RASP: Real-time Network Analytics with Distributed NoSQL Stream Processing

Georgios Touloupas, Ioannis Konstantinou and Nectarios Koziris
School of Electrical and Computer Engineering, National Technical University of Athens
Email: g.touloupas@gmail.com, {ikons, nkoziris}@cslab.ece.ntua.gr

Abstract—**In this paper we present *RASP*, a system that combines latest distributed stream processing and NoSQL engines to enable the real-time low latency storage and joining of incoming data streams with external datasets of arbitrary sizes through an extensible, SQL compliant manner. We achieve low latency, real time execution by employing the Kafka and Storm frameworks to join incoming tuples as they arrive, while the denormalized result is being stored in HBase, a distributed NoSQL engine with the use of Phoenix, a framework that fully supports SQL. We fine-tune the topology execution to achieve maximum performance and we also apply a set of optimizations both in the HBase storage and the Phoenix SQL execution framework. We use *RASP* to solve a network analytics problem using real data. *RASP* performs its computations utilizing an extensible pipeline of Storm bolts that incrementally augment incoming tuples with the execution of different algorithms. We deploy our system over an IaaS cloud and we evaluate its performance for various workloads, cluster sizes and configurations, where we show that in some cases *RASP* achieves a throughput increase of more than 140% and a latency drop of more than 65% compared to a vanilla setting.**

I. Introduction

Over the past decades, the Internet is continuously growing, driven by ever greater amounts of online information and knowledge, commerce, entertainment and social networking. Recent studies forecast that global Internet traffic will grow with a compound annual growth rate of 26% over the next years, reaching 136.1 Exabytes per month in 2019, up from 42.4 Exabytes per month in 2014 [6]. Large portions of the Internet traffic are routed through Internet Exchange Points (IXPs). An IXP consists of one or more network switches, to which Internet Service Providers (ISPs) connect and exchange Internet traffic between their networks. The IXP allows these networks to interconnect directly, rather than through their upstream transit providers, thereby reducing costs and bandwidth.
Recent studies have shown that large IXPs have visibility to a large fraction of the Internet and fit the role of being global Internet vantage points [16]. Therefore, one can extract information about the global state of the Internet by analyzing the traffic of a large IXP over a sufficient period of time. The typical approach to perform network traffic analysis on a large IXP is by sampling the traffic over a period of time and saving the capture in a file. Then the capture is processed in a centralized manner by a script, where the network traffic analysis is performed. This approach has two main drawbacks. From one hand it does not scale for a larger amount of network data. From the other hand processing offline network traffic captures limits the "freshness" of the data in cases where real-time network information is required [18], e.g., for anomaly detection, DDoS attack detection and SDN reconfiguration.
A variety of distributed technologies and frameworks have been developed to process and store Big Data [1], [21], [2]. Systems such as Kafka, Storm, HDFS, HBase and Phoenix can be used to implement scalable systems that process and analyze data streams in real time. By using such technologies for processing and analyzing the IXP network traffic, the issues mentioned in the previous paragraph can be alleviated. Although previous approaches [20], [17], [22] have employed such tools for network monitoring, they do not offer a real-time environment and focus only on batch/offline workloads.
The objective of work is the design, performance optimization and evaluation of a distributed stream processing system that allows the execution of analytical (i.e., SQL) queries that join a real-time data stream and an external dataset.
The contributions of this paper are the following:

- We combine state-of-the-art distributed stream processing (i.e., Kafka [19] and Storm [21]) and NoSQL Big Data technologies and techniques (i.e., HBase [2] and Phoenix [5]) to minimize the execution latency of SQL queries that join a real-time data stream with external datasets.
- We present *RASP*[1], a system that performs and materializes the join once at data arrival time and allows the fast execution of subsequent queries. Moreover, we provide a way of extending the system by adding external datasets of any size that are joined with the data stream.
- We present a combination of optimizations and fine tuning methodologies that are applied in different stages of the data flow, from data collection to insertion and querying that increase the system's performance.
- We configured *RASP* to join a real-time network data stream, generated by sampling IXP traffic, and external datasets containing Autonomous System and DNS information.
- We deploy our system on top of a cloud platform and we utilize real networking data in order to measure the system's scalability and performance. We show that in some cases our optimizations achieve a throughput increase of more than 140% and a latency drop of more than 65%. Finally, we showcase *RASP*'s scalability.

II. System Description

System Overview: From a high level, the system implemented for our IXP network data use case consists of 4 major parts

[1]available for download at https://github.com/gttm/RASP

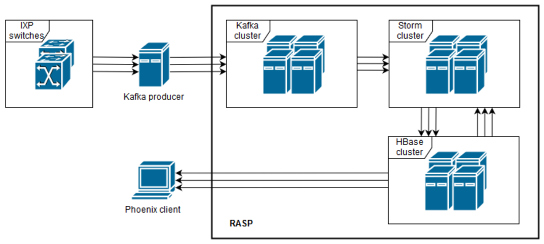

Fig. 1: RASP architecture overview

that can be seen in Figure 1. In the first part, the network data is generated by the switches of an IXP and collected by a host running a Kafka producer. There, the useful fields are extracted from the headers of the captured packets and published to the Kafka topic. The second component of the system is the Kafka topic that temporarily stores the data stream at the Kafka cluster. In the next part, the data stream is processed by a Storm topology. The topology contains the IP to AS Bolt, that performs the join of the data stream and the AS dataset in-memory, since the size of the dataset is small enough. It also contains the IP to DNS Bolt, that performs the join of the data stream and the Reverse DNS dataset using `Get` operations on the HBase table where the dataset is stored, since it does not fit in the bolt's memory. Finally, in the last part the denormalized network data is stored at a Phoenix table in HBase, allowing Phoenix clients to perform low latency SQL queries to it. We employ data Denormalization because it allows fast query execution by avoiding expensive joins during execution while inserting a storage overhead that is easily addressed by the NoSQL database.

The system's **scalability** is achieved by using distributed frameworks and technologies for its implementation. Kafka topics consist of partitions that are distributed over a cluster of Kafka brokers. Storm topologies run over a cluster of Supervisors and multiple instances of any component of the topology (spout or bolt) can run at the same time. The output Phoenix table is stored in HBase, and subsequently in the HDFS, which are both distributed technologies that run on clusters of DataNodes and RegionServers respectively. Moreover, Phoenix can parallelize queries to take full advantage of the HBase cluster. Using the Storm framework provides **extensibility** to our system. Extending the functionality of the Storm topology for a new dataset is as simple as adding an extra bolt to the topology. The data stream that is processed by our system is generated by an *sFlow agent* [13] running on a switch that processes traffic in an IXP.

Kafka Producer: The sFlow datagrams are sent by the sFlow agents of the IXP switches to an sFlow collector running at a specified host. This sFlow collector collects the flow samples from all of the switches and makes them available for further processing. Fields like sourceIP, destinationIP, dateTime, etc., are extracted and sent to Kafka for each sampled packet.

Kafka Topic: The preprocessed messages containing the useful fields in CSV format are stored at the `netdata` Kafka topic in the Kafka cluster. To ensure scalability and load balancing, we set the number of the topic's partitions equal to the number of the brokers of the Kafka cluster. The write and read requests of the producer and the consumers respectively are distributed over the cluster.

The **Storm topology** is the heart of our system. This is where the processing of the data stream is performed. The topology consists of one spout and four bolts in a pipeline setup: Kafka Spout, Split Fields Bolt, IP to AS Bolt, IP to DNS Bolt and Phoenix Bolt. The topology reads messages from a Kafka topic, extracts the useful fields from the messages, performs the join of the data stream and the external datasets and stores the denormalized data stream in Phoenix.

The source of data stream in our topology is the **Kafka Spout**. The spout is a Kafka consumer that reads messages from the `netdata` Kafka topic and emits them to the Split Fields Bolt. The maximum parallelism of the Kafka spout is the number of the topic's partitions, because any instances of the spout further than that would not read any data. The Kafka Spout stores the offset of the consumer for each partition of the topic in Zookeeper. In this way, if a failure happens the topology can be restarted and resume reading messages from the last one that was executed successfully by the topology.

The tuple emitted by the Kafka Spout has a single field: a message from the topic containing the useful fields of the packet in CSV format. The **Split Fields Bolt** extracts these fields from the message. In addition to that the bolt computes the integer representations of the source and destination IP addresses, which are usually more useful than the IP addresses in dot-decimal notation. After processing the Kafka message, the Split Fields Bolt emits a tuple containing the fields sourceIP, sourceIPInt, destinationIP, destinationIPInt, protocol, sourcePort, destinationPort, ipSize, dateTime.

The join of the data stream and the Autonomous System dataset is performed by the **IP to AS Bolt**. The Autonomous System dataset maps IP address ranges to AS number and name. The data has 3 fields: the first IP address contained in the AS, the last IP address contained in the AS and the AS number and name. The defining characteristic of the Autonomous System dataset is that its size (13 MB) is small enough to fit in the memory, which is the optimal way to perform the join of the stream and the dataset. During the initialization of the topology the `prepare` method of the IP to AS Bolt is called and loads the dataset in a TreeMap structure. For each record of the dataset we insert two records in the TreeMap, containing the start and the stop IP address for each AS along with the AS number and name. We use the in-memory TreeMap to locate the IP to AS mapping and we emit this information along with the input tuples to the next bolt of the topology.

The join of the data stream and the Reverse DNS dataset is performed by the **IP to DNS Bolt**. The Reverse DNS dataset maps IP addresses to domain names. The data contained in the dataset have 2 fields: the IP address in dot-decimal notation and the corresponding domain name. The defining characteristic of the Reverse DNS dataset is that its size (55 GB uncompressed) is larger than the memory size, therefore loading it in every bolt's memory is not an option. We store it in the `rdns` HBase table, where the IP addresses are used as the row key and the domain names are stored in the column `d:dns`. This allows the bolt to perform `Get` operations on the table for an IP address row key to receive the corresponding domain name. HBase can perform low latency `Get` operations by using *Bloom filters* [15]. The bolt executes two `Get` operations per to locate sourceDNS & destinationDNS and emits them to the next bolt. The last component of the topology is the **Phoenix Bolt**, which inserts using SQL statements the denormalized data stream into a Phoenix table named `netdata`. The schema of this table is important because it affects the way queries are executed. In our use case, the queries performed will be topN AS or topN DNS queries over a time window for the data. The queries performed on the table have a time window constraint. To benefit from HBase `Scan` operations that perform sequential reads, we use the packet's capture timestamp as the row key (i.e, index) in the underlying HBase table.

The use case queries concern either AS or DNS information. In HBase only the column families needed for the query are cached. Having separate column families containing AS, DNS and other information reduces query latency by reducing the data that have to be cached during each query. Therefore we separate the table's columns in 3 column families: one for the AS fields, another for DNS fields and a default column family that contains the rest of the packet's fields.

III. HBase and Phoenix Optimizations

In this section we present the optimizations we performed to HBase and Phoenix storage/querying frameworks to maximize throughput. Every optimization is evaluated in Subsection IV-B.

HDFS Short-Circuit Local Reads: In HDFS, all reads normally go through the DataNode. When a RegionServer asks the DataNode to read a file, the DataNode reads that file from the disk and sends the data to the RegionServer over a TCP socket. The downside of this approach for local reads is the overhead of the TCP protocol in the kernel, as well as the overhead of DataTransferProtocol used for the communication with the DataNode. When the RegionServer is co-located with the data and *short-circuit local reads* are enabled, local reads bypass the DataNode [10]. This allows the RegionServer to read the data directly from the local disk. Short-circuit local reads provide a substantial performance boost in data transfer from the disk to the BlockCache when the data is local.

Compression and Data Block Encoding: Physical data size on disk can be decreased by using compression and data block encoding [3]. *Compression* reduces the size of large opaque byte arrays in cells and can significantly reduce the storage space needed to store uncompressed data. *Data block encoding* attempts to limit duplication of information in keys, taking advantage of some of the fundamental designs and patterns of HBase, such as sorted row keys and the schema of a given table. Compression and data block encoding can be used together on the same column family. Aside from on-disk data size, compression and data block encoding can reduce the data size in the BlockCache. Data is cached by default on their encoded format. In addition to that, compressed BlockCache can be enabled, allowing compressed data to be cached in their compressed and encoded on-disk format. Between all of our compression options, Snappy [14] is the most fitting to our use case, since minimizing query latency is our priority. It does not aim for maximum compression, but instead aims for very high speeds and reasonable compression. Compared to gzip, Snappy is an order of magnitude faster for most inputs, but the compression ratio is 20% to 100% lower. Regarding data block encoding, Fast Diff is enabled by default in HBase.

Both compression (with compressed BlockCache enabled) and data block encoding reduce the in-cache data size. This means that more rows can be cached at the same time, while data transfer time from the disk to the BlockCache for the same data is reduced. However, every time the cached data is used in a query they must be decompressed or decoded or both. These performance hits increase query latency, while is our priority is to minimize it. To achieve the best in-cache query latency we decide to use Snappy compression for our final Phoenix table, in conjunction with enabled compressed BlockCache and no data block encoding.

Salting: Rows in HBase are sorted lexicographically by row key. The row key for the underlying HBase table where our Phoenix table is stored must be the timestamp associated with the packet, in order to optimize scans for queries over a specified time window. Since the timestamp is always increasing for live data, the row key is also monotonically increasing. However, monotonically increasing row keys are a common source of hotspotting [9]. *Salting* the row key provides a way to mitigate the problem [12] by adding a randomly-assigned prefix to the row key, to cause it to sort differently than it otherwise would.

Since data is placed in multiple buckets during writes, we have to read from all of those buckets when doing scans based on original start and stop keys and merge-sort the data. These scans can be run in parallel on the different RegionServers serving the salt buckets, which may lead to an increase in read performance. Phoenix provides a way to transparently salt the row key with a salting byte for a particular table. Load is evenly distributed among HBase nodes by setting the salt bucket number equal to the cluster size.

IV. Evaluation and Performance Tuning

In this section, we make a performance tuning and evaluation of the subsystems that *RASP* consists of. We describe the utilized datasets and the underlying cloud infrastructure, we evaluate the scalability of Kafka, Storm and HBase/Phoenix and we measure the performance gain achieved in the Storm and HBase/Phoenix case through performance tuning or optimization.

A. Experiment Setup

Datasets: The main streaming data used for the evaluation of the system is network traffic collected by GR-IX [8], the Greek IXP, through which ISPs exchange traffic between their networks without using their upstream transit providers. GR-IX is handling aggregate traffic peaking at multiple Gigabytes per second. Using sFlow, IP packets were captured with a random sampling rate of 1 out of 2000 over a period of six months

(July 2013 to February 2014) resulting in 1.9 billion captured packets (110 packets per second).
One of the external datasets used by the topology is the GeoLite ASN IPv4 database [7] (Autonomous System Dataset). This dataset maps IPv4 address ranges to Autonomous System Numbers (ASN) and is updated by MaxMind every month. The dataset comes in a CSV file, having a size of 13 MB. This file is stored at HDFS in order to be available to the Storm Supervisors. The data contained in it has the fields ipIntStart, ipIntEnd (int representation of the first and last IP address contained in the AS) and the AS number and name.
The other external dataset used by the topology is the Rapid7 Reverse DNS dataset [11]. This dataset maps IPv4 addresses to domain names. Rapid7 Labs creates this data by performing a DNS PTR lookup for all IPv4 addresses. It is updated every 2 weeks and is made available at The Internet-Wide Scan Data Repository (scans.io). The data format is a gzip-compressed CSV file, having a size of 5.7 GB compressed and 55 GB uncompressed, while containing 1.2 billion records. The fields of the dataset are the IP address in dot-decimal notation and the domain name. The Reverse DNS dataset is stored in the `rdns` HBase table. The field `ip` is used as the row key and `dns` is stored at a column.
Cluster Description: We use virtual machines (VMs) operating on a private OpenStack cluster. Each VM has 4 vCPUs @2.4 GHz (no CPU over-provisioning, i.e., 1-1 vCPU to physical CPU mapping), 8 GB of RAM and 80 GB HDD. For the performance tuning experiments we create 10 virtual machines, where one node runs the Zookeeper service, another node runs the HDFS/HBase master process, 4 nodes run the storm cluster and 4 nodes run the HBase and Kafka clusters (Kafka has a low CPU utilization and can be co-located with HBase). To conduct the scalability experiments, we increase the number of nodes in the Storm and Kafka/ HBase clusters up to 16 for each, with the same deployment configuration.

B. Performance Tuning and Scalability Evaluation

Kafka Scalability and fine tuning: The producer can be configured to accumulate data in memory and to send out larger batches in a single request for each partition [4]. *Batching* leads to larger network packets and larger sequential broker disk operations, which allows Kafka to turn a stream of random message writes into linear writes. This increases performance on both the producer and the broker. We experiment with different batch sizes and measure the message input throughput for our topic. The effects of batching on throughput can be observed in Table I. Even though a bigger batch size can increase throughput by orders of magnitude, it also increases the time a message is waits in the producer to be sent in the next batch. Since even a low batch size 100 can achieve greater throughput (12658 messages/sec) than the storm topology in the maximum configuration of our scalability experiment (3988 messages/sec as we present in Table V), we choose a small batch size in order to reduce message latency. The producer handles bursts of more packets with a batch size of **200**.

Batch size	Throughput (messages/sec)
100	12658
200	25516
400	49397
800	104597
1600	188730
3200	293877
6400	381859

TABLE I: Producer batch size effect on topic throughput

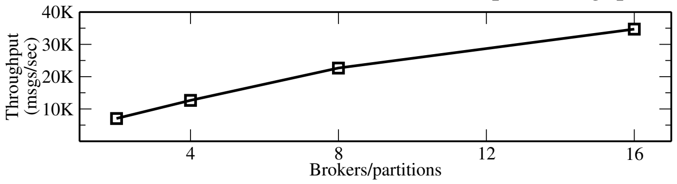

Fig. 2: Topic throughput scalability with Kafka cluster size

Partitions allow the topic to scale in size by being distributed over the brokers of the cluster and act as the unit of parallelism, providing load balancing over the write and read requests of the producers and the consumers respectively.To evaluate the scalability of the Kafka topic with the Kafka cluster size, we measure the message input throughput for clusters with different numbers of brokers. The number of the topic's partitions is adjusted according to the number of the brokers. In Fig. 2 topic throughput scales almost linearly with Kafka cluster size.
Storm Parallelism Tuning: To achieve maximum topology throughput, we experiment with the parallelism of its components (spout and bolts). Parallelism tuning in Storm is performed with the help of the capacity metric. The *capacity metric* tells us what percentage of the time in the last 10 minutes the bolt spent executing tuples. If this value is close to 1, then the bolt is 'at capacity' and is a bottleneck in our topology. The solution to at-capacity bolts is to increase the parallelism of that bolt. Capacity is defined as : $capacity = (executedTuplesNumber * averageExecuteLatency)/measurementTime$
During the parallelism tuning experiments, when we see that a bolt's capacity is close to 1, we increase its parallelism in the next experiment. We continue tuning until we achieve maximum topology throughput. The parallelism and capacity for each bolt during the parallelism tuning experiments are presented on the first eight columns of Table II. The name of each experiment denotes the parallelism of each component of the topology: Kafka Spout - Split Fields Bolt - IP to AS Bolt - IP to DNS Bolt - Phoenix Bolt. Note that capacity is computed based on topology statistics, therefore its value may sometimes appear to be larger than 1. The parallelism of the Kafka Spout is always 4 to match the number of the topic's partitions. As we can see in the last column of Table II we can achieve maximum topology throughput with the parallelism combination **4-4-4-16-28**. We use these parallelism settings for the rest of the benchmarks. We also record the average CPU utilization for the Storm and HBase clusters during the tuning experiments and present them in the % cluster CPU util columns of Table II. We notice that the processors of the Storm and HBase clusters are not saturated at maximum topology throughput, which indicates that the topology workload is I/O

intensive. This was expected since the IP to DNS Bolt and the Phoenix Blot perform reads and writes respectively to HBase.

Bolt Execute Latencies: A useful metric that allows us to identify the bottlenecks in our topology is the execute latency of each bolt. *Execute latency* is the average time a tuple spends in the execute method of a bolt. We record the execute latencies for each bolt of the topology at maximum throughput and present them in Table III. We also compare the % each bolt attributes to the entire latency. It is clear that the tuples spend practically all of their execute time in the IP to DNS Bolt and the Phoenix Bolt. This was expected since these bolts perform reads and writes to HBase tables, while the other bolts execute simple commands in memory.

Bolt	Execute latency (msec)	% time
Split Fields Bolt	0.047	0.3
IP to AS Bolt	0.052	0.4
IP to DNS Bolt	4.779	38
Phoenix Bolt	7.784	61

TABLE III: Average bolt execute latency and total time %

Phoenix Bolt Write Performance with Salting: In this experiment we use a salted and a non-salted table, and compare the throughput of the topology, the write request and CPU utilization on the RegionServers, as well as the execute latency of the Phoenix Bolt. The salted table has 4 salt buckets that are split among the 4 RegionServers of the HBase cluster. The first two columns of Table IV demonstrate that salting serves its purpose by eliminating write hostspotting. S1-S4 denote the different 4 HBase RegionServers that consume incoming workload. Whereas all the write requests were directed to a single RegionServer (S1) for the non-salted table, the load is evenly distributed for the salted table. Note that higher aggregate CPU utilization while using the salted table is linked to better utilization of the cluster's resources, leading to higher topology throughput. Salting also decreases the Phoenix Bolt's execute latency by **74%**, as we can see in the third column of Table IV. The execute latency of the Bolt when writing to the non-salted table was increased due to the strain put on the RegionServer that handled all the write requests. Finally, the last column of Table IV demonstrates that salting increases the topology throughput by **140%**.

Total System Latency: An important performance indicator for our system is *total system latency*, the time it takes for a message to be sent by the Kafka producer to the topic, consumed by the Kafka Spout, processed by the bolts of the topology and eventually be stored in the Phoenix table and made available for queries. We feed the topology with real-time messages and query the table for the row with the latest timestamp. By comparing this timestamp to the current time we can measure the total latency. Total system latency is measured at **1.161 sec** on average at maximum topology throughput.

Storm Scalability: In Table V we evaluate the Storm/HBase topology scalability speedup in terms of throughput (msgs/sec) for different cluster sizes. We increase Storm and HBase cluster sizes simultaneously, meaning that on each test there are as many Supervisors as RegionServers. We also adjust accordingly the number of partitions for the topic, the component parallelism in the topology and the number of salt buckets for the table. After any change to the size HBase cluster we distribute the `rdns` table evenly among the RegionServers and compact it for data locality. Throughput is documented in the second column whereas speedup in the third. Average CPU utilization for the Storm and HBase clusters during the scalability experiments is presented in the fourth and fifth columns respectively.

Cluster Size	Throughput (msgs/sec)	Speedup	Storm CPU util.	HBase CPU util
2	2163	1X	61%	65%
4	2805	1.3X	49%	53%
8	3453	1.6X	37%	37%
16	3988	1.8X	23%	18%

TABLE V: Storm/HBase cluster throughput scalability and % CPU utilization

We notice that the topology throughput scales by adding more servers. Nevertheless it does not scale linearly (speedup column) and the processors are underused for larger cluster sizes (cpu usage columns). This is due to the limitations posed by the underlying cloud infrastructure: although there is a direct mapping of vCPUs to physical CPUs, this is not the case for the physical hard disks. The VM disks of the HBase/Storm clusters are physically co-located, and therefore the aggregate disk I/O throughput is not proportionally increased in contrast to computational power by adding more VMs.

HBase and Phoenix Performance Tuning: The comparison basis of the following benchmarks is our final Phoenix table, after all optimizations are applied. The table uses Snappy compression and no data block encoding, it is split in 3 column families and it is salted in 4 buckets. All the tables are compacted and their regions are distributed evenly among the RegionServers. Queries are performed over 10M rows that are already cached in the BlockCache, unless stated otherwise. We perform two queries, the *count* and *topN AS* query:

```
SELECT COUNT(*) FROM TABLE netdata;
SELECT as.asS, as.asD, COUNT(*) AS pairCount
    FROM netdata
GROUP BY as.asS, as.asD ORDER BY pairCount
    DESC LIMIT 10;
```

The *count* query iterates over the rows of the default column family. This query is useful to measure read performance without any additional calculations. The *topN AS* query returns the top 10 AS pairs in this table ordered by the number of exchanged packets. We also perform the *topN DNS* alternative on some benchmarks, however this query is more computationally intensive, since the `GROUP BY` clause creates many more distinct pairs for domain names than for autonomous systems. Therefore significantly bigger sets that have to be sorted during the calculations and thus subsequently larger query latency.

HDFS Short-Circuit Local Reads: In this experiment we perform a count query over 1 million rows, at first with HDFS short-circuit local reads disabled and afterwards enabled (see Section III). We measure the total query latency, which includes data transfer time to the BlockCache as well as query processing

Experiment	Split Fields Bolt		IP to AS Bolt		IP to DNS Bolt		Phoenix Bolt		% cluster CPU util.		Throughput
	Parallelism	Capacity	Parallelism	Capacity	Parallelism	Capacity	Parallelism	Capacity	Storm	HBase	
4-4-4-4-4	4	0.013	4	0.011	4	0.600	4	**0.987**	22	16	831
4-4-4-12-12	4	0.021	4	0.042	12	0.491	12	**1.068**	41	38	1509
4-4-4-12-20	4	0.040	4	0.043	12	**1.043**	20	**0.911**	40	48	2385
4-4-4-16-28	4	0.043	4	0.062	16	0.879	28	**1.049**	49	53	2782
4-4-4-16-36	4	0.067	4	0.077	16	0.816	36	**1.086**	49	54	2805

TABLE II: Bolt capacity, Storm and HBase cluster CPU utilization and topology throughput during parallelism tuning experiments

HBase throughput (reqs/sec)								HBase % CPU util.								Phoenix bolt latency (msec)		Topol. throughput (msec)	
No Salt				Salt				No Salt				Salt				No Salt	Salt	No Salt	Salt
S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4				
1024	0	0	0	865	868	863	865	48.6	14.5	19.1	14.8	46.1	54	56.7	56.8	30	7.8	1143	2782

TABLE IV: HBase throughput, CPU % utilization, Phoenix Bolt latency ant topology throughput using salting vs vanilla settings

Compression Type	Disk Size (GB)	Count query Latency (sec)	TopN query Latency (sec)
Fast Diff	3.0	5.1	8.5
Snappy	1.6	3.8	7.2
Both	1.0	6.0	10.1

TABLE VI: Query latency and data size vs compression

Cluster Size	Count query Latency (sec)	TopN query Latency (sec)
2	6.55	11.2
4	3.78	7.2
8	3.2	6.3
16	2.7	5.1

TABLE VII: Count/TopN latency, various HBase cluster sizes

time. When HDFS short-circuit local reads are enabled total query time is reduced by **62%** from 4.4 sec to 1.7 sec.

Compression and Data Block Encoding can be used to reduce on-disk data size as well as in-cache data size. However this comes with a performance hit for decompression, decoding or both when reading the cached data as we can see in Table VI. In our experiment we compare on-disk size and in-cache query latency for three compression schemes, namely Fast Diff encoding, Snappy compression with the enabling of Compressed BlockCache and for both Snappy and Fast Diff encoding. As we can see in the second column of Table VI, using both compression and data block encoding reduces the data size further than the other options. Reduced data size allows more rows to be cached at the same time and reduces data transfer time from the disk to the BlockCache. However, the best in-cache query latency is achieved by compression alone (third and fourth columns of Table VI). The data size difference between second and third tables is not big enough to outweigh the query latency advantage of the compressed table, so we chose snappy compression and no data block encoding.

Salting Read Performance also improves read throughput. Phoenix scans the salted data, sorted within each bucket, in parallel and merge-sorts them at the Phoenix client. We perform a count and a topN AS query on a non-salted and a salted table and compare query latency. Salting speeds up queries by **68%**, dropping latencies from 11.7 sec to 3.8 sec and from 23.1 sec to 7.2 sec for the count and topN AS queries.

HBase/Phoenix scalability: We measure the query latency for clusters with different numbers of RegionServers. The number of the table's salt buckets is adjusted according to the number of the RegionServers. Results are presented in Table VII.

V. Conclusion

In this paper we presented the design, implementation, performance optimization and experimental evaluation of RASP, a system that employs distributed stream processing and NoSQL technologies to allow the execution of low latency SQL queries that join a real-time data stream with networking information and an external dataset. Finally, we evaluated the performance of the system using a cluster of VMs. We recorded and analyzed the performance for system component while tuning the system and applying the aforementioned optimizations. The results demonstrated that our system can process packets with a satisfactory throughput, with a low total system latency and allows low latency queries. We experimentally show that in some cases *RASP* achieves a throughput increase of more than 140% and a latency drop of more than 65%.

Acknowledgment

This paper is supported by European Union's Horizon 2020 RIA programme under GA No 690588, project SELIS.

References

[1] Apache Hadoop. http://hadoop.apache.org.
[2] Apache HBase. http://hbase.apache.org.
[3] Apache HBase Reference Guide. http://hbase.apache.org/book.html.
[4] Apache Kafka. http://kafka.apache.org/documentation.html.
[5] Apache Phoenix. http://phoenix.apache.org.
[6] Cisco Visual Networking Index: Forecast and Methodology, 2014-2019 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.
[7] GeoLite Database. http://dev.maxmind.com/geoip/legacy/geolite.
[8] GR-IX. http://www.gr-ix.gr.
[9] HBaseWD: Avoid RegionServer Hotspotting Despite Sequential Keys. http://blog.sematext.com/2012/04/09/hbasewd-avoid-regionserver-hotspotting-despite-writing-records-with-sequential-keys.
[10] HDFS Short-Circuit Local Reads. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.
[11] Rapid7 Reverse DNS. http://scans.io/study/sonar.rdns.
[12] Salted Tables. http://phoenix.apache.org/salted.html.
[13] sFlow. http://www.sflow.org.
[14] Snappy, a fast compressor/decompressor. http://google.github.io/snappy.
[15] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[16] N. Chatzis, G. Smaragdakis, J. Böttger, T. Krenc, and A. Feldmann. On the benefits of using a large IXP as an Internet vantage point. In *IMC*, pages 333–346, 2013.
[17] A. M. Hendawi et al. Hobbits: Hadoop and Hive based Internet traffic analysis. In *Bigdata*. IEEE, 2016.
[18] A. Khurshid et al. Veriflow: Verifying Network-wide Invariants in Real Time. In *NSDI*, pages 15–27, 2013.
[19] J. Kreps et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.
[20] D. Sarlis, N. Papailiou, I. Konstantinou, G. Smaragdakis, and N. Koziris. Datix: A System for Scalable Network Analytics. *ACM SIGCOMM Computer Communication Review*, 45(5):21–28, 2015.
[21] A. Toshniwal et al. Storm@ twitter. In *SIGMOD*, pages 147–156, 2014.
[22] X. Zhou et al. Exploring Netfow data using Hadoop. In *Proceedings of the Second ASE International Conference on Big Data Science and Computing*, 2014.