Network Analysis Query Optimization in Sonata

Marc Leef, Fermi Ma, Cyril Zhang

Abstract

Sonata is a novel traffic monitoring system that aims to solve issues of specificity and performance that are evident in its peers. The two key developments that Sonata uses are iterative query refinement and query partitioning. Query refinement concerns repeatedly modifying the original query to focus on more pertinent traffic, while partitioning involves exporting easily executable parts of the query to the data plane to reduce load on the centralized stream processor. Our work primarily focuses on an exploratory implementation and validation of query refinement, while also contributing to other areas that move the system closer to production-ready status.

1 Introduction

Modern networks contain a veritable treasure trove of data that is crucially useful to a variety of stakeholders. Network administrators can leverage flow timing information to assess network performance, identify bottlenecks, and perform general network-wide debugging. Network security engineers can examine packet headers and assess macroscopic network patterns to identify, combat, and mitigate potential attacks such as DNS reflection, port scanning, and distributed denial of service. Additionally, engineers and business people alike can harness the power of large-scale network analytics to profile internal applications, collect user metrics, and generally strengthen the robustness and adaptability of their networks. Collecting useful analytical information from these networks is an arduous and precarious task, however.

Data pertinent to analytical queries is distributed geographically and resides on a diverse array of networking devices including routers, bridges, switches, and gateways. The amount of data is vast, and both per-flow and per-packet aggregation strategies result in enormous datasets, due in large part to the ever-increasing number of internet connected devices and associated number of packets in-flight. The problem network administrators face is thus three-fold: identifying and locating relevant data, gathering the data to be processed, and finally processing the data in an efficient manner to hopefully yield actionable insight. These problems are exacerbated by the magnitude of the data to be transported and processed and the variable level of granularity at which modern SDN frameworks can collect data. Furthermore, performing network analytics presents a unique challenge in that satisfying user queries will inevitably affect the performance and usability of the network. Data must be transferred and processed using the very same infrastructure primarily targeted by query-based evaluation. These unresolved challenges represent a dire need for a real-time, granular, network monitoring system that minimally affects the host network.

Sonata, a system specifically designed to address these needs, is currently under development by Feamster et al. [4]. Its primary mechanisms are query refinement and partitioning. Query refinement enables iterative improvement of the original query through repeated augmentation to zoom in on relevant and interesting traffic subsets. Query partitioning facilitates the division of work between the data plane and a centralized stream processor by exporting easily executable parts of the query to programmable switches throughout the network. These two features represent novel solutions to a few of the above problems, enabling users to execute real-time network analytics queries with minimal network-wide side effects.

1.1 Mechanisms

Sonata is made up of several smaller services that work in conjunction to facilitate the query execution process. Figure 1 details how an application interface, query run-time, fabric and streaming managers, packet parsers, and a stream processor work together to process a query. The application inter-

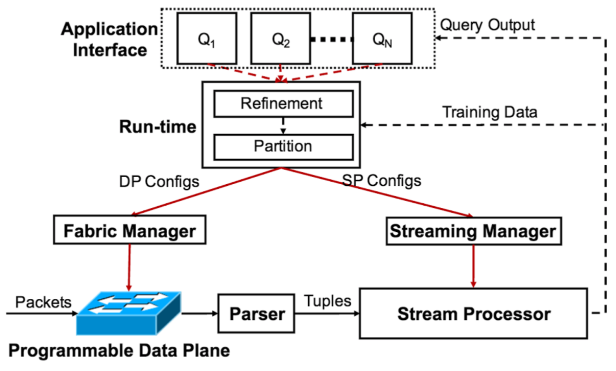

Figure 1: Sonata consists of a number of smaller components that interact with each other to handle a query.

face allows users to write Spark-style queries to execute on their networks thanks to Sonata's "packet as a tuple" abstraction [15]. Sonata queries can be built using many familiar operators including map, reduce, filter, and distinct. Additionally, the query system is not Spark-specific and can be easily extended to handle other query languages. An example query to identify IP addresses that are sending large amounts of data may consist of 1) transforming the original packet stream by mapping the packet tuples to `(source IP, packet size)` pairs, 2) reducing and summing these pairs, and 3) filtering these pairs by comparing the total bytes from each source IP against some threshold.

The query run-time receives user-defined queries and generates both refinement and partitioning plans for the data plane and stream processor. The data plane specific configurations for the query are sent to the fabric manager, which is responsible for updating forwarding tables by interfacing with P4 [2]. Stream processor configuration information is sent to the stream manager which acts as an interface between the run-time and the centralized stream processor. Packets from the data plane are forwarded to parsers, which transform raw packets into tuples before forwarding them to the stream processor. The run-time refines the query per the originally conceived refinement plan at set time intervals and updates the managers accordingly. In this manner, more relevant traffic is identified while extraneous traffic is disregarded, striking a balance between efficient big-data processing and high quality query results.

Before this work, Sonata was in its prototype stages, with large portions of the system unimplemented and untested. The primary achievement of this project has been moving Sonata closer to production-ready status by improving the design and implementation of both query refinement and partitioning. Along the way, we contributed to the creation of Sonata's packet ingestion system and strengthened the theoretical guarantees of Sonata's refinement plan generation algorithm.

2 Design

2.1 Packet Ingestion

Our first contribution to Sonata's improved architecture involved the implementation of a packet-ingestion and tuple transformation module. Before the addition of this component, Sonata simply read in NetFlow [5] records from static text files, an approach starkly at odds with the desired real-time and scalable nature of the system.

In addition, NetFlow records do not provide the specificity necessary to execute fine-grained queries, as information from individual packets is lost when it is packaged into larger flow units. While it was possible to accumulate NetFlow records then transform them into Sonata readable tuples for querying, this gap represented a significant blocker to making the system production ready. We leveraged two pieces of open source technology, Pycapa [8] and Apache Kafka [6], to fill in the missing pieces.

Pycapa is a lightweight Python tool that enables packets to be sniffed from a specified interface on the host machine and optionally published to a Kafka topic. Kafka is a Java/Scala based stream processing tool that leverages a producer/consumer paradigm to facilitate high-throughput, low-latency message passing. Kafka is routinely utilized in industry to simplify logging, diagnostics, and notification systems. We first made some small modifications to Pycapa to enable compatibility with the latest version of Kafka. We then implemented the logic to consume raw packets from a given Kafka topic, extract the pertinent information, transform this data into tuples, and forward on these tuples to the specified destination (usually the stream processor). The scalability of this system stems from Kafka's robustness in handling many to many producer/consumer relationships. Using this design, Pycapa producers can be deployed across many machines and the number of packet consumers can be load-balanced accordingly.

2.2 Query Refinement

We briefly summarize the initial design for how Sonata performs iterative query refinement. Sonata attempts to select a refinement plan that minimizes a linear combination of the following three cost metrics: (1) Detection delay D: The time it takes for traffic satisfying a query to be detected once it has started. This scales roughly linearly with the number of refinement levels. (2) Count-bucket cost B: The number of count buckets needed to detect traffic satisfying the query, across all nodes in the network. This serves as a cost that penalizes the refinement scheme for not being sufficiently granular, which decreases roughly exponentially in the number of levels. (3) Packet processing time N: The number of packet tuples the stream processor must see to detect traffic satisfying the query. This increases superlinearly with the number of levels.

The refinement plan is viewed as a tree in order to simplify reasoning about cost. The root represents a query satisfied by all possible packets and maintains a single count bucket. Each layer of the tree represents some refinement of the query, and the lowest level of the tree corresponds to the network operator's original query. More specifically, each node in the tree corresponds to a single count bucket at that node's aggregation level. For example, for a level of the tree corresponding to `dIP/16`, there can be at most 65536 nodes. Additionally, nodes in the tree will only exist if the count bucket corresponding to their parent node exceeds some threshold within some measurement window. Thus, a useful feature of this tree abstraction is that the number of nodes in the tree will represent the count bucket component of the overall cost.

A crucial observation from [4] is that each of these cost functions is convex in the number of refinement levels, in the following sense: given a fixed number of refinements k, the cost of the cheapest refinement schedule with exactly k levels is convex. As such, any positive linear combination of these cost factors is also convex. The precise weighting of this linear combination is left up to the operator, and amounts to a time-memory tradeoff between processing all levels and having more efficient count buckets. When such an aggregate objective is defined, the convexity observation implies that there is a well-behaved "optimum" refinement tree depth.

2.3 Generalizable Aggregation Plans

The essence of Sonata's refinement tree algorithm is a series of transformations applied to the original tuple stream (these can also be thought of as augmentations of the original query). The purpose of these transformations, or aggregation plans, is to increase the specificity of the query, providing better results to the user while minimizing the processing of superfluous data. Our first attempt at formalizing these abstract units of data manipulation involved defining the most common and useful types of transformations, focusing on specific tuple fields and the most natural way to pare down the packet stream using the values of these fields. We settled on three primary types of aggregation plans: prefix, containment, and range plans.

Prefix plans were an obvious focus, as the original Sonata paper places a heavy emphasis on leveraging the hierarchical nature of IP to reduce data processing costs. Such plans operate on either the destination or source IP tuple fields, and are used in conjunction with a map command to reduce or increase the size the IP address space considered by the query. The canonical example of this aggregation plan is to map all tuple source IPs to the their 16-bit prefixes (`IP/16`), greatly reducing the number of distinct IPs that must be stored in memory. Containment plans leverage information gleaned from earlier iterations of the query to identify higher relevance data. This information can be a list of values for any tuple attribute, such as source MAC addresses or protocols. Used with a filter command, an example containment plan may remove all tuples not sent to a suspicious-looking group of destination MAC addresses detected earlier in the query process. Lastly, range plans, like containment plans, filter tuples based on field values, differing only in that these values are compared against integer ranges instead of accumulating lists. Example use cases might include filtering out tuples with small data sizes or timestamps that fall outside of a targeted window.

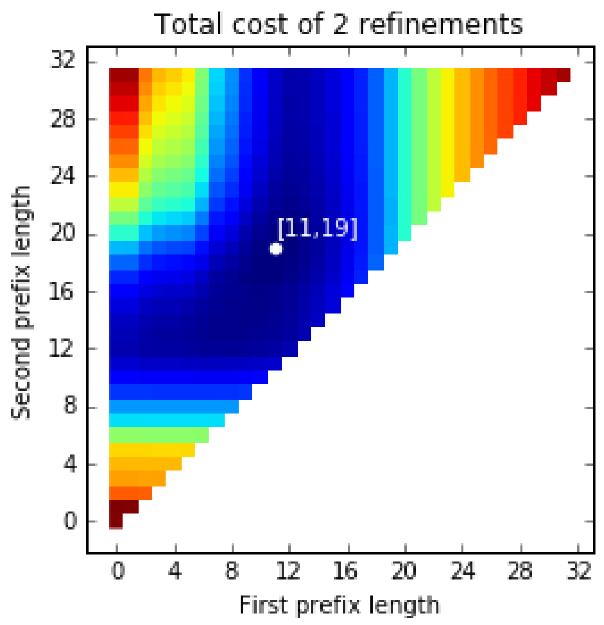

Figure 2: Plot of total costs of a 2-level refinement plan on the IP header, with count bucket cost at weight 1 and packet processing cost at weight 0.002. A value of x on the horizontal axis corresponds to a level of refinement at `dIP/`x, and similarly a value of y on the vertical axis corresponds to the other level of refinement being at `dIP/`y. Blue represents lower costs and red represents higher costs. The optimum refinement plan is denoted with a white dot.

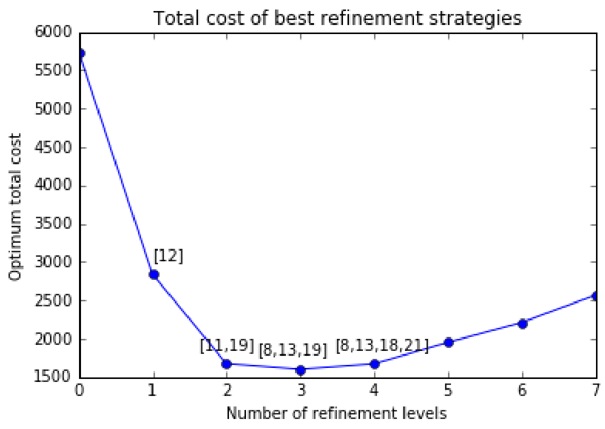

Figure 3: Plot of total costs of a k-level refinement plan on the IP header, as a function of k. We assign a weight of 1 to count bucket cost, 0.002 to packet processing cost, and 20 to detection delay cost. Here, the optimal number of levels is 3, and the optimal plan is to refine at `dIP/8`, then `dIP/13`, and finally at `dIP/19`.

3 Evaluation

3.1 Query refinement: IPs

In order to construct an efficient configuration of query refinements, we devise a simple model of each sub-objective described in Section 2.2, based on statistics easily captured from IPFIX logs. In these logs, we are given a recording of all packet sources and destinations (MAC addresses) and routing hops (IPs) over a period of time; for realistic results, we use a recording of a DNS reflection attack. Our optimization amounts to inferring the relationship between IP prefixes and the hierarchical structure of the concentric neighborhoods around a victim. Using this, we can predict cost terms B and N.

First, we review how costs are computed in this model. Suppose our first refinement level corresponds to `dIP/16`. This involves grouping together each distinct 16-bit prefix that appears and preserving all prefixes that appear frequently enough (i.e. exceed the fixed threshold stated in the query). The number of count buckets at this level is simply the number of distinct 16-bit prefixes that remain. If we perform a further level of refinement to `dIP/24` starting from `dIP/16`, we group all the distinct 24-bit prefixes that remain from the first level of refinement, and then preserve only the ones that exceed the the threshold. The total count bucket cost for this two level plan is just the total number of count buckets at both levels of refinement. The total packet processing cost is computed by summing the total number of packets not filtered out at each level. Recall that the detection delay cost D simply scales linearly with the number of levels, so it will simply be constant when considering a fixed number of refinements.

Our first exploratory test was to fix the number of refinement levels to be 2, and then use brute force to compute the count bucket cost and packet processing cost of first refining at `dIP/x` and then at `dIP/y` for $x, y \in \{1, 2, \dots, 32\}, x < y$. The resulting total cost is a weighted sum of these costs (as well as the constant detection delay cost), and we found a weighting of 1 for count bucket and 0.01 for packet processing to be reasonable (packet processing should be much cheaper). The resulting total costs are given in Figure 2.

This exploratory analysis revealed that if we fix the number of refinement levels, the total cost function appears to be very close to convex, favoring a balanced spread of prefix lengths. This motivates the application of gradient descent to find the optimal (or very near-optimal) refinement plan among k level plans for fixed k. We implemented gradient descent to find the optimum plan for each k, and varied k. The resulting plot of this optimal total cost versus the number of levels k is given in Figure 3. For reasonable weightings of the three component costs, we found that the total cost tends to have a clear minimum. In summary, this method allows us to quickly pick the optimal number of levels of refinement, and produce an optimal refinement plan.

3.2 Sampling Rate Optimization

A way to drastically reduce costs is to sample packets at some rate r before each level of refinement. Of course, this comes at the risk of possibly drawing too few samples of the interesting data and losing it entirely. For example, suppose we set a count threshold at 100, and that the relevant traffic has a count of 120. If we sample at a rate of 0.4 and accordingly adjust the count threshold to 40, there is a $F(40;120,0.4) = 8\%$ (where $F(k;n,p)$ is the binomial cumulative distribution function (cdf) defined as $F(k;n,p) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$) probability that we do not sample enough to hit the threshold. Thus, in our model we implement risk-aware sampling, in which the operator can specify a desired confidence guarantee (for example, 95%) and using the binomial cdf to automatically choose a lower threshold to ensure this confidence guarantee. This amounts to the probability that no false negatives will arise at any refinement level from undersampling desired traffic, which may be of use in more delicate traffic monitoring settings.

With this framework in place, we can optimize the sampling rates for each refinement level without worrying about losing interesting data, except with a small fixed probability. We performed an experiment on the optimal refinement plans selected via the methodology of Section 3.1, minimizing total cost with respect to sampling rates for each level. The result of this experiment is given in Figure 4. We focus exclusively on the optimal 2-level refinement plan of first refining at `IP/11` and then at `IP/19`, and we toggled the sampling rates at these two levels. The optimal sampling rates of 0.255 for the first level and 1 for the second level are unsurprising. Many distinct destination IPs remain upon refining on the

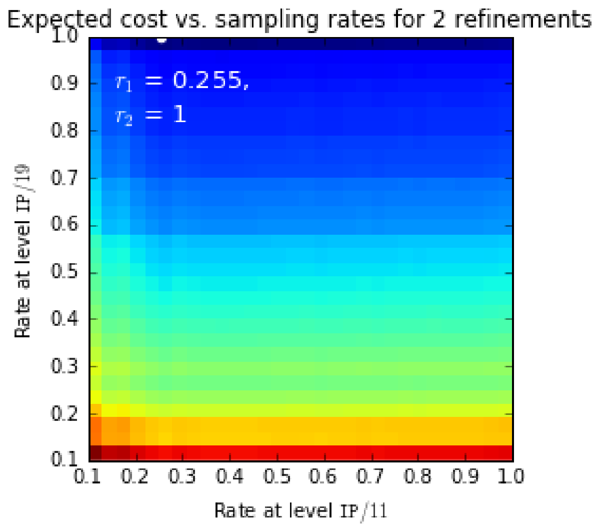

Figure 4: Plot of total cost of an optimal 2-level refinement plan on the IP header, as sample rates for the first (x axis) and second level (y axis) of refinement are varied. Blue represents lower costs and red represents higher costs. The optimal sample rates are denoted by the white dot.

first 11 bits of the IP prefix, so sampling greatly reduces costs at this level, while far fewer IPs remain after the second level of refinement. This empirically shows that the benefits of further sampling disappear towards the end of the refinement. In larger-scale settings, one might benefit from using our optimization procedure to choose sampling rates for the first one or two refinement levels.

4 Related Work

4.1 Stream Processor and Programmable Data Plane

The original Sonata paper discusses other projects that represent the current state of the art in both streaming and data plane based network analytics. Many of these related works fall on two opposite ends of a design spectrum.

On one end of the spectrum, there are several platforms that execute their network monitoring queries with the stream processor. One such example is Chimera [3], an SQL variant designed specifically for network data processing. A number of related platforms have emerged in recent years that use advanced streaming analytics machinery to quickly process network data. However, since all these platforms execute their monitoring queries in the user space, they all suffer from scalability issues at high data rates. Sonata manages to mitigate this scaling problem through interaction between the stream processor and the programmable data plane. This interaction allows Sonata to refine the data stream and decrease the load on the stream processor.

The other end of the spectrum consists of architectures such as OpenSketch [13], which contains a library of useful functions for switches, such as count-min sketch and reversible sketch. The problem with this type of design is that the queries enabled by the data plane are not well-suited for a number of common applications, such as whenever many data streams must be joined. The takeaway is that by taking advantage of both the stream processor and the programmable data plane, Sonata is able to handle a wide range of applications at high data rates.

4.2 Iterative Refinement

The iterative query refinement process is not unique to Sonata. Notably, ProgME [14] is a programmable traffic measurement architecture capable of adapting to changing traffic conditions. It uses iterative refinement on queries solely within the data plane in order to detect heavy hitter traffic. ProgME faces scalability issues due to the fact that its design requires it to process certain packets several times.

4.3 Data Locality Optimization

Iridium [9] is a system developed to increase the efficiency of analytics performed on large-scale, geographically distributed data. Both the unpredictable capacity and utilization of data center WAN links and the volume of data that needs to be centralized for useful processing make this a difficult proposition. Iridium's proposed solution involves intelligently placing subsets of the desired data across different sites per current network allowances before scheduling tasks to process the data. These scheduling decisions are determined by a few online heuristics, as the problem of optimally placing tasks is computationally intractable. This enables large volumes of data to be moved throughout the network while minimizing the creation of bottlenecks. Similarly, Geode [12] leverages Apache Hive [11] and graph-based heuristics to proactively geo-locate data and schedule processing tasks accordingly. While the real-time nature of Sonata's targeted data makes it more difficult to organize data geographically, concepts from these systems can certainly be applied to network monitoring to improve query response-time and minimize network-wide effects.

4.4 Analytics for Specific Applications

While Sonata aims to provide a broadly applicable system, a few other interesting projects aim to tackle specific, subtle problems in network analytics. One such system [7] attempts to leverage large-scale machine learning to make online topology changes in response to changing network conditions. Another project, AntMonitor [10], is a tool that aims to facilitate productive mobile network analytics while maintaining and strengthening user privacy. These tools provide narrow solutions to context-specific problems that may be better suited for a more general-purpose system, such as Sonata. Other projects combine versatility and problem specific features to provide a powerful and compelling toolkit. One such system is DBStream [1], a system that leverages background processing and intelligent task placement to continually provide an up-to-date view of the network. The system also integrates with a few other monitoring systems, making it both adaptable and immediately useful.

5 Conclusions

Sonata is a network traffic monitoring system that answers operator queries using automatic refinement and partitioning to return relevant traffic at the appropriate granularity. Our project focused on bringing Sonata closer to a production-ready state through implementing intended features and improving the query refinement mechanisms. On the implementation side, we built a packet-ingestion and tuple transformation module, as well as an extensible set of building blocks for tuple transformations and query refinement. On the query refinement side, we formulated a near-optimal method for handling IP prefix plans and computing optimal sampling rates. We tested our optimization methods on real IPFIX data and found promising results that support the notion that optimal query refinement can drastically reduce network costs. While there remains plenty of room for further exploration and implementation, we believe this work is a notable refinement of the Sonata architecture.

References

[1] A. Baer, P. Casas, A. D?Alconzo, P. Fiadino, L. Golab, M. Mellia, and E. Schikuta. Dbstream: A holistic approach to large-scale network traffic monitoring and analysis. *Computer Networks*, 2016.

[2] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 328–341. ACM, 2016.

[3] K. Borders, J. Springer, and M. Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 365–379, 2012.

[4] A. Gupta and N. Feamster. Sonata: Scalable Streaming Analytics for Network Monitoring. Princeton University.

[5] R. Hofstede, P. Čeleda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras. Flow monitoring explained: from packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014.

[6] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[7] F. Morales, M. Ruiz, and L. Velasco. Virtual network topology reconfiguration based on big data analytics for traffic prediction. In *Optical Fiber Communication Conference*, pages Th3I–5. Optical Society of America, 2016.

[8] T. Piscitell. Python Packet Capture (pycapa). `https://github.com/OpenSOC/pycapa`, 2015.

[9] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[10] A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, and A. Markopoulou. Antmonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices. In *Proceedings of the 2015 Workshop on Wireless of the Students, by the Students, & for the Students*, pages 25–27. ACM, 2015.

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.

[12] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 323–336, 2015.

[13] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42, 2013.

[14] L. Yuan, C.-N. Chuah, and P. Mohapatra. Progme: towards programmable network measurement. *IEEE/ACM Transactions on Networking (TON)*, 19 (1):115–128, 2011.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.