.9, the first and third

column have one unique token, while the second column has four unique tokens. splitRel is the threshold about the ratio between the number of unique tokens and the number of tokens (i.e., RT) in a column. For example, the number of tokens for all the columns in Fig. 4.9 is four. If RT is larger than splitRel, and AT is larger than splitAbs, POP regards the tokens in the split token position as variables, because there are too many unique tokens. Otherwise, POP regards the tokens as constants.

We conduct the sensitivity analysis experiments for both splitRel and splitAbs. Since their results are very similar, we demonstrate the results for splitRel here.

From Fig. 4.11, we can observe that the accuracy of POP is insensitive to the value of splitRel. There are two main reasons. First, with a suitable (but easy to find) GS, we can already send some complete groups to step 4, which provides a lower bound for the accuracy. Second, splitRel and splitAbs control the partitioning of incomplete groups together in step 3. Thus with a suitable splitAbs, changing the value splitRel will not cause big accuracy change. We provide both splitRel and splitAbs to allow finer tuning by users. The accuracy of POP has observable change on BGL when varying the value of splitRel, and the accuracy peaks when splitRel is in range [0.1, 0.3]. This demonstrates the effectiveness of having both splitRel and splitAbs.

We also evaluate the impact of splitRel when splitAbs is 0 as in Fig. 4.12. We can observe that the accuracy of POP is consistent even we set splitAbs to 0. For BGL, HPC, HDFS, and Zookeeper, the accuracy is low when both splitRel and splitAbs is 0. Under this parameter setting, POP always regards the tokens in the split token position of the incomplete groups as variables. Thus, POP will send all the incomplete groups to step 4 without further partitioning. This lead to log groups containing log messages with different log events, which lowers the accuracy. For the parameter values in range [0.1, 1.0], POP consistently achieves high accuracy. To set a suitable

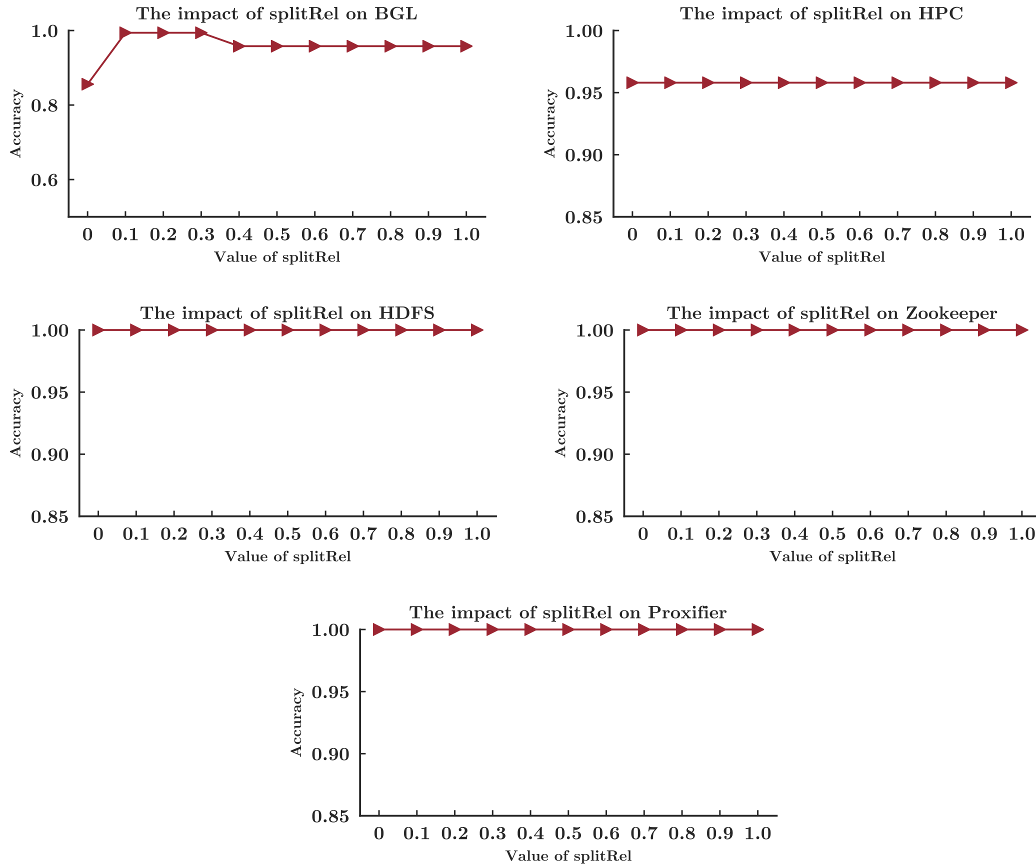


Figure 4.11: Impact of splitRel

splitRel (or splitAbs) value, we can first pick a reasonable value, for example 0.1 (or 10). Then we tune this value by evaluating the accuracy of POP on small sample datasets. The parameter value that has the highest accuracy can be used on the original dataset.

Impact of maxDistance

Parameter maxDistance is used in step 5 to merge the similar log groups based on the log events (i.e., log templates) extracted in step 4. POP calculates the Manhattan distance of log events in step 5 to merge log groups. Intuitively, maxDistance is the maximum number of different tokens allowed between the farthest two log events in two merging groups. In our paper, we set maxDistance

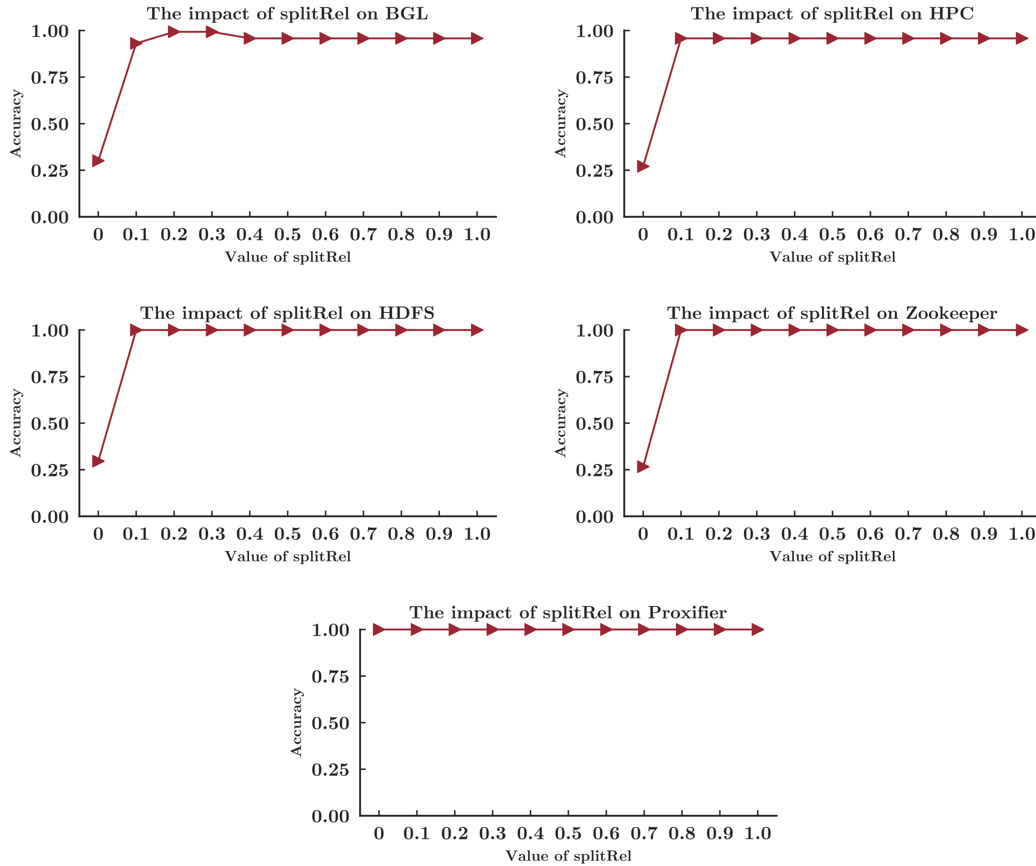


Figure 4.12: Impact of splitRel (splitAbs=0)

to 0 for BGL, HPC, HDFS, and Zookeeper, where only two groups with the exactly same log event will be merged. Because POP can already achieve high accuracy, we do not use a larger maxDistance value. For Proxifier, if we set maxDistance to 0, the accuracy is 0.89. Besides, some log groups are over-parsed by step 3 on Proxifier. Thus, we set maxDistance to 10 to address over-parsing. We demonstrate the impact of maxDistance on accuracy in Fig. 4.13.

From Fig. 4.13, we can observe that the accuracy peaks when maxDistance is in range [4, 12]. When maxDistance is in range [0, 4), logs have been over-parsed, so the accuracy of POP is relatively low. When maxDistance is in range (12, 20], or is larger than 20,

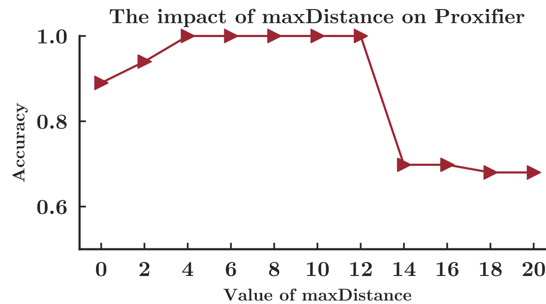


Figure 4.13: Impact of maxDistance

log groups that contain log messages with different log events are merged, which lowers the accuracy of POP. To find the suitable maxDistance, similar to other parameters, we can start with a value, for example, set maxDistance to 10. Then we tune this parameter by evaluating the accuracy of POP on a small sampled dataset. After finding the best parameter, we can directly apply it to the original dataset.

To pick a suitable value, we could first set the parameter to a reasonable value according to its physical meaning. Then we tune it on a small sample dataset by evaluating the resulting accuracy. After finding the best parameter, we can apply it to the original dataset. We add Section 4.5 in the revised manuscript to demonstrate the above parameter analysis results. The sensitivity analysis would greatly facilitate the parameter setting.

4.3.6 Observations

Among the existing log parsers, LKE has quadratic time complexity, while the running time of others scales linearly with the number of log messages. LogSig is accurate on most datasets. IPLoM is accurate and efficient on small datasets. SLCT requires the least running time. Although these widely used log parsing methods have their own merits, none of them can perform accurately and efficiently on various modern datasets. First, SLCT is not accurate

enough. Because of its relatively low parsing accuracy, in our case study in Section 4.3.4, the false alarm rate of the subsequent anomaly detection task increases to 40%, which causes 7,515 false positives. Secondly, LKE and LogSig cannot handle large-scale log data efficiently. Specifically, LKE has quadratic time complexity, while LogSig needs computation-intensive iterations. Moreover, LKE and LogSig both require non-trivial parameter tuning effort. Finally, IPLoM cannot efficiently handle large-scale log data (e.g., 200 million log messages) due to the limited computing power and memory of a single computer. Our proposed POP is the only log parser that performs accurately and efficiently on all the datasets.

4.4 Discussions

Training Log Data. Usually we hope to train our log parser on as many logs as possible. This can increase the generalizability of the results obtained by POP. This is also why we propose a parallel log parsing method that aims for parsing large-scale logs. However, we agree that in case we have too many historical logs for processing, sampling is an effective way. We suggest two methods to sample training data. (1) Using the latest logs. This sampling method is more likely to get the newest log events produced by new-version systems. (2) Collecting the logs periodically (e.g., collecting the logs every single day). This sampling method can allow the variability of logs. The quantity of sample logs depends on the training time we can afford. For example, in case of POP, if we want to finish the training process in 7 minutes for HDFS logs, then we can use the latest 200 million log messages.

Log Event Changes. Logs change over time, a log message may not be matched by the current list of log events. To solve this problem, developers can use POP to periodically retrain on new training data to update the list. In runtime, if a log message is not matched by any log events, we mark it as "other events" and record

it. When retraining, the developer can retrain on the log messages marked as "other events", and add the new log events to the log event list. To avoid the burst of not-matched logs (e.g., a billion times), we can maintain a counter to remember the number of log messages marked as "other events" after the latest training. If it is larger than a threshold, an alarm is reported to call for retraining.

POP for Big Data. We propose the parallel log parser POP in the manuscript because the existing nonparallel log parsers and SinglePOP cannot handle the large volume of logs generated by modern systems in the big data era. We can observe from the Fig. 4.7 that the increasing speed of SinglePOP's running time (i.e., slope) is faster than POP as the log size becomes larger. The running time of SinglePOP will be longer than that of POP on production level log data (e.g., over 200m log messages). For example, the running time of SinglePOP is already larger than that of POP on the 10m HDFS dataset as illustrated. Thus, although SinglePOP is efficient, we need POP, a parallel designed on top of Spark, to handle production level log data efficiently.

4.5 Summary

This chapter targets automated log parsing for the large-scale log analysis of modern systems. Based on the result of the evaluation study in Chapter 3, we propose a parallel log parsing method (POP). POP employs specially designed heuristic rules and hierarchical clustering algorithm. It is optimized on top of Spark by using tailored functions for selected Spark operations. Extensive experiments are conducted on both synthetic and real-world datasets, and the results reveal that POP can perform accurately and efficiently on large-scale log data. POP has been publicly released to make it reusable and thus facilitate future research.

□ **End of chapter.**

Chapter 5

Online Log Parsing via Fixed Depth Tree

Although offline log parsers can fulfill the need of many systems by periodical parser running, some systems expect more accurate and timely parsing rules mining. To address this problem, this chapter presents an online log parser, namely Drain. Different from the offline parsers, Drain can parse incoming logs in a streaming manner and dynamically updated its parsing rules. The main points of this chapter are as follows. (1) It presents the design of an online log parser Drain. (2) It explains the search phase and update phase based on fixed depth tree. (3) It implements and open-source releases Drain.

5.1 Introduction

The prevalence of cloud computing, which enables on-demand service delivery, has made Service-oriented Architecture (SOA) a dominant architectural style. Nowadays, more and more developers leverage existing Web services to build their own systems because of their rich functionality and "plug-and-play" property. Although developing Web service based system is convenient and lightweight, Web service management is a significant challenge for both service providers and users. Specifically, service providers (e.g., Amazon

EC2 [1]) are expected to provide services with no failures or SLA (service-level agreement) violations to a large number of users. Similarly, service users need to effectively and efficiently manage the adopted services, which have been discussed in many recent works (e.g., Web service monitoring [30]). In this context, log analysis based service management techniques, which employ service logs to achieve automatic or semi-automatic service management, have been widely studied.

Logs are usually the only data resource available that records service runtime information. In general, a log message is a line of text printed by logging statements (e.g., *printf(), logging.info()*) written by developers. Thus, log analysis techniques, which apply data mining models to get insights of system behaviors, are in widespread use for service management. For service providers, there are studies in anomaly detection [94, 35], fault diagnosis [91, 108] and performance improvement [84, 106]. For service users, typical examples include business model mining [27, 71] and user behavior analysis [97, 76].

Most of the data mining models used in these log analysis techniques require structured input (e.g., an event list or a matrix). However, raw log messages are usually unstructured, because developers are allowed to write free-text log messages in source code. Thus, the first step of log analysis is log parsing, where unstructured log messages are transformed into structured events. An unstructured log message, as in the following example, usually contains various forms of system runtime information: *timestamp* (records the occurring time of an event), *verbosity level* (indicate the severity level of an event, e.g., INFO), and *raw message content* (free-text description of a service operation).

```
081109 204655 556 INFO dfs.DataNode$PacketResponder
: Received block blk_3587508140051953248 of size 67
108864 from /10.251.42.84
```

Traditionally, log parsing relies heavily on regular expressions

[54], which are designed and maintained manually by developers. However, this manual method is not suitable for logs generated by modern services for the following three reasons. First, the volume of logs is increasing rapidly, which makes the manual method prohibitive. For example, a large-scale service system can generate 50 GB logs (120∼200 million lines) per hour [68]. Second, as open-source platforms (e.g., Github) and Web service become popular, a system often consists of components written by hundreds of developers globally [94]. Thus, people in charge of the regular expressions may not know the original logging purpose, which makes manual management even harder. Third, logging statements in modern systems updates frequently (e.g., hundreds of new logging statements every month [93]). In order to maintain a correct regular expression set, developers need to check all logging statements regularly, which is tedious and error-prone.

Log parsing is widely studied to parse the raw log messages automatically. Most of existing log parsers focus on offline, batch processing. For example, Xu et al. [94] design a method to automatically generate regular expressions based on source code. However, source code is often inaccessible in practice (e.g., Web service components). For general log parsing, recent studies propose data-driven methods [35, 64], which directly extract log templates from raw log messages. These log parsers are offline, and limited by the memory of a single computer. Besides, they fail to align with the log collecting manner. A typical log collection system has a log shipper installed on each node to forward log entries in a streaming manner to a centralized server that contains a log parser [31]. The offline log parsers need to employ all logs after log collection for a certain period (e.g., 1h) for the parser training. In contrast, an online log parser parses logs in a streaming manner, and it does not require an offline training step. Thus, current systems highly demand online log parsing, which is only studied in a few preliminary works [69, 31]. However, we observe that the parsers

proposed in these works are not accurate and efficient enough, which make them not eligible for log parsing in modern Web service or Web service based systems.

In this paper, we propose an online log parsing method, namely Drain, that can accurately and efficiently parse raw log messages in a streaming manner. Drain does not require source code or any information other than raw log messages. Drain can automatically extract log templates from raw log messages and split them into disjoint log groups. It employs a parse tree with fixed depth to guide the log group search process, which effectively avoids constructing a very deep and unbalanced tree. Besides, specially designed parsing rules are compactly encoded in the parse tree nodes. We evaluate Drain on five real-world log data sets with more than 10 million raw log messages. Drain demonstrates the highest accuracy on four data sets, and comparable accuracy on the remaining one. Besides, Drain obtains 51.8%$\sim$81.47% improvement in running time compared with the state-of-the-art online parser [31]. We also demonstrate the effectiveness of Drain in log analysis by tackling a real-world anomaly detection task [94].

In summary, our paper makes the following contributions:

- This paper presents the design of an online log parsing method (Drain), which encodes specially designed parsing rules in a parse tree with fixed depth.
- Extensive experiments have been conducted on five real-world log data sets, which determine the superiority of Drain in terms of accuracy and efficiency.
- The source code of Drain has been publicly released [3], allowing for easy use by researchers and practitioners for future study.

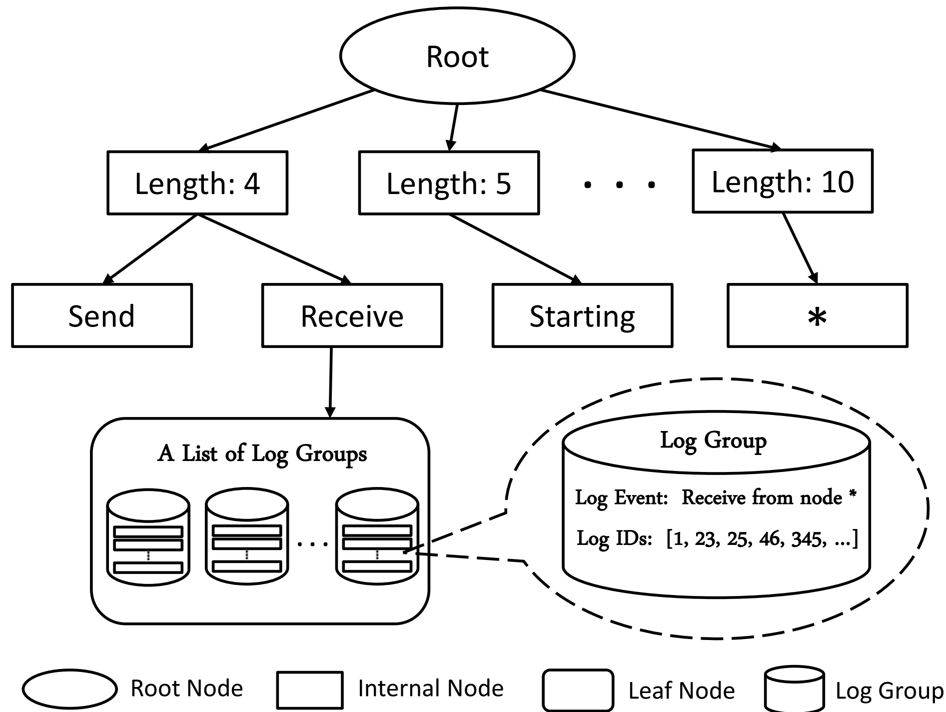


Figure 5.1: Structure of Parse Tree in Drain ($depth$ = 3)

5.2 Methodology

In this section, we briefly introduce Drain, a fixed **d**epth t**r**ee b**a**sed onl**i**ne log parsi**n**g method. When a new raw log message arrives, Drain will preprocess it by simple regular expressions based on domain knowledge. Then we search a log group (i.e., leaf node of the tree) by following the specially-designed rules encoded in the internal nodes of the tree. If a suitable log group is found, the log message will be matched with the log event stored in that log group. Otherwise, a new log group will be created based on the log message. In the following, we first introduce the structure of the fixed depth tree (i.e., parse tree). Then we explain how Drain parses raw log messages by searching the nodes of the parse tree.

5.2.1 Overall Tree Structure

When a raw log message arrives, an online log parser needs to search the most suitable log group for it, or create a new log group. In this process, a simple solution is to compare the raw log message with log event stored in each log group one by one. However, this solution is very slow because the number of log groups increases rapidly in parsing. To accelerate this process, we design a parse tree with fixed depth to guide the log group search, which effectively bounds the number of log groups that a raw log message needs to compare with.

The parse tree is illustrated in Figure 5.1. The $root\ node$ is in the top layer of the parse tree; the bottom layer contains the $leaf\ node$s; other nodes in the tree are $internal\ node$s. Root node and internal nodes encode specially-designed rules to guide the search process. They do not contain any log groups. Each path in the parse tree ends with a leaf node, which stores a list of log groups, and we only plot one leaf node here for simplicity. Each log group has two parts: log event and log IDs. Log event is the template that best describes the log messages in this group, which consists of the constant part of a log message. Log IDs records the IDs of log messages in this group. One special design of the parse tree is that the depth of all leaf nodes are the same and are fixed by a predefined parameter $depth$. For example, the depth of the leaf nodes in Figure 5.1 is fixed to 3. This parameter bounds the number of nodes Drain visits during the search process, which greatly improves its efficiency. Besides, to avoid tree branch explosion, we employ a parameter $maxChild$, which restricts the maximum number of children of a node. In the following, for clarity, we define an n-th layer node as a node whose depth is n. Besides, unless otherwise stated, we use the parse tree in Figure 5.1 as an example in following explanation. In practice, $depth$ can be tuned on a sample dataset of small size. Usually, $depth$ will be 3 or 4 according to our experience.

5.2.2 Step 1: Preprocess by Domain Knowledge

According to our previous empirical study on existing log parsing methods [39], preprocessing can improve parsing accuracy. Thus, before employing the parse tree, we preprocess the raw log message when it arrives. Specifically, Drain allows users to provide simple regular expressions based on domain knowledge that represent commonly-used variables, such as IP address and block ID. Then Drain will remove the tokens matched from the raw log message by these regular expressions.

The regular expressions employed in this step are often very simple, because they are used to match tokens instead of log messages. Besides, a data set usually requires only a few such regular expressions. For example, the data sets used in our evaluation section require at most two such regular expressions.

5.2.3 Step 2: Search by Log Message Length

In this step and step 3, we explain how we traverse the parse tree according to the encoded rules and finally find a leaf node.

Drain starts from the root node of the parse tree with the preprocessed log message. The 1-st layer nodes in the parse tree represent log groups whose log messages are of different log message lengths. By log message length, we mean the number of tokens in a log message. In this step, Drain selects a path to a 1-st layer node based on the log message length of the preprocessed log message. For example, for log message “Receive from node 4”, Drain traverse to the internal node “Length: 4” in Figure 5.1. This is based on the assumption that log messages with the same log event will probably have the same log message length. Although it is possible that log messages with the same log event have different log message lengths, it can be handled by simple postprocessing. Besides, our experiments in Section 5.3.2 demonstrate the superiority of Drain in terms of parsing accuracy even without postprocessing.

5.2.4 Step 3: Search by Preceding Tokens

In this step, Drain traverses from a 1-st layer node, which is searched in step 2, to a leaf node. This step is based on the assumption that tokens in the beginning positions of a log message are more likely to be constants. Specifically, Drain selects the next internal node by the tokens in the beginning positions of the log message. For example, for log message "Receive from node 4", Drain traverses from 1-st layer node "Length: 4" to 2-nd layer node "Receive" because the token in the first position of the log message is "Receive". Then Drain will traverse to the leaf node linked with internal node "Receive", and go to step 4.

The number of internal nodes that Drain traverses in this step is $(depth - 2)$, where $depth$ is the parse tree parameter restricting the depth of all leaf nodes. Thus, there are $(depth - 2)$ layers that encode the first $(depth - 2)$ tokens in the log messages as search rules. In the example above, we use the parse tree in Figure 5.1 for simplicity, whose depth is 3, so we search by only the token in the first position. In practice, Drain can consider more preceding tokens with larger depth settings. Note that if $depth$ is 2, Drain only considers the first layer used by step 2.

In some cases, a log message may start with a parameter, for example, "120 bytes received". These kinds of log messages can lead to branch explosion in the parse tree because each parameter (e.g., 120) will be encoded in an internal node. To avoid branch explosion, we only consider tokens that do not contain digits in this step. If a token contains digits, it will match a special internal node "*". For example, for the log message above, Drain will traverse to the internal node "*" instead of "120". Besides, we also define a parameter $maxChild$, which restricts the maximum number of children of a node. If a node already has $maxChild$ children, any non-matched tokens will match the special internal node "*" among all its children.

5.2.5 Step 4: Search by Token Similarity

Before this step, Drain has traversed to a leaf node, which contains a list of log groups. The log messages in these log groups comply with the rules encoded in the internal nodes along the path. For example, the log group in Figure 5.1 has log event "Receive from node *", where the log messages contain 4 tokens and start with token "Receive".

In this step, Drain selects the most suitable log group from the log group list. We calculate the similarity $simSeq$ between the log message and the log event of each log group. $simSeq$ is defined as following:

$$simSeq = \frac{\sum_{i=1}^{n} equ(seq_1(i), seq_2(i))}{n}, \tag{5.1}$$

where seq_1 and seq_2 represent the log message and the log event respectively; $seq(i)$ is the i-th token of the sequence; n is the log message length of the sequences; function equ is defined as following:

$$equ(t_1, t_2) = \begin{cases} 1 & \text{if } t_1 = t_2 \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

where t_1 and t_2 are two tokens. After finding the log group with the largest $simSeq$, we compare it with a predefined similarity threshold st. If $simSeq \geq st$, Drain returns the group as the most suitable log group. Otherwise, Drain returns a flag (e.g., None in Python) to indicate no suitable log group.

5.2.6 Step 5: Update the Parse Tree

If a suitable log group is returned in step 4, Drain will add the log ID of the current log message to the log IDs in the returned log group. Besides, the log event in the returned log group will be updated. Specifically, Drain scans the tokens in the same position of the log

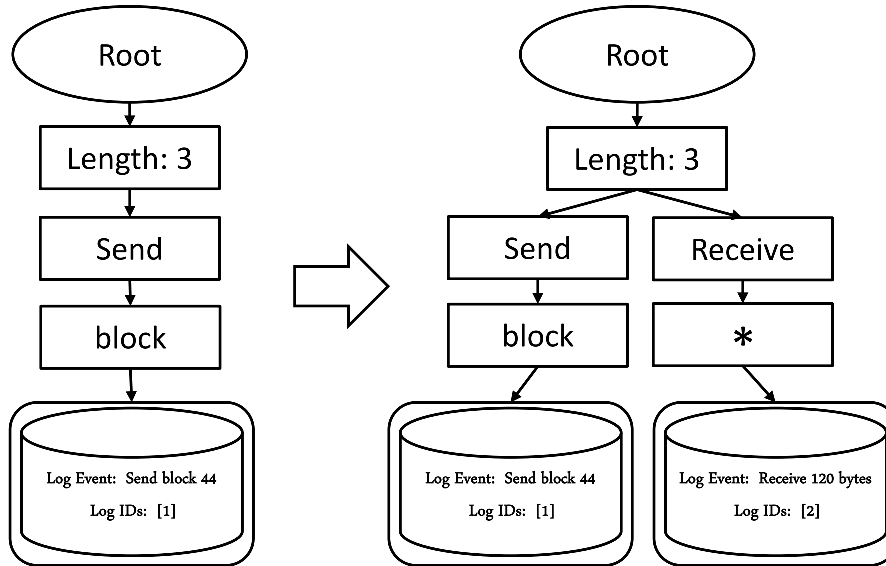


Figure 5.2: Parse Tree Update Example ($depth$ = 4)

message and the log event. If the two tokens are the same, we do not modify the token in that token position. Otherwise, we update the token in that token position by wildcard (i.e., *) in the log event.

If Drain cannot find a suitable log group, it creates a new log group based on the current log message, where log IDs contains only the ID of the log message and log event is exactly the log message. Then, Drain will update the parse tree with the new log group. Intuitively, Drain traverses from the root node to a leaf node that should contain the new log group, and adds the missing internal nodes and leaf node accordingly along the path. For example, assume the current parse tree is the tree in the left-hand side of Figure 5.2, and a new log message "Receive 120 bytes" arrives. Then Drain will update the parse tree to the right-hand side tree in Figure 5.2. Note that the new internal node in the 3-rd layer is encoded as "*" because the token "120" contains digits.

5.3 Evaluation

5.3.1 Experimental Settings

Log Data Sets

The log data sets used in our evaluation are summarized in Table 5.1. These five real-world data sets range from supercomputer logs (BGL and HPC) to distributed system logs (HDFS and Zookeeper) to standalone software logs (Proxifier). Companies rarely release their log data to the public, because it may violates confidential clauses. We obtained three log data sets from other researchers with their generous support. Specifically, BGL is a log data set collected by Lawrence Livermore National Labs (LLNL) from BlueGene/L supercomputer system [73]. HPC logs are collected from a high performance cluster, which has 49 nodes with 6,152 cores and 128GB memory per node [57]. HDFS is a log data set collected from a 203-node cluster on Amazon EC2 platform in [94]. We also collect two log data sets for evaluation. One is collected from Zookeeper installed on a 32-node cluster in our lab. The other are logs of a standalone software Proxifier.

Table 5.1: Summary of Log Data Sets

System	Description	#Log Messages	Log Message Length	#Events
BGL	BlueGene/L Supercomputer	4,747,963	10~102	376
HPC	High Performance Cluster (Los Alamos)	433,490	6~104	105
HDFS	Hadoop File System	11,175,629	8~29	29
Zookeeper	Distributed System Coordinator	74,380	8~27	80
Proxifier	Proxy Client	10,108	10~27	8

Comparison

To prove the effectiveness of Drain, we compare its performance with four existing log parsing methods in terms of accuracy, efficiency and effectiveness on subsequent log mining tasks. Specifically, two of them are offline log parsers, and the other two are online log parsers. The ideas of these log parsers are briefly introduced as following:

- LKE [35]: This is an offline log parsing method developed by Microsoft. It employs hierarchical clustering and heuristic rules.
- IPLoM [64]: IPLoM conducts a three-step hierarchical partitioning before template generation in an offline manner.
- SHISO [69]: In this online parser, a tree with predefined number of children in each node is used to guide log group searching.
- Spell [31]: This method uses longest common sequence to search log group in an online manner. It accelerates the searching process by subsequence matching and prefix tree.

Evaluation Metric and Experimental Setup

We use F-measure [66, 9], which is a typical evaluation metric for clustering algorithms, to evaluate the accuracy of log parsing methods. The definition of accuracy is as the following.

$$Accuracy = \frac{2 * Precision * Recall}{Precision + Recall}, \tag{5.3}$$

where $Precision$ and $Recall$ are defined as follows:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}, \tag{5.4}$$

Table 5.2: Parameter Setting of Drain

	BGL	HPC	HDFS	Zookeeper	Proxifier
depth	3	4	3	3	4
st	0.3	0.4	0.5	0.3	0.3

where a true positive (TP) decision assigns two log messages with the same log event to the same log group; a false positive (FP) decision assigns two log messages with different log events to the same log group; and a false negative (FN) decision assigns two log messages with the same log event to different log groups. This evaluation metric is also used in our previous study [39] on existing log parsers.

We run all experiments on a Linux server with Intel Xeon E5-2670v2 CPU and 128GB DDR3 1600 RAM, running 64-bit Ubuntu 14.04.2 with Linux kernel 3.16.0. We run each experiment 10 times to avoid bias. For the preprocessing step of Drain (step 1), we remove obvious parameters in log messages (i.e., IP addresses in HPC&Zookeeper&HDFS, core IDs in BGL, and block IDs in HDFS). The parameter setting of Drain is shown in Table 5.2. Besides, we empirically set $maxChild$ to 100 for all experiments. The number of children of a tree node rarely exceeds $maxChild$, because the encoded rules in the parse tree can already distribute the logs evenly to different paths. We also re-tune the parameters of other log parsers to optimize their performance, which is not presented here because of the space limit. We put them in our released source code [14] for further reference.

5.3.2 Accuracy of Drain

Accuracy demonstrates how well a log parser matches raw log messages with the correct log events. Accuracy is important because parsing errors can degrade the performance of subsequent log mining task. Intuitively, an offline log parsing method could obtain higher accuracy compared with an online one, because an offline

Table 5.3: Parsing Accuracy of Log Parsing Methods

	BGL	HPC	HDFS	Zookeeper	Proxifier
Offline Log Parsers					
LKE	0.67	0.17	0.57	0.78	0.85
IPLoM	0.99	0.65	0.99	0.99	0.85
Online Log Parsers					
SHISO	0.87	0.53	0.93	0.68	0.85
Spell	0.98	0.82	0.87	0.99	**0.87**
Drain	**0.99**	**0.84**	**0.99**	**0.99**	0.84

method enjoys all raw log messages at the beginning of parsing, while an online method adjusts its parsing model gradually in the parsing process.

In this section, we evaluate the accuracy of two offline and two online log parsing methods on the data sets described in Table 5.1. The evaluation results are in Table 5.3. LKE fails to handle the data sets except Proxifier, because its $O(n^2)$ time complexity makes it too slow for the other data sets. Thus, for the other four data sets, as with the existing work [39, 85], we evaluate LKE's accuracy on sample data sets with 2k log messages randomly extracted from the original ones, while all parsers are evaluated on the 2k sample data sets in our previous paper [39].

We observe that the proposed online parsing method, namely Drain, obtains the best accuracy on four data sets, even compared with the offline log parsing methods. For data set Proxifier, Drain also has comparable accuracy (i.e., 0.84) to Spell, which obtains the highest accuracy (i.e., 0.87) on this data set. LKE is not that good on some data sets, because it employs an aggressive clustering strategy, which can lead to under-partitioning. IPLoM obtains high accuracy on most data sets because of its specially-designed heuristic rules. SHISO uses the similarity of characters in log messages to search the corresponding log events. This strategy is too coarse-grained, which causes inaccuracy. Spell is accurate, but its strategy only based on longest common subsequence can lead to under-partitioning. Drain has the overall best accuracy for three reasons. First, it

Table 5.4: Running Time (Sec) of Log Parsing Methods

	BGL	HPC	HDFS	Zookeeper	Proxifier
Offline Log Parsers					
LKE	N/A	N/A	N/A	N/A	8888.49
IPLoM	140.57	12.74	333.03	2.17	0.38
Online Log Parsers					
SHISO	10964.55	582.14	6649.23	87.61	8.41
Spell	447.14	47.28	676.45	5.27	0.87
Drain	115.96	8.76	325.7	1.81	0.27
Improvement	**74.07%**	**81.47%**	**51.85%**	**65.65%**	**68.97%**

compounds both the log message length and the first few tokens, which are effective and specially-designed rules, to construct the fixed depth tree. Second, Drain only uses tokens that do not contain digits to guide the searching process, which effectively avoids over-partitioning. Third, the tunable tree depth and similar threshold st allows users to conduct fine-grained tuning on different data sets.

5.3.3 Efficiency of Drain

To evaluate the efficiency of Drain, we measure the running time of it and four existing log parsers on five real-world log data sets described in Table 5.1. In Table 5.4, we demonstrate the running time of these log parsers. LKE fails to handle four data sets in reasonable time (i.e., days or weeks), so we mark the corresponding results as not available.

Considering online parsing methods, SHISO takes too much time on some data sets (e.g., takes more than 3h on BGL). This is mainly because SHISO only limits the number of children for its tree nodes, which can cause very deep parse tree. Spell obtains better efficiency performance, because it employs a prefix tree structure to store all log events found, which greatly reduces its running time. However, Spell does not restrict the depth of its prefix tree either, and it calculates the longest common subsequence between two log messages, which is time consuming. Compared with the existing online parsing methods, our proposed Drain requires the

least running time on all five data sets. Specifically, Drain only needs 2 min to parse 4m BGL log messages and 6 min to parse 10m HDFS log messages. Drain greatly improves the running time of existing online parsing methods. The improvements on the five real-world data sets are at least 51.85%, and it reduce 81.47% running time on HPC. Drain also outperforms the existing offline log parsing methods. It requires less running time than IPLoM on all five data sets. Moreover, as an online log parsing method, Drain is not limited by the memory of a single computer, which is the bottleneck of most offline log parsing methods. For example, IPLoM needs to load all log messages into computer memory, and it will construct extra data structures of comparable size in runtime. Thus, although IPLoM is efficient too, it may fail to handle large-scale log data. Drain is not limited by the memory of single computer, because it processes the log messages one by one.

Table 5.5: Log Size of Sample Datasets for Efficiency Experiments

BGL	400	4k	40k	400k	4m
HPC	600	3k	15k	75k	375k
HDFS	1k	10k	100k	1m	10m
Zookeeper	4k	8k	16k	32k	64k
Proxifier	600	1200	2400	4800	9600

Because log size of modern systems is rapidly increasing, a log parsing method is expected to handle large-scale log data. Thus, to simulate the increasing of log size, we also measure the running time of these log parsers on 25 sampled log data sets with varying log size (i.e., number of log messages) as described in Table 5.5. The log messages in these sampled data sets are randomly extracted from the real-world data sets in Table 5.1.

The evaluation results are illustrated in Figure 5.3, which is in logarithmic scale. In this figure, we observe that, compared with other methods, the running time of LKE raises faster as the log size increases. Because the time complexity of LKE is $O(n^2)$, and the time complexity of other methods is $O(n)$, while n is the number

of log messages. IPLoM is comparable to Drain, but it requires substantial amounts of memory as explained above. Online parsing methods (i.e., SHISO, Spell, Drain) process log message one by one, and they all use a parse tree to accelerate the log event search process. Drain is faster than others because of two main reasons. First, Drain enjoys linear time complexity. The time complexity of Drain is $O(\ (d + cm)n\ )$, where d is the depth of the parse tree, c is the number of candidate log groups in the leaf node, m is the log message length, and n is the number of log messages. Obviously, d and m are constants. c can also be regarded as a constant, because the quantity of candidate log groups in each leaf node is nearly the same, and the number of log groups is far less than that of log messages. Thus, the time complexity of Drain is $O(n)$. For SHISO and Spell, the depth of the parse tree could increase during the parsing process. Second, we use the specially-designed $simSeq$ to calculate the similarity between a log message and a log event candidate. Its time complexity is $O(m_1 + m_2)$, while m_1 and m_2 are number of tokens in them respectively. In Drain, $m_1 = m_2$. By comparison, SHISO and Spell calculate the longest common subsequence between two sequences, whose time complexity is $O(m_1 m_2)$.

5.3.4 Effectiveness of Drain on Real-World Anomaly Detection Task

In previous sections, we demonstrate the superiority of Drain in terms of accuracy and efficiency. Although high accuracy is necessary for log parsing methods, it does not guarantee good performance in the subsequent log mining task. For example, because log mining could be sensitive to some critical events, little parsing error may cause an order of magnitude performance degradation in log mining [39]. To evaluate the effectiveness of Drain on subsequent log mining tasks, we conduct a case study on a real-world anomaly

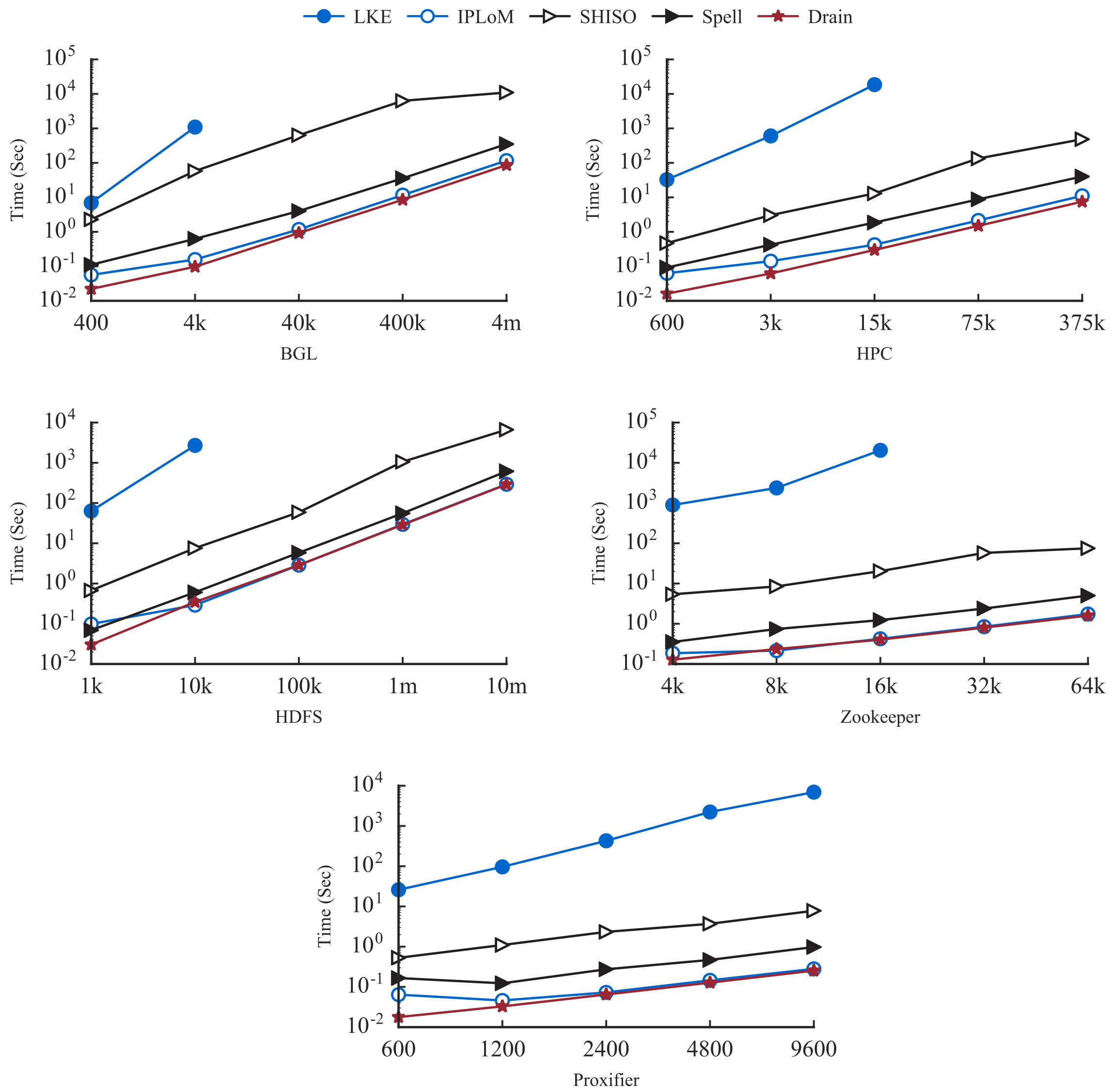


Figure 5.3: Running Time of Log Parsing Methods on Data Sets in Different Size

detection task.

We use the HDFS log data set in this case study. Specifically, raw log messages in the HDFS data set [94] records system operations on 575,061 HDFS blocks with a total of 29 log event types. Among these blocks, 16,838 are manually labeled as anomalies by the original authors. In the original paper [94], the authors employ Principal Component Analysis (PCA) to detect these anomalies. Next, we will briefly introduce the anomaly detection workflow, including log parsing and log mining. In log parsing step, all the raw log messages are parsed into structured log messages. Each structured log message contains the corresponding HDFS block ID and a log event. A source code-based log parsing method is used in the original paper, which is not discussed here because source code is inaccessible in many cases (e.g., in third party libraries). In log mining, we first use the structured log messages to generate an event count matrix, where each row represents an HDFS block; each column represents a log event type; each cell counts the occurrence of an event on a certain HDFS block. Then we use TF-IDF [80] to preprocess the event count matrix. Intuitively, TF-IDF gives lower weights to common event types, which are less likely to contribute to the anomaly detection process. Finally, the event count matrix is fed into PCA, which automatically marks the blocks as normal or abnormal.

In our case study, we evaluate the performance of the anomaly detection task with different log parsing methods used in the parsing step. Specifically, we use different log parsing methods to parse the HDFS raw log messages respectively and, hence, we obtain different sets of structured log messages. For example, an HDFS block ID could match with different log events by using different log parsing methods. Then, we generate different event count matrices, and fed them into PCA, respectively.

The experimental results are shown in Table 5.6. In this table, $reported\ anomaly$ is the number of anomalies reported by the PCA

s

Table 5.6: Anomaly Detection with Different Log Parsing Methods (16,838 True Anomalies)

	Parsing Accuracy	Reported Anomaly	Detected Anomaly	False Alarm
IPLoM	0.99	10,998	10,720 (63%)	278 (2.5%)
SHISO	0.93	13,050	11,143 (66%)	1,907 (14.6%)
Spell	0.87	10,949	10,674 (63%)	275 (2.5%)
Drain	0.99	10,998	10,720 (63%)	278 (2.5%)
Ground truth	1.00	11,473	11,195 (66%)	278 (2.4%)

model; $detected\ anomaly$ is the number of true anomalies reported; $false\ alarm$ is the number of wrongly reported ones. We use four existing log parsing methods to handle the parsing step of this anomaly detection task. We do not use LKE because it cannot handle this large amount of data. $Ground\ truth$ is the experiment using exactly correct parsed results.

We can observe that Drain obtains nearly the optimal anomaly detection performance. It detects $10,720$ true anomalies with only 278 false alarms. Although 37% of anomalies have not been detected, it is caused by the log mining step. Because even when all the log messages are correctly parsed, the log mining model still leaves 34% of anomalies at large. Note that although IPLoM demonstrates the same anomaly detection performance as Drain, their parsing results are different. We also observe that SHISO, although has a high parsing accuracy (0.93), does not perform well in this anomaly detection task. By using SHISO, we would report $1,907$ false alarms, which are 6 times worse than others. This will largely increase the workload of developers, because they usually need to manually check the anomalies reported. Among the online parsing methods, Drain not only has the highest parsing accuracy as demonstrated in Section 5.3.2, but also obtains nearly optimal performance in the anomaly detection case study.

5.4 Summary

This paper proposes an online log parsing method, namely Drain, that parses raw log messages in a streaming manner. Drain adopts a fixed depth parse tree to accelerate the log group search process, which encodes specially designed rules in its tree nodes. To evaluate the effectiveness of Drain, we conduct experiments on five real-world log data sets. The experimental results show that Drain greatly outperforms existing online log parsers in terms of accuracy and efficiency. Drain even obtains better performance than the state-of-the-art offline log parsers, which are limited by the memory of a single computer. Besides, we conduct a case study on a real-world anomaly detection task, which demonstrates the effectiveness of Drain on log analysis tasks.

□ **End of chapter.**

Chapter 6

Prioritizing Operational Issues via Hierarchical Log Clustering

It is error-prone and inefficient for system operators to manually handle a large number of operational issues every day. To alleviate the burden of operators and improve the efficiency of issues handling, we design an issues prioritization framework based on hierarchical log clustering. The key point is that similar operational issues have similar accompanied log sequences. The main points of this chapter are as follows. (1) It designs an operational issues prioritization framework, namely POI, based on hierarchical log clustering. (2) It proposes a novel weighting method Inverse Cardinality (IC). (3) It conducts experiments on a real-world dataset to demonstrate the effectiveness of POI.

6.1 Introduction

Modern systems are becoming increasingly large-scale and complex, especially distributed systems. Different from traditional systems, distributed systems inevitably encounter failures (e.g., node failure). However, distributed systems often run on a 24×7 basis to serve millions of users globally, which highlights the importance of reliability assurance. To enhance system reliability, one important task for developers (or operators) is to handle the user-reported

operational issues efficiently.

An operational issue is a system problem reported by a user. For example, when a user of Amazon EC2 [1] finds that her node becomes extremely slow, she will report the node slowness as an issue to Amazon. Typically, a reported issue contains user information, system configuration, a log sequence at runtime, etc. To handle an operational issue, developers mainly inspect the runtime log sequence to understand the system events that cause the issue. Specifically, a log sequence is a list of log messages that records various system events. Based on the log sequence, developers will identify the system problem, and finally address the problem.

However, as the distributed systems become increasingly large-scale and complex, the volume of the accompanied logs grows rapidly as well; for example, it generates at a rate of approximately 50 GB/h (120$\sim$200 million lines) [68]. Thus, traditional operational issue handling methods based on manual inspection become prohibitive and error-prone. Moreover, not all log messages are useful, because log messages are printed by logging statements written in development time. Therefore, most of the log messages record normal system runtime status, which increases the difficulty of the manual method. This problem is compounded by the fact that the number of issues reported in modern distributed systems is large. Operators may need to handle tons of operational issues every day. Last but not least, because of the complexity of distributed systems, operational issues are highly diverse. Intuitively, operators need to handle the issues that affect more users first, which is hard to achieve using traditional methods. Thus, an automated operational issue handling method is highly in demand.

To address this problem, in this chapter, we propose an automated operational issue prioritization framework, namely POI. POI clusters all the issues into different issue groups, and prioritize the issue groups based on the number of issues inside. Thus, compared with traditional methods, the superiority of POI is two-fold. First,

with POI, developers only need to inspect several issue groups instead of tons of issues. Second, developers can focus on the issue groups with high priority, which guides them to resolve the problems affecting many users first. This is useful for distributed systems, which serve millions of users and need to avoid non-trivial downtime.

We notice that, although the number of operational issues is large, many of them are similar or even redundant issues, where similar issues often share similar log sequences. Thus, it is possible to cluster the similar issues into a group based on the accompanied log sequences. Specifically, POI contains a coarse-grained clustering step and a fine-grained clustering step. Besides, to calibrate the weights of different log events, we design a weighting method Inverse Cardinality (IC), which is employed in both clustering steps of POI. We evaluate POI on a real-world dataset collected on Hadoop Distributed File System (HDFS) [24], which contains 11,175,629 log messages and 16,838 issues. The experimental results show that POI achieves the highest F-measure [66, 9] and the best issue coverage. Thus, POI demonstrates its superiority to improve the efficiency of issue handling via operational issue prioritization.

In summary, this chapter makes the following contributions:

- We design an operational issue handling framework, namely POI, based on hierarchical log clustering, including a coarse-grained clustering and a fine-grained clustering.
- We propose a novel weighting method Inverse Cardinality (IC), which is inspired by the popular weighting method TF-IDF [80] in texting mining.
- Experiments have been conducted on a real-world dataset. POI achieves the highest F-measure and the best issue coverage compared with the existing methods.

6.2 POI Framework

In this section, we introduce the POI framework in detail, which is illustrated by Fig. 6.1. We first introduce the input of our framework with an example. Then we explain the four stages of the POI framework in difference subsections, including log parsing, vector generating, log clustering, and issue prioritization.

6.2.1 Raw Logs

Raw logs are log messages printed by logging statements during system runtime. Logs are widely employed by developers in reliability assurance tasks, because they are often the only data available for post-mortem analysis. When an operational issue is reported by a user, the related raw logs will be collected and sent to developers along with the issue. Traditionally, developers identify the problem hidden behind the issue mainly by inspecting the accompanied raw logs. Thus, the input of POI is also the accompanied logs. Note that we do not directly solve the problem hidden by our POI framework. We think the problem-solving stage still remains in developers' hand. POI aims at the reduction of human inspection effort on log messages. POI will present several issue clusters to developers and assign different priorities. Thus, developers can inspect a few issues from different issue groups and start with the one with the highest priority.

6.2.2 Log Parsing

Raw log messages collected with operational issues from the users are usually unstructured, because developers are allowed to write free-text log messages in source codes. Specifically, a typical raw log message (e.g., in Fig. 6.2) contains *timestamp* (records the occurring time of an event), *verbosity level* (indicate the severity level of an event, e.g., INFO), and *raw message content* (free-

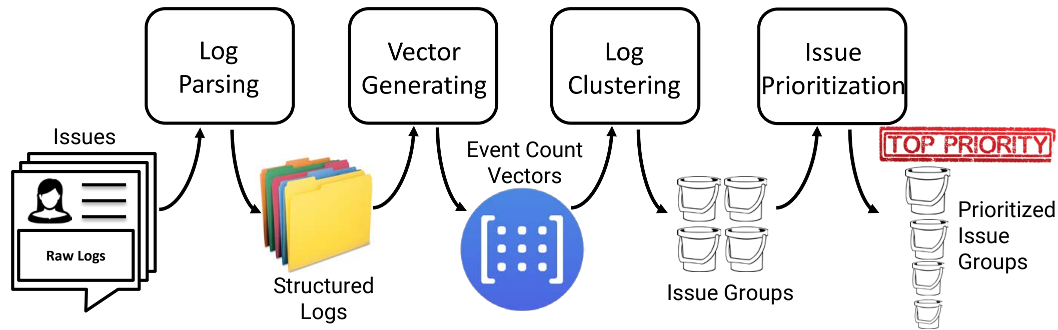


Figure 6.1: Overview of Our Approach

text description of a service operation). Log event type, which is illustrated in green color in Fig. 6.2 explains what system operation a specific raw log message refers to. To automatically analyze logs, we have to figure out the log event type first from the unstructured raw logs. We use the open-source log parsing toolkit provided by He et al. [39] to extract log event type. We use this toolkit because it only requires raw logs as input. After parsing, each raw log message will be transformed into the log event type and corresponding interested fields. For example, if developers want to know the log event type and corresponding block ID for the raw log message in Fig. 6.2, the resulted structured log message will be "blk_1847751055431572519 Receiving block src: dest:".

6.2.3 Vector Generating

After parsing raw log messages into structured log events, we need to further transform them into numerical vectors, which are the input of our hierarchical clustering step. To do so, we first slice structured logs into a set of log sequences by a specific field, for example, the

081109 204020 339 INFO dfs.DataNode$DataXceiver:
Receiving block blk_1847751055431572519 src:
/10.251.126.227:39104 dest: /10.251.126.227:50010

Figure 6.2: A Raw Log Message of Hadoop File System (HDFS)

block ID in the structured log message generated from Fig. 6.2. Each log sequence records the system operations of an operational issue. Then for each log sequence, we generate an event count vector, where each element is the occurrence number of a specific log event.

6.2.4 Log Clustering

After vectorization, we have numerical vectors, each of which is the event count vector of a log sequence related to an operational issue. In this step, we cluster these event count vectors into different groups by two clustering process: clustering by log event appearance and clustering by event count. The first clustering is a coarse-grained procedure, which separates the event count vectors into different groups based on the appearance of log events. The second clustering is performed on the results of the first clustering. For each group produced by the first cluster, POI conducts fine-grained clustering based on the event count numbers of different log events. At the end of this step, POI generates different groups of event count vectors, where each group represents a specific operational issue. This step is the core technical part of our POI approach, which will be introduced in detail in the following sections (Section 6.3.2 and 6.3.3).

6.2.5 Issue Prioritization

In this step, we have operational issue groups. The groups will be sorted by the number of operational issues (i.e., event count vectors) inside, because developers should fix the issues that affect more users first. Then the sorted operational issue groups together with the corresponding logs will be returned to the developers. With the sorted groups, the developers can easily inspect different kinds of operational issues instead of manually checking tons of raw log messages.

6.3 Methodology

In this section, we introduce the log clustering stage in Fig. 6.1. POP's clustering stage contains two clustering step: clustering by log event appearance and clustering by log event count. First, we will introduce our proposed novel weighting method Inverse Cardinality (IC). Then, we will introduce these two steps in detail in the next two subsections.

6.3.1 Inverse Cardinality

Calculate Cardinality

Directly clustering vectors may cause errors, because some log events are useless. Typically, such log events often appear in various system running cases. These log events do not carry much critical information. For example, a *ReceivingBlock()* function may print a log message saying "A package from * received". Then any outer code that calls this function will trigger this logging statement, which makes it appears in various system running cases. However, these kinds of log events rarely carry information that helps us separate different operational issues. On the contrary, they can cause errors, especially false negatives, which puts log sequences with the same operational issue into different groups. To avoid these errors, we define $ECard$ as following to remove some potentially useless log events:

$$ECard(j) = |\{A_{i,j} : 0 < i \leq n\}| \tag{6.1}$$

where $A_{i,j}$ is the value of the j-th element in the i-th event count vector, n is the number of event count vectors. The $ECard(j)$ is the cardinality of the set formed by j-th element values of all event count vectors.

To figure out the importance of different log events, we design a Inverse Cardinality (IC) weighting technique as following:

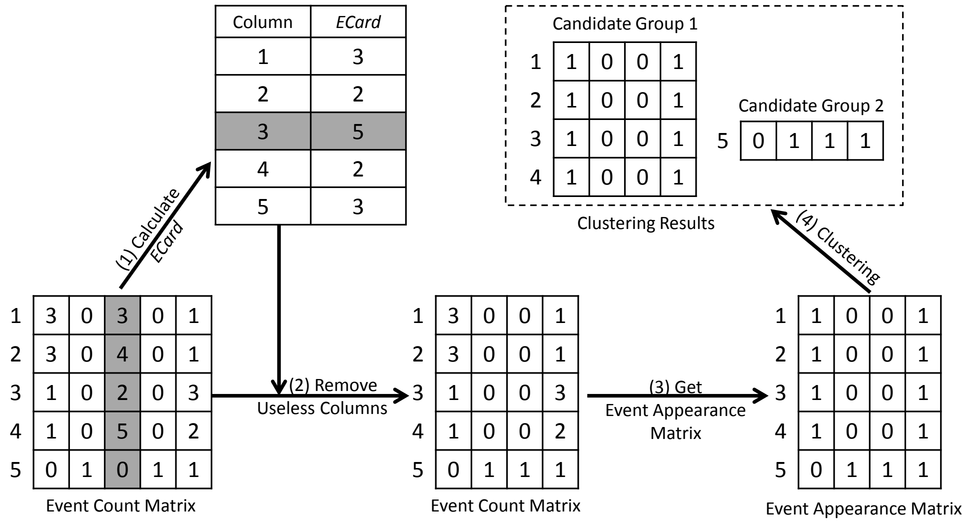


Figure 6.3: An Example of Clustering by Log Event Appearance

$$IC(j) = \begin{cases} \log(2) & \text{if } ECard(j) = 1 \\ \log(\frac{n}{ECard(j)} + 1) & \text{else} \end{cases}$$

where $IC(j)$ is the weight for column j, n is the number of rows in this matrix (i.e., number of operational issues in the issue groups). When $ECard(j) = 1$, all the values in the j-th column are the same, so the j-th column is not helpful in the clustering process. Thus we give it a low weight. Besides, we use $\log(2)$ instead of $\log(1)$ because $\log(1)$ may lead to null vectors in the weighted event count matrix, which do not carry any information. When $ECard(j) \neq 0$, the larger the $ECard(j)$, the smaller the $IC(j)$. This aligns with our intuition that the log event appears in more use cases is less important.

6.3.2 Clustering by Log Event Appearance

The "Log Clustering" step of POI includes a coarse-grained clustering and a fine-grained clustering. In this section, we will introduce the coarse-grained clustering: clustering by log event appearance, which is illustrated by Fig. 6.3. For the matrices in Fig. 6.3,

each row represents a vector related to an operational issue, while a column represents a log event.

Remove Useless Columns

We define $removeThred$ as a log event removing threshold, which can be tuned by developers. If $ECard(j) > rmoveThred$, then POI will remove the j-th column of the event count matrix. For example, in Fig. 6.3, $removeThred = 4$, so the third column, whose $ECard$ is 5, will be removed by POI.

Get Event Appearance Matrix

Similar operational issues will go through similar program paths in system runtime, and therefore will trigger similar logging statements. Thus, the non-zero columns of their event count vector should be similar. Besides, for an operational issue, its signature can be the appearance of one single log event. That means when a developer spots that particular log event in the log sequence of an operational issue, the developer can already put the operational issue in a specific issue group. Based on these two intuitions, we focus on the event appearance in this substep. We get the event appearance matrix from the event count matrix by changing the non-zero elements to 1, which is described in Fig. 6.3.

Clustering

Now we have an event appearance matrix. We calculate the cosine similarity between any two d-dimensional event appearance vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ as following:

$$
\begin{aligned}
Similarity(\boldsymbol{a}, \boldsymbol{b}) &= \frac{\boldsymbol{a} \cdot \boldsymbol{b}}{||\boldsymbol{a}||||\boldsymbol{b}||} \\
&= \frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} a_i^2}\sqrt{\sum_{i=1}^{d} b_i^2}}
\end{aligned} \tag{6.2}
$$

Having calculated the similarity between any two event appearance vectors, we employ Agglomerative Hierarchical Clustering [36] to cluster the event appearance vectors into different groups. Agglomerative hierarchical clustering regards every single vector as a group at the beginning. Then it iteratively merges the nearest two groups until the distance between the two nearest groups is larger than a distance threshold $maxD$. *Complete linkage* is employed to calculate the distance between two groups, which is the distance between two vectors in the two groups respectively that are farthest away from each other. We use complete linkage because we want the resulted groups compact. At the end of this coarse-grained clustering step, the operational issues have been clustered into several groups according to their event appearance vectors. We call these groups *candidate groups*. In Fig. 6.3, we have two candidate groups.

6.3.3 Clustering by Log Event Count

After the coarse-grained clustering, event count vectors have been clustered into candidate groups, where each contains operational issues having similar log event appearance. However, different operational issues may have the same event appearance vectors. To separate these operational issues, we need to conduct a fine-grained clustering on the candidate groups generated by the coarse-grained clustering. In this section, these groups will be further separated into final operational issue groups by the log event counts. The overview of this clustering step is illustrated by Fig. 6.4.

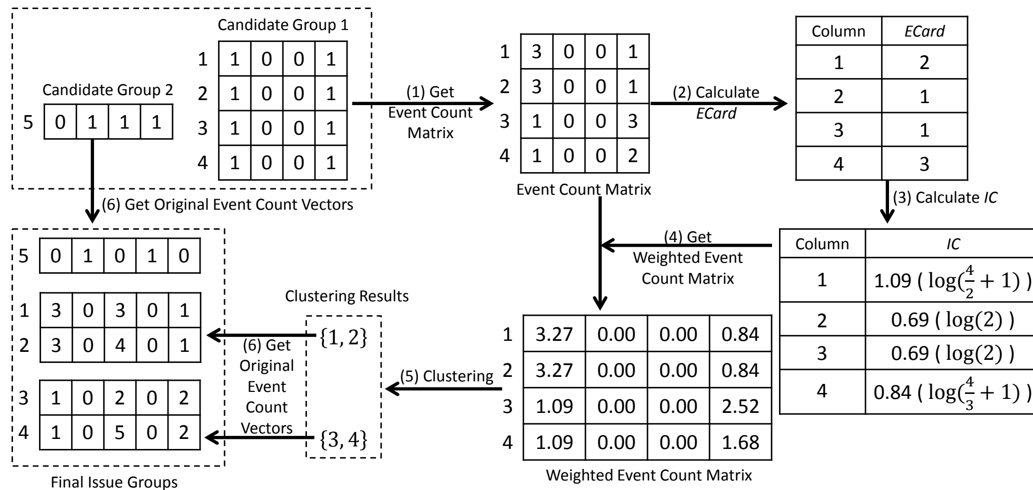


Figure 6.4: An Example of Clustering by Log Event Count

Get Event Count Matrix

Currently, we have several candidate groups. For flexibility, POI allows developers to manually set a threshold $fineThred$ to control the candidate groups that will go through this clustering step. Specifically, if the number of operational issues in a candidate group is larger than $fineThred$, this group will be further separated in this clustering step; otherwise, this group will be regarded as a final group. For example, in Fig. 6.4, we set $fineThred = 1$, so candidate group 1 is further separated while candidate group 2 is regarded as a final group. For a candidate group that will be separated, we get the corresponding event count vectors for the event appearance vectors inside the group. These event count vectors form a event count matrix. Note that the columns removed in the coarse-grained clustering step will not be employed in this fine-grained clustering step, neither.

Calculate Inverse Cardinality (IC)

Directly clustering the event count matrix treats all the log events equally. However, in practice, some log events are more important

than others when clustering operational issues. As discussed before, the log event (i.e., column) with larger cardinality indicates that it appears in more system runtime cases. Similarly, log event with small cardinality indicates that it appears in few system runtime cases. For example, if the cardinality of a log event is 2 and the possible values are 0 and 1, then probably the log event only appears in an anomalous case.

Get Weighted Event Count Matrix

After calculating the weight for each column of the event count matrix, we can transform the event count matrix into a weighted event count matrix. Specifically, we times each element in column j by $IC(J)$.

Clustering

After getting the weighted event count matrix, we compute the cosine similarity (Eq. 6.2) between any two weighted event count vectors. Then we use agglomerative hierarchical clustering to separate the vectors in a candidate group into several final groups. For example, in Fig. 6.4, the candidate group 1 is separated into two groups.

Get Original Vectors

Finally, for each final issue group, we get the original event count vector before log event removing and weighting. We also get the corresponding raw log messages and match them with the event count vectors (not in Fig. 6.4 for simplicity). Then all the final issue groups will be output and passed to the Issue Prioritization step in Fig. 6.1.

6.4 Evaluation

In this section, we conduct experiments to evaluate the effectiveness of our proposed POI method. First, the experiment settings are explained, including dataset, comparison methods, and experiment environment. Then, we use two evaluation metrics for evaluation. First, we evaluate the clustering effect of POI via F-measure [9], because the core technique of POI is hierarchical log clustering. Second, we evaluate the coverage ability of POI, where the higher the coverage ability, the less issue inspection effort needed.

6.4.1 Experiment Settings

Log dataset. In the experiments, we use the HDFS logs from [94] that have well-established system anomaly class labels, which can be regarded as different operational issues in practice. The labels are made based on domain knowledge, which are suitable for our evaluations on prioritizing operational issues. Specifically, the dataset records anomalous system operations on 16,838 HDFS blocks (i.e., 16,838 issues) with a total of 26 anomaly types. Each anomalous HDFS block has exactly one anomaly type, which we use as ground truth in our evaluation. Our goal is to cluster the issues with the same anomalous label into the same group.

Comparison. To prove the effectiveness of POI, we compare it with three representative methods in terms of clustering performance and issue coverage. The ideas of these methods are briefly introduced as follows:

- *Random*: In this method, we randomly present a subset of reported issues to developers without prioritization effort. We regard this method as the baseline method.
- *Exactsame*: In this method, issues with the exact same event count vector will be clustered into the same issue group.

- *LogCluster* [56]: LogCluster is the state-of-the-art log-based method that clusters event count vectors into different groups by hierarchical clustering. It employs IDF (Inverse Document Frequency) and occurrence in normal issues to weight the log events. In the experiments, we do not consider the occurrence in normal issues because it is only used to spot the difference between log sequences in lab and production environment, which is not the focus of this paper.

Experimental Setup. We run all experiments on a Linux server with Intel Xeon E5-2670v2 CPU and 128GB DDR3 1600 RAM, running 64- bit Ubuntu 14.04.2 with Linux kernel 3.16.0. For the Random method, we run each related experiment 10 times and use their average results to avoid bias. For other methods, we run each related experiment once because they are deterministic.

6.4.2 Evaluation of Clustering Algorithm

Evaluation Metric

We use F-measure (i.e., F_1 Score) [9, 66], a commonly-used evaluation metric for clustering algorithms, to evaluate the accuracy of Dedup. The definition is as the following.

$$F_1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{6.3}$$

where $Precision$ and $Recall$ are defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{6.4}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.5}$$

where a true positive (TP) decision assigns two log sequences with the same issue type to the same group; a false positive (FP) decision assigns two log sequences with different issue types to the same

	Precision	Recall	F-measure
Exactsame	1	0.6461	0.7850
LogCluster	0.9559	0.7049	0.8115
POI	0.8904	0.8969	0.8936

Figure 6.5: Evalution of Clustering Ability

group; and a false negative (FN) decision assigns two log sequences with the same issue type to different log groups. If the log sequences are under-partitioning, the precision will be low because it leads to more false positives. If a method over-partitions the log sequences, its recall will decrease because it has more false negatives. Thus, we use F-measure, which is the harmonic mean of precision and recall, to represent parsing accuracy.

Results

The experimental results are illustrated in Fig. 6.5. We do not evaluate the F-measure of the *random* method because it is not a clustering-based method. The *random* method indicates an operational issue handling strategy that the developer randomly choose the next issue from all the candidates. Besides, both LogCluster and POI have some parameters. We use a brute-force method to find the parameter that demonstrates the highest F-measure value. For example, LogCluster needs a distance threshold to control the maximum cosine similarity among different clusters. Since the range of cosine similarity is [0, 1], we run LogCluster with the distance threshold value from 0 to 1, with step size 0.05.

We observe that POI achieves the highest F-measure (i.e., 0.8936) compared with the state-of-the-art methods, which demonstrate the superiority of POI's clustering algorithm. On one hand, we notice that the precision of Exactsame and LogCluster is slightly higher than POI. This means that, compared with existing methods, POI

tends to cluster issues with different issue type into the same group. On the other hand, we observe that the recall of POI is much higher than that of Exactsame and LogCluster. This means that the existing methods will separate issue with the same issue type into different issue groups, causing unnecessary groups. Thus, although the precision of existing methods is slightly higher, they may generate too many unnecessary groups that lead to more human inspection effort. Since our main goal is to alleviate the burden of developers, we believe that our method not only achieves the best F-measure, but is also more practical compared with the others. We will further evaluate its practicability in next section.

6.4.3 Evaluation of Coverage Ability

The goal of POI is to alleviate the burden of developers on operational issues handling. Thus, the fewer log sequences developers need to inspect, the better the prioritization method is. Typical prioritization methods present a list of issue groups to developers ordered by the number of issues inside. Then developers inspect the most important issue group first. When the current issue group is resolved, developers will head to the next issue group in the list. Thus, assume that there are n operational issue types and N issues, the best prioritization technique will generate n issue groups, where each group contains issues with the same issue type. With the best prioritization technique, developers can handle all the N issues by inspecting the n issue groups. We call it perfect coverage, which means that the groups can cover all the issues. In the contrary, an imperfect prioritization technique could either generate too many issue groups (low recall), or lead to more than one issue types in one issue group (low precision). To better evaluate the coverage ability of different prioritization techniques, we draw a coverage ability curve, as in Fig. 6.6.

We evaluate the coverage ability of POI and other three methods

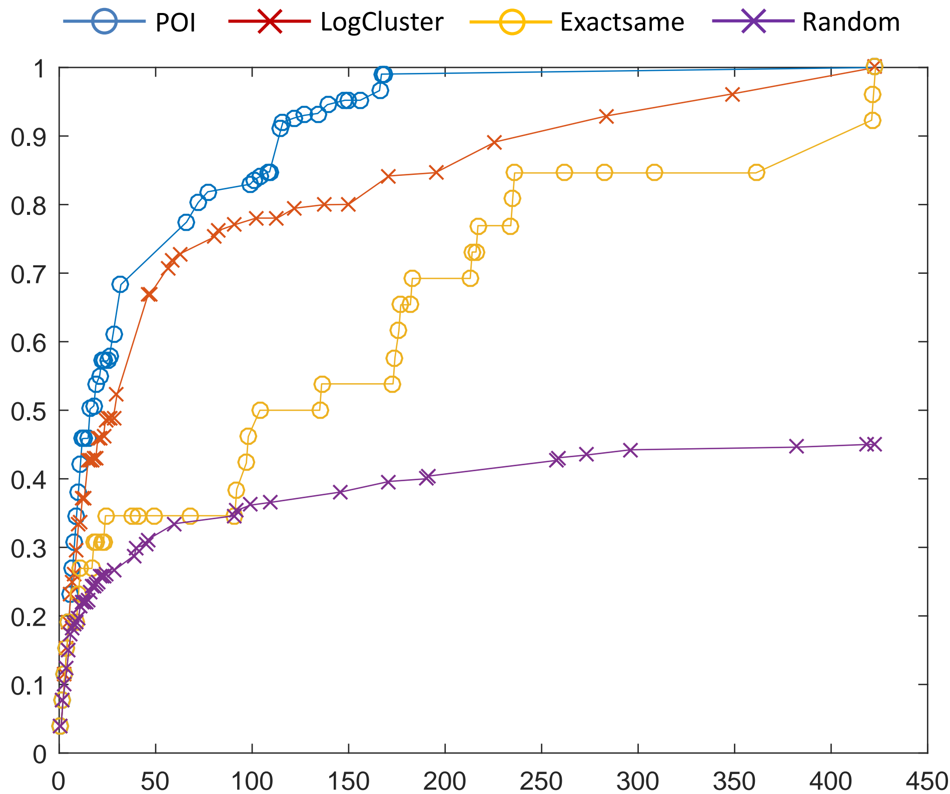


Figure 6.6: Coverage Ability Curve of Different Methods

on the same dataset as in the last section. The results are illustrated by Fig. 6.6. In this figure, the four lines represent the four prioritization methods. The x-coordinate represents the number of issues inspected by developers, while the y-coordinate represents the coverage ratio. For example, assume that we have n issue types, a method can cover $\frac{1}{2}n$ issue types after inspecting 10 issues. Then for this method, we should draw a point at $(10, 0.5)$. For each method, we draw 45 points. To calculate the issue types that the prioritization result covers, we go through all the resulted issue groups. We regard the majority issues in an issue group as the issue type of that group. For example, if a group contains 100 issues, while 80 issues are issue type A, then we think this group covers issue A.

Specifically, both POI and LogCluster have parameters. Different parameter settings will lead to different prioritization performance.

To draw the coverage figure, we use different parameter settings, which generates different clustering results. For example, if we use 0.05 as the maximum distance threshold used in both clustering step of POI, then POI will cluster all the issues into 99 groups, while these groups cover 21 issue types. Given that the dataset contains 26 issue types, the coverage ratio is $21 \div 26 = 0.81$, which is a point in the figure.

Exactsame does not have parameters. For this dataset, it will generate 423 issue groups. To draw the coverage line for Exactsame, we start from the largest issue group, and find out the issue type of this group, and generate the first point $(1, \frac{1}{26})$. Then we find out the issue type of the second issue group. If the issue type has been covered before (i.e., the same as the first issue group), the second point will be $(2, \frac{1}{26})$. Otherwise, the second point will be $(2, \frac{2}{26})$. Following this rule, we generate 423 points for our datasets. Then we select 45 points for clarity and simplicity in Fig. 6.6. For method Random, we randomly select 423 points and use the same drawing rule to generate 423 points. Then similar to drawing Exactsame line, we select 45 points as in Fig. 6.6.

For POI and LogCluster, varying the parameter settings cannot generate points with arbitrary x-values, because the number of generated issue groups (i.e., x-value) are decided by both the parameters and the algorithm. For clarity, we also draw points by the rules employed by Exactsame and Random.

As illustrated by Fig. 6.6, POI's coverage ratio is higher than the other three methods given the same x-value. POI can cover nearly all issues by 170 issue groups, while LogCluster needs 349 issue groups, Exactsame needs 422 issue groups, and Random needs much more. These demonstrate the effectiveness of POI on reducing human effort.

LogCluster also has good coverage ability. However, as we show in Section 6.4.2, the recall of LogCluster is low, which means LogCluster generates unnecessary issue groups. Thus, developers

tend to find that the issue type of current issue group has been identified in prior groups. Compared with LogCluster, Exactsame achieves higher precision but very low recall. Thus, Exactsame generates more unnecessary groups. The random line rises quickly at the beginning, but slows down at around 50. This is because without any prioritization method, random selection can easily encounter redundant issues, which wastes human effort.

Besides, POI, LogCluster, and Exactsame all achieve coverage ratio 1 after developer inspecting 423 issues. This observation shows that an issue type in our dataset can be easily neglected by methods except for Exactsame and Random, because the amount of issues with that issue type is small and their log sequences are similar to some issues with different issue type.

6.5 Summary

In this chapter, we propose an operational issue prioritization framework POI. Specifically, POI contains a coarse-grained log clustering and a fine-grained log clustering. Besides, to better calibrate the weights of event count matrix, we design a novel weighting method Inverse Cardinality. Extensive experiments have been conducted on a real-world dataset with 16,838 issues. The experimental results demonstrated the effectiveness of our framework.

□ **End of chapter.**

Chapter 7

Location-Based Web Services QoS Prediction via Historical QoS Logs

In service-oriented development, developers need to employ the most suitable Web services based on QoS values (e.g., response time). Most of these values are predicted by QoS values recorded by limited historical QoS logs. We design a location-based QoS prediction method that applies a novel hierarchical matrix factorization (HMF) model on historical QoS logs. The main points of this chapter are as follows. (1) It designs a hierarchical matrix factorization (HMF) model to predict QoS values via limited historical QoS logs. (2) It leverages location information of both developers and services in our model. (3) It conducts extensive experiments to demonstrate the effectiveness of HMF.

7.1 Introduction

Nowadays, more and more Web services designed by different organizations emerge on the Internet, providing a variety of functionalities. These Web services are widely employed as components in complex distributed systems, which greatly reduce software development time. But during the developing process, the developer will often find out a lot of functionally-equivalent Web services due to the fact that the number of Web services is experiencing a rapid growth.

Consequently, Web service recommendation [103, 96] is recognized as an effective solution to assist developers in determining the most suitable candidate services.

To facilitate effective Web service recommendation, the quality of candidate services needs to be assessed from non-functional properties. Quality-of-Service (QoS) is a group of attributes (e.g., response time, throughput, reputation, etc.) that are usually employed to characterize the non-functional properties of Web services [26, 87, 59]. In principle, during Web service recommendation, services with similar functionalities are compared with each other automatically based on their QoS properties. Then the most suitable Web services in terms of user-defined QoS requirements can be recommended to the user. As a result, the user can just select from a small list of Web services returned by the recommendation system instead of struggling in all candidate services.

However, in practice, it is not easy to obtain the QoS values of all the candidate services, due to the following three reasons: 1) The QoS values need to be assessed from point of view of users (i.e., developers), because different users may perceive different QoS values. 2) Each user usually just invokes a handful of Web services. Thus developers only have limited historical QoS logs. 3) It is time-consuming and resource-consuming to assess all the QoS values by invoking candidate services one by one, due to the large number of users and services. The QoS values of Web services observed by different users can be represented as a user-service matrix, whose rows represent users, columns represent services and entries are observed QoS values. But there are many missing values in the user-service matrix. To address this problem, QoS prediction is proposed to get approximated QoS values for those missing values in the user-service matrix.

Matrix factorization (MF) is a model-based collaborative filtering method, which has been used to make QoS prediction and widely studied in recent years. As other model-based collaborative fil-

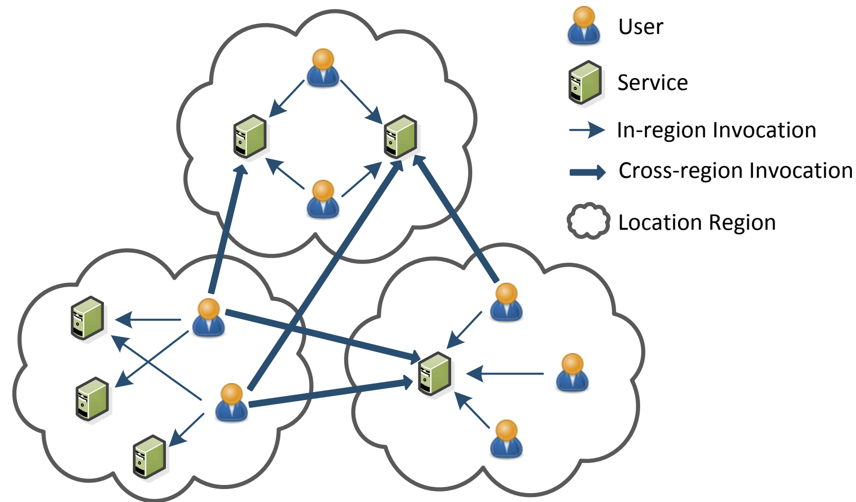


Figure 7.1: Web Services Invocation Scenario

tering methods do, matrix factorization trains a model according to historical invocation records and uses patents found to predict QoS values for the missing values in the user-item matrix. In matrix factorization, we suppose that user-perceived QoS values are determined by a few latent features. These latent features (e.g. network bandwidth, I/O operations, firewalls) not only affect users, but also have impact on the service side. Thus, the user-service matrix is approximated by two low-rank matrices, which represent the influence of latent features on users as well as services, respectively. Matrix factorization achieves good performance in traditional recommendation systems (e.g. movie recommendation systems) where entries in matrix are user-given ratings on different items (e.g. movies). Because each existing entity in that matrix is rated by a specific user, which is subjective, it can reflect the user's preference on an item. However, in Web service context, user-perceived QoS values of services are largely affected by physical factors. Location is such a key factor. For example, users and services in close locations tend to have small response time values. In Fig. 7.1, we have users and services which are in three location regions. Among all invocations, corresponding QoS values (e.g.

response time values) of invocations in the same location region are more likely to have higher similarity (they tend to be small) compared with those cross-region ones.

To make use of the location information and improve the performance of our model, we propose a hierarchical matrix factorization method to predict QoS values for the unobserved user-service pairs. We firstly make use of clustering methods to separate users and services into several user-service groups according to their location. After that, QoS values of these users and services are represented as local user-service matrices. Different from global matrix which contains all users and services, local matrices only contain users and services in the same location region (same as cluster in this chapter). A simple way to utilize location information is to only conduct matrix factorization on local matrices and simply put together their prediction results. However, it will degrade the prediction performance because we don't use any information of global matrix at all. To utilize both global and local information, our model performs matrix factorization on local matrices and global matrix sequentially in each approximation step. This model is run in a hierarchical way that in each approximation step, we linearly combine the predicted results of both global matrix and local matrices. Finally, based on the integrated predicted results, services with the best QoS values are recommended to corresponding users.

The main contributions of this chapter are as follows:

- We design a hierarchical matrix factorization (HMF) model to predict QoS values via limited historical QoS logs.
- HMF leverages location information of both developers and services in our model.
- Extensive experiments have been conducted, which demonstrates the effectiveness of our proposed HMF model.

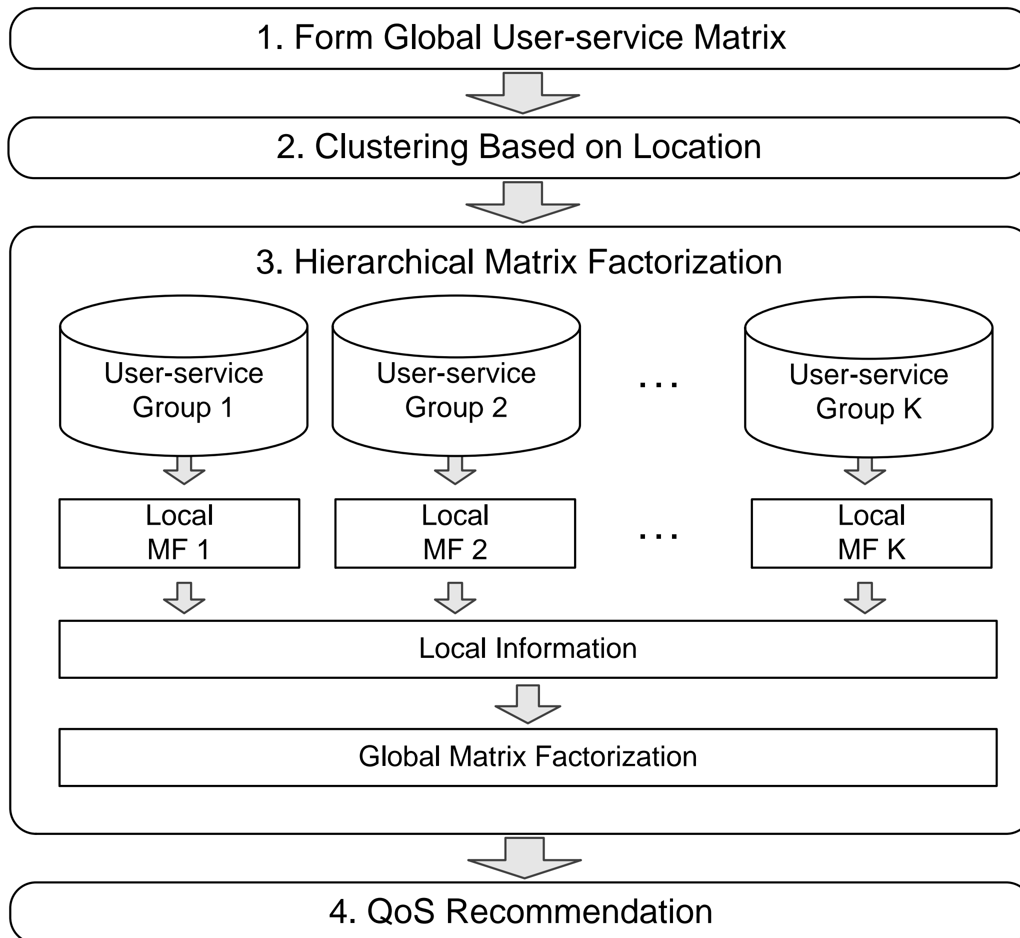


Figure 7.2: Framework of Hierarchical Web Service Recommendation System

7.2 Framework of Web Service Recommendation

As we mentioned in Section 7.1, nowadays developers tend to utilize existing Web services provided by third parties to build complex distributed systems. An important issue for them is to choose the most suitable services among all functionally-equivalent ones without invoking all service candidates by themselves. Our recommendation system acts as a platform for these users to share their historical invocation records and obtain believable service recommendation according to their non-functional needs. The overview of our QoS-based hierarchical Web service recommendation system is shown in

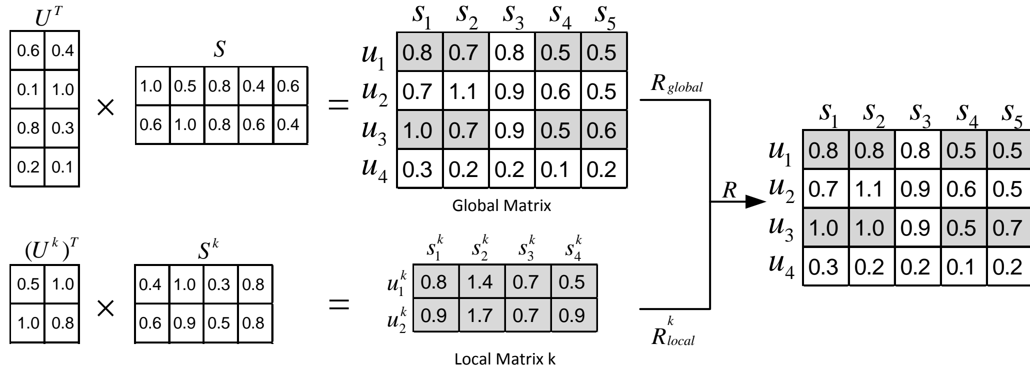


Figure 7.3: An Example of QoS Prediction by Hierarchical Matrix Factorization ($\alpha = 0.8$)

Fig. 7.2, which includes these main steps:

1. We collect and formalize existing information of users and services, including their IP address, longitude, latitude, invocation records shared by them, to name a few. Then all existing Web service invocation records will be used to form a global user-service matrix. Due to the fact that most of the users only called a few Web services before, the global matrix is very sparse.

2. In this step, we utilize longitude and latitude information to map users as well as services into a 2-dimensional space. Then we cluster all user nodes and service nodes into several user-service groups according to their coordinates in that space. In our method, each group contains both users and services, which coincides with the idea to make use of the location of users and services at the same time. After clustering, users, services and their corresponding invocation records in each group can be used to form a local user-service matrix.

3. The above-mentioned steps can be viewed as preprocessing steps, whose outcomes are one global matrix and several local matrices. Then our hierarchical matrix factorization model is used to predict missing values. In each approximation step, we firstly perform matrix factorization on all local matrices.

After local matrix factorization, we obtain approximate QoS values for local matrices, which are referred to local information in Fig. 7.2. Then we calculate QoS values in global matrix. Instead of using only the product of user feature vectors and service feature vectors like traditional matrix factorization methods, for QoS values between users and services in user-service groups, we linearly combine predicted result of global matrix factorization and local matrix factorization to obtain the final prediction. User feature vectors and service feature vectors are columns of low-rank matrices used to perform QoS prediction during matrix factorization.

4. Now, all unobserved entries in the global matrix are predicted by our hierarchical matrix factorization model. Thus, our system is ready to recommend the services with the most suitable non-functional properties to all users in our system based on predicted results. For example, if an existing user, who is the i-th one in our global matrix, wants to find out a Web service with least response time among functionally-equivalent candidates. We just need to take out the i-th row from the global matrix, which can be regarded as a 1-dimensional vector. The service with the smallest numeric value in this vector will be recommended to that user.

7.3 Hierarchical Matrix Factorization

7.3.1 Overview

Our hierarchical matrix factorization consists of two main steps, which are clustering step and prediction step (modeling step). In clustering step (SubSection 7.3.2), users and services are clustered into different user-service groups according to their location information. In prediction step, our model is trained hierarchically on historical invocation records. There are two procedures in each training

iteration, which are local matrix factorization (SubSection 7.3.3) and global matrix factorization (SubSection 7.3.4). Local matrix factorization will be done first and the result of it is used by global matrix factorization in each iteration.

To explain the "hierarchical" concept clearly, a straightaway example is given in Fig. 7.3. In this example, we assume there is one local matrix for simplicity, which means one user-service group is found by k-means. The calculation of Local Matrix k ($R^k_{local} = (U^k)^T S^k$) is local matrix factorization, while the remaining part of this figure is global matrix factorization. In global matrix factorization, the product of two low-rank matrices U^T and S is calculated at first, which is Global Matrix in Fig. 7.3. Then we combine Global Matrix and Local Matrix k to obtain the hierarchical prediction in this iteration, which is the rightmost matrix in this figure. u^k_1, u^k_2 in Local Matrix k and u_1, u_3 in Global Matrix are actually the same users, respectively. s^k_1,s^k_2,s^k_3 and s^k_4 are corresponding services in Local Matrix k to s_1, s_2, s_4 and s_5 in Global Matrix. Blocks in matrix containing QoS values between users and services which are both in the user-service group are marked as grey background, while the remaining ones are white. In the following, we will focus on block (u_3, s_5) in the rightmost matrix. To predict the QoS value $R(3, 5)$ inside, we calculate in this way:

$$R(3, 5) = 0.8 \times R_{global}(3, 5) + (1 - 0.8) \times R^k_{local}(2, 4) \quad (7.1)$$

where $R_{global}(3, 5)$ is the QoS value of (u_3, s_5) in Global Matrix and $R^k_{local}(2, 4)$ is the QoS value of (u^k_2, s^k_4) in Local Matrix k. The impact of parameter α will be discussed in Section 7.4.4.

7.3.2 Users and Services Clustering

Location is used in our hierarchical matrix factorization method to improve prediction accuracy because of these following reasons: (1) Location information, which is represented by longitude and latitude

in this paper, is an attribute that owned by every user and service. (2) Longitude and latitude of all users as well as services can be crawled on the Internet. (3) In Web service context, location does carry valuable information because geographically-close nodes tend to share similar network infrastructure, which to an extent affects QoS values such as response time. Although users and services located in close places may employ different network configurations, which also affect QoS values to varying degrees, it has been observed that this distinction has much less influence than the location information [58]. Because matrix factorization performs better on matrix with smaller variance, users and services are clustered into some user-services groups according to their longitude and latitude, which form local matrices in our method.

Since longitude and latitude information is selected to cluster user nodes and service nodes, now the problem is how to cluster 2-dimensional points into different groups. We chose k-means to cluster nodes in this problem because it is fast and tends to form globular clusters, which fits our locational similarity concept well. One big problem of k-means algorithm is that the clustering effect is closely related to the choice of initial mean points. However, in this problem, since user nodes as well as service nodes are all 2-dimensional, it is convenient for us to visualize all the points and predefine suitable initial mean points. After clustering, we may find out some clusters in which the number of users or services is very limited (close to or even less than the number of latent features we set in our model). This phenomenon often exists because of the geographical maldistribution of users or services in real-world datasets. Users and services belong to those clusters are defined as outliers. For these outliers, we do not consider them in local matrix factorization, but they will be used in the global matrix factorization step. Because too few users or services do not carry enough information to train the local matrix factorization model well, which leads to inaccurate prediction results.

7.3.3 Local Matrix Factorization

We employ traditional matrix factorization to perform prediction for all local user-service matrices. In local matrix factorization, each local user-service matrix is predicted by two low-rank matrices U^k and S^k, whose sizes are $d \times m_k$ and $d \times n_k$, where m_k is the number of users, n_k is the number of services and d is the number of latent features in our model. For matrix U^k or S^k, columns represent how much corresponding latent features will affect QoS values on user side or service side. Missing QoS values in local user-service matrix k are predicted by minimizing the following formula:

$$\mathcal{L}_k = \frac{1}{2}\sum_{i=1}^{m_k}\sum_{j=1}^{n_k} I_{ij}^k (R_{ij}^k - (U_i^k)^T S_j^k)^2 + \frac{\lambda_u^k}{2}\|U^k\|_F^2 + \frac{\lambda_s^k}{2}\|S^k\|_F^2 \quad (7.2)$$

where I_{ij}^k indicates whether QoS value on service j observed by user i in local matrix k is missing. If it is missing, I_{ij}^k will be 0, otherwise, its value is 1. R_{ij}^k means the available QoS value that user i experienced on service j in local matrix k. The rest two terms are regularization terms that help us get rid of overfitting issues.

To get a local minimum of the objective function in Equ. 7.2, we apply the gradient descent algorithm on both U_i^k and S_j^k:

$$(U_i^k)' \leftarrow U_i^k - \eta_u^k \frac{\partial \mathcal{L}_k}{\partial U_i^k} \quad (7.3)$$

$$(S_j^k)' \leftarrow S_j^k - \eta_s^k \frac{\partial \mathcal{L}_k}{\partial S_j^k} \quad (7.4)$$

where $\frac{\partial \mathcal{L}_k}{\partial U_i^k}$ and $\frac{\partial \mathcal{L}_k}{\partial S_j^k}$ are calculated by:

$$\frac{\partial \mathcal{L}_k}{\partial U_i^k} = \sum_{j=1}^{n_k} I_{ij}^k((U_i^k)^T S_j^k - R_{ij}^k)(S_j^k) + \lambda_u^k U_i^k \tag{7.5}$$

$$\frac{\partial \mathcal{L}_k}{\partial S_j^k} = \sum_{i=1}^{m_k} I_{ij}^k((U_i^k)^T S_j^k - R_{ij}^k)((U_i^k)^T) + \lambda_s^k S_j^k \tag{7.6}$$

We set $\lambda_u^k = \lambda_s^k$ in all experiments to reduce the complexity of our model.

7.3.4 Global Matrix Factorization

As mentioned in Section 7.1, matrix factorization predicts all missing values by minimizing the error between prediction results and historical QoS records. In traditional matrix factorization, location information is not taken into consideration, which to some extent degrades the performance of the model. But we observed that the global user-service matrix can actually be regarded as several user-service groups, where users and services are located in similar places, and remainders which are geographically far apart from those groups. Thus, we cluster users and services into user-service groups according to longitude and latitude, each of which contains QoS values with small variance that benefits the performance of matrix factorization. An extreme idea to utilize knowledge given by local matrices is to only apply matrix factorization model on those user-service groups independently, which will make fully use of local information but lead to the loss of global structure. Thus, to consider both global and local information, it is natural to predict QoS values by the linear combination of matrix factorization on global matrix as well as local matrices. Hence, the following term is designed:

$$\alpha U_i^T S_j + (1-\alpha)\hat{R}_{ij}^k \tag{7.7}$$

where U_i^T and S_j represent global user feature vector and global service feature vector respectively. $\hat{R}_{ij}^k$ is calculated by $(U_i^k)^T S_j^k$, which are the corresponding local ones. Notice that although U_i and U_i^k are related to the same user, the value of i is actually different because the numbers of users in these two matrices are not equal. We both use i here just for simplicity and clarity. It is also the case for S_j and S_j^k. This term integrates the approximate values given by global vectors and local vectors, which coincides with the idea to utilize both global structure and local information at the same time. α is a tunable parameter that indicates how much global information we use in our hierarchical model. The value of α is related to the dataset we use. This term will be integrated into traditional matrix factorization to obtain our hierarchical model:

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{n} I_{ij}(R_{ij} - (\alpha U_i^T S_j + (1-\alpha)\hat{R}_{ij}^k))^2 + \frac{\lambda_u}{2}\|U\|_F^2 + \frac{\lambda_s}{2}\|S\|_F^2 \tag{7.8}$$

Since some users and services do not belong to any user-service groups, for those corresponding missing QoS values, we only use global matrix factorization to perform prediction. To formalize this concept, α is selected by:

$$\alpha = \begin{cases} 1 & \text{if } u_i \text{ or } s_j \text{ or both are not in any local groups} \\ \alpha_k & \text{if } u_i \text{ and } s_j \text{ are both in local group } k \end{cases} \tag{7.9}$$

Similar to local matrix factorization, we apply gradient descent to approximate. User feature vectors and service feature vectors are updated as following:

$$U_i' \leftarrow U_i - \eta_u \frac{\partial \mathcal{L}}{\partial U_i} \tag{7.10}$$

$$S_j' \leftarrow S_j - \eta_s \frac{\partial \mathcal{L}}{\partial S_j} \tag{7.11}$$

where $\frac{\partial \mathcal{L}}{\partial U_i}$ and $\frac{\partial \mathcal{L}}{\partial S_j}$ are calculated by:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial U_i} =& \sum_{j=1}^{n} I_{ij}(\alpha U_i^T S_j + (1-\alpha)\hat{R}_{ij}^k \\ &- R_{ij})(\alpha S_j) + \lambda_u U_i \end{aligned} \tag{7.12}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial S_j} =& \sum_{i=1}^{m} I_{ij}(\alpha U_i^T S_j + (1-\alpha)\hat{R}_{ij}^k \\ &- R_{ij})(\alpha U_i^T) + \lambda_s S_j \end{aligned} \tag{7.13}$$

In all experiments, we set $\lambda_u = \lambda_s$ for simplicity.

7.4 Experiments

7.4.1 Dataset Description

A real-world dataset consisting of 339 users and 5,825 services is used in all our experiments. This dataset contains much useful information about users and services, including IP address of each user, WSDL address of each service, users' longitude as well as latitude and so on. This dataset was introduced in detail in a related paper [105]. However, longitude and latitude information of services was not published. Thus we obtain their location information by an iplocation service [1].

7.4.2 Metrics

We use Mean Absolute Error (MAE) and Normalized Mean Absolute Error (NMAE) to measure the prediction accuracy of our proposed model. The definition of MAE is:

$$MAE = \frac{\sum_{ij} |R_{ij} - \hat{R}_{ij}|}{N} \tag{7.14}$$

[1] http://www.iplocation.net/

where R_{ij} represents the observed QoS value between user i and service j, while $\hat{R}_{ij}$ denotes the QoS value predicted by our hierarchical matrix factorization model between the corresponding user-service pair. N is the number of missing QoS values in the user-service matrix. Different from MAE that calculate the absolute average error, NMAE is the standard MAE normalized by the mean of expected QoS values [103]:

$$NMAE = \frac{MAE}{\sum_{ij} R_{ij}/N} \tag{7.15}$$

7.4.3 Comparison

To prove the effectiveness of our hierarchical matrix factorization method, we ran extensive experiments on state-of-the-art QoS prediction methods and compare our method with them. Here is a brief introduction about those popular methods:

- **UMEAN:** UMEAN uses the mean of QoS values perceived by a user on all services he/she called.
- **IMEAN:** In this method, we predict a QoS value by calculating the mean of all existing historical records on that service observed by different users.
- **UPCC:** This approach [82] utilize historical invocation records of similar users to perform prediction.
- **IPCC:** The overall idea of this approach is the same with UPCC, but instead of making use of similar users, it pays attention to digging out some services alike. Then the QoS values of similar services observed by the specific user are used in prediction step.
- **WSRec:** This method [103] is the hybrid one that linearly combines UPCC and IPCC, which takes advantage of both similar users and similar services.

Table 7.1: Parameters

Parameter	Value
λ_u, λ_s	35
λ_u^1, λ_s^1	10
λ_u^2, λ_s^2	20
Dimensionality	10

Table 7.2: Value of α

Density	0.15	0.20	0.25	0.30
α	0.8	0.8	0.8	0.9

- **PMF:** This method is proposed by Mnih and Salakhutdinov [70]. The product of two low-rank matrices is used as the predicted user-service matrix.
- **LBR2:** Lo et al. [58] considered location of users and add a related regularization term to PMF model.

Since the user-service matrix is sparse in real-world cases, we randomly remove some historical records to make our experiments more realistic. Then we will have some user-service matrices with different densities. In our experiment, each QoS prediction method is run on 4 different matrices, whose densities are 15%, 20%, 25% and 30% respectively. A matrix with 20% density means that there are 20% available user-service invocation records for us to regard as training set, while the remaining 80% are ones waiting to be predicted. K-means algorithm helps us separate all the users and services into 5 user-service groups. After the check of the number of users and services in each group, 3 groups are deleted because there are too few users in them. As we mention in Section. 7.3, the number of useful groups is significantly related to the distribution of user nodes and service nodes as well as the number of nodes in the dataset. The parameters we used in our experiment are listed in Table. 7.1 and Table. 7.2. For the purpose of simplicity, we set $\alpha_1 = \alpha_2$ in all our experiments, and α is used to represent for these

Table 7.3: Performance Comparison (MAE)

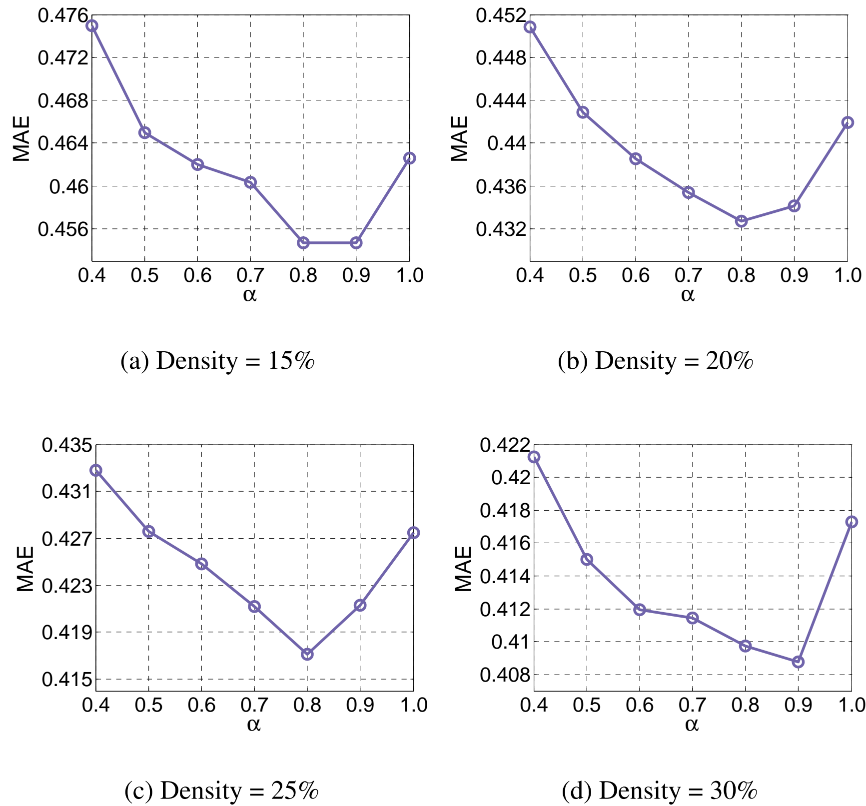
Methods	Density 15%	Density 20%	Density 25%	Density 30%
UMEAN	0.8767	0.8735	0.8740	0.8735
IMEAN	0.6823	0.6806	0.6781	0.6789
UPCC	0.5196	0.4911	0.4715	0.4574
IPCC	0.5244	0.4629	0.4389	0.4211
WSRec	0.4999	0.4530	0.4310	0.4147
PMF	0.4626	0.4420	0.4275	0.4173
LBR2	0.4596	0.4404	0.4242	0.4153
HMF	**0.4547**	**0.4327**	**0.4171**	**0.4088**

Table 7.4: Performance Comparison (NMAE)

Methods	Density 15%	Density 20%	Density 25%	Density 30%
UMEAN	0.9650	0.9608	0.9604	0.9599
IMEAN	0.7512	0.7489	0.7453	0.7461
UPCC	0.5740	0.5368	0.5168	0.5019
IPCC	0.5753	0.5079	0.4814	0.4681
WSRec	0.5501	0.4961	0.4727	0.4593
PMF	0.5091	0.4865	0.4705	0.4595
LBR2	0.5060	0.4846	0.4667	0.4574
HMF	**0.5006**	**0.4766**	**0.4589**	**0.4501**

two.

Table 7.3 and Table 7.4 show us the MAE and NMAE of different methods respectively on 4 matrices with density from 15% to 30%. The MAE and NMAE of all methods decrease as the matrix density become larger, which means more information of users and services will benefit the prediction performance. Besides, it is obvious that the MAE and NMAE of our method are consistently lower than others under all matrix density settings. That means our method outperforms others under all circumstances. Thus, performing matrix factorization hierarchically and making use of geographical information really help us improve QoS prediction model in prediction accuracy.



(a) Density = 15%

(b) Density = 20%

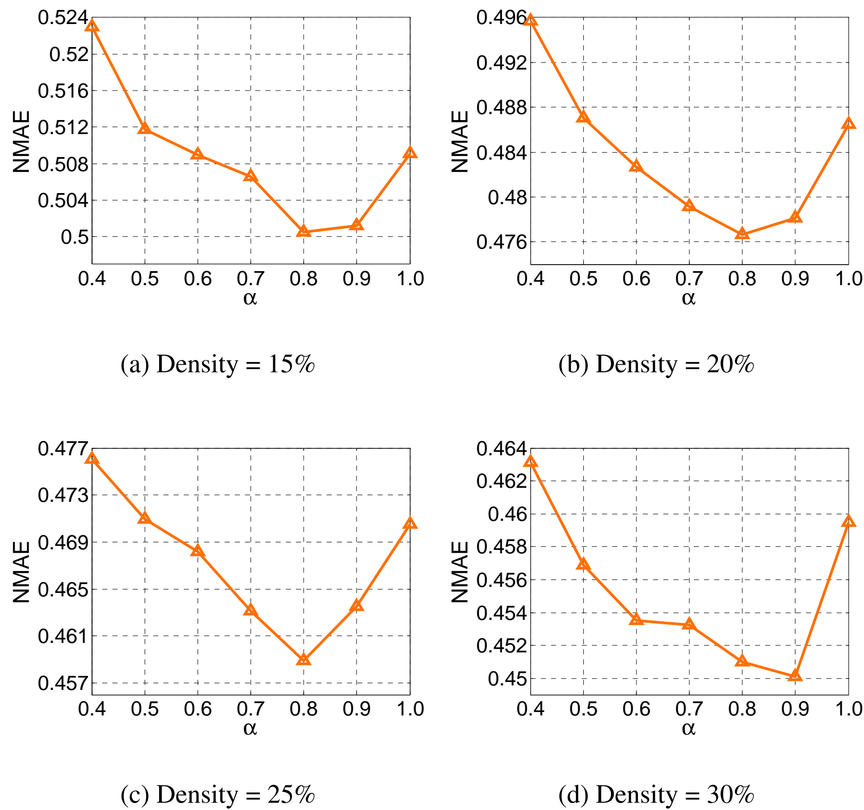
(c) Density = 25%

(d) Density = 30%

Figure 7.4: Impact of α on MAE

7.4.4 Impact of α

In our hierarchical matrix factorization model, parameter α controls how much local information we use in QoS prediction procedure. If α is set to be 1, no local information is taken into consideration. If α is 0, QoS values, whose corresponding users and services are in the same user-service group, are predicted by local matrix factorization independently without any information from global context. That is to say, we only use historical invocation records of geographically-close users and services to perform prediction. In a word, α is utilized to keep a good balance between global context and local information. To study the influence of α on our model and find an optimal one, we tune density from 15% to 30%, with a step size 5%.



(a) Density = 15%

(b) Density = 20%

(c) Density = 25%

(d) Density = 30%

Figure 7.5: Impact of α on NMAE

Fig. 7.4 and Fig. 7.5 show us, under 4 different matrix density settings, the change of MAE and NMAE as the value of α varies from 0.4 to 1.0. We can see that given matrix density 15%, 20% or 25%, both MAE and NMAE are the lowest when α is around 0.8. That means our hierarchical matrix factorization model performs the best when $\alpha = 0.8$ for density 15%, 20% or 25%, while for density 30%, $\alpha = 0.9$ is the most suitable choice.

In this paragraph, we will raise a discussion for the condition that matrix density is 15%. We observe from the figure that when α is 0.4, both MAE and NMAE are high. As the value of α changes from 0.4 to 0.8, MAE as well as NMAE drop down sharply at first but become smoothly as α gets close to the optimal value. That indicates too little global context usage harms the performance of

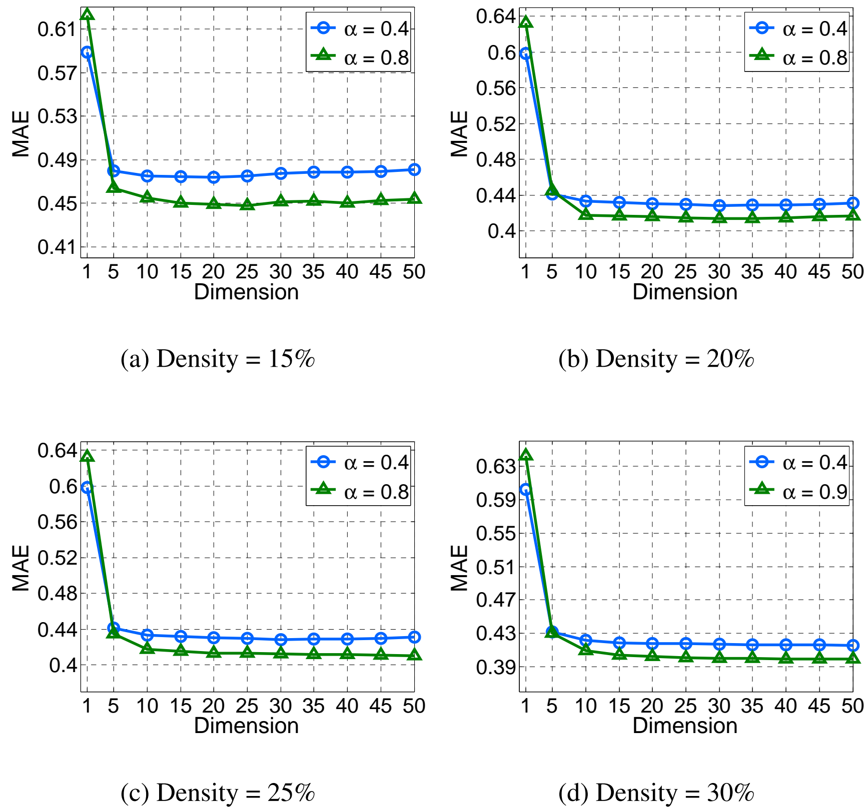
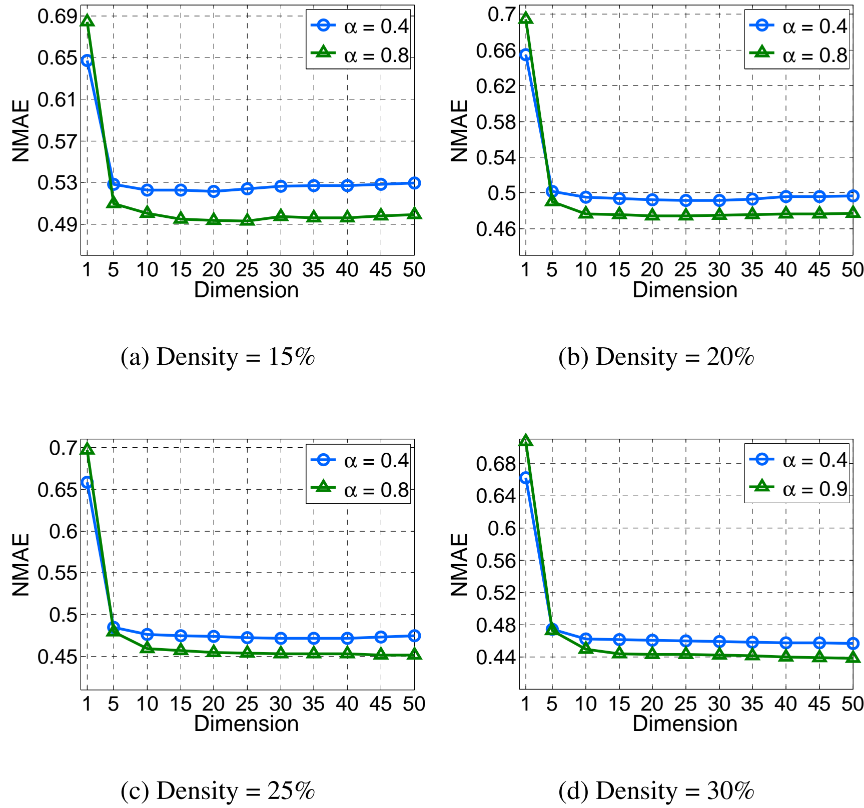


Figure 7.6: Impact of Dimensionality on MAE

our model. Adding the impact of global context will highly increase the prediction accuracy at the beginning, but the effect becomes small when the model is near a balance between local information and global context. We can also notice that when α is larger than 0.8, the MAE and NMAE get larger, which tells us that ignorance of local information will lead to the degradation of performance. When matrix density is 20%, 25% or 30%, the changing tendency and the reason of the change is similar to what we have just discussed.

7.4.5 Impact of Dimensionality

In our proposed method, dimensionality means the number of latent features that will affect the user-perceived QoS values on services. If



(a) Density = 15%

(b) Density = 20%

(c) Density = 25%

(d) Density = 30%

Figure 7.7: Impact of Dimensionality on NMAE

this parameter is small, it indicates that only a few key latent features determine QoS value. If dimensionality is set to be a large number, it is assumed that there are many latent features contribute collectively to the final prediction result. To study the impact of dimensionality on our model we tune $\alpha = 0.4$ and $\alpha = 0.8$ for matrix density 15%, 20%, or 25%. When density is 30%, α was set to be 0.4 and 0.9.

Fig. 7.6 and Fig. 7.7 illustrate the impact of dimensionality on MAE and NMAE of our model respectively. Both MAE and NMAE are high at first and decrease rapidly as dimension increase. It shows that only few latent features can not lead to a good prediction result, so we can effectively improve the performance by raising the dimensionality. However, the speed of decrease slows down as the dimensionality goes up. When dimensionality exceeds a threshold,

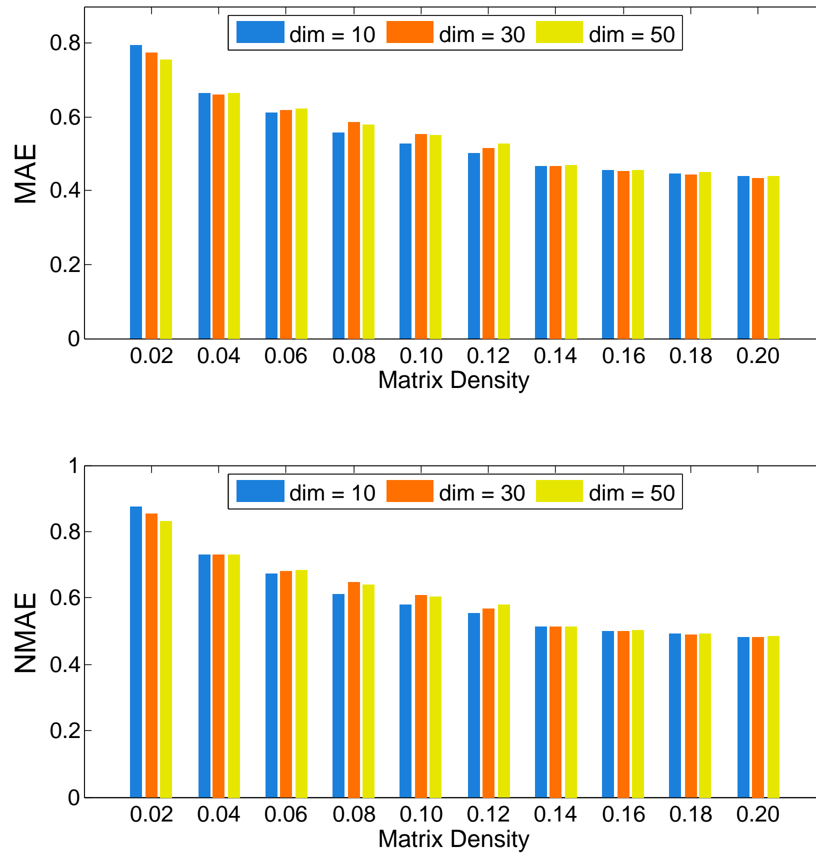


Figure 7.8: Impact of Matrix Density

MAE and NMAE even begin to increase little by little. These can be explained by following two reasons: (1) If dimensionality is larger than a threshold, our model comes across the overfitting problem, which will degrade the performance. (2) The number of users as well as services in our local user-service groups is smaller than that of the global matrix. When making use of local information, an oversize dimensionality will result in bad prediction performance. Thus the overall prediction accuracy will be affected negatively.

7.4.6 Impact of Matrix Density

In the problem context discussed in this paper, matrix density is the ratio of the number of observed user-service invocation records against the product of the number of users and services. It also indicates how much available information we have to help us make prediction. To study the effect of matrix density, we set $\alpha = 0.8$. Besides, we consider three different values of dimensionality, which are 10, 30 and 50.

Fig. 7.8 shows us the corresponding MAE and NMAE of our model from matrix density 2% to 20% with a step size 2%. The figure illustrates that as the matrix density goes up, the MAE and NMAE decrease rapidly at first. When matrix density becomes larger, the speed of decrease slows down. That means, when there are hardly any historical invocation records in the user-service matrix, the best way to improve recommendation performance is to motivate users to report more QoS values or try some services which have not been called before. But when the number of invocation records grows larger, it would be better to focus on improvement of prediction model instead.

7.5 Summary

This chapter presents a new model-based collaborative filtering method to predict missing QoS values via limited historical QoS logs. Geographically-close users and services are clustered to form groups, which makes up several small user-service matrices. A hierarchical matrix factorization model is designed to integrate global matrix factorization and local matrix factorization. Experimental results show that our model outperforms the state-of-the-art methods.

□ **End of chapter.**

Chapter 8

Conclusion and Future Work

In this chapter, we summarize the main contributions of this thesis and provide several interesting future directions.

8.1 Conclusion

System reliability management is critical to modern systems. However, in many cases, traditional approaches relying on manual inspection are impractical for modern systems, because modern systems are often complex and large-scale, so the volume of runtime data (e.g., system logs) is also large. In this thesis, we have developed novel approaches to automatically handle runtime data of modern systems, especially system logs, and further analyze these data in system reliability management.

In particular, in Chapter 3, we conduct an evaluation study on four representative log parsers in terms of accuracy, efficiency, and effectiveness on subsequent log mining. Specifically, we evaluate their accuracy and efficiency on five real-world log datasets with more than 10 million log messages in total. We evaluate their effectiveness on subsequent log mining on an HDFS dataset with 16,838 anomalies. Based on the evaluation results, we summarize six insightful findings to guide the design of log parsers in future. Moreover, we publicly release the implementation of all studied log

parsers as a reusable toolkit.

In Chapter 4, we propose a parallel log parsing framework, namely POP, to parse large-scale system logs for modern systems. POP contains specially designed heuristic parsing rules and log group clustering algorithm. We implement POP on top of Spark, a large-scale data processing platform, with customized functions and usage of Spark operations. POP outperforms existing log parsing methods in terms of accuracy on all the five real-world log datasets. Besides, POP is efficient, and it can parse 200 million HDFS log messages in 7 mins. The source code of POP has been released for reuse by researchers and practitioners.

In Chapter 5, we design an online log parsing method, namely Drain, to parse system logs in a streaming manner. The core idea of Drain is a fixed depth tree with specially designed heuristic rules embedded. Drain can parse log messages online and update its parsing rules dynamically. Drain achieves the highest parsing accuracy compared with state-of-the-art online parsers. Besides, it largely improves the efficiency of online parsers. The source code of Drain has been made open-source for future reuse.

In Chapter 6, we propose an operational issues prioritization framework, namely POI, to facilitate the operational issues handling process. Specifically, we cluster issues into issue groups according to corresponding log sequences, and prioritize them based on the number of issues inside. Besides, we design a hierarchical log clustering algorithm, which contains a coarse-grained clustering and a fine-grained clustering. Extensive experiments have been conducted on a real-world HDFS issue dataset. The experimental results show that POI achieves the highest F-measure, and can cover most issues with same manual effort.

In Chapter 7, we design a location-based hierarchical matrix factorization method to predict QoS values via limited historical QoS logs. Based on historical QoS logs, we generate a user-service matrix, and employ our proposed location-based hierarchical

matrix factorization to predict the missing values in the matrix. Then, based on the predicted QoS values, developers can choose the most suitable Web services easily. Extensive experiments have been conducted, and the results show that our method outperforms existing QoS prediction methods.

In summary, we design novel techniques to automatically process and analyze system runtime data, especially system logs. Specifically, our proposed technique aims at effective and efficient system reliability management, including an evaluation study on log parsers, a parallel log parser, an online log parser, an operational issue prioritization method via log sequences, a location-based QoS prediction method via QoS logs. Moreover, for ease of reproducing our research results and to promote future research on related topics, all implemented methods have been released publicly for reuse in future.

8.2 Future Work

System reliability management via automatic runtime data analysis has been widely studied in recent years, and it is a promising research topic. Although we have proposed a number of novel techniques that advance the state-of-the-art solutions, there are still many interesting research directions which are considered as future work.

What to Log

Logging statements are written by developers to enhance system reliability. In this thesis, we propose a number of techniques to assist developers in automated log analysis, including log parsing, operational issues prioritization, and QoS prediction. However, writing logging statements of high quality is also an important research field in the first place. A logging statement mainly prints two

parts: logging text and runtime variables. Logging text describes the runtime system behaviors, which will be employed by both developers and automated log analysis algorithms. Elaborate logging text can largely facilitate the reliability management process, while immature text may mislead the developers and further slows down the whole process.

Thus, Logging text writing is an important problem. However, it is challenging because currently there is no rigorous specification, and developers mainly design logging text based on domain knowledge. Besides, modern systems have become large-scale and more complex. The volume of generated runtime data, especially system logs, increases rapidly. Logging text needs to be compact.

To address this issue, we plan to explore automated logging text generation techniques. Logging text usually describes system operations, which can be summarized with the understanding of the coding block. Thus, we will manually inspect a number of coding blocks containing logging statements. Then we will try to design a specialized sequence-to-sequence model that is trained by coding blocks extracted from existing open-source projects. Finally, the trained model can be employed in development to automatically generate logging text for developers.

Distributed and Parameter-Free Online Log Parsing

Existing log parsing methods mainly have two limitations. Firstly, current online log parsing methods are designed for single machines, such as Drain proposed in this thesis. This limitation makes existing online log parsers inefficient when parsing large-scale logs for modern systems. Secondly, existing online log parsers require developers to tune model parameters, such as the maximum distance between two log messages in the same group. However, a robust online log parser is expected to dynamically re-tune its parameters.

Designing a distributed and parameter-free online log parser is challenging. A practical distributed online parser needs to run

normally at each node in the system, and it should synchronize with all the parsers located in different nodes. Besides, existing methods rely on developers to manually tune the parameters based on their domain knowledge, because it is difficult to use simple and rigorous rules that can handle various types of system logs.

In our future work, we will maintain a tree, similar to the fixed depth tree in Drain, to guide the online parsing processing in each node. Then, we will explore a method to coordinate all the trees and synchronize them accurately and efficiently. Besides, we will study the relationship between the parameters we use and the characteristics of system logs. For example, the value of a parameter may be linearly dependent on the number of log messages in a log group. Finally, we plan to implement the distributed online log parser on top of a popular distributed system, such as Hadoop Distributed File System.

AI Software Performance Prediction

Recently, artificial intelligence (AI) techniques have been widely employed in various research fields, and they are regarded as crucial components in modern systems. Thus, these components based on AI techniques are also known as AI services. More and more industrial companies provide their AI techniques as services online to facilitate their wide adoption, so the number of functionally equivalent AI services grows rapidly. From a developer's perspective, how to select the best AI service becomes an important yet challenging problem.

For traditional service, developers can select the best service according to QoS values (e.g., response time). However, when using AI services, developers aim at employing the AI service that can provide the most accurate results, for example, the highest f-measure for classifiers. Besides, the number of instances (e.g., figures) for AI task is large. Thus, it is difficult to directly predict the performance of an AI service on a specific instance.

To address this problem, we plan to study automated AI service performance prediction technique based on limited historical AI service invocation history. Specifically, for each AI service invocation, we will generate a specially designed signature for the instance submitted. Then, we will store the signature and its corresponding AI service performance (e.g., accuracy). When a developer wants to submit her dataset to the AI service, for each instance, we will find the similar historical instances based on their signatures. Then based on the similar instances and their AI service performance, our algorithm will search the most suitable AI service for the instance.

□ **End of chapter.**

Appendix A

List of Publications

1. **Pinjia He**, Jieming Zhu, Shilin He, Jian Li, Michael R. Lyu. *Towards Automated Log Parsing for Large-Scale Log Data Analysis*. IEEE Transactions on Dependable and Secure Computing (TDSC), accepted.

2. **Pinjia He**. *An End-to-end Log Management Framework for Distributed Systems*. The 36th International Symposium on Reliable Distributed Systems (SRDS), 2017.

3. Jieming Zhu, **Pinjia He**, Zibin Zheng, Michael R. Lyu. *Online QoS Prediction for Runtime Service Adaptation via Adaptive Matrix Factorization*. IEEE Transactions on Parallel and Distributed Systems (TPDS), Volume 28, Issue 10, 2017.

4. Jieming Zhu, **Pinjia He**, Zibin Zheng, Michael R. Lyu. *CARP: Context-Aware Reliability Prediction of Black-Box Web Services*. The 24th International Conference on Web Service (ICWS), 2017.

5. **Pinjia He**, Jieming Zhu, Zibin Zheng, Michael R. Lyu. *Drain: An Online Log Parsing Approach with Fixed Depth Tree*. The 24th International Conference on Web Service (ICWS), 2017.

6. Jian Li, **Pinjia He**, Jieming Zhu, Michael R. Lyu. *Software Defect Prediction via Convolutional Neural Network*. The

International Conference on Software Quality, Reliability and Security (QRS), 2017.

7. Shilin He, Jieming Zhu, **Pinjia He** Michael R. Lyu. *Experience Report: System Log Analysis for Anomaly Detection Factorization*. The 27th International Symposium on Software Reliability Engineering (ISSRE), 2016.

8. **Pinjia He**, Jieming Zhu, Shilin He, Jian Li, Michael R. Lyu. *An Evaluation Study on Log Parsing and Its Use in Log Mining*. The 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2016.

9. Cuiyun Gao, Baoxiang Wang, **Pinjia He**, Jieming Zhu, Yangfan Zhou, Michael R. Lyu. *PAID: Prioritizing App Issues for Developers by Tracking User Reviews Over Versions*. The 26th International Symposium on Software Reliability Engineering (ISSRE), 2015.

10. Jieming Zhu, **Pinjia He**, Qiang Fu, Hongyu Zhang, Michael R. Lyu, Dongmei Zhang. *Learning to Log: Helping Developers Make Informed Logging Decisions*. The 37th International Conference on Software Engineering (ICSE), 2015.

11. Jieming Zhu, **Pinjia He**, Zibin Zheng, Michael R. Lyu. *A Privacy-Preserving QoS Prediction Framework for Web Service Recommendation*. The 22nd International Conference on Web Service (ICWS), 2015.

12. **Pinjia He**, Jieming Zhu, Zibin Zheng, Jianlong Xu, Michael R. Lyu. *Location-Based Hierarchical Matrix Factorization for Web Service Recommendation*. The 21st International Conference on Web Service (ICWS), 2014.

13. Jieming Zhu, **Pinjia He**, Zibin Zheng, Michael R. Lyu. *Towards Online, Accurate, and Scalable QoS Prediction for Run-*

time Service Adaptation. The 34th International Conference on Distributed Computing Systems (ICDCS), 2014.

14. Tong Zhao, Junjie Hu, **Pinjia He**, Hang Fan, Michael R. Lyu, Irwin King. *Exploiting Homophily-based Implicit Social Network to Improve Recommendation Performance*. The International Joint Conference on Neural Networks (IJCNN), 2014.

15. **Pinjia He**, Jieming Zhu, Jianlong Xu, Michael R. Lyu. *A Hierarchical Matrix Factorization Approach for Location-Based Web Service QoS Prediction*. The International Workshop on Internet-based Virtual Computing Environment (iVCE), 2014.

□ **End of chapter.**

Bibliography

[1] Amazon ec2 https://aws.amazon.com/tw/ec2/.

[2] Apache hadoop http://hadoop.apache.org/.

[3] Drain source code http://appsrv.cse.cuhk.edu.hk/ pjhe/drain.py.

[4] Google cloud https://cloud.google.com/.

[5] https://venturebeat.com/2017/02/28/aws-is-investigating-s3-issues-affecting-quora-slack-trello/.

[6] Microsoft azure https://azure.microsoft.com/.

[7] Slct - simple logfile clustering tool. http://ristov.github.io/slct/.

[8] What is the best log analysis tool that you used? http://stackoverflow.com/questions/154982/what-is-the-best-log-analysis-tool-that-you-used, 2008.

[9] Evaluation of clustering http://nlp.stanford.edu/ir-book/html/htmledition/evaluation-of-clustering-1.html, 2009.

[10] Is there a log file analyzer for log4j files? http://stackoverflow.com/questions/2590251/is-there-a-log-file-analyzer-for-log4j-files, 2010.

[11] Apache spark (http://spark.apache.org/), 2012.

[12] Facebook loses $24,420 a minute during outages (http://algerian–news.blogspot.hk/2014/10/facebook-loses-24420-minute-during.html), 2014.

[13] The cost of downtime at the world's biggest online retailer (https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer), 2016.

[14] https://github.com/logpai/logparser, 2017.

[15] https://spark.apache.org/docs/latest/configuration.html, 2017.

[16] http://www.edupristine.com/blog/hadoop-ecosystem-and-components, 2017.

[17] Kibana. http://kibana.org, 2017.

[18] Logstash. http://logstash.net, 2017.

[19] Splunk. http://www.splunk.com, 2017.

[20] Towards automated log parsing for large-scale log data analysis (supplementary report), 2017.

[21] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient QoS-aware service composition. In *Proc. 18th Int'l Conf. World Wide Web (WWW'09)*, pages 881–890. ACM, 2009.

[22] S. Banerjee, H. Srikanth, and B. Cukic. Log-based reliability analysis of software as a service (saas). In *ISSRE'10*.

[23] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE'11*, 2011.

[24] D. Borthakur. Hdfs architecture guide. *Hadoop Project Website*, 2008.

[25] S. Chen, Y. Liu, I. Gorton, and A. Liu. Performance prediction of component-based applications. *JSS'05: Journal of Systems and Software*, 74(1):35–43, 2005.

[26] X. Chen, Z. Zheng, X. Liu, Z. Huang, and H. Sun. Personalized qos-aware web service recommendation and visualization. *IEEE Transactions on Services Computing*, 6(1):35–47, 2013.

[27] H. J. Cheng and A. Kumar. Process mining on noisy logs-can log sanitization help to improve performance? *Decision Support Systems*, 79:138–149, 2015.

[28] J. Cubo, N. Gamez, E. Pimentel, and L. Fuentes. Reconfiguration of service failures in damasco using dynamic software product lines. In *SCC'15: Proc. of the 12nd International Conference on Services Computing*, pages 114–121, 2015.

[29] C. Di Martino, M. Cinque, and D. Cotroneo. Assessing time coalescence techniques for the analysis of supercomputer logs. In *DSN'12*, 2012.

[30] R. Ding, H. Zhou, J. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *ATC'15: Proc. of the USENIX Annual Technical Conference*, 2015.

[31] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *ICDM'16 Proc. of the 16th International Conference on Data Mining*, 2016.

[32] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu. Everything as a service (xaas) on the cloud: origins, current and future trends. In *CLOUD'15: Proc. of the 8th International Conference on Cloud Computing*, pages 621–628, 2015.

[33] J. El Hadad, M. Manouvrier, and M. Rukoz. TQoS: Transactional and QoS-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing*, 3(1):73–85, 2010.

[34] B. S. Everitt, S. Landau, M. Leese, and D. Stahl. *Cluster Analysis*. Wiley, 5 edition, 2011.

[35] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM'09: Proc. of International Conference on Data Mining*, 2009.

[36] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 18:54–64, 1969.

[37] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *DSN'15*, 2015.

[38] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen. Logmine: Fast pattern recognition for log analytics. In *CIKM'16 Proc. of the 25th ACM International Conference on Information and Knowledge Management*, 2016.

[39] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *DSN'16: Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[40] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing*, 2017, accepted.

[41] P. He, J. Zhu, Z. Zheng, and M. R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *ICWS'17: Proc. of the 24th International Conference on Web Services*, 2017.

[42] S. He, J. Zhu, P. He, and M. Lyu. Experience report: System log analysis for anomaly detection. In *ISSRE'16: Proc. of the 27th International Symposium on Software Reliability Engineering*, 2016.

[43] Y. Hong, J. Vaidya, H. Lu, P. Karras, and S. Goel. Collaborative search log sanitization: Toward differential privacy and boosted utility. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 12:504–518, 2015.

[44] A. F. Huang, C.-W. Lan, and S. J. Yang. An optimal QoS-based web service selection scheme. *Information Sciences*, 179(19):3309–3322, 2009.

[45] L. Huang, X. Ke, K. Wong, and S. Mankovskii. Symptom-based problem determination using log data abstraction. In *CASCON'10 Proc. of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 313–326, 2010.

[46] S. Y. Hwang, W. P. Liao, and C. H. Lee. Web services selection in support of reliable web service choreography. In *ICWS'10: Proc. of the 17th International Conference on Web Services*, pages 115–122, 2010.

[47] Y. Jiang, C. Perng, and T. Li. Meta: Multi-resolution framework for event summarization. In *SDM'14: Proc. of the SIAM International Conference on Data Mining*, pages 605–613, 2014.

[48] Z. Jiang, A. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events.

Journal of Software Maintenance and Evolution: Research and Practice - Special Issue on Program Comprehension through Dynamic Analysis (PCODA), 20:249–267, 2008.

[49] R. Jurca, B. Faltings, and W. Binder. Reliable qos monitoring based on client feedback. In *WWW'07: Proc. of the 16th International Conference on World Wide Web*, pages 1003–1012, 2007.

[50] A. Kattepur, N. Georgantas, and V. Issarny. QoS composition and analysis in reconfigurable web services choreographies. In *Proc. 20th Int'l Conf. Web Services (ICWS'13)*, pages 235–242. IEEE, 2013.

[51] K. KC and X. Gu. Elt: Efficient log-based troubleshooting system for cloud computing infrastructures. In *SRDS'11 Proc. of the 30th IEEE International Symposium on Reliable Distributed Systems*, 2011.

[52] E. F. Krause. *Taxicab Geometry*. Dover Publications, Revised edition, 1987.

[53] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM'04: Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 219–230, 2004.

[54] D. Lang. Using SEC. *USENIX ;login: Magazine*, 38:38–43, 2013.

[55] J. Li, J. Cheng, Y. Zhao, F. Yang, Y. Huang, H. Chen, and R. Zhao. A comparison of general-purpose distributed systems for data processing. In *International Conference on Big Data, BigData*, 2016.

[56] Q. Lin, H. Zhang, J. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*, 2016.

[57] L. A. N. S. LLC. Operational data to support and enable computer science research, 2007.

[58] W. Lo, J. Yin, S. Deng, Y. Li, and Z. Wu. Collaborative web service QoS prediction with location-based regularization. In *Proc. 19th Int'l Conf. Web Services (ICWS'12)*, pages 464–471. IEEE, 2012.

[59] W. Lo, J. Yin, S. Deng, Y. Li, and Z. Wu. An extended matrix factorization approach for QoS prediction in service selection. In *Proc. 9th Int'l Conf. Services Computing (SCC'12)*, pages 162–169. IEEE, 2012.

[60] J. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *ATC'10: Proc. of the USENIX Annual Technical Conference*, 2010.

[61] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25, 1978.

[62] A. Makanju, A. Zincir-Heywood, and E. Milios. Clustering event logs using iterative partitioning. In *KDD'09: Proc. of International Conference on Knowledge Discovery and Data Mining*, 2009.

[63] A. Makanju, A. Zincir-Heywood, and E. Milios. Fast entropy based alert detection in super computer logs. In *DSN-W'10: Proc. of International Conference on Dependable Systems and Networks Workshops*, pages 52–58, 2010.

[64] A. Makanju, A. Zincir-Heywood, and E. Milios. A lightweight algorithm for message type extraction in system

application logs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24:1921–1936, 2012.

[65] A. Makanju, A. Zincir-Heywood, and E. Milios. Investigating event log analysis with minimum apriori information. In *IM'13: Prof. of International Symposium on Integrated Network Management*, pages 962–968, 2013.

[66] C. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[67] S. Meng, Z. Zhou, T. Huang, D. Li, S. Wang, F. Fei, W. Wang, and W. Dou. A temporal-aware hybrid collaborative recommendation method for cloud service. In *ICWS'16: Proc. of the 23rd International Conference on Web Services*, pages 252–259, 2016.

[68] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24:1245–1255, 2013.

[69] M. Mizutani. Incremental mining of system log format. In *SCC'13: Proc. of the 10th International Conference on Services Computing*, pages 595–602, 2013.

[70] A. Mnih and R. Salakhutdinov. Probabilistic matrix factorization. In *Advances in neural information processing systems*, pages 1257–1264, 2007.

[71] H. R. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, and F. Casati. Deriving protocol models from imperfect service conversation logs. *TKDE'08: IEEE Transactions on Knowledge and Data Engineering*, 20(12):1683–1698, 2008.

[72] K. Nagaraj, C. Killian, and J. Neville. structured comparative analysis of systems logs to diagnose performance problems.

In *NSDI'12: Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[73] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN'07*, 2007.

[74] A. Oprea, Z. Li, T. Yen, S. Chin, and S. Alrwais. Dectection of early-stage enterprise infection by mining large-scale log data. In *DSN'15*, 2015.

[75] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 8:640–655, 2011.

[76] N. Poggi, V. Muthusamy, D. Carrera, and R. Khalaf. Business process mining from e-commerce web logs. In *Business Process Management*, pages 65–80. 2013.

[77] H. Ringberg, A. Soule, J. Rexford, and C. Diot. Sensitivity of pca for traffic anomaly detection. In *SIGMETRICS'07: Proc. of International Conference on Measurement and Modeling of Computer Systems*, 2007.

[78] S. Ryza. How-to: Tune your apache spark jobs (part 2). https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/, 2015.

[79] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive logfiles for autonomic systems. In *IPDPS'04: Proc. of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[80] G. Salton and C. Buckley. Term weighting approaches in automatic text retrival. Technical report, Cornell, 1987.

[81] W. Shang, Z. Jiang, H. Hemmati, B. Adams, A. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *ICSE'13: Proc. of the 35th International Conference on Software Engineering*, pages 402–411, 2013.

[82] L. Shao, J. Zhang, Y. Wei, J. Zhao, B. Xie, and H. Mei. Personalized QoS prediction forweb services via collaborative filtering. In *Proc. 14th Int'l Conf. Web Services (ICWS'07)*, pages 439–446. IEEE, 2007.

[83] M. Silic, G. Delac, and S. Srbljic. Prediction of atomic web services reliability based on k-means clustering. In *Proc. 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 70–80. ACM, 2013.

[84] Y. Sun, H. Li, I. G. Councill, J. Huang, W. C. Lee, and C. L. Giles. Personalized ranking for digital libraries based on log analysis. In *WIDM'08: Proc. of the 10th ACM workshop on Web information and data management*, pages 133–140, 2008.

[85] L. Tang, T. Li, and C. Perng. LogSig: generating system events from raw textual logs. In *CIKM'11: Proc. of ACM International Conference on Information and Knowledge Management*, 2011.

[86] L. Tang, T. Li, L. Shang, F. Pinel, and G. Grabarnik. An integrated framework for optimizing automatic monitoring systems in large it infrastructures. In *KDD'13: Proc. of International Conference on Knowledge Discovery and Data Mining*, pages 1249–1257, 2013.

[87] M. Tang, Y. Jiang, J. Liu, and X. Liu. Location-aware collaborative filtering for QoS-based service recommendation. In

Proc. 19th Int'l Conf. Web Services (ICWS'12), pages 202–209. IEEE, 2012.

[88] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IPOM'03: Proc. of the 3rd Workshop on IP Operations and Management*, 2003.

[89] R. Vaarandi. Mining event logs with slct and loghound. In *NOMS'08: Proc. of the IEEE/IFIP Network Operations and Management Symposium*, 2008.

[90] R. Vaarandi and K. Podis. Network ids alert classification with frequent itemset mining and data clustering. In *CNSM'10: Proc. of the Conference on Network and Service Management*, 2010.

[91] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an rbf neural network. *TR'12: IEEE Transactions on Reliability*, 2012.

[92] Q. Wu, A. Iyengar, R. Subramanian, I. Rouvellou, I. Silva-Lepe, and T. Mikalsen. Combining quality of service and social information for ranking services. In *Proc. 7th Int'l Conf. Service Oriented Computing (ICSOC'09)*, pages 561–575. Springer, 2009.

[93] W. Xu. *System Problem Detection by Mining Console Logs*. PhD thesis, University of California, Berkeley, 2010.

[94] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordon. Detecting large-scale system problems by mining console logs. In *SOSP'09: Proc. of the ACM Symposium on Operating Systems Principles*, 2009.

[95] J. Yao, S. Chen, C. Wang, D. Levy, and J. Zic. Modelling collaborative services for business and qos compliance. In

ICWS'11: Proc. of the 18th International Conference on Web Services, pages 299–306, 2011.

[96] L. Yao, Q. Z. Sheng, A. Segev, and J. Yu. Recommending web services via combining collaborative filtering with content-based features. In *Proc. 20th Int'l Conf. Web Services (ICWS'13)*, pages 42–49. IEEE, 2013.

[97] X. Yu, M. Li, I. Paik, and K. H. Ryu. Prediction of web user behavior by discovering temporal relational rules from web log data. In *DEXA'12: Proc. of the 23rd International Conference on Database and Expert Systems Applications*, pages 31–38, 2012.

[98] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *OSDI'12: Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 293–306, 2012.

[99] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12: Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[100] S. Zawoad, A. Dutta, and R. Hasan. Towards building forensics enabled cloud through secure logging-as-a-service. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 13:148–162, 2016.

[101] J. Zhang, B. Iannucci, M. Hennessy, K. Gopal, S. Xiao, S. Kumar, D. Pfeffer, B. Aljedia, Y. Ren, M. Griss, S. Rosenberg, J. Cao, and A. Rowe. Sensor data as a service–a federated platform for mobile data-centric service development

and sharing. In *SCC'13: Proc. of the 10th International Conference on Services Computing*, 2013.

[102] L.-J. Zhang, J. Zhang, and H. Cai. Services computing. In *Springer and Tsinghua University Press*, 2007.

[103] Z. Zheng, H. Ma, M. R. Lyu, and I. King. WSRec: A collaborative filtering based web service recommender system. In *Proc. 16th Int'l Conf. Web Services (ICWS'09)*, pages 437–444. IEEE, 2009.

[104] Z. Zheng, H. Ma, M. R. Lyu, and I. King. Qos-aware web service recommendation by collaborative filtering. *IEEE Transactions on Services Computing*, 4(2):140–152, 2011.

[105] Z. Zheng, Y. Zhang, and M. R. Lyu. Distributed QoS evaluation for real-world web services. In *Proc. 17th Int'l Conf. Web Services (ICWS'10)*, pages 83–90. IEEE, 2010.

[106] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *ICSE'15*, 2015.

[107] J. Zhu, P. He, Z. Zheng, and M. R. Lyu. Towards online, accurate, and scalable qos prediction for runtime service adaptation. In *Proc. 34th Int'l Conf. Distributed Computing Systems (ICDCS'14)*. IEEE, 2014.

[108] D. Q. Zou, H. Qin, and H. Jin. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of Computer Science and Technology*, 31(5):1038–1052, 2016.