Experience with big data frameworks for IP flow collector

Martin Zadnik, Pavel Krobot, Jan Wrona
CESNET
Zikova 4, Prague, CZ
Email: zadnik, krobot, wrona@cesnet.cz

***Abstract*—Network traffic monitoring requires the collection of information exported from network observation nodes. Given the significant increase in the amount of information collected it is necessary to consider utilization of big data framework for collecting and querying the stored data. This work describes our approach to evaluate frameworks from the perspective of their fitness to the basic collector tasks. We evaluate three big data frameworks and compare them with our MPI implementation.**

I. Introduction

An abstract yet detailed network traffic visibility (e.g. via NetFlow [1], IPFIX [2]) is a key prerequisite to network management including tasks such as traffic engineering, application performance monitoring and network security monitoring. In the recent years the volume of collected information has grown significantly due to the cloud services, mobile traffic, mutli-media content, network threats as well as higher link throughput. While there are probes that are capable of flow monitoring at high link speeds [3], naturally, there is a need for a high-performance flow collector.

The flow collector performs basic tasks such as receiving and storing incoming stream of flow records so as to avoid data loss and answering ad hoc user queries. Additionally, the collector may also assemble periodic reports and perform data analysis (e.g. network behavioral analysis).

The goal is to capture by experiments basic characteristic of several big data frameworks in the view of the two basic collector tasks, storing and querying large sets of flow records. We experiment with Hadoop and its extensions, ElasticSearch and Vertica. We hope that these experiments will simplify decision whether to use these frameworks as an underlying technology of the distributed flow collector or in case of similar applications. Our experiments deliberately omits the additional collector tasks as these are ussualy handled by different type of technology (e.g. stream processing such as DBStream, Spark), in the so called lambda architecture. Moreover, these stream processing engines were already to evaluated for these tasks.

The rest of the paper is organized as follows. Related work on big data frameworks and benchmarks is discussed in Section II. Section III describes the goals of the proposed experiments as well as the experiments themselves. Section IV evaluates the frameworks and gives comparison with our MPI solution. The paper is concluded with summary and future work in Section V.

II. Related work

The collector exhibits several specific characteristics which often renders traditional tools ineffective:

- constantly arriving flow records at high rate (hundreds of thousands up to several million of flow records per second),
- profiling of arriving data leading to data multiplication, hence increased load on the storage system,
- in parallel to the previous, periodic batch processing (ussually 5 minutes) and/or stream analysis,
- in parallel to the previous, ad hoc user queries targeting up to 3 months history [4].

We can observe the effort to improve network collectors since their performance was not sufficient to handle growing amount of network data. The question of which data processing system is suitable for a NetFlow processing appears already in the work by Hofstede et al. [5]. They compared a flow processing speed of NfDump and MySQL database. Giura [6] has investigated possibility to utilize columnar database and proposed several optimizations. Various database engines were also compared in [7]. Nevertheless, these were solutions running on a single machine with none or poor support for distribution. With the advent of the MapReduce computation concept [8], novel distributed data processing systems were introduced with Hadoop [9] at the forefront. It enabled a development of flow data tools [10], [11], [12].

The benchmarking of big data frameworks plays an important role not only in design decisions but also during optimization. HiBench [13] is a benchmark suite dedicated for Hadoop MapReduce and Hive consisting of several Hadoop synthetic as well as real world applications. It allows user to measure various statistics and optimize configuration of particular deployment. Another benchmark for Hadoop MapReduce is GridMix [14] with tasks for text data.

Big data bench [15] is a general benchmark designed to evaluate big data frameworks. It selects several Internet service datasets, however, the closest available dataset to flow data set is e-commerce and it is several orders of magnitude smaller than the dataset we aim for as well as the operations and hence the workload differ. As Chen et al. has shown in [16] only real data and workload can reveal the real system characteristics. Therefore the real world data and workload is preferred in big data benchmark [17]. Since the benchmarks are general and/or focused on a single framework, we decided to conduct our own specific experiments to reveal characteristics of the big data frameworks from the perspective of flow data queries and flow data storage.

III. Experiments

A. Goals

The goal of our experiments is not to evaluate robustness such as consistency, availability or partitioning characteristics of the frameworks. These characteristics can be evaluated by the benchmarks such as [15]. Our experiments rather focus on performance and look for answers to questions such as:

a) How fast can the framework retrieve flow records stored on the disks: The queried data volume (flow records) ussually exceeds the amount of available memory and together with the ad hoc nature of queries prevents disk caching. Thus it is utterly important that the evaluated framework exploits full throughput of all the disks available to the framework.

b) How large is the overhead of the framework: Current single-node collectors are often well-tuned systems. Any overhead renders it less responsive for its potential user. Naturally, users are ready to tolerate proportional reponse time to the amount of data queried. Any overhead of the framework that exhibit itself especially during queries targeting small amount of data renders the framework infeasible as a basis for collector.

c) Is it possible to store flow records as they come without a loss: The collector must process constantly arriving stream of flow records. In large deployment the number of arriving flow records may easily exceed 1 mil. flow records per second especially during DoS attacks and it is very important for post mortem analysis that the framework does not collapse in case of an overload and remains its peek performance.

d) Is it effective in small as well as in large deployments: The collector must scale from just several nodes up to tens of nodes. A universal collector is better then dedicated collectors for small and large deployment. Moreover, it is often the case once the network monitoring is deployed on one link the improved visibility of the network drives further deployments of probes resulting in growing number of flow records anyway.

B. Data set

The data set contains flow records, for the purpose of our experiments, a flow is defined as a set of packets identified by the same 5-tuple – IP addresses, port numbers and protocol. We propose to utilize simplistic flow records consisting of the following items:

- source and destination IP address,
- source and destination IP ports,
- protocol,
- start and end timestamp,
- number of packet and bytes,
- TCP flags.

We utilize such a basic record to simplify obtaining the data sets as these items are, in most cases, always available in the exported flow records. We are aware that the collector must potentially deal with more complex flow records such as VLAN tags, MAC addresses, user identifiers as well as L7 fields of variable length such as url, dns answers, sip identifiers. However, the core functionality of the collector rests in quering the basic flow records as there is only a minority of probes capable of reporting L7 fields, moreover, the L7 fields will be less and less reported with the increasing penetration of encryption.

The data set is sliced into 24 subsets. Each subset contains previous subset plus 1 hour increment, i.e. subsets with size 1/24, 2/24, ... 24/24 of the whole set were created. As a result 1 hour increment is an average number of flow records collected throughout a single day and deliberately does not correspond with the actual time.

C. Queries

In order to evaluate ad hoc user queries we have collected history of the queries at the collector. We selected four queries that in their nature capture typical queries and operations utilized during flow data analysis. The selected queries are of the increasing complexity.

- Number of all flows, packets and bytes in the data set.
- Number of all flows with destination port 53.
- List of all flows records that match protocol TCP and port 22.
- List source IP addresses with the associated number of flows, packets and bytes sorted by the number of flows. (aggregation, sorting)

The first query is a basic one and tests implementation of aggregation operation. The partial sums should be calculated on each node locally and the total sum is an aggregation of the partial sums. The second query tests the filter operation and minimize the data transfer by summing the results locally while the third query selects relatively a lot of flow records and transfers them over the network. The fourth query is a more complex one as it aggregates flow records according to the IP address locally and may partially sort them locally as well, transfers the local results and aggregates and sort them globally.

D. Measurement

Each query is run three times and the time of the longest run is the resulting query time.In order to eliminate the influence of a disk cache it is necessary to flush the cache before each query. Besides the time to answer the query an interesting characteristic is the performance per single node. We define the performance per single node as the total number of queried flow records divided by the number of nodes and the time to answer the query. Last but not least, we measure the time to upload the data set into the cluster. The data set is stored on a local disk of the access node the cluster. This means that the disk of the access node may become bottleneck itself. According to our measurement a single disk can provide more than 3 millions flow records per second. If this bottleneck is reached the framework can easily meet the requirements of the collector which typically should sustain continous upload of 50 thousand flow records per second during normal operation and quater a million during peaks on per monitored 10 Gbps link. We leave the distribution of the data set in the cluster up to the framework itself unless the framework requires manual upload. In such a specific case the flow records in the uploaded subset are distributed in a round robin manner into N smaller subsets where N is the number of available nodes in the cluster and each smaller subset is uploaded on a dedicated node provided the replication factor is 1.

IV. Evaluation

In order to characterize behavior of several frameworks we utilize a real data set. The data set utilized in this evaluation was collected from a large peering link between two national research and education networks [18]. It consists of the flow records exported from the observation point over a single day. In total, the data set contains 880 mil. of flow records collected from 30 bil. packets and 27 terabytes of data corresponding to network traffic captured on a 10 Gbps link over 24 hours. The resulting data set consumes approximately 56 GB in binary format and 86 GB in plain text csv format. We make this data set publicly available [1].

Our experiments were carried out with the above described data set and queries described in Sec. III-C. We were, however, limited by the available hardware available at the time of each experiment and therefore the clusters are not consistent in their configuration. In order to assess the frameworks we describe each hardware setup in detail as well as we discuss the performance per single node which, although biased due to differing hardware, provide a relative measure of effectivity.

A. Hadoop

Our evaluation started with Hadoop framework as the widely popular big data framework. The queries for Hadoop were written as a dedicated implementation in Java (map and reduce objects) and subsequently were executed over binary as well as csv data set. Besides native implementation, Hive and Pig providing SQL-like, respectively, functional interface, were also evaluated. The evaluation captures comparison of performance not only with a widely spread NfSen/NfDump tool but also of various configurations aiming at the optimization of Hadoop cluster and of the multiple queries running in parallel. We utilize NfDump as a base line of effectivity as it was utilized as a base line in previous works (e.g. in [7]).

The underlying hardware is a dedicated cluster for Hadoop. The cluster consists of 27 nodes (24 worker nodes) equipped with 16-core processors (hyperthreading, Intel Xeon CPU E5-2630 v3@2.40GHz) and 128 GB of operational memory and local disks in each node with total capacity of 1.02 PB. The data replication is by default set to a factor of four. Various configurations aim at decreasing the latency that is related to communication overhead in the cluster. Therefore each experiment describes in detail these specific parameters.

We display only the results for the second query since the results of others follow similar characteristic despite resulting

[1]The anonymized data set is available at www.liberouter.org/anonymized

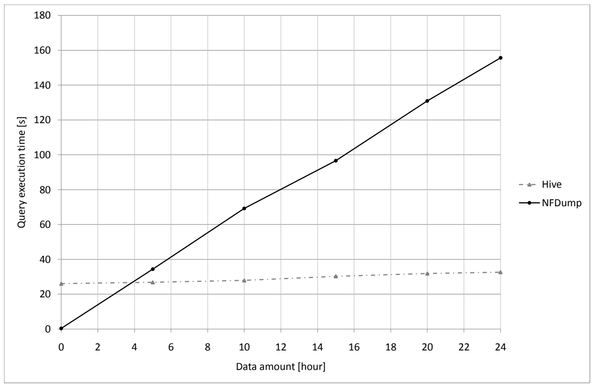

Fig. 1. NfDump performance

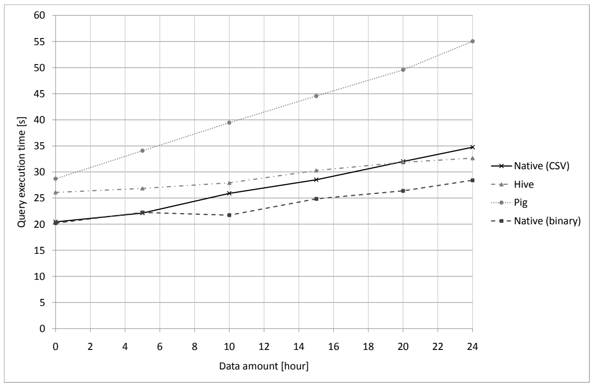

Fig. 2. Hadoop default performance

times vary. Fig. 2 captures the query execution time in dependence on the data set size expressed in hours. It is possible to observe significant speed-up of execution time in comparison with single node NfDump tool (see Fig.1) or that of a smaller cluster tested in our initial experiments with small Hadoop cluster [?]. It can also be seen, that the extensions Pig and Hive perform worse than the native implementations. Another interesting characteristic is that the execution time rises slowly (especially in case of native implementation) with the increasing amount of data. On the other hand despite empty data set is queried the execution time is 20 s. Such an overhead is caused by the nature of Hadoop communication in the cluster. We try to reduce this overhead and our effort is expressed in the following experiments.

An important parameter is Heartbeat interval. This interval determines frequency of message exchange in the cluster. In the above experiment the Heartbeat interval was a default interval of 3 seconds. Since this interval significantly influence the latency to distribute and collect results from the nodes we try to decrease it to a single second. Fig. 3 displays the reply times with 1 second interval. It can be seen that there is no significant speed up and Pig delivers even worse in some cases due to failing nodes.

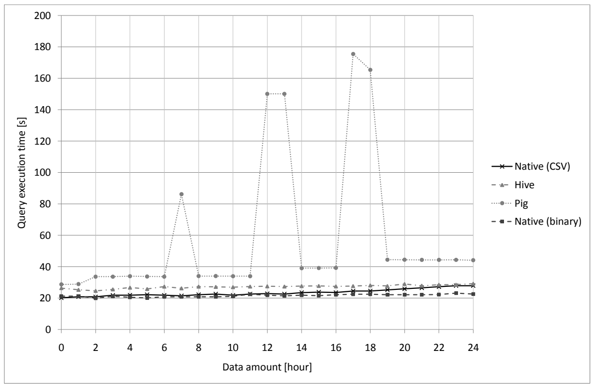

Fig. 3. Hadoop performance with HeartBeat 1s

Hadoop (hadoop-0.23, resp. YARN Hadoop2) also offers an alternative approach of communication utilizing so called uber mode. This mode offers faster delivery of results in case the task can be handled on a single node. The graph in Fig. 4 displays the results for a query over small data sets. In case the query targets a very small data set, not larger than the HDFS block size, than the time to receive result is reduced. On the other hand, the interval is still long enough not to be considered interactive and in case of larger data set the the overhead, naturally, remains. We also noticed increased number of failures of Hive when uber mode was active.

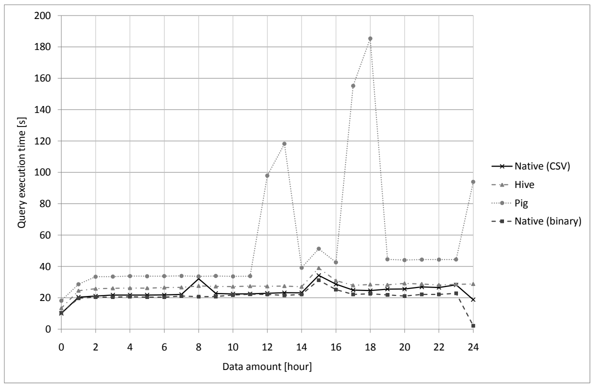

Fig. 4. Hadoop performance with uber mode (first five hours)

Another parameter that can be tuned is the replication factor. Intuitively, we expect that the more replicas in the cluster the better distribution of the workload in the cluster and higher reliability. The graph in Fig. 5 displays results for various setup of replication factor (1-meaning no replication, 2, 4 - default and 6). The graph captures results of Hadoop native implementation since but other implementations follow the same characteristics but perform worse similarly to Fig. 2. The results shows that the replication factor 1 leads to a significant decrease of performance and this also holds for replication factor 2. Replication factor 6 brings only little improvement over replication factor 4 at the cost of 50

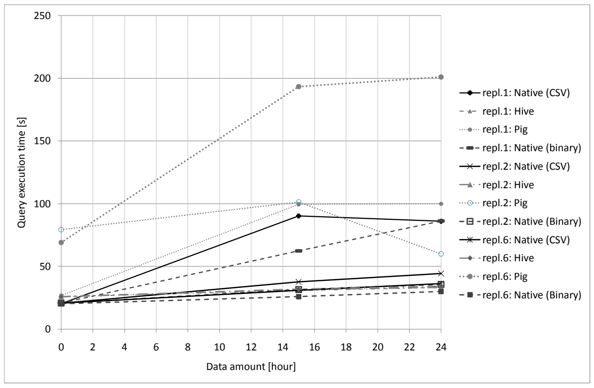

Fig. 5. Hadoop performance with various replication factors

In summary, the default Hadoop setup performs the best. In order to compare solutions based on different frameworks to each other we recalculate the performance per single node as the number of records processed by a single node per second. This characteristic is captured in Fig. 6.

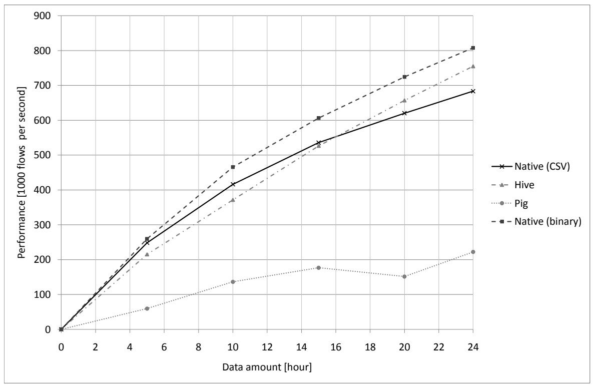

Fig. 6. Hadoop performance of a single node

It can be seen that the Hadoop native implementation achieves the best performance but in comparison with nfdump running on a single node it reaches one third performance per single node. The lower performance is caused by the overhead related to distribution of the task in cluster (especially communication) and programming language.

Further experiments with multiple queries running in parallel show that Hadoop can support multiple queries running in parallel, moreover, the performance per single node improves with the increased load as the ratio between the overhead and the load decreases. The data set was manually duplicated to multiple copies and each query targets a different copy to avoid reading the data from the disk cache. The results are displayed in Fig. **??**.

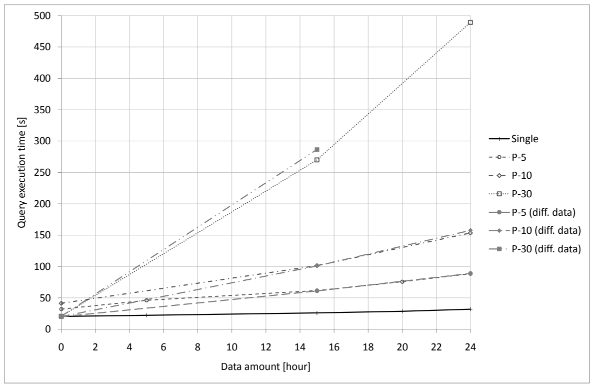

Fig. 7. Hadoop performance with multiple queries running in parallel

The graph shows characteristic for 1, 5, 10 and 30 queries running in parallel. Although the overall time to receive results increases the communication overhead remains almost the same but the amount of data read is increased up to 30 times and hence the performance per single node improves as well. In this case the performance per single is around 2 mil. flows per second which is about 50% of that of dedicated single node nfdump which achieves up to 4 mil. flows/s.

Besides queries, the collector must also be able to store the flow records as they arrive. This is usually a bottleneck in case of classical database systems since there is typically tens of thousands flow records arriving each second (10 Gbps link). To this end we also evaluate the storage performance when loading the flow records into HDFS file system. The original data set is stored locally at the master node and the whole data set is loaded into HDFS or Hive tables at once. We were able to achieve storage speed of up to 1.5 mil. flow records per second in case of a binary data directly stored into the HDFS and more than 800 thousand flow records per second in case of csv or Hive tables.

If we consider that the up to date collectors are single node solutions then the previous experiments were performed over relatively high number of nodes. Therefore we also performed the experiments on a smaller cluster namely 1 master and 5 worker nodes, however the outcomes were inline with the previous summary and we refer interested reader to our results in technical report [19].

Since the native nor Hive nor Pig performance were not satisfactory enough due to long latency and large overhead we tested more recent Apache projects, namely Impala: A Modern, Open-Source SQL Engine for Hadoop [20]. Due to the security issues with authentication we deployed Impala only on 3 nodes of the above described cluster, however, the results shows that even such a small Impala cluster achieves similar results as Hive on the large cluster. As described in [20], Impala achieves these results utilizing only the HDFS but reimplementing the execution engine thus it is not limited by the latency of the native MapReduce and achieves low overhead due to its backend C++ implementation.

TABLE I
COMPARISON OF HIVE VS IMPALA.

Query	Hive [s]	Impala txt [s]	Impala Parq. [s]
1 hour data set			
1	23.29	1.7	1.1
2	23.1	1.1	0.9
3	26.5	1.5	1.1
4	38.7	2.9	2.3
upload	45	44	63
24 hours data set			
1	147.2	18.3	5.8
2	165.6	5.5	2.9
3	177.8	23.2	17.6
4	364.4	39.3	33.7
upload	910	905	1278

The evaluation of Impala variants is captured in Table I and compared with Hive running on the small cluster as well. We evaluated Impala native format and Impala combined with Parquet [**?**], a columnar storage for Hadoop ecosystem. The table displays response times to all four queries and the time to upload data set into the cluster for two corner cases small and large data set. The small data set shows that the response time is not burdened with any large communication overhead while the lower part of the table shows the order of magnitude better response time in case of large data set.

B. Vertica

In order to evaluate a framework from an opposite side of the spectrum, we select Vertica, a commercial distributed columnar database with SQL interface. Vertica is freely available up to 3 nodes, hence we allocate only three nodes (E5-2670@2600 MHz, 4 GB of RAM, local disk).

Fig. 8 displays the graph which compares results of Vertica with Hive as well. The frameworks are compared utilizing all queries but the results are displayed for the second and the fourth query only as these queries reveal significant characteristics. The results show that the times to receive answers increase proportionally to the amount of data without any significant overhead which is clearly visible at the beginning of the graph when the data set is small. The resulting time grows only little steeper in comparison with Hive despite significantly smaller cluster. Such a performance is caused by multiple factors: other form of communication in the cluster, thread-level parallelism at the node level and the columnar database allowing to read only required fields in the records.

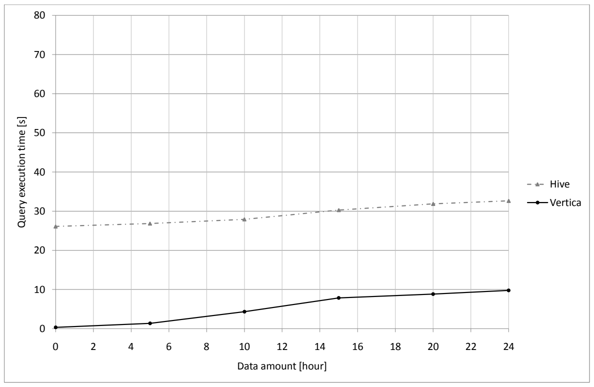

Fig. 8. Vertica performance in case of the second query

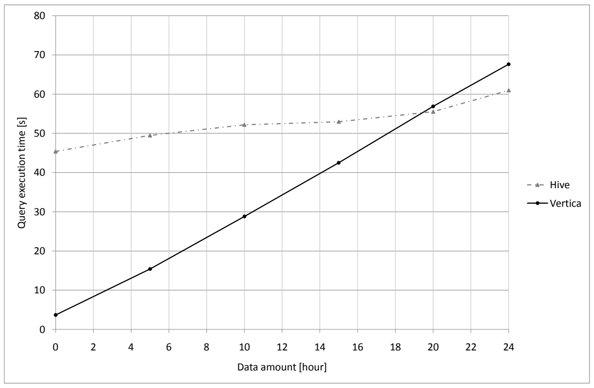

Fig. 9. Vertica performance running the fourth query

Fig. 9 captures characteristic of the fourth query. This query includes aggregation which means additional and intensive access in the memory (in the associative field). It is possible to observe that the times to answer the query meet when the large data sets are used. While the execution of query 2 hits the disk barrier in case of query 4 the limit is the access to the memory. In case of the large cluster the aggregated memory access is much higher than in case of the three machines.

While adding more nodes improves Vertica performance nearly linearly we do not observe any improvement of upload time in comparison to a single node setup.

C. Elastic Search

Elastic Search represents schemaless database for full text search. Since the collector must deal with multiple templates (structures of flow records) which are exported together with the data itself as well as IPFIX allows for variable length elements, it is tempting to utilize such a flexible technology to process and query the flow records. Experiments with Elastic Search were performed on a 9-node (8 workers) cluster with four core Intel Xeon CPU E3-1280 V2@3.60GHz and 32 GB memory each.

TABLE II
COMPARISON OF ELASTIC SEARCH (8 NODES) VS IMPALA (DUPLICATE FIGURES FROM TABLE I).

Query	Elastic Search [s]	Impala Parq. [s]
24 hours data set		
1	41.7	5.8
2	1.4	2.9
3	2.4	17.6
4	474.1	33.7
upload	10 810	1278

Table II shows the results of Elastic Search in comparison with Impala. In some cases the Elastic Search exhibits even shorter response times than Impala. To the best of our knowledge such a short response time is caused by indexing which in our case consumes 4 times the size of the data itself since the index is built over all elements of the 5-tuple. Hovewer in some cases, such as query one when all flow records must be queried to obtain the results the performance degrades since the indexing cannot be utilized at all and it is more expensive to retrieve the data in JSON format which is native for Elastic Search. Query 4 shows that when the query requires large memory to execute the Elastic Search structures are probably larger than those of Imapala and thus cannot utilize cache effectively. Last but not least a standard tool (logstash) to process and upload data into the cluster together with indexing require prohibitive amount of time and renders Elastic Search very slow in comparison with other tested frameworks to upload data into the cluster.

V. CONCLUSION

The paper proposed a benchmark targeting distributed processing of flow records collected from network probes in hundreds of thousands per second. The benchmark focuses on key aspects of the collector, that is, the store and query performance.

The evaluation revealed several key characteristics of the tested big data frameworks. Based on the results, we consider native Hadoop, Hive and Pig as obsolete technology from the perspective of flow collector as was proposed in several previous works, e.g. [11]. A feasible solution to store and query flow records seems to be Impala from the Hadoop ecosystem as well as Vertica which performs comparably if we consider weaker hardware setup.

Currently, we undertake effort into implementing a dedicated storage and query collector based on MPI and we plan to compare the results with general big data frameworks to quantify the overhead related with robustness of big data frameworks in comparison to plain distributed environment. Our long term goal is to build distributed and robust collector combining store&query approach with stream processing for on-the-fly flow analysis.

Acknowledgement

This research was supported by the Technology Agency of the Czech Republic under No. TA04010062 *Technology for processing and analysis of network data in big data concept*. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme ”Projects of Large Infrastructure for Research, Development, and Innovations” (LM2010005), is greatly appreciated.

References

[1] B. Claise, “Cisco systems netflow services export version 9,” Internet Engineering Task Force, RFC(Informational) 3954, 2004. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3954.txt

[2] ——, “Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information,” RFC 5101 (Proposed Standard), Internet Engineering Task Force, January 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5101.txt

[3] L. Kekely, V. Puš, and J. Kořenek, “Software defined monitoring of application protocols,” in *Proceedings of IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. IEEE Computer Society, 2014, pp. 1725–1733. [Online]. Available: http://www.fit.vutbr.cz/research/view_pub.php?id=10657

[4] T. Kosnar, “Notes to flow-based traffic analysis system design,” CESNET, Prague, Czech Republic, 2004, http://archiv.cesnet.cz/doc/techzpravy/2004/ftas-design/.

[5] R. Hofstede, A. Sperotto, T. Fioreze, and A. Pras, “The network data handling war: Mysql vs. nfdump.” in *EUNICE*, ser. Lecture Notes in Computer Science, F. A. Aagesen and S. J. Knapskog, Eds., vol. 6164. Springer, 2010, pp. 167–176.

[6] P. Giura, “Efficient methods to store and query network data,” Ph.D. dissertation, 2010, aAI3457995.

[7] P. Velan, “Practical experience with ipfix flow collectors,” in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, May 2013, pp. 1021–1026.

[8] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

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

[10] J. T. Morken, “Distributed netflow processing using the map-reduce model,” p. 73, 2010.

[11] Y. Lee and Y. Lee, “Toward scalable internet traffic measurement and analysis with hadoop,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 1, pp. 5–13, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2427036.2427038

[12] Y. Lee, W. Kang, and H. Son, “An internet traffic analysis method with mapreduce,” in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, April 2010, pp. 357–361.

[13] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, “The hibench benchmark suite: Characterization of the mapreduce-based data analysis,” in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, March 2010, pp. 41–51.

[14] H. Yang, Z. Luan, W. Li, and D. Qian, “Mapreduce workload modeling with statistical approach,” *J. Grid Comput.*, vol. 10, no. 2, pp. 279–310, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10723-011-9201-4

[15] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, “Bigdatabench: A big data benchmark suite from internet services,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 488–499.

[16] Y. Chen and U. B. Cloudera, “We dont know enough to make a big data benchmark suite - an academia-industry view,” 2012.

[17] W. Xiong, Z. Yu, Z. Bei, J. Zhao, F. Zhang, Y. Zou, X. Bai, Y. Li, and C. Xu, “A characterization of big data benchmarks,” in *Big Data, 2013 IEEE International Conference on*, Oct 2013, pp. 118–125.

[18] CESNET, www.cesnet.cz.

[19] M. Zadnik, P. Krobot, L. Kekely, V. Pus, and J. Korenek, “Distributed collector of records of ip flows: the draft and the first experiment,” CESNET, Prague, Czech Republic, 2015, https://www.cesnet.cz/wp-content/uploads/2015/04/securitycloud.pdf/.

[20] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder, “Impala: A modern, open-source sql engine for hadoop.” in *CIDR*. www.cidrdb.org, 2015. [Online]. Available: http://dblp.uni-trier.de/db/conf/cidr/cidr2015.html#KornackerBBBCCE15