Fine-tuning Resource Allocation of Apache Spark Distributed Multinode Cluster for Faster Processing of Network-trace Data

Shyamasundar L B*[1]
Department of CSE
CMR Institute of Technology
Bengaluru, Karnataka, India

V Anilkumar[2]
CSIR-Fourth Paradigm Institute
NAL Belur Campus
Bengaluru, Karnataka, India

Jhansi Rani P[3]
Department of CSE
CMR Institute of Technology
Bengaluru, Karnataka, India

***Abstract*—In the field of network security, the task of processing and analyzing huge amount of Packet CAPture (PCAP) data is of utmost importance for developing and monitoring the behavior of networks, having an intrusion detection and prevention system, firewall etc. In recent times, Apache Spark in combination with Hadoop Yet-Another-Resource-Negotiator (YARN) is evolving as a generic Big Data processing platform. While processing raw network packets, timely inference of network security is a primitive requirement. However, to the best of our knowledge, no prior work has focused on systematic study on fine-tuning the resources, scalability and performance of distributed Apache Spark cluster (while processing PCAP data). For obtaining best performance, various cluster parameters like number of cluster nodes, number of cores utilized from each node, total number of executors run in the cluster, amount of main-memory used from each node, executor memory overhead allotted for each node to handle garbage collection issue, etc., have been fine-tuned, which is the focus of the proposed work. Through the proposed strategy, we could analyze 85GB of data (provided by CSIR Fourth Paradigm Institute) in just 78 seconds, using 32 node (256 cores) Spark cluster. This would otherwise take around 30 minutes in traditional processing systems.**

***Keywords*—Big data; packet data analysis; network security; distributed apache spark cluster; Yet Another Resource Negotiator (YARN); parameter tuning**

I. Introduction

Big data could be defined as data with high variety, volume, velocity and veracity information assets [1]. It claims optimal, cost effective and innovative techniques of processing information. Which in turn provides better insight and much better decision making. Such processing is difficult to conduct using centralized approaches in a highly scalable, high-throughput and fault-tolerant way [2].

In the proposed work, the PCAP data analysis will be implemented on top of a SPARK cluster which is deployed over Hadoop YARN (Yet Another Resource Negotiator).

The proposed work analyzes large amount of network capture data that has been collected for a period of four months (which constitutes to Big data). The collected data will be in *.pcap* format. The complex PCAP analysis includes processing huge amount of data collected. The processing must be performed on large stored datasets by an analyst to detect security incidents or to perform security audits.

A. Apache Hadoop versus Apache Spark

Apache Hadoop uses MapReduce processing framework, by using YARN for cluster management and Hadoop Distributed File System (HDFS) for distributed storage. Hadoop MapReduce provides fault-tolerant and distributed execution of 'jobs', which includes the following processing steps:

1) Reading the input data from HDFS blocks and splitting them to mappers.
2) Map: Applies an user-defined function and outputs one file per mapper.
3) Combining output from the mappers (using user defined functions), which is optional.
4) Partition, shuffle, sort and merge the data into the reducers.
5) Reduce the data.
6) Output one file to each reducer and store data into HDFS.

Tasks where Hadoop MapReduce is chosen over SPARK:

- *Processing huge data sets in a linear manner:* MapReduce allows processing huge amount of data parallelly, where large chunk of data is broken into smaller ones. They are processed separately on the data nodes. The results are gathered automatically across multiple nodes and then a single result is returned. If the dataset is bigger than the available RAM, then MapReduce may outperform SPARK.
- *When immediate results are not expected:* MapReduce will be a good solution only if the processing speed is not critical.

Tasks where Apache Spark is chosen over Mapreduce:

- *Faster data processing:* In-memory data processing makes Spark much faster than processing using MapReduce. In essence, 10 times faster processing of data in storage and 100 times faster using Random Access Memory (RAM) [3] [4] [5].
- *Iterative data processing:* If the data is processed often, then SPARK can be chosen over Mapreduce. Multiple map operations can be done using Spark's Resilient Distributed Datasets (RDDs), whereas MapRe-

duce has to write intermediate results back to a disk, which increases input/output (I/O) overhead.

- *Near real-time data processing:* If immediate insights are needed, then Spark should be opted since it performs in-memory processing.
- *Processing graphs:* Computational model of Spark is good to perform iterative computations since it has GraphX, which is dedicated for graph computation.
- *Machine learning tasks:* Spark has built-in machine learning library - MLib which has out-of-the-box algorithms that run in memory. Whereas, Hadoop Mapreduce requires a third-party to provide a library for machine learning. Also in SPARK, there is a provision to tune and adjust the algorithms.
- *Dataset joins:* Spark can perform combinations much faster, while Hadoop requires many shuffles and sorts for joining datasets.

B. *Motivation for the Work*

In recent times, Apache Spark in combination with Hadoop is evolving as a generic big data processing platform. While processing raw network packets, timely inference of network security is a primitive requirement. However, to the best of our knowledge, no prior work has focused on systematic study on fine-tuning the resources, scalability and performance of Apache Spark cluster in a multi-node environment (while processing PCAP data). For obtaining best performance, various cluster parameters like number of nodes in the cluster, number of cores utilized from each node, total number of executors run in the cluster, amount of RAM used from each node, YARN executor memory overhead allotted for each node to handle garbage collection issue, etc. have been fine-tuned, which is the focus of the proposed work.

II. Background and Related Work

Apache spark is specially designed for handling big data processing problems, which can analyze data in a very less time. It is a cluster computing platform that is open source and designed for processing big data. It provides an user friendly application program interface (API) to write queries and handle the jobs [6]. It has basic functionalities such as memory management, task scheduling, interaction with storage systems, fault recovery and RDDs, which are the main programming abstraction. For parellel processing of data, a set of functions are distributed across memory, which involves transformations and actions. *Transformations* include reduceByKey, distinct, map, intersection, join, union, filter, aggregateByKey, sortByKey and so forth. *Actions* include collect, saveAsTextFile, count, first, countByKey, takeSample, foreach and so forth [7].

Apache Spark elegantly handles iterations, memory availability and is suitable for processing both stream and batch jobs. Overall it outperforms Hadoop by orders-of-magnitude for several applications.

As illustrated in Fig. 1, Spark has three major layers. *Spark Core* - a generalized layer where all the basic functions are defined and all the other extensions and functionalities are developed on top of Spark Core. *Spark Ecosystem* contains additional libraries to operate on top of DataFrames and Spark Core. These components give power in the fields of machine learning, Structured Query Language (SQL) capabilities, real time processing of big data etc.

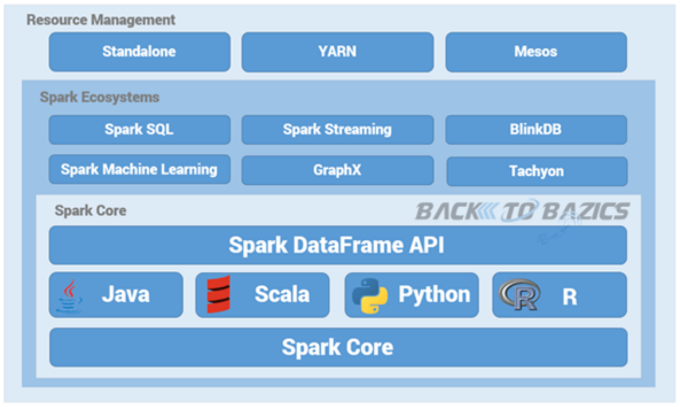

Fig. 1. Apache Spark Architecture and Ecosystem

A. *Job Execution on a Spark Cluster*

When a job is submitted by a driver process, first the request is sent to the *YARN Resource manager*. YARN checks for data locality and performs task scheduling by finding best available slave nodes. Then the submitted job will split into several stages. Each stage in-turn splits into multiple tasks, based on available resources and data locality. *Driver* daemon will send the necessary details related to the job to each node, prior to execution of task. Currently executing tasks are tracked by the driver and updates are sent to the *master* node. This can be checked with Master Node's user interface. Aggregate values from all the nodes are shared with the master node, once the job is completed. Hadoop and Spark coexist in the same cluster as shown in Fig. 2.

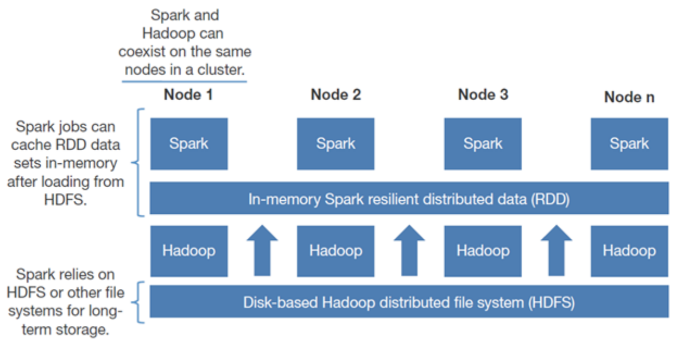
Spark and Hadoop can coexist on the same nodes in a cluster.
Node 1
Node 2
Node 3
Node n
Spark jobs can cache RDD data sets in-memory after loading from HDFS.
Spark
Spark
Spark
Spark
In-memory Spark resilient distributed data (RDD)
Hadoop
Hadoop
Hadoop
Hadoop
Spark relies on HDFS or other file systems for long-term storage.
Disk-based Hadoop distributed file system (HDFS)

Fig. 2. Apache Spark and Apache hadoop co-existing on the same cluster

There exists an *ApplicationMaster* process for each application instance in YARN. Application requests resources from *ResourceManager* and informs *NodeManagers* to initiate containers on its behalf, after allocating the resources. The *Spark driver* will run in the *ApplicationMaster* on top of cluster host, once the resources have been allocated. The process is shown in Fig. 3 and 4.

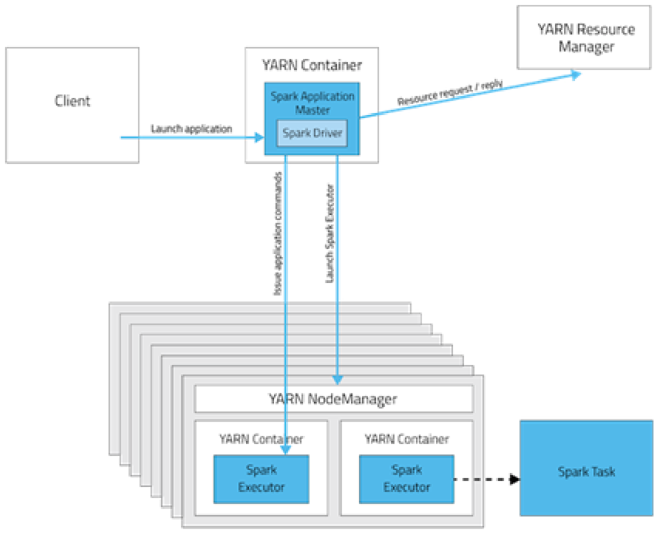

Fig. 3. Spark daemons running when deployed on top of YARN

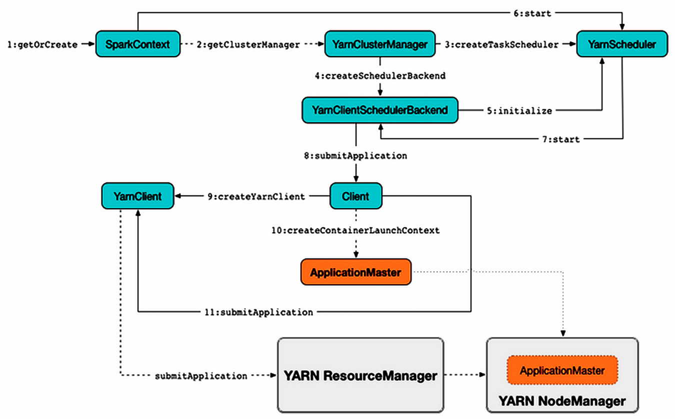

Fig. 4. Submitting Spark Application to YARN Cluster (aka Creating SparkContext with yarn Master URL and client Deploy Mode)

Spark includes all the necessary transformations that are pipe-lined in just a single stage for boosting the performance. Then, the data must be *shuffled* among different stages [8]. When a shuffle is done, in-memory output will be shuffled from previous stage to the storage system. Extra data will be stored to disk, if insufficient memory is allocated and intermediate data is transferred across the network. When employing more machines, due to shuffling overhead, it will achieve speed-ups much smaller than 'm', the number of machines (nodes); In general, overhead incurred due to shuffling is directly proportional to number of machine pairs, i.e.,

$$O(m^2)$$

.

Bachupally [9] has analyzed the network trace data and found anomalous connections made to the network by using HDFS. Some features of PCAP data with size 131 Megabytes (MB) were extracted using Wireshark and stored in HDFS as Comma Seperated Value (CSV) format. However, Wireshark has been used and very small data has been handled. Wireshark [10] is an opensource software and one of the most famous analysis tools for network packets. But there are some limitations.

In [11], authors present four different network monitoring tools that can monitor and analyze the network traffic. The disadvantage of Wireshark has been discussed. According to them, it will not detect malicious activities on the network and it means that Wireshark may not be useful to study network security. In addition, Wireshark cannot handle the large packet data. If the user has a large capture file more than 100MB, Wireshark will become slow while loading, filtering and alike actions.

In [12], attention is paid on the benefits of Apache Spark when compared to Hadoop MapReduce. In case of MapReduce, data will be read from disk and results are written to HDFS after a specific iteration. Then, the data will be read from HDFS for further iteration. The entire process utilizes lot of time and disk space. This results in the issue of lower fault tolerance and high latency of the entire system. To overcome these issues, Apache Spark is being used instead of MapReduce. The authors also focused on time-series data analysis using SPARK environment in real-time. Patterns were generated out of analysis, which in-turn gave a clear glimpse of characteristics and statistics of data. Thus, making MapReduce less efficient compared to SPARK. But the dataset under consideration is not related to packet capture data and the dataset used in the proposed work requires resource allocation to be tuned, which has not been explored.

In [13], SQL queries have been executed to analyze about 85 Gigabyte (GB) of network packet dataset provided by the University of New Brunswick. The way to analyze huge size of PCAP files on Hadoop and visualize the analyzed results on web browser by using Hue (Hadoop User Experience) has been discussed.The experiments were conducted using MapReduce, whereas experiments in the proposed method are done using SPARK. It has been set up on a standalone mode in Hadoop and SQLs are executed on a local machine, and not on a distributed cluster.

In [14] and [15], a method is proposed where input data is converted into RDDs. This allows in-memory computation on huge clusters of Spark, in a fault-tolerant way. Lazy transformations are applied to RDDs, which in-turn creates new RDDs and store them into HDFS or onto the driver. This method has been incorporated in the proposed work.

In [16], researchers integrate a network monitoring architecture into Apache Spark. NetFlow data has been processed as the way in traditional processing approaches. Moreover, implementation using stream processing will find out novel information which could not be found using traditional packet monitoring approaches.

Prakasam [17] expresses that, due to shortage of options in inter-stage communication facilities of processes, it makes MapReduce unsuitable for interactive workloads (Interactive Data mining, Stream data processing and Analysis) and iterative (Machine Learning and Graph Processing) processing. He dives into architecture of MapReduce and its disadvantages. At the end, alternatives such as Apache Spark and Apache Tez were used because of their suitability in interactive and iterative processing.

In [18] and [19], authors conclude that execution time of a particular job on Apache Spark platform can vary significantly depending on the input data type and size, method and implementation of the algorithm and computing capability (e.g., number of nodes, memory size, Central Processing Unit - CPU speed etc). This makes it extremely difficult to predict job performance, which is often needed to optimize resource allocation. Performance prediction can help to locate execution stages with abnormal resource usage pattern [20]. Although these issues are considered, the nature of PCAP data is different from their data and this requires different resource allocation mechanisms based on the number of cores to be allotted for data processing. This issue is resolved in the proposed work.

In [21], authors claim to propose a distributed Intrusion Detection System (IDS) based architecture that is capable of detecting anomalies in the network in real-time using Apache Spark framework and Netmap. But, Center for Applied Internet Data Analysis - CAIDA's dataset [22] has been used to detect Distributed Denail of Service (DDoS) attacks, which is an offline dataset. Their setup takes more than 3 minutes to detect an attack. In real scenarios, $>$ 3 minutes of time to find an attack is not an optimal solution. Proposed framework has a greater performance speed and the cluster can handle 3 GB of data in seconds, since it is possible to process 85 GB of data in 78 seconds. Also, the query that has been posed doesn't execute the jobs in parallel, which is the reason that their jobs take more time for processing.

In [23], authors claim that the analysis is done on real time data. But offline dataset "KDD 1999" has been used and supervised ML technique is used for processing the data. Also, the size of the dataset is not huge enough to fit into big data, which could be processed using traditional systems itself, without the need of Apache SPARK. The size of the dataset is just 743MB. But dataset considered in the proposed work is 85GB. Moreover, there is no mention about the configurations of SPARK cluster, regarding number of nodes in the cluster, number of cores, size of RAM on each system, etc... This makes it difficult to compare performance of the proposed work with their results.

None of the above referenced papers concentrate on fine-tuning the resource allocation of spark cluster, which enhances the performance by controlling excess usage of resources or limiting garbage collection.

The initial cluster startup takes around 12 seconds, independent of size of the cluster. In real world, applications run for dozens of minutes, which is an acceptable overhead. During each execution of the job, intermediate data will be produced which are stored on local disks and not on HDFS (as in [24]). This helps to yield the best performance.

From these research reviews, it is found out that one of the most challenging task is to handle the large packet data and analyze it. Thus, in this study, the designed analysis environment could handle the large packet data by using Apache SPARK on top of Hadoop YARN cluster.

III. Theory Framework and Modeling

A. Cluster Setup and Spark Application Submission

Spark scripts read a configuration file, describing the application and the Spark cluster configuration, provided by the user. This submits one or more jobs to the workload manager. Once a job is chosen by the job scheduler, a definite number of nodes are reserved for HDFS (if requested) and Spark cluster. Different services will be started after resource allocation procedure. The HDFS namenode service will be started if Distributed File System (DFS) is requested. Then all the HDFS datanode services will be started and connected to the NameNode. Spark cluster setup is done in a similar manner (wait for master node to be ready and then start the worker nodes). In Apache Spark, *master* node is the standalone Spark manager and *worker* nodes are nothing but Spark worker services, the place where the executors are launched.

B. Resource Allocation Schemes

Apache Spark doesn't provide any storage (like HDFS) or any Resource Management capabilities. It is just a unified framework for processing large amount of data near to real time [25]. It accesses Hadoop data store (HDFS) and runs on top of existing Hadoop cluster. Spark allows applications in Hadoop cluster to run up to 10x faster when running on disk and 100x faster when running in memory. This is possible by reducing the actual number of read/ write operations to and from the disc. Spark also gives the feature to quickly write applications in Scala, Java, or Python. This in return helps developers to create and run their applications in the programming languages they are more familiar with, thereby making it easy to build parallel apps.

Worker nodes will have the available resources (memory, CPU cores and disk). Master node is responsible for allocating the available resources to the required applications. Each application create executor processes where the tasks run in parallel. Resource allocation can be done using the following three mechanisms:

- **Default:** Here the applications are submitted without specifying any details of resource allocation. All the applications run in a First In First Out (FIFO) manner and every application consume the resources from all the worker nodes. Hence, when a single application is running, it will utilize all the worker nodes and create the executors.
- **Static:** Here, the user will specify the number of cores, executors, memory etc. that an application can have. The resources are shared among multiple applications submitted by one or more users.
- **Dynamic:** Here, applications may release executors that are idle and give back some of the resources to the Spark cluster. The free resources can be taken back in future, if required.

However, each one of the resource allocation schemes have some problems.

- Firstly, when only one application is running with default resource allocation scheme, it consumes all the resources. Hence, resources are not shared among applications.

- Secondly, in case of static resource allocation scheme, user manually sets the resources that each application will use.
- Thirdly, with dynamic resource allocation scheme, the initial amount of resources is still set by the user. Hence, incorrect resource allocation will cause severe performance issues.
  Finally, if any production cluster demands user-specific deadlines, then default allocation of resources may not work because application having a strict deadline has to wait in FIFO queue. Also, inappropriate allocation of resources in both dynamic and static resource allocation schemes will affect the deadlines.

C. SPARK_CSV (Comma Separated Value) Library Package

In the proposed scheme, network trace files in PCAP format are converted to CSV format to ease the purpose of querying the data files on top of SPARK framework. This package reads CSV files as Spark DataFrames. The API accepts many different options when reading the files:

- **path:** Location of files are specified here.
- **header:** When this variable has been set true, the first line in the file will be used for naming the columns and will be excluded from the data. By default, it's value is false.
- **delimiter:** Columns will be delimited using ',' (comma) by default. But any character can be set as delimiter.
- **quote:** The quote character is " (double quote) by default. But any character can be set. Delimiters inside the quotes will be ignored.
- **escape:** Escape character is \ (backslash) by default. One can set it to any other character.
- **parserLib:** It is "commons" by default and one can set it to "univocity" for CSV parsing.
- **mode:** Determines the mode of parsing. It is PERMISSIVE by default. Possible values include:
  PERMISSIVE: It parses all the lines. Missing tokens are inserted by null and extra tokens will be ignored.
  DROPMALFORMED: Drop the lines that have more or fewer tokens than expected or contain tokens that do not match with the schema
  FAILFAST: If any malformed line is encountered, it aborts with *RuntimeException.*
- **charset:** It will be 'UTF-8' by default and can be set to any other charset names.
- **inferSchema:** Column types are inferred automatically and by default it is set to false.
- **comment:** Lines starting with a specific character are ignored. "#" is the default. One can disable comments by having value for the variable as *null.*
- **nullValue:** A string is specified that indicates null value. Any other fields that match this string are set as nulls.
- **dateFormat:** Indicates format of the strings to use while reading timestamps or dates.

D. Tuning Resource Allocation

There exist some situations where even-though a 100-node cluster is setup and an application is run, only two tasks are executing. These kind of situations are not unfair, when a number of parameters influence resource utilization of Spark. So, in the proposed work, for the best performance the target is to make best use of the available resources from the cluster.

If resource allocation is not configured correctly, submitted job may consume all of the cluster resources and in-turn other applications will starve for resources.

The steps for a Spark job in a cluster mode include:

- *SparkContext* will connect to YARN cluster manager from the driver node.
- Resources are allocated to the cluster manager across other applications.
- Spark will acquire executors on the nodes in a cluster where each application will get it's own executor processes. Application code (python files/jar/python egg files) will be sent to executors.
- Tasks will be sent by *SparkContext* to these executors.

This has been shown in Fig. 5.

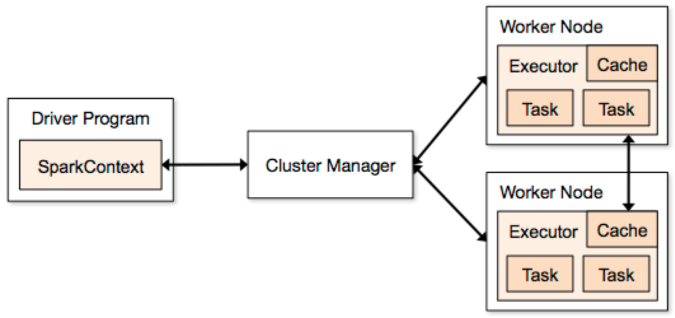

Fig. 5. Steps involved in cluster mode for a Spark Job

From the above mentioned steps, it is very clear that the total number of executors and their allocated memory setting will play an important role in spark's job performance. If executors are run with too much memory, it results in delays due to excessive garbage collection.

E. Utilized Dataset Description

In this paper, a collection of raw network packets obtained from a "/24" darknet setup has been utilized. A darknet is a set of globally routed and valid Internet Protocol (IP) addresses, which are not assigned to any host or devices. As the IP address space is not assigned to any hosts or devices, under ideal condition, no genuine traffic is expected to reach the darknet (Thus only malicious traffic has been captured, which constitutes to 85GB collected over a period of four months). However, Internet measurement and analysis in the past have shown that substantial amount of traffic arrives at darknet IP address space, and they are typically triggered by malicious activities like Internet wide port scanning, botnet recruitment and expansion process, reflections from IP spoofed Denail-of-Service (DoS) and DDoS attacks, etc. The darknet packets were collected continually for a period of 122 days (4 months) from 01/06/2017 to 30/09/2017. It consists of about 500 million ($\tilde{0}$.5 billion) Transmission Control Protocol (TCP) three-way handshake packets: Synchronize (SYN) request, SYN/Acknowledgment (ACK) response and ACK confirmation. The packets were originated from all over the world and were subjected to source IP address spoofing by responding to incoming SYN requests with appropriate SYN/ACK responses and validating the third ACK packet suing the sequence and acknowledgment number of these packets.

F. Utilized Testbed Description

The experiments were performed on a dedicated multi-node network testbed. It consists of 60 rack-mountable servers of 1U size and interconnected using 1 Gigabit per second (Gbps) Ethernet switches. Each node consists of one Intel Xeon E3-1230 v3 processor @ 3.30 GHz, with 8 MB cache. The processor consists of 4 cores and hyper threading is enabled on each core leading to an effective number of 8 CPUs available for concurrent job execution. Each node is also equipped with 2 numbers of 3 terabyte SATA hard disk drives at 7200 rotations per minute for storing the network packets. The testbed is hosted in a self-contained data center spread across three racks and equipped with inbuilt cooling, uniterrupted power supply, fire detection and suppression, water leakage detection, etc. As described under section 4, in order to determine the performance and scalability, multiple experiments are conducted with configurations of 1, 2, 4, 8, 16 and 32 nodes (leading up to 256 SPARK executors concurrently) in the testbed, with turbo mode off. HDFS and Apache SPARK are configured on the nodes. Each node runs Linux Operating System and Java version 8.

IV. Description of Models Involved, while Evaluating the Performance of Spark

A. Model to Estimate Execution Time

Since a job will be executed in several stages and each stage will be containing several tasks, jobs and stages can be represented as in equations 1 and 2, respectively.

$$Job = \{Stage_a \mid 0 \leq a \leq X\} \tag{1}$$

$$Stage_a = \{Task_{a,b} \mid 0 \leq b \leq Y\} \tag{2}$$

Where, the number of stages running within a job is X and number of tasks running within a stage is Y. Since different stages inside the job are sequentially executed, one can represent the time taken for executing a job as summation of execution times at each stage, *startup time* for the job and *cleanup time* for the job. This has been represented in equation 3.

$$SparkJobTime = StartupTime + \sum_{k=1}^{X}(Stage\_Time_k) + CleanupTime \tag{3}$$

In each of the stages, one core of CPU will execute a single task at a time. If a cluster has *W* worker nodes, one can calculate *P*, the number of parallel tasks as per equation 4.

$$W = \sum_{a=1}^{W}(Num\_Cores_a) \tag{4}$$

Where, the number of cores in CPU of a slave (worker) node *a* is *Num_Cores$_a$* and *W* is number of slave nodes within the cluster. Therefore, in each stage of execution, tasks will execute in batches and each batch contains *W* parallel running tasks. However, in a heterogeneous cluster, if the computing capacity of each slave is different, there will be an inherent uncertainty during execution which results in a significant variation in the execution time of the submitted job.

Hence, the time that has been spent in each stage can be calculated by doing a summation of the time taken for execution of sequential tasks in each stage, startup time of each stage and cleanup time of each stage, as represented in equation 5.

$$Stage\_Time = StartupTime + \max_{c=1}^{W}\sum_{a=1}^{S_c}(Task\_Time_{c,a}) + CleanupTime \tag{5}$$

Where total number of CPU cores (number of tasks running in parallel) is *W*, number of tasks running sequentially on each core is S_c.

Finally, since several tasks within a stage run with the same execution pattern, the time taken for executing a task can be calculated as per equation 6.

$$Task\_Time = Deserialize\_Time + Run\_Time + Serialize\_Time \tag{6}$$

Where, time taken for deserialization of input data is *DeserializeTime*, time taken for serialization of input data is *SerializeTime* and the actual amount of time spent to perform several operations on input data like *map, filter, etc.* is *Run_Time*.

B. Model to Estimate Memory Consumption

Since Spark performs in-memory computations, sufficient memory must be allocated to avoid execution slowdown, while creating the RDDs. Sometimes during configuration settings, with a lack of memory will lead to unexpected termination of the program execution. To avoid these adverse effects, a model to calculate the minimum amount of memory required for creating RDDs is proposed. Specifically, if *Y* tasks are running, once can define the total amount of memory required for execution of a job as summation of time required for executing each task. This has been defined by equation 7.

$$JobMemRDD = \sum_{a=1}^{Y}(TaskMemRDD_a) \tag{7}$$

C. Model to Estimate I/O Cost

Within a stage, current RDDs are generated by making use of previously generated RDDs using the transformation operation, *Shuffle_Map*. The result data is generated by performing *Result* operation. The cost of I/O operations involved in the above tasks can be categorized into two types, namely: *ShuffleReadCost* and *ShuffleWriteCost*. *ShuffleWriteCost* is the cost incurred while storing the interim data onto the disk buffer and *ShuffleReadCost* is the cost incurred while fetching interim data from the slave nodes.

Shuffle stage is the most I/O intensive stage that involve data transmission (reading data from the slaves) and fetching (storing data onto the disk) in a frequent manner. Stage wise I/O cost can be calculated by following equations 8 and 9.

$$Stage\_IO\_Write_a = \sum_{b=1}^{Y}(Task\_IO\_Write_{a,b}) \tag{8}$$

$$Stage\_IO\_Read_a = \sum_{b=1}^{Y}(Task\_IO\_Read_{a,b}) \tag{9}$$

D. Model to Predict the Performance

At first, calculate the time taken for execution of a job. For that, the actual number of tasks being executed is found out by equation 10.

$$Y = \frac{Input\_Size}{Block\_Size} \tag{10}$$

Where, *Input_Size* is nothing but total size of the dataset and *Block_size* represents the actual one data block stored in HDFS. As discussed, tasks will run batch wise within a job and the total number of tasks running is each batch is computed as per equation 4. If a cluster has nodes with different computing capabilities, The average amount of time taken to execute a task in a stage for a worker node w, can be calculated as per equations 11 and 12.

$$Task\_Run\_Time_{w,a} = Deserialize\_time_{w,a} + Run\_Time_{w,a} + Serialize\_Time_{w,a} \tag{11}$$

$$Avg\_Task\_Time_w = \frac{1}{n_w}\sum_{a=1}^{n_w}(Task\_Run\_Time_{w,a}) \tag{12}$$

Where, the number of tasks being run in the worker node within a stage of the submitted job *w* is n_h. During the experimentation, there were slight differences in the execution time of different batches within a particular stage. This can be calculated as per equation 13.

$$RatioOfTimeDifference_w = \frac{\frac{1}{n_w - W_w}\sum_{a=W_w+1}^{n_w}(Task\_Time_{w,a})}{\frac{1}{W_w}\sum_{b=1}^{W_w}(Task\_Time_{w,b})} \tag{13}$$

Where, the number of tasks that are running in a worker node *w* is n_w and the number of tasks running in a batch is W_w. Since tasks are executed on different worker nodes in parallel, for predicting the time taken for execution of a particular stage, the *StartupTime* and *CleanupTime* remain constant. Then, time taken for execution of a stage and task can be calculated as in equations 14 and 15.

$$Est\_Stage\_Time = StartupTime + \max_{c=1}^{W}\sum_{a=1}^{S_c}(Avg\_Task\_Time_{c,a}) + CleanupTime \tag{14}$$

$$Est\_Task\_Time_{c,a} = \begin{cases} Avg\_Task\_Time_c, & a = 1 \\ Avg\_Later\_Task\_Time_c, & a > 1 \end{cases} \tag{15}$$

Where, the total number of cores of CPU is *W* as per equation 4. The number of sequential tasks that are running in each core is S_c.

$Avg\_Task\_Time_c$ gives us the average time to execute batch of a CPU core within a worker node as in equation 12. The average time taken to execute following tasks of batches, $Avg\_Later\_Task\_Time_c$ can be calculated as in equation 16.

$$Avg\_Later\_Task\_Time_c = RatioOfTimeDifference_w * Avg\_Task\_Time_w \tag{16}$$

For predicting the I/O costs, the average of shuffle R/W costs incurred by a task is computed. Then, I/O cost incurred for a particular stage b can be calculated as specified in equations 17 and 18.

$$Est\_Stage\_IO\_Write_b = \sum_{w=1}^{W}(Y_w * \frac{1}{n_w}\sum_{a=1}^{n_w}(Task\_IO\_Write_{w,a})) \tag{17}$$

$$Est\_Stage\_IO\_Read_b = \sum_{w=1}^{W}(Y_w * \frac{1}{n_w}\sum_{a=1}^{n_w}(Task\_IO\_Read_{w,a})) \tag{18}$$

Where, the number of slaves (worker nodes) is *W*, Y_w is the total number of tasks that are running on worker node *w* and n_w is the number of tasks that are running on worker node *w*, at stage b.

Finally, calculate average RDD memory footprint required for each stage as defined in equation 19.

$$Est\_RDD\_Mem = \sum_{w=1}^{W}\left(\frac{Y_w}{n_w}\sum_{a=1}^{n_w} Task\_RDD\_Mem_{w,a}\right) \tag{19}$$

E. Estimation and Evaluation of Spark Performance, with an Example

Now, once could understand some basic analytics and flow of execution of an application in Spark, with an example.

Spark will schedule an application for execution, in a distributed cluster platform. RDDs generated within Spark will be used to identify the way temporary data and inputs are generating during computation phase. RDDs will be divided into equal sized partitions that could be configured to store on disk, in memory, or both. Multiple operations are pipelined within every partition to execute several operations in parallel. Since each partition will handle a task, the number of jobs running will be equal to the number of partitions created.

As illustrated in Fig. 6, considering a *join* operation, inputs are taken from multiple partitions. A stage will be created for the input data, so that the execution of the tasks are made to run in parallel manner. Considering the example above, *map* operation executes in parallel taking the RDD from Hadoop, on all the partitions, which will be aggregated to Stage 1 (grey part of the diagram). The *filter* operation also holds the same logic (green part of the diagram). Then, the *join* operation needs to wait for stages 1 and 2 to finish and after the results are obtained from these stages, a separate stage 3 will be scheduled. The intermediate results generated from mapped and filtered RDDs are logical RDDs, representing intermediate results.

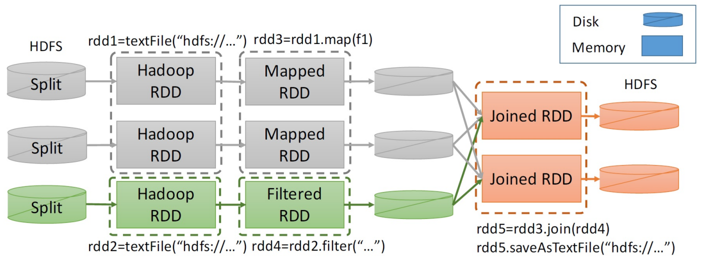

Fig. 6. Scheduling a Spark application for execution

If fewer machines are allocated, stages 1 and 2 cannot run concurrently and will run sequentially, thus the execution slows down. The end user of Spark application will have the flexibility to configure the storage location of these RDDs and also the size of them. If the RDDs stored in memory, care has to be taken so that entire RDDs fit into the memory. Otherwise, some partitions are spilled to the disk, or recomputed later, or even discarded.

In the phase of *Application Profiling*, several times an application is executed and entire dataset is divided into subsets of input data with an increased size (In real scenarios, this can be done using sampling), for evaluation of its dependency upon the important tuning parameters. During evaluation of performance, an entire dataset will be considered to estimate each stage's execution time from the collected information and then the entire application's performance is evaluated by considering the estimated time of execution for each stage.

The *Application Profiling* stage analyzes how tuning parameters will affect the overall performance of a Spark application in terms of storage and time taken to complete the entire task.

Now, consider the parameter *storage.memoryfraction* to analyze the parameters related to memory that affects the performance of a Spark application. In essence, Spark will divide available at every slave node into two important areas. The first part will account to 60% of the memory available by default, that will be allocated to caching of RDDs. The second part will account to 20% of remaining memory to the computations of *shuffle* stage. Remaining memory will be used for the computations of functions defined by the user. *storage.memoryfraction* parameter defines the memory being allotted to cache RDDs. If in case, some parts of RDDs does not fit in the main memory, those RDDs are spilled to the disk, or recomputed later, or even discarded. This will definitely affect the performance of the application.

To deeply understand the situation where entire RDDs does not fit into the main memory and to understand the effect of *storage.memoryfraction* parameter, some experiments were conducted. The entire dataset used for the experiments constituting to 85GB and minimal memory was considered, where all the RDDs will not fit into the main memory. From the results, it was observed that, the required time for completion of processing task increased dramatically (as per Fig. 7) when the memory utilized for caching RDDs, exceeds some value (i.e., 0.76).

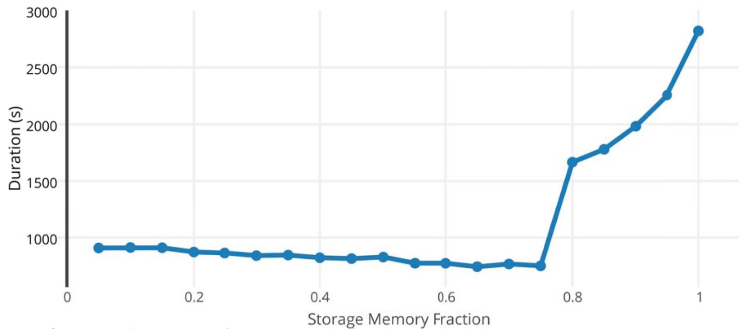

Fig. 7. Tuning *storage.memoryfraction* parameter

This concludes that, there was insufficient memory for *shuffle* operations and most of the memory was utilized for caching RDDs. In these cases, RDDs are spilled to the disk which incurs I/O overhead. Considering these cases, for the experiments conducted in the proposed work, care is taken that entire RDDs fit into the main memory to achieve the best possible performance.

V. Results and Discussion

In the proposed work, 4 months of network trace data which contains 85GB of data has been analyzed. This data has been processed in stages of 1 month, 2 months and 4 months. 32 nodes have been used from the test bed, each having 8 cores of CPU with 32GB of RAM. This data has been processed using different use-cases considering 5 cores, 7 cores and 8 cores of CPU from each node. With these setups, 7 cores per node achieved best results.

All Spark executors of an application will have same number of cores that are fixed. The total number of cores could be specified either with *–executor-cores* flag while invoking *spark-shell, spark-submit and pyspark* from command line, or by setting *spark.executor.cores* property in *spark-defaults.conf* file or through a *SparkConf* object. The cores control the number of tasks that an executor can concurrently run. For example, "*–executor-cores 7*" implies that each executor will run with a maximum number of seven tasks at the same time.

The *spark.executor.instances* configuration property or *–num-executors* command-line flag controls the total number of executors requested. One can avoid setting of this property by switching to dynamic allocation with *spark.dynamicAllocation.enabled* property, which enables a Spark application by requesting executors during backlog of pending tasks. This frees up executors when idle.

It is also important to tune how the resources requested by Spark could fit into what is available with YARN. The relevant properties of YARN are:

- *yarn.nodemanager.resource.memory-mb:* This controls maximum total memory that can be used by the containers present on each node.
- *yarn.nodemanager.resource.cpu-vcores:* This controls maximum total cores that can be used by the containers present on each node.

Fig. 8 shows a geographical distribution of the origin of the darknet malicious packets mapped using IP2Location mapping from the source IP address of the SYN requests.

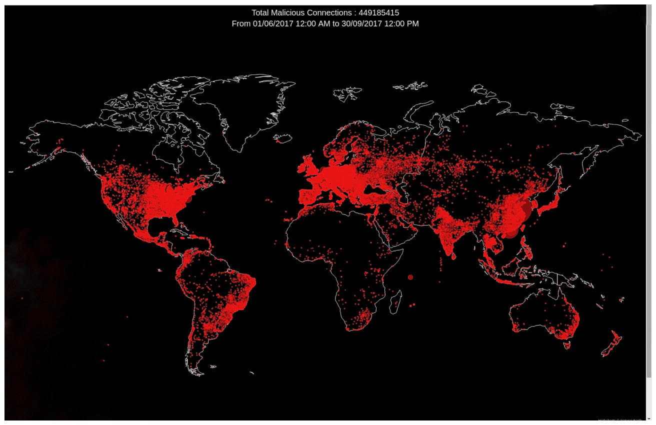
Total Malicious Connections : 449185415
From 01/06/2017 12:00 AM to 30/09/2017 12:00 PM

Fig. 8. Total malicious connections for a period of four months: ~449 million packets

To make all these a little more concrete, consider an example of configuring a Spark application to utilize most of the cluster resources; The proposed framework has a cluster with 32 nodes running NodeManagers, each equipped with 8 cores and 32GB of memory. To acheive the best performance, the NodeManager capacities, *yarn.nodemanager.resource.cpu-vcores* and *yarn.nodemanager.resource.memory-mb* should probably be set to 7 (cores) and 31GB * 1024 = 31744 (megabytes), respectively.

Consider the case *yarn.nodemanager.resource.cpu-vcores* with a value 8 and *yarn.nodemanager.resource.memory-mb* with value 32768MB (32GB * 1024). One must always avoid allocating 100% of the available resources to YARN containers, since a node requires some of the resources to run Hadoop and Operating System (OS) daemons. In this case, one has to leave a core and a gigabyte for these system processes. Hence, set the parameters *yarn.nodemanager.resource.cpu-vcores* and *yarn.nodemanager.resource.memory-mb* to 7 and 31744 respectively.

Running *yarn.nodemanager.resource.cpu-vcores* with a value *5* and having just enough memory required to run a single task will throw away the benefits that could come from running several tasks in a single Java Virtual Machine (JVM).

The results have been summarized in Tables I-IX. The same results have been visualized in Fig. 9-11.

Tables I-IX represent the time taken to process data in batches of one month (24GB), two months (42GB) and four months (85GB). This data has been analyzed by making use of 7 CPU-cores per node in the first set of experiments, then in the next set using 5 CPU-cores per node and finally all the 8 CPU-cores are used to process the data.
The difference in run-time between the number of CPU-cores utilized is mainly observed when the experiments are run on four months of data using a single node. Consider the following 3 cases:

- If we use 5 cores of CPU from a node, it takes 1346 seconds to process 85GB of data.
- If we use 8 cores of CPU from a node, it takes 1090 seconds to process 85GB of data.
- If we use 7 cores of CPU from a node, it takes 971 seconds to process 85GB of data.

In the first case, since some of the CPU cores are not at all utilized, it takes more time to process the data. Whereas in the second case, we are not leaving any core of CPU for Operating System and YARN daemons and all the resources are utilized only for Spark process. Hence it takes a bit more time to process the same 85GB of data (when compared to last case). Finally, leaving one core of CPU for OS and YARN daemons and utilizing the remaining cores to process the data will yield the best run-time performance.

TABLE I. TIME TAKEN FOR PROCESSING ONE MONTH DATA (24GB), USING 7 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (7 cores)	281
2 nodes (14 cores)	171
4 nodes (28 cores)	103
8 nodes (56 cores)	62
16 nodes (112 cores)	37
32 nodes (224 cores)	22

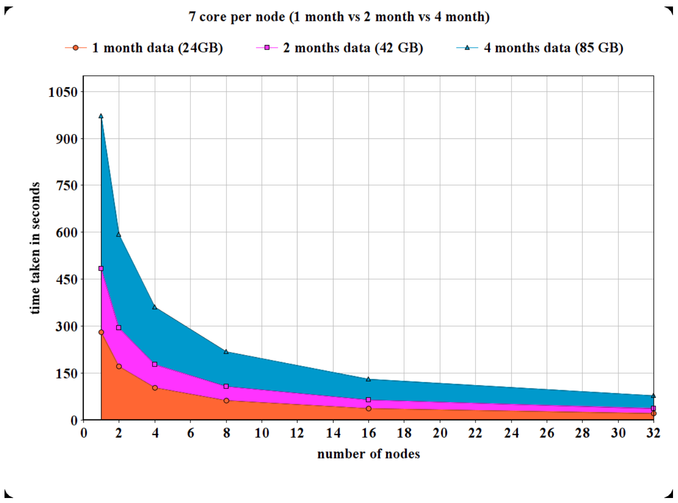

Fig. 9. 7 cores per node

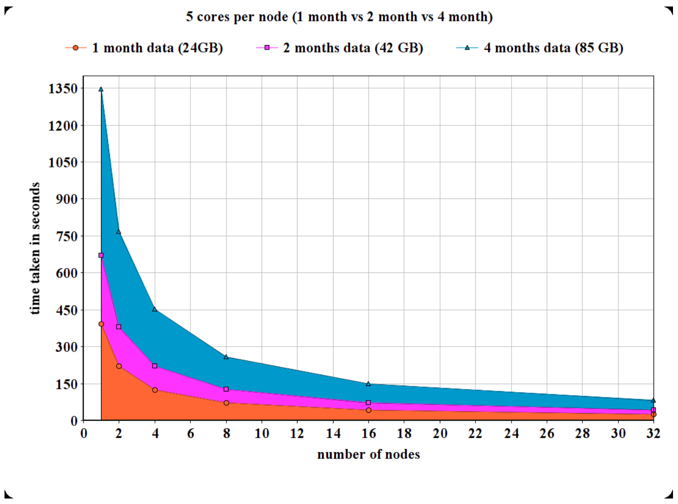

Fig. 10. 5 cores per node

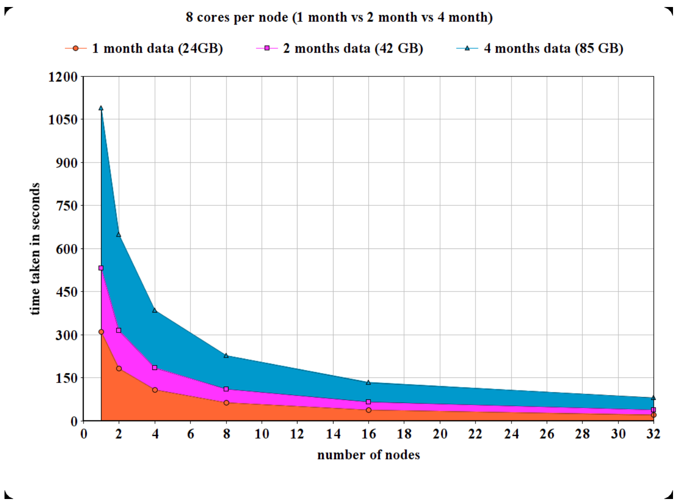

Fig. 11. 8 cores per node

TABLE II. TIME TAKEN FOR PROCESSING OF TWO MONTHS DATA (42GB), USING 7 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (7 cores)	483
2 nodes (14 cores)	294
4 nodes (28 cores)	178
8 nodes (56 cores)	107
16 nodes (112 cores)	64
32 nodes (224 cores)	38

TABLE III. TIME TAKEN FOR PROCESSING OF FOUR MONTHS DATA (85GB), USING 7 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (7 cores)	971
2 nodes (14 cores)	592
4 nodes (28 cores)	360
8 nodes (56 cores)	218
16 nodes (112 cores)	131
32 nodes (224 cores)	**78**

TABLE IV. TIME TAKEN FOR PROCESSING ONE MONTH DATA (24GB), USING 5 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (5 cores)	393
2 nodes (10 cores)	220
4 nodes (20 cores)	124
8 nodes (40 cores)	72
16 nodes (80 cores)	42
32 nodes (160 cores)	24

TABLE V. TIME TAKEN FOR PROCESSING OF TWO MONTHS DATA (42GB), USING 5 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (5 cores)	670
2 nodes (10 cores)	380
4 nodes (20 cores)	221
8 nodes (40 cores)	127
16 nodes (80 cores)	73
32 nodes (160 cores)	41

TABLE VI. TIME TAKEN FOR PROCESSING OF FOUR MONTHS DATA (85GB), USING 5 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (5 cores)	1346
2 nodes (10 cores)	768
4 nodes (20 cores)	452
8 nodes (40 cores)	257
16 nodes (80 cores)	148
32 nodes (160 cores)	83

TABLE VII. TIME TAKEN FOR PROCESSING ONE MONTH DATA (24GB), USING 8 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (8 cores)	310
2 nodes (16 cores)	183
4 nodes (32 cores)	108
8 nodes (64 cores)	64
16 nodes (128 cores)	38
32 nodes (256 cores)	22

When the same job was made to run on a traditional system without SPARK, having configurations of Intel Xeon E3-1230 v3 processor @ 3.30 GHz, with 8 MB cache, it took 29 minutes

TABLE VIII. TIME TAKEN FOR PROCESSING OF TWO MONTHS DATA (42GB), USING 8 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (8 cores)	531
2 nodes (16 cores)	315
4 nodes (32 cores)	186
8 nodes (64 cores)	110
16 nodes (128 cores)	65
32 nodes (256 cores)	38

TABLE IX. TIME TAKEN FOR PROCESSING OF FOUR MONTHS DATA (85GB), USING 8 CPU-CORES PER NODE

No. of nodes (cores)	Time taken in seconds
1 node (8 cores)	1090
2 nodes (16 cores)	648
4 nodes (32 cores)	385
8 nodes (64 cores)	228
16 nodes (128 cores)	135
32 nodes (256 cores)	80

and 8 seconds to process 85 GB of data.

VI. CONCLUSIONS AND FUTURE WORK

The nature of processing PCAP data is different from processing other data formats. This requires tuning resource allocation based on the number of cores, number of executors and the amount of memory to be allotted for faster data processing. This issue has been addressed in the proposed work.

In the literature, small amount of work is done in the field of processing network trace data using SPARK technology. 85GB of data has been analyzed in just 78 seconds using 32 node (256 cores) SPARK cluster, which would otherwise take around 30 minutes in traditional processing systems. Best results were achieved by allotting 7 (out of 8) cores of CPU per node and 31744MB of memory (leaving one GB of memory for OS and YARN daemons).

Spark avoids the file system to a greater extent. It retains most of the data distributed in memory, across multiple phases of the same job. RDDs hide the details of fault-tolerance and distribution for huge collections of items. RDDs apply the same operation of map, filter and join to many data items. RDDs are computed lazily for the first time they are used, so that it can later pipeline the transformations; In the proposed work, the focus is on the cases where aggregate memory can hold the entire input RDD within main memory, so that the queries (job) submitted can be executed faster.

A. Future Work

In all experiments, enough memory resources has been allocated so that RDDs can fit into the main memory. Thorough investigation of storage alternatives, especially when RDDs cannot fit in the main memory, are out of scope and left for future work.

As a future work, one can utilize the techniques to analyze the large network packet data in near real-time and apply some machine learning algorithms to develop near real-time automatic detection systems against network attacks.

REFERENCES

[1] Mashooque Memon, Safeeullah Soomro, Awais Jumani, and Muneer Kartio. Big data analytics and its applications. *Annals of Emerging Technologies in Computing*, 1, 10 2017.

[2] A. A. Cárdenas, P. K. Manadhata, and S. P. Rajan. Big data analytics for security. *IEEE Security Privacy*, 11(6):74–76, Nov 2013.

[3] Apache spark. https://spark.apache.org/.

[4] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Big data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences*, 30(4):431 – 448, 2018.

[5] Peter P. Nghiem and Silvia M. Figueira. Towards efficient resource provisioning in mapreduce. *Journal of Parallel and Distributed Computing*, 95:29 – 41, 2016. Special Issue on Energy Efficient Multi-Core and Many-Core Systems, Part I.

[6] Apache hadoop [Online]. http://hadoop.apache.org/.

[7] B. Akil, Y. Zhou, and U. Röhm. On the usability of hadoop mapreduce, apache spark apache flink for data science. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 303–310, Dec 2017.

[8] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015.

[9] Y. R. Bachupally, X. Yuan, and K. Roy. Network security analysis using big data technology. In *SoutheastCon 2016*, pages 1–4, March 2016.

[10] Wireshark. https://www.wireshark.org/.

[11] D. Mistry, P. Modi, K. Deokule, A. Patel, H. Patki, and O. Abuzaghleh. Network traffic measurement and analysis. In *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–7, April 2016.

[12] R. C. Maheshwar and D. Haritha. Survey on high performance analytics of bigdata with apache spark. In *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, pages 721–725, May 2016.

[13] Keisuke Kato and Vitaly Klyuev. Hadoop environment for the analysis of large network packets. In *Proceedings of the 2nd International Conference on Applications in Information Technology*, pages 56–59.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[16] M. Čermák, T. Jirsík, and M. Laštovička. Real-time analysis of netflow data for generating network traffic statistics using apache spark. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pages 1019–1020, April 2016.

[17] Prakasam Kannan. Beyond hadoop mapreduce apache tez and apache spark.

[18] S. Chaisiri, B. S. Lee, and D. Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing*, 5(2):164–177, April 2012.

[19] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[20] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 285–294, June 2012.

[21] A. M. Karimi, Q. Niyaz, Weiqing Sun, A. Y. Javaid, and V. K. Devabhaktuni. Distributed network traffic feature extraction for a real-time ids. In *2016 IEEE International Conference on Electro Information Technology (EIT)*, pages 0522–0526, May 2016.

[22] The caida ucsd ddos attack 2007 dataset, 2007. https://www.caida.org/data/passive/ddos-20070804_dataset.xml.

[23] M. Kulariya, P. Saraf, R. Ranjan, and G. P. Gupta. Performance analysis of network intrusion detection schemes using apache spark. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 1973–1977, April 2016.

[24] S. Michael, A. Thota, and R. Henschel. Hpchadoop: A framework to run hadoop on cray x-series supercomputers. *Cray USer Group (CUG)*, 2014.

[25] K. Wang and M. M. H. Khan. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 166–173, Aug 2015.