TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Science

Centre for Digital Forensics and Cyber Security

C-BASED IMPLEMENTATION OF LOGCLUSTER, A DATA CLUSTERING AND PATTERN MINING ALGORITHM FOR EVENT LOGS

Master's thesis

ITC70LT

Author: Chen Zhuge

Student code: 132116IVCMM

Supervisor: Risto Vaarandi, Ph.D

Tallinn 2016

Declaration

I hereby declare that I am the sole author of this thesis. The work is original and has not been submitted for any degree or diploma at any other university. I further declare that the materials obtained from other sources have been duly acknowledged in the thesis.

....
(date)

....
(signature)

Abstract

This thesis studies data mining algorithms for mining patterns from event logs, and focuses on clustering algorithms that mine line patterns from textual log files. The main contribution of this thesis is the development of a C-based implementation of the LogCluster algorithm which is publicly available under GNU GPLv2 license [41]. Another contribution of this thesis is a performance assessment of this implementation through number of benchmarks and comparison with other similar data mining tools.

LogCluster is a novel data mining algorithm introduced by Risto Vaarandi and Mauno Pihelgas in 2015 [2]. Compared to its predecessor Simple Logfile Clustering Tool (SLCT), LogCluster produces more meaningful clusters, but consumes more computing resource.

Since the prototype implementation of LogCluster was written in Perl, Risto Vaarandi and Mauno Pihelgas concluded that LogCluster still needs to be implemented in C for achieving the best performance. Also, the paper that introduced LogCluster provides a performance comparison of the Perl-based implementations of LogCluster and SLCT, while detailed performance assessment of the C-based implementations has been identified as future work [2]. The current thesis addresses this open research issue by describing a set of experiments for assessing the performance of the C-based implementations.

This thesis demonstrates that the C-based implementation of LogCluster realizes all the functions in the Perl-based prototype implementation (version 0.03) and produces the same output, with significantly reduced processing time and the same level of memory consumption. Regarding the performance assessment of the C-based implementations, this thesis demonstrates that LogCluster is not slower than SLCT, while it consumes much more memory than SLCT because of its algorithmic complexity.

From a practical perspective, system administrators who use LogCluster to analyze event logs would benefit from this efficient C-based implementation.

Table of abbreviations and terms

GNU   recursive acronym for GNU's Not Unix

GNU GPLv2   GNU General Public License version 2

APT   Advanced Persistent Threat

IT   Information Technology

SLCT   Simple Logfile Clustering Tool

FP-Growth   Frequent Pattern Growth

BSD   Berkeley Software Distribution

CLIQUE   CLustering In QUEst

MAFIA   MAximal Frequent Itemset Algorithm

CACTUS   CAtegorical ClusTering Using Summaries

PROCLUS   PROjected CLUStering

IDS   Intrusion Detection System

IBM   International Business Machines

IPLoM   Iterative Partitioning Log Mining

PCA   Principal Component Analysis

OPTICS   Ordering Points To Identify the Clustering Structure

HLAer   Heterogeneous Log Analyzer

DBSCAN   Density Based Spatial Clustering of Applications with Noise

CFDR   the Computer Failure Data Repository

CPU	Central Processing Unit
DDR3	Double Data Rate type 3
RAM	Random Access Memory
rpm	revolutions per minute
LCC	LogClusterC
LCP	LogClusterP
I/O speed	Input/Output speed
SSD	Solid-State Drive

Table of contents

List of figures

List of tables

1.   Introduction

Dealing with system event logs is an important part in system administrators' daily work. For the detection of critical events, system administrators have real-time log file monitoring tools at their disposal, which instantly analyze event logs and notify system administrators when necessary.

The service quality of real-time log file monitoring tools relies heavily on the configuration, which reflects system administrators' knowledge about the system. For example, to detect system faults, the tools continuously query a database that contains system fault patterns, which means only the faults within system administrators' knowledge scope can be detected. To detect previously unknown faults that are not in the database, system anomaly detection approach can be used, that the tools continuously query a system profile that contains normal system activity patterns. However, maintenance of a sufficient system profile is not only time-consuming, but also requiring exhaustive knowledge about the system [1].

Questions like what are the most frequently occurring event types, and what are the fewest (most abnormal) event types, often need quick answers. In order to get a whole picture of the system behaviors during a certain time period in the past, system administrators need offline log analysis as a method of knowledge acquisition.

Acting as a powerful supplement, offline log analysis discovers knowledge for real-time log file monitoring tools [4, 6-15, 18, 22]. For the services that require high level of security or face Advanced Persistent Threat (APT), regularly extracting knowledge via offline log analysis is especially indispensable. Offline log analysis can also be used in other fields such like event pattern visualization [16, 17].

Because modern IT systems produce large volumes of event logs which are difficult and laborious for system administrators to manually analyze, the use of different data mining algorithms has been suggested for easing this task since the 1990s.

From different data mining algorithms, data clustering methods have been often chosen to discover patterns in log files [1-5, 11-14, 16, 17, 22-25, 27]. Recently developed data

clustering algorithms assume that each event log line represents some event. The algorithms divide the event log lines into clusters, so that events in each cluster are similar to each other and match the same line pattern. Instead of reporting individual events in each cluster, the algorithms report line patterns that are the concise descriptions of the events in detected clusters. Unusual events which do not fit into any of the clusters (in other words, do not match any of the detected line patterns) are called "outliers".

In 2015, Risto Vaarandi and Mauno Pihelgas [2] introduced LogCluster, a novel event log data mining algorithm using density-based clustering approach to discover event log patterns. The comparative experiments in [2] demonstrated that LogCluster produces more meaningful clusters than its predecessor SLCT. As LogCluster allows to address many event log pattern discovery tasks, it was concluded that the prototype implementation of LogCluster is fully capable of being used in industrial environments as a productive tool.

Built for academic experimentation purposes, and for the sake of simplicity and portability, the prototype implementation of LogCluster was written in Perl. Therefore, in the conclusion section of [2], the authors stated that a C-based implementation of LogCluster should be considered as future work. In order to fully realize the algorithm's potential as a productive solution for system administration, the C-based implementation should produce the same output as the Perl-based prototype implementation, but with reduced computing resource.

For the sake of convenience, in the rest of this thesis we refer to the Perl-based prototype implementation of LogCluster as "LogClusterP", and the C-based implementation of LogCluster as "LogClusterC".

1.1. Problem statement and contribution of the thesis

The first contribution of this thesis is the coding of LogClusterC, addressing the problem that LogClusterP has regarding computing efficiency. According to the experiment results in [2], conducting data mining for a 3.8 GB log file using

LogClusterP on a mid-range virtual server[1] will require about 52 minutes, which leaves space for further optimization. The experiment results in Chapter 4 demonstrate that LogClusterC significantly optimizes the computing efficiency.

Since LogCluster is an algorithm that is still experimental and actively developed, LogClusterP is frequently updated. At the time of writing this thesis, the latest version is 0.08 with new features such as parsing input and creating word classes with precompiled Perl functions, resuming data mining from previously stored disk-based data structures, etc. Currently, all functions that are supported by LogClusterP (version 0.03) are supported by LogClusterC. We followed the design principle that LogClusterC will generate the same output as LogClusterP, if they are given the same input (same command line).

The second contribution of this thesis is the conduction of a series of performance comparison experiments between LogClusterP and LogClusterC. The experiments demonstrate that LogClusterC is an efficient solution that can be used in industrial environments and is helpful for many system administration tasks.

Additionally, given that the Perl-based implementation of LogCluster is 1.28-1.62 times slower than the Perl-based implementation of SLCT [2], this paper demonstrates that there is no difference between the C-based implementations of the two algorithms regarding processing time. However, LogClusterC consumes larger amounts of memory than SLCT[2].

This thesis verifies the validity of the approach through empirical experiments. The experiments were designed to satisfy the following conditions [30]:

- Repeatability – Tests can be run again when needed;
- Reproducibility – The tools used in the experiments (LogClusterC [41], LogClusterP [42] and SLCT [43]) are publicly available under GNU GPLv2 license. Log files in the experiments are selected from public data repository

---

[1] A Linux virtual server with Intel Xeon E5-2680 CPU and 64GB of memory.

[2] As the original implementation of SLCT algorithm in 2003 was written in C, and the Perl-based version of SLCT is no longer involved in the rest part of this thesis, the term "SLCT" means either the algorithm or the C-based implementation depending on the context. This convention applies to the rest of this thesis.

[39]. Command lines and the bash scripts used in the experiments are provided in [41].

- Rigor – Hardware configuration in the experiments is provided in Chapter 4. Experiment data is accurate and obtained using the time utility of Linux [44]. The raw plain text outputs in the experiments are publicly available in [41].

1.2. Outline

Chapter 1 provides information regarding the problem background.

Chapter 2 presents an overview of existing event log data mining algorithms and previous academic research.

Chapter 3 describes the implementation details of LogClusterC.

In chapter 4, a series of comparative experiments are conducted with LogClusterC, LogClusterP and SLCT.

Chapter 5 concludes this thesis and suggests future work.

1.3. Acknowledgments

The author would at this point like to expand his gratitude to his family members for supporting him. To Tallinn University of Technology and Cyber Security programme for offering such a perfect platform to study and gain practical skills. To Risto Vaarandi and Mauno Pihelgas, for creating LogCluster algorithm, upon which this thesis is conducted. Last but by no means least to Risto Vaarandi, for sharing experiences, offering ideas and opinions, providing source code, testing LogClusterC and mentoring the thesis.

2. Overview of existing solutions and related work

The study of data mining techniques for event logs started as early as the 1990s, when IT systems started to dramatically expand and produce large volumes of event logs that system administrators were no longer able to manually analyze.

As data mining techniques for event logs developed, many solutions were suggested by researchers. Depending on the algorithms they use, these solutions can be classified into two different approaches: association rule approach and data clustering approach.

2.1. Association rule approach

Association rule approach is suggested by many papers [6-10, 15, 19-21, 28, 29, 33] and widely used in the telecommunication industry.

Association rule algorithms are designed for the detection of correlated patterns of event types. For example, the following association rule could be identified: event type "link down" is followed by event type "link up" within 10 seconds with the probability of 95%, which represents the link short outage phenomenon. Therefore, association rule approach is inherently suitable for data mining tasks that aim to get knowledge for writing usable event correlation rules.

Association rule approaches usually employ frequent itemset mining algorithms (such as Apriori and FP-Growth) for detecting associations. Apriori is the most commonly used algorithm which harnesses iterative approach for detecting frequent itemsets [26]. At the *k-th* step of the algorithm, a pass over input data is made and itemsets with *k* elements are identified. When the algorithm completes, detected itemsets represent frequently occurring event type patterns, and association rules are derived from these patterns. However, if the largest itemset contains *N* elements, Apriori makes *N* passes over input data set, which involves considerable runtime overhead when event log contains longer event type patterns.

Although powerful, association rule approach is not directly applicable to many log files, since this approach relies heavily on the good quality of event logs. Event type and event occurrence time are the two required event attributes in this approach. In practice,

as there is no universal standard for event log format, event log messages do not consistently have event type. The BSD syslog protocol which is a widely accepted event log standard, also does not define the event type attribute [37].

To derive event types from raw event log data, many different solutions have been suggested, such as using simple heuristic techniques to replace specific strings with generic ones [28, 29], analyzing original application's source code [14], and using data clustering approach.

The line patterns found by using data clustering approach can represent event types. For example, if a data clustering algorithm discovers the line pattern "*sshd: Failed password for * from * port * ssh2*", this line pattern represents the event type "*SSH Login Failure with Wrong Password*".[1] It is worth mentioning that data clustering approach can also be further used to cluster the rule sets that are generated by association rule approach [6, 20].

2.2. Data clustering approach

Although data clustering approach is described as a preprocessing or postprocessing step of association rule approach in the previous section, it is widely and independently applied in industry as well. During the last two decades, many papers [1-5, 11-14, 16, 17, 22-25, 27] have suggested various data clustering methods for event logs.

Data clustering is a process of dividing a set of data points into multiple subsets (clusters), making each subset contain the points that share a certain level of similarity. In the end of the clustering process, usually a special subset called "cluster of outliers" is generated, which contains the data points that do not belong to the other existing subsets.

Compared to association rule approach, data clustering approach has its own advantages. In particular, data clustering algorithms can be often applied for knowledge discovery from regular log files with no (or minimal) preprocessing of the log data. Also, data clustering approach usually detects line patterns from event logs, and this output is very

_____

[1] In the terminology of data mining for event logs, line pattern and event type have different meanings. This example demonstrates the relationship between them.

valuable for designing regular expression signatures, constructing event correlation rules, building system profiles, and detecting system anomalies.

When applying data clustering for event logs, however, the situation becomes more complex and different from traditional data clustering.

In the case of traditional data clustering such as a product similarity research in supermarket, each point (product) has a constant number of attributes, e.g., price, amount in stock, and production date, and each attribute has a numeric value. Since data points have numerical attributes, clustering algorithms can use distance functions (such as Manhattan or Euclidean distance) for measuring similarity between any pair of data points.

In the case of data clustering for event logs, it is no longer reasonable to apply the distance function based algorithms. Unlike the traditional data clustering, each point (event) has a variable number of attributes (each event log line has a variable number of words), and the attributes (words) are categorical. Also, points (events) can often have a large number of attributes (words) that make event log data high-dimensional. Unfortunately, the clustering of high-dimensional data sets is known as a complex problem [1].

To provide solutions for this particular problem, many researchers have suggested different algorithms, and the rest of this section will review the most common algorithms.

2.2.1. Simple Logfile Clustering Tool (SLCT)

Although several algorithms like CLIQUE, MAFIA, CACTUS and PROCLUS exist for clustering generic high-dimensional categorical data sets, they are not suitable for clustering large event logs. Their drawbacks include:

- Many of the high-dimensional data clustering algorithms such as CLIQUE and MAFIA are variants of Apriori, which are inefficient for mining longer patterns [5].
- Compared to detecting frequent patterns, the detection of infrequent patterns is also important [21], which is however not addressed by the Apriori-based data clustering algorithms [5].

- Some algorithms detect clusters which do not represent valid event patterns.

Because the aforementioned data clustering algorithms are not applicable for clustering event logs, they promoted the emergence of SLCT in 2003 [1], a density-based data clustering algorithm specifically designed for event logs.

SLCT addresses the problems of previous data clustering algorithms and focuses on finding line patterns. SLCT is faster than Apriori-based algorithms, and is able to discover both frequent line patterns and infrequent events.

Like the CACTUS algorithm, SLCT consists of three steps. It first makes a pass over the data set to build a vocabulary. Then it makes the second pass over the data set, referring to the vocabulary, to generate cluster candidates. Finally the clusters are selected from the cluster candidates, based on an user-provided support threshold parameter [5].

Each cluster is reported to the user as line pattern, where all lines from the cluster match this pattern. Therefore, the line pattern can be regarded as a concise description of the entire cluster, and in the rest of the thesis we use the terms "line pattern" and "cluster (candidate) description" interchangeably.

The author of SLCT has released a publicly available implementation of SLCT which is written in C, and which is tested during the comparative experiments in this thesis. Besides the primary line pattern mining functions, SLCT also supports several other features, such as regular expression based input filtering and cluster refinement [32].

From the programming perspective, the implementation harnesses three techniques worth mentioning. Firstly, the implementation of a fast string hashing function developed by M. V. Ramakrishna and Justin Zobel [34] has essentially improved the overall processing speed. Secondly, the summary vector technique introduced by Risto Vaarandi [1] significantly reduces the memory consumption, which is vital when processing large size log files. Thirdly, the move-to-front technique in hash table proposed by Justin Zobel et al. [35] is used to optimize the computing performance, taking full advantage of the nature of event log data (most words in event log messages

only occur few times) [1]. The same three techniques are used again in the following algorithms LogHound [3] and LogCluster[1] [2].

SLCT has been used in many log analysis systems, such as Sisyphus log mining toolkit developed at Sandia National Labs [16, 31], event log clusters visualization tool LogView developed by Adetokunbo Makanju et al. [17]. Risto Vaarandi and Kārlis Podiņš [22] deployed SLCT in an unsupervised real time alert classification method for network IDS. Thomas Reidemeister et al. [11-13] extended SLCT algorithm and employed it in a system faults and anomalies detection framework.

In [27], Xia Ning et al. commented SLCT as one of the first event log clustering algorithms, whose design influenced many of the following event log clustering algorithms.

However, SLCT has four shortcomings:

- In the cluster descriptions, the wildcards (infrequent word would be displayed as wildcards in the cluster description) after the last frequent word are not displayed [2, 33]. For instance, cluster description "*Interface * down *"* and "*Interface * down * * *"* are displayed as "*Interface * down*".[2]
- It is sensitive to shifts in word positions, because it considers words with their positions in the event log line [2, 13]. For instance, with "*Interface*" and "*down*" being the frequent words, the line "*Interface eth1 port down*" would not be assigned to the cluster candidate "*Interface * down*", but would rather generate a separate cluster candidate "*Interface * * down*".
- A low support threshold can cause over-fitting problem, that larger clusters are split and line patterns are too specific [2].
- It is inflexible in cluster generation, since it rigidly relies on the frequent word mechanism [27]. Using the case presented by Xia Ning et al. in [27] as the example, "* *node- * node empera ure 1* 1 ambien =30*", "* *node- * node empera ure 1* 1 ambien =29*", and "* *node- * node empera ure 1* 1 ambien

---

[1] In LogCluster, the term "summary vector" is renamed as "sketch".

[2] The cluster refinement('-r' option) of SLCT can fix this problem. However, the cluster refinement is an optional function, which is conducted after the clusters are found, requiring an extra pass over the data set.

=*32*" are generated as three distinct clusters instead of one, despite only the words after word "*ambien*" are different.

2.2.2. LogHound

Introduced by Risto Vaarandi in 2004 [3], LogHound is a tool that was designed for finding frequent patterns from event log data sets with the help of a breadth-first frequent itemset mining algorithm [36].

While SLCT focuses only on mining frequent line patterns, LogHound can be employed for both mining frequent line patterns and mining frequent event type patterns.

The author of LogHound has released a publicly available implementation of LogHound which is written in C. From the programming perspective, it is worth mentioning that the implementation harnesses a prefix tree data structure, which significantly speeds up the log mining process. In LogCluster, a similar prefix tree data structure is applied in the Aggregate_Supports heuristics [2], which is used to address the cluster over-fitting problem of SLCT, as discussed in Section 2.2.1.

Sharing many common features with SLCT, LogHound has the same shortcomings, such as the ignorance of wildcards in the end of cluster descriptions, and being sensitive to shifts in word positions.

2.2.3. Teiresias

Teiresias was developed by IBM in 1997 for the purpose of bioinfomatics pattern discovery, then it was further developed for security anomaly detection [16]. Similar as SLCT and LogHound, it requires a support threshold as user input.

In order to select the suitable algorithm for Sisyphus log mining toolkit [31], Jon Stearley [16] conducted comparative experiments between Teiresias and SLCT, and concluded that Teiresias is an effective pattern discovery engine for syslog messages. However, being limited only to process small data sets due to its high memory consumption was also pointed out as its critical disadvantage.

2.2.4. Iterative Partitioning Log Mining (IPLoM)

IPLoM is a novel log clustering algorithm introduced by Adetokunbo Makanju et al. in 2009 [23-25]. Not based on the Apriori algorithm, it works by creating a hierarchical partitioning of the event log data [23].

Differing from SLCT, LogHound and Teiresias, IPLoM does not require a support threshold as user input parameter. IPLoM is also able to detect tail wildcards in cluster descriptions, which is an advantage over SLCT and LogHound.

According to the comparative experiments conducted in [23, 24], IPLoM outperforms SLCT, Loghound and Teiresias.

However, IPLoM has two shortcomings:

- To maintain a good clustering quality, event log lines with the same number of words shall have the same formats, which is hard to achieve when processing massive event logs [27].
- Same as SLCT and LogHound, IPLoM is sensitive to shifts in word positions, since it considers words with their positions in event log lines [2].

2.2.5. LogCluster

LogCluster is a density-based data clustering algorithm for event logs, introduced by Risto Vaarandi and Mauno Pihelgas in 2015 [2].

LogCluster can be considered as "next generation SLCT", coming with many features that aim to solve existing shortcomings in SLCT. All the four shortcomings aforementioned in Section 2.2.3. are fixed in LogCluster. According to the comparative experiments in [2], LogCluster generates more meaningful line patterns than SLCT.

As this thesis revolves around LogCluster, its algorithmic details will be explained in next chapter by describing the implementation of LogClusterC.

2.2.6. Other data mining algorithms for event logs

Wei Xu et al. proposed a method for mining console logs in [14]. This method combines text mining with source code analysis and log parsing to extract structure

from the console logs, and uses Principal Component Analysis (PCA) to detect anomalous patterns.

Xia Ning et al. implemented Ordering Points To Identify the Clustering Structure (OPTICS) [40], a hierarchical data clustering algorithm, into a heterogeneous log analysis system HLAer [27]. The OPTICS algorithm is based on a density-based data clustering algorithm DBSCAN [38].

3.   The C-based Implementation of LogCluster

In this chapter, details of the C-based implementation of LogCluster will be described. Apart from Section 3.1. and Section 3.2., this chapter is organized in a problem-oriented style, explaining how LogClusterC addresses specific event log data mining problems.

3.1.   General information

Risto Vaarandi and Mauno Pihelgas [2] wrote the prototype implementation of LogCluster in Perl, taking advantage of many Perl's built-in features, such as hash table type and its corresponding operations. As a result, the code of LogClusterP is very compact.

Additionally, LogClusterP is especially easy to deploy. Once the source code file has been downloaded, one can easily run it by utilizing the perl command, which is widely pre-installed in popular Unix environments. For example, to cluster a log file named "sample.log" with a support threshold of 200, the command line is shown below.

```
perl logcluster.pl --input=sample.log --support=200
```

LogClusterC is written in C, and borrows many code blocks from SLCT. Like SLCT, LogClusterC builds hash tables and other data structures from scratch. Because of LogCluster's algorithmic complexity, the code base of LogClusterC is larger than SLCT.

Same as LogClusterP and SLCT, LogClusterC is publicly available under GNU GPLv2 license [41].

Unlike LogClusterP, one has to compile the source code file before running it. Same as SLCT, all source code of LogClusterC resides in one file, making the compiling process convenient. For example, to compile the source code file into an executable file named "logclusterc", the command line is shown below.

```
gcc LogClusterC.c -o logclusterc
```

After compiling, one can use LogClusterC with the same command line syntax as LogClusterP. For example, to cluster a log file named "sample.log" with a support threshold of 200, the command line is shown below.

```
./LogClusterC --input=sample.log --support=200
```

The general differences between LogClusterC, LogClusterP and SLCT are summarized in Table 1.

Table 1. General differences between LogClusterC, LogClusterP and SLCT

	SLCT (version 0.01)	**SLCT** (version 0.05)	**LogClusterP** (version 0.03)	**LogClusterC** (version 0.03)	**LogClusterP** (version 0.08)
Year of release	2003	2007	2015	2016	2016
Language	C	C	Perl	C	Perl
Size of source file	35KB	40KB	37KB	190KB	45KB
Supported options	13	14	19	24	26

3.2. Program workflow

As shown in Figure 1, the program workflow is divided into four phases: "Preparation", "Frequent words", "Cluster candidates", and "Clusters and outliers". Each phase consists of multiple steps.

There are two kinds of special tags in Figure 1. A black dot means that this step is optional which can be enabled by the user. A flag means that this step requires one pass over the data set. When clustering large log files, passing over the data set is the most expensive task that consumes most computing resource.
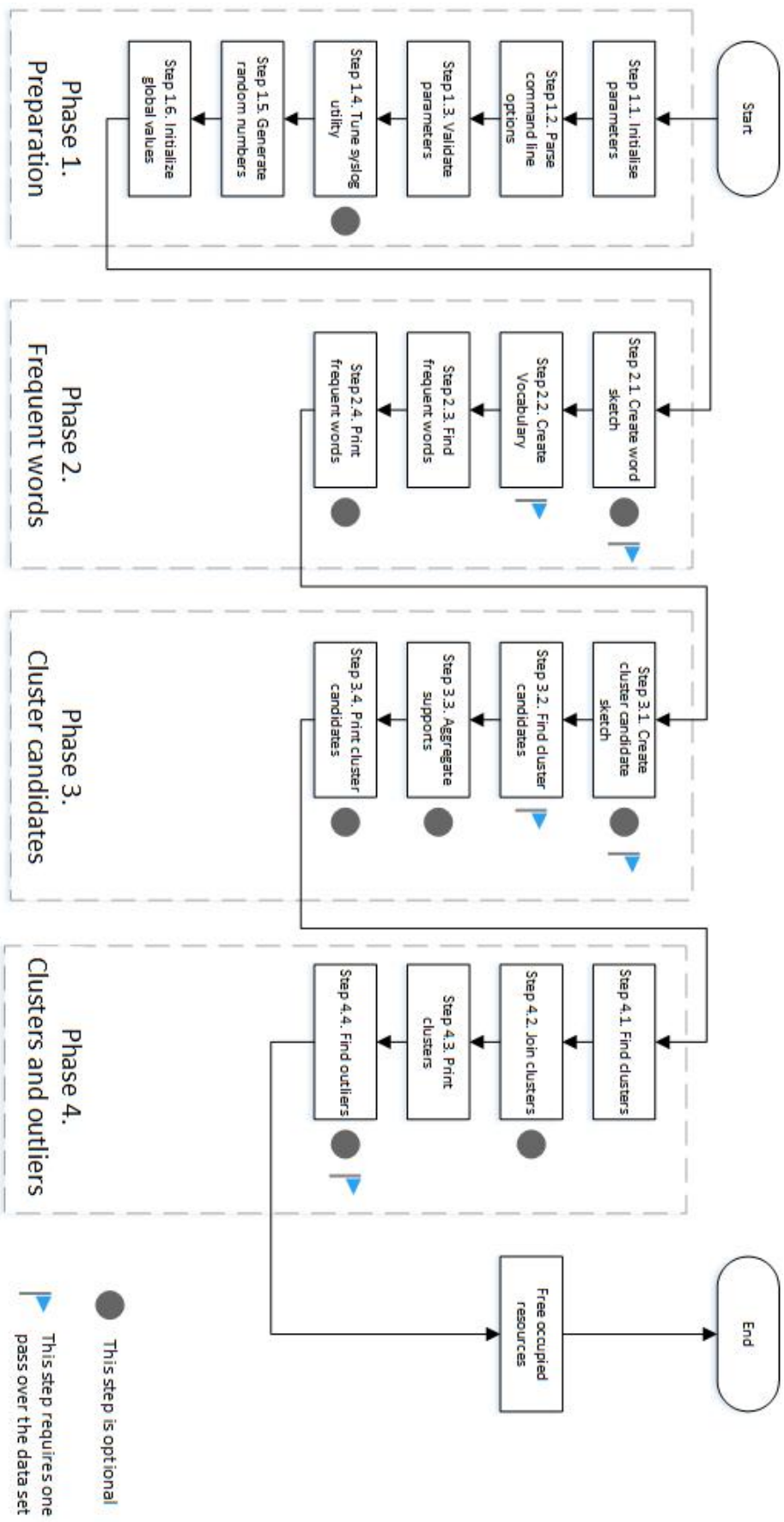
Figure 1. Program workflow

In the simplest case, LogClusterC will pass over the data set twice, where only two parameters need to be set: '--input=<file name>' and '--support=<support threshold>'. We use the following command line as an example to demonstrate the basic workflow of LogClusterC.

```
./LogClusterC --input=sample.log --support=200
```

The first pass over the data set is accomplished in "Step 2.2. Create vocabulary". LogClusterC reads through log file sample.log line by line, splitting each line by delimiter (which is space by default), querying each word from the word table. If the word is already in the word table, its counter will be incremented. Otherwise, the word will be added into the word table (a dedicated memory space will be allocated for the word) and its counter is initially set to one.

In "Step 2.3. Find frequent words", the word table is traversed. If a word's counter is less than the support threshold 200, the word will be removed from the word table and its dedicated memory space will be freed. In other words, this step sanitizes the word table for memory consumption optimization and the convenience of later usage.

The second pass over the data set is accomplished in "Step 3.2. Find cluster candidates". LogClusterC reads through log file sample.log line by line again, splitting each line by delimiter, querying each word from the word table to see whether it is a frequent word. A word will then have one of the two statuses: frequent word (constant) or infrequent word (variable). After each word's status is determined, and if there is at least one frequent word in this event log line, a cluster candidate will be generated for this line. Then LogClusterC queries this cluster candidate from the cluster table. Like with the word table, if the cluster candidate is already in the cluster table, its counter will be incremented. Otherwise, the cluster candidate will be added into the cluster table (a dedicated memory space will be allocated for the cluster candidate), and its counter is initially set to one. The cluster candidate is uniquely identified by a string, which is called "cluster (candidate) identifier", that consists of all the frequent words following their original order in the event log line. For example, with "*Interface*" and "*down*"

being frequent words, the event log line "*Interface eth0 down*" will generate a cluster candidate whose identifier is "*Interface\ndown\n*".[1]

In "Step 4.1. Find clusters", LogClusterC traverses the cluster table. If a cluster candidate's counter is less than the support threshold 200, the cluster candidate will be dropped from the cluster table and its dedicated memory space will be freed. After that, every remaining cluster candidate in the cluster table is recognized as a cluster.

Finally, in "Step 4.3. Print Clusters" LogClusterC traverses the cluster table again, printing every cluster's description and its corresponding counter.

The construction of the word table and the cluster table is based on hash calculation, converting a string (word or cluster candidate identifier) to an integer hash value. For the sake of processing speed optimization, LogClusterC uses the Shift-Add-Xor string hashing algorithm [34] like SLCT does, and constructs hash tables with the move-to-front technique [35]. Details of the move-to-front technique will be described in Section 3.3.

To address SLCT's shortcoming of being sensitive to shifts in word positions, LogClusterC has an essential change in the backbone, that the vocabulary (the word table) is created without consideration of the word's position. That is to say, a word in the word table is purely itself, without a prefix representing its position like in SLCT ("*Interface*" in LogClusterC instead of "*0001Interface*" in SLCT). The word's location is now recorded by a utility named "wildcard recorder", which memorizes the minimum number and maximum number of infrequent words before a frequent word.

The wildcard recorder is utilized in "Step 3.2. Find cluster candidates". In contrast to SLCT's word prefix approach which generates constant cluster candidate descriptions, the wildcard recorder approach generates flexible cluster candidate descriptions, which are kept being adjusted according to new emerging cluster candidates. For example, assume there are two event log lines: "*Interface eth0 down*" and "*Interface HQ Link*

---

[1] The "\n" character (ASCII 10 or newline) is used as a separator between frequent words.

down", and "*Interface*" and "*down*" are frequent words[1]. In SLCT, there will be two distinct cluster candidates generated respectively, whose descriptions are "*Interface * down*" and "*Interface * * down*". On the other hand, in LogClusterC, there will be only one cluster candidate generated, whose description is initially "*Interface *{1,1} down*", which is then adjusted to "*Interface *{1,2} down*" with the help of wildcard recorder, where "*{1,2}" means that the number of infrequent words between "*Interface*" and "*down*" is at least one but at most two.

The wildcard recorder is also able to track the number of infrequent words after the last frequent word, which addresses another shortcoming of SLCT: the ignorance of wildcards in the end of cluster descriptions. For example, with "*Interface*" and "*down*" being the frequent words, if a third event log line "*Interface eth1 down HF*" is processed, the previous cluster candidate description would be adjusted to "*Interface *{1,2} down *{0,1}*".[2]

As the following sections are problem-oriented, the phases and steps in Figure 1 will not be individually described in a linear order. Instead, only the key options and features will be explained and demonstrated. To see the descriptions of all the available options, please use the '--help' option of LogClusterC. In the source code, there are also detailed comments providing more insights into individual phases and steps.

3.3.   Computing optimizations that take advantage of the nature of log file data

By studying the nature of log file data and taking advantage of it, LogClusterC optimizes the clustering process in both processing speed and memory consumption. The hash table's move-to-front technique described in Section 3.3.2. is introduced by Justin Zobel et al. [35]. The sketch preprocessing technique described in Section 3.3.3. is introduced by Risto Vaarandi [1].

_____

[1] Strictly speaking, in this example, there are three frequent words in SLCT: "*0001Interface*", "*0003down*" and "*0004down*", as SLCT considers the word with its position. On the other hand, there are two frequent words in LogClusterC, which are "*Interface*" and "*down*".

[2] For more details of the wildcard recorder, please refer to "int wildcard[MAXWORDS + 1]" in the source code.

3.3.1. The nature of log file data

From the perspective of word frequency statistics, log file data shares lots in common with the data from web sites. The majority of the words in log files occur only few times.

Risto Vaarandi conducted an experiment for estimating the occurrence times of words in log file data [1]. Results indicated that a majority of words were very infrequent, and a significant fraction of words appeared just once in the data set.

In LogClusterC, there is a statistics debug feature which can be used here to demonstrate this phenomenon. We selected one log file from a publicly available Computer Failure Data Repository (CFDR) [39] as our sample.

The log file sample is named "BGL" (743.2 MB), which records event logs collected between 2004 and 2006 on supercomputing system Blue Gene/L. We used percent support of 0.01% for this log file with the command line below.[1]

```
./LogClusterC --input=BGL --rsupport=0.01
```

We got the statistics data of vocabulary as shown in Figure 2.

```
5,632,912 words were inserted into the vocabulary.
Finding frequent words from vocabulary...
1,878 frequent words were found.
91% - 5,173,591 words in vocabulary occur 1 time.
92% - 5,222,663 words in vocabulary occur 2 times or less.
94% - 5,298,390 words in vocabulary occur 5 times or less.
96% - 5,424,315 words in vocabulary occur 10 times or less.
97% - 5,470,353 words in vocabulary occur 20 times or less.
99.97% - 5,631,034 words in vocabulary occur less than 474(support) times.
```

Figure 2. Word frequency statistics of log file BGL

From the 5,632,912 distinct words appearing in this log file, only 1,878 words (0.03%) were counted as frequent words. The majority part (91%) of the words occurred only one time, and 97% of the words occurred 20 times or less.

---

[1] When using percentage support, the real support threshold is calculated as *number of lines * percentage support*. For example, if we use percent support of 0.01% to a log file that has 2,000,000 lines, the support threshold will be 200.

3.3.2. Processing time optimization: the move-to-front technique in hash table

The feature described in this section is fundamental and built-in by default. Thus, LogClusterC does not provide an option to tune this feature.

Taking the log file sample BGL in Section 3.3.1. as the example, in "Step 2.2. Create vocabulary", LogClusterC has added 5,632,912 distinct words into the word table. Since the default word table size is 100,000 in LogClusterC[1], and the Shift-Add-Xor string hashing algorithm uniformly distributes the input string into integer within the given space (from 1 to 100,000)[2], we can predict that each word table slot is shared by approximately 56 words. LogClusterC uses a linked list data structure to arrange all the words that share the same slot.

As shown in Figure 3, assuming words "*bob*", "*22:30:54*", "*login*" and "*0x145=0*" are hashed to integer 2016 by the Shift-Add-Xor string hashing algorithm, they share the same word table slot.[3] Therefore, word "*bob*" resides in slot 2016, and the other words are connected to the slot with a linked list data structure.

We can see that word "*login*" appears more often than other three words. Using the data structure described in Figure 3, every time LogClusterC increments the counter of word "*login*", it has to traverse over words "*bob*" and "*22:30:54*" to locate the target. However, this approach is not efficient, especially in scenarios where one slot stores hundreds of distinct words.

---

[1] The word table size can be changed by using option '--wtablesize'. For example, to use a word table with a size of 300, 000, the command line is: ./LogClusterC --input=BGL --support=0.01% --wtablesize=300000.

[2] Internally, the numbers run from 0 to 99,999. However, for the convenience of reading, the numbers range from 1 to 100,000 here. This convention applies to the rest of this thesis.

[3] Note that words that are hashed to the same hash value in this session, will not necessarily be hashed to the same hash value again in next session, because there are random numbers involved in the Shift-Add-Xor string hashing algorithm. Each time we run LogClusterC, a new series of random numbers will be generated.
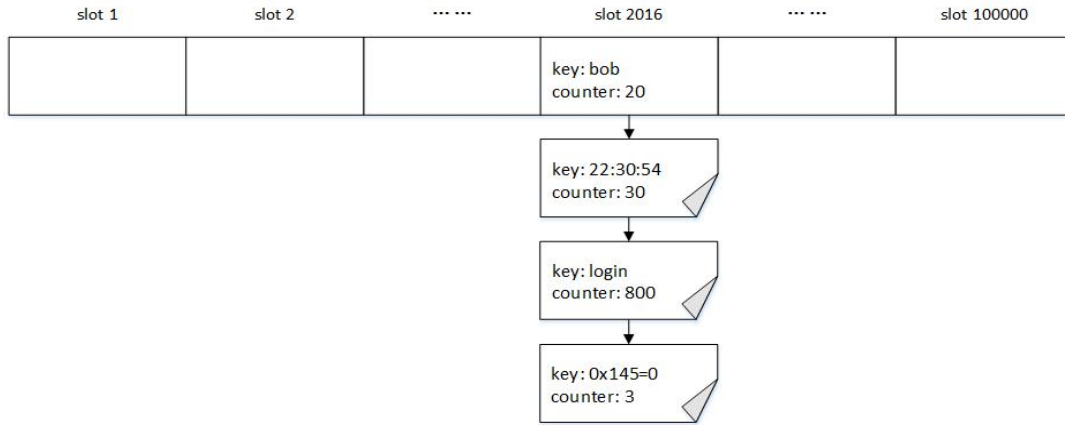
Figure 3. The word table's data structure

The move-to-front technique was designed for this kind of situation. When LogClusterC adds a new word into the word table or increments an existing word's counter, this word will be moved to the most front position of the linked list data structure (where word "*bob*" resides in Figure 3). For example, if LogClusterC has incremented the counter of word "*login*", the state of the word table will change as expressed in Figure 4.
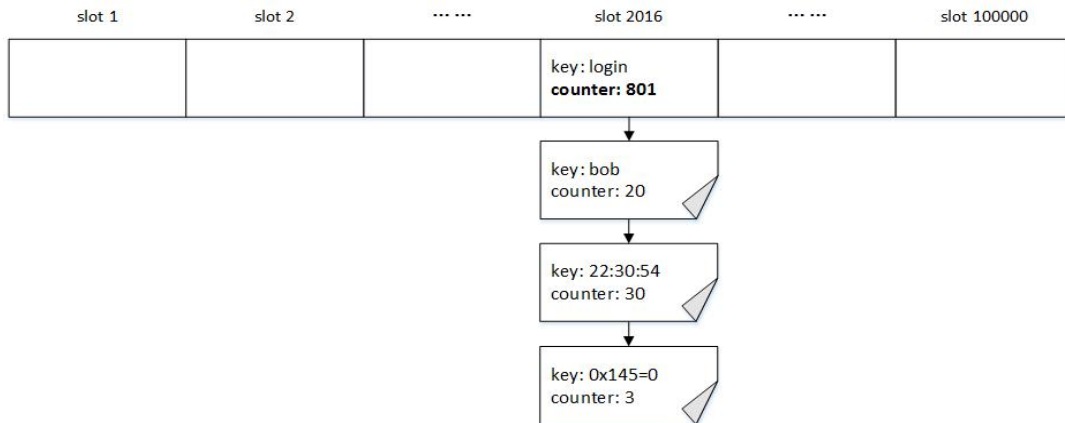
Figure 4. The word table's data structure with the move-to-front technique

If the word "*login*" appears again in next line of the log file, LogClusterC will efficiently locate the target word by directly looking into slot 2016.

By implementing the move-to-front technique, the words that appear more frequently would tend to be in front positions in the linked list data structure. Considering the nature of log file data, the average processing time of word queries is significantly reduced.

Moreover, the move-to-front technique is also implemented in the cluster table which is created during "Step 3.2. Find cluster candidates". Since cluster candidate identifiers have the same statistical feature as words, that the majority of cluster candidate identifiers occur infrequently, the move-to-front technique provides additional performance benefits when used for the cluster table.

3.3.3. Memory consumption optimization: the sketch preprocessing

The features described in this section can be turned on by using '--wsize=<word sketch size>' and '--csize=<cluster candidate sketch size>' options.

Taking the log file sample BGL in Section 3.3.1. as the example, in "Step 2.2. Create vocabulary" LogClusterC has added 5,632,912 distinct words into the word table. In LogClusterC, every time a new word is added to vocabulary, there will be a 40-byte memory space being allocated to it.[1] Therefore, approximately 214.9 MB memory is occupied during "Step 2.2. Create vocabulary". For an environment that suffers from shortage of memory, such an amount of memory may be too expensive to consume.

Considering the nature of log file data, the majority of words appears only few times. Therefore, these words can not have an occurrence counter greater than the support threshold, and allocating memory space to them is a waste of computing resource. To avoid this kind of waste, we use the sketch preprocessing to prevent them from being added to the vocabulary.

Referring to Figure 1, "Step 2.1. Create word sketch" is an optional process that happens before "Step 2.2. Create vocabulary" and requires one pass over the data set. The option that enables this feature is '--wsize=<word sketch size>'. To use a word sketch that has a size of 1,000,000 counters with the example in Section 3.3.1., the command line is shown below.

```
./LogClusterC --input=BGL --support=0.01% --wsize=1000000
```

As the number of distinct words in log file BGL is 5,632,912, a slot in the word sketch would be shared by approximately six words since the words are uniformly distributed over the 1,000,000 sketch counters by the Shift-Add-Xor string hashing algorithm. In

---

[1] In 64-bit environment. For details, please refer to "struct Elem" in the source code.

the following we assume that some of the words have the properties as shown in Table 2.

Table 2. Statistics data of words in log file BGL (hypothetical)

Hash value	Words and their occurrence times
1	omitted
... ...	omitted
123456	0x12e=0 (5), 00e28d98 (10), 003a9260 (10), R72-M1-N7-C (10), 1125084401 (5)
123457	2005.08.26 (200), 1524480 (20), 570616 (10), 02494c2c (10)
123458	1131060638 (10), **login (30000)**, R67-M1-NA-C (10), 953845 (10), J02-U11 (5)
123459	2005-11-03 (100), 02494c1c (10), 0x01fc1a20 (10), 4519885 (20), R23-M1-Nb-C (10), J03-U01 (10)
... ...	omitted
1000000	omitted

During this pass over the data, LogClusterC calculates the hash value of every word, and increments the counter of the corresponding slot in the word sketch. For the words in Table 2, the completed word sketch is presented in Figure 5.
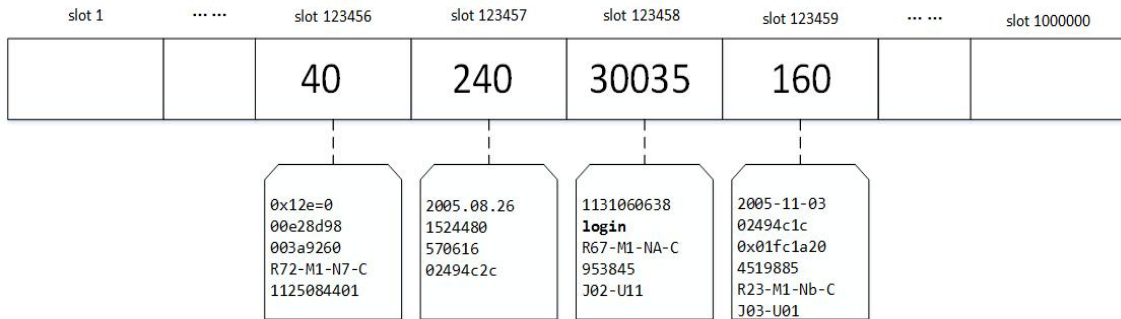
Figure 5. The completed word sketch

The completed word sketch is then used in "Step 2.2. Create vocabulary", where one word is added to the word table only if its corresponding word sketch slot has a counter equal or greater than the support threshold. For example, word "*0x12e=0*" will be skipped from the word table because its word sketch slot 123456 has a counter of 40, which is smaller than the support threshold 474. On the other hand, words "*1131060638*" and "*login*" will be added to the word table, because their word sketch slot 123458 has a counter of 30,035. Considering the nature of log file data, that the majority of words appear few times, many word sketch slots will have counters that are

less than the support threshold. Therefore, many infrequent words are not added to the word table.

The word sketch shown in Figure 5 occupies only 7.6 MB memory[1], but it will bring promising memory consumption optimization for "Step 2.2. Create vocabulary". Despite the benefit, there is a trade-off that one extra pass over the data set is needed.

To demonstrate the performance of the sketch preprocessing, a simple comparison experiment was conducted, using a simplified version of LogClusterC[2] and the time utility.

To get vocabulary of log file BGL without using the sketch preprocessing, the command line is shown below, and the output is in Appendix 1.1. The format explanation of the time utility can be found in [44].

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./voca --input=BGL --support=0.01%
```

To get vocabulary of log file BGL with a word sketch of size 1,000,000, the command line is shown below, and the output is in Appendix 1.2.

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./voca --input=BGL --support=0.01% --
wsize=1000000
```

From the results of these two sessions, we can see that the word sketch preprocessing significantly reduced the memory consumption: occupied memory decreased from 517,436 KB (505.3 MB) to 13,128 KB (12.8 MB). Surprisingly, we did not see a growth in processing time. Instead, when using the word sketch preprocessing, runtime decreased from 29.99 seconds to 24.36 seconds.[3] This can be explained with the fact

---

[1] The counter in each slot has the unsigned long data type, which occupies eight bytes in 64-bit environment.

[2] This version only has functions in "Phase 1. Preparation" and "Phase 2. Frequent words". The rest of the phases are deleted.

[3] Different hardware configurations can get different results in this experiment. The hardware configuration we used is in Chapter 4.

that the word sketch reduced the workload in the second pass over the data set (Step 2.2.), allowing to skip the processing of infrequent words (allocating memory, querying and incrementing counters).

To provide an example case where the word sketch increases the processing time, we selected a large log file named "TDB" (39.1 GB), which records events collected between 2004 and 2006 on supercomputing system Thunderbird. In the comparison experiment, we used the similar two command lines which were previously applied to log file BGL. The outputs are in Appendix 1.3. and 1.4.

The results regarding log file TDB once again demonstrated that sketch preprocessing has an outstanding ability to reduce memory consumption: occupied memory decreased from 1,926,316 KB (1.8 GB) to 32884 KB (32.1 MB). The processing time was increased due to the extra pass over data set: runtime increased from 929.14 seconds to 1,185.10 seconds.

If we set a relatively low support threshold for the clustering process, it is possible that there would be a large amount of cluster candidates generated. Each cluster candidate consumes 4.1 KB of memory space.[1] Compared to a word which occupies 40 bytes of memory space, the memory cost of a cluster candidate is much more higher. To save the unnecessarily wasted memory space on cluster candidates that are infrequent and have no possibility to become a cluster, the cluster candidate sketch preprocessing can be used.

The cluster candidate sketch preprocessing works according to the same principle as the word sketch preprocessing. It can be turned on by using option "--cize=<cluster candidate sketch size>", and an example command line is shown below.

```
./LogClusterC --input=sample.log --support=200 --csize=1000000
```

---

[1] In 64-bit environment. For details, please refer to "struct Elem", "struct Cluster", "struct ClusterWithToken" and "struct Token" in the source code.

3.4. Implementation of heuristics for mitigating over-fitting

There are two heuristics in LogCluster that have been designed for mitigating over-fitting: Aggregate_Supports and Join_Clusters. This section describes the details of their implementation in LogClusterC.

3.4.1. The Aggregate_Supports heuristic

The feature described in this section can be turned on by using '--aggrsup' option.

To address SLCT's shortcoming that larger meaningful clusters often disappear (being split) due to over-fitting, LogClusterC implements the Aggregate_Supports heuristic to aggregate the support values of potentially split cluster candidates, in order to increase the support value of more general cluster candidates.

Referring to Figure 1, the Aggregate_Supports heuristic (Step 3.3.) is applied right after all the cluster candidates are generated, before the clusters are selected.

If cluster candidate *X's* description are more specific than cluster candidate *Y's* description, then all event log lines that match the description of *X* also match the description of *Y*. For each cluster candidate, the Aggregate_Supports heuristic tries to find other cluster candidates that have a more specific cluster candidate description. For each more specific cluster candidate, its support value will be added to the support value of the original cluster candidate (the more general one). With the Aggregate_Supports heuristic activated, one event log line could belong to multiple cluster candidates. In other words, cluster candidates can overlap.

For example, assume there are three cluster candidates whose descriptions and support values are shown below.

```
Interface eth0 no shutdown 22:23:24, support 5
Interface *{1,1} no shutdown 22:23:24, support 10
Interface *{1,1} no shutdown *{1,1}, support 100
```

For cluster candidate "*Interface *{1,1} no shutdown *{1,1}*", cluster candidates "*Interface *{1,1} no shutdown 22:23:24*" and "*Interface eth0 no shutdown 22:23:24*" are more specific cluster candidates. Similarly, for cluster candidate "*Interface *{1,1} no shutdown 22:23:24*", cluster candidate "*Interface eth0 no shutdown 22:23:24*" is

more specific. Therefore, after the Aggregate_Supports heuristics, the support values of these three cluster candidates would be changed as shown below.

```
Interface eth0 no shutdown 22:23:24, support 5
Interface *{1,1} no shutdown 22:23:24, support 15
Interface *{1,1} no shutdown *{1,1}, support 115
```

Note that the process of finding more specific candidates is not only about matching word with wildcard, but also about matching wildcard with wildcard. For instance, cluster candidates "*Interface *{1,1} no shutdown *{1,1}*" and "*Interface *{1,1} HQ no shutdown *{1,1}*" are both considered more specific than cluster candidate "*Interface *{1,2} no shutdown *{1,1}*".

In order to achieve efficient querying and support accumulation between similar cluster candidates, LogClusterC uses the prefix tree data structure to organize all the cluster candidates. For example, consider a set of cluster candidates whose descriptions are shown in Figure 6. The corresponding prefix tree for these cluster candidates is displayed in Figure 7.[1]

```
User bob login from 10.1.1.1
User *{1,1} login from 10.1.1.1
User *{1,1} login from *{1,1}
User *{1,1} HQ login from *{1,1}
User *{1,2} login from *{1,1}
Interface eth0 down
Interface eth0 *{2,2}
Interface *{1,1} down
Interface HQ Link down
*{1,2} service available
```

Figure 6. The example cluster candidates set

---

[1] For more details about the construction and utilization of the prefix tree data structure in LogClusterC, please refer to function "void step_2_aggregate_support(struct Parameters *pParam)" in the source code.
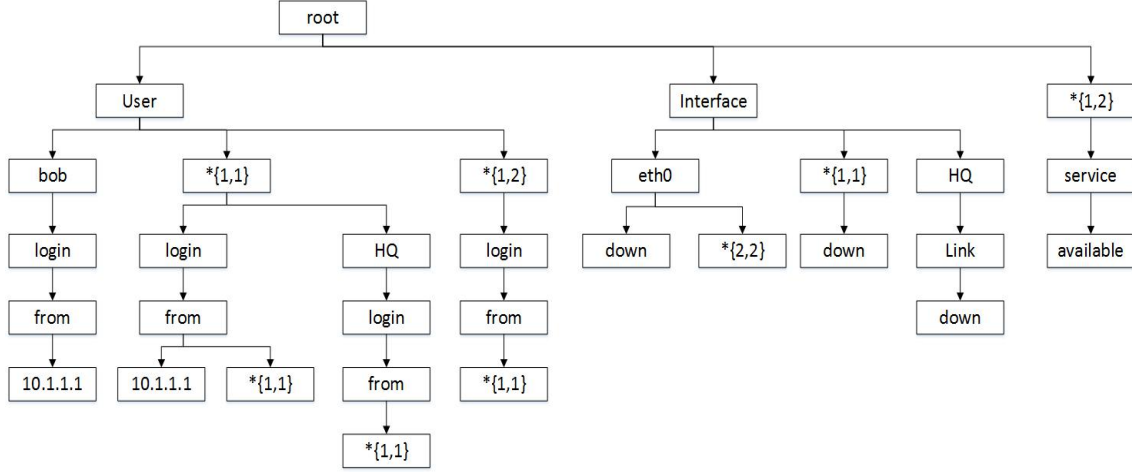
Figure 7. The prefix tree data structure of the cluster candidates set

To demonstrate the usage of the prefix tree data structure, we take finding more specific cluster candidates for cluster candidate "*User *{1,2} login from *{1,1}*" as the example. Using the prefix tree, LogClusterC starts with locating the first wildcard's parent, which is "*User*". LogClusterC then traverses all the children of "*User*" to find the eligible cluster candidates, and safely ignores the other cluster candidates under branches other than "*User*". As the result, all cluster candidates under "*User*" (except the original cluster candidate itself) are considered as more specific cluster candidates. The prefix tree allows to narrow down the search to relevant candidates only and thus considerably increases the performance of the Aggregate_Supports heuristic.

3.4.2. The Join_Clusters heuristic

The feature described in this section can be turned on by using '--wweight=<word weight threshold>' option. Option '--weightf=<word weight function>' can be used for additional tuning.

As mentioned in Section 2.2.2., SLCT is inflexible in cluster generation, since it rigidly relies on the frequent word mechanism [27]. For example, "*Interface eth0 down*", "*Interface eth1 down*" and "*Interface eth2 down*" are generated as three distinct clusters instead of one, although only the second word is different. Moreover, the second word represents the interface name which is a variable field[1], while words "*Interface*" and "*down*" have a constant nature. A straightforward solution could be presenting these

---

[1] Though words "eth0", "eth1" and "eth2" are frequent words (constant), they are relatively variable when compared to other frequent words.

similar clusters as a joint cluster with one pattern, such as *"Interface (eth0|eth1|eth2) down"* whose support value is the sum of the support values of the joined clusters. To provide this solution, the Join_Clusters heuristic is introduced by LogCluster [2].

Referring to Figure 1, the Join_Clusters heuristic (Step 4.2.) is applied after the clusters are found from cluster candidates. It can be considered as a postprocessing step that refines the descriptions of clusters. The heuristic starts with the identification of words in the cluster description which are weakly correlated to other words of the description. The heuristic then masks weakly correlated words in the cluster description with the same token, and joins two clusters if their descriptions match after masking.

To help determine whether a word is weakly correlated, two properties are defined for the words in the cluster description: word dependency and word weight. A word will be considered as weakly correlated, if its word weight is below the word weight threshold, which is given by the user via option '--wweight=<word weight threshold>'.

To calculate a word's weight, the user can choose from two alternative functions with option '--weightf=<word weight function>'. Function 1 is used as default if the user does not set this option.[1]

Function 1 is described as below, where k represents the number of frequent words in the cluster description, and $dep(w_j, w_i)$ represents the word dependency from w_j to w_i.

$$weight(w_i) = \sum_{j=1}^{k} dep(w_j, w_i) / k \qquad\qquad (3.1)$$

Function 2 is described as below, where unique frequent words are first found out, and p represents the number of unique frequent words in the cluster description.

$$weight(w_i) = \begin{cases} \left(\sum_{j=1}^{p} dep(w_j, w_i) - dep(w_i, w_i)\right)/(p-1) & (p > 1) \\ 1 & (p = 1) \end{cases} \qquad (3.2)$$

---

[1] Function 2 was introduced in LogClusterP version 0.03. The number of functions is constantly growing.

Word dependency $dep(w_j, w_i)$ reflects how frequently word w_i occurs in event log lines which contain word w_j. To calculate $dep(w_j, w_i)$, we follow the below equation, where I_w represents the lines that contain word w.

$$dep(w_j, w_i) = |I_{w_j} \cap I_{w_i}| / |I_{w_j}|$$ (3.3)

Note that $dep(w_i, w_i) = 1$ and $0 < dep(w_j, w_i) \leq 1$ [1]. Therefore, $1/k \leq weight(w_i) \leq 1$ in word weight function 1, and $0 < weight(w_i) \leq 1$ in word weight function 2.

LogClusterC calculates word dependency by querying the word dependency matrix, which is built via one pass over the data set. For the sake of processing time optimization, this pass over data set is integrated into "Step 3.2. Find cluster candidates". In other words, when LogClusterC finishes extracting the cluster candidate from an event log line, it continues to update the word dependency matrix according to the frequent words that are found in this event log line.

The size of the word dependency matrix is $N*N$, where N is the number of frequent words in the data set.

To demonstrate the process of building the word dependency matrix, we take the cluster candidates set in Figure 6 as the example, assuming these cluster candidates are the results of the extraction from ten event log lines in "Step 3.2. Find cluster candidates". Figure 8 expresses the final state of the word dependency matrix after all the ten event log lines are processed.

---

[1] $dep(w_j, w_i)$ is always greater than zero, since word w_i and word w_j belong to the same cluster description. Therefore, there are event log lines that contain both word w_i and w_j, and thus $|I_{w_j} \cap I_{w_i}| > 0$.

	1. User	2. bob	3. login	4. from	5. 10.1.1.1	6. HQ	7. Interface	8. eth0	9. down	10. Link	11. service	12. available
1. User	**5**	1	5	5	2	1						
2. bob	1	**1**	1	1	1							
3. login	5	1	**5**	5	2	1						
4. from	5	1	5	**5**	2	1						
5. 10.1.1.1	2	1	2	2	**2**							
6. HQ	1		1	1		**2**	1		1	1		
7. Interface						1	**4**	2	3	1		
8. eth0							2	**2**	1			
9. down						1	3	1	**3**	1		
10. Link						1	1		1	**1**		
11. service											**1**	1
12. available											1	**1**

Figure 8. The final state of the word dependency matrix

Referring to the word dependency matrix and equation 3.3, we now can calculate the word dependency between two given words. For instance,

$$dep(from,bob) = matrix(4,2)/matrix(4,4) = 1/5 = 20\%$$

means that dependence from word "*from*" to word "*bob*" is 20%. That is to say, if an event log line contains word "*from*", there is 20% chance that it also contains word "*bob*". Whereas,

$$dep(bob, from) = matrix(2,4)/matrix(2,2) = 1/1 = 100\%$$

means that dependence from word "*bob*" to word "*from*" is 100%. That is to say, if an event log line contains word "*bob*", we are 100% sure that it must also contain word "*from*".

With word dependency values calculated, LogCluserC then determines whether a word should be masked in the cluster description. For example, to determine whether word "*bob*" should be masked in cluster "*User bob login from 10.1.1.1*" using Function 1, LogClusterC gets the word weight of word "*bob*":

$$weight(bob) = \left(\begin{array}{l} dep(User, bob) + dep(bob, bob) + dep(\log in, bob) + \\ dep(from, bob) + dep(10.1.1.1, bob) \end{array} \right) / 5$$
$$= (1/5 + 1/1 + 1/5 + 1/5 + 1/5) / 5 = 9/25$$
$$= 0.36$$

If the user sets a word weight threshold greater than 0.36, the word "*bob*" will be masked. Masking often produces identical cluster descriptions for clusters that represent similar events. For example, clusters "*User bob login from 10.1.1.1*", "*User jim login from 10.1.1.1*" and "*User mike login from 10.1.1.1*" all represent events that manifest user login from 10.1.1.1. If all user names are masked in the above cluster descriptions, the resulting descriptions are identical, and therefore the heuristic will generate the joined cluster "*User (bob|jim|mike) login from 10.1.1.1*" from above three clusters.

To demonstrate how the Join_Clusters heuristic performs in practice, we applied it to log file BGL with a word weight threshold of 0.4, using the command line below.

```
./LogClusterC --input=BGL --support=200 --wweight=0.4
```

LogClusterC found 813 clusters, and 797 clusters contained frequent words under word weight threshold. Those 797 clusters were then joined into 95 clusters. Some of the most representative joined clusters are listed in Figure 9.

```
- *{1,1} (2005.06.03|2005.06.22|2005.06.23|2005.06.25|2005.07.01|2005.07.07|
2005.07.09|2005.07.13|2005.08.26|2005.08.28) *{3,3} RAS KERNEL INFO *{1,1}
double-hummer alignment exceptions
Support : 119,696

- *{1,1} 2005.06.14 *{3,3} RAS KERNEL FATAL program interrupt: (illegal|
privileged|trap|unimplemented|imprecise|fp) (instruction......0|
instruction...0|instruction........0|operation..0|exception......0|compare....
..........0)
Support : 40,362

- *{1,1} 2005.06.26 *{3,3} RAS KERNEL INFO (machine|byte|instruction)
(check:|ordering|address:) (0x0062fdf4|i-fetch.....................0|
exception...................0|0x0062fe04)
Support : 9,161
```

Figure 9. The most representative joined clusters generated by the Join_Clusters heuristic

4.   Performance comparison experiments

This section presents experiments for evaluating the performance of LogClusterC, LogClusterP and SLCT. The parameters of evaluation are processing speed (measured in terms of runtime and CPU time consumption in system and user mode), memory consumption, and the number of clusters. Note that since all tested tools are single-threaded and almost fully utilize one CPU core during execution, runtime and consumed CPU time figures are almost identical for all experiments.

Seven log files were involved in experiments which were selected from CFDR [39]. Their aliases in the following experiments and descriptions are listed in Table 3.

Table 3. Log files involved in experiments

Alias	Description	Size (MB)
Cray_A	This data set comes from one or more Cray XT series machines, running Linux. It is selected from CFDR - The Cray data - Data set 6.	20.86
Cray_B	This data set comes from one or more Cray XT series machines, running Linux. It is selected from CFDR - The Cray data - Data set 4.	52.12
Cray_C	This data set comes from one or more Cray XT series machines, running Linux. It is selected from CFDR - The Cray data - Data set 1.	172.72
BGL	This data set contains event logs collected between 2004 and 2006 on supercomputing system: Blue Gene/L. It is selected from CFDR - The HPC4 data - BlueGene/L.	708.76
LBR	This data set contains event logs collected between 2004 and 2006 on supercomputing system: Liberty. It is selected from CFDR - The HPC4 data - Liberty.	30,235.34 (29.53 GB)
TDB	This data set contains event logs collected between 2004 and 2006 on supercomputing system: Thunderbird. It is selected from CFDR - The HPC4 data - Thunderbird.	30,386.44 (29.67 GB)
SPT	This data set contains event logs collected between 2004 and 2006 on supercomputing system: Spirit. It is selected from CFDR - The HPC4 data - Spirit.	38,236.88 (37.34 GB)

The experiments were conducted on a computer running Ubuntu Server 14.04.[1] The hardware configuration was: Intel i7-4720 HQ 2.6 GHz CPU, Kingston 16 GB DDR3 1600 MHz RAM, Western Digital 1 TB 5400 rpm hard disk.

The figures of runtime, CPU time system, CPU time user and memory consumption in Table 4 and Table 6 are the average values calculated from five times' repeated experiments. Conducting the experiment for one time costs approximately 75 hours on this computer. The computer was shut down and rebooted after each time's experiment.

To support the experiment principle of reproducibility and rigor, the bash script to run the experiment and the raw plain text outputs in each experiment are available in [41].

4.1. Performance comparison of basic functions between LogClusterC, LogClusterP and SLCT

4.1.1. Experiment introduction

This section compares the performance in basic mode which involves finding clusters via two passes over the data set. For all algorithm implementations, their respective command lines have only two options: '--input=<file name>' and '--support=<support threshold>'.

The tools evaluated in this experiment are LogClusterC (version 0.03), LogClusterP (version 0.03) and SLCT (version 0.05). LogClusterP-0.08 substitutes LogClusterP-0.03 in Row 15, Row 18 and Row 21, to provide the cluster candidate sketch feature that is not present in version 0.03.

The seven log files adequately represent daily event log clustering scenarios. The first four log files have sizes less than 1 GB, imposing moderate workload of an event log clustering task. The last three log files are around 30 GB by their size, representing computationally more demanding clustering scenarios.

We used three different support thresholds to each log file: percentage support 0.5%, percentage support 0.1% and constant support threshold 200.

---

[1] This was a minimum installation of Ubuntu Server 14.04, only gcc 4.8 package was installed besides the kernel. The other popular software, such like OpenSSH server, DNS server, LAMP server, were not installed.

When using percentage support, the real support threshold is calculated after the first pass over data set ("Step 2.1. Create word sketch" or "Step 2.2. Create vocabulary") with the knowledge of the number of lines in the log file. The real support threshold is then calculated as *number of lines * percentage support*. Using percentage support is very handy in circumstances where we do not know the details of the clustering target.

The constant support 200 was selected for testing the efficiency of memory-resident data structures used by the tools. Applying this small support threshold will generate a large amount of frequent words and cluster candidates. Thus, the memory consumption and workload of the tools would increase considerably.

We used the time utility for doing all measurements during the experiment (the usage explanation of the time utility can be found in [44]). The command lines used in this experiment are listed below, taking Row 1 as the example.[1]

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./LogClusterC --input=Cray_A --
rsupport=0.5
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' perl logcluster.pl --input=Cray_A --
rsupport=0.5
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./SLCT -s 0.5% Cray_A
```

Note that in Row 15, Row 18, and Row 21, we applied the word sketch and cluster candidate sketch preprocessing, because with a support threshold of 200, these large log files (LBR, TDB and SPT) generated too many frequent words and cluster candidates that overwhelmed the 16 GB RAM. The word sketch size was set to two million, and the cluster candidate sketch size was set to 200 million. As previously mentioned, we evaluated LogClusterP-0.08 instead of LogClusterP-0.03 in these three Rows. The command lines used in these three Rows are listed below, taking Row 15 as the example.

---

[1] Option '--support=0.5%' and option '--rsupport=0.5' have the same meaning. They are interchangeable in LogClusterC.

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./LogClusterC --input=LBR --
support=200 --wsize=2000000 --csize=200000000


/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' perl logcluster008.pl --input=LBR --
support=200 --wsize=2000000 --csize=200000000


/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./SLCT -s 200 -v 2000000 -z 200000000
LBR
```

The results of the experiment are presented in Table 4. The abbreviation "LCC" is short for LogClusterC, and "LCP" is short for LogClusterP. This abbreviation applies to the rest tables of this thesis.

LogClusterP does not output the number of words in vocabulary. Theoretically, this value should be the same as LogClusterC's number of words in vocabulary in the same Row, except in Row 15, Row 18 and Row 21, where word sketch feature has randomly reduced the amount of infrequent words added into vocabulary. To get this value, one way is modifying the source code of LogClusterP. Another way is using third party word count tools, because LogCluster algorithm is not position sensitive (every word is added to vocabulary without modification). We used third party word count tools[1] as the solution, which also verified the principle of LogCluster algorithm (being insensitive to word position) from a new perspective. Thus, the corresponding blanks in Row 15, Row 18 and Row 21 are left with "N/A".

Memory consumption represents the maximum resident set size of the process during its lifetime [44].

---

[1] The tools are all pre-installed Linux unities. For example, to get the distinct word numbers of log file Cray_A, we used the command line: *cat Cray_A | tr " " "\n" | sort | uniq -c | tee Cray_A_output | wc -l.*

This approach has a shortcoming that the result may be slightly different from LogClusterC's value. For example, we got log file TDB's value as 23,330,816, while LogClusterC's value is 23,330,821. Considering the difference is only 0.00002% and the values match in all other six log files, we think this approach is reliable.

Table 4. Performance comparison of basic functions

Row # / Log file	Event log size in megabytes	Event log size in lines	Support threshold	Measure	Run time — LCC	Run time — LCP	Run time — SLCT	Memory (MB) — LCC	Memory (MB) — LCP	Memory (MB) — SLCT	No. clusters — LCC	No. clusters — LCP	No. clusters — SLCT	Cluster candidates — LCC	Cluster candidates — LCP	Cluster candidates — SLCT	Frequent words — LCC	Frequent words — LCP	Frequent words — SLCT	Words in vocabulary — LCC	Words in vocabulary — LCP	Words in vocabulary — SLCT
Row 1 / Cray_A	20.86	379,457	1,897	Run time in seconds	1.43	7.02	1.29	270.15	218.41	24.84	0	0	0	63,876	63,876	53,615	116	116	150	13,737	13,737	19,144
				CPU time system	0.05	0.04	0.01															
				CPU time user	1.24	6.61	1.27															
Row 2 / Cray_A	20.86	379,457	379	Run time in seconds	1.26	7.13	1.26	342.67	272.30	35.35	27	27	29	81,044	81,044	79,280	347	347	420	13,737	13,737	19,144
				CPU time system	0.06	0.05	0.01															
				CPU time user	1.20	7.07	1.25															
Row 3 / Cray_A	20.86	379,457	200	Run time in seconds	1.47	7.99	1.34	639.97	527.96	64.84	13	13	16	151,707	151,707	150,299	1,051	1,051	1,146	13,737	13,737	19,144
				CPU time system	0.11	0.08	0.01															
				CPU time user	1.35	7.90	1.33															
Row 4 / Cray_B	52.12	958,075	4,790	Run time in seconds	2.82	16.65	2.94	30.30	26.67	6.44	32	32	32	5,700	5,700	910	77	77	78	4,853	4,853	6,849
				CPU time system	0.03	0.02	0.01															
				CPU time user	2.71	16.63	2.93															
Row 5 / Cray_B	52.12	958,075	958	Run time in seconds	3.54	21.13	2.98	1,614.05	1,412.11	12.79	10	10	10	382,322	382,322	15,667	776	776	447	4,853	4,853	6,849
				CPU time system	0.29	0.31	0.01															
				CPU time user	3.24	20.83	2.97															
Row 6 / Cray_B	52.12	958,075	200	Run time in seconds	4.08	22.23	3.38	1,960.11	1,638.82	176.64	323	323	323	465,187	465,187	412,317	1,598	1,598	1,619	4,853	4,853	6,849
				CPU time system	0.35	0.33	0.05															
				CPU time user	3.72	21.9	3.33															
Row 7 / Cray_C	172.72	3,170,514	15,852	Run time in seconds	9.13	54.66	9.61	42.44	36.44	7.22	26	26	26	8,622	8,622	1,011	70	70	69	9,766	9,766	16,741
				CPU time system	0.06	0.03	0.03															
				CPU time user	8.93	54.65	9.58															

Table 4 cont. Performance comparison of basic functions

Row # / Log file	Event log size in megabytes	Event log size in lines	Support threshold	Run time metric	Run time LCC	Run time LCP	Run time SLCT	Memory LCC	Memory LCP	Memory SLCT	Clusters LCC	Clusters LCP	Clusters SLCT	Cand. LCC	Cand. LCP	Cand. SLCT	Freq words LCC	Freq words LCP	Freq words SLCT	Vocab LCC	Vocab LCP	Vocab SLCT
Row 8 Cray_	172.72	3,170,514	3,170	Run time in seconds	11.79	67.90	10.85	4,408.69	3,850.46	362.61	44	44	44	1,046,947	1,046,947	836,461	928	928	909	9,766	9,766	16,741
				CPU time system	0.85	0.83	0.10															
				CPU time user	10.94	67.10	10.75															
Row 9 Cray_C	172.72	3,170,514	200	Run time in seconds	15.40	76.54	13.18	6,313.13	5,355.29	622.70	2,044	2,044	2,045	1,501,118	1,501,118	1,500,167	3,018	3,018	3,148	9,766	9,766	16,741
				CPU time system	1.20	1.17	0.16															
				CPU time user	12.77	73.88	11.57															
Row 10 BGL	708.76	4,747,963	23,739	Run time in seconds	44.25	129.58	48.36	506.09	1,125.18	515.82	55	55	58	4,160	4,160	1,890	203	203	245	5,632,912	5,632,912	5,758,648
				CPU time system	0.42	0.49	0.37															
				CPU time user	43.46	129.14	48.01															
Row 11 BGL	708.76	4,747,963	4,747	Run time in seconds	45.36	135.64	47.89	506.52	1,125.71	515.92	162	162	162	34,844	34,844	17,213	564	564	749	5,632,912	5,632,912	5,758,648
				CPU time system	0.33	0.50	0.35															
				CPU time user	45.04	135.20	47.56															
Row 12 BGL	708.76	4,747,963	200	Run time in seconds	48.05	149.51	51.76	2,351.26	2,947.31	516.60	813	813	814	543,035	543,035	538,370	5,794	5,794	9,059	5,632,912	5,632,912	5,758,648
				CPU time system	0.65	0.86	0.39															
				CPU time user	46.44	147.72	50.40															
Row 13 LBR	30,235.34	265,569,231	1,327,846	Run time in seconds	1,467.49	7,269.19	1,552.10	1,150.06	2,737.07	1,184.69	36	36	36	93,010	93,010	54,573	225	225	238	14,971,358	14,971,358	15,411,637
				CPU time system	12.85	15.13	12.99															
				CPU time user	1,444.58	7,256.64	1,536.58															
Row 14 LBR	30,235.34	265,569,231	265,569	Run time in seconds	1,501.50	7,438.88	1,593.84	1,149.27	2,738.11	1,184.81	274	274	274	165,197	165,197	111,536	408	408	450	14,971,358	14,971,358	15,411,637
				CPU time system	13.78	15.88	13.84															
				CPU time user	1,472.08	7,424.48	1,562.62															

Table 4 cont. Performance comparison of basic functions

Row # / Log file	Event log size in megabytes	Event log size in lines	Support threshold	Run time in seconds — total (LCC)	(LCP)	(SLCT)	CPU time system (LCC)	(LCP)	(SLCT)	CPU time user (LCC)	(LCP)	(SLCT)	Memory (MB) LCC	LCP	SLCT	No. of clusters LCC	LCP	SLCT	No. of cluster candidates LCC	LCP	SLCT	No. of frequent words LCC	LCP	SLCT	No. of words in vocabulary LCC	LCP	SLCT
Row 15 * LBR	30,235.34	265,569,231	200	2,822.85	15,700.70	3,149.36	27.65	32.19	27.80	2,747.38	15,656.57	3,071.76	1,978.85	6,774.05	1,971.76	5,738	5,738	5,738	7,406	7,505	7,386	172,121	172,121	173,468	3,949,529	N/A	4,052,483
Row 16 TDB	30,386.44	211,212,192	1,056,060	1,582.64	6,105.66	1,641.37	14.92	20.99	15.20	1,567.01	6,086.68	1,620.74	1,881.92	4,474.65	1,930.45	6	6	9	197,847	197,847	85,862	404	404	453	23,330,821	23,330,816	23,769,344
Row 17 TDB	30,386.44	211,212,192	211,212	1,603.16	6,647.79	1,670.34	16.14	22.07	15.33	1,579.02	6,626.28	1,645.55	4,761.91	5,602.45	1,930.08	22	22	28	1,126,218	1,126,218	913,546	1,232	1,232	1,362	23,330,821	23,330,816	23,769,344
Row 18 * TDB	30,386.44	211,212,192	200	2,796.66	13,251.65	3,031.52	30.22	34.44	30.07	2,741.90	13,213.65	2,976.01	2,286.60	7,005.44	2,296.88	1,683	1,683	1,728	3,292	3,272	3,314	381,278	381,278	385,662	5,617,563	N/A	5,850,133
Row 19 SPT	38,236.88	272,298,969	1,361,494	1,800.08	9,298.23	1,885.33	16.78	20.26	17.16	1,782.77	9,282.24	1,866.67	1,020.01	2,476.78	1,030.72	39	39	39	110,722	110,722	50,506	198	198	205	13,184,442	13,184,442	13,310,394
Row 20 SPT	38,236.88	272,298,969	272,298	1,832.82	9,605.73	1,919.34	17.39	20.86	17.08	1,811.80	9,588.40	1,898.01	1,721.32	2,479.82	1,031.41	122	122	123	407,835	407,835	325,568	449	449	484	13,184,442	13,184,442	13,310,394
Row 21 * SPT	38,236.88	272,298,969	200	4,178.85	20,927.79	4,400.34	35.10	41.35	35.17	3,669.47	20,431.56	3,889.86	2,890.17	7,818.73	2,028.95	164,776	164,776	164,776	237,505	237,420	237,083	2,760,221	N/A	2,792,219	369,684	369,684	371,145

4.1.2. Experiment data analysis

From the measurement data collected during the experiments, we can conclude that

- LogClusterC generates the same output as LogClusterP.
- LogClusterC works faster than LogClusterP.
- Despite being algorithmically more complex, LogClusterC is not slower than SLCT.
- When processing smaller log files (Cray_A, Cray_B and Cray_C), similarly as LogClusterP, LogClusterC requires a larger amount of memory than SLCT.
- When processing larger log files (BGL, LBR, TDB and SPT), LogClusterC requires a much less amount of memory than LogClusterP. LogClusterC's memory consumption remains at the same level as SLCT's.

LogClusterC generates the same output as LogClusterP. Regarding the number of frequent words, the number of words in vocabulary, the number of clusters and the number of cluster candidates, we saw match between LogClusterC and LogClusterP. Additionally, when manually checking the cluster descriptions and their support values, we found that the outputs of LogClusterC and LogClusterP are identical.[1] This demonstrated that LogClusterC fully implements the LogCluster algorithm.[2]

Additionally, by comparing the number of words in vocabulary between LogCluster (LogClusterC and LogClusterP have the same values[3]) and SLCT, we saw that SLCT always generated a bigger vocabulary. This phenomenon was caused by the fact that SLCT considers words with positional information, and will thus create one more entry in the vocabulary if the word appears in another position.

---

[1] The raw plain text outputs that contain cluster descriptions and their support values are publicly available in [41].

[2] Strictly speaking, this only demonstrated that LogClusterC completely applies LogCluster algorithm regarding the basic functions. In Section 4.2., the experiment demonstrated LogClusterC also completely applies LogCluster algorithm regarding the heuristics.

[3] LogClusterC and LogClusterP have the same values except in log file TDB. The detailed explanation is in the footnote 1 on page 45.

Moreover, LogCluster always found the same or less number of clusters than SLCT. Due to its sensitivity to shifts in word positions, SLCT might accidentally detect several clusters for the same event type, while LogCluster will detect a single cluster in this case. This phenomenon was also observed in the experiments in [2].

LogClusterC works faster than LogClusterP. For comparing the runtime between LogClusterC and LogClusterP in a more detailed way, we have arranged runtime data into Table 5. Relation A is calculated as *the runtime of LogClusterP / the run time of LogClusterC*, which means how many times LogClusterC is faster than LogClusterP. Relation B is calculated as *1- (1 / Relation A)*, which demonstrates how much processing time LogClusterC can save.

On average, LogClusterC is 4.89 times faster than LogClusterP (79.55% of the time can be saved). Since the runtime closely estimates consumed CPU time, Table 5 also illustrates that LogClusterC requires much less CPU time.

Table 5. Runtime comparison summary between LogClusterC and LogClusterP

Row #	Relation A	Relation B	Row #	Relation A	Relation B
Row 1 Cray_A	4.91	79.63%	Row 13 LBR	4.95	79.80%
Row 2 Cray_A	5.66	82.33%	Row 14 LBR	4.95	79.80%
Row 3 Cray_A	5.44	81.62%	Row 15* LBR	5.56	82.01%
Row 4 Cray_B	5.90	83.05%	Row 16 TDB	3.86	74.09%
Row 5 Cray_B	5.97	83.25%	Row 17 TDB	4.15	75.90%
Row 6 Cray_B	5.45	81.65%	Row 18* TDB	4.74	78.90%
Row 7 Cray_C	5.99	83.31%	Row 19 SPT	5.17	80.66%
Row 8 Cray_C	5.76	82.64%	Row 20 SPT	5.24	80.92%
Row 9 Cray_C	4.97	79.88%	Row 21* SPT	5.01	80.04%
Row 10 BGL	2.93	65.87%			
Row 11 BGL	2.99	66.56%			
Row 12 BGL	3.11	67.85%	Average	**4.89**	**79.55%**

Despite the algorithm complexity, LogClusterC is not slower than SLCT. In [2] the experiments demonstrate that the Perl version of SLCT is 1.28-1.62 times faster than LogClusterP, because SLCT algorithm has a simple candidate generation procedure. In the runtime comparison of the C-based implementations, we did not see a big difference

between SLCT and LogCluster. Surprisingly, in some Rows (especially for the large log files), the runtime of LogClusterC is slightly smaller than SLCT.

Considering this abnormal phenomenon that a complex algorithm does not suffer from runtime overhead, we suspected that the I/O speed became the bottleneck of SLCT algorithm. We did an extra runtime comparison experiment between the C-based implementations on hardware environment that has faster I/O speed (with SSD hard drive), and this extra experiment did not yield much different results. The details of this extra experiment are provided in Appendix 2.

Overall, we conclude that on commodity hardware LogClusterC is not slower than SLCT.

When processing smaller log files (Cray_A, Cray_B, Cray_C), similarly as LogClusterP, LogClusterC requires a larger amount of memory than SLCT. Since memory consumption is a highly variable parameter that depends on multiple factors, such like log file content, event log line length and support threshold, it is pointless to analyze this parameter precisely. However, as shown in Figure 10, Figure 11 and Figure 12[1], we did observe a pattern that when processing smaller log files, LogClusterC has the same level of memory consumption as LogClusterP, which is much higher than SLCT's memory consumption level.

Considering the complex data structure of word instance and cluster candidate instance in LogCluster algorithm, we expected to observe this pattern.

---

[1] To achieve a better figure expression, the order of the Rows are rearranged according to the maximum values of their memory consumption.
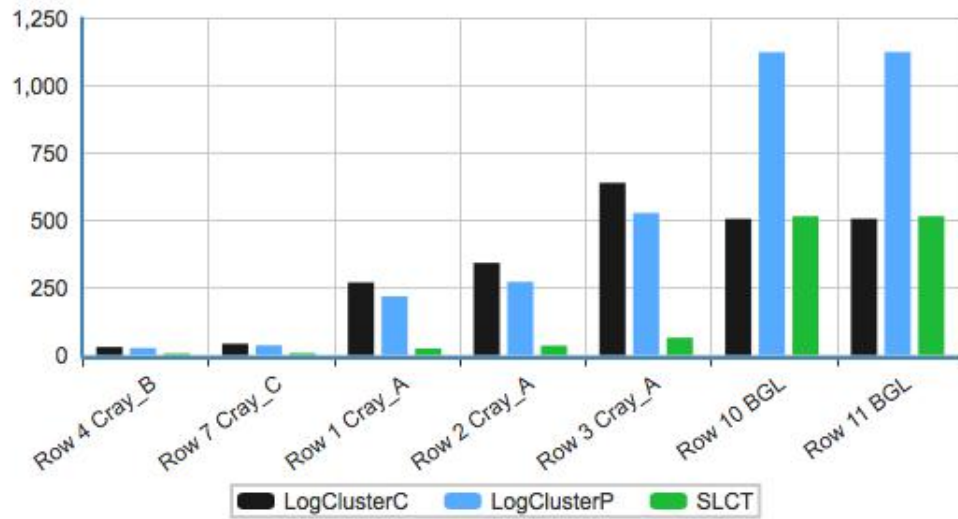
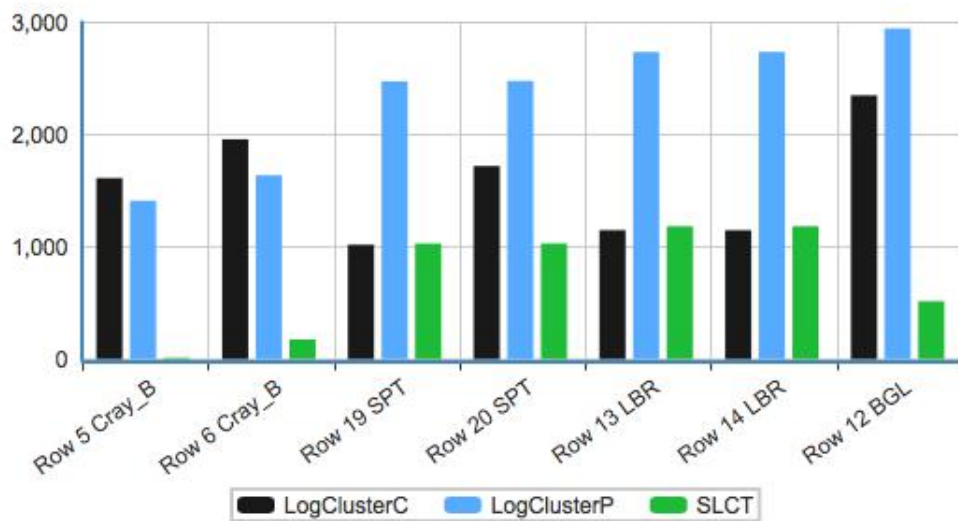*Figure 10. Comparison of memory consumption - 1 (unit: MB)*

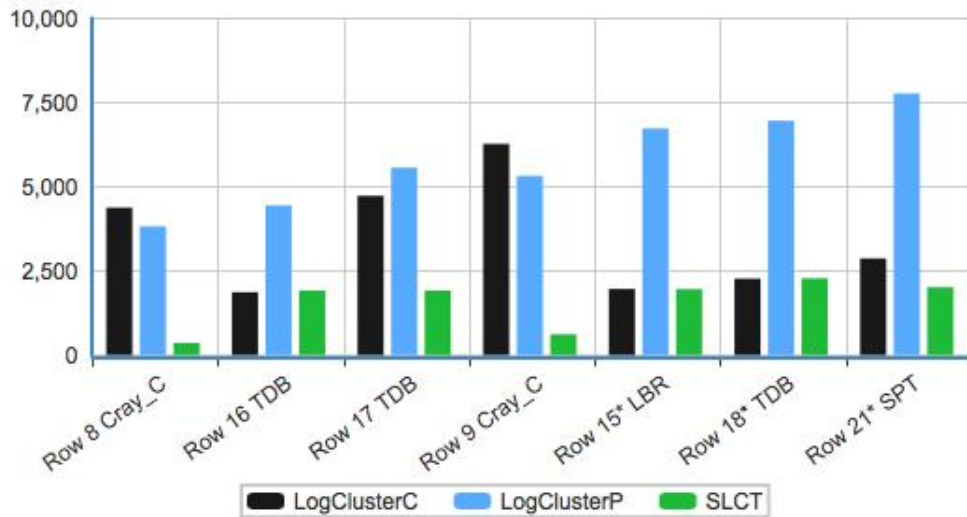*Figure 11. Comparison of memory consumption - 2 (unit: MB)*

Figure 12. Comparison of memory consumption - 3 (unit: MB)

When processing larger log files (BGL, LBR, TDB and SPT), LogClusterC requires a much less amount of memory than LogClusterP. LogClusterC's memory consumption remains at the same level as SLCT's. Surprisingly, LogClusterC and SLCT have the same level of memory consumption when processing larger log files, while LogClusterP consumes a larger amount of memory. An explanation to this phenomenon is that Perl data structures involve a lot more memory overhead when large amounts of data need to be stored. Since the maximum memory consumption have reached 1 GB in those cases, using sketch preprocessing is recommended to reduce the unnecessarily waste of memory. Therefore, we keep on holding the conclusion that LogClusterC's memory consumption remains at the same level as LogClusterP's, which is much higher than SLCT's.

4.2. Performance comparison of Aggregate_Supports and Join_Clusters heuristics between LogClusterC and LogClusterP

4.2.1. Experiment introduction

As the "next generation SLCT", LogCluster not only improved the basic clustering process, but also introduced two heuristics that mitigate over-fitting and optimize the output. In this section, experiments are conducted to evaluate LogClusterC's processing speed, memory consumption and output accuracy when these heuristics are involved.

The tools evaluated in this experiment are LogClusterC (version 0.03) and LogClusterP (version 0.03). We used the same seven log files as in Section 4.1.1.

We used percentage support 0.5% in this experiment. Each log file was processed three times with the Aggregate_Supports heuristic, with the Join_Clusters heuristic, and with both the heuristics. According to the experiment results in [2], LogClusterP yielded the most meaningful output with word weight threshold set between 0.5 to 0.8 in the Join_Clusters heuristic. Thus, we set the word threshold to 0.6 in this experiment.

The results are presented in Table 6. The suffix after the Row number represents the operation in this Row, taking Row1 to Row 4 as the example: Row 1_B means the clustering in basic mode with both heuristics disabled (the results for this mode are copied from Section 4.1.1.); Row 2_A means the Aggregate_Supports heuristic is involved; Row 3_J means the Join_Clusters heuristic is involved, and Row 4_AJ means both the heuristics are involved.

With the time utility harnessed for performance measurements, the six command lines used from Row 2_A to Row 4_AJ are listed below (two command lines for each row).

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./LogClusterC --input=Cray_A --
rsupport=0.5 --aggrsup
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' perl logcluster.pl --input=Cray_A --
rsupport=0.5 --aggrsup
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' ./LogClusterC --input=Cray_A --
rsupport=0.5 --wweight=0.6
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' perl logcluster.pl --input=Cray_A --
rsupport=0.5 --wweight=0.6
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time  user:  %U\nMemory  consumption:  %M'  ./LogClusterC  --input=Cray_A  --
rsupport=0.5 --aggrsup --wweight=0.6
```

```
/usr/bin/time -f 'Command line: %C\nRuntime: %e\nCPU time system: %S\nCPU
time user: %U\nMemory consumption: %M' perl logcluster.pl --input=Cray_A --
rsupport=0.5 --aggrsup --wweight=0.6
```

Table 6. Performance comparison of heuristics

Row # Log file	Runtime		CPU time system CPU time user		Memory consumption in megabytes		Number of clusters	
	LCC	LCP	LCC	LCP	LCC	LCP	LCC	LCP
Row 1_B Cray_A	1.43	7.02	0.05 1.24	0.04 6.61	270.15	218.41	0	0
Row 2_A Cray_A	1.41	30.31	0.05 1.35	0.09 30.23	278.75	433.19	2	2
Row 3_J Cray_A	1.34	12.10	0.06 1.28	0.05 12.05	270.96	219.24	0	0
Row 4_AJ Cray_A	1.50	35.89	0.05 1.45	0.08 35.82	278.85	433.99	1	1
Row 5_B Cray_B	2.82	16.65	0.03 2.71	0.02 16.63	30.30	26.67	32	32
Row 6_A Cray_B	2.73	17.87	0.01 2.71	0.02 17.85	30.70	46.56	74	74
Row 7_J Cray_B	3.02	30.98	0.02 3.00	0.02 30.96	30.60	27.09	10	10
Row 8_AJ Cray_B	3.00	32.43	0.01 2.98	0.02 32.42	30.48	46.89	24	24
Row 9_B Cray_C	9.13	54.66	0.06 8.93	0.03 54.65	42.44	36.44	26	26
Row 10_A Cray_C	9.05	57.00	0.04 9.01	0.05 56.97	42.63	66.02	77	77
Row 11_J Cray_C	9.86	101.78	0.05 9.82	0.08 101.76	41.93	36.79	6	6
Row 12_AJ Cray_C	9.89	103.82	0.04 9.85	0.08 103.79	42.90	66.41	19	19
Row 13_B BGL	44.25	129.58	0.42 43.46	0.49 129.14	506.09	1,125.18	55	55
Row 14_A BGL	44.05	132.14	0.35 43.71	0.46 131.74	506.80	1,125.21	92	92
Row 15_J BGL	46.57	232.49	0.31 46.27	0.62 231.98	505.76	1,125.21	24	24
Row 16_AJ BGL	46.60	240.03	0.34 46.27	0.59 239.55	506.10	1,125.22	42	42

Table 6 cont. Performance comparison of heuristics

Row # Log file	Runtime		CPU time system CPU time user		Memory consumption in megabytes		Number of clusters	
	LCC	LCP	LCC	LCP	LCC	LCP	LCC	LCP
Row 17_B LBR	1,467.49	7,269.19	12.85 1,444.58	15.13 7,256.64	1,150.06	2,737.07	36	36
Row 18_A LBR	1,479.76	7,697.95	13.21 1,454.73	15.29 7,685.22	1,149.64	2,738.11	194	194
Row 19_J LBR	1,678.72	14,783.74	13.50 1,658.45	22.78 14,766.25	1,149.26	2,737.12	6	6
Row 20_AJ LBR	1,667.04	15,219.14	13.76 1,639.30	23.15 15,200.64	1,149.28	2,737.92	32	32
Row 21_B TDB	1,582.64	6,105.66	14.92 1,567.01	20.99 6,086.68	1,881.92	4,474.65	6	6
Row 22_A TDB	1,590.04	7,250.00	15.54 1,570.57	21.12 7,231.51	1,882.34	4,476.14	66	66
Row 23_J TDB	1,708.17	11,824.16	15.78 1,687.22	30.09 11,797.52	1,881.83	4,474.63	5	5
Row 24_AJ TDB	1,722.30	12,933.70	15.92 1,699.70	30.30 12,907.62	1,882.35	4,476.11	23	23
Row 25_B SPT	1,800.08	9,298.23	16.78 1,782.77	20.26 9,282.24	1,020.01	2,476.78	39	39
Row 26_A SPT	1,804.09	9,786.45	16.92 1,784.93	20.66 9,769.46	1,020.04	2,477.72	157	157
Row 27_J SPT	2,072.61	19,646.80	17.11 2,052.99	30.26 19,625.60	1,019.68	2,476.84	6	6
Row 28_AJ SPT	2,107.45	20,422.95	17.40 2,087.34	30.35 20,401.99	1,018.93	2,477.69	32	32

4.2.2. Experiment data analysis

From the experiment data collected for heuristics, we can conclude that:

- LogClusterC generates the same output as LogClusterP.
- LogClusterC works faster than LogClusterP.
- When processing smaller log files (Cray_A, Cray_B and Cray_C), LogClusterC's memory consumption remains at the same level as LogClusterP's.

- When processing larger log files (BGL, LBR, TDB and SPT), LogClusterC has a significantly reduced memory consumption compared to LogClusterP.

LogClusterC generates the same output as LogClusterP. Regarding the number of clusters, we saw match between LogClusterC and LogClusterP. Manually comparison of the cluster descriptions and their support values confirmed that LogClusterC and LogClusterP generated the same outputs.[1] This demonstrates that LogClusterC fully implements the LogCluster algorithm.

We observed that the Aggregate_Supports heuristic significantly increased the number of clusters, which is reasonable because it accumulated the support values of some cluster candidates, making them entitled to be clusters.

While the Join_Clusters heuristic decreases the number of clusters, it actually just enriched the information one cluster can describe. In other words, the Join_Clusters heuristic expresses the original information with fewer clusters.

When using both the heuristics, the number of clusters either increased or decreased, which highly depends on the characters of the log file. Nevertheless, it is certain that these clusters described more information than those generated without using heuristics. To conclude, using both the heuristics is an effective solution against the over-fitting problem, which generates the most information with the fewest clusters.

LogClusterC works faster than LogClusterP. Same as in Section 4.1.2., we again present a runtime statistics summary in Table 7. Relation A is calculated as *the runtime of LogClusterP / the run time of LogClusterC*, which means how many times LogClusterC is faster than LogClusterP. Relation B is calculated as *1- (1 / Relation A)*, which demonstrates how much processing time LogClusterC can save.

In the experiments of "Aggregate_Supports" and "both the heuristics", log file Cray_A consistently yielded much greater Relation A values (around 21.5 and 24.0) than the

---

[1] The raw plain text outputs that contain cluster descriptions and their support values are publicly available in [41].

other six log files, while on another computer[1] log file Cray_A consistently yielded similar Relation A values (around 8.5 (46.5/5.5) and 9.0 (53.0/6.0)) as the other six log files. Therefore, we decided to calculate the average values without counting log file Cray_A's results. Nevertheless, as references, the average values that are calculated with log file Cray_A's results are displayed in small fonts in parentheses.

Table 7. Runtime comparison summary between LogClusterC and LogClusterP (heuristic)

Aggregate_Supports	Relation A	Join_Clusters	Relation A	Both	Relation A
Row 2_A Cray_A	21.50	Row 3_J Cray_A	9.03	Row 4_AJ Cray_A	23.93
Row 6_A Cray_B	6.55	Row 7_J Cray_B	10.26	Row 8_AJ Cray_B	10.81
Row 10_A Cray_C	6.30	Row 11_J Cray_C	10.32	Row 12_AJ Cray_C	10.50
Row 14_A BGL	3.00	Row 15_J BGL	4.99	Row 16_AJ BGL	5.15
Row 18_A LBR	5.23	Row 19_J LBR	8.81	Row 20_AJ LBR	9.13
Row 22_A TDB	4.56	Row 23_J TDB	6.92	Row 24_AJ TDB	7.51
Row 26_A SPT	5.42	Row 27_J SPT	9.48	Row 28_AJ SPT	9.69
Relation A average	**5.18** (7.51)	Relation A average	**8.46** (8.54)	Relation A average	**8.80** (10.96)
Relation B average	**80.69%** (86.68%)	Relation B average	**88.18%** (88.29%)	Relation B average	**88.64%** (90.88%)
Relation A average basic functions					**4.80** (4.82)
Relation B average basic functions					**79.17%** (79.25%)

Compared to the clustering process in basic mode, LogClusterC further reduced the runtime in clustering process with heuristics. As the result, when heuristics are involved, LogClusterC is about about 7.48 times faster than LogClusterP (86% time can be saved).

Besides the horizontal comparison between values from the same table row for LogClusterC and LogClusterP, we also observed an obvious pattern in vertical comparison between rows. We kept experiencing a steady runtime increase when heuristics were introduced to LogClusterC, while there were always runtime bursts

---

[1] OS X operating system 10.10. Intel Core i7 2.2 GHz CPU, 16 GB DDR3 1600 MHz RAM, 256GB PCIe-based flash storage.

when heuristics were introduced to LogClusterP. This phenomenon demonstrates that the heuristic data structures (the prefix tree and the word dependency matrix) were well optimized and utilized in C, while Perl does not support for such optimization.

When processing smaller log files (Cray_A, Cray_B and Cray_C), LogClusterC's memory consumption remains at the same level as LogClusterP's. In vertical comparison, we again observed the pattern that LogClusterP had a burst in memory consumption when the Aggregate_Supports heuristic was introduced, while LogClusterC kept a stable memory consumption level. To provide an overall impression of memory consumption, we express the statistics data when using both the heuristics (Row 4, Row 8, Row 12, Row 16, Row 20, Row 24 and Row 28) in Figure 13.

When processing larger log files (BGL, LBR, TDB and SPT), compared to LogClusterP, LogClusterC has a significantly reduced memory consumption. Regarding the larger log files, as shown in Figure 13, LogClusterC consumed approximately only half of the memory that LogClusterP consumed. As previously mentioned in section 4.1.2., this phenomenon is due to the fact that Perl data structures are much less efficient than similar C-based data structures when large amounts of data need to be stored. Since the maximum memory consumption have reached 1 GB in those cases, using sketch preprocessing is recommended to reduce the unnecessarily waste of memory. Therefore, we keep on holding the conclusion that LogClusterC's memory consumption remains at the same level as LogClusterP's,.
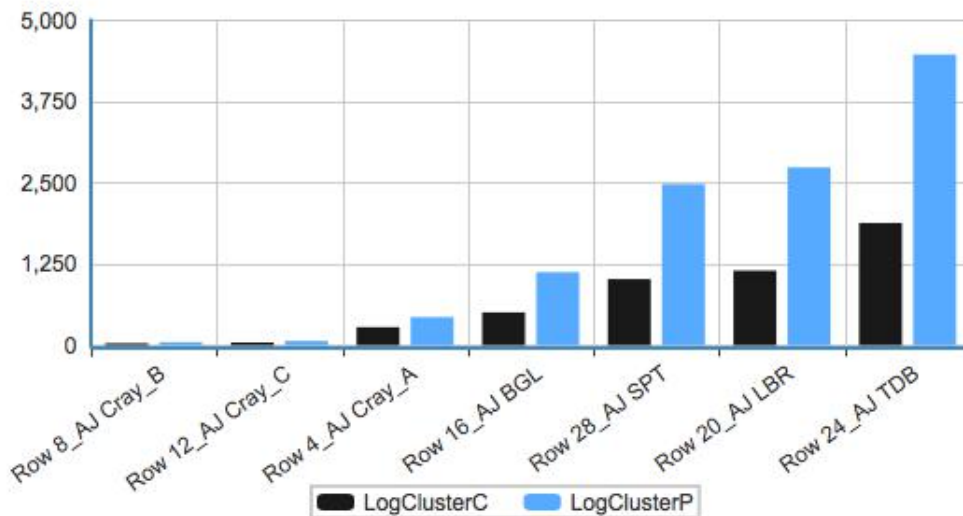

Figure 13. Comparison of memory consumption - 4 (unit: MB)

60

5.   Conclusion and future work

Background study was done in Chapter 2 to explain the importance of event log offline analysis using data mining techniques and the various approaches. The C-based implementation of LogCluster algorithm is the primary contribution, which was described in Chapter 3. In Chapter 4, two comparison experiments were conducted as another contribution, whose results demonstrated LogClusterC's output accuracy and optimized performance. Additionally, Chapter 4 also addressed the open research issue in [2] by comparing the performance of the C-based implementations of LogCluster algorithm and SLCT algorithm. The experiment results demonstrated that, despite being algorithmically more complex, LogCluster is not slower than its predecessor SLCT.

With performance optimization borne in mind, LogClusterC was implemented with many redundant code blocks to achieve a better processing speed. As a single-file project, those redundant code blocks have increased the project management difficulty. It is foreseeable that the scale of the project would expand in the future updates. Therefore, transforming LogClusterC into a multi-file project is considered as future work.

LogClusterC is designed as a single-thread tool, which is not able to fully utilize the computing power of hardware. Although it has significantly reduced the processing time, data mining for large log files is still a time-consuming work. As shown in Section 4.1.1., conducting data mining for a 37.34 GB log file using LogClusterC still costs about 33 minutes. To further reduce the processing time, updating LogClusterC to be multi-threaded is considered as future work.

Another solution of reducing processing time is to migrate LogCluster algorithm into cloud, overcoming the I/O speed bottleneck problem with parallel computing. Since LogCluster algorithm appears to be very compatible with the MapReduce model, this solution could be promising.

There is still an inherent problem of LogCluster algorithm, that the user must provide a support threshold to imply how deeply clustered the event log data set would be. In practice, the user usually has to try several different support thresholds before getting a

relatively satisfying output. For large log files, this kind of attempts could be time-consuming and laborious. Finally, with further optimized processing speed that is fast enough, we can address this problem by intellectually trying different support thresholds in program background. Therefore, instead of asking the user for the support threshold, we can directly ask the user for the desired number of clusters.

Klasterdamis- ja logimustrite kaevandamise algoritmi LogCluster C-põhine realisatsioon

Magistritöö kood ITI70LT

Tudeng: Chen Zhuge

Matrikkli number: 132116IVCMM

Juhendaja: Risto Vaarandi, Ph.D

Resümee

Käesolev magistritöö käsitleb andmekaevandamist sündmuste logide kontekstis ja keskendub klasterdamisalgoritmidele, mis on mõeldud tekstilise sisuga logidest reamustrite leidmiseks. Töö peamine tulemus on algoritmi LogCluster realisatsiooni loomine C keeles. Realisatsioon on GNU GPLv2 litsentsiga vabavaraline UNIXi tööriist ning avalikult kättesaadav [41]. Töö teiseks oluliseks tulemuseks on eksperimentide läbiviimine, mille raames võrreldi loodud tööriista jõudlust teiste samalaadsete tööriistade omaga, mis realiseerivad teisi tekstiliste logide klasterdamise algoritme.

LogCluster on Risto Vaarandi ja Mauno Pihelgase poolt 2015. aastal loodud algoritm [2]. Võrreldes oma eelkäija SLCT algoritmiga suudab LogCluster logisid kõrgema kvaliteediga klasterdada -- algoritmi väljund nõuab vähem järeltöötlust ja on kasutaja jaoks kergemini mõistetav. Kahjuks nõuab LogCluster rohkem arvutiresurssi kui SLCT.

Kuna LogCluster'i algoritmi esialgne realisatsioon loodi Perl'is, jõudsid algoritmi autorid oma artiklis järeldusele, et parima jõudluse saavutamiseks tuleks algoritm realiseerida ka C keeles. Artiklis võrreldi omavahel LogCluster'i ja SLCT Perl'i põhiseid tööriistu ning eksperimente C põhiste lahenduste jõudluse võrdlemiseks nähti edasist uurimistööd nõudva küsimusena [2]. Käesolev magistritöö annab sellele küsimusele vastuse, esitades jõudluse mõõtmiseks läbiviidud eksperimentide kirjelduse ning saadud tulemused.

Magistritöös näidatakse, et loodud C põhine LogCluster'i tööriist on funktsionaalsuse ja klasterdamise tulemuste poolest identne algoritmi autorite pool loodud Perl'i põhise tööriistaga (versioon 0.03), nõudes seejuures tunduvalt vähem mälu ning protsessoriaega. Magistritöös näidatakse samuti, et LogCluster'i ja SLCT C põhised

lahendused töötavad enamvähem ühesuguse kiirusega, kuid LogCluster nõuab algoritmiliste iseärasuste tõttu märgatavalt rohkem mälu.

Praktilisest vaatepunktist on magistritöö tulemusena loodud LogCluster'i algoritmi realiseeriv tööriist vajalik süsteemiadministraatoritele, kes saavad selle abil vähese ajakuluga logisid analüüsida ja leitud reamustrite abil süsteeme profileerida.

References

[1] Risto Vaarandi, "A Data Clustering Algorithm for Mining Patterns From Event Logs," in *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM)*, pp. 119-126, 2003.

[2] Risto Vaarandi and Mauno Pihelgas, "LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs," in *Proceedings of the 11th International Conference on Network and Service Management (CNSM)*, pp. 1-7, 2015.

[3] Risto Vaarandi, "A Breadth-First Algorithm for Mining Frequent Patterns From Event Logs," in *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems*, pp. 293-308, 2004.

[4] Risto Vaarandi, "Mining Event Logs With SLCT and LogHound," in *Proceedings of the 2008 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pp. 1071-1074, 2008.

[5] Risto Vaarandi, "Tools and Techniques for Event Log Analysis," PhD thesis, Tallinn University of Technology, 2005.

[6] Mika Klemettinen, "A Knowledge Discovery Methodology for Telecommunication Network Alarm Databases," PhD thesis, University of Helsinki, 1999.

[7] Qingguo Zheng, Ke Xu, Weifeng Lv and Shilong Ma, "Intelligent Search of Correlated Alarms From Database Containing Noise Data," in *Proceedings of the 2002 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pp. 405-419, 2002.

[8] Sheng Ma and Joseph L. Hellerstein, "Mining Partially Periodic Event Patterns With Unknown Periods," in *Proceedings of the 17th International Conference on Data Engineering*, pp. 205-214, 2001.

[9] James J. Treinen and Ramakrishna Thurimella, "A Framework for the Application of Association Rule Mining in Large Intrusion Detection Infrastructures," in

Proceedings of the 2006 Symposium on Recent Advances in Intrusion Detection (RAID), pp. 1-18, 2006.

[10] Chris Clifton and Gary Gengo, "Developing Custom Intrusion Detection Filters Using Data Mining," in *Proceedings of the 21st Century Military Communications Conference (MILCOM)*, vol. 1, pp. 440-443, 2000.

[11] Thomas Reidemeister, Mohammad A. Munawar and Paul A.S. Ward, "Identifying Symptoms of Recurrent Faults in Log Files of Distributed Information Systems," in *Proceedings of the 2010 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pp. 187-194, 2010.

[12] Thomas Reidemeister, Miao Jiang, and Paul A.S. Ward, "Mining Unstructured Log Files for Recurrent Fault Diagnosis," in *Proceedings of the 12th IEEE/IFIP International Symposium on Integrated Network Management (IM 2011) and Workshops*, pp. 377-384, 2011.

[13] Thomas Reidemeister, "Fault Diagnosis in Enterprise Software Systems Using Discrete Monitoring Data," PhD thesis, University of Waterloo, 2012.

[14] Wei Xu, Ling Huang, Armando Fox, David A. Patterson and Michael I. Jordan, "Mining Console Logs for Large-Scale System Problem Detection," in *Proceedings of the 3rd Workshop on Tackling Computer Systems Problems With Machine Learning Techniques*, 2008.

[15] Daniela Brauckhoff, Xenofontas Dimitropoulos, Arno Wagner and Kavè Salamatian, "Anomaly Extraction in Backbone Networks Using Association Rules," *Journal of IEEE/ACM Transactions on Networking*, vol. 20, no. 6, pp. 1788-1799, 2012.

[16] Jon Stearley, "Towards Informatic Analysis of Syslogs," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pp. 309-318, 2004.

[17] Adetokunbo Makanju, Stephen Brooks, A. Nur Zincir-Heywood and Evangelos E. Milios, "Logview: Visualizing Event Log Clusters," in *Proceedings of the 6th Annual Conference on Privacy, Security and Trust*, pp. 99-108, 2008.

[18] Eduard Glatz, Stelios Mavromatidis, Bernhard Ager and Xenofontas Dimitropoulos, "Visualizing Big Network Traffic Data Using Frequent Pattern Mining and Hypergraphs," *Journal of Computing*, vol. 96, no. 1, pp. 27-38, 2014.

[19] Heikki Mannila, Hannu Toivonen and A. Inkeri Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Journal of Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259-289, 1997.

[20] Mika Klemettinen, Heikki Mannila and Hannu Toivonen, "Rule Discovery in Telecommunication Alarm Data," *Journal of Network and Systems Management*, vol. 7, no. 4, pp. 395-423, 1999.

[21] L. Burns, J. L. Hellerstein, S. Ma, C.S. Perng, D.A. Rabenhorst and D. Taylor, "A Systematic Approach to Discovering Correlation Rules for Event Management," in *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 345-359, 2001.

[22] Risto Vaarandi and Kārlis Podiņš, "Network IDS Alert Classification With Frequent Itemset Mining and Data Clustering," in *Proceedings of the 2010 International Conference on Network and Service Management (CNSM)*, pp. 451-456, 2010.

[23] Adetokunbo Makanju, A. Nur Zincir-Heywood and Evangelos E. Milios, "A Lightweight Algorithm for Message Type Extraction in System Application Logs," *Journal of IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921-1936, 2012.

[24] Adetokunbo Makanju. "Exploring Event Log Analysis With Minimum Apriori Information," PhD thesis, Dalhousie University, 2012.

[25] Adetokunbo Makanju, A. Nur Zincir-Heywood and Evangelos E. Milios, "Clustering Event Logs Using Iterative Partitioning," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1255-1264, 2009.

[26] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, vol. 1215, pp. 487-499, 1994.

[27] Xia Ning, Geoff Jiang, Haifeng Chen and Kenji Yoshihira, "HLAer: A System for Heterogeneous Log Analysis," *Technical Report 2013-TR097 NEC Labs America*, 2013.

[28] Chinghway Lim, Navjot Singh and Shalini Yajnik, "A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems," in *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 398-403, 2008.

[29] Kenji Yamanishi and Yuko Maruyama, "Dynamic Syslog Mining for Network Failure Monitoring," in *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pp. 499-508, 2005.

[30] Jan Vitek and Tomas Kalibera. "Repeatability, Reproducibility and Rigor in Systems Research," in *Proceedings of the 2011 International Conference on Embedded Software (EMSOFT)*, pp. 33-38, 2011.

[31] Sisyphus Log Data Mining Toolkit, http://www.cs.sandia.gov/sisyphus/

[32] Manpage of slct, http://ristov.github.io/slct/slct.html

[33] Heikki Mannila, Hannu Toivonen and A. Inkeri Verkamo, "Discovering Frequent Episodes in Sequences Extended Abstract," in *Proceedings of the 1st Conference on Knowledge Discovery and Data Mining*, pp. 210-215, 1995.

[34] M.V. Ramakrishna and Justin Zobel, "Performance in Practice of String Hashing Functions," in *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, pp. 215-224, 1997.

[35] Justin Zobel, Steffen Heinz and Hugh E. Williams, "In-Memory Hash Tables for Accumulating Text Vocabularies," *Journal of Information Processing Letters*, vol. 80, no. 6, pp. 271-277, 2001.

[36] Manpage of loghound, http://ristov.github.io/loghound/loghound.html

[37] RFC 3164 - The BSD Syslog Protocol, https://tools.ietf.org/html/rfc3164

[38] Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu, "A Density-based Algorithm for Discovering Clusters in Large Spatial Databases With Noise," *Journal of Kdd*, vol. 96, no. 34, pp. 226-231, 1996.

[39] The Computer Failure Data Repository (CFDR) | USENIX, https://www.usenix.org/cfdr

[40] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel and Jörg Sander, "OPTICS: Ordering Points to Identify the Clustering Structure," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pp. 49-60, 1999.

[41] zhugehq/thesis-project-logclusterc, https://github.com/zhugehq/thesis-project-logclusterc

[42] Releases ristov/logcluster, https://github.com/ristov/logcluster/releases

[43] ristov.github.io/slct, http://ristov.github.io/slct/

[44] time (1) - Linux manual page, http://man7.org/linux/man-pages/man1/time.1.html

Appendix 1. The performance of the word sketch preprocessing

Appendix 1.1. Getting vocabulary of log file BGL (708.76 MB) without word sketch preprocessing.

Thu Mar 31 18:50:51 2016: File BGL is added
Thu Mar 31 18:50:51 2016: Starting...
Thu Mar 31 18:50:51 2016: Creating vocabulary...
Thu Mar 31 18:51:20 2016: 5,632,912 words were inserted into the vocabulary.
Thu Mar 31 18:51:20 2016: Finding frequent words from vocabulary...
Thu Mar 31 18:51:21 2016: 1,878 frequent words were found.
Thu Mar 31 18:51:21 2016: 91% - 5,173,591 words in vocabulary occur 1 time.
Thu Mar 31 18:51:21 2016: 92% - 5,222,663 words in vocabulary occur 2 times or less.
Thu Mar 31 18:51:21 2016: 94% - 5,298,390 words in vocabulary occur 5 times or less.
Thu Mar 31 18:51:21 2016: 96% - 5,424,315 words in vocabulary occur 10 times or less.
Thu Mar 31 18:51:21 2016: 97% - 5,470,353 words in vocabulary occur 20 times or less.
Thu Mar 31 18:51:21 2016: 99.97% - 5,631,034 words in vocabulary occur less than 474(support) times.
Command line: ./voca --input=BGL --support=0.01%
Runtime: 29.99
CPU time system: 0.28
CPU time user: 29.72
Memory consumption: 517436

Appendix 1.2. Getting vocabulary of log file BGL (708.76 MB) with word sketch preprocessing.

Thu Mar 31 18:52:28 2016: File BGL is added
Thu Mar 31 18:52:28 2016: Starting...
Thu Mar 31 18:52:28 2016: Creating the word sketch...
Thu Mar 31 18:52:39 2016: 3,980 slots in the word sketch >= support threshhold
Thu Mar 31 18:52:39 2016: Creating vocabulary...
Thu Mar 31 18:52:53 2016: 26,495 words were inserted into the vocabulary.
Thu Mar 31 18:52:53 2016: Finding frequent words from vocabulary...
Thu Mar 31 18:52:53 2016: 1,878 frequent words were found.
Thu Mar 31 18:52:53 2016: 77% - 20,590 words in vocabulary occur 1 time.
Thu Mar 31 18:52:53 2016: 78% - 20,785 words in vocabulary occur 2 times or less.
Thu Mar 31 18:52:53 2016: 79% - 21,084 words in vocabulary occur 5 times or less.
Thu Mar 31 18:52:53 2016: 81% - 21,560 words in vocabulary occur 10 times or less.
Thu Mar 31 18:52:53 2016: 82% - 21,742 words in vocabulary occur 20 times or less.
Thu Mar 31 18:52:53 2016: 92.91% - 24,617 words in vocabulary occur less than 474(support) times.
Command line: ./voca --input=BGL --support=0.01% --wsize=1000000
Runtime: 24.36

CPU time system: 0.22
CPU time user: 24.15
Memory consumption: 13128

Appendix 1.3. Getting vocabulary of log file TDB (29.67 GB) without word sketch preprocessing.

Thu Mar 31 18:11:58 2016: File TDB is added
Thu Mar 31 18:11:58 2016: Starting...
Thu Mar 31 18:11:58 2016: Creating vocabulary...
Thu Mar 31 18:27:24 2016: 23,330,821 words were inserted into the vocabulary.
Thu Mar 31 18:27:24 2016: Finding frequent words from vocabulary...
Thu Mar 31 18:27:27 2016: 9,938 frequent words were found.
Thu Mar 31 18:27:27 2016: 32% - 7,561,438 words in vocabulary occur 1 time.
Thu Mar 31 18:27:27 2016: 47% - 11,110,168 words in vocabulary occur 2 times or less.
Thu Mar 31 18:27:27 2016: 72% - 17,011,865 words in vocabulary occur 5 times or less.
Thu Mar 31 18:27:27 2016: 86% - 20,268,254 words in vocabulary occur 10 times or less.
Thu Mar 31 18:27:27 2016: 92% - 21,536,130 words in vocabulary occur 20 times or less.
Thu Mar 31 18:27:27 2016: 99.96% - 23,320,883 words in vocabulary occur less than 21121(support) times.
Command line: ./voca --input=TDB --support=0.01%
Runtime: 929.14
CPU time system: 7.96
CPU time user: 920.78
Memory consumption: 1926316

Appendix 1.4. Getting vocabulary of log file TDB (29.67 GB) with word sketch preprocessing.

1 18:29:45 2016: File TDB is added
Thu Mar 31 18:29:45 2016: Starting...
Thu Mar 31 18:29:45 2016: Creating the word sketch...
Thu Mar 31 18:37:59 2016: 10,172 slots in the word sketch >= support threshhold
Thu Mar 31 18:37:59 2016: Creating vocabulary...
Thu Mar 31 18:49:30 2016: 247,654 words were inserted into the vocabulary.
Thu Mar 31 18:49:30 2016: Finding frequent words from vocabulary...
Thu Mar 31 18:49:30 2016: 9,938 frequent words were found.
Thu Mar 31 18:49:30 2016: 31% - 77,012 words in vocabulary occur 1 time.
Thu Mar 31 18:49:30 2016: 45% - 113,026 words in vocabulary occur 2 times or less.
Thu Mar 31 18:49:30 2016: 69% - 173,235 words in vocabulary occur 5 times or less.
Thu Mar 31 18:49:30 2016: 83% - 205,993 words in vocabulary occur 10 times or less.
Thu Mar 31 18:49:30 2016: 88% - 218,718 words in vocabulary occur 20 times or less.
Thu Mar 31 18:49:30 2016: 95.99% - 237,716 words in vocabulary occur less than 21121(support) times.
Command line: ./voca --input=TDB --support=0.01% --wsize=1000000
Runtime: 1185.10
CPU time system: 13.02

71

CPU time user: 1171.59
Memory consumption: 32884

Appendix 2. The extra runtime comparison experiment between the C-based implementations of LogCluster and SLCT with faster I/O speed

The experiments were conducted on a computer running Ubuntu Server 14.04.[1] The hardware configuration was: Intel i7-4720 HQ 2.6 GHz CPU, Kingston 16 GB DDR3 1600 MHz RAM, Samsung 850 EVO 250 GB SSD hard disk. As this experiment did not yield results that is much different from the experiment in Section 4.1., we attached the results here in Appendix 3.

Same as the experiment in Section 4.1., the figures in the table below were the average values calculated from five times' repeated experiments. Conducting the experiment for one time costs approximately 11 hours on this computer. The computer was shut down and rebooted after each time's experiment.

To support the experiment principle of reproducibility and rigor, the bash script to run the experiment and the raw plain text outputs in each experiment are available in [41].

Row # Log file	Runtime Origin		Runtime SSD		CPU time system CPU time user SSD	
	LCC	SLCT	LCC	SLCT	LCC	SLCT
Row 1 Cray_A	1.43	1.29	1.25	1.22	0.06 1.19	0.00 1.22
Row 2 Cray_A	1.26	1.26	1.26	1.26	0.06 1.19	0.01 1.25
Row 3 Cray_A	1.47	1.34	1.49	1.34	0.12 1.36	0.01 1.32
Row 4 Cray_B	2.82	2.94	2.72	2.91	0.02 2.69	0.01 2.89
Row 5 Cray_B	3.54	2.98	3.54	2.99	0.29 3.25	0.01 2.97

---

[1] This was a minimum installation of Ubuntu Server 14.04, only gcc 4.8 package was installed besides the kernel. The popular software, such like OpenSSH server, DNS server, LAMP server, were not installed.

Row # Log file	Runtime Origin		Runtime SSD		CPU time system CPU time user SSD	
	LCC	SLCT	LCC	SLCT	LCC	SLCT
Row 6 Cray_B	4.08	3.38	4.08	3.39	0.37 3.70	0.05 3.34
Row 7 Cray_C	9.13	9.61	8.96	9.67	0.06 8.91	0.04 9.63
Row 8 Cray_C	11.79	10.85	12.04	10.82	0.86 11.19	0.10 10.72
Row 9 Cray_C	15.40	13.18	15.63	13.07	1.26 12.92	0.17 11.45
Row 10 BGL	44.25	48.36	43.90	47.34	0.42 43.49	0.36 46.99
Row 11 BGL	45.36	47.89	44.95	47.05	0.33 44.63	0.36 46.71
Row 12 BGL	48.05	51.76	46.74	49.79	0.72 45.06	0.37 48.45
Row 13 LBR	1,467.49	1,552.10	1,448.60	1,573.80	13.39 1,436.00	13.83 1,560.75
Row 14 LBR	1,501.50	1,593.84	1,487.32	1,583.96	13.87 1,474.25	13.92 1,570.92
Row 15* LBR	2,822.85	3,149.36	2,947.82	3,083.26	28.62 2,789.87	27.82 3,041.43
Row 16 TDB	1,582.64	1,641.37	1,557.91	1,601.98	14.76 1,543.99	15.06 1,587.82
Row 17 TDB	1,603.16	1,670.34	1,578.19	1,636.38	15.69 1,563.31	14.86 1,622.37
Row 18* TDB	2,796.66	3,031.52	2,903.61	2,995.29	30.37 2,743.49	29.63 2,961.20
Row 19 SPT	1,800.08	1,885.33	1,811.46	1,876.69	17.51 1,794.96	17.65 1,860.15
Row 20 SPT	1,832.82	1,919.34	1,808.14	1,892.01	17.50 1,791.45	17.65 1,875.37
Row 21* SPT	4,178.85	4,400.34	4,333.37	4,393.98	36.56 3,669.06	36.39 3,890.07