Conference on Electronics, Telecommunications and Computers – CETC 2013

Memory Management for Big Data Mining
– Cache Hit Rate Estimation of LessFU

Kenichi Yoshida

*University of Tsukuba, Ostuka 3-29-1, Bunkyo, Tokyo 112-0012, Japan

Abstract

We have developed a network monitor which can find IP packets sent by Internet Virus from Internet backbone traffic. A data mining engine which can handle 10M transactions per second is the main component of the monitor. Although the data mining engine have to analyze over 200G byte data in theory, a memory management strategy named LessFU removes non-essential data to realize efficient processing. Our past experiments which use real Internet traffic shows the advantage of our approach. However, there exits no method to evaluate the cache hit rate of LessFU. Since the cache hit rate results in serious consequences on the data mining results, this paper proposes a method to estimate the cache hit rate of LessFU. The experimental results which show the advantage of the proposed method are also reported in this paper.

Keywords: Big Data, Data Mining, Memory management, Zipf’s law, Bloom Filter.

1. Introduction

“Big Data” and “Data Mining” are vogue buzz words. Although their definitions vary among researchers, we recognize the importance of the various data mining techniques for big data. Among such techniques, we are studying the memory management efficiency of data mining techniques since memory management efficiency is important in the analysis of big data.

Particularly, we have developed a network analyzer [1] that can find IP packets sent by undesirable applications such as Internet viruses and distributed denial of service (DDoS) software from Internet backbone traffic. A data mining engine that can handle 10 M transactions per second is the main component of the analyzer. Although data mining engines theoretically have to analyze over 200 GB of data, a memory management strategy named LessFU (Less Frequently Used) removes nonessential data to realize efficient processing. A spam filter, which we developed in a related study [2], also uses the same memory management strategy.

* Kenichi Yoshida Tel.: +81-3-3942-6896; fax: +81-3-3942-6829. E-mail address: yoshida@gssm.otsuka.tsukuba.ac.jp
Our past experiments, which used real Internet traffic, show the advantage of our approach [1,2]. However, there exists no method to evaluate the cache hit rate of LessFU. Because the cache hit rate results in serious consequences on the data mining results, this paper proposes a method to estimate the cache hit rate of LessFU.

The rest of this paper is organized as follows. Section 2 summarizes the related studies. Section 3 proposes a mechanism to estimate the cache hit rate of LessFU. Section 4 reports on the experimental results, and Section 5 summarizes our findings.

2. Related Works

2.1. Data size of network analysis

Figure 1 shows an example of the outputs produced by our network analyzer. It shows that there exists a specific node that sends packets to more than 1000 destinations every 30 min. The node sends only a single UDP packet to each destination and receives a single UDP acknowledgment from each destination. After receiving the acknowledgments, the node does not send any packet for 30 min. Because this is a typical characteristic of the keep-alive behavior of Botnet, finding this type of strange packet flow is important to manage networks.\(^1\)

Because the size of data to be analyzed is so huge, the finding of this type of hidden Botnets is difficult. For example, to find the flow shown in Figure 1, a network analyzer must analyze at least 216 GB of data. That is, a 10 Gbps network can send 10 M packets per second. To find nodes that send packets to various destinations, the network analyzer must analyze a 4 byte destination IP address, a 2 byte destination port number, a 4 byte source IP address, and a 2 byte source port number. Because the behavior of Botnets changes every 30 min, the network analyzer has to store the information of at least a 30 min period. Thus, the total data size becomes 216 Gbyte, i.e., 10 M transactions every 1800 s and 12 byte.

2.2. DRAM limitation

The difficulty associated with the network analysis is the required processing speed, i.e., 10 M transactions per second. To find packets sent by an Internet virus, DDoS software, and other undesirable Internet applications, a similar scale analysis is required. Because the analysis over 10 M transactions per second with 216 GB of data requires unfeasible CPU and memory resources, we developed LessFU [1], which removes nonessential data.

Note that the random I/O performance of DRAM systems is not very fast. For example, our network analyzer requires at least 15 data renewals, i.e., random memory read and write operations, per transaction. Thus, 10 M

\(^1\) Botnets have the tendency to remain silent in order to hide their existence. They then become suddenly active for some specific purpose. Botnets used for DDoS attacks are a typical example of such Botnets. To prevent such undesirable use of a network, it is important to find this type of hidden Botnets.
transactions per second require 300 M random I/O transactions, i.e., 10 M transactions at 15 read and write operations per second. As clearly shown in Figure 2, this performance requirement far exceeds the current computer systems. If the size of the data is less than 10 MB, the cache system of computers can handle 60 M I/O operations per second. However, if the size exceeds 10 MB, the DRAM slows down to 10 M I/O operations per second.

By removing nonessential data from the cache, LessFU decreases the number of I/O operations and enables more efficient processing.

2.3. LessFU

LessFU is a fake LFU. Like LFU, LessFU tries to discard the least frequently used data. However, LessFU does not seek the least frequently used data in the entire memory. It only compares few randomly selected data entries, and then discards the least frequently used data among the compared data. Figure 3 shows the basic concept of LessFU. It compares N (typically 4 ∼ 8) cache entries, which are selected by N hash functions, and selects the least frequently used cache entry from among the N entries. Then, it replaces the selected data entries with new data. Here, N hash functions are used as N random position generators. Because the use of hash functions enables LessFU to recalculate the same random positions from the input data, LessFU does not use the association memory, which is used in various memory management programs. This also contributes to the memory efficiency of LessFU.

A common characteristic of the data handled by these applications is Zipf’s law. Heavy duplication in Internet spams causes the distribution of similar e-mails to follow Zipf’s law [2]. Various Internet viruses and DDoS software also generate similar IP packets, which show Zipf’s law distribution [1]. Thus, the analysis of frequent data is important to analyze the characteristics of the Internet data.
Although the concept and its implementation are extremely simple, LessFU has a good cache hit rate on the data that follow Zipf’s law. By removing less frequent data, LessFU enables the data mining methods to concentrate on frequent data analysis (See Section 4 for the experimental results, which justify this approach).

3. Cache Hit Rate Estimation of LessFU

3.1. Outline

In general, data mining techniques can handle small errors in input data as noise. LessFU removes less frequent data as nonessential data, i.e., noise. If the percentage of removed data is sufficiently small, data mining methods can find packets of undesirable Internet applications from huge Internet backbone packet data. Our past experiments [1,2] prove this fact. However, the cache miss rate, i.e., the amount of error in the input, results in serious consequences on the data mining results. Thus, this paper proposes a method for estimating the cache hit rate of LessFU.

The concept behind our method is simple: data mining methods with LessFU sometimes mistake the data that it encountered in the past as the first occurrence of the data. In other words, input data of data mining methods from LessFU have false negative errors. By checking the past occurrence of the encountered data using a Bloom filter [3], the data mining methods can notice the possibility of the false negative error. Although the Bloom filter has false positive errors, it does not have false negative errors. Thus, the check using the Bloom filter enables data mining methods that know the possibility of the false negative error. Because this check gives the upper bound of false negative errors, we can estimate the upper bound of the cache miss rate.

3.2. Implementation

Figure 4 shows the pseudocode of LessFU with a Bloom filter. The black lines show the basic LessFU pseudocode, and the red lines show the modification proposed in this paper.

When LessFU stores new data in the cache, it compares N (typically 4 ~ 8) cache entries selected by N hash functions. Then, it selects the least frequently used cache entry among the N entries, and replaces the selected data entries with the new data. If the data are already stored in the cache, it increments the corresponding counter by 1. Using the Bloom filter to estimate the cache hit rate, it also calculates k hash values and makes k Bloom filter entries to be 1 (See red lines in “Store” procedure).

Note that the modified code shown in red also decrements the randomly selected counter by 1. This modification is performed to handle stream data. The distribution of stream data tends to vary. For example, the Botnet node, which generates many packets during the DDoS attack, does not send any packet when it does not attack any victim. To handle this change in distribution, the pseudocode randomly decrements the counter to remove old information. This operation deletes old data from the cache by slowly decreasing its counter.

To retrieve data from the cache, LessFU checks N cache entries using N hash functions. If one out of N entries stores the data, the “Retrieve” function returns the frequency of the data. If the cache does not have the corresponding entry, basic LessFU returns 0. However, LessFU with the Bloom filter also checks the possibility of the past occurrence, and returns “Error” if the Bloom filter has the corresponding entry (See red lines in “Retrieve” function).

3.3. Handling of stream by 2 bloom filters

Our main target is various network data, e.g., Internet backbone traffic and spam mails. Thus, the handling of stream data is important. Because the use of the same single Bloom filter continues to increase its false positive rate, we use two Bloom filters alternately. Figure 5 illustrates this concept.

Suppose we are interested in N hours of data, and data older than N hours are not important. In such case, by storing the data in two Bloom filters and initializing the two Bloom filters (i.e., set all elements to be 0), every N hours in turn, we can always access Bloom filters that hold information of at least an N hours period. In other words, the newly initialized Bloom filter holds information of 0 to N hours. Another Bloom filter holds information of slightly old N hours.

Although the use of the latter Bloom filter above increases the false positive error rate, it is good enough to estimate the upper bound of the cache miss rate.
Global Variable

- Cache[]: Cache Memory
- Counter[]: Frequency of Cache Element
- BloomFilter[]: Bit Array

Procedure Store

Input
- Data: Data to be stored in Cache

Variable
- Hash[]: Table of Hash Values
- Idxs[]: Table of Cache Index
- Idx: Selected Cache Index

begin
  Calculate N hash values from Data and store them into Hash[]
  Idxs[] = Hash[] % Cache Size
  Idx = member of Idxs that refers least frequent entry
  If Cache[Idx] stores past occurrence of Data
  Increment Counter[Idx]
  else
    Cache[Idx] = Data
    Counter[Idx] = 1

  Calculate k hash values from Data and store them into Hash[]
  Idxs[] = Hash[] % BloomFilter Size
  Set all BloomFilter[Idxs] = 1
  Idx = Random % Cache Size
  Decrement Counter[Idx]

end

Function Retrieve

Input
- Data: Data to be retrieved

Variable
- Hash[]: Table of Hash Values
- Idxs[]: Table of Cache Index

begin
  Calculate N hash values from Data and store them into Hash[]
  Idxs[] = Hash[] % Cache Size
  If One of Cache[Idxs] stores Data
  Return Counter[Idxs]

  Calculate k hash values from Data and store them into Hash[]
  Idxs[] = Hash[] % BloomFilter Size
  If all BloomFilter[Idxs] == 1
  Return -1 as Error

  Return 0

end

Fig. 4. Pseudo code of LessFU with Bloom Filter
4. Experimental Results

4.1. Cache hit rates

Figure 6 compares the cache hit rates of LessFU with those of standard memory management strategies [4,5]. To compare the performance of memory management strategies, we generated data that follow Zipf’s law. More specifically, data whose frequency and rank are shown below are generated first:

\[ C = \text{rank} \times \text{frequency} \]
\[ 10,000,000 = \sum \text{frequency} \]

The actually generated data include 763,424 sets of unique data item. The most frequent data item appears 694,879 times. Among the data, 699 data items appear more than 1,000 times. Table 1 summarizes the characteristics of the data used in the experiments.

Then, programs that implement each of the memory management strategies with cache memories of different sizes are used to measure the cache hit rate. In Figure 6, the X-axis represents the number of cache entries, while the Y-axis represents the cache hit rate (%) of each memory strategy.

- As expected, the random replacement strategy shows the worst cache hit rate.
- The cache hit rate of FIFO is almost the same. Data obtained with Zipf’s law distribution have few frequent items and many rare items. The memory space of FIFO is consumed by many rare items that appear only once. Thus, FIFO cannot improve the cache hit rate.
- LRU shows a slightly better cache hit rate. However, the difference between LRU and the random strategy is not significant. This low hit rate is also caused by the large number of rare items in the data.
- Simplified 2Q uses two queues. The first queue is controlled by FIFO and the second queue is controlled by LRU. Although both LRU and FIFO show low hit rates, the combination of FIFO and LRU has a much higher cache hit rate.

Table 1. Data Used in Experiments

<table>
<thead>
<tr>
<th>Exponent</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total volume</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Number of Unique data item</td>
<td>763,424</td>
</tr>
<tr>
<td>Data item which appears more than 1,000</td>
<td>699</td>
</tr>
</tbody>
</table>
The cache hit rate of LessFU is similar to that of simplified 2Q. In the experiment, hash2 first compares four cache entries. By comparing the eight entries, we confirmed that LessFU can achieve a better cache hit rate. Because comparing many cache entries slows down LessFU, we prefer a comparison involving only four entries. Still, LessFU can achieve a reasonably high cache hit rate.

Note that the use of FIFO and LRU requires additional I/O operations for Simplified 2Q. Because the overhead of memory operations is smaller, LessFU appears to have an advantage.

4.2. Estimated cache miss and recall

Figure 7 shows the estimated cache miss rate and recall rate. The proposed method, which uses a Bloom filter, is used to evaluate the cache miss rate.

As shown in the figure, as the number of cache entries increases, the cache miss rate decreases. After the number of cache entries exceeds 100,000, the cache miss rate decreases rapidly. Because the use of Bloom filters overestimated the cache miss rate, the estimated cache miss rate will not become 0 even if the program uses a cache memory whose size is larger than the original data size.

Because frequently appearing data are important in various applications, Figure 7 also shows the recall rate. To differentiate between spam mails and ordinary mails, the frequency of occurrence of similar mails has important information. [2] reports a spam filter that classifies mails whose frequency exceeds 1000 as spam. [1] reports a network analyzer that finds virus infected nodes by searching nodes that send packets to more than 600 destinations. Because the virus infected nodes tend to access various destinations to find their next victims, knowing the variety of destinations from some specific source nodes is important to find Internet viruses.
Figure 7 shows the recall rate of data that appear more than 1000 times. As shown in the figures, as the number of cache entries increases, the recall rate also increases. After the number of cache entries exceeds 100,000, the recall rate increases rapidly.

Note that the recall rate with a high estimated cache miss rate is low. We interpret these results shown in Figure 7 as follows. The data mining results obtained with a cache miss rate higher than 20% are not reliable. The LessFU with the Bloom filter can check the cache miss rate, and verify the quality of data mining results. Although the recall rate shown in Figure 7 is a simple example of data mining results, the frequency of the data is the basis of various data mining analysis. Because the hugeness of recent “Big Data” makes the use of memory management indispensable, the results shown in Figure 7 highlight the importance of cache hit rate monitoring.

5. Conclusion

Memory management is an important issue in handling a huge data stream. Recent increases in “Big Data” have also increased the importance of the memory management study. In particular, the memory management strategy for the data that follow Zipf’s law has become important.

In this paper, we describe a memory management strategy, LessFU, and a mechanism to estimate its cache hit rate. The contributions of this paper are summarized as follows:

- The importance of the proposed mechanism, especially LessFU and its cache miss rate estimation, is explained.
- A mechanism to estimate the cache miss rate of LessFU is proposed.
- The experimental results show the advantage of the proposed mechanism.

The experimental results also show the importance of the cache miss rate estimation to ensure the quality of data mining results.

This paper reports only one example that shows the importance of cache miss rate estimation for data mining analysis. Further evaluations on the effects of the cache miss rate on various data mining analysis will be performed in the future.

Acknowledgements

Part of this study was supported by JSPS Grants-in-Aid for Scientific Research 25280114.

References