Datix: A System for Scalable Network Analytics

Dimitrios Sarlis
CSLAB, NTUA
dsarlis@cslab.ece.ntua.gr

Nikolaos Papailiou
CSLAB, NTUA
npapa@cslab.ece.ntua.gr

Ioannis Konstantinou
CSLAB, NTUA
ikons@cslab.ece.ntua.gr

Georgios Smaragdakis
MIT & TU Berlin
gsmaragd@csail.mit.edu

Nectarios Koziris
CSLAB, NTUA
nkoziris@cslab.ece.ntua.gr

ABSTRACT

The ever-increasing Internet traffic poses challenges to network operators and administrators that have to analyze large network datasets in a timely manner to make decisions regarding network routing, dimensioning, accountability and security. Network datasets collected at large networks such as Internet Service Providers (ISPs) or Internet Exchange Points (IXPs) can be in the order of Terabytes per hour. Unfortunately, most of the current network analysis approaches are ad-hoc and centralized, and thus not scalable.

In this paper, we present Datix, a fully decentralized, open-source analytics system for network traffic data that relies on smart partitioning storage schemes to support fast join algorithms and efficient execution of filtering queries. We outline the architecture and design of Datix and we present the evaluation of Datix using real traces from an operational IXP. Datix is a system that deals with an important problem in the intersection of data management and network monitoring while utilizing state-of-the-art distributed processing engines. In brief, Datix manages to efficiently answer queries within minutes compared to more than 24 hours processing when executing existing Python-based code in single node setups. Datix also achieves nearly 70% speedup compared to baseline query implementations of popular big data analytics engines such as Hive and Shark.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Monitoring; H.2.4 [**Systems**]: Query Processing

Keywords

sFlow, Hadoop, HBase, MapReduce, Map-Join, K-d Tree

1. INTRODUCTION

The Internet has become the dominant channel for innovation, commerce, and entertainment. Both Internet traffic and penetration increases at a pace that makes it difficult to track Internet growth and trends in a systematic and scalable way. Indeed, recent studies show that Internet traffic continues to grow by more than 30% annually as it has done the last twenty years and is expected to continue at the same pace in the future [2]. Yet, operators of Internet Service Providers (ISPs) and Internet Exchange Points (IXPs) have to analyze large network datasets to optimize parameters regarding network routing, dimensioning, accountability and security. ISPs serve, depending on their footprint thousands to tens of millions of end-users daily and facilitate billions of network connections daily [27]. IXPs consist of physical machines (core switches) to facilitate traffic exchange among different types of networks [14]. Some of the most successful IXPs, connect more than 600 networks and are handling aggregate traffic that is peaking at multiples of Terabytes per second. To put this traffic into perspective, on an average business day in 2013, one of the largest IXPs, AMS-IX in Amsterdam, exchanged around 25 PB while AT&T and Deutsche Telekom reported carrying 33 PB and 16 PB of data traffic respectively [14].

To monitor traffic at that scale, specialized technologies such as sFlow [6] or NetFlow [5] are used. An sFlow record contains Ethernet frame samples and captures the first 128 bytes of each sampled frame. This implies that in the case of IPv4 packets the available information consists of the full IP and transport layer headers (i.e., source and destination IPs and ports, protocol information, and byte count) and 74 and 86 bytes of TCP and UDP payload, respectively. These records are then collected in a centralized location for processing. NetFlow does not capture part of the payload but only source/destination IP and port, interface, protocol, and type of service information. Typically data scientists that analyze sFlow or NetFlow records rely on centralized approaches to execute queries. Centralized processing of these records is not scalable when it comes to processing multiple Terabytes of data. A number of distributed approaches [22, 12] have been proposed to tackle the scalability issue, but they are too slow to efficiently execute popular join and filtering queries, e.g., for a particular time period.

In this paper, we present the design and implementation of Datix, a network analytics system that utilizes a number of techniques to both accelerate network data (sFlow or NetFlow) processing and efficiently execute join and filtering queries. Datix is based on distributed techniques of data analytics such as MapReduce [15] and incorporates log processing techniques [11] to speed up join queries. Datix also efficiently handles data in the form of a star schema [17] that yields additional performance improvements. To evaluate the performance benefits that Datix offers we consider a number of popular join queries that combine the information from a main dataset, in our case being the sFlow data collected at an IXP, with additional complementary information provided by secondary datasets, such as the mapping of IP addresses to their corresponding AS, IP geolocation, and IP profile information, e.g., reverse DNS lookup as reported by Internet measurement studies such as ZMap [16].

Our contributions can be summarized as follows:

- We introduce a smart way of pre-partitioning the dataset

in files that contain records of a certain range of values, so as to facilitate data processing and query execution.
- Using this particular partitioning scheme we are able to efficiently execute filtering queries, for example across a certain time period, avoiding the need to process the entire dataset but instead accessing only the necessary files.
- We integrate these features into Datix, an open-source[1], SQL compliant network data analysis system by implementing distributed join algorithms, such as map join [11] in combination with custom-made user-defined functions that are aware of the underlying data format.

2. SYSTEM DESCRIPTION

Datix operates on top of cloud or dedicated computing resources and runs user queries on data that reside in distributed file systems such as HDFS [28] or HBase [13]. It uses either Hive over Hadoop [30] or Shark over Spark [32] to run distributed MapReduce jobs, generated by translating the user queries from an SQL-like language (HiveQL). Datix divides input datasets in two categories. The first category is the central (*log*) dataset which in our case is the sFlow data collected at the IXP for its operational purposes. This dataset is the central table of our star-schema data formulation and needs to be joined with several other datasets (*meta-datasets*) on one or several of its columns. In our IXP data case, as well as in the general log processing case [11], this dataset is expected to be orders of magnitude larger than the rest *meta-datasets*. Apart from the *log* dataset, Datix supports the import of several *meta-datasets* which, in this study, are the IP to AS and IP to Country mappings [3] and the IP to reverse DNS lookup mapping [4]. We choose to use these *meta-datasets* for our analysis because they are publicly available sources. Nevertheless, Datix supports the import of arbitrary *meta-datasets* even proprietary ones depending on each user's preferences. *Meta-dataset* sizes can range vastly according to the information they provide. In order to handle all *meta-dataset* cases, we divide them in two categories: (i) the small sized ones that are in the order of several Megabytes and can fit in the main memory of mappers (i.e., a single process entity in the MapReduce terminology) and (ii) the large sized ones which are bigger and do not abide to the main memory constraints of mappers. Small sized *meta-datasets* are stored in HDFS files while large sized ones are stored in HBase tables, indexed according to their star-schema join attributes.

By summarizing raw traffic data and joining it with *meta-data* using Datix, it is possible to efficiently answer popular operational queries, such as heavy hitter queries, e.g., which are the most popular IPs, AS-pairs, or ports by volume or by frequency of appearance, summary queries, e.g., the aggregated traffic per AS or IP, or range queries, e.g., which are the IPs that are active in two different time periods (when denial of service attacks took place) that are responsible for more than 1% of the overall traffic and more than 3% of the HTTP request traffic.

Figure 1 illustrates the architecture of Datix. A graphical interface assists the user to provide custom query parameters. Then, Datix takes over and appropriately rewrites the user input to HiveQL compliant queries (upper part of Figure 1) that can be forwarded to the Processing Engine layer comprised by either Hive or Shark. The Processing Engine layer handles the query execution by translating the HiveQL input to a sequence of distributed processing jobs that take input from the required Datix datasets stored in HDFS or HBase (lower part of Figure 1). In detail, Datix consists of the following four layers:

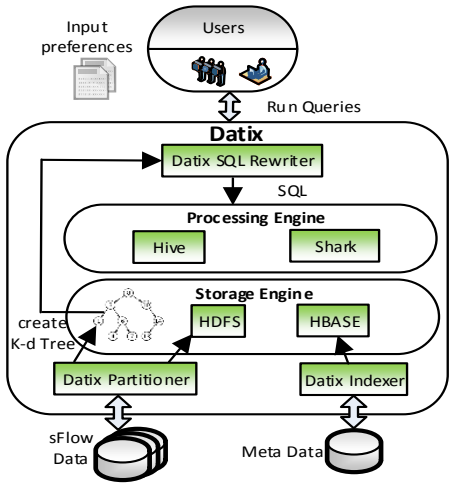

Figure 1: Datix Architecture.

Datix Partitioner/Indexer: This layer is responsible for pre-partitioning the datasets, according to user-specified partitioning attributes while utilizing a K-d Tree [24] (Section 3.3). These attributes can be either join attributes (e.g., source and destination IP) or filtering attributes (e.g., timestamp, protocol, port). Our pre-partitioning aims to achieve a number of optimization objectives:
- Meta-data required for the processing of each partition of the *log* data must fit in a mapper's main memory in order to perform efficient map-joins [11].
- Large sized *meta-datasets* should be efficiently indexed in order for mappers to be able to retrieve their respective meta-data with minimum overhead.
- Apart from join attribute partitioning the *log* dataset must be partitioned according to filtering attributes that are used to efficiently run queries on a subset of the *log* dataset.

Furthermore, large *meta-datasets* are indexed using HBase.

Storage Engine: This layer stores and indexes all necessary datasets. Each partition of the *log* dataset is stored in a separate HDFS file while the partitioning information (K-d Tree) is also stored in HDFS. As mentioned before, small *meta-datasets* are stored in HDFS while larger ones are stored and indexed using HBase.

Processing Engine: This layer is responsible for the distributed execution of HiveQL queries that take input from the respective Datix datasets residing in either HDFS or HBase. In particular, either Hive or Shark produces a multi-step plan for executing the query in accordance with the number of actions requested.

Datix SQL Rewriter: This module is responsible for translating the user input preferences into a HiveQL query that both Hive and Shark can interpret and execute. The Datix Rewriter utilizes custom made user-defined functions in order to inject Datix related code inside the HiveQL query execution. It also consults the K-d Tree partitioning scheme, stored in HDFS, in order to apply filtering constraints and reduce the query input to only the required partitions. One of the desired properties of the K-d tree is to locate the portions of a dataset that have to be processed and thus avoid unnecessary processing (Section 3.3). Hence, with this scheme it is possible to efficiently execute range queries e.g., for a particular time period.

[1] `https://github.com/dsarlis/datix`

3. DESCRIPTION OF ALGORITHMS

In this section, we give an overview of the suite of distributed join algorithms currently implemented in Datix. We present the two major types of algorithms implemented: (i) when the *meta-dataset* is small enough to fit in the main memory of a map task (Sections 3.1, 3.2), and (ii) when the *meta-dataset* does not fit in memory (Section 3.3). Even though Hive and Shark support map-joins, there are some prerequisites that have to be fulfilled. First, map-joins need to be equi-joins and second, one of the two tables should be small enough to fit in a mapper's main memory. In the following sections, we describe how Datix manages to overcome the aforementioned restrictions in each case.

3.1 Map equi-join

Problem Statement: For two tables, a large table L and a small table S, the goal is to perform the equi-join $L \bowtie_{L.c=S.c} S$ on a specific characteristic (column) c of the two tables, with $|S| \ll |L|$, so that S can fit in the main memory of a mapper task. This assumption holds true for the case of IP to AS and IP to Country mapping files which are less than 12 Megabytes in size each.

Instead of using the basic, shuffle join, implementation of Hive which requires a lot of disk I/O and data transfer, we utilize a join method based on the map-join technique [29] and more specifically, on Broadcast Join as described in [11]. Our map equi-join performs the join operation during the map phase. At each node, S is retrieved from HDFS and stored locally in the Distributed Cache of each mapper. Each map task then uses an in-memory hash table to join a split of L with the appropriate records of S.

In the beginning of a map task, it is checked whether S has already been stored in memory. If not, the hash table containing the key-value pairs of S is built. Then, while each record of L is processed, the map function extracts the join key and searches the hash table to produce the desirable join output. We note here that this process only transfers the small table S to all cluster nodes and thus avoids the costly data shuffling of the large table L, minimizing any time consuming data transfers. However, a possible drawback is that S has to be loaded several times, since each mapper runs as a separate process. This can be optimized by loading S only once per node, using a shared memory among mappers.

3.2 Map theta-join

Problem Statement: Similar to Section 3.1 consider the case where we have two tables L and S but now the goal is to perform a compound theta-join on a specific column, i.e., our goal is to compute $L \bowtie_{L.c \geq S.c_1 \wedge L.c \leq S.c_2} S$. The rational is that the files containing the IP to AS and IP to Country mappings consist of IP ranges that are part of an AS or country dataset and not a record for each IP address with its corresponding information. Thus, it is clear that we cannot perform an equi-join without blowing up the size of files from a few MB to some Gigabytes.

To perform the theta-join required, we must now use a different data structure rather than a hash table and so we choose to design our algorithm using an order-preserving data structure, such as a TreeMap [7]. The methodology, in this case, consists of the following steps. We transfer table S to each map task and import it in a TreeMap main memory structure. To produce the join results, for each record of L we extract the join key and perform a range search against the TreeMap structure. We integrate this functionality in Hive and Shark by using custom made user-defined functions that are responsible for the aforementioned operations.

3.3 Map equi-join with large meta-dataset

Problem Statement: The definition of the problem is almost identical to this in Section 3.1 except for the fact that now S is quite large and cannot fit in a mapper's main memory as a whole. An example of this restriction is the IP to reverse DNS lookup mapping file which is around 57 Gigabytes in size. In this case, we must follow a slightly different approach to implement a map-join and avoid unnecessary data transfers. In particular, the key idea here is that each sFlow data file contains a limited number of unique IP addresses and thus, it is not required to transfer the entire mapping file into memory but only the portion that contains the information about these unique IPs (see semi-join in [11]). Our approach is to pre-partition the sFlow data. Knowing beforehand the range of IPs in each file we can retrieve the respective records from HBase. Thus, utilizing the order-preserving storage of HBase we can perform range scans and transfer only the required IP to reverse DNS (IP-DNS) pairs into a mapper's memory.

Pre-partitioning can be performed using various approaches but we turn our attention towards two methods:

Method 1: Static partitioning. The first approach in partitioning is to use a uniform partition scheme across the join fields (IP addresses) of the *log* dataset. The meta-data required for each partition of the *log* (sFlow) dataset should fit in the main memory of a map task. In our IXP use case, we divide the IP-DNS *meta-dataset* in chunks of IP-DNS pairs that can fit in memory, and then apply the same uniform partitioning to the sFlow data. We must take into account that each sFlow file contains a source and a destination IP. Thus, the partitioning needs to be in two dimensions with each one corresponding to one of these IP types. This way of partitioning the dataset is quite straightforward. However, it fails to produce balanced output sFlow data files. This particular partitioning scheme does not consider the distribution of IP addresses in the sFlow files and hence, which IP pairs tend to exchange more traffic than others. Furthermore, the number of output files is quite large without being equally balanced in size (actually a lot of files end up empty) which results in poor performance during the partitioning phase. The implementation consists of two steps:

(i) Partitioning: A MapReduce job is responsible for performing the actual partitioning after defining the split points from the mapping file and produces sFlow files containing records with IP addresses in a given range as well as files containing the actual unique IPs in every chunk. This process is done off-line before any actual query is issued.

(ii) Query: The second step is to integrate the creation of the hash table containing the unique IP to reverse DNS lookup pairs and the equi-join implementation logic in a user-defined function which is aware of the underlying partitioning scheme.

Method 2: Dynamic partitioning. The second approach overcomes the restrictions and problems of the aforementioned method by using a dynamic data structure for partitioning the dataset, called K-dimensional Tree (K-d Tree) [24]. This data structure is well suited, among other alternatives (e.g., R-Tree), for space partitioning such as the one we are interested in for the following reasons: (i) it leaves no empty space when partitioning the data, (ii) it

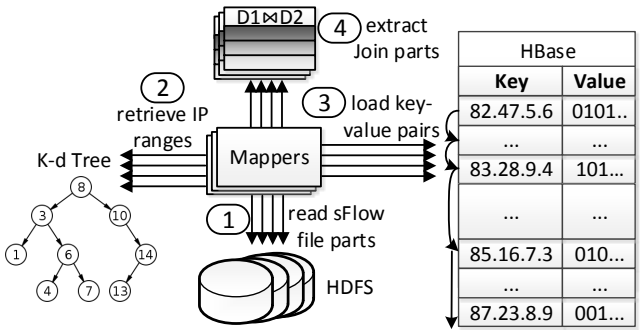

Figure 2: Join Execution using K-d Tree.

ensures that all output files will contain a balanced number of records, and (iii) it allows multiple dimensions in data partitioning. This last feature leads, naturally, in an efficient way of executing multi-dimensional filtering queries. Although the K-d Tree structure does not perform efficiently when the number of dimensions increases beyond a certain limit, it is suitable for our cause since the number of dimensions for the important analytics queries in the sFlow dataset does not exceed 10. For values close to 10 the performance of K-d Tree is slightly reduced but nevertheless it is still quite acceptable. There are three steps:

(i) Sampling: First, a relatively small (about 1%) sampling of the input data is performed to create the K-d Tree that holds the information of the split points used during the partitioning step. During the sampling step, the number of maximum records m that each file will contain after the partitioning has taken place, is also set. This value is chosen to allow the creation of similar in length *log* table partitions while their size is close to the HDFS's predefined block size for well-balanced load distribution among mappers.

(ii) Partitioning: The second step is the use of the K-d Tree for partitioning the dataset according to the split points specified in the previous step. This process is executed in a separate MapReduce job that outputs the *log* table partition files that contain data within a hypercube of the partitioning space attributes and files containing the unique join values present inside each partition, in ascending order. The latter information is used to efficiently retrieve the respective meta-data values from the HBase indexed *meta-datasets*.

(iii) Query: The final step is to construct a user-defined function that integrates the above described functionality. The operations of this function are clearly illustrated in Figure 2. The user-defined function takes as input a join attribute (e.g., IP) and the K-d Tree structure. It uses the partition's join range as well as the file that contains the unique join values of the respective partition in order to transfer the relevant meta-data to a mapper's memory. Essentially, it follows a merge join technique between the unique join values and the HBase indexed *meta-dataset*. When the relevant meta-data are retrieved, an equi-join similar to that in Section 3.1 produces the join output (see Algorithm 1).

As mentioned before, the required IP-DNS pairs are transferred from HBase, a fact that renders network throughput a determining factor for the system's performance. In order to make the HBase merge-join operation more efficient, we introduce a new scan utility that enhances the performance when reading a range of IPs from HBase. Our scan utilizes a method called `seekTo()` that uses a heuristic to decide whether the next key-value pair will be accessed sequentially using the `next()` method or it will be accessed immediately by jumping directly to it (see Figure 2). Essentially, it is best to choose the latter when the number of intermediate key-value pairs is above a certain threshold defined by the cost of initializing a jump to the next pair compared to the cost of sequentially accessing all of the intermediate pairs [26].

Algorithm 1 Join execution using K-d Tree

```
1: function EVALUATE
2:     if DnsMap == NULL then
3:         kd = readTreePartitionFile()
4:         kd.findBuckets(min, max, l)
5:         l.sort()
6:         partNum = l.indexOf(partNum)
7:         line = readLineFrom(uniqueIPFile)
8:         s = HBaseTable.getScanner(scan.setStartRow(line))
9:         result = s.next()
10:        while line != NULL && result != NULL do
11:            if UniqueIP > ScanIP then
12:                result=s.seekTo(line, UniqueIP-ScanIP)
13:            else
14:                line = br.seekTo(result.getRowKey())
15:            if result.getRowKey().equals(line) then
16:                keyValue.putInHashMap()
17:    return DnsMap.get(ip)
```

The combination of the aforementioned techniques results in a significant speed up in the execution time of each query. Apart from that, it provides us with a straightforward way of implementing range queries, e.g., across a certain time window, by exploiting the properties of a K-d Tree to find the specific sFlow files that contain records whose target-values lie in the specified range. This way we can limit the number of files that need to be processed and avoid a time consuming scan of the entire dataset.

3.4 Dynamic Mapping Files

The mapping files of IPs to ASes, IP geolocation, and reverse DNS lookup are not static and they do change over time. These changes might not be that often, e.g., IP block assignments to ASes, or they may get updated daily, e.g., geolocation information. Datix supports the import of multiple time-varying mapping files. To accomplish this, we use two different storage approaches based on the size of *meta-datasets*. Small ones are stored in an HDFS directory containing all the different timestamp versions (each file's name is appended with the timestamp version). As mentioned before, large mapping files are stored in HBase which natively supports multiple key-values that have different timestamp versions. In each case, during query time the appropriate key-value pairs are loaded into memory according to the *log* partition's timestamp range.

4. EXPERIMENTS

Cluster Configuration: The experimental setup consists of an ∼okeanos IaaS [21] cluster of 15 VMs. The HDFS, MapReduce, HBase and Spark master is equipped with 4 virtual CPUs, 4GB of RAM and 10GB of disk space. There are also 14 slave nodes hosting all other required processes each of which has 4 virtual CPUs, 8GB of RAM and 60GB of disk space, summing up to a total of approximately 900GB of disk space. Each worker VM runs up to 4 map tasks and 4 reduce tasks, each consuming 768MB of RAM. We utilized Hadoop v1.2.1, HBase v0.94.5, Hive v0.12.0, Spark v0.9.1 and Shark v.0.9.1 respectively.

System-level Comparison: We compare the performance of Datix when running the same queries using either Hive or Shark as the data analysis tools. We evaluate the performance of these two systems based on the query execu-

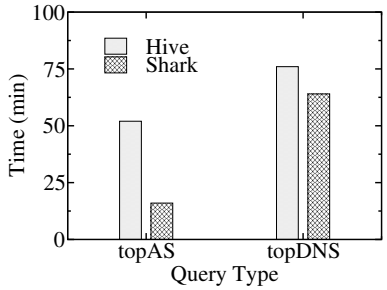

Figure 3: Hive vs Shark.

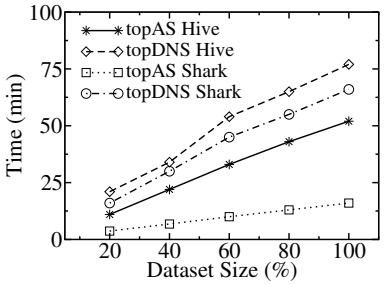

Figure 4: Dataset Scalability.

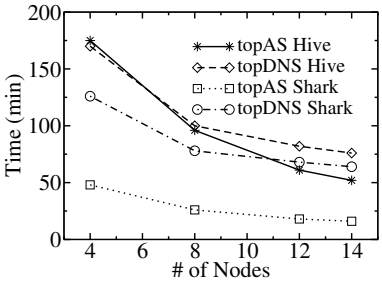

Figure 5: Nodes Scalability.

tion time and we comment on the performance advantages and disadvantages of each system.

Dataset Specification: In our experiments we used a set of sFlow data coming from a national Internet Exchange Point. This dataset spans a period of about six months from 31/07/2013 until 17/02/2014 and is around 1TB in size. In addition, supplementary information of IP to AS and country mappings was retrieved from Geolite [3]. These two files are 12MB and 7MB in size respectively. Finally, IP to reverse DNS lookup mapping had a size of 57GB and was retrieved from the ZMap public research repository that archives Internet-wide scans (scans.io [4]).

Dataset Partitioning: Table 1 depicts the overhead of pre-partitioning the dataset for variable dataset sizes. Both sampling and partitioning have been incorporated into the resulting values. As we can see, data loading time scales linearly in respect to dataset size especially for smaller values. For larger datasets, there is an additional slight overhead, mainly due to the fact that more mapper and reducer processes have to be scheduled to complete the job. In addition, pre-partitioning takes roughly 4 times more compared to a simple scan of the respective dataset.

Dataset Size (%)	20%	40%	60%	80%	100%
Loading (min)	24	50	78	102	150

Table 1: Data Loading vs Dataset Size.

Query Type	Python	Hive	Datix-Hive		Datix-Shark	
			2D	3D	2D	3D
topAS	58min	170min	50min	52min	15min	16min
topDNS	>24h	135min	35min	76min	30min	64min
topDNS 1 week	>24h	116min	30min	9min	28min	8min

Table 2: Base Join Implementation vs Datix.

Table 2 reports the execution times for two query types using a Python based centralized approach, the default join implementation of Hive and Datix. These queries compute the top-k AS or DNS pairs, i.e., we refer to Fully Qualified Domain Names (FQDN), exchanging network data for the entire time period of the dataset we possess. These queries, although simple, cover the *basic* SQL functionalities and can be used to form more complex or specific ones. In particular, these two queries use: join operators, *GROUP BY* operators, *COUNT* operators and *ORDER BY* operators. By combining some or all of the aforementioned operators, more complex queries can be formed to utilize in full scale Datix capabilities. For example, a popular query would be "Calculate the daily traffic of a specific web-server over time" that uses some of the primitives described above (i.e., filtering in time dimension and *GROUP BY/COUNT* operators to calculate daily traffic). In all tables and graphs, 3D partitioning (i.e., source, destination IP, timestamp) results are presented, unless it's explicitly stated otherwise.

For our Python based experiments, we used a host machine featuring a Core i7-4820K CPU with 8 threads, 48GB of RAM and 8TB disk. For the topAS query, where the *meta-dataset* is small enough and fits in a main memory dictionary, the Python based implementation is nearly 4 times slower than Datix on top of Shark while it is almost the same for Datix using Hive. There are several reasons why the latter takes place. First, the machine we used for our Python experiments has a CPU clocked 2 times faster than that of the VMs, thus, in the case where all operations are in-memory, the difference in speed is obvious. Second, the performance of VMs running on top of an IaaS that might over-provision its resources cannot be as good as a physical machine. Furthermore, the mapping of virtual CPUs may not be one-to-one with the physical CPUs of each host. Lastly, the communication overhead between mappers and reducers when using MapReduce-based applications is another thing to consider. On the other hand, the *meta-dataset* for the topDNS query is quite large and thus, it is not feasible to store it in main memory. A possible solution to this problem is to store the *meta-dataset* in a database (e.g., MySQL) and for each sFlow record issue a query to retrieve the required information. Due to high latency when accessing the database, this implementation is inefficient and the query requires more than one day to be executed even in the case of a week's amount of data. In contrast, Datix manages to execute these queries offering scalability and efficiency as discussed in detail in the following paragraphs.

As a second observation, we note that our distributed join algorithms outperform the ones in Hive by nearly 70% both for topAS and topDNS queries because we manage to efficiently exploit in-memory computations and avoid expensive disk I/O operations compared to the simple join algorithm used by Hive. In the case of Shark, the base join implementation fails during execution due to lack of memory, while our implementation is more robust and is executed successfully in short time as shown in Table 2. Another point of interest is the behavior of our system when using a 2D (i.e., source and destination IPs) or 3D (including timestamp) partitioning scheme. In the case of 3D partitioning, a considerable increase in the execution time occurs for the topDNS query. To explain this behavior, we note that when more dimensions are used, the sFlow files produced contain IPs in a wider range and therefore, more IP-reverse DNS lookup pairs have to be transferred from HBase. This fact is verified, since the execution time of the topAS query, where all operations are performed in-memory, is not affected. Despite this noticeable overhead, using 3D partitioning results in much better performance when it comes to filtering queries as shown in the third row of Table 2. To explain this, we note that adding another dimension (i.e.,

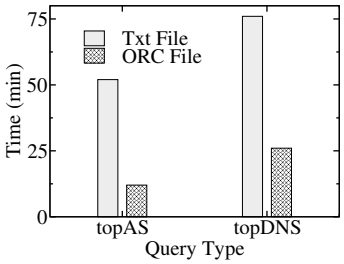

Figure 6: Text vs ORC file format.

timestamp of sFlow records) results in much faster query execution since the processing is limited only to the sFlow files that contain the appropriate records.

In order to provide a direct comparison of the two different big data analysis tools, we then test the performance of Hive versus Shark. Figure 3 shows the execution time for both systems for two different queries. For a fair comparison we used the exact same HiveQL query for Hive and Shark. We observe that for the topDNS query the execution time is similar in both systems. This behavior is expected since this query requires a large amount of data to be transferred from HBase, a process that is limited by the available network throughput and therefore is independent of the characteristics of each tool. However, for the topAS query Shark is significantly faster.

Figures 4 and 5 show the scalability of our system in respect of dataset size and number of available nodes respectively. In Figure 4, we vary the volume of data processed while we keep the number of nodes at 14. In contrast, in Figure 5 we vary the number of nodes while processing the entire dataset. A first observation is that our system scales linearly with the dataset size regardless of the tool used. Shark is faster in all cases when compared to the corresponding implementation in Hive. This is mainly due to the following reasons. First, Shark avoids data spilling to disk for all the intermediate results in a multi-staged MapReduce job, thus, it avoids time consuming I/O operations. Second, it utilizes an efficient task scheduling algorithm and hence, overcomes the expensive launch procedure of mapper or reducer tasks that takes place in traditional MapReduce engines, like Hadoop. Moreover, we observe that for small number of nodes the system's scalability is close to linear while experiencing a slight degradation in performance as this number increases. Lastly, the difference in execution time when using Hive and Shark is quite significant for the topAS query regardless of the cluster or dataset size.

Figure 6 shows the speedup in execution time for both query types when using the ORC file format [19] instead of a plain text format on top of Hive. ORC file format uses a column based approach when storing a Hive table data and is able to retrieve only the required column data. Therefore, in queries that combine information only from a few columns we observe a significant speedup in total execution time.

As an overall comment, Datix yields significant performance gains when the *meta-dataset* is quite large and the available RAM is not sufficient to perform a simple map-join. In particular, our partitioning scheme is designed to overcome such limitations and enables us to efficiently answer various queries. However, when the *meta-dataset* is small enough, a Python-based approach running on a single node with lots of resources (CPUs and RAM) is expected to yield comparable performance to Datix. Nevertheless, our system still overpowers such methods (even by a small percentage) and is by far better than the baseline join of Hive and Shark.

5. RELATED WORK

Distributed Join Algorithms: Distributed join over MapReduce-like systems is challenging and therefore different approaches have been proposed to address this issue. A first attempt to introduce join algorithms for log processing was presented in [11], where the authors compare different algorithms, depending on whether the *meta-dataset* can fit in memory, that can be used to implement equi-joins in MapReduce. However, this work does not tackle the problem of theta-joins which consist of more general join conditions. In fact, our approach modifies the Broadcast Join algorithm presented in this work to effectively deal with theta-joins. In [31], the authors introduce techniques that support other joins and different implementations, but it is also required to extend the MapReduce model. Furthermore, users have to implement non-trivial functions that can handle the dataflow in the distributed system. Hence, this work cannot support high-level languages that run on top of MapReduce (i.e., Hive) compared to our approach. In [25, 8], the authors propose algorithms that perform partitioning during query time to speed up execution time, whereas our approach focuses on pre-partitioning data in order to efficiently use map-phase joins.

Network Analytics: There is a number of systems that have been proposed for network traffic analysis and each one tackles a particular aspect of this broad research area. In [22], an approach was presented to analyze network traffic by processing libpcap files in a distributed environment provided by Hadoop's MapReduce in combination with Hive as a data warehousing tool. The authors implemented an intelligent NetFlow reader in Hadoop that was aware of the particular format of libpcap files which could be spread across different nodes in the cluster. Our work is rather orthogonal as our join algorithms can be integrated in their system and by using the same *meta-datasets* can extract additional information about network traffic. In [23] the authors introduce a machine learning paradigm to classify host roles based on network traffic analysis by collecting sFlow records. Essentially, this work tries to extract information like our approach by analyzing sFlow packets and utilizing MapReduce as the execution framework and NoSQL databases as storage. The difference of our proposed approach is that by enabling the use of arbitrary *meta-datasets* we can extract much richer information about network routing, dimensioning and security features rather than only classifying host roles. A system that can perform both streaming and batch processing of network traffic in order to analyze the constantly increasing volume of network traffic data is presented in [12]. It addresses the scalability problem of existing systems by using distributed methodologies such as MapReduce but it does not support the use of high-level languages over Hadoop framework and thus, cannot be executed over Spark without considerable effort.

The Bro monitoring system [1] offers powerful solutions for collecting and analyzing massive network data but it requires significant effort to run on top of cloud computing infrastructures and to support queries on data that reside on distributed files systems. An increasing number of companies such as DeepField, Guavus and Conviva have heavily invested in the deployment of scalable network analytics

platforms on top of Spark or Hadoop but they are commercial and closed source solutions.

The work presented in [18] deals with hierarchical partitioning, optimizing the query plan to prune processing only to required data partitions. For each data dimension there is a different level in the partitioning tree structure. It involves changing the optimizer module by adding extra features to decide on the chain of joins performed. The main focus of the authors is on traditional RDBMSs while they claim that their approach can be extended to parallel databases. Our approach differs in the fact that we use a flat partitioning scheme dictated by the K-d Tree structure and our system design is tailored to using NoSQL storage and distributed system techniques. Furthermore, our partitioning scheme focuses on splitting the volume of data appropriately to fit in a mapper's memory to perform efficient map-side joins.

In [20], the authors present TidalRace which builds on data streaming applications and show how to optimize partition-based operations. Datix focuses on log processing (batch operations) and overall optimization of query execution in various cases. In contrast, TidalRace supports incremental updates to partitioning information, partition re-organization, and partition-wise optimizations.

DBStream [10] is a Data Stream Warehouse solution for Network Traffic Monitoring and Analysis applications. The queries we execute could also be deployed on this system by extending the functionality of DBStream to support the import of various *meta-datasets* like Datix and then evaluate the resulting performance. DBStream supports real-time data analysis and incremental queries apart from batch processing jobs. However, it is deployed in a centralized manner over a traditional RDBMS (PostgreSQL) while our system is fully decentralized, designed to be able to scale to a large number of nodes and gain extra performance when more resources become available. TicketDB [9], which is the predecessor of DBStream, was compared to vanilla MapReduce jobs performing reduce-side joins, but it does not use a partitioning scheme similar to our K-d Tree approach to enable efficient execution of map-side joins.

6. CONCLUSION

In this paper we introduced a novel network analytics system that depends on distributed processing techniques and is able to effectively execute filtering queries over state-of-the-art distributed processing engines. We introduced a smart pre-partitioning scheme to speed up the execution time of filtering queries (i.e., over a particular time period or set of IP addresses) and we integrated this functionality into an SQL compliant system by using custom-made user-defined functions that are aware of the data format and implement a custom variation of map-join algorithm. Our approach reduced query execution time compared to the basic Hive and Shark implementation by nearly 70%, while efficiently answering queries that took over a day to be processed with existed Python-based code. In this work, we used sFlows as a log dataset from which various information was recovered. Extensions of our work include the assessment of the system's performance when sFlow records are being streamed rather than stored in advance.

7. ACKNOWLEDGMENTS

Nikolaos Papailiou was supported by IKY fellowships of excellence for postgraduate studies in Greece - SIEMENS program. Georgios Smaragdakis was supported by the EU Marie Curie Fellowship CDN-H and EU project BigFoot.

8. REFERENCES

[1] Bro. http://www.bro.org/.
[2] Cisco Visual Networking Index: Forecast and Methodology, 2013 – 2018. Available at http://www.cisco.com.
[3] GeoLite. http://dev.maxmind.com/geoip/legacy/geolite/.
[4] Internet Scans. http://scans.io/.
[5] NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.
[6] sFlow. http://www.sflow.org/.
[7] TreeMap. http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html.
[8] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.
[9] A. Baer, A. Barbuzzi, P. Michiardi, and F. Ricciato. Two Parallel Approaches to Network Data Analysis. In *LADIS*, 2011.
[10] A. Bar, P. Casas, L. Golab, and A. Finamore. DBStream: an Online Aggregation, Filtering and Processing System for Network Traffic Monitoring. In *IWCMC*, 2014.
[11] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *ACM SIGMOD*, 2010.
[12] V. K. Bumgardner and V. W. Marek. Scalable Hybrid Stream and Hadoop Network Analysis System. In *ACM SPEC*, 2014.
[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS 26(2)*, 2008.
[14] N. Chatzis, G. Smaragdakis, A. Feldmann, and W. Willinger. There is More to IXPs than Meets the Eye. *CCR 45(5), 2013.*
[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Comm. of the ACM 51(1)*, 2008.
[16] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium*, 2013.
[17] V. Gopalkrishnan, Q. Li, and K. Karlapalem. Star/Snow-Flake Schema Driven Object-Relational Data Warehouse Design and Query Processing Strategies. In *DaWaK*. 1999.
[18] H. Herodotou, N. Borisov, and S. Babu. Query Optimization Techniques for Partitioned Tables. In *ACM SIGMOD*, 2011.
[19] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major Technical Advancements in Apache Hive. In *ACM SIGMOD*, 2014.
[20] T. Johnson and V. Shkapenyuk. Data Stream Warehousing in Tidalrace. In *CIDR*, 2015.
[21] V. Koukis, C. Venetsanopoulos, and N. Koziris. ∼okeanos: Building a Cloud, Cluster by Cluster. *IEEE Internet Computing 17(3)*, 2013.
[22] Y. Lee and Y. Lee. Toward Scalable Internet Traffic Measurement and Analysis with Hadoop. *CCR 43(1), 2013.*
[23] B. Li, M. H. Gunes, G. Bebis, and J. Springer. A Supervised Machine Learning Approach to Classify Host Roles On Line Using sFlow. In *ACM HPDC*, 2013.
[24] B. J. Louis. Multidimensional Binary Search Trees Used for Associative Searching. *Comm. of the ACM 18(9)*, 1975.
[25] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *ACM SIGMOD*, 2011.
[26] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2RDF+: High-performance Distributed Joins over Large-scale RDF Graphs. In *IEEE BigData*, 2013.
[27] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann. Improving Content Delivery using Provider-aided Distance Information. In *IMC*, 2010.
[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *IEEE MSST*, 2010.
[29] L. Tang and N. Jain. Join Strategies in Hive. *Hive Summit.*
[30] A. Thusoo et al. Hive: A Warehousing Solution over a Map-Reduce Framework. *VLDB*, 2009.
[31] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *ACM SIGMOD*, 2007.
[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX conference on Hot topics in cloud computing*, 2010.