2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops

A hadoop extension to process mail folders and its application to a spam dataset

Pedro H. B. Las-Casas
Computer Science Department
Federal University of Minas Gerais
pedro.lascasas@dcc.ufmg.br

Vinicius Santos Dias
Computer Science Department
Federal University of Minas Gerais
vvsdias@gmail.com

Renato Ferreira
Computer Science Department
Federal University of Minas Gerais
renato@dcc.ufmg.br

Wagner Meira
Computer Science Department
Federal University of Minas Gerais
meira@dcc.ufmg.br

Dorgival Guedes
Computer Science Department
Federal University of Minas Gerais
dorgival@dcc.ufmg.br

***Abstract*—Even as the web 2.0 grows, e-mail continues to be one of the most used forms of communication in the Internet, being responsible for the generation of huge amounts of data. Spam traffic, for example, accounts for terabytes of data daily. It becomes necessary to create tools that are able to process these data efficiently, in large volumes, in order to understand their characteristics. Although mail servers are able to receive and store messages as they arrive, applying complex algorithms to a large set of mailboxes, either for characterization, security reasons or for data mining goals is challenging. Big data processing environments such as Hadoop are useful for the analysis of large data sets, although originally designed to handle text files in general. In this paper we present a Hadoop extension used to process and analyze large sets of e-mail, organized in mailboxes. To evaluate it, we used gigabytes of real spam traffic data collected around the world and we showed that our approach is efficient to process large amounts of mail data.**

I. Introduction

Even today, e-mail continues to be one of the most used forms of communication in the Internet. Recent reports [1] show that there is a daily amount of more than 43 billion messages around the Internet, and of this total, about 29 billion, or 66%, refer to the spam traffic. Although e-mails servers are capable of receiving and storing messages as they arrive, due to the large amount of data generated, it becomes difficult to perform complex operations in those datasets effectively. Thus, analyzing e-mail data, either for characterization, data mining or for security issues is a challenge.

An important problem related to e-mail is the spread of spam. Due to the huge amount of data generated, spam has become extremely expensive, both from the point of view of computational resources and the time spent to combat it. Spam campaigns, usually coming in bursts, require high computational demands for its fight. However, the infrastructure used for the analysis and detection of spam is typically fragile and not scalable. Therefore, the processing and analysis of spam traffic data, aiming its combat, increasingly require high performance computational power. Substantial efforts are made to implement algorithms and tools to be used to detect spam, but the vast majority of those studies are implemented in a non scale order. Considering the raise in the amount of spam traffic data daily, it becomes harder to use anti-spam tools in a timely manner. Las-Casas et al., for example, deal with a huge amount of data, reaching several terabytes [2]. Because of this, using only one high performance computer is not feasible to perform real time analysis without loss of information.

Therefore, the efficient handling of data from e-mail servers requires parallel and distributed solutions. We propose in this paper a tool to be used for processing email implemented in parallel and scalable manner. This tool uses Hadoop to efficiently process large sets of mailboxes. Hadoop [3], implementing the programming model MapReduce [4], has the goal of providing a simple yet powerful parallel and distributed computing paradigm in a reliable, fault-tolerance manner. Our tool implements a new format for reading files from Hadoop (new *InputFormat*). As default, Hadoop reads a file line by line. In this work, we modified the reading format, enabling it to read each e-mail from mailboxes as a single entry. Finally, we structured the Map and Reduce, where the actual processing of e-mails will be performed.

Using a set of real spam traffic data collected worldwide, our test showed a performance gain of up to 13.5 times through the proposed Hadoop tool regarding the execution of sequential experiments. We were also able to show that the replication of input files enables a gain of performance, the variation in the number of *reduce* tasks impact the performance and that our tool is scalable since, as we increased the number of execution nodes, the performance also increased. Thus, through our results, we show that it is possible to implement a Hadoop extension that efficiently process e-mail, helping its characterization and, mainly, the fight against spam.

The methodology used in the development of our tool is given in section II. Section III shows the experimental evaluation and discuss the most important results. Section IV summarizes related work. Conclusion and future work are included in section V.

II. Methodology

MapReduce is a programming paradigm used to large scale processing in a distributed and parallel manner. This

978-1-4799-7014-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SBAC-PADW.2014.25

programming model has as main features the operations *map* and *reduce*. Programmers implement their own *map* and *reduce* functions, while the system is responsible for scheduling and synchronizing these tasks. Hadoop is an open-source framework used to large scale processing of data-sets on clusters of commodity hardware. Hadoop has two major components: HDFS (Hadoop Distributed File Systems) and Hadoop MapReduce. HDFS provides reliable and scalable data storage [5], while Hadoop MapReduce is the implementation of MapReduce programming model. The main advantages of MapReduce and Hadoop are: (i) scalability - using commodity hardware, the framework is able to be successfully deployed on thousands of nodes [6]; (ii) efficiency - this programming model is very efficient for applications that require processing the data only once (or only a few times); (iii) flexibility - since programmers implement their own *map* and *reduce* functions, there is considerable flexibility in specifying the exact processing that is required over the data and (iv) fault tolerance - Hadoop provides fault tolerance mechanisms through both HDFS and MapReduce. HDFS provides fault tolerance by replication, keeping multiple replicas of each data block in several different nodes, while MapReduce provides job level fault tolerance, that is, if a job fails, it would be re-assigned to another node [7]. The disadvantage of MapReduce is on its reduced functionality, since it is usable only for certain types of applications (e.g., those that do not require iterative computations). However, considering that our application aims to process a set of e-mails in a directly manner, this model fits correctly to our goal.

Under a logic perspective, on the MapReduce programming model, all data is treated as a key-value pair (K, V). Countless *mappers* and *reducers* can be used for processing these pairs. Initially a *mapper* receives a pair $\{Ki, Vi\}$, performs operations on the pair, treating it as needed and then sends it to a *reducer*. A *reducer* receives all the pairs that have the same key and performs operations on these pairs, then sending the final result. The execution of a *reducer* can not start while the mappers processing its referred key are still working. All functions *map* and *reduce* are performed independently and in parallel. Figure 1 shows how the MapReduce architecture works.

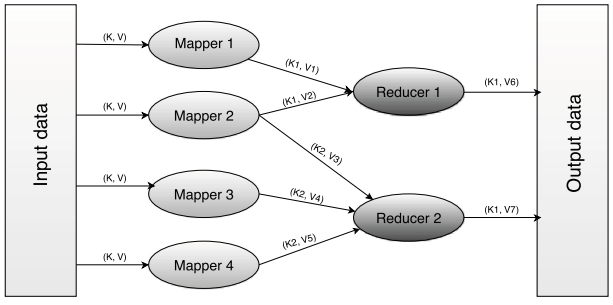

Fig. 1. MapReduce Architecture.

A. RecordReader Implementation

Before beginning *map* and *reduce* operations, it is necessary to perform some tasks. First of all, the input files need to be added to HDFS. HDFS file system fragments the files in blocks, typically 64 MB, which are used as the unit to be processed by MapReduce tasks. Each block is stored in a computing node, being replicated for fault tolerance, and then divided into splits to be processed by a mapping task. Each split is then partitioned into records, which are passed to a *mapper*. Partitioning the blocks into logical splits and records is performed by the *InputFormat* and the *RecordReader* that compose it. The *InputFormat* jobs are (i) validate the input, checking if the data is available; (ii) split the input blocks and files into logical chunks, each of which is assigned to a map task for processing and (iii) create a *RecordReader* implementation to be used to create key/value to be sent to the mappers. A *RecordReader* uses the data within the boundaries created by the input split to generate key/value pairs, that will be used by the mappers. The default Hadoop *RecordReader* consider a record of a split as a simple line, that is sent to a *map* task.

As our tool aims to process mailboxes, it was necessary to implement new *InputFormat* and *RecordReader* to consider a record as being a single e-mail. A mailbox is a simple text file that stores e-mail messages. The beginning of each message is indicated by a line whose first five characters consist of *From* followed by the sender's e-mail address and the date/time of the message. An example of a simple mailbox containing two messages is showed in Figure 2. Our new *RecordReader* uses a regular expression to represent the beginning of an e-mail. A new record is identified when the line being read matches the regular expression, and the end of the record occurs when a line matches the beginning of a new e-mail or when it found the end of the file (*EOF*). If a different type of mailbox, with different format needs to processed, all we have to do is change the regular expression used.

```
From author@example.com  Sat Feb  1 03:50:48 2014
Received: from 229.96.106.193 by ; Sat, 01 Feb 2014 05:40:45 +0200
Message-ID: <author@example.com>
From: Author <author@example.com>
To: Recipient <recipient@example.com>
Subject: Message 1
Date: Sat, 01 Feb 2014 00:39:45 -0300
...

From author2@example.com  Sat Feb  1 03:58:50 2014
Received: from 229.96.106.193 by ; Sat, 01 Feb 2014 05:42:34 +0200
Message-ID: <author2@example.com>
From: Author <author2@example.com>
To: Recipient <recipient2@example.com>
Subject: Message 2
Date: Sat, 01 Feb 2014 00:42:35 -0300
...
```

Fig. 2. Mailbox example.

As stated earlier, the first step in the processing of each block is to create a logical split and to divide it into records. In an ideal world, a logical split would be the exactly the size of the block and its division would be made splitting it into equally sized parts, in a way that it begins at the beginning of a line (or e-mail, in our case) and finishes in the end of a line. However, other options exists and need to be considered because, otherwise, it would be loss of information. Figure 3 shows the division possibilities for the blocks into logical splits and records.

The first possibility, and most trivial one, is when a block starts at the beginning of an e-mail and finishes at the end of

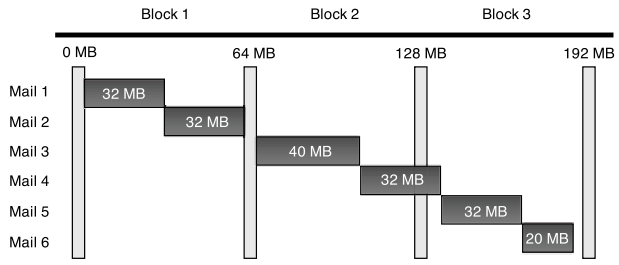

Fig. 3. Record Reader.

another one. Block 1, in Figure 3, represents that case. Mail 1 and 2 fits exactly into the block. In that case, the logical split would have the same size as the block. Another possibility is showed by Block 2. In this example, the end of the block happens at the middle of an e-mail. So, the first part of mail 4 will be read from Block 2 and second part will be read remotely from Block 3. In that case, the logical split will start at the same point as Block 2, but will end when mail 4 reaches its end, at Block 3. Each mail from the logical split will be considered as a record and sent as a key/value to a *mapper*. The third case shows when a block starts at the middle of an e-mail that have already been read. Block 3 represents that case. As can be seen in Figure 3, this block starts at the middle of mail 4. To solve that problem, we backtracked the reading in order to find the beginning of that mail, at Block 2, and restarted the reading process, ignoring the first e-mail (mail 4, in that case). For this example, the logical split would initiate at the beginning of mail 4. However, the records to be sent to a *mapper* starts at mail 5. Block 3 also shows the case when the end of the file is reached before the end of the block. That means that the logical split would end at *EOF*, been smaller than the block.

When splitting the logical split into records, our *RecordReader* extracts each information about the e-mail, such as the sender of the message, IP address of the sender, the recipients to where message is being sent, data and time, subject and content of the mail, among others. All these features compose a Mail object that were created to represent the messages and to be sent to the *mappers* as a key.

After splitting the blocks, creating the Mail objects and sending it to the *mapper*, the next step is to implement the *map* and *reduce* tasks. The implementations of *map* and *reduce* tasks are in charge of the developer using our tool and depend on the assignment that wants to perform on the input data. In the next section, we show two different implementations for the *map* and *reduce* tasks, used to evaluate our tool.

III. Experimental Results

In this section we describe different experiments performed to evaluate the performance of our tool. For the experiments, we used a small Hadoop test bed consisting of a master node and four data nodes. Each node has quad-core CPU, 4 GB memory, and 80 GB hard disk. HDFS is used for the cluster filesystem. We first explain the dataset used in our tests. Then, we explain an application used to count the number of e-mails in a mailbox and to count the number of messages intended for the recipients.

A. Dataset

As input for the evaluation of our tool, we used spam traffic collected by honeypots around the world. Data were collected using a set of sensors that implement low-interaction honeypots. Honeypots are computing resources used to collect, analyze, block or deflect attacks and/or abuse from the network aiming a service or system. The honeypot offers a service, which appears to be legitimate, to thereby attract the targets and be abused and/or attacked by them, and then, record their activity. Low-interaction honeypots only emulate the services to be abused, not being real implementations. The honeypots used in this work emulate open proxies and open mail relays, machines on the network that are traditionally abused by spammers.

In our project [1], we have 13 honeypots deployed in 11 different countries. However, due to space limitations in our cluster, we used data collected only by the honeypot TW-01, located in Taiwan. For each connection received by the honeypot, information such as the source IP address, destination IP address (in case of HTTP and SOCKS proxies), the TCP port abused in the honeypot, the protocol used (SMTP, HTTP or SOCKS) and the date/time of the connection are recorded. Other information collected also include the probable operating system associated with each source. For each source and destination IP addresses, the Autonomous System (AS), network prefix and related Country Code (CC) were also recorded at the collection time. Finally, the system stores the headers and full content of each message.

The collection of *spam* stored by honeypots is done at regular periods by a central server. The collection mechanism was implemented using remote copy and synchronization program *rsync* through an encrypted SSH tunnel. Messages collected by this server are stored in mailboxes. The mailboxes are organized by honeypot, per day and for each IP address. Having made the collection of spam messages in different mailboxes, the next step is to concatenate these files to store them in HDFS. For this, we generated a script that runs each directory and concatenates the mailboxes to a single file that contains all the spam messages captured for each day. After running this script, the generated file is then added to HDFS.

As said before, we used data collected from the honeypot deployed in Taiwan. The considered period was from February 01, 2014 to February 07, 2014. Table III-A shows more details about our datasets.

Dataset	Period	Avg. Size/Day	Total Size	# of Messages
1-week	Feb, 01 - Feb, 07	9.92 Gb	69.5 Gb	24,159,736

B. Counting the e-mails

To evaluate the performance of the proposed Hadoop extension, we first implemented a simple program to count the e-mails in a mailbox. In this implementation, the *mappers* receive each e-mail in the mailbox by the *RecordReader*. After receiving it, all *mappers* create a key/value pair with a single

[1] http://http://spammining.speed.dcc.ufmg.br

key and the value equal to 1. After that, the pairs are sent to a *reducer*, that process and count them. Figure 4 shows a schema of the application. In order to compare with the Hadoop execution, we also implemented a sequential program to count the total messages in a mailbox. This program was implemented using Python and performs the exactly same thing as the Hadoop program. For all experiments, we performed 10 different executions and calculated the confidence interval with a confidence level of 95%.

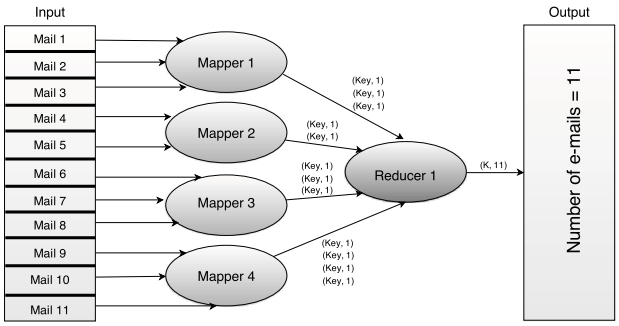

Fig. 4. Message Count.

Figure 5 shows the execution time, in seconds, for each number of nodes used to run the application and the sequential implementation. From these curves, we can observe that the application scales properly as the number of nodes used grows. Using only one DataNode results in a poor performance, requiring more than an hour to finish the processing. Increasing to 2 DataNodes, we already see a big improvement. In that case, the execution time is around 16.6 minutes, with a speedup when comparing to the 1 node execution of 3.9. The use of 3 and 4 nodes also improve the performance. The 3 nodes execution spent 14.5 minutes, obtaining a 1.15 speedup, when comparing to the 2 nodes execution. Using 4 nodes results in an execution time of 8.8 minutes, which means that the speedup is 1.65 if we compare to the 3 nodes execution. Another interesting result is obtained when we compare our tool with the sequential implementation, using Python. The sequential implementation expends around 119 minutes to perform the processing of all data. If we compare with the best performance of our tool (using 4 nodes) the speedup is 13.5. Those results show that our application scales as we increase the number of nodes used for data processing.

As stated earlier, HDFS is used to store our data. This filesystem allows replication, which factor is configurable. Because of replication, HDFS tries to satisfy a read request from a replica that is closest to the reader node, in order to minimize bandwidth consumption and read latency. For example, if there exists a replica on the same rack as the reader node, that replica will be used by that node. In order to evaluate the impact of the replication factor in our application, we performed experiments varying the number of replicas used. In that experiment, it was used the 1-week dataset, and we varied replication factor from 1 to 4.

Figure 6 shows that using replication factor equal to 1 presents the worse performance. That happens because, since there is only 1 replica of each block, when a reader node needs a block, there is a considerable probability that this block

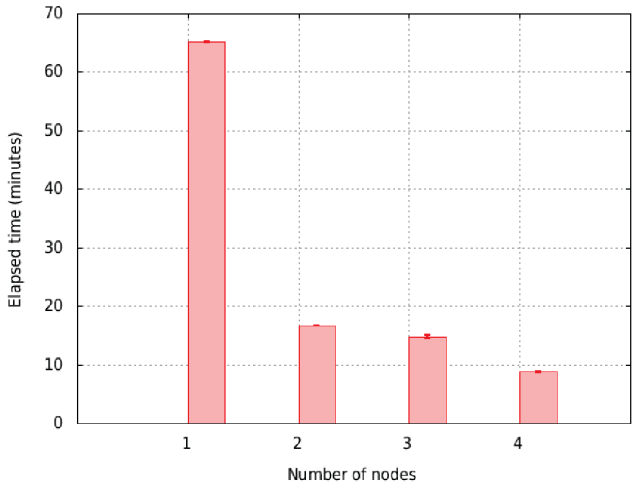

Fig. 5. Performance when varying number of nodes.

will not be local, so the node will have to request the block from another node, creating an overhead caused by bandwidth consumption and read latency. The execution considering 2, 3 and 4 replicas did not present difference among them. The major reason for this is because the cluster used in our tests is small, composed by only 4 machines, so, even with 2 replicas of each block, the amount of requisition for remote blocks does not impact in the overall performance of the application.

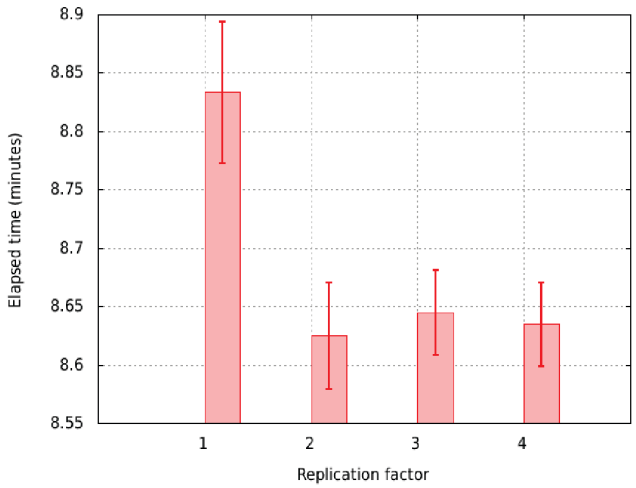

Fig. 6. Replication effects.

C. Counting e-mails by recipients

The other experiment performed aimed to count the number of messages that would be sent to each recipient. Figure 7 shows how the flow would work for this application. In that case, the e-mails from the mailboxes would be read by the *RecordReader* and sent to the *map* tasks. It is noteworthy that when reading the e-mails, the *RecordReader* generates an array containing all recipients of the message. So, the *mapper* will read this array and, for each recipient, will create a key/value pair, where the recipient is the key and will send it to a *reducer*.

Same recipients were sent to the same *reducer*. At the *reduce* task, we iterated at the list of recipients received, counting the number of e-mails for each of them.

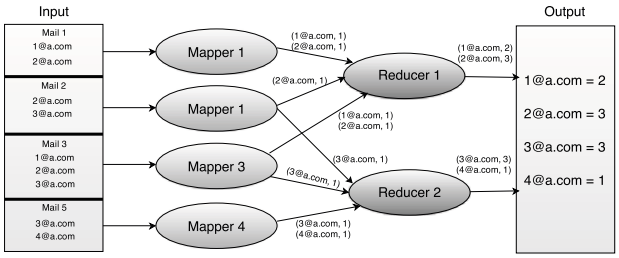

Fig. 7. Recipient Count.

In the first application, used to count the number of messages, it was necessary only one *reducer*, since there were only one type of key. However, in this second application, it is required to use more *reducers*, since there are a huge amount of different types of keys. Thus, to analyze the impact of the number of *reducers* in the overall performance of the application, we performed experiments varying the number of *reducers* used. For each experiment, we performed 30 executions. The confidence interval considers a 95% confidence level.

Figure 8 shows the impact of varying the number of reducers. As can be seen, all experiments showed a great variation, showed by the high confidence interval. This is due to the fact that the application generates a large amount of intermediate data, that is written in the hard disk and also need to be read. Those tasks have great variation in performance, which impacted in the executions of our experiments. Considering the average time, the use of 12 reducers presented the best results, spending around 19 minutes. The use of 8 reducers resulted in the worst performance, with an execution time of more than 22 minutes.

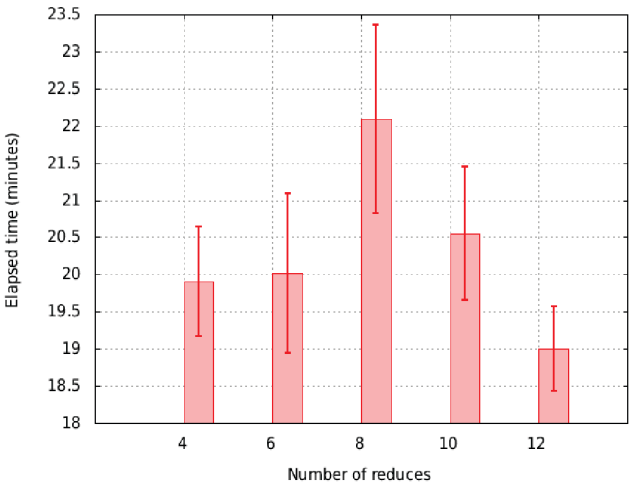

Fig. 8. Variation of the reducer.

IV. Related Work

E-mail, one of the oldest Internet application, continues to be very popular, even as the web 2.0 grows. Recent reports [1] show that over 43 billion emails traffic the Internet daily. Considering that emails have an average size of around 8 Kb, as stated by Bertolotti et al. [8], every day is generated around 300 terabytes of email data. Lee and Kim show that the interarrival time of SMTP [2] is, on average, 0.26 seconds [9]. Due to the huge amount of data and the small time between the arrival of messages, it is very important to create an efficient and scalable environment to be used in the analysis, characterization and application of complex algorithms on large sets of e-mails.

Under the spam traffic perspective, several works depicts the spam scenario. Gomes et al. analyzed an e-mail workload consisting of the messages received in a university network, pointing out a number of features that can be used to differentiate spam from legitimate messages [10]. In an extension of that work, the authors concluded that legitimate traffic tends to have lower entropy than the traffic generated by spammers, who usually send e-mails indiscriminately to all their targets [11]. Kim et al. showed that the interval between arrivals of spam is below the range of legitimate emails (less than 5 seconds in 95% of cases) [12]. Las-Casas et al. give an overview of spam traffic collected in 8 distinct points around the globe [2]. Using a dataset composed by almost 2 billion messages and 7 TB, they showed characteristics related to protocol used to send the spam message, the behavior of the traffic over the time and the influence of spam campaigns in the traffic observer by the machines collecting the data.

Despite all efforts made to analyze e-mail workloads as well as to study, characterize and combat spam, as showed in those works before, due to the large amount of data and to the need to timely manner responses, specially in the case of spam combat, it is more and more necessary to have an efficient mechanism to process the collected data. Shu et al., for example, present a framework for security log analysis that allows analysis of a massive number of system, network, and transaction logs efficiently and scalably, using a distributed and parallel environment [13]. François et al. use MapReduce and Hadoop for detecting botnets using real network traces from an Internet operator [14]. Indyk et al. proposed a collective classification method using MapReduce for web spam detection [15] and Caruana et al. presented an anti-spam architecture that also uses Hadoop and MapReduce [16]. Extending their work, the authors implemented SVM on MapReduce for large scale spam filtering [17]. In this work, we propose an extension for Hadoop that enables the analysis of mail folders and can be used both for characterization of mail workloads and as a tool to analyze the spam traffic, helping to combat it.

V. Conclusion and Future Work

In this paper we presented a Hadoop extension used to process mail folders. The extension of the file formats supported by the Hadoop File System provides an important funcionality, in this era of Big Data: direct access for hadoop programmers to large mail datasets, often available in the form of user mailboxes, or other mailbox-organized collections, like the

[2] Simple Mail Transfer Protocol

spam data used in our work. Our preliminary results show that the extension is effective in making a new set of applications possible in Hadoop (previous solutions to handle mail in the system were often too complex or slow to consider). Although performance can still be improved, our preliminary results indicate there is still room for that.

As future work we intend to continue to work on improving the system performance for that kind of applications, and to apply the framework in a more detailed analysis of the spam corpus available to us.

ACKNOWLEDGMENT

The dataset used was collected and made available to us by Cristine Hoepers, Klaus Steding-Jessen and Marcelo H.P. Chaves, from CERT.br, part of NIC.br.

This work was partially sponsored by NIC.br, Fapemig, Capes, CNPq, and the National Institute of Science a Technology of the Web, InWeb (MCT/CNPq 573871/2008-6).

REFERENCES

[1] Symantec, "Internet Security Threat Report, Volume 19," Online, April 2014.

[2] P. H. B. Las-Casas, D. Guedes, W. M. Jr., C. Hoepers, K. Steding-Jessen, M. H. P. Chaves, O. Fonseca, E. Fazzion, and R. E. A. Moreira, "Análise do tráfego de spam coletado ao redor do mundo," in *Brazilian Symposium on Computer Networks and Distributed Systems (SBRC) (In Portuguese)*. SBC, 2013.

[3] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[6] K. V. Shvachko, "Apache hadoop: The scalability update," *login: The Magazine of USENIX*, vol. 36, pp. 7–13, 2011.

[7] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using mapreduce," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 31:1–31:42, Jan. 2014.

[8] L. Bertolotti and M. Calzarossa, "Workload characterization of mail servers," *Proceedings of the 2000 Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2000.

[9] Y. Lee and J.-S. Kim, "Characterization of Large-Scale SMTP Traffic: the Coexistence of the Poisson Process and Self-Similarity," Sep. 2008, pp. 1–10.

[10] L. H. Gomes, C. Cazita, J. M. Almeida, V. Almeida, and W. M. Jr., "Workload Models of Spam and Legitimate E-mails," *Performance Evaluation*, vol. 64, no. 7-8, pp. 690–714, August 2007.

[11] L. H. Gomes, V. Almeida, J. M. Almeida, F. Castro, and L. Bettencourt, "Quantifying Social And Opportunistic Behavior In Email Networks," *Advances in Complex Systems*, vol. 12, no. 1, pp. 99–112, January 2009.

[12] J. Kim and H. Choi, "Spam Traffic Characterization," in *Int'l Technical Conference on Circuits/Systems, Computers and Communications*, Shimonoseki City, Japan, 2008.

[13] X. Shu, J. Smiy, D. Daphne Yao, and H. Lin, "Massive distributed and parallel log analysis for organizational security," in *Globecom Workshops (GC Wkshps), 2013 IEEE*. IEEE, 2013, pp. 194–199.

[14] J. Francois, S. Wang, W. Bronzi, R. State, and T. Engel, "Botcloud: Detecting botnets using mapreduce," in *Proceedings of the 2011 IEEE International Workshop on Information Forensics and Security*, ser. WIFS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–6.

[15] W. Indyk, T. Kajdanowicz, P. Kazienko, and S. Plamowski, "Web spam detection using mapreduce approach to collective classification." in *CISIS/ICEUTE/SOCO Special Sessions*, ser. Advances in Intelligent Systems and Computing, vol. 189. Springer, 2012, pp. 197–206.

[16] G. Caruana, M. Li, and H. Qi, "SpamCloud: A MapReduce based anti-spam architecture," in *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 6. IEEE, Aug. 2010, pp. 3003–3006.

[17] G. Caruana, M. Li, and M. Qi, "A MapReduce based parallel SVM for large scale spam filtering," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, vol. 4, Jul. 2011, pp. 2659–2662.