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

ITI70LT

Author: Zhuge Chen

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 material obtained from other sources has 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 the implementation through number of benchmarks and comparison with other similar algorithms.

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 resources.

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 Perl-based implementations of LogCluster and SLCT, while detailed performance assessment of C-based versions has been identified as future work [2]. The current thesis addresses this open research issue by presenting the C-based version of LogCluster and describing a set of experiments for assessing its performance.

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. This thesis also demonstrates that the C-based implementation of 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 and build system profiles would benefit from this fast and efficient C-based implementation.

## Table of abbreviations and terms

| | |
|---|---|
| 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 three |

| | |
|---|---|
| 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 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.

All functions that are supported by LogClusterP (version 0.03) are supported by LogClusterC. While we followed the design principle that LogClusterP and LogClusterC will generate the same outputs if they are given the same input, there are some new features added to LogClusterC without inferring this design principle, such as real-time debug mode that shows the status of the data mining process, and variable output styles.

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. Since many of the new features are specific to Perl and are cumbersome to implement in C, LogClusterC will be updated independently from the experimental releases of LogClusterP.

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

The experiments were designed to satisfy the following conditions [30]:

- Repeatability – Tests can be run again when needed;

[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, the term "SLCT" means both the algorithm and the C-based implementation. This convention applies to the rest of this thesis.

- Reproducibility – The tools used in the experiments (LogClusterC, LogClusterP and SLCT) are publicly available under GNU GPLv2 license. Log files in the experiments are selected from public data repository [39]. Command lines in the experiments are provided.
- Rigor – Hardware configuration in the experiments is provided. Experiment data is accurate and obtained using the time utility of Linux [42].

## 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 and other event log data mining solutions.

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, also known as sequential mining of event types, 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 sequential patterns of event types. For example, event type "*link down*" is followed by event type "*link up*" within 10 seconds, 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 [26] and FP-Growth) for detecting associations. Apriori is the most commonly used algorithm which harnesses iterative approach for detecting frequent itemsets. 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. 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 valuable for designing regular expression signatures, constructing event correlation rules, building system profiles, and detecting system anomalies.

[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.

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 larger 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 has been 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 Krlis Podiš [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 =32*" are generated as three distinct clusters instead of one, despite only the words after word "*ambien*" are different.

[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.

### 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, like ignorance of wildcards in the end of line pattern, 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's 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 and LogHound, 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, 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 SLCT, LogClusterP and LogClusterC are summarized in Table 1.

*Table 1. General differences between SLCT, LogClusterP and LogClusterC*[1]

| | **SLCT (version 0.05)** | **LogClusterP (version 0.03)** | **LogClusterC (version 0.03)** | **LogClusterP (version 0.08)** |
|---|---|---|---|---|
| **Language** | C | Perl | C | Perl |
| **Lines of code** | 1691 | 1220 | 6339 | 1457 |
| **Size of source file** | 40KB | 37KB | 190KB | 45KB |
| **Supported options** | 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.

[1] Though trying to be as precise as possible, the statistics data in Table 1 are largely influenced by personal coding style.

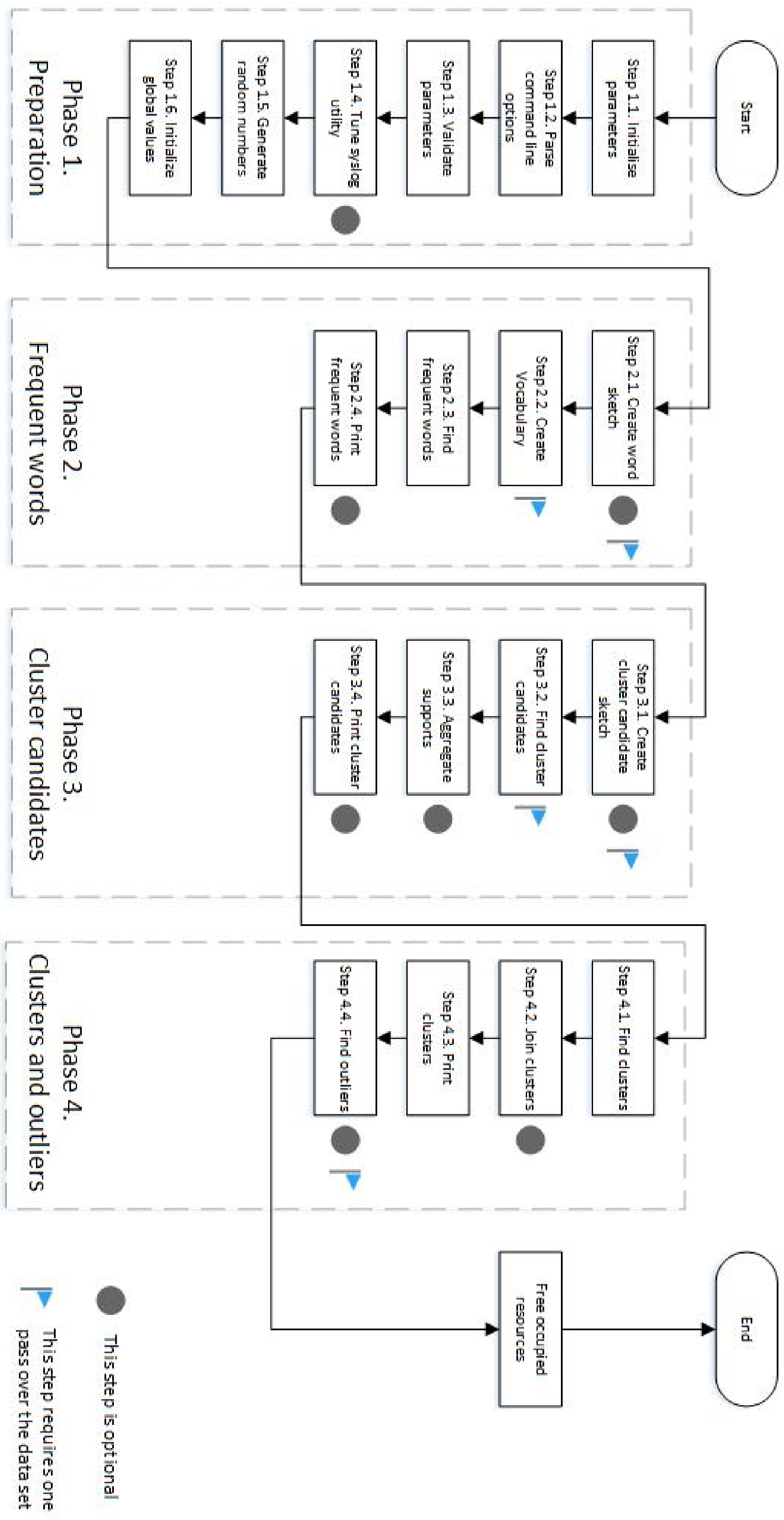

Start
Phase 1. Preparation
Step 1.1. Initialise parameters
Step 1.2. Parse command line options
Step 1.3. Validate parameters
Step 1.4. Tune syslog utility
Step 1.5. Generate random numbers
Step 1.6. Initialize global values
Phase 2. Frequent words
Step 2.1. Create word sketch
Step 2.2. Create Vocabulary
Step 2.3. Find frequent words
Step 2.4. Print frequent words
Phase 3. Cluster candidates
Step 3.1. Create cluster candidate sketch
Step 3.2. Find cluster candidates
Step 3.3. Aggregate supports
Step 3.4. Print cluster candidates
Phase 4. Clusters and outliers
Step 4.1. Find clusters
Step 4.2. Join clusters
Step 4.3. Print clusters
Step 4.4. Find outliers
Free occupied resources
End
This step is optional
This step requires one pass over the data set

*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 1.

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 constant 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 1. 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 feature [35]. Details of the move-to-front feature 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 LogCluster instead of "*0001Interface*" in SLCT). The word's location is now recorded by a utility named "wildcard recorder", which memorizes the minimum 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 is more flexible, which keeps adjusting the existing cluster candidate descriptions according to new emerging cluster candidates. For example, assume there are two event log lines: "*Interface eth0 down*" and "*Interface HQ Link down*", and "*Interface*" and "*down*" are frequent words[2]. In SLCT, there will

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

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

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: ignorance of the wildcards in the end of line pattern. 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 as "*Interface *{1,2} down *{0,1}*".[1]

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 description of all 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].

### 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.

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

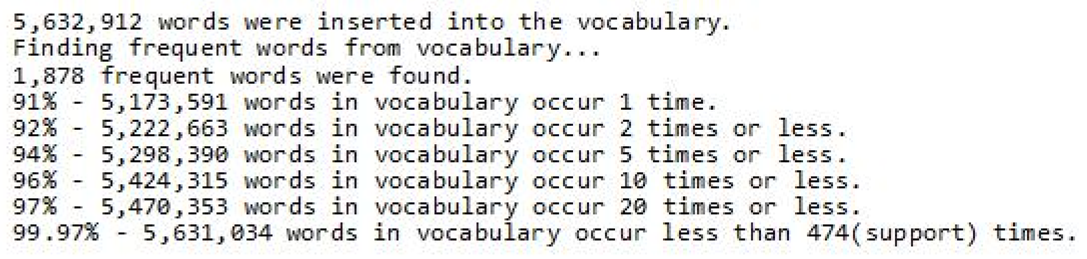
Risto Vaarandi conducted an experiment for estimating the occurrence times of words in log file data [1]. Results indicated that a majority of words were very infrequent, and a significant fraction of words appeared just once in the data set.

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.

### 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

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

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 is not efficient, especially in scenarios where one slot stores hundreds of distinct words.

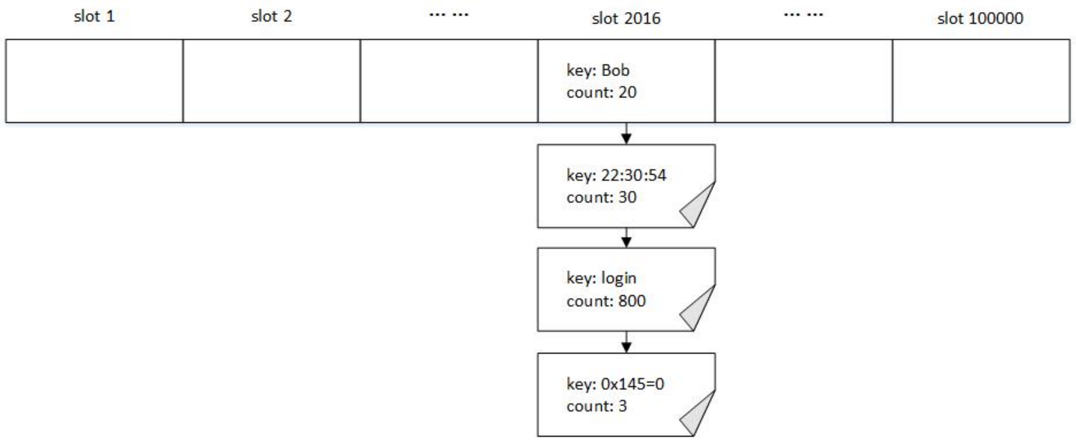


*Figure 3. The word table's data structure*

[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 same hash value in this session, will not necessarily be hashed to the same 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.

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 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.

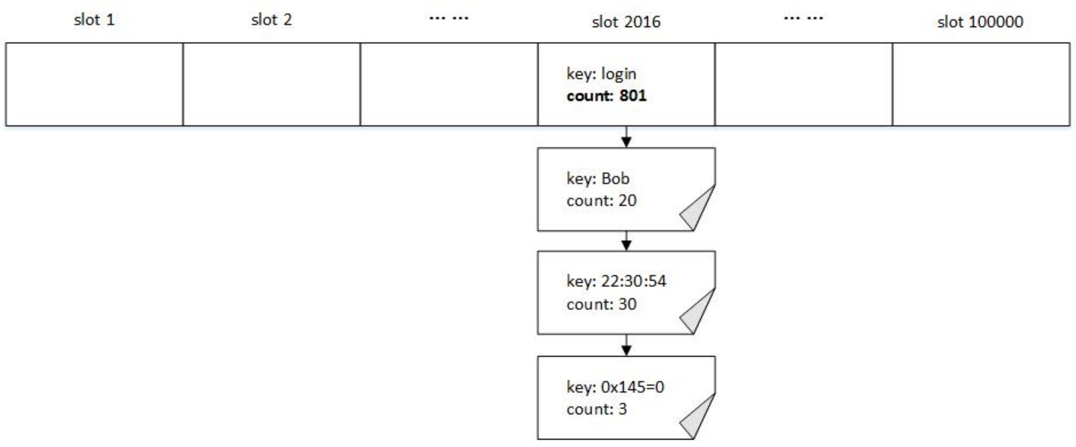


*Figure 4. The word table's data structure with move-to-front technique*

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 locations 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 (version 0.03), 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 resources. 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 if the words are uniformly distributed over the 1,000,000 sketch counters with a hashing function. In the following we assume that some of the words have the properties as shown in Table 2.

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

*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 with the Shift-Add-Xor string hashing algorithm, 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.

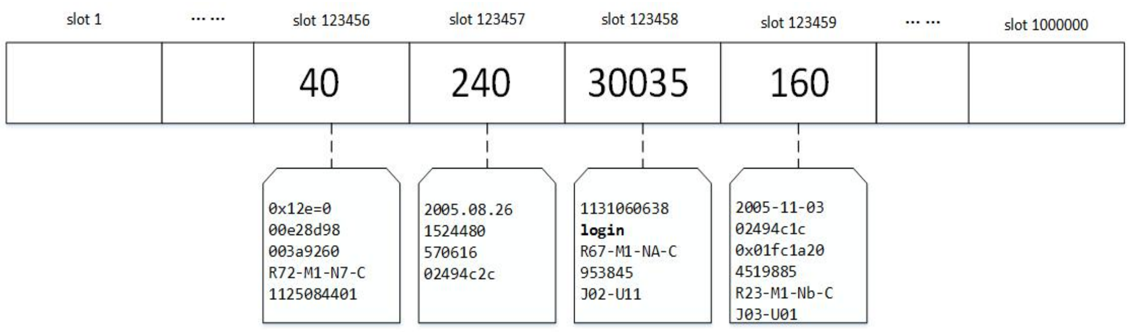


*Figure 5. The completed word sketch*

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”. Though the sketch preprocessing can significantly reduce memory consumption, an obvious trade-off is that it requires one extra pass over the data set, which increases processing time.

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 as below, and the output is in Appendix 1.1. The format explanation of time utility can be found in [42].

```
/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 result 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

[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 from the main function.

decreased from 29.99 seconds to 24.36 seconds.[1] This can be explained with the fact 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 commands 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.[2] 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] Different hardware configurations can get different results in this experiment. The hardware configuration we used is described in Chapter 4.

[2] 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.

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

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.

```
User bob login from 10.1.1.1, support 5
User *{1,1} login from 10.1.1.1, support 10
User *{1,1} login from *{1,1}, support 100
```

For cluster candidate "*User *{1,1} login from *{1,1}*", cluster candidates "*User *{1,1} login from 10.1.1.1*" and "*User bob login from 10.1.1.1*" are more specific cluster candidates. Similarly, for cluster candidate "*User *{1,1} login from 10.1.1.1*", cluster candidate "*User bob login from 10.1.1.1*" is more specific. Therefore, after the

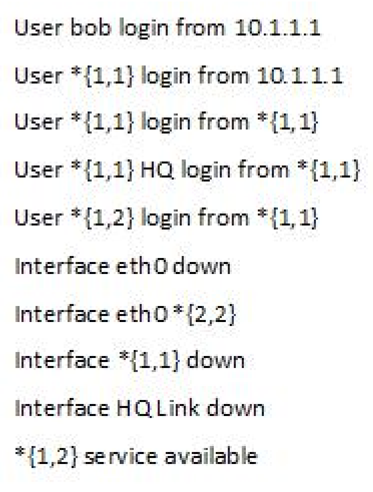
Aggregate_Supports heuristics, the status of these three cluster candidates would be changed to the states as shown below.

```
User bob login from 10.1.1.1, support 5
User *{1,1} login from 10.1.1.1, support 15
User *{1,1} login from *{1,1}, support 115
```

Support accumulation involves some subtle technical issues which are further described in Appendix 2.

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 "*User *{1,1} login from *{1,1}*" and "*User *{1,1} HQ login from *{1,1}*" are both considered more specific than cluster candidate "*User *{1,2} login from *{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.

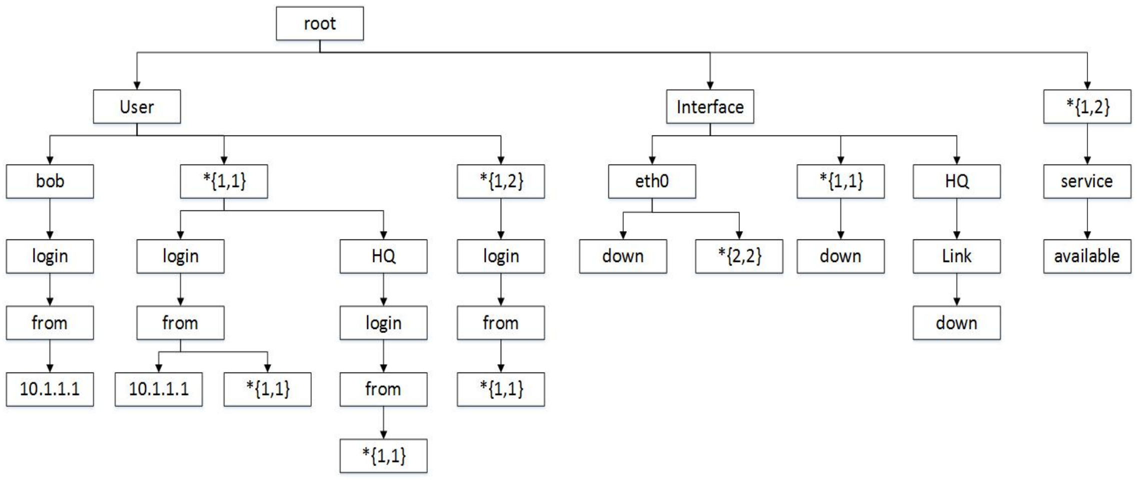


*Figure 7. The prefix tree data structure of the cluster candidates set*

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, while words "*Interface*" and "*down*" have a constant nature. A straightforward solution could be presenting these 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 a solution for this issue, the Join_Clusters heuristic is introduced by LogCluster [2]. The Join_Clusters heuristic (Step 4.2.) is applied after the clusters are found from cluster candidates (see Figure 1). 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 cluster descriptions 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 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 $w_i$.

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

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.

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

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 $w$.

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

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

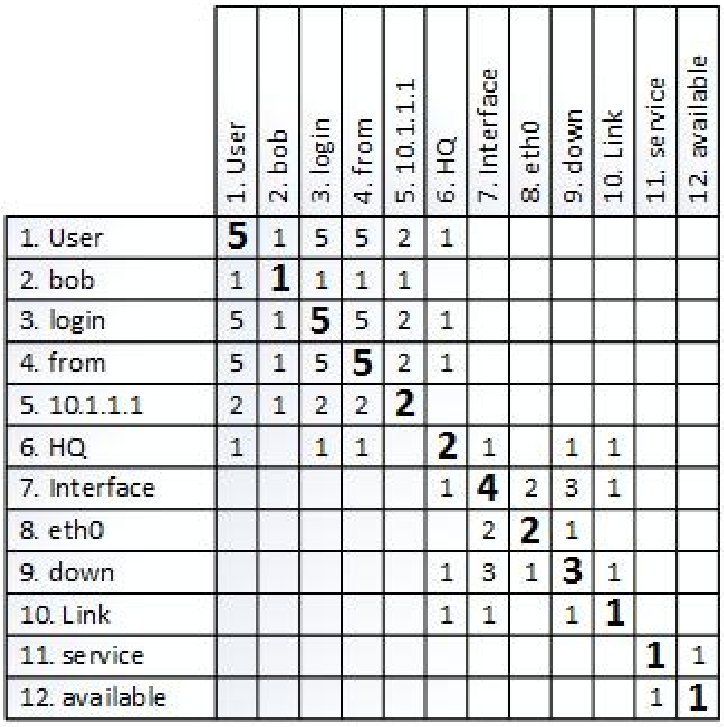
Note that $dep(w_i, w_i) = 1$ and $0 < dep(w_j, w_i) \leq 1$ [1]. Therefore, $1/k \leq weight(w_i) \leq 1$ in word weight function 1, and $0 < weight(w_i) \leq 1$ in word weight function 2.

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. Note that LogClusterC drops duplicate frequent words before updating the word dependency matrix, since one event log line could contain the same frequent word more than one time.

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 $\left|I_{w_j} \cap I_{w_i}\right| > 0$.



| | 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*", it must 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*":

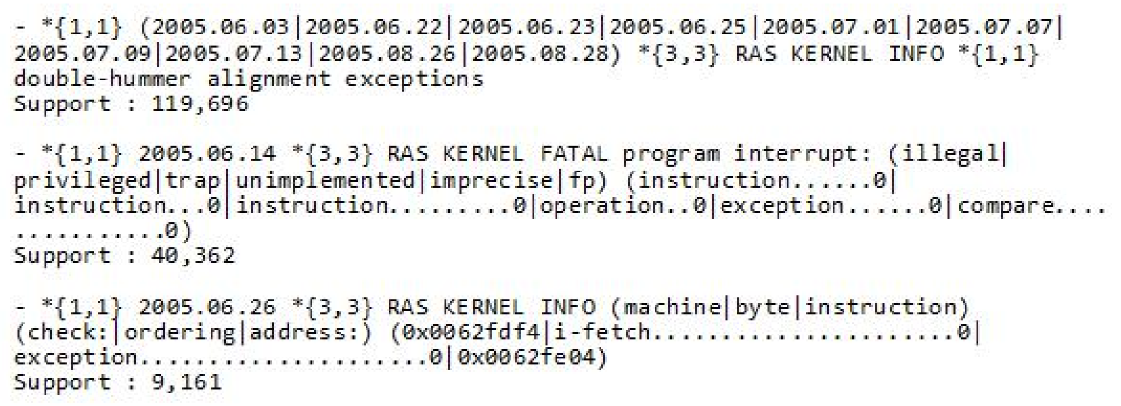
$$
\begin{aligned}
weight(bob) &= \begin{pmatrix} dep(User, bob) + dep(bob, bob) + dep(\log in, bob) + \\ dep(from, bob) + dep(10.1.1.1, bob) \end{pmatrix} / 5 \\
&= (1/5 + 1/1 + 1/5 + 1/5 + 1/5)/5 = 9/25 \\
&= 0.36
\end{aligned}
$$

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 Join_Clusters performs in practice, we applied it to log file BGL with a word weight threshold of 0.4, using the command 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*

## 3.5. Event log line preprocessing

Quite often, raw event logs need preprocessing before clustering to remove irrelevant fields from events, or to convert the format of events. In order to facilitate fast event log preprocessing, LogClusterC provides three event log line preprocessing options:

byte_offset, line_filter, and word_class. The tasks of byte_offset and line_filter are pure text manipulation, which could also be done by using third party text file processing tools, whereas the task of word_class is strongly related to the clustering process.

### 3.5.1. The byte_offset option

The feature described in this section can be turned on by using '--byteoffset=<byte offset>' option.

Since the popular event log protocols usually start an event log line with time stamp and host name, sometimes we need to sanitize this kind of irrelevant information for a more effective clustering. With byte_offset option turned on, LogClusterC ignores the first <byte offset> bytes of every line during clustering. The default value for this option is zero, i.e., no bytes are ignored.

### 3.5.2. The line_filter option

The feature described in this section can be turned on by using '--lfilter=<line filter regular expression>' option. Option '--template=<line conversion template>' can be used for additional tuning. With the line_filter option turned on, LogClusterC ignores all event log lines that do not match the regular expression.[1]

The <line filter regular expression> can contain capture groups, and when '--template=<line conversion template>' option has been given, the match variables (like \$1 or \$2) set by capture groups can be used in the template string <line conversion template>. After variables have been substituted in the template string, the resulting string will replace the original event log line. For example, if we use the command line as shown below, the event log line "*sshd[1344]: connect from 192.168.1.1*" will be converted to "*connect from 192.168.1.1*" as the result [32].

```
./logclusterc --input=sample.log --support=200 --lfilter='sshd\[[0-
9]+\]:(.+)' --template='$1'
```

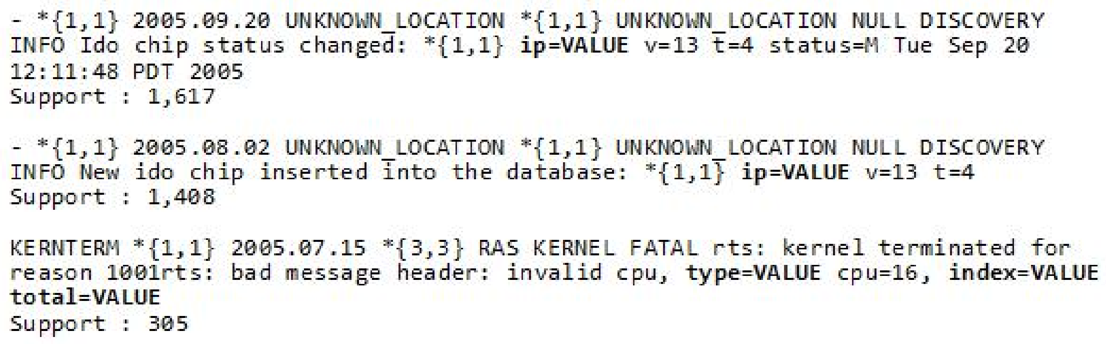
[1] When line_filter and byte_offset options are used together, byte_offset option is applied first.

### 3.5.3. The word_class option

The feature described in this section can be turned on by using '--wfilter=<word filter regular expression>', '--wsearch=<word search regular expression>', and 'wreplace=<word replace string>' options. These options fully utilize the support for regular expression, providing advanced preprocessing. This preprocessing allows to convert many infrequent words with the same format to one word (so called "word class"). If word class is detected as frequent, it acts as a representative for many infrequent words in cluster descriptions (please see [2] for a more detailed discussion). Two examples are given to demonstrate the usage of word classes.

To locate event log lines that contain the equal sign (=), and replace the characters after the equal sign with the string "*VALUE*", we use the below command line. Some of the most representative clusters are listed in Figure 10, with bold font emphasizing word classes in detected cluster descriptions.

```
./logclusterc --input=BGL --support=200 --wfilter='=' --wsearch='=.+' --
wreplace='=VALUE'
```

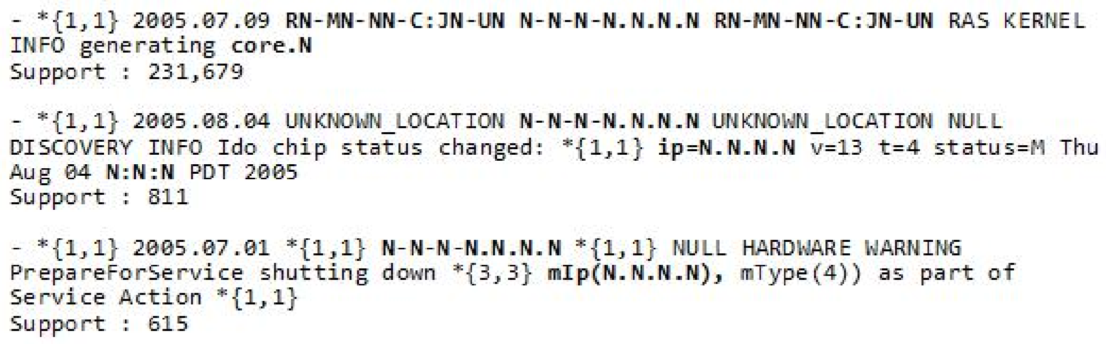


```
- *{1,1} 2005.09.20 UNKNOWN_LOCATION *{1,1} UNKNOWN_LOCATION NULL DISCOVERY
INFO Ido chip status changed: *{1,1} ip=VALUE v=13 t=4 status=M Tue Sep 20
12:11:48 PDT 2005
Support : 1,617

- *{1,1} 2005.08.02 UNKNOWN_LOCATION *{1,1} UNKNOWN_LOCATION NULL DISCOVERY
INFO New ido chip inserted into the database: *{1,1} ip=VALUE v=13 t=4
Support : 1,408

KERNTERM *{1,1} 2005.07.15 *{3,3} RAS KERNEL FATAL rts: kernel terminated for
reason 1001rts: bad message header: invalid cpu, type=VALUE cpu=16, index=VALUE
total=VALUE
Support : 305
```

*Figure 10. The most representative clusters when using word_class option - 1*

To blur the numbers in IP addresses, dates, times and serials, we locate event log lines that contain the dot sign (.) or colon sign (:), and replace the numbers with the capital letter "*N*", using the below command line. Some of the most representative clusters are listed in Figure 11, with bold font emphasizing the effect of using word classes.

```
./logclusterc --input=BGL --support=200 --wfilter='[.:]' --wsearch='[0-9]+'
--wreplace='N'
```



```
- *{1,1} 2005.07.09 RN-MN-NN-C:JN-UN N-N-N-N.N.N.N RN-MN-NN-C:JN-UN RAS KERNEL
INFO generating core.N
Support : 231,679

- *{1,1} 2005.08.04 UNKNOWN_LOCATION N-N-N-N.N.N.N UNKNOWN_LOCATION NULL
DISCOVERY INFO Ido chip status changed: *{1,1} ip=N.N.N.N v=13 t=4 status=M Thu
Aug 04 N:N:N PDT 2005
Support : 811

- *{1,1} 2005.07.01 *{1,1} N-N-N-N.N.N.N *{1,1} NULL HARDWARE WARNING
PrepareForService shutting down *{3,3} mIp(N.N.N.N), mType(4)) as part of
Service Action *{1,1}
Support : 615
```

*Figure 11. The most representative clusters when using word_class option - 2*

## 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.

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

### 4.1.1. Experiment introduction

This section compares the performance of algorithms 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 on clustering algorithms. The last three log files are around 30 GB by their size, representing computationally more demanding clustering scenarios.

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

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.

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

The constant support 200 was selected for testing the efficiency of memory-resident data structures used by the algorithms. Applying this small support threshold will generate a large amount of frequent words and cluster candidates. Thus, the memory consumption and workload of the algorithms 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 [42]). The commands 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 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 2 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 commands used in these three Rows are listed below, taking Row 15 as the example.

```
/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
```

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

The results of the experiments 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 [42].

[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 #<br>Log file | Event log size in megabytes | Event log size in lines | Support threshold | Run time in seconds<br>CPU time system<br>CPU time user | | | Memory consumption in megabytes | | | Number of clusters<br>Number of cluster candidates | | | Number of frequent words<br>Number of words in vocabulary | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | LCC | LCP | SLCT | LCC | LCP | SLCT | LCC | LCP | SLCT | LCC | LCP | SLCT |
| Row 1<br>Cray_A | 20.86 | 379,457 | 1,897 | 1.52 | 6.70 | 1.37 | 270.24 | 218.40 | 24.16 | 0 | 0 | 0 | 116 | 116 | 150 |
| | | | | 0.08<br>1.37 | 0.05<br>6.64 | 0.02<br>1.34 | | | | 63,876 | 63,876 | 53,615 | 13,737 | 13,737 | 19,144 |
| Row 2<br>Cray_A | 20.86 | 379,457 | 379 | 1.41 | 7.25 | 1.42 | 344.22 | 272.46 | 34.22 | 27 | 27 | 29 | 347 | 347 | 420 |
| | | | | 0.12<br>1.28 | 0.12<br>7.13 | 0.05<br>1.36 | | | | 81,044 | 81,044 | 79,280 | 13,737 | 13,737 | 19,144 |
| Row 3<br>Cray_A | 20.86 | 379,457 | 200 | 1.59 | 8.23 | 1.49 | 640.02 | 527.91 | 64.07 | 13 | 13 | 16 | 1,051 | 1,051 | 1,146 |
| | | | | 0.12<br>1.47 | 0.15<br>8.07 | 0.04<br>1.45 | | | | 151,707 | 151,707 | 150,299 | 13,737 | 13,737 | 19,144 |
| Row 4<br>Cray_B | 52.12 | 958,075 | 4,790 | 2.91 | 16.65 | 3.07 | 30.51 | 26.71 | 5.49 | 32 | 32 | 32 | 77 | 77 | 78 |
| | | | | 0.07<br>2.82 | 0.06<br>16.60 | 0.04<br>3.02 | | | | 5,700 | 5,700 | 910 | 4,853 | 4,853 | 6,849 |
| Row 5<br>Cray_B | 52.12 | 958,075 | 958 | 3.69 | 21.39 | 3.13 | 1,613.12 | 1,412.24 | 13.00 | 10 | 10 | 10 | 776 | 776 | 447 |
| | | | | 0.31<br>3.38 | 0.34<br>21.06 | 0.04<br>3.09 | | | | 382,322 | 382,322 | 15,667 | 4,853 | 4,853 | 6,849 |
| Row 6<br>Cray_B | 52.12 | 958,075 | 200 | 4.51 | 22.66 | 3.92 | 1,959.16 | 1,639.20 | 176.49 | 323 | 323 | 323 | 1,598 | 1,598 | 1,619 |
| | | | | 0.74<br>3.77 | 0.67<br>21.99 | 0.44<br>3.47 | | | | 465,187 | 465,187 | 412,317 | 4,853 | 4,853 | 6,849 |
| Row 7<br>Cray_C | 172.72 | 3,170,514 | 15,852 | 9.15 | 54.80 | 9.76 | 42.80 | 36.48 | 6.22 | 26 | 26 | 26 | 70 | 70 | 69 |
| | | | | 0.09<br>8.98 | 0.06<br>54.76 | 0.07<br>9.70 | | | | 8,622 | 8,622 | 1,011 | 9,766 | 9,766 | 16,741 |

*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 / CPU time system / CPU time user | | | Memory consumption in megabytes | | | Number of clusters / Number of cluster candidates | | | Number of frequent words / Number of words in vocabulary | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | LCC | LCP | SLCT | LCC | LCP | SLCT | LCC | LCP | SLCT | LCC | LCP | SLCT |
| Row 8 Cray_C | 172.72 | 3,170,514 | 3,170 | 11.93 | 69.16 | 10.99 | 4,409.15 | 3,851.15 | 363.00 | 44 | 44 | 44 | 928 | 928 | 909 |
| | | | | 0.93 | 0.87 | 0.14 | | | | 1,046,947 | 1,046,947 | 836,461 | 9,766 | 9,766 | 16,741 |
| | | | | 11.00 | 68.31 | 10.86 | | | | | | | | | |
| Row 9 Cray_C | 172.72 | 3,170,514 | 200 | 16.94 | 77.83 | 14.40 | 6,312.79 | 5,356.25 | 623.68 | 2,044 | 2,044 | 2,045 | 3,018 | 3,018 | 3,148 |
| | | | | 3.98 | 3.87 | 2.88 | | | | 1,501,118 | 1,501,118 | 1,500,167 | 9,766 | 9,766 | 16,741 |
| | | | | 12.97 | 73.99 | 11.52 | | | | | | | | | |
| Row 10 BGL | 708.76 | 4,747,963 | 23,739 | 44.31 | 127.70 | 46.48 | 507.21 | 1,125.35 | 517.45 | 55 | 55 | 58 | 203 | 203 | 245 |
| | | | | 0.52 | 0.67 | 0.40 | | | | 4,160 | 4,160 | 1,890 | 5,632,912 | 5,632,912 | 5,758,648 |
| | | | | 43.63 | 127.09 | 46.10 | | | | | | | | | |
| Row 11 BGL | 708.76 | 4,747,963 | 4,747 | 44.57 | 135.01 | 47.37 | 507.14 | 1,125.74 | 517.27 | 162 | 162 | 162 | 564 | 564 | 749 |
| | | | | 0.66 | 0.73 | 0.60 | | | | 34,844 | 34,844 | 17,213 | 5,632,912 | 5,632,912 | 5,758,648 |
| | | | | 43.93 | 134.34 | 46.78 | | | | | | | | | |
| Row 12 BGL | 708.76 | 4,747,963 | 200 | 48.07 | 148.92 | 50.49 | 2,350.47 | 2,947.30 | 515.54 | 813 | 813 | 814 | 5,794 | 5,794 | 9,059 |
| | | | | 2.87 | 2.92 | 2.40 | | | | 543,035 | 543,035 | 538,370 | 5,632,912 | 5,632,912 | 5,758,648 |
| | | | | 45.22 | 146.06 | 48.10 | | | | | | | | | |
| Row 13 LBR | 30,235.34 | 265,569,231 | 1,327,846 | 1,538.20 | 7,466.18 | 1,633.57 | 1,148.96 | 2,737.00 | 1,183.64 | 36 | 36 | 36 | 225 | 225 | 238 |
| | | | | 14.18 | 15.89 | 14.22 | | | | 93,010 | 93,010 | 54,573 | 14,971,358 | 14,971,358 | 15,411,637 |
| | | | | 1,523.76 | 7,450.91 | 1,619.44 | | | | | | | | | |
| Row 14 LBR | 30,235.34 | 265,569,231 | 265,569 | 1,593.00 | 7,711.56 | 1,711.67 | 1,148.95 | 2,738.15 | 1,183.52 | 274 | 274 | 274 | 408 | 408 | 450 |
| | | | | 14.76 | 17.68 | 14.61 | | | | 165,197 | 165,197 | 111,536 | 14,971,358 | 14,971,358 | 15,411,637 |
| | | | | 1,578.20 | 7,697.27 | 1,696.97 | | | | | | | | | |

*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 | | | Memory consumption in megabytes | | | Number of clusters | | | Number of frequent words | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | CPU time system<br>CPU time user | | | | | | Number of cluster candidates | | | Number of words in vocabulary | | |
| | | | | LCC | LCP | SLCT | LCC | LCP | SLCT | LCC | LCP | SLCT | LCC | LCP | SLCT |
| Row 15 * | 30,235.34 | 265,569,231 | 200 | 3,134.79 | 16,276.74 | 3307.19 | 1,979.28 | 6,773.91 | 1971.25 | 5,738 | 5,738 | 5,738 | 172,121 | 172,121 | 173,468 |
| LBR | | | | 43.61 | 47.97 | 41.68 | | | | 7,406 | 7,505 | 7,386 | 3,949,529 | N/A | 4,052,483 |
| | | | | 3,090.69 | 16,236.82 | 3265.23 | | | | | | | | | |
| Row 16 | 30,386.44 | 211,212,192 | 1,056,060 | 1,540.18 | 5,994.84 | 1,607.51 | 1,881.18 | 4,474.70 | 1,931.15 | 6 | 6 | 9 | 404 | 404 | 453 |
| TDB | | | | 13.68 | 19.33 | 14.82 | | | | 197,847 | 197,847 | 85,862 | 23,330,821 | 23,330,816 | 23,769,344 |
| | | | | 1,526.01 | 5,978.14 | 1,592.90 | | | | | | | | | |
| Row 17 | 30,386.44 | 211,212,192 | 211,212 | 1,581.13 | 6,518.60 | 1,666.06 | 4,761.32 | 5,602.46 | 1,931.28 | 22 | 22 | 28 | 1,232 | 1,232 | 1,362 |
| TDB | | | | 15.70 | 21.55 | 14.58 | | | | 1,126,218 | 1,126,218 | 913,546 | 23,330,821 | 23,330,816 | 23,769,344 |
| | | | | 1,564.86 | 6,500.28 | 1,651.10 | | | | | | | | | |
| Row 18 * | 30,386.44 | 211,212,192 | 200 | 2,975.43 | 14,048.84 | 3,195.31 | 2,287.13 | 7,005.70 | 2,297.57 | 1,683 | 1,683 | 1,728 | 381,278 | 381,278 | 385,662 |
| TDB | | | | 34.88 | 39.70 | 32.41 | | | | 3,292 | 3,272 | 3,314 | 5,617,563 | N/A | 5,850,133 |
| | | | | 2,938.78 | 14,015.32 | 3,162.77 | | | | | | | | | |
| Row 19 | 38,236.88 | 272,298,969 | 1,361,494 | 2,020.18 | 9,777.22 | 2,051.93 | 1,018.96 | 2,476.77 | 1,032.31 | 39 | 39 | 39 | 198 | 198 | 205 |
| SPT | | | | 16.74 | 19.78 | 18.17 | | | | 110,722 | 110,722 | 50,506 | 13,184,442 | 13,184,442 | 13,310,394 |
| | | | | 2,003.53 | 9,761.08 | 2,033.52 | | | | | | | | | |
| Row 20 | 38,236.88 | 272,298,969 | 272,298 | 2,023.13 | 9,862.94 | 2,048.31 | 1,719.99 | 2,479.79 | 1,030.18 | 122 | 122 | 123 | 449 | 449 | 484 |
| SPT | | | | 18.16 | 21.44 | 17.68 | | | | 407,835 | 407,835 | 325,568 | 13,184,442 | 13,184,442 | 13,310,394 |
| | | | | 2,005.12 | 9,846.10 | 2,030.22 | | | | | | | | | |
| Row 21 * | 38,236.88 | 272,298,969 | 200 | 4,764.68 | 23,059.44 | 4,933.44 | 2,889.80 | 7,818.63 | 2,029.32 | 164,776 | 164,776 | 164,776 | 369,684 | 369,684 | 371,145 |
| SPT | | | | 511.29 | 578.44 | 563.62 | | | | 237,505 | 237,420 | 237,083 | 2,760,221 | N/A | 2,792,219 |
| | | | | 4,253.34 | 22,488.76 | 4,370.16 | | | | | | | | | |

### 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, Cray_C and BGL), similarly as LogClusterP, LogClusterC requires a larger amount of memory than SLCT.
- When processing larger log files (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 produced by the algorithms, we found that the outputs of LogClusterC and LogClusterP are identical. This demonstrated that LogClusterC fully implements the LogCluster algorithm.[1]

Additionally, by comparing the number of words in vocabulary between LogCluster (LogClusterC and LogClusterP have the same values[2]) 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.

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

[1] 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.

[2] LogClusterC and LogClusterP have the same values except in log file TDB. The detailed explanation is in the footnote 1 on page 47.

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, 78.77% of the time can be saved by using LogClusterC. 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 | 4.41 | 77.32% | Row 13 LBR | 4.85 | 79.38% |
| Row 2 Cray_A | 5.14 | 80.54% | Row 14 LBR | 4.84 | 79.34% |
| Row 3 Cray_A | 5.18 | 80.69% | Row 15* LBR | 5.19 | 80.73% |
| Row 4 Cray_B | 5.72 | 82.52% | Row 16 TDB | 3.89 | 74.29% |
| Row 5 Cray_B | 5.80 | 82.76% | Row 17 TDB | 4.12 | 75.73% |
| Row 6 Cray_B | 5.02 | 80.08% | Row 18* TDB | 4.72 | 78.81% |
| Row 7 Cray_C | 5.99 | 83.31% | Row 19 SPT | 4.84 | 79.34% |
| Row 8 Cray_C | 5.80 | 82.76% | Row 20 SPT | 4.88 | 79.51% |
| Row 9 Cray_C | 4.59 | 78.21% | Row 21* SPT | 4.84 | 79.34% |
| Row 10 BGL | 2.88 | 65.28% | | | |
| Row 11 BGL | 3.03 | 67.00% | | | |
| Row 12 BGL | 3.10 | 67.74% | Average | **4.71** | **78.77%** |

**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 disk), and this extra experiment did not yield much different results. The details of this extra experiment are provided in Appendix 3.

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 12, Figure 13 and Figure 14[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 frequent 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.

In Row 5 Cray_B, Figure 13, the memory consumption of SLCT is 13.00 MB. Because 13.00 MB is a very small value compared to the memory consumption of LogClusterC and LogClusterP (1613.12 MB and 1412.24 MB), SLCT's pillar appears to be void in this figure.

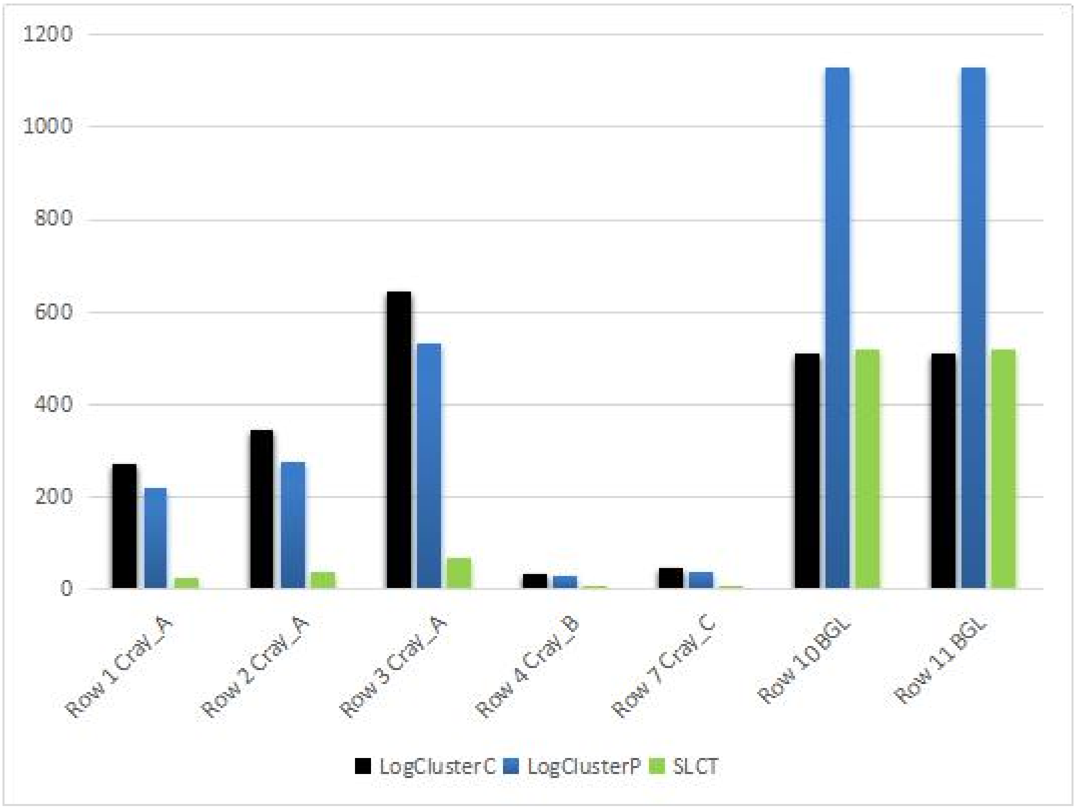


*Figure 12. Comparison of memory consumption - 1 (unit: MB)*

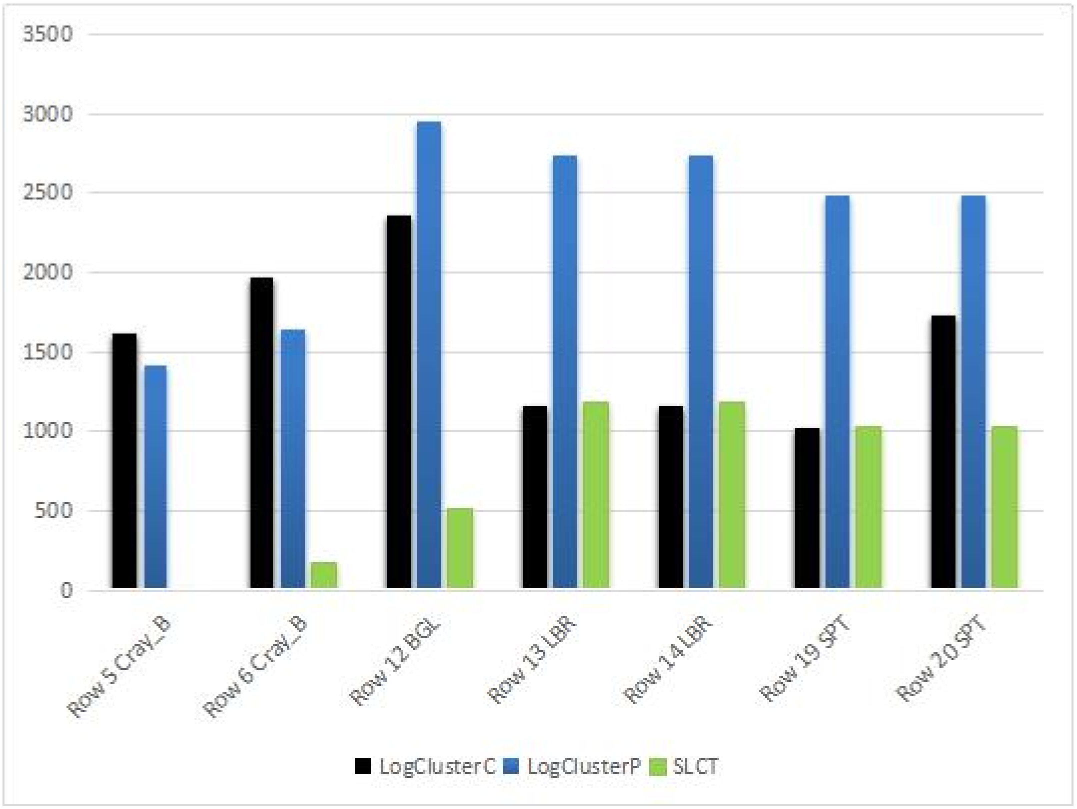


*Figure 13. Comparison of memory consumption - 2 (unit: MB)*

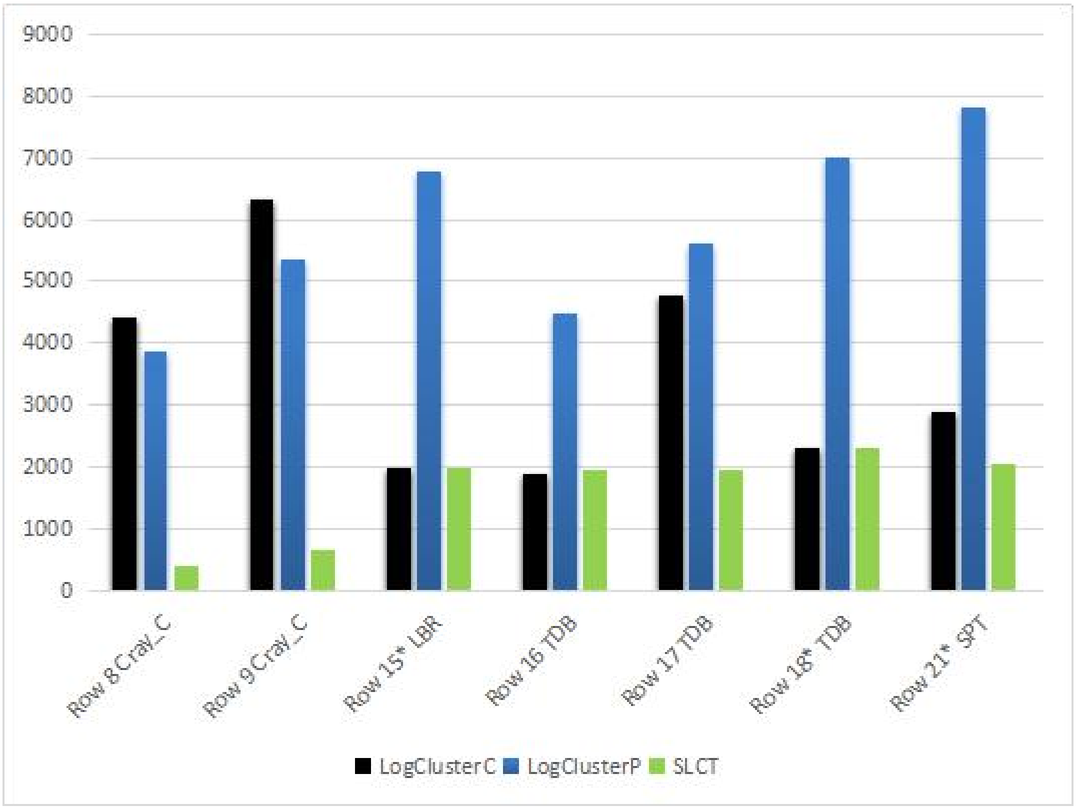


*Figure 14. Comparison of memory consumption - 3 (unit: MB)*

**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 1GB in those cases, using sketch preprocessing is recommended to reduce the unnecessarily waste of memory. Therefore, we still conclude that LogClusterC's memory consumption remains at the same level as LogClusterP'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, 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 commands used from Row 2_A to Row 4_AJ are listed below (two commands 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 #<br>Log file | Runtime | | CPU time system<br>CPU time user | | Memory consumption in megabytes | | Number of clusters | |
|---|---|---|---|---|---|---|---|---|
| | LCC | LCP | LCC | LCP | LCC | LCP | LCC | LCP |
| Row 1_B<br>Cray_A | 1.52 | 6.70 | 0.08<br>1.37 | 0.05<br>6.64 | 270.24 | 218.40 | 0 | 0 |
| Row 2_A<br>Cray_A | 1.59 | 29.61 | 0.08<br>1.46 | 0.12<br>29.50 | 282.86 | 433.19 | 2 | 2 |
| Row 3_J<br>Cray_A | 1.66 | 12.75 | 0.08<br>1.50 | 0.08<br>12.67 | 270.22 | 219.26 | 0 | 0 |
| Row 4_AJ<br>Cray_A | 1.68 | 35.60 | 0.09<br>1.59 | 0.11<br>35.51 | 282.97 | 434.00 | 1 | 1 |
| Row 5_B<br>Cray_B | 2.91 | 16.65 | 0.07<br>2.82 | 0.06<br>16.60 | 30.51 | 26.71 | 32 | 32 |
| Row 6_A<br>Cray_B | 2.95 | 18.29 | 0.10<br>2.82 | 0.07<br>18.23 | 30.00 | 46.55 | 74 | 74 |
| Row 7_J<br>Cray_B | 3.14 | 32.71 | 0.04<br>3.09 | 0.04<br>32.69 | 30.64 | 27.11 | 10 | 10 |
| Row 8_AJ<br>Cray_B | 3.20 | 33.11 | 0.07<br>3.12 | 0.05<br>33.08 | 31.55 | 46.92 | 24 | 24 |
| Row 9_B<br>Cray_C | 9.15 | 54.80 | 0.09<br>8.98 | 0.06<br>54.76 | 42.80 | 36.48 | 26 | 26 |
| Row 10_A<br>Cray_C | 9.21 | 57.50 | 0.13<br>9.02 | 0.10<br>57.43 | 44.11 | 66.07 | 77 | 77 |
| Row 11_J<br>Cray_C | 9.93 | 105.38 | 0.09<br>9.84 | 0.07<br>105.37 | 42.78 | 36.82 | 6 | 6 |
| Row 12_AJ<br>Cray_C | 9.98 | 106.44 | 0.11<br>9.87 | 0.13<br>106.37 | 42.89 | 66.43 | 19 | 19 |
| Row 13_B<br>BGL | 44.31 | 127.70 | 0.52<br>43.63 | 0.67<br>127.09 | 507.21 | 1,125.35 | 55 | 55 |
| Row 14_A<br>BGL | 44.48 | 133.69 | 0.57<br>43.92 | 0.76<br>133.00 | 506.13 | 1125.26 | 92 | 92 |
| Row 15_J<br>BGL | 46.38 | 232.98 | 0.43<br>45.96 | 0.77<br>232.33 | 505.41 | 1125.23 | 24 | 24 |
| Row 16_AJ<br>BGL | 46.37 | 238.42 | 0.45<br>45.93 | 0.78<br>237.76 | 506.20 | 1125.25 | 42 | 42 |

*Table 6 cont. Performance comparison of heuristics*

| **Row #**<br>**Log file** | **Runtime** | | **CPU time system**<br>**CPU time user** | | **Memory consumption in megabytes** | | **Number of clusters** | |
|---|---|---|---|---|---|---|---|---|
| | **LCC** | **LCP** | **LCC** | **LCP** | **LCC** | **LCP** | **LCC** | **LCP** |
| Row 17_B<br>LBR | 1,538.20 | 7,466.18 | 14.18<br>1,523.76 | 15.89<br>7,450.91 | 1,148.96 | 2,737.00 | 36 | 36 |
| Row 18_A<br>LBR | 1,464.68 | 7,669.96 | 15.36<br>1,448.55 | 14.35<br>7653.90 | 1,148.88 | 2,737.82 | 194 | 194 |
| Row 19_J<br>LBR | 1,633.65 | 14,770.63 | 11.75<br>1619.64 | 21.21<br>14,752.82 | 1,150.79 | 2,736.76 | 6 | 6 |
| Row 20_AJ<br>LBR | 1,656.20 | 15,257.69 | 15.97<br>1,640.21 | 21.28<br>15,237.15 | 1,148.88 | 2737.52 | 32 | 32 |
| Row 21_B<br>TDB | 1,540.18 | 5,994.84 | 13.68<br>1,526.01 | 19.33<br>5,978.14 | 1,881.18 | 4,474.70 | 6 | 6 |
| Row 22_A<br>TDB | 1,595.53 | 7,025.64 | 15.90<br>1,567.23 | 21.03<br>7,007.24 | 1,882.80 | 4,476.16 | 66 | 66 |
| Row 23_J<br>TDB | 1,659.32 | 11,534.95 | 16.45<br>1,638.10 | 28.60<br>11,511.18 | 1,881.25 | 4474.62 | 5 | 5 |
| Row 24_AJ<br>TDB | 1,674.62 | 12,473.37 | 16.61<br>1,653.74 | 29.41<br>12,448.56 | 1,883.10 | 4,476.16 | 23 | 23 |
| Row 25_B<br>SPT | 2,020.18 | 9,777.22 | 16.74<br>2,003.53 | 19.78<br>9,761.08 | 1,018.96 | 2,476.77 | 39 | 39 |
| Row 26_A<br>SPT | 1,998.40 | 9,746.79 | 20.19<br>1,977.94 | 22.71<br>9,728.89 | 1,020.93 | 2,477.65 | 157 | 157 |
| Row 27_J<br>SPT | 2,063.29 | 19,667.34 | 19.60<br>2,043.40 | 32.26<br>19,645.52 | 1,018.94 | 2,476.62 | 6 | 6 |
| Row 28_AJ<br>SPT | 2,081.07 | 19,766.70 | 20.28<br>2,060.63 | 32.14<br>19,744.21 | 1,018,90 | 2,477.55 | 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 confirmed that LogClusterC and LogClusterP generated the same outputs. 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.** 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 Realtion A values (around 18.5 and 21.0) than the other six log files, while on another computer 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 | 18.62 | Row 3_J Cray_A | 7.68 | Row 4_AJ Cray_A | 21.19 |
| Row 6_A Cray_B | 6.20 | Row 7_J Cray_B | 10.41 | Row 8_AJ Cray_B | 10.35 |
| Row 10_A Cray_C | 6.24 | Row 11_J Cray_C | 10.61 | Row 12_AJ Cray_C | 10.67 |
| Row 14_A BGL | 3.01 | Row 15_J BGL | 5.02 | Row 16_AJ BGL | 5.14 |
| Row 18_A LBR | 5.24 | Row 19_J LBR | 9.04 | Row 20_AJ LBR | 9.21 |
| Row 22_A TDB | 4.40 | Row 23_J TDB | 6.95 | Row 24_AJ TDB | 7.45 |
| Row 26_A SPT | 4.88 | Row 27_J SPT | 9.53 | Row 28_AJ SPT | 9.50 |
| Relation A average | **5.00** (6.94) | Relation A average | **8.59** (8.46) | Relation A average | **8.72** (10.50) |
| Relation B average | **80.00%** (85.59%) | Relation B average | **88.36%** (88.18%) | Relation B average | **88.53%** (90.48%) |
| | | | | | |
| Relation A average basic functions | | | | | **4.70** (4.65) |
| Relation B average basic functions | | | | | **78.72%** (78.49%) |

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, about 85% time can be saved using LogClusterC.

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 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 15.

**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 15, 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 still conclude that LogClusterC's memory consumption remains at the same level as LogClusterP's.

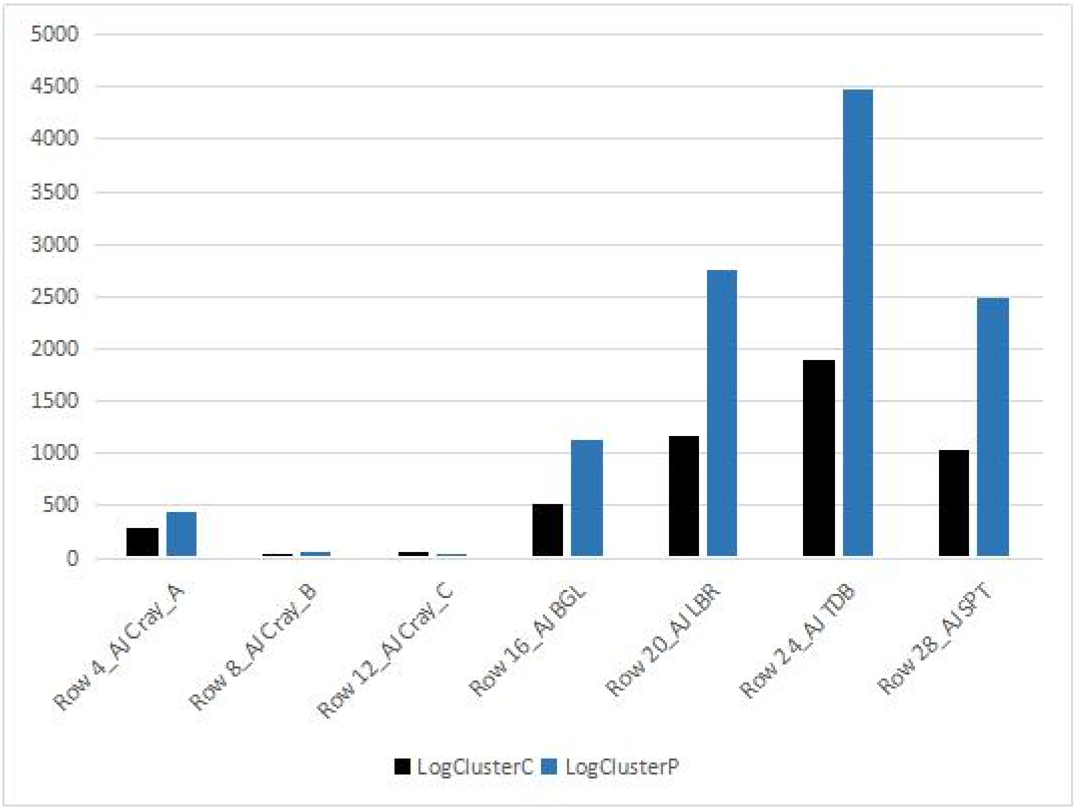


*Figure 15. Comparison of memory consumption - 4 (unit: MB)*

# 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.

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 takes about 33 minutes. To further reduce the processing time, updating LogClusterC to be multi-threaded is considered as future work.

Another hypothetical 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 reality, 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 time that is small 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: Zhuge Chen

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] Vaarandi, Risto. "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] Vaarandi, Risto, and Mauno Pihelgas. "LogCluster-A Data Clustering and Pattern Mining Algorithm for Event Logs." In Network and Service Management (CNSM), 2015 11th International Conference on, pp. 1-7. IEEE, 2015.

[3] Vaarandi, Risto. "A breadth-first algorithm for mining frequent patterns from event logs." In Intelligence in Communication Systems, pp. 293-308. Springer Berlin Heidelberg, 2004.

[4] Vaarandi, Risto. "Mining event logs with slct and loghound." In Network Operations and Management Symposium, 2008. NOMS 2008. IEEE, pp. 1071-1074. IEEE, 2008.

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

[6] Klemettinen, Mika. "A knowledge discovery methodology for telecommunication network alarm databases." (1999): 100.

[7] Zheng, Qingguo, Ke Xu, Weifeng Lv, and Shilong Ma. "Intelligent search of correlated alarms from database containing noise data." In Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP, pp. 405-419. IEEE, 2002.

[8] Ma, Sheng, and Joseph L. Hellerstein. "Mining partially periodic event patterns with unknown periods." In Data Engineering, 2001. Proceedings. 17th International Conference on, pp. 205-214. IEEE, 2001.

[9] Treinen, James J., and Ramakrishna Thurimella. "A framework for the application of association rule mining in large intrusion detection infrastructures." In Recent Advances in Intrusion Detection, pp. 1-18. Springer Berlin Heidelberg, 2006.

[10] Clifton, Chris, and Gary Gengo. "Developing custom intrusion detection filters using data mining." In MILCOM 2000. 21st Century Military Communications Conference Proceedings, vol. 1, pp. 440-443. IEEE, 2000.

[11] Reidemeister, Thomas, Mohammad A. Munawar, and Paul AS Ward. "Identifying symptoms of recurrent faults in log files of distributed information systems." In Network Operations and Management Symposium (NOMS), 2010 IEEE, pp. 187-194. IEEE, 2010.

[12] Reidemeister, Thomas, Miao Jiang, and Paul AS Ward. "Mining unstructured log files for recurrent fault diagnosis." In Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on, pp. 377-384. IEEE, 2011.

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

[14] Xu, Wei, Ling Huang, Armando Fox, David A. Patterson, and Michael I. Jordan. "Mining Console Logs for Large-Scale System Problem Detection." SysML 8 (2008): 4-4.

[15] Brauckhoff, Daniela, Xenofontas Dimitropoulos, Arno Wagner, and Kavè Salamatian. "Anomaly extraction in backbone networks using association rules." IEEE/ACM Transactions on Networking (TON) 20, no. 6 (2012): 1788-1799.

[16] Stearley, Jon. "Towards informatic analysis of syslogs." In Cluster Computing, 2004 IEEE International Conference on, pp. 309-318. IEEE, 2004.

[17] Makanju, Adetokunbo, Stephen Brooks, A. Nur Zincir-Heywood, and Evangelos E. Milios. "Logview: Visualizing event log clusters." In Privacy, Security and Trust, 2008. PST'08. Sixth Annual Conference on, pp. 99-108. IEEE, 2008.

[18] Glatz, Eduard, Stelios Mavromatidis, Bernhard Ager, and Xenofontas Dimitropoulos. "Visualizing big network traffic data using frequent pattern mining and hypergraphs." Computing 96, no. 1 (2014): 27-38.

[19] Mannila, Heikki, Hannu Toivonen, and A. Inkeri Verkamo. "Discovery of frequent episodes in event sequences." Data mining and knowledge discovery 1, no. 3 (1997): 259-289.

[20] Klemettinen, Mika, Heikki Mannila, and Hannu Toivonen. "Rule discovery in telecommunication alarm data." Journal of Network and Systems Management 7, no. 4 (1999): 395-423.

[21] Burns, L., 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] Vaarandi, Risto, and Kãrlis Podiņš. "Network ids alert classification with frequent itemset mining and data clustering." In Network and Service Management (CNSM), 2010 International Conference on, pp. 451-456. IEEE, 2010.

[23] Makanju, Adetokunbo, A. Nur Zincir-Heywood, and Evangelos E. Milios. "A lightweight algorithm for message type extraction in system application logs." Knowledge and Data Engineering, IEEE Transactions on 24, no. 11 (2012): 1921-1936.

[24] Makanju, Adetokunbo. "Exploring Event Log Analysis With Minimum Apriori Information." (2012).

[25] Makanju, Adetokunbo AO, 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. ACM, 2009.

[26] Agrawal, Rakesh, and Ramakrishnan Srikant. "Fast algorithms for mining association rules." In Proc. 20th int. conf. very large data bases, VLDB, vol. 1215, pp. 487-499. 1994.

[27] Ning, Xia, Geoff Jiang, Haifeng Chen, and Kenji Yoshihira. "HLAer: a System for Heterogeneous Log Analysis." http://www.nec-labs.com/~gfj/xia-sdm-14.pdf

[28] Lim, Chinghway, Navjot Singh, and Shalini Yajnik. "A log mining approach to failure analysis of enterprise telephony systems." In Dependable Systems and Networks

With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, pp. 398-403. IEEE, 2008.

[29] Yamanishi, Kenji, and Yuko Maruyama. "Dynamic syslog mining for network failure monitoring." In Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pp. 499-508. ACM, 2005.

[30] Vitek, Jan, and Tomas Kalibera. "R3: Repeatability, reproducibility and rigor." ACM SIGPLAN Notices 47, no. 4a (2012): 30-36.

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

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

[33] Mannila, Heikki, Hannu Toivonen, and A. Inkeri Verkamo. "Discovering frequent episodes in sequences Extended abstract." In Proceedings the first Conference on Knowledge Discovery and Data Mining, pp. 210-215. 1995.

[34] Ramakrishna, M. V., and Justin Zobel. "Performance in Practice of String Hashing Functions." In DASFAA, pp. 215-224. 1997.

[35] Zobel, Justin, Steffen Heinz, and Hugh E. Williams. "In-memory hash tables for accumulating text vocabularies." Information Processing Letters 80, no. 6 (2001): 271-277.

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

[37] Lonvick, Chris. "The BSD syslog protocol." (2001).

[38] Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A density-based algorithm for discovering clusters in large spatial databases with noise." In Kdd, vol. 96, no. 34, pp. 226-231. 1996.

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

[40] Ankerst, Mihael, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. "OPTICS: ordering points to identify the clustering structure." In ACM Sigmod Record, vol. 28, no. 2, pp. 49-60. ACM, 1999.

[41] zhugechen/logclusterc, https://github.com/zhugechen/logclusterc

[42]  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
CPU time user: 1171.59
Memory consumption: 32884
```

# Appendix 2 - The potential logic bug in the Aggregate_Support heuristic's support value accumulation and its solution

In LogClusterC, Aggregate_Support heuristic has a potential logic bug.

For example, there are three cluster candidates,

```
User bob login from 10.1.1.1, support 5
User *{1,1} login from 10.1.1.1, support 10
User *{1,1} login from *{1,1}, support 100
```

There are different aggregate orders. In this example, there are order A(right) and order B(wrong).

Order A(right):

```
1. Aggregate "User *{1,1} login from *{1,1}", check for more specific cluster
candidates, found 2. Support changed to 115 (origin + 10 + 5).
2. Aggregate "User *{1,1} login from 10.1.1.1", check for more specific
cluster candidates, found 1. Support changed to 15 (origin + 5).
3. Aggregate "User bob login from 10.1.1.1", check for more specific cluster
candidates, found 0.
```

Order B(wrong), the potential logic bug:

```
1. Aggregate "User bob login from 10.1.1.1", check for more specific cluster
candidates, found 0.
2. Aggregate "User *{1,1} login from 10.1.1.1", check for more specific
cluster candidates, found 1. Support changed to 15 (origin + 5).
3. Aggregate "User *{1,1} login from *{1,1}", check for more specific cluster
candidates, found 2. Support changed to 120 (origin + 15 + 5). (wrong here)
```

This potential logic bug should be rare, as LogClusterC chooses the cluster candidates to aggregate with a constants ascending order. Aggregation of cluster candidates that have five constants will not start, until the aggregation of cluster candidates that have four constants have been finished. However, there is still possibility this logic error happens.

To eliminate this potential logic error, isolation of the support values that participant in the Aggregate_Support heuristics is implemented.

Taking advantage of the fact that {struct Elem} and {struct Cluster} contains the same counter value of a cluster candidate, in Aggregate_Supports heuristic process, LogClusterC only changes the counter in {struct Elem} and remain the counter in {struct Cluster} unchanged. When a cluster candidate finds another cluster candidate as a more specific cluster candidate, the former cluster candidate adds its counter in {struct Elem} by the counter in the later one's {struct Cluster}. After the whole Aggregate_Support heuristic process is finished, LogClusterC traverses all cluster candidates, and replace every cluster candidates' counter in {struct Cluster} with its corresponding counter in {struct Elem}.

# Appendix 3- The extra runtime comparison experiment between the C-based implementations of LogCluster and SLCT

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.

| **Row #**<br>**Log file** | **Runtime Origin** | | **Runtime**<br>**SSD** | | **CPU time system**<br>**CPU time user**<br>**SSD** | |
|---|---|---|---|---|---|---|
| | LCC | SLCT | LCC | SLCT | LCC | SLCT |
| Row 1<br>Cray_A | 1.52 | 1.37 | 1.54 | 1.36 | 0.09<br>1.38 | 0.01<br>1.34 |
| Row 2<br>Cray_A | 1.41 | 1.42 | 1.43 | 1.44 | 0.14<br>1.29 | 0.04<br>1.39 |
| Row 3<br>Cray_A | 1.59 | 1.49 | 1.64 | 1.50 | 0.17<br>1.46 | 0.04<br>1.46 |
| Row 4<br>Cray_B | 2.91 | 3.07 | 2.88 | 3.03 | 0.08<br>2.80 | 0.05<br>2.98 |
| Row 5<br>Cray_B | 3.69 | 3.13 | 3.74 | 3.13 | 0.40<br>3.34 | 0.03<br>3.10 |
| Row 6<br>Cray_B | 4.51 | 3.92 | 4.62 | 3.90 | 0.80<br>3.82 | 0.44<br>3.46 |
| Row 7<br>Cray_C | 9.15 | 9.76 | 9.05 | 9.78 | 0.08<br>8.97 | 0.08<br>9.70 |
| Row 8<br>Cray_C | 11.93 | 10.99 | 12.14 | 11.02 | 1.04<br>11.09 | 0.18<br>10.84 |
| Row 9<br>Cray_C | 16.94 | 14.40 | 16.99 | 14.58 | 4.08<br>12.92 | 3.00<br>11.58 |

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

| **Row #<br>Log file** | **Runtime Origin** | | **Runtime<br>SSD** | | **CPU time system<br>CPU time user<br>SSD** | |
|---|---|---|---|---|---|---|
| | LCC | SLCT | LCC | SLCT | LCC | SLCT |
| Row 10<br>BGL | 44.31 | 46.48 | 43.89 | 46.54 | 0.48<br>43.43 | 0.45<br>46.11 |
| Row 11<br>BGL | 44.57 | 47.37 | 44.73 | 47.24 | 0.69<br>44.06 | 0.74<br>46.52 |
| Row 12<br>BGL | 48.07 | 50.49 | 48.01 | 50.11 | 2.90<br>45.12 | 2.44<br>47.68 |
| Row 13<br>LBR | 1,538.20 | 1,633.57 | 1,456.43 | 1,539.10 | 14.00<br>1,443.28 | 13.94<br>1,526.05 |
| Row 14<br>LBR | 1,593.00 | 1,711.67 | 1,479.11 | 1,572.91 | 14.34<br>1,465.63 | 14.50<br>1,559.32 |
| Row 15*<br>LBR | 3,134.79 | 3,307.19 | 2,762.10 | 3,068.20 | 40.52<br>2,723.24 | 40.90<br>3,029.11 |
| Row 16<br>TDB | 1,540.18 | 1,607.51 | 1,543.14 | 1,609.81 | 14.97<br>1,529.07 | 14.92<br>1,595.86 |
| Row 17<br>TDB | 1,581.13 | 1,666.06 | 1,579.82 | 1,643.18 | 15.77<br>1,564.99 | 15.10<br>1,629.05 |
| Row 18*<br>TDB | 2,975.43 | 3,195.31 | 2,748.68 | 3,014.42 | 33.70<br>2,716.62 | 33.63<br>2,982.59 |
| Row 19<br>SPT | 2,020.18 | 2,051.93 | 1,792.71 | 1,872.41 | 15.81<br>1,777.97 | 17.05<br>1,856.46 |
| Row 20<br>SPT | 2,023.13 | 2,048.31 | 1,796.31 | 1,900.48 | 16.70<br>1,780.66 | 16.73<br>1,884.86 |
| Row 21*<br>SPT | 4,764.68 | 4,933.44 | 4,130.23 | 4,350.36 | 501.76<br>3,630.91 | 501.87<br>3,850.87 |