# Exploring Netflow Data using Hadoop

X. Zhou[1], M. Petrovic[2],T. Eskridge[3],M. Carvalho[4],X. Tao[5]
Xiaofeng Zhou, CISE Department, University of Florida
Milenko Petrovic, Florida Institute for Human and Machine Cognition
Tom Eskridge, Florida Institute for Human and Machine Cognition
Marco Carvalho, Florida Institute for Human and Machine Cognition
Xi Tao, CISE Department, University of Florida
xiaofeng@cise.ufl.edu, mpetrovic@ihmc.us, teskridge@ihmc.us, mcarvalho@ihmc.us, xtao@cise.ufl.edu

## ABSTRACT

Analyzing network flows is challenging both because of the complexity of interactions that it captures, but also because of the sheer volume of the data that can be captured from routers and monitors in large networks. Hadoop is a popular parallel processing framework that is widely used for working with large datasets. However, there is a lack of information about effective uses of Hadoop on NetFlow datasets. Typically, research publications focus on presenting results of work built on top of Hadoop, rather than enlightening about effective uses of the popular framework. In this paper we make a first step in achieving that goal. We identify basic tasks making up any exploratory analysis process of netflow datset, describe their realization in Hadoop framework and characterize their performance in two commonly used Hadoop deployments.

## I INTRODUCTION

NetFlow is a popular protocol for reporting summary metrics from network routers. It is implemented by virtually all existing router hardware. Netlow metrics have been used extensively for network monitoring and administration. There has been a slew of research into using NetFlow in cyber security domain for intrusion detection and forensics as well as anomaly detection. Key advantage of using NetFlow metrics over statefull packet inspection and capture is significantly reduced data requirements since packet contents are not examined. Moreover, NetFlow provides summary metrics of packet flows instead of traces of individual packets. Previous research has shown that this representation is sufficient for a wide variety of applications.

While the size of the NetFlow data is significantly reduced compared to statefull packet processing, the amount is still far from negligible. For example, public NetFlow datasets generated from major Internet backbones can range in terabytes for even just a single hour. Similarly, routers in large enterprises and university campuses can generate significant amounts of NetFlow data. Performing any kind of exploratory analysis on this volume of the data, thus, precludes using desktop-class or non-parallel analytic tools. Typically, such analysis needs to be limited to a smaller fraction of a dataset and custom solutions are developed to perform this analysis. Sampling is another alternative to reduce the size of the data. Both of these techniques reduce the fidelity, which poses a problem in detecting low volume anomalies and attacks. Following exploratory analysis, sophisticated statistical machine learning is often used to create traffic classifiers. This process is highly computationally expensive on large data sizes.

Hadoop is a popular batch-oriented data parallel framework. It implements MapReduce computation pattern, originally developed at Google [1]. Hadoop has been widely used for Big Data analysis in a variety of domains. One of the key strengths of Hadoop is simple programming model, and ability to use commodity hardware for scalable and repliabe data processing. Moreover, the framework is Java-based which makes it easy to deploy on a variety of platforms.

A vast majority of network intrusion detection systems based on NetFlow is targeted towards real-time detection of anomalies and attacks. However, there is a lack of scalable tools for exploratory analysis of NetFlow data, which, typically preceeds development of real-time classifiers.

While it is possible to use Hadoop "out-of-the-box" for analyzing NetFlow datasets, as we show, the structure of NetFlow data results in under-utilization of resources in this case. The main contributions of this paper are

- Evaluation of Hadoop for exploratory analysis of NetFlow datasets on large multicore server and Amazon Cloud
- Efficient representation for NetFlow data
- Characterization of frequent MapReduce patterns for exploratory analysis of NetFlows

Our results indicate that Hadoop can be an effective platform for exploratory analysis of NetFlow data. The rest of the paper is organized as follows: in the next section, we give additional background on Hadoop framework, relevant specifics of NetFlow protocol and widely used CAIDA dataset. We then briefly overview the related work in section III. In section IV, we detail structure of typical exploratory analysis as well as give an example of identifying a particular network attack pattern. We provide a thorough evaluation of these patterns in two different environments in section V. Finally, in section VI, we summarize the main results and outline the future work.

## II BACKGROUND

In this section we provide detailed information about Apache Hadoop [1] and NetFlow data.

### 1 APACHE HADOOP

Hadoop consists of two main components: a failure-tolerant distributed file system (HDFS) that can be deployed across thousands of commodity machines and a parallel processing framework implementing the MapReduce [1] paradigm. To illustrate how Hadoop works, Figure 1 shows the typical data flow of a simple counting words example. Here suppose we have a file “textfile1” in HDFS which contains some text, and we want to calculate the count of each word in the file. We can see that HDFS uses two blocks to store file “textfile1”. To do the counting, Figure 1 shows: first the input text file is split to two mappers, where each mapper parses their assigned portion of the text, then emits <key,value>pairs. Then Hadoop framework will shuffle the output of mappers, sort it, and partition the result by key and generate <key,list(values)>pairs. Finally, the two reducers will process the <key,list(values)>pairs, i.e., sum up all the values in the list, and output the result to their different files. And we only implement the mapper code and reducer code, Hadoop will take care of orchestrating mappers and reducers and sorting the output of mappers, i.e., the intermediate results in a efficient way.

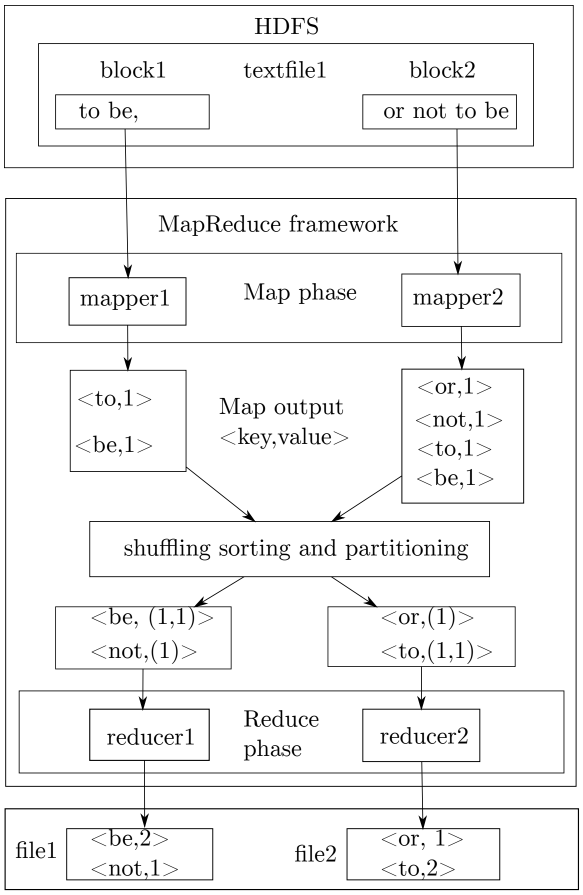

Figure 1: Typical data flow of the word counting MapReduce job

An interesting note about HDFS is that files in HDFS are stored in blocks of large size: the default HDFS block size is 128 MB. This is because HDFS is design to handle large files, and for every block of a file, the namenode, which manage all the meta-data of files in HDFS, will need to keep record of this. Bigger block size can help reduce the workload of namenode, and reduce the network load if a distance host is requesting data from a local host in a cluster for big files. Block size will also affect the number of mappers in a MapReduce (MR) job because it is the upper bound

[1]`http://hadoop.apache.org/`

of split size, which will be used by Hadoop to split input files into fragments and allocate to mappers.

### 2 NETFLOW

The NetFlow format data set that we use in this paper, The CAIDA, abbreviation for "The Cooperative Association for Internet Data Analysis", Anonymized Internet Traces 2012 Dataset [2], is collected from CAIDA's monitors on high-speed Internet backbone links. The original data set is in plain text format, and each line represents a record of several fields separated by white space. Each record has a fixed size of 150 bytes, with white space separating each field and large numbers abbreviated with suffix of 'M', 'G'. Table 1 shows some of the fields that are used in this paper.

| Field name | Example value |
|---|---|
| Date flow start | 2010-07-04 21:43:43.944 |
| Duration | 29.1 |
| Proto | TCP |
| Src IP Addr:Port | 63.35.127.29:15231 |
| Dst IP Addr:Port | 229.63.245.121:9000 |
| Packets | 2 |
| Bytes | 92 M |

Table 1: Some of the fields in a record and example values ('proto' is short for 'protocol', 'src' for 'source' and 'dst' for 'destination')

## III RELATED WORK

While Hadoop has been used widely for various large scale data analysis tasks, there have been only a few studies of using Hadoop to analyses NetFlow records. Most of such work focuses on describing tools or analysis developed on top of Hadoop, without providing much insights into how to effectively use Hadoop platform for NetFlow analysis [2–6]. Majority of other work on NetFlow record analysis are attempting to create real-time systems and do not use Hadoop [7].

In [8], authors develop a tool that uses Hadoop to compute a predetermined set of metrics over full packet traces as captured by libpcap. The authors extend Hadoop to allow storing and processing of packet traces in the format as used by libpcap. We chose to focus on NetFlow records only. In addition, we investigate effectiveness of SequenceFile, a key-value format commonly used to store data in Hadoop, as a serialization format. Our approach, however, is to store NetFlow records, without packet content, using Hadoop SequenceFiles. In addition, we focus on investigating efficiency of exploratory analysis of NetFlow traces, as opposed to a computing fixed set of metrics.

In [9], the authors compare two different frameworks that use MapReduce processing paradigm: Hadoop and Disco, which is a Python-based framework. They compare the frameworks in terms of efficiency of NetFlow representation as well as processing performance. The authors identify Disco as a higher performance platform. Our work focuses on identifying and evaluating building blocks of any exploratory analysis of NetFlow records using MapReduce paradigm, and Hadoop in particular. In addition, we examine use of SQL to express the analysis and evaluate it on Amazon Cloud platform.

## IV USING HADOOP AND HIVE TO ANALYZE NETFLOWS

Typically, any interesting record of NetFlow data will be very large. To understand the structure of such large datasets, exploratory analysis is used. One of the characteristics of exploratory analysis, is that it is very data intensive, but not very computationally intensive. Hadoop platform is especially suitable for this type of analysis.

SQL is widely used for managing data in relational database management systems, it is simple and easy to learn, and used by many database. Hive provides SQL utility to Hadoop

In this section we focus on how exploratory analysis of large NetFlow dataset is done using Hadoop and how common types of analysis are expressed effectively in MapReduce and SQL.

### 1 DATA REPRESENTATION

Routers generate NetFlow data in plain text format (see Section II). The main advantage of plain text format is its simplicity and readability. However, this comes at a cost in terms of storage and parsing. Figure 2 illustrates the transformation from text data to binary data and to SequenceFile format in Hadoop.

[2]`http://www.caida.org/data/passive/passive_2012_dataset.xml`

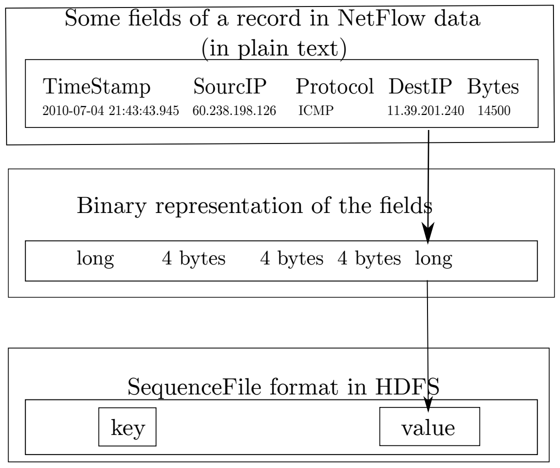
Some fields of a record in NetFlow data (in plain text)
TimeStamp SourcIP Protocol DestIP Bytes
2010-07-04 21:43:43.945 60.238.198.126 ICMP 11.39.201.240 14500
Binary representation of the fields
long 4 bytes 4 bytes 4 bytes long
SequenceFile format in HDFS
key
value

Figure 2: Transform the text representation of NetFlow records into binary representation, and store them in HDFS as SequenceFile format. The fields of a record are stored as the value in the key/value pairs in SequenceFile format

Since storing plain text data can be inefficient in terms of space utilization, we convert text representation to binary and store it using Hadoop SequenceFiles. Structure of SequenceFile is shown in Figure 2. Since SequenceFile consists of key/value pairs, it is simple and efficient to parse and MapReduce framework can use it as input/output format.

SequenceFile has three serialized on-disk formats: uncompressed, value-compressed and block-compressed, which combines several values together before applying compression. By taking advantage of SequenceFile, the size of NetFlow records can be further reduced by simply switching to one of the compressed serialization formats. Compression is often beneficial in reducing disk I/O which is often the bottleneck for data-intensive mapreduce jobs–typical of exploratory analysis. Both simplicity and repetitiveness of NetFlow records make it a good candidate for compression. Compression can provide a better balance between the CPU utilization and disk utilization, as we show in the next section, and also can reduce the size of data transferred over the network between nodes of Hadoop clusters.

Similar to SequenceFile in Hadoop, Hive uses Record Columnar File(RCFile), which is flat file consisting of binary key/values and determines how to store relational tables in distributed systems and can also be compressed.

## 2 FILTERING AND COUNTING

Exploratory analysis typically involves parsing the raw data into features and computing simple statistics (e.g., totals and distributions) across some subset of the features or for some metrics (e.g, record counts) even across the full dataset. Filtering and counting are, thus, the basic building blocks for computing such statistics.

Algorithm 1 does a very simple filtering and counting task, yet offers very useful and fundamental analysis on the NetFlow data set. We can use it to count the percentage of the different protocols used among the network, activeness of a set of specific nodes, or just filtering out records that we need by directly output the records in the algorithm.

Algorithm 1 shows the skeleton of a basic filtering and counting algorithm expressed in MapReduce for both NetFlow records and text based records using binary representation described above.

For example, using Algorithm 1 we count the numbers of records in the dataset or the number of records containing specific fields, such as records that uses the TCP protocol. To do this, we just need to modify the condition in line 4 to "record has 'TCP' in the protocol field" and emit the key as "TCP".

**Algorithm 1** Basic filtering and counting

```
   Map phase
1: function MAP(inKey inkey, inValue record)
2:     Parse the record to get each fields of the record b
3:     if record has the feature we need then
4:         emit(key, 1)
5:     end if
6: end function

   Reduce Phase
7: function REDUCE(inKey inkey, inValues values)
8:     sum ← 0
9:     for each value in values do
10:        sum ← sum+value
11:    end for
12:    ouput(inkey, sum)
13: end function
```

**Algorithm 2** Aggregation

```
   Map phase
1: function MAP(inKey $inkey$, inValue $record$)
2:     Parse the record to get each fields of the record
3:     emit($records.SourceIP + label : srcip$, $records.BytesTransferred$)
4:     emit($records.DestinationIP + label : destip$, $records.BytesTransferred$)
5: end function

   Reduce Phase
6: function REDUCE(inKey $inkey$, inValues $values$)
7:     $sum \leftarrow 0$
8:     for each $value$ in $values$ do
9:         $sum \leftarrow sum + value$
10:    end for
11:    ouput($inkey$, $sum$)
12: end function
```

Going beyond computing a single metric for NetFlow attributes (e.g., total bytes per protocol), we can also compute multiple metrics to find specific patterns. For example we can compute total traffic volume incoming and outgoing from a node by aggregating bytes transmitted from destination IPs and source IPs simultaneously in Algorithm 2. Computing multiple metrics simultaneously, reduces amount of data that needs to be read in the map phase. However, it also increases the size of intermediate results that Hadoop needs to store between Map and Reduce stages.

Since MapReduce jobs are typically run in clusters, where mappers are run on different nodes that are far apart, communication cost may be expensive. And Hadoop needs to write the map output to disk and shuffle the outputs of all mappers globally, it will help reduce this cost significantly if we can reduce the output of mappers. Here is a specific scenario, if we are trying to get the volume of data flow from each IP address, for a mapper it may output two key/value pairs (127.0.0.1, 100 bytes), (127.0.0.1, 200 bytes) from two records it has processed and write the two key/value pairs to disk. In this case we can do a reduce job locally on this mapper, i.e., combine the two key/values pairs in to one (127.0.0.1, 300 bytes) in the map phase, thus the mapper will only write one key/value pair. This way we can reduce the size of intermediate data between map phase and reduce phase. This kind of local reducers are called "combiners" and can be used to scale up aggregation.

While in Hive, suppose *table* is our created table and *sourceip* is a tuple inside the table and is of string type, which gives the source IP string. And here are the similar SQL queries that does the counting job:

- Record-counting Query

  ```
  SELECT COUNT(*) FROM table;
  ```

- Aggregation Query(Group size of 1 only)

  ```
  SELECT SUBSTR(sourceip, 1, instr(sourceip, '.') -1) , SUM(bytes) as cntbytes FROM table GROUP BY SUBSTR(sourceip, 1, instr(sourceip, '.') -1);
  ```

For most queries, Hive compiler will generate MapReduce jobs accordingly, which will be submitted to Hadoop be executed.

## 3 COMPUTING TOP-K

Another common and important analysis is to find the extreme 'points' in the data set. For example we may want to get the top 10 popular IPs among a subnet with the most connection number or largest data volume flow.

Algorithm 3 shows how to find the top-K records that have for example, most data volume flow from a specific host IP. The map phase is the same as former two algorithms, but in the reduce phase we keep a sorted array list of the top-K records we found so far. And we update the list as the reducer is fed with output from mappers. When the reduce finishes, the reducer's top-K list is the final top-K list we need. It is worth noting that the initialization and finalization of the array list are done only once before and after the multiple calls to reduce function, and the reducer only need to store a array of size K to run, where K is often a small number. And the algorithm only need to scan through the data set only once. This is yet another common important analysis.

**Algorithm 3** top-K

```
    Map phase
 1: function MAP(inKey inkey, inValue record)
 2:     Parse the record to get each fields of the record
 3:     if record 's Destination IP is 255.255.255.255 then
 4:         emit(records.SourceIP, records.Bytes)
 5:     end if
 6: end function

    Reduce Phase
 7: create list(topk)
 8: function REDUCE(inKey inkey, inValues values)
 9:     sum ← 0
10:     for each value in values do
11:         sum ← sum+value
12:     end for
13:     if sum > smallest(list(topk)) then
14:         list(topk).remove(smallest(list(topk)))
15:         list(topk).add(sum)
16:     end if
17: end function
18: Finalize: ouput list(topk)
```

However, it is notable that we need to use one reducer to get the top-K because of shared state. In cluster cases where we are dealing with huge amounts of records, this approach of one job with only one reducer is not scalable due to the fact that only one reducer will have to process outputs from many mappers.

A plausible solution would be to chain multiple finding top-K Algorithm 3 jobs, and only run the last one job with only one reducer. More specifically, for first job we run with multiple reducers to get the top-K records of each reducer. The top-K records that we need would be among the output of first job, but the size of the output of the first job would be substantially smaller than the output of the mappers in the first job, which contains a key/value pair for every record in the original data set. Then for second job take the output of the first job as input and run the algorithm again with only one reducer. Since K is normally a very small number in comparison to the number of records in the data set we are dealing with, the output of the first job would be very small in most cases and the second job would be reasonably fast enough. Anyway if the output of first job is still too big for one reducer, we can always have third job or even more, as long as we only set the reducer number to one in the last job.

Similarly using SQL, we can compute top-10 source IPs wth largest data flow using the SQL query below:(same assumptions are made as in former subsection)

- Top-k Query

  SELECT sourceip, sum(bytes) AS cntbytes FROM table GROUP BY sourceip ORDER BY cntbytes DESC LIMIT 10;

## 4 PATTERN DISCOVERY

We can do more complex analysis such as pattern discovery on NetFlow data set with the three algorithms explained above. One typical example is the watering hole attack pattern, which is explained in Figure 3.

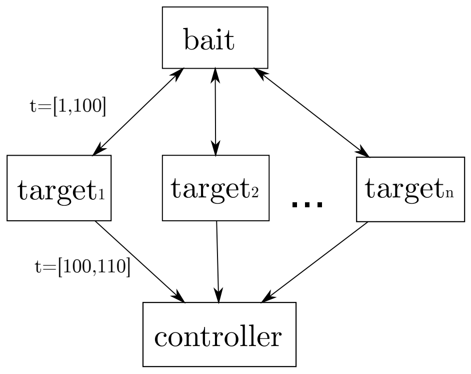

Figure 3: Watering hole attack pattern. Here a popular website ‘bait’ is compromised by an attacker and malware is placed. During a relative long time interval [1,100], multiple users called ‘targets’ will access the bait and download the malware. Then in a short following time interval, the targets will contact an external service called controller, which is controlled by the attacker and is rarely accessed before.

So our interest lies in identifying patterns such as the watering hole attack in the NetFlow data set in order to defend against such attacks. For this specific task, we can run several combinations of the counting, aggregation and finding top-k operations to complete our goal. Algorithm 4 show the outline of identifying possible watering hole attack patterns, which consists of three basic steps: 1. find candidate controllers; 2.

find candidate target sets; 3. find candidate baits. The algorithm has five main MapReduce (MR) jobs of the basic three algorithms.

It is worth noting that we can set several thresholds in Algorithm 4 to filter out records. For example, in MR job 2, we can choose the top-K with highest spikes, or we can set a threshold on the spike/average ratio. And in MR job 5 the threshold of percentage to filter out candidate baits IP is also configurable.

**Algorithm 4** Finding possible watering hole attack patterns in NetFlow data set

```
   Step 1: Find candidate controllers
1: MR aggregation job 1: Aggregate by destination IP + short time interval, and get count of incoming connections to the destination IP during the short time interval.
2: MR top-k job 2: For each destination IP, get the histogram of counts against short time interval series, and output those destination IPs + short time intervals that have the top-K highest spikes in counts, as candidate controllers.

   Step 2: Find candidate target set of each candidate controllers
3: MR filtering job 3: For each candidate controller found in Step 1, find those source IPs in the original dataset that have connections to the candidate controller during its short time intervals accordingly, as candidate target set of that candidate controller.

   Step 3: Find candidate baits, identify possible watering hole attack patterns
4: MR filtering job 4: With the candidate target set found in Step 2, find those IPs that have connections to IPs in candidate target set before the short time intervals, as candidate baits.
5: MR counting and filtering job 5: For each candidate bait found in MR job 4, count the number of different IPs that it has connection to in each candidate target set. If the candidate bait has connection to more than a certain threshold percentage of the candidate target set of a candidate controller, output the
   (candidate controller, candidate bait, candidate target set)
   as possible watering hole pattern.
```

## V EVALUATION

We evaluate the performance of Hadoop for NetFlow data analysis in three contexts: MapReduce-based exploratory analysis on large multicore server, (2) SQL-based exploratory analysis in a public cloud (Amazon Web Services), and (3) pattern discovery. We chose these contexts as representative of typical approaches to using Hadoop for NetFlow analysis based on published literature [7]. In Section IV, we identify building blocks used for analyzing NetFlow data and here we examine their the performance. As such, we expect the results here provide insight into Hadoop performance across broad range of NetFlow analysis.

## 1 METHODOLOGY

Our multicore evalution platform consits of a large server with two AMD Opteron 6272@1.4GHz processors, each with 16 cores and 64GB of RAM. The server also has eight 3TB@7200rpm hard disk drives (HDD). The operating system is Linux x64 3.11.6 (Fedora 19), and we perform all experiments using Apache Hadoop 2.2.0. We configure Hadoop framework to allow a maximum of 30 concurrent tasks and up to 1GB of RAM per task. For all the experiments, HDFS is configured to use all available disks and to use a fixed block size of 128MB for all data.

We use Amazon Web Services (AWS) as a second evaluation environment. All the experiments are performed using a cluster of three node, with one as the master. The type of each node in the cluster is M1.large, each with 4 virtual cores and 15GB of RAM. Similar to multicore environment, Apache Hadoop 2.2.0 is used. We leave all Hadoop configuration unchanged as set by Amazon.

For all experiments, we use CAIDA NetFlow dataset. The original dataset is provided in NetFlow format which is in plaintext format with fields are separated by white space.

We measure the performance in terms of time for query execution.

## 2 LARGE MULTICORE SERVER

### 2.1 DATA REPRESENTATION

In this experiment, we evaluate the effect of data representation on the framework performance. We use record counting as a type of analysis that lacks any computation overhead, and thus it can be used to measure framework bandwidth. Here we run the record-counting program based on Algorithm 1, which simply traverse the data set once in map phase, and use one reducer in reduce phase to collect all the output from map phase and get the count of records.

Figure 4 shows the running time of using record-counting program on different size of data and with different NetFlow serializations.

We can see from Figure 4 that, for example, a 20GB plain text data set can be reduced to only 14.6% (2994MB) when stored as CompressedSequenceFile. And for SequenceFile, which we implemented by converting each field of the records to SequenceFile format data as illustrated in Figure 2, is around half the size of the original data set. So we can reduce the size of the original text data set by around a significant 85% with the CompressedSequenceFileformat. Thus, relatively simple and repetitive nature of NetFlow text format makes this format amiable to compression, significantly reducing the storage footprint, ans consequently, increasing the rate at which records can be transferred to main memory.

However, the downside of a more compact representation from compression is increased processing time due to decompression processing and an additional layer of expensive parsing needed when using CompressedSequenceFile. In the Figure 4, we quantify the added overhead due to use of compression. We measuring the running time of record-counting program. We can see that for data set smaller than 5GB, the running time on text data is faster. This is because:(1) savings in I/O due to data size is too small to compensate for the overhead of decompression time; (2) for CompressedSequenceFiledata set, the data size is 748MB, thus Hadoop only uses 7 mappers for the job due to the 128MB block size limit per mapper. This results in decreased parallelism, which is also why the running time on SequenceFiledata is almost the same as on text data. On the other hand, for data size larger than 10GB, we can see that the running time on CompressedSequenceFiledata is shorter than both on SequenceFileformat data and text data. This is when the data size starts to play the major role, even at the cost of compensating the decompression cost of the CompressedSequenceFile. From this experiment we can see the benefits of CompressedSequenceFileis that it not only reduce the text data size significantly.

In the rest of the experiments in this section, we use

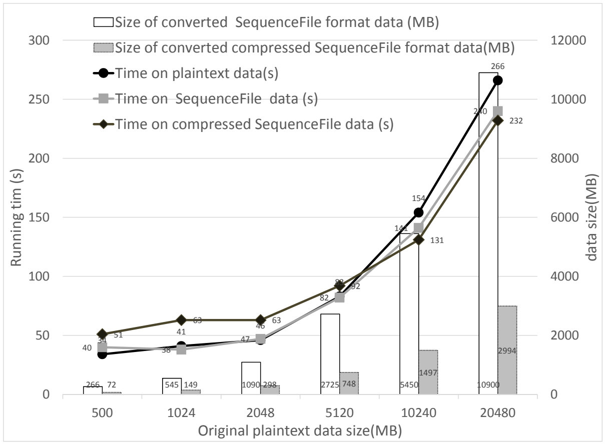

Figure 4: Running time of record-counting program on different serialization schemes for NetFlow records. We observer that only for significant amount of data, 5GB in this case, it is worth using more sophisticated serialization formats.

SequenceFileserialization.

## 2.2 AGGREGATION

In this section we are interested in the effects of different number of reducers on the computing top-K results scenario. As explained in former section, we have to run one reducer to get top-K results, but that contradicts the parallelism we want. So the solution is to chain multiple jobs, with only last job using one reducer. Here we use chain 2 jobs and vary the number of reducers the first job to explore the difference it make.

Figure 5 shows the running time of computing top 10 source IPs with the largest data volume on the 5 GB data set with different number of reducers in the first job. We can see that the running time with only 2 reducer is the longest, but as number of reducers grow, the running time gets shorter. Because the first job runs faster with more reducers. However as number of reducers continues to grow after 8, the running time get longer, this is due to the overhead of launching reducers. This experiment shows that for computing top-K on large data set, it is necessary to avoid running only one job with only one reducer, but we may need to set the number of reducers properly with respect to the limitations of resource we have. The optimal setting for number of reducers depends on the workload of reducers and the system's capacity. A plausible setting can be found at Hadoop

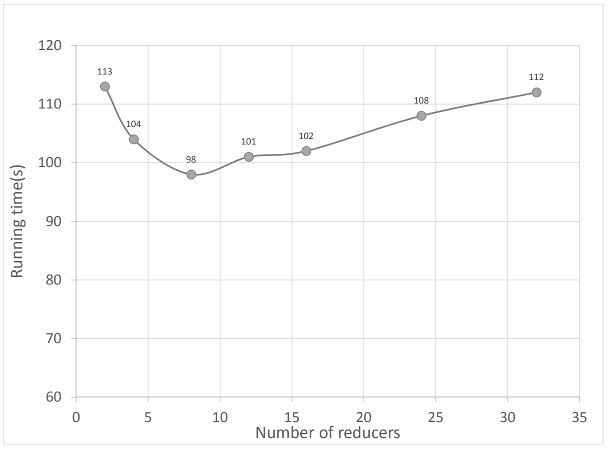

Figure 5: Running time of computing top 10 source IPs with the largest data volume, with different number of reducers. We add the one job approach as comparison.

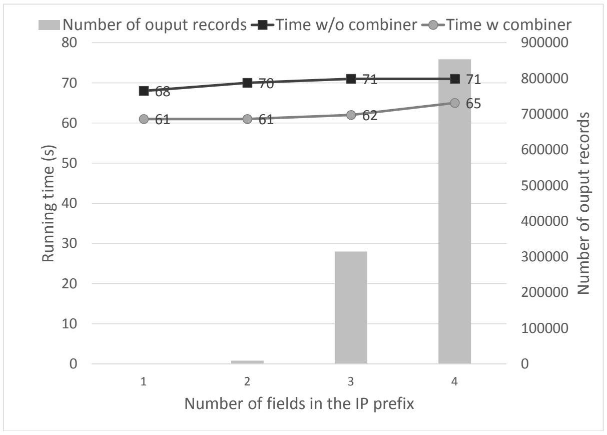

Figure 6: Running time of aggregation by source IPs(or prefix of IPs).

official website[3].

### 2.3 SCALING UP AGGREGATION

In this experiment we evaluate the effect combiners make on aggregation operations. Here we choose to aggregate by source IPs and get the data volume of each IP. Similarly from former test with different data formats in Figure 4, we choose to run on SequenceFiledata set.

Figure 6 shows the running time to aggregate source IPs (or prefix of IPs i.e., IPs that have prefix of “127.0”, or IP sets “127.0.x.x”) and get the sum of data flow from those IPs (or IPs sets with the same prefix). This can be used to get total data volume flow from subnets. We change the aggregation group size by varying the number of fields in the IP prefix. To demonstrate the effect of combiners, we also run the test with combiners to do mapper-local reduce job. We can see that with combiner, the running time decrease by 14% in the prefix=2 case. So combiners does help in this case, and combiners can make the performance worse if the extra cost of running combiners does not exceeds the benefits of reduce mappers’ output when there are not so many duplicate keys to combine.

[3]http://wiki.apache.org/hadoop/HowManyMapsAndReduces

## 3 AMAZON CLOUD

We perform similar experiments on SQL from Amazon Web Service and evaluate the results. Experiment environment is explained in former subsection Methodology 1.

In this part we evaluate SQL on different data formats: text format, RCFile format and compressed RCFile format, by using the SQL record counting program in former section on different data set size and compare the results. Similar to the experiments above, we get the results as shown in Figure 7.

From Figure 7, we can see that the compressed RCFile serialization reduces 20GB datset to 5GB, a 75% reduction. For smaller data set (less than 1024MB), running time is almost the same for all three types of files. With larger data sets (larger than 10GB), running time on compressed RCFile data set is shortest. These trends are consistent with experiments on the multicore server.

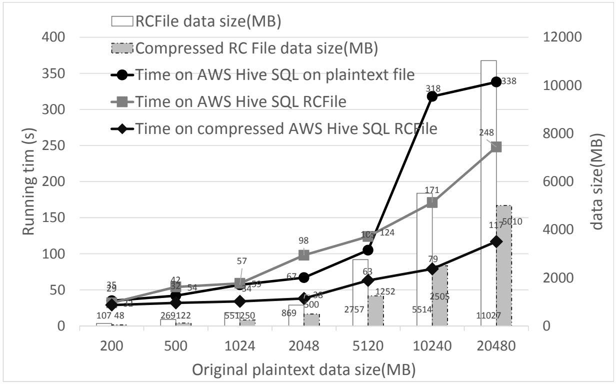

Figure 7: Running time of record-counting program on original plaintext data set, RCFile data set, and compressed RCFile data set against the size of RCFile data set and compressed RCFile data set.

## VI CONCLUSIONS AND FUTURE WORK

In this paper, we used Hadoop to analyze NetFlow data and evaluate the efficiency of different data formats and we explored the MapReduce implementation of several basic yet fundamental analysis of Netflow data set. We found that the compact binary representation in HDFS using SequenceFile efficient in reducing space usage and access time for files of big size, and compressed SequenceFile even more efficient in both aspects. We also explored the shared state aggregation and experimented with different number of reducers, and we concluded that shared state aggregation is sensitive to Hadoop setup, and it is important to identify correct number of reducers for a particular job and system resources. We also evaluated the effects of combiners and concluded that combiners can help scale up aggregation. We also utilized the cloud computing service–the Amazon Web Service–and conducted similar experiments used HiveQL on RCFile format. We concluded similarly that compressed RCFile is also efficient just as compressed SequenceFile format.

In future work, we plan to explore more complex analysis on NetFlow data set using Hadoop, such as network attack pattern mining. We also plan to conduct such experiments similarly on AWS and explore other tools that AWS has to offer to NetFlow data analysis.

## References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] J. Francois, S. Wang, W. Bronzi, T. Engel *et al.*, "Botcloud: Detecting botnets using mapreduce," in *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on.* IEEE, 2011, pp. 1–6.

[3] A. Saranya, M. Saravanan, and K. Kumar, "Mitigating the traffic in a cloud using clustering," in *Advances in Engineering, Science and Management (ICAESM), 2012 International Conference on.* IEEE, 2012, pp. 219–223.

[4] X. Mu and W. Wu, "A parallelized network traffic classification based on hidden markov model," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on.* IEEE, 2011, pp. 107–112.

[5] P. Nemeth, "Flowy improvement using mapreduce," 2010.

[6] C. Wagner, J. François, T. Engel *et al.*, "Machine learning approach for ip-flow record anomaly detection," in *NETWORKING 2011.* Springer, 2011, pp. 28–39.

[7] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes, "A survey of network flow applications," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 567–581, 2013.

[8] Y. Lee and Y. Lee, "Toward scalable internet traffic measurement and analysis with hadoop," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 5–13, 2012.

[9] J. T. Morken, "Distributed netflow processing using the map-reduce model," Ph.D. dissertation, Norwegian University of Science and Technology, 2010.