PIQ: Persistent Interactive Queries for Network Security Analytics

Oliver Michel
University of Colorado Boulder
oliver.michel@colorado.edu

John Sonchack
University of Pennsylvania
jsonch@seas.upenn.edu

Eric Keller
University of Colorado Boulder
eric.keller@colorado.edu

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

Abstract

Network monitoring is an increasingly important task in the operation of today's large and complex computer networks. In recent years, technologies leveraging software defined networking and programmable hardware have been proposed. These innovations enable operators to get fine-grained insight into every single packet traversing their network at high rates. They generate packet or flow records of all or a subset of traffic in the network and send them to an analytics system that runs specific applications to detect performance or security issues at line rate in a *live* manner.

Unexplored, however, remains the area of detailed, interactive, and retrospective analysis of network records for debugging or auditing purposes. This is likely due to technical challenges in storing and querying large amounts of network monitoring data efficiently. In this work, we study these challenges in more detail. In particular, we explore recent advances in time series databases and find that these systems not only scale to millions of records per second but also allow for expressive queries significantly simplifying practical network debugging and data analysis in the context of computer network monitoring.

CCS Concepts

• **Networks** → **Network monitoring**; *Network security*;

ACM Reference Format:
Oliver Michel, John Sonchack, Eric Keller, and Jonathan M. Smith. 2019. PIQ: Persistent Interactive Queries for Network Security Analytics. In *ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFVSec '19), March 27, 2019, Richardson, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3309194.3309197

1 Introduction

Continuous network monitoring is essential to the operation of data center, enterprise, and wide area networks. Network monitoring enables operators to detect security issues, misconfigurations, equipment failure and perform traffic engineering [7, 8, 12, 13, 21, 24]. Today's networks are larger and more complex than ever before; they carry more traffic, run more advanced applications and are continuing to grow. Therefore, careful and detailed network monitoring is increasingly imperative.

With the introduction of modern programmable switches [6, 9, 20], there is new opportunity to meet these challenges. In fact, network monitoring has become one of the 'killer apps' for modern programmable switches. These network monitoring systems can generally be organized in two parts: telemetry systems and analytics systems. Telemetry systems leverage the programmability of switches to collect and report real time measurements in the data path at rates and fidelity not possible before [1, 15, 22, 23]. Analytics systems focus on the practical analysis of this data for performance monitoring, intrusion detection, and failure detection [11, 14]. They use advances in parallel software-based data processing, such as stream processing as well as kernel bypass technologies for data input [2, 10, 16, 19].

Together, telemetry and analytics systems provide fine-grained visibility into live network conditions that is useful for many applications. But equally important and unaddressed by current systems is visibility into *past* network conditions. There are a variety of reasons why such information matters. For some applications, such as network auditing, historic information is simply required. For others, such as debugging, it is essential to not only identify that the network is in a certain state, but also *how* it got into that state. In many cases, historic data is necessary because analysis is too expensive to do in real time, and many times need human interaction to investigate. For example, in security systems, a common scenario is to identify anomalies in real time and tag related network monitoring records (packet or flow records) for offline analysis using a heavier weight analysis system or assistance from a network administrator.

All of the above applications rely on *retrospective* queries about the network, which requires some level of record persistence. This poses a significant challenge given the volume and velocity of record-based monitoring data in networks. The challenge is not only due to the shear amount of records that modern telemetry systems can generate, but also due to

the high rates (hundreds of millions of packets per second) at which today's wide-area and data center networks operate.

In this paper, we analyze this challenge in more detail and take first steps towards a telemetry system that supports not only live queries, but also retrospective queries. We explore the requirements of such a system, identify time-series databases as a promising starting point, and design strategies for using modern database engines with state of the art network telemetry and analytics systems. Finally, we sketch the design of a next generation network monitoring architecture, centered around programmable hardware that is composed of telemetry, analytics, and persistence planes, that supports high performance expressive and retrospective queries.

2 Background

Databases have been successfully deployed and used for decades in a wide range of applications and are the backbone of systems across all industries. Traditionally, databases have been used for online transaction processing workloads, like in the financial, production and transportation industries. Advances over the past 10-15 years in data transmission rates and storage capacity have driven the demand for a revolution in database and data analytics technologies. These workloads that are significantly higher in both velocity and volume are commonly referred to as *Big Data* workloads. In this section, we look at this spectrum of database technology to understand suitability for network monitoring.

2.1 Database Models

Although initially database systems were designed around the rigid mathematical relational model, the emergence of *Big Data* has led to new database designs straying from this original model. These databases are often referred to as *NoSQL* databases as their relational counterparts use the Structured Query Language (SQL) as their interface.

Relational database management systems (RDBMS) require data to be in a fixed format that often needs to be broken up in several relations in order to fit in this model. Especially for modern workloads, this process comes with significant performance drawbacks due to frequent joins. On the other hand, relational databases are extremely powerful: They allow for complex data models, can enforce rigid integrity constraints, follow a strong transactional model and have a very expressive query interface through SQL.

NoSQL databases generally do not require this fixed storage format and are, as a result, easier to adapt to custom, irregular and unstructured data. Additionally, they are optimized for large volumes of data and often have better I/O performance and horizontal scalability properties than their relational counterparts. This is mostly due to the lack of features that SQL-based systems implement and enforce in the database layer, such as integrity constraints.

2.2 Time-Series Databases

Alongside the emergence of *Big Data* applications, the widespread deployment of IoT and general sensor data applications has triggered a shift from traditional transactional database applications to applications where data has a strong temporal character. A common example for such data is measurement-related data where an observation is associated with a time. A key characteristic of such workloads is that this data is typically only written once, never updated and from that point on exclusively read (queried). As a result, database systems optimized for this type of workload and equipped with time-series related functions were proposed. These systems are referred to as time-series databases.

As network monitoring data normally consists of measurements of some sort that are associated with their time of observation, time-series databases are a natural fit for our problem domain. We experimented with several systems and soon realized that an expressive query interface, as well as join operations across data stored at different levels of granularity (*e.g.*,packet records vs. flow records) are essential requirements for designing a practical and flexible persistence scheme for network monitoring data.

Unfortunately, the vast majority of time-series optimized databases are implemented as some sort of non-relational key-value store. While this is suitable for multiple independent series of measurements that are never put in context with each other, this is not suitable for network monitoring data (see section 5). TimescaleDB [25] provides a promising alternative: A relational database management system that is optimized for time-series data. Timescale is an extension for the PostgreSQL RDBMS, a database system that has been used for decades and is an industry standard because of its many features, reliability and scalability [17]. TimescaleDB provides high write throughput, good scalability and most importantly does not compromise on any traditional database features by providing a full-featured SQL query interface and allowing for constraints and joins.

2.3 Database Requirements for Network Record Persistence

TimescaleDB appears to be a good match for the application of network monitoring. In the remainder of this paper we seek to evaluate the suitability of TimescaleDB for this application. In particular, we answer three key questions:

Is the query interface expressiveness enough? A network administrator must be able to quickly query the database system using a flexible, fast, and intuitive query system that meets the needs of interactive analysis of historical network telemetry data. In Section 3 we provide examples of such queries.

At what rates can we insert data? Inserting data into the database is a key challenge that limits the applicability of

the database. Network monitoring records are commonly generated at rates of several million per second. As a result, a database system must be optimized for high write rates, and we need to understand the degree of aggregation which is required to meet the insert limits. In Section 4, we evaluate optimizations for write performance into TimescaleDB.

At what scale can we store data? At these high insert rates, massive amounts of data can accumulate in short periods of time. The database system must be scalable enough to cope with data volumes of billions of records. This, coupled with the aggregation levels, then determine the window of time which network operators can interactively query. In section 5, we analyze the storage requirements for network records at different granularities.

3 Querying Network Records

Before even discussing performance, we look at the expressiveness of TimescaleDB in the context of network monitoring. We wish to run retrospective queries and allow for exploratory data analysis on network records.

3.1 Network Queries

To demonstrate the flexibility of SQL for network monitoring records, we show a set of example queries highlighting different language features of SQL and TimescaleDB:

1. *Bin packet and byte counts in 1s time intervals:* Timescale's `time_bucket()` function is useful to make large amounts of time-series data manageable. For example, this query can be used to generate a traffic graph over time.

```
SELECT time_bucket('1 seconds', ts_us) AS interval,
  SUM(pkt_count) AS pkt_count, SUM(byte_count) AS byte_count
  FROM gpv GROUP BY interval ORDER BY interval ASC
```

2. *Count the number of packets per IP address from a particular IP subnet within the last hour:* PostgreSQL's `INET` datatypes allows specifying queries that are not limited to a direct match on an IP address but can efficiently query at the granularity of IP prefixes.

```
SELECT ip_src, SUM(pkt_count) AS pkt_count FROM gpv WHERE ip_src
  << inet '60.70.0.0/16' AND gpv.ts_us > NOW() - interval '1 hours'
  GROUP BY ip_src
```

3. *List packets that came from a particular IP subnet:* A `JOIN` allows to query data across relations and in this case can result in per-packet records including individual timestamps, packet sizes, or TCP information. If the underlying packet records were already deleted, this query would still succeed but only show aggregate information such as total byte and packet counts.

```
SELECT * FROM (gpv RIGHT JOIN pkt ON gpv.gpv_id = pkt.gpv_id)
  WHERE gpv.ip_src << inet '60.70.0.0/16'
```

3.2 Retrospective Queries and Debugging

In section 1 we explained how network analytics suites are optimized to process large amounts of data quickly. This is important for the timely detection of intrusions or other issues within the network. In the case of an anomaly, an analytics system can generate alerts but does not have the ability to inspect the problem.

For example, a network record stream processor could detect an unusually high queue occupancy and queuing delay in a single queue of a switch; a problem often caused by incorrect load balancing schemes or an adversarial traffic pattern. We now show how retrospective network queries can in this scenario help a network administrator to determine if indeed a misconfiguration is causing this effect or whether the network is *simply* at capacity.

As a first step it is of value to determine what packets actually were in the queue while the alert was raised:

```
SELECT DISTINCT gpv.ip_src, gpv.ip_dst, gpv.ip_proto, gpv.tp_src,
  gpv.tp_dst FROM (gpv RIGHT JOIN pkt ON gpv.gpv_id = pkt.gpv_id)
  WHERE pkt.ingress_ts >= '2018-02-19 12:59:11.595' AND
  pkt.ingress_ts < '2018-02-19 12:59:11.600' AND pkt.queue_id = '5'
```

This query returns a list of flows whose packets were in the respective queue at this specified time. The administrator sees that the majority of packets were destined for particular IP subnet, a cluster that serves video content. The routing configuration for this subnet shows, that it is reachable via an ECMP group. After determining the links that are part of this group, the following query can be used to get insight into the distribution of packets across these links:

```
SELECT pkt.queue_id, COUNT(pkt.queue_id) FROM (gpv RIGHT JOIN pkt
  ON gpv.gpv_id = pkt.gpv_id) WHERE gpv.ip_dst << inet '53.231/16'
  AND pkt.queue_id IN (4,5,6,7) GROUP BY pkt.queue_id;
```

If the returned distribution is roughly uniform, the load balancing scheme works, otherwise there is an issue with this particular ECMP group and its hashing algorithm.

While the stream processor could include a digest of the packets which were in the particular queue at the time, the problem may be located well beyond this single queue and finding the root cause can require a more in-depth analysis of the state of the network. Record persistence and an interactive query system allow an operator to analyze the problem in more detail across different network devices with the goal of identifying the underlying issue.

Furthermore, as all of these queries, took less than one second to complete on a 200M packet record dataset, this method is perfectly suitable for interactive debugging and exploratory data analysis. While query performance might degrade with larger databases, there are ways to overcome this issue through precompiling queries [3] or using specialized indices or views for frequently performed queries.

4 Inserting Network Records

A key challenge for the design and implementation of network record based monitoring systems lies in dealing with high traffic rates of today's networks. Database systems, as previously explained, are usually not optimized for these write-heavy workloads and represent a performance bottleneck. Therefore, the database write performance determines how many network records can be saved in a given amount of time and at which level of aggregation they can be stored for network analysis. We evaluate TimescaleDB in this regard and show optimizations that vastly improve insert performance.

Figure 1: GPV INSERT vs. COPY performance

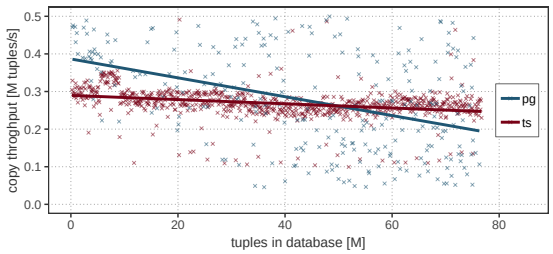


Figure 2: GPV COPY performance as a function of database size for PostgreSQL and TimescaleDB

The injection process (commonly implemented through SQL `INSERT` statements) is associated with complex underlying logic and tasks, such as updating indices, checking constraints, partitioning data, and running triggers. These tasks can significantly hurt injection performance. TimescaleDB helps in this case as it is optimized for mainly appending data as opposed to performing random insert and update operations. TimescaleDB organizes data in chunks indexed by timestamp that fit into memory. Each chunk has its own index. This means that the individual indices are small and efficiently manageable. Chunks get written to disk asynchronously only after a chunk has been filled entirely.

The insert performance can further be optimized by not using complex data constraints and injecting data in chunks using the SQL `COPY` statement as opposed to using `INSERT`. PostgreSQL (Timescale's underlying database) has a custom binary format and allows for fast inserts of chunks of data in this format. Details on this feature and the format can be found in [18]. We compared the performance of `COPY` and `INSERT` for different chunk sizes. Figure 1 shows the mean throughput of injecting 10 million rows. At a batch size of 100K packets using `COPY`, we achieve an average write throughput of 360K records per second. This is over an order of magnitude higher compared to using `INSERT`.

We also compared the write-performance of TimescaleDB and standard PostgreSQL with respect to the number of tuples already inserted. Figure 2 shows the results of this experiment. We can see that PostgreSQL's write performance is initially higher but degrades with database size. TimescaleDB's performance only slightly decreases with database size.

Field	Length [Byte]	Description
`ts_us`	8	absolute timestamp in μs
`gpv_id`	8	unique identifier
`flow_key`	26	IP 5-tuple
- `ip_src`	8	IP source address
- `ip_dst`	8	IP destination address
- `tp_src`	4	source port
- `tp_dst`	4	destination port
- `ip_proto`	2	IP Protocol
`sample_id`	2	sample identifier
`tap_id`	2	tap identifier (*e.g.*, switch)
`duration`	4	GPV duration in μs
`pkt_count`	2	number of packets in GPV
`byte_count`	3	number of Bytes in GPV

Table 1: Grouped Packet Vector Format

Field	Length [Byte]	Description
`gpv_id`	8	unique identifier (foreign key)
`ts_us`	8	absolute timestamp in μs
`queue_id`	2	unique queue ID
`tcp_flags`	2	TCP Flags
`egress_delta`	4	ingress - egress timestamp
`byte_count`	2	total packet length
`queue_depth`	2	experienced queue length
`ip_id`	2	IP identification field
`tcp_seq`	4	TCP sequence number

Table 2: Packet Record Format

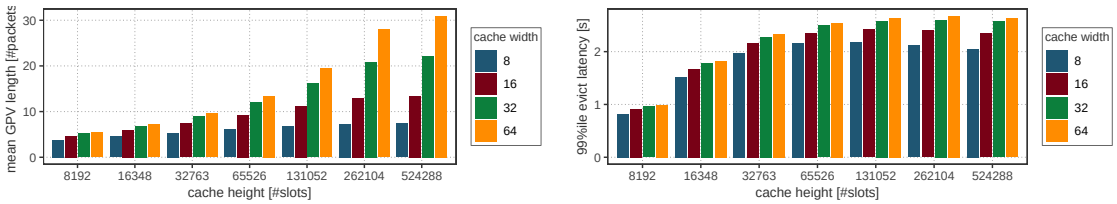
While a write throughput of 300K - 400K records is still over an order of magnitude lower than packet rates in high-speed networks, we explain in section 6 that this rate can actually be sufficient for most applications since not necessarily every packet needs to be stored at the highest level of granularity. Using flow-based aggregation schemes, compression rates of 30-40×, while maintaining a good level of detail per flow, are possible. We further elaborate on this in the next section. Additionally, we believe that these write rates can be further improved by optimizing PostgreSQL storage and transaction settings, as well as inserting records using multiple threads simultaneously.

5 Storing Network Records

As our goal is to enable interactive retrospective queries, the storage of the data base is critical in determining the time window under which we can query. In order to maintain the ability to analyze stored network records using expressive and powerful queries, a carefully designed storage format is imperative. Furthermore, given the volume of records that can be generated, compression and data retention strategies must be addressed. In this section we detail the data format and analyze storage tradeoffs an operator can take for a particular use case.

5.1 Grouped Packet Vectors

We use the grouped packet vector format (GPV), proposed in [23], as the record format for our prototype implementation. A grouped packet vector contains an IP 5-tuple flow key and a variable length vector of feature tuples from sequential



(a) mean GPV length **(b) 99%ile eviction latency**

Figure 3: GPV generation properties depending on cache configuration

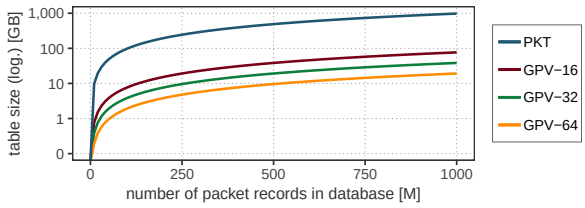


Figure 4: Timescale physical database size for different relations as a function of packets stored

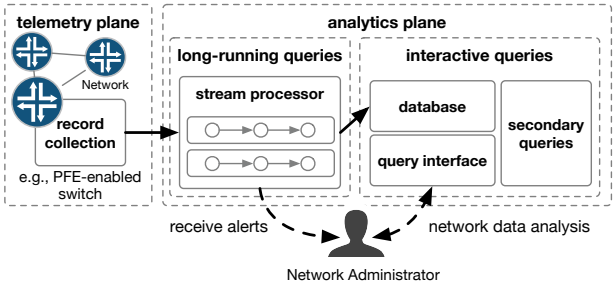


Figure 5: Overall System Architecture

packets in the respective flow. A GPV effectively is a hybrid between a packet record and a flow record. It inherits some of the best attributes of both formats and also compresses packet records by deduplicating the IP 5-tuple.

In order to query GPV data in a relational database system in a flexible manner, the GPV format must be normalized through two relations, one for the GPV header (flow data) and one for the packet features. The columns for the two relations, including their storage size requirements in TimescaleDB, are listed in Table 1 and in Table 2.

As opposed to proper flow record generation using TCP flags and timeouts, we generate GPVs using a simple hash table-based cache data structure as we use this format mainly for reasons of data compression. The cache is organized in a number of slots (cache height) with a fixed amount of packet features that fit in each slot (cache width). The slot index is determined through a hash function from the IP 5-tuple. Individual packets are then appended to the packet feature vector. A GPV is evicted from the cache when either a hash collision happens or when the feature vector is full. This data structure can be implemented in hardware and further optimized using secondary caches for high-activity flows.

For this work, however, we use a simple single cache implementation in software. The cache performance (in terms of eviction ratio) is directly dependent on the cache dimensions (number of slots and slot width). Figure 3 depicts the effects on the eviction behavior for different cache dimensions through experimentation using real-world WAN packet traces [5]. The larger the cache is in either dimension, the higher the achieved compression ratio (GPV length) is (see figure 3a). On the other hand, a large cache requires more memory and also extends the eviction latency, *i.e.,* the time a record spends in the cache before being evicted (see figure 3b). This can be important when running applications on *live* data. As a result, the cache dimensions must be carefully chosen for the application's requirements.

For the remainder of this paper, we chose a cache height of $2^{18} = 262144$ with a width of 32. With these parameters, the size of the cache in memory is 206 MB. In our simulations, the mean GPV length is 20.82 with a mean eviction latency of 0.85 seconds and a 99%ile latency of 2.60 seconds.

5.2 Storage and Record Retention

Storing data in a database generally requires more storage space than a custom, optimized binary format. In this case, a GPV header in C++ only occupies 48 Bytes, whereas its representation in TimescaleDB occupies 56 Bytes. A packet record requires 34 Bytes in Timescale, but only 24 Bytes in our custom storage format. The difference mainly stems from use of different data types, as well as added foreign key columns. Additionally, TimescaleDB maintains indices and other metadata for tables. We measured the physical disk storage required to store 100 million records of each type. Together with all metadata and indices, a GPV header occupies 122.7 Bytes and a packet record 97.8 Bytes. In either case. this is a more than 2× increase over the binary format. The storage requirements grow linearly with record count.

Looking at these numbers at scale, we can see that using our format and database layout, 1 billion packet records require approximately 1TB of disk storage. In addition, each packet record belongs to a GPV header which occupies storage. The storage requirements of the GPV header in respect to the total number of records depends on the average GPV

length. At an average GPV length of 16, a billion packet records require approximately 76GB of GPV records. At an average length of 64, the requirement goes down to roughly 19GB. The total database size is the sum of the sizes of the `pkt` and `gpv` relations. Figure 4 shows these results in detail.

In our experiments, we inserted up to around 1 billion records into TimescaleDB. While we did not go beyond this, Timescale has been successfully used with database sizes beyond 500 billion records [4]. Our dataset of a 10Gbit/s wide-area link had an average packet rate of 330K/s [5]. Given a storage budget of 500 billion rows, at this packet rate, TimescaleDB could store around 17 days of packet-level data.

In a practical deployment we imagine that an operator would not necessarily store every single packet record forever. For example, a data retention policy could be defined in which every GPV header is stored and packet records are only retained within a storage budget. Various different policies are imaginable. We leave this discussion open for future research. Our relational model is designed such that most queries would still succeed when a GPV header does not reference any packet records anymore. The query would then return already aggregated instead of per-packet data.

6 End-to-End Design of a Retrospective Monitoring System

As the examples and preliminary results in this paper have demonstrated, retrospective network queries are a powerful abstraction. It is also practical for next generation monitoring systems to support. Figure 5 illustrates the architecture of such a system, which would leverage the components described in this paper along with other recent work. At a high level, the system can be conceptualized as three planes:
(1) A telemetry plane that collects network records at high rates in the network and sends them to a software platform.
(2) A real time analytics plane for long-running network queries, that leverages scale out stream processing engine to scan network records and detect problems. Suspicious packets are marked for later analysis.
(3) The component explored in this paper: a persistence plane and query subsystem in which network records can be saved over longer times at different granularities and retrospectively queried to further investigate issues in the network either by a human or by secondary analysis systems.

We envision a general persistence plane with adapters to consume data from both the telemetry and analytics planes. In the simplest case, input could be truncated packet headers cloned from network switches. Most commodity switches support these features, which are sufficient to enable many of the examples described in this paper. The persistence plane could also consume input from more advanced telemetry systems that leverage programmable forwarding engines, e.g., P4 hardware. This would provide two benefits: additional data about switch state, e.g., queue depths; and reduced network overhead, *e.g.,*by preprocessing packet headers into GPVs in the data plane [23]. More advanced still, the persistence plane could consume input from the real-time analytics plane. There are at least two use cases for such integration. First, alerts from the real-time analytics system could trigger more advanced retrospective queries. For example, an alert indicating high queue depths or a dropped packet could automatically invoke the diagnostics queries described in the prior section. Additionally, the real-time analytics plane can serve as a preprocessor, normalizing record formats and prefiltering out data that does not need to be stored.

7 Conclusion

Network monitoring is increasingly important in the operation of today's large and complex network. With the introduction of modern programmable switches, there are more opportunities than ever before to collect high fidelity measurements. As recent real time telemetry and analytics systems have demonstrated, this can provide visibility into network conditions that enable powerful new monitoring applicaitons. But often, visibility into current network conditions is not enough. For debuggers, security systems, and many other applications, it is critical to also have visibility into *past* network conditions.

In this paper, we identify this need for *retrospective* network analytics and show how such a system can be realized. We leverage recent trends in database technology, namely time-series databases. While most time series databases are implemented as *NoSQL*, key-value stores with custom query interfaces, we motivate why for this workload, a traditional relational database model is better-suited. We study the feasibility of using a relational model based time-series database (TimescaleDB) for network monitoring records.

We identify the main challenges of this approach and design strategies and optimizations to tailor an existing database engine for retrospective network analytics. These strategies improve system efficiency significantly and the resulting prototype serves as both a demonstration of feasibility and an important first step towards a complete solution. With this prototype, we explore features of retrospective queries and motivating use cases. Finally, we sketch the end-to-end architecture of a next generation monitoring system that leverages programmable switches, advanced telemetry systems, and our contributions to support high performance and expressive retrospective queries.

Acknowledgements
This research was supported in part by the National Science Foundation under grants 1700527 (SDI-CSCS) and 1652698 (CAREER).

References

[1] Barefoot Deep Insight. https://www.barefootnetworks.com/products/brief-deep-insight/.

[2] Data Plane Development Kit. https://dpdk.org.

[3] DB Toaster. https://dbtoaster.github.io.

[4] Talk: Rearchitecting a SQL Database for Time-Series Data. https://www.youtube.com/watch?v=eQKbbCg0NqE.

[5] Trace statistics for caida passive oc48 and oc192 traces – 2015-02-19. https://www.caida.org/data/passive/trace$_s$tats/.

[6] P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (jul 2014), 87–95.

[7] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)* (2010), vol. 7, pp. 19–19.

[8] Benson, T., Anand, A., Akella, A., and Zhang, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), ACM, p. 8.

[9] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), SIGCOMM '13, ACM, pp. 99–110.

[10] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 15–28.

[11] Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., and Willinger, W. Sonata: Query-driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 357–371.

[12] Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), vol. 14, pp. 71–85.

[13] Li, Y., Miao, R., Kim, C., and Yu, M. FlowRadar: A Better NetFlow for Data Centers. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation* (Santa Clara, CA, 2016), USENIX Association, pp. 311–324.

[14] Michel, O., Keller, E., Sonchack, J., and Smith, J. M. Packet-level analytics in software without compromises. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing* (Boston, MA, 2018), USENIX Association.

[15] Narayana, S., Sivaraman, A., and Nathan, V. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of SIGCOMM '17* (Los Angeles, CA, 2017), ACM, pp. 85–98.

[16] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. Netbricks: Taking the v out of nfv. In *OSDI* (2016), pp. 203–216.

[17] Postgres. Postgresql database system. https://www.postgresql.org.

[18] Postgres. Postgresql documentation: Copy. https://www.postgresql.org/docs/current/sql-copy.html.

[19] Rizzo, L. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX Association, pp. 101–112.

[20] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 15–28.

[21] So-In, C. A survey of network traffic monitoring and analysis tools. *Cse 576m computer system analysis project, Washington University in St. Louis* (2009).

[22] Sonchack, J., Aviv, A. J., Keller, E., and Smith, J. M. Turboflow: Information Rich Flow Record Generation on Commodity Switches. In *Proceedings of the Thirteenth EuroSys Conference* (New York, NY, USA, 2018), EuroSys '18, ACM, pp. 11:1—-11:16.

[23] Sonchack, J., Michel, O., Aviv, A. J., Keller, E., and Smith, J. M. Scaling hardware accelerated monitoring to concurrent and dynamic queries with *flow. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)* (Boston, MA, 2018), USENIX Association.

[24] Sperotto, A., Schaffrath, G., Sadre, R., Morariu, C., Pras, A., and Stiller, B. An overview of ip flow-based intrusion detection. *IEEE communications surveys & tutorials 12*, 3 (2010), 343–356.

[25] Timescale. Timescale db. https://www.timescale.com/.