Sonata: Query-Driven Streaming Network Telemetry

Arpit Gupta
Princeton University

Rob Harrison
Princeton University

Marco Canini
KAUST

Nick Feamster
Princeton University

Jennifer Rexford
Princeton University

Walter Willinger
NIKSUN Inc.

ABSTRACT

Managing and securing networks requires collecting and analyzing network traffic data in real time. Existing telemetry systems do not allow operators to express the range of queries needed to perform management or scale to large traffic volumes and rates. We present Sonata, an expressive and scalable telemetry system that coordinates joint collection and analysis of network traffic. Sonata provides a declarative interface to express queries for a wide range of common telemetry tasks; to enable real-time execution, Sonata partitions each query across the stream processor and the data plane, running as much of the query as it can on the network switch, at line rate. To optimize the use of limited switch memory, Sonata dynamically refines each query to ensure that available resources focus only on traffic that satisfies the query. Our evaluation shows that Sonata can support a wide range of telemetry tasks while reducing the workload for the stream processor by as much as seven orders of magnitude compared to existing telemetry systems.

1 INTRODUCTION

Network operators routinely perform continuous monitoring to track events ranging from performance impairments to attacks. This monitoring requires continuous, real-time measurement and analysis—a process commonly referred to as *network telemetry* [54]. Existing telemetry systems can collect and analyze measurement data in real time, but they either support a limited set of telemetry tasks [34, 40], or they incur substantial processing and storage costs as traffic rates and queries increase [7, 10, 57].

Existing telemetry systems typically trade off scalability for expressiveness, or vice versa. Telemetry systems that rely on stream processors alone are expressive but not scalable. For example, systems such as NetQRE [57] and OpenSOC [40] can support a wide range of queries using stream processors running on general-purpose CPUs, but they incur substantial bandwidth and processing costs to do so. Large networks can require performing as many as 100 million operations per second for rates of 1 Tbps and packet sizes of 1 KB. Scaling to these rates using modern stream processors is prohibitively costly due to the lower (2–3 orders of magnitude) processing capacity per core [37, 39, 41, 58]. On the other hand, telemetry systems that rely on programmable switches alone can scale to high traffic rates, but they give up expressiveness to achieve this scalability. For example, Marple [34] and OpenSketch [55], can perform telemetry tasks by executing queries solely in the data plane at line rate, but the queries that they can support are limited by the capabilities and memory in the data plane.

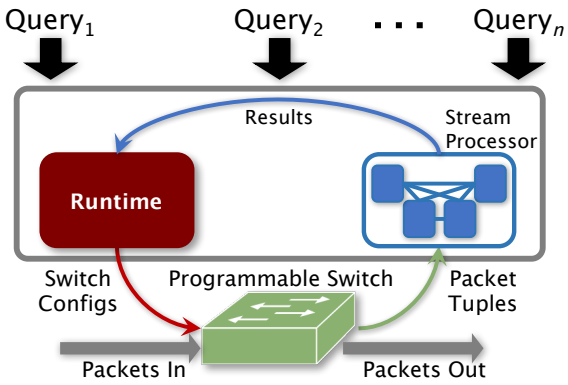


Figure 1: *Sonata architecture.*

Rather than accepting this apparent tradeoff between expressiveness and scalability, we observe that stream processors and programmable switches share a common processing model; they both apply an ordered set of transformations over structured data in a pipeline. This commonality suggests that an opportunity exists to combine the strengths of both technologies in a single telemetry system that supports expressive queries, while still operating at line rate for high traffic volumes and rates.

To explore this idea, we develop *Sonata* (Streaming Network Traffic Analysis), a *query-driven* network telemetry system. Figure 1 shows the design of Sonata: it provides a declarative interface that can express queries for a wide range of telemetry tasks and also frees the network operator from reasoning about where or how the query will execute. To scale query execution, Sonata (1) makes use of both programmable data-plane targets and scalable stream processors and (2) iteratively zooms-in on subsets of traffic that satisfy the query—making the best use of limited data-plane resources. By unifying stream processing and data-plane capabilities, Sonata's runtime can refine query execution in the data plane to reduce load on the stream processor. This ability

to dynamically refine queries is important because telemetry queries often require finding "needles in a haystack" where the fraction of total traffic or flows that satisfies these queries is tiny. We present the following contributions:

Unified query interface. (Section 2) We design a query interface that unifies the parsing and compute capabilities of a programmable switch with those of stream processors. This interface allows network operators to apply familiar dataflow operators (e.g., map, filter, reduce) over arbitrary combinations of packet fields without regard for where the query will execute. We show that a wide-range of network telemetry tasks can be expressed in fewer than 20 lines of Sonata code (Table 5).

Query partitioning based on data-plane constraints. (Section 3) To reduce load on the stream processor, we design an algorithm that partitions queries between the switch and the stream processor. We first show how dataflow queries can be partitioned without compromising accuracy. However, data-plane resources, such as switch memory and processing stages, are quite limited and inelastic by design. To make the best use of these resources, we develop an accurate model of common data-plane resource constraints and show how high-level dataflow operators consume these resources. Sonata's query planner uses this model to decide how to partition query execution between the switch and stream processor.

Dynamic query refinement based on query and workload. (Section 4) To efficiently use limited switch resources, we develop a dynamic refinement algorithm that ignores most of the traffic and only focuses on the subsets of traffic that actually satisfy a query. We show that this technique applies to a wide range of telemetry queries and demonstrate how Sonata's query planner considers the structure of queries and representative traffic traces to compute a refinement plan for each query.

Modular and extensible software architecture. (Section 5) To support different types of data-plane and streaming targets, we design Sonata so that it could be extended to support operations over arbitrary packet fields. The queries expressed using the Sonata interface are agnostic to the underlying switch and streaming targets. Our current prototype implements drivers for both hardware (e.g., Barefoot Tofino [48]) and software (e.g., BMV2 [49]) protocol-independent switches as well as the Spark Streaming [47] stream processor. The current prototype parses packet headers for standard protocols and can be extended to extract other information, such as queue size along a path [19].

The Sonata prototype is publicly available on Github [51], and consists of about 9,000 lines of code; it currently compiles queries to a single programmable switch. We use real packet traces from operational networks to demonstrate that Sonata's query planner reduces the load on the stream processor by as much as *seven orders of magnitude* over existing telemetry systems (Section 6). We also quantify how Sonata's performance gains depend on data-plane constraints and traffic dynamics. To date, our open-source software prototype has been used by both researchers at a large ISP and in a graduate networking course [5].

2 UNIFIED QUERY INTERFACE

This section presents Sonata's query interface and shows example queries to illustrate the types of queries that existing systems can and cannot support. Sonata provides a query interface that is as expressive as modern stream processors but opportunistically achieves the scalability of data-plane execution. Although Sonata uses programmable switches to scale query execution, the expressiveness of its query interface does not depend on the computational capabilities of these switches. Sonata's stream processor simply executes the operations for which no more data-plane resources are available or which are not supported by the switch (e.g., payload processing or floating-point arithmetic).

2.1 Dataflow Queries on Tuples

Extensible tuple abstraction. Information in packet headers naturally constitute key-value tuples (e.g., source and destination IP address, and other header values). This structure lends itself to a tuple-based abstraction [16]. Of course, an operator may want to write queries based on information that is not in the IP packet header, such as the application protocol, or DNS query type. To facilitate a broader range of queries, Sonata allows an operator to extend the tuple interface to include other fields that could be extracted by either the programmable switch or the stream processor. For example, they can specify customized packet-parsing operations for programmable switches in a language such as P4. Based on such a specification, the parser can extract all portions of the packet that pertain to the query. Sonata parses tuples on the switch whenever possible, shunting packets to the stream processor only when the query requires sophisticated parsing (e.g., parsing of the packet's payload) or join operations that the switch itself cannot support.

Expressive dataflow operators. Many network telemetry queries require computing aggregate statistics over a subset of traffic and joining the results from multiple queries, which can be expressed as a sequential composition of dataflow operators (e.g., filter, map, reduce). Gigascope [10], Chimera [7], and Marple [34] all use such a programming model, which is both familiar and amenable for compilation to programmable switches [34]. Section 3 describes how Sonata compiles queries across the stream processor and switch. Table 1 summarizes Sonata's dataflow operators. Stateful dataflow operators are all executed with respect to a query-defined time interval, or window. For example, applying `reduce` over a

```
1  packetStream(W)
2  .filter(p => p.tcp.flags == 2)
3  .map(p => (p.dIP, 1))
4  .reduce(keys=(dIP,), f=sum)
5  .filter((dIP, count) => count > Th)
```

Query 1: *Detect Newly Opened TCP Connections.*

sum operation will return a result at the end of each window. Each query can explicitly specify the interval's duration for stateful operations.

Operator	Description
filter(p)	Filter packets that satisfy predicate p.
map(f)	Transform each tuple with function f.
distinct()	Emit tuples with unique combinations of fields.
reduce(k, f)	Emit result of function f applied on key k over the input stream.
join(k, q)	Join the output of query q on key field k

Table 1: *Sonata's Dataflow Operators. All stateful operators execute with respect to a window interval of W seconds.*

Limitations. Sonata supports queries operating at packet-level granularity, as in existing declarative telemetry systems [7, 10, 34, 40]; it cannot support queries that require reassembling a byte stream, as in Bro [42]. Sonata also currently compiles each query to a single switch, *not* across multiple switches. We leave compiling arbitrary queries to multiple switches as future work, but the set of single-switch queries considered in this paper is nonetheless directly applicable to real-world deployments such as a border switch or an Internet exchange point (IXP).

2.2 Example Network Telemetry Queries

We now present three example queries: one that executes entirely in the data plane, a second that involves a join of two sub-queries, and a third that requires parsing packet payloads. Table 5 summarizes the queries that we have implemented and released publicly along with the Sonata software [52].

Computing aggregate statistics over a subset of traffic. Suppose that an operator wants to detect hosts that have too many recently opened TCP connections, as in a SYN flood attack. Detection requires parsing each packet's TCP flags and destination IP address, as well as computing a sum over the destination IP address field. Query 1 first applies a `filter` operation (line 2) over the entire packet stream to select TCP packets with just the `SYN` flag set. It then counts the number of packets it observed for each host (lines 3–4) and reports the hosts for which this count exceeds threshold `Th` at the end of the window (line 5). This query can be executed entirely on the switch, so existing systems (e.g., Marple [34]) can also execute this type of query at scale.

```
1  packetStream
2  .filter(p => p.proto == TCP)
3  .map(p => (p.dIP,p.sIP,p.tcp.sPort))
4  .distinct()
5  .map((dIP,sIP,sPort) =>(dIP,1))
6  .reduce(keys=(dIP,), f=sum)
7  .join(keys=(dIP,), packetStream
8    .filter(p => p.proto == TCP)
9    .map(p => (p.dIP,p.pktlen))
10   .reduce(keys=(dIP,), f=sum)
11   .filter((dIP, bytes) => bytes > Th1) )
12 .map((dIP,(byte,con)) => (dIP,(con/byte))
13 .filter((dIP, con/byte) => (con/byte > Th2)
```

Query 2: *Detect Slowloris Attacks.*

Joining the results of two queries. A more complex query involves joining the results from two sub-queries. To detect a Slowloris attack [44], a network operator must identify hosts which use many TCP connections, each with low traffic volume. This query (Query 2) consists of two sub-queries: the first sub-query counts the number of unique connections by applying a `distinct`, followed by a `reduce` (lines 1–6). The second sub-query counts the total bytes transferred for each host (lines 8–11). The query then joins the two results (line 7) to compute the average connections per byte (line 12) and reports hosts whose average number of connections per byte exceeds a threshold `Th2` (line 13). Marple [34] cannot support this query as it applies a join after an aggregation operation (reduce). Also, this query cannot be executed entirely in the data plane as computing an average requires performing a division operation. Even state-of-the-art programmable switches (e.g., Barefoot Tofino [48]) do not support the division operation in the data plane. In general, existing approaches that only use the data plane for query execution cannot support queries that require complex operations that cannot be executed in the switch. In contrast, Sonata's query planner partitions the query for partial execution on the switch and performs more complex computations at the stream processor.

Although this query is written to detect hosts for which the average bytes per connection exceeds a threshold, it is equivalent to detecting hosts for which the average bytes per connection is *less than* a different threshold; we explain in Section 4 why it is desirable to express this query using a "greater than" instead of a "less than" condition.

Processing packet payloads. Consider the problem of detecting the spread of malware via telnet [35], which is a common tactic targeting IoT devices [1]. Here, miscreants use brute force to gain shell access to vulnerable Internet-connected devices. Upon successful login, they issue a sequence of shell commands, one of which contains the keyword "zorro." The query to detect these attacks first looks

```
1  packetStream
2  .filter(p => p.tcp.dPort == 23)
3  .join(keys=(dIP,), packetStream
4    .filter(p => p.tcp.dPort == 23)
5    .map(p => ((p.dIP,p.nBytes/N), 1))
6    .reduce(keys=(dIP, nBytes), f=sum)
7    .filter(((dIP,nBytes),cnt1) => cnt1 > Th1))
8  .filter(p => p.payload.contains('zorro'))
9  .map(p => (p.dIP,1))
10 .reduce(keys=(dIP,), f=sum)
11 .filter((dIP, count2) => count2 > Th2)
```

Query 3: *Detect Zorro Attacks.*

for hosts that receive many similar-sized telnet packets followed by a telnet packet with a payload containing the keyword "zorro." The query for this task has two sub-queries (Query 3): the first part identifies hosts that receive more than Th1 similar-sized telnet packets rounded off by a factor of N (lines 4–7). The second part joins (line 3) the output of the first sub-query with the other and reports hosts that receive more than Th2 packets and contain the keyword "zorro" in the payload (lines 8–11). Since this query requires parsing packet payloads, many existing approaches cannot support it. In contrast, Sonata can support and scale these queries by performing as much computation as possible on the switch and then performing the rest at the stream processor.

3 QUERY PARTITONING

Sonata partitions a given query across a stream processor and a protocol-independent switch that performs part of the query, ultimately reducing the load (Section 3.1) on the stream processor. Section 3.2 discusses the constraints of these switches that the Sonata query planner considers; the planner solves an optimization problem to partition the query, as described in Section 3.3.

3.1 Data Reduction on the Switch

A central contribution of Sonata is to use the capabilities of programmable switches to reduce the load on the stream processor. In contrast to conventional switches, protocol-independent switch architecture (PISA) switches (e.g., RMT [9], Barefoot Tofino [20], Netronome [53]) offer programmable parsing and customizable packet-processing pipelines, as well as general-purpose registers for stateful operations. These features provide opportunities for Sonata to perform more of the query on the switch, reducing the amount of data sent to the stream processor.

3.1.1 Abstract Packet Processing Model. Figure 2 shows how Query 1 naturally maps to the capabilities of the packet processing model of a PISA switch. On PISA switches, a reconfigurable parser constructs a packet header vector (PHV)

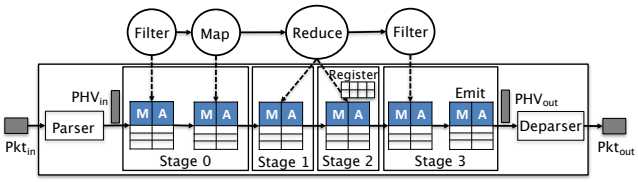


Figure 2: *Compiling a dataflow query (Query 1) to a sequence of match-action tables for a PISA switch. Each query consists of an ordered sequence of dataflow operators, which are then mapped to match-action tables in the data plane.*

for each incoming packet. The PHV contains not only fixed-size standard packet headers but also custom metadata for additional information such as queue size. A fixed number of physical stages, each containing one match-action unit (MAU), then processes the PHVs. The packet processing pipeline is a sequence of custom match-action tables; MAUs implement these abstract tables in hardware. Each MAU performs a self-contained set of match-action operations, consuming PHVs as input and emitting transformed PHVs as output. If fields in the PHV match a given rule in the MAU, then a set of custom actions corresponding to that rule are applied to the PHV. These actions can be stateless or stateful; the stateful operations use register memory to maintain state. Finally, a deparser reassembles the modified PHV into a packet before sending it to an output port.

The PISA processing model aligns well with streaming analytics platforms such as Spark Streaming [58] or Apache Flink [38]. The processing pipelines for both can be represented as a directed, acyclic graph (DAG) where each node in the graph performs some computation on an incoming stream of structured data. For stream processors, the nodes in the DAG are dataflow operators and the stream of structured data consists of tuples. For PISA switches, the nodes in the DAG are match-action tables and the stream of structured data consists of packets. Given this inherent similarity, an ordered set of dataflow query operators could map to an ordered set of match-action tables in the data plane. We now describe how Sonata takes advantage of this similarity to execute dataflow operators directly in the data plane.

3.1.2 Compiling Each Operator. Compiling dataflow queries to PISA switches requires translating the DAG of dataflow operators into an equivalent DAG of match-action tables. Prior work [34] also faced the challenge of compiling high-level queries to match-action tables, but limited the set of input queries to those that can be performed entirely on the switch. Rather than constraining the set of input queries, Sonata's query planner partitions all input queries into a set of dataflow operators than can be executed on the switch and a set that must be executed at the stream processor. Before Sonata's query planner can make this partitioning

decision, it must first quantify the resource requirements for individual dataflow operators.

Filter requires a single match-action table to match a set of fields in the PHV. For example, line 1 of Query 1 requires a single match-action table where the six-bit `tcp.flags` field is a column and the value 2 is a single rule (row), as shown in Figure 2. In general, the match-action table for a `filter` operation has a column for each field in the predicate. A filter predicate with multiple clauses connected by "and" leads to multiple rules, one per clause.

Map also requires a single match-action table. For example, line 2 of Query 1 transforms all incoming packets into a tuple consisting of the `ipv4.dIP` field from the packet's header and the value 1. These values are stored in query-specific metadata for further processing. Although Sonata's query interface does not constrain the transformations that `map` might perform over a set of tuples, the operator cannot be compiled to the data plane if the switch cannot perform the corresponding transformation.

Reduce requires maintaining state across sequences of packets; Sonata uses registers, which are simply arrays of values indexed by some key, to do so. Query-specific metadata fields permit loading and storing values from the registers. As a result, stateful operations require two match-action tables: one for computing the index of the value stored in the register and the other for updating state using arithmetic and logic operators supported by the switch, such as `add` and `bit_or`. A corresponding metadata field carries the updated state after applying the arithmetic operation. For example, executing the `reduce` operator for Query 1 in Figure 2 requires a match-action table to compute an index into the register using the `dIP` header field. A second table performs the stateful action that increments the indexed value in the register and stores the updated value. In Section 3.3, we describe how Sonata's query planner uses representative training data to configure the number of entries for each register.

Distinct operations are similar to a `reduce`, where the function `bit_or` (with argument 1) is applied to a single bit.

Join operations are costly to execute in the data plane. In the worst case, this operation maintains state that grows with the square of the number of packets. Sonata executes `join` operations at the stream processor by iteratively dividing the query into a set of sub-queries. For example, Sonata divides Query 2 into two sub-queries: one that computes the number of unique connections, and a second that computes the number of bytes transferred for each host. Sonata independently decides how to execute the two sub-queries and ultimately joins their results at the stream processor.

3.1.3 Compiling Dataflow Queries. In addition to mapping individual dataflow operators to match-action tables, the resulting data-plane mapping must be synthesized in a way that respects the following additional considerations.

Preserving packet forwarding decisions. Sonata preserves packet forwarding decisions by transforming only query-specific metadata fields, rather than the packet contents that might affect forwarding decisions (e.g., destination address, application headers, or even payload). The switch extracts values from the packets' original header fields and copies them to auxiliary metadata fields before performing any additional processing. This process leaves the original packet unmodified.

Reporting intermediate results to the stream processor. When a query is partitioned across the stream processor and the switch, the stream processor may need either the original packet or just an intermediate result from the switch, so that it can perform its portion of the query. To facilitate this reporting, the switch maintains a one-bit `report` field in the metadata for each packet. Each query partitioned to the switch marks this field whenever a query-specific condition is met that requires a packet be sent to the stream processor. If this field is set at the conclusion of the entire processing pipeline, the switch sends to the stream processor all intermediate results needed to complete processing the query, including the original packet if needed by the query. If the last operator is stateful (e.g., `reduce`), then the switch sends only one packet for each key to the stream processor. This informs the stream processor which register indices in the data plane must be polled at the end of each window to retrieve aggregated values stored in the switch (see Section 5 for details).

Detecting and mitigating hash collisions. Sonata must detect and mitigate hash collisions that may result at the switch. The probability of a hash collision is proportional to the number of hashes performed on unique keys and the size of the output hash as a consequence of the pigeonhole principle. In theory, a 32-bit hash has a 50% chance of a collision after hashing fewer than 80,000 keys. Since true hash-tables with collision resolution are not available on the switch, we instead use registers with hash-based indices. In practice, these registers contain far fewer rows than the number of unique values in the hash output, making collisions even more likely. To detect collisions, switches store the original key when performing `reduce` and `distinct` operations. To mitigate collisions, Sonata uses a sequence of up to d registers, each using a different hash function for determining indices. If a key generates a collision, Sonata iterates through each of the d registers, storing the key in the first register that does not result in a collision. If after iterating through all d registers, the key still generates a collision, Sonata sends the packet to the stream processor. At the end of each window, the stream processor adjusts the

results received from the switch with the additional packets processed due to collisions.

3.2 Data-Plane Resource Constraints

Sonata's query planner must consider the finite resource constraints of PISA switches for parsing packet header fields, performing actions on packets, storing state in register memory and performing all of these operations in a limited number of stages.

Parser. The cost of parsing increases with the number of fields to extract from the packet. This cost is quantified as the number of bits to extract and the depth of the parsing tree. The size of the PHV limits the number of fields that can be extracted for processing. Typically, PISA switches have PHVs about 0.5–8 Kb [9] in size. Let M denote the maximum storage for metadata in the PHV.

Actions. Most stream processors execute multiple queries in parallel, where each query operates over its own logical copy of the input tuple. In contrast, PISA switches transform raw packets to PHVs and then concurrently apply multiple operations over the PHV in pipelined stages. These mechanisms suggest that PISA switches would be amenable to parallel query execution. In practice, there is a limit on how many actions can be applied over a PHV in one stage, which limits the number of queries that can be supported in the data plane. Typically, PISA switches support 100–200 stateless and 1–32 stateful actions per stage [9]; we denote the maximum number of stateful actions per stage as A.

Registers. The amount of memory required to perform stateful operations grows with the number of packets and the number of queries. Stream processors scale by adding more nodes for maintaining additional state. In contrast, stateful operations in PISA switches can only access register memory locally available to their physical stage. This register memory is bounded for each stage, which affects the switch's ability to handle both increased traffic loads and additional queries. Within a stage, the amount of memory available to a single register is also bounded. Typically, PISA switches support 0.5–32 Mb memory for each stage [9]. Let B denote the maximum number of register bits available in each stage.

Stages. Queries that lack available resources in a given stage must execute in a later stage. PISA switches typically support 1–32 physical stages [9]; we denote the maximum number of stages as S.

3.3 Computing Query Partitioning Plans

Consider a switch with $S = 4$ stages, $B = 3,000$ Kb, and $A = 4$ stateful actions per stage. These constraints are more strict than Barefoot's Tofino switch [48], but they illustrate how the data-plane resource constraints affect query planning. Sonata runs Query 1 over a one-minute packet trace from CAIDA [11] to compute that the switch requires 2,500 Kb to count the number of TCP SYN packets per host (Figure 5). Since 2,500 Kb $< B$, Sonata can execute the entire query on the switch, sending only the 77 tuples that satisfy the query to the stream processor. If B or S were smaller, Sonata could not execute the `reduce` operator on the switch and would need to partition the query. The rest of this section describes how Sonata computes such query plans.

Switch Constraints	
M	Amount of metadata stored in switch.
A	Number of stateful actions per stage.
B	Register memory (in bits) per stage.
S	Number of stages in match-action pipeline.
Input from Queries	
O_q	Ordered set of dataflow operators for query q.
T_q	Ordered set of match-action tables for query q.
M_q	Amount of metadata required to perform query q.
Z_t	Indicates whether table t performs a stateful operation.
Input from Workload	
$N_{q,t}$	Number of packets generated after table t of query q.
$B_{q,t}$	State (bits) required for executing table t of query q.
Output	
$P_{q,t}$	Indicates whether t is the last table partitioned to the switch for query q.
$X_{q,t,s}$	Indicates whether table t of query q executes at stage s in the switch.
$S_{q,t}$	Stage id for table t for query q.

Table 2: *Summary of variables in the query planning problem.*

Sonata's query planner solves an Integer Linear Program (ILP) that minimizes the number of packet tuples sent to the stream processor based on a partitioning plan, subject to switch constraints, as summarized in Table 3. Our approach is inspired by previous work on a different problem that partitions multiple logical tables across physical tables [22]. Table 2 summarizes the variables in the query planning problem. To select a partitioning plan, the query planner determines the capabilities of the underlying switch, estimates the data-plane resources needed to execute individual queries, and estimates the number of packets sent to the stream processor given a partitioning of operators on the switch.

Input. For the set of input queries (Q), Sonata interacts with the switch to compile the ordered set of dataflow operators (O_q) in each query q to an ordered set of match-action tables (T_q) that implement the operators on the switch. In some cases, more than one dataflow operator can be compiled to the same table. For instance, the filter operator that checks the threshold after the reduce in Query 1 can be compiled to the same table as the reduce operator. Z_t indicates to the query planner whether a given table contains a stateful operator.

Using training data in the form of historical packet traces, the query planner estimates the number of packet tuples ($N_{q,t}$) sent to the stream processor and the amount of state ($B_{q,t}$) required to execute table t for query q on the switch. The planner applies all of the packets in the historical traces to each query q. After applying each table t that contains a

Goal

$$min(N = \sum_{q} \sum_{t} P_{q,t} \cdot N_{q,t})$$

Constraints

$$C1: \quad \forall s : \sum_{q} \sum_{T_q} X_{q,t,s} \cdot B_{q,t} \leq B$$

$$C2: \quad \forall s : \sum_{q} \sum_{T_q} Z_t \cdot X_{q,t,s} \leq A$$

$$C3: \quad \forall q, t : S_{q,t} < S$$

$$C4: \quad \forall q, i < j, i, j \in T_q : S_{q,j} > S_{q,i}$$

$$C5: \quad \forall q : \sum_{q} M_q \leq M$$

Table 3: *ILP formulation for the query partitioning problem.*

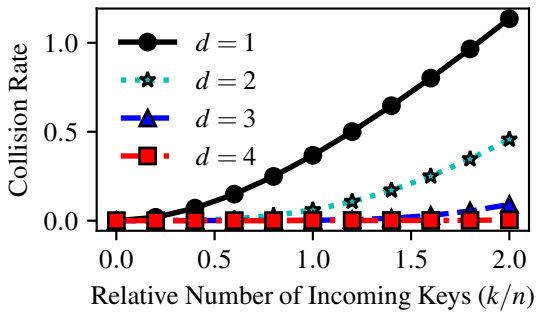


Figure 3: *Relationship between collision rate and the number of unique incoming keys (k) relative to the estimate (n).*

stateful operator, the planner estimates the amount of state required to perform the stateful operation based on the total number of keys processed in the historical traces. It also estimates the number of packets sent to the stream processor ($N_{q,t}$) after table t processes the packets from the historical traces. The planner divides the historical traces into time windows of size W, computes $B_{q,t}$ and $N_{q,t}$ per window, and inputs the median value across all intervals to the ILP.

Objective. The objective of Sonata's query planning ILP is to minimize the number of packets processed by the stream processor. The query planner models this objective by introducing a binary decision variable $P_{q,t}$ that captures the partitioning decision for each query; $P_{q,t} = 1$ one if t is the last table for query q that is executed on the switch. For each query, only one table corresponding to one operator can be set as the last table on the switch: $\sum_{T_q} P_{q,t} \leq 1$. The total number of packets processed by the stream processor is then the sum of all packets emitted by the last table processed on the switch for all queries.

Switch constraints. To ensure that Sonata respects the constraints from Section 3.2, we introduce variables X and S. $X_{q,t,s}$ is a binary variable that reflects stage assignment: $X_{q,t,s} = 1$ only if table t for query q executes at stage s in the match-action pipeline. Similarly, $S_{q,t}$ returns the stage number where table t for query q is executed. These two variables are related: if $X_{q,t,s} = 1$, then $S_{q,t} = s$ for a given stage. We will now summarize how Sonata's query planner models various data-plane constraints.

C1: Register Memory per stage (B). For each stage, the amount of state allocated for Sonata's packet processing cannot exceed B. Since PISA targets can only configure tables with stateful operations in a single stage, the amount of state required to execute query q at stage s is $\sum_{T_q} X_{q,t,s} \cdot B_{q,t}$. This sum over all queries captures the total memory required for each stage s.

C2: Number of Actions per stage (A). For each stage, the total number of stateful operations cannot exceed A. We can again use the X variable to model this constraint. The expression $\sum_{T_q} Z_t \cdot X_{q,t,s}$ captures the number of stateful operations performed at stage s for query q. This sum over all queries captures the total number of stateful actions for each stage s.

C3: Number of Stages (S). The total number of stages required to execute a query in the data plane cannot exceed S. The variable $S_{q,t}$ represents the stage where table t for query q is executed. For every table of each query, this variable should always be less than S because the last stage is reserved to determine which packet needs to be reported to the stream processor.

C4: Intra-Query Ordering. We can also use S to express intra-query ordering constraints. For example, in the Slowloris query (Query 2), the tables for the `reduce` operator can only be executed after the `distinct` operator has been applied in a previous stage. For each query q and any two indices (i, j) in the ordered set of tables T_q where $(i < j)$, $S_{q,j}$ is always greater than $S_{q,i}$.

C5: Total Metadata (M). Finally, since the PHV consists of a fixed-size, (M) represents the maximum space available in the PHV to add query-specific metadata fields. The total metadata used for all queries must then be less than M, *i.e.* $\sum_q M_q \leq M$.

Monitoring traffic dynamics. The query planner uses training data to decide how to configure the number of entries (n) for each register, and how many registers (d) to use for each stateful operation. It is possible that the training data might underestimate the number of expected keys (k) for a stateful operation due to variations in traffic patterns. In Figure 3, we show how the collision rates increase as the number of unique keys grows beyond the original estimate (n) for a sequence of (d) registers. Here, the x-axis is the number of incoming keys relative to the original estimate and the y-axis is the collision rate. The collision rate increases as the relative number of incoming keys increases and decreases as the number of registers increases.

Since collision rates are predictable, we choose values of (n) and (d) to keep collision rates low but still high enough to send a signal to Sonata's runtime when the switch is

storing many more unique keys than originally expected. Sonata's query planning ILP considers both the number of additional packets processed by the stream processor and the additional switch memory while computing the optimal query partitioning plans.

4 DYNAMIC QUERY REFINEMENT

For certain queries and workloads, partitioning a subset of dataflow operators to the switch does not reduce the workload on the stream processor enough; in these situations, Sonata uses historical packet traces to refine input queries dynamically.

To do so, Sonata's query planner modifies the input queries to start at a coarser level of granularity than specified in the original query (Section 4.1). It then chooses a sequence of finer granularities that reduces the load on the stream processor. This process introduces additional delay in detecting the traffic that satisfies the input queries. The specific levels of granularity chosen and the sequence in which they are applied constitute a *refinement plan*. To compute an optimal refinement plan for the set of input queries, Sonata's query planner estimates the cost of executing different refinement plans based on historical packet traces. Sonata's query planner then solves an extended version of the ILP from Section 3.3 that determines both partitioning as well as refinement plans to minimize the workload on the stream processor (Section 4.2).

4.1 Modifying Queries for Refinement

Identifying refinement keys. A refinement key is a field that has a hierarchical structure and is used as a key in a stateful dataflow operation. The hierarchical structure allows Sonata to replace a more specific key with a less specific version without missing any traffic that satisfies the original query. This applies to all queries that filter on aggregated counts greater than a threshold. For example, `dIP` has a hierarchical structure and is used as a key for aggregation in Query 1. As a result, the query planner selects `dIP` as a refinement key for this query. Other fields that have hierarchical structure can also serve as refinement keys, such as `dns.rr.name` and `ipv6.dIP`. For example, a query for detecting malicious domains that requires counting the number of unique resolved IP address for each domain [6], can use the field `dns.rr.name` as a refinement key. Here, a fully-qualified domain name is the finest refinement level and the root domain (`.`) is the coarsest. A query can contain multiple candidate refinement keys and Sonata independently selects refinement keys for each query. Additionally, expressing the second sub-query in Query 2 as the one that reports flows for which the average connections per bytes exceeds

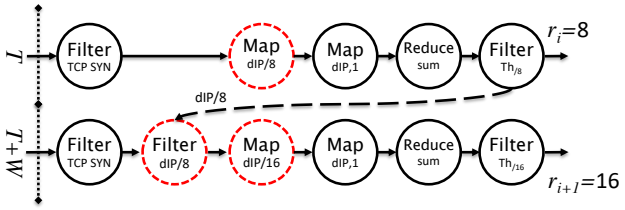


Figure 4: *Query augmentation for Query 1. The query planner adds the operators shown in red to support refinement. Query 1 executes at refinement level r_i = `/8` during window T and at level r_{i+1} =`/16` during window $(T + W)$. The dashed arrow shows the output from level r_i feeding a filter at level r_{i+1}.*

$r_i \longrightarrow r_{i+1}$	N_1	B (Kb)	N_2
$*\longrightarrow 32$	570K	2,500	77
$*\longrightarrow 16$		180	99
$*\longrightarrow 8$		6	33
$8 \longrightarrow 32$	526K	1,900	77
$8 \longrightarrow 16$		50	98
$16 \longrightarrow 32$	450K	1,200	77

Figure 5: *The N and B cost values for executing Query 1 at refinement level r_{i+1} after executing it at level r_i.*

the threshold ensures that it can benefit from iterative refinement, because replacing a more specific key with a less specific one will not miss any traffic that satisfies the original query.

Enumerating refinement levels. After identifying candidate refinement keys, the query planner enumerates the possible levels of granularity for each key. Each refinement key consists of a set of levels $R = \{r_1 \ldots r_n\}$ where r_1 is the coarsest level and r_n is the finest. The inequality $r_1 > r_n$ means that r_1 is coarser than r_n. The semantics of the i^{th} refinement level is specific to each key; $r_i = 32$ would correspond to a `/32` IP prefix for the key `dIP` and $r_i = 2$ would correspond to the second-level domain for the key `dns.rr.name`. For each refinement key, the query planner will choose a subset of these refinement levels for a refinement plan. For simplicity, we will refer to the chosen subset of refinement levels as R.

Augmenting input queries. To ensure that the finer refinement levels only consider the traffic that has already satisfied coarser ones, Sonata's query planner augments the input queries. For example, Figure 4 shows how it augments Query 1 with refinement key `dIP` and R ={8, 16, 32} to execute the query at level $r_{i+1} = 16$ after executing it at level $r_i = 8$. The query planner first adds a `map` at each level to transform the original reduction key into a count bucket for the current refinement level. For example, r_i and r_{i+1} rewrite `dIP` as `dIP/8` and `dIP/16`, respectively. By transforming the

reduction key for each refinement level, the rest of the original query can remain unmodified. At refinement level r_{i+1}, the query planner also adds a `filter`. At the conclusion of the first time window, the runtime feeds as input to the `filter` operator the `dIP/8` addresses that satisfy the query at $r_i = 8$. This filtering ensures that refinement level r_{i+1} only considers traffic that satisfies the query at r_i.

Sonata's query planner also augments queries to increase the efficiency of executing refined queries. Because counting at coarser refinement levels (e.g., `/8`) will result in larger sums than at finer levels (e.g., `/32`), using the *original* query's threshold values at coarser refinement levels would still be correct but inefficient. Sonata's query planner instead uses training data to calculate relaxed threshold values for coarser refinement levels that do not sacrifice accuracy (e.g., $\mathtt{Th}_{/8} > \mathtt{Th}_{/16}$ in Figure 4). For each query and for each refinement level, the planner selects a relaxed threshold that is the minimum count for all keys satisfying the original query aggregated at that refinement level.

Dynamic refinement is also appropriate for queries that require `join` operations (e.g., Query 2). The two sub-queries use the same refinement plan and their output at coarser levels determines which portion of traffic to process for the finer levels.

By its very nature, dynamic refinement introduces additional delay (D) in detecting the traffic that satisfies the original input queries. In the worst case, Sonata can only identify network events lasting at least $W \times |R|$ seconds for each query. Here, W is the interval size and $|R|$ is the total number of refinement levels considered. However, by specifying an upper bound on the acceptable delay (D_q), the network operator can force Sonata to consider fewer refinement levels and reduce the delay to detect traffic that satisfies the original query.

4.2 Computing Refinement Plans

Dynamic query refinement example. Sonata's query planner applies the augmented queries over the training data to generate Figure 5 for Query 1. This figure shows the costs to execute Query 1 with refinement key `dIP` and refinement levels $R = \{8, 16, 32\}$ over the training data. It shows the number of packets sent to the stream processor depending on which refinement level (r_{i+1}) is executed after level r_i. If only the `filter` operation is executed on the switch, then N_1 packets are sent to the stream processor. If the `reduce` operation is also executed on the switch, then N_2 packets are sent, but then B bits of state must also be maintained in the data plane. For simplicity of exposition, we assume that these counts remain the same for three consecutive windows.

Consider an approach, *Fixed-Refinement*, that applies a fixed refinement plan for all input queries. In this example, the query planner augments the original queries to always run at refinement levels 8, 16, and 32. The runtime updates the filter for the query at level 16 with the output from level 8 and the filter of level 32 with the output from 16. The costs of this plan are shown in rows $* \rightarrow 8$, $8 \rightarrow 16$, and $16 \rightarrow 32$ of Figure 5. Because the switch only supports two stateful operations ($A = 2$), the `reduce` operator could only be performed on the switch for the first two refinement levels. This would result in sending 33 packets (N_2 for $* \rightarrow 8$) at the end of the first window, 98 packets (N_2 for $8 \rightarrow 16$) at the end of the second window, and 450,000 (N_1 for $16 \rightarrow 32$) packets at the end of the third window to the stream processor. Compared to the solution without any refinement from beginning of Section 3.3, *Fixed-Refinement* reduces the number of tuples reported to the stream processor from 570 K to 450 K at the cost of delaying two additional time windows to detect traffic that satisfies the query.

In contrast, Sonata's query planner uses the costs in Figure 5 combined with the switch constraints to compute the refinement plan $* \rightarrow 8 \rightarrow 32$. Executing the query at refinement level $* \rightarrow 8$ requires only 6 Kb of state on the switch and sends 33 packet tuples to the stream processor at the end of the first window. Each packet represents an individual `dIP/8` prefix that satisfies the query in the first window. Sonata then applies the original input query (`dIP/32`) over these 33 `dIP/8` prefixes in the second window interval, processing 526,000 packets (N_1 for $8 \rightarrow 32$) and consuming only 1900 Kb on the switch. At the end of the second window, the switch reports 77 `dIP/32` addresses to the stream processor. This refinement plan sends 110 packet tuples to the stream processor over two window intervals, significantly reducing the workload on the stream processor while costing only one additional window of delay.

ILP for dynamic refinement. The ILP for jointly computing partitioning and refinement plans is an extension to the ILP from Section 3.3. Table 4 presents the full version of the extended ILP, including these new constraints. The objective is the same, but the query planner must also compute the cost of executing combinations of refined queries (*i.e.*, $N_{q,t,r}$ and $B_{q,t,r}$) to estimate the total cost of candidate query plans. We add new decision variables $I_{q,r}$ and F_{q,r_1,r_2} to model the workload on the stream processor in the presence of refined queries. $I_{q,r}$ is set to one if the refinement plan for query q includes level r. F_{q,r_1,r_2} is set to one if level r_2 is executed after r_1 for query q. These two variables are related by $\sum_{r_1} F_{q,r_1,r_2} = I_{q,r_2}$. We also augment X and S variables with subscripts to account for refinement levels.

Additional constraints. For queries containing `join` operators, the query planner can select refinement keys for each sub-query separately, but it must ensure that both sub-queries use the same refinement plan. We then add the constraint $\forall q, r$ and $\forall q_i, q_j \in q : I_{q_i,r} = I_{q_j,r}$. The variables q_i

Goal	
	$min(N = \sum_q \sum_{r_2} L_{q,t,r_2} \cdot N_{q,t,r_2})$
	$N_{q,t,r_2} = I_{q,r_2} \cdot \sum_{r_1} F_{q,r_1,r_2} \cdot N_{q,t,r_1,r_2}$
Constraints	
$C1:$	$\forall s: \sum_q X_{q,s,t} \cdot B_{q,t} \leq B_{max}$ $B_{q,t} = \sum_{r_2} I_{q,r_2} \sum_{r_1} F_{q,r_1,r_2} \cdot B_{q,r_2,t}$
$C2:$	$\forall s: \sum_q \sum_t X_{q,t,s} \leq W_{max}$ $X_{q,t,s} = \sum_r I_{q,r} \cdot X_{q,t,s,r}$
$C3:$	$\forall q, t, r: S_{q,t,r} \leq S_{max} - 1$
$C4:$	$\forall q, r, i < j: S_{q,j,r} < S_{q,i,r}$
$C5:$	$\forall q: \sum_q \sum_r I_{q,r} \cdot M_{q,r} \leq M$
$C6:$	$\forall q_i, q_j, r: I_{q_i,r} = I_{q_j,r}$
$C7:$	$\forall q: \sum_r I_{q,r} \leq D_q$

Table 4: *Extension of ILP to support dynamic refinement.*

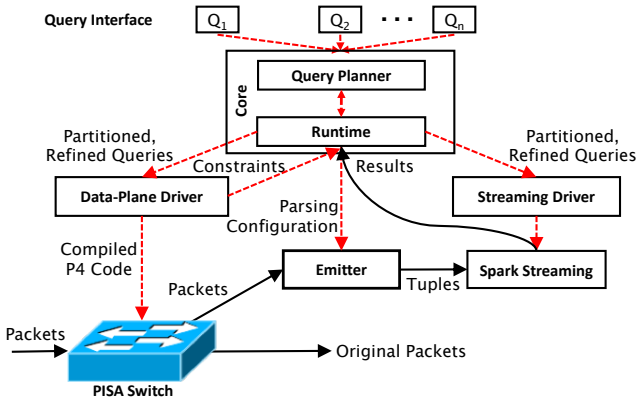


Figure 6: *Sonata's implementation: red arrows show compilation control flow and black ones show packet/tuple data flow.*

and q_j represent sub-queries of query q containing a `join` operation. The query planner also limits the maximum detection delay for each query, $\forall q : \sum_r I_{q,r} \leq D_q$. Here, D_q is the maximum delay query q can tolerate expressed in number of time windows.

5 IMPLEMENTATION

Figure 6 illustrates the Sonata implementation. For each query, the *core* generates partitioned and refined queries; *drivers* compile the parts of each query to the appropriate component. When packets arrive at the PISA switch, Sonata applies the packet-processing pipelines and mirrors the appropriate packets to a monitoring port, where a software *emitter* parses the packets and sends the corresponding tuples to the stream processor. The stream processor reports the results of the queries to the runtime, which then updates the switch, via the data-plane driver, to perform dynamic refinement.

Core. The core has two modules: (1) the query planner and (2) the runtime. Upon initialization or re-training, the runtime polls the data-plane driver over a network socket to determine which dataflow operators the switch is capable of executing, as well as the values of the data-plane constraints (*i.e.*, M, A, B, S). It then passes these values to the query planner which uses Gurobi [17] to solve the query planning ILP offline and to generate partitioned, refined queries. The runtime then sends partitioned and refined queries to the data-plane and streaming drivers. It also configures the emitter—specifying the fields to extract from each packet for each query; each query is identified by a corresponding query identifier (`qid`). When the switch begins processing packets, the runtime receives query outputs from the stream processor at the end of every window. It then sends updates to the data-plane driver, which in turn updates table entries in the switch according to the dynamic refinement plan. When it detects too many hash collisions, the runtime triggers the query planner to re-run the ILP with the new data.

Drivers. Data-plane and streaming drivers compile the queries from the runtime to target-specific code that can run on the switch and stream processor, respectively. The data-plane drivers also interact with the switch to execute commands on behalf of the runtime, such as updating `filter` tables for iterative refinement at the end of every window. The Sonata implementation currently has drivers for two PISA switches: the BMV2 P4 software switch [49], which is the standard behavioral model for evaluating P4 code; and the Barefoot Wedge 100B-65X (Tofino) [48], which is a 6.5 Tbps hardware switch. The data-plane driver communicates with these switches using a Thrift API [2]. The current implementation also has a driver for the Apache Spark [47] streaming target for processing packet tuples in the user-space and reporting the output of each query to Sonata's runtime.

Emitter. The emitter consumes raw packets from the data-plane's monitoring port, parses the query-specific fields in the packet, and sends the corresponding tuples to the stream processor. The emitter uses Scapy [50] to extract the unique (`qid`) from packets. It uses this identifier to determine how to parse the remainder of the query-specific fields embedded in the packet based on the configuration provided by the runtime. As discussed in Section 3.1.3, the emitter immediately sends the output of stateless operators to the stream processor, but it stores the output of stateful operators in a local key-value data store. At the end of each window interval, it reads the aggregated value for each key in the local data store from the data-plane registers before sending the output tuples to the stream processor.

#	Query	Lines of Code		
		Sonata	*P4*	*Spark*
1	Newly opened TCP Conns. [57]	6	367	4
2	SSH Brute Force [21]	7	561	14
3	Superspreader [55]	6	473	10
4	Port Scan [24]	6	714	8
5	DDoS [55]	9	691	8
6	TCP SYN Flood [57]	17	870	10
7	TCP Incomplete Flows [57]	12	633	4
8	Slowloris Attacks [57]	13	1,168	15
9	DNS Tunneling [7]	11	570	12
10	Zorro Attack [35]	13	561	14
11	DNS Reflection Attack [25]	14	773	12

Table 5: *Implemented Sonata Queries. We report lines of code considering the same: refinement and partitioning plans, executing as many dataflow operators in the switch as possible.*

6 EVALUATION

In this section, we first demonstrate that Sonata is expressive (Table 5). We then use real-world packet traces to show that it reduces the workload on the stream processor by 3–7 orders of magnitude (Figure 7) and that these results are robust to various switch resource constraints (Figure 8). Finally, we present a case study with a Tofino switch to demonstrate how Sonata operates end-to-end, discovering "needles" of interest without collecting the entire "haystack" (Figure 9).

6.1 Setup

Telemetry applications. To demonstrate the expressiveness of Sonata's query interface, we implemented eleven different telemetry tasks, as shown in Table 5. We show how Sonata makes it easier to express queries for complex telemetry tasks by comparing the lines of code needed to express those tasks. For each query, Sonata required far fewer lines of code to express the same task than the code for the switch [8] and streaming [47] targets combined. Not only does Sonata reduce the lines of code, but also the queries expressed with Sonata are platform-agnostic and could execute unmodified with a different choice of hardware switch or stream processor, e.g., Apache Flink.

Packet traces. We use CAIDA's anonymized and unsampled packet traces [43], which were captured from a large ISP's backbone link between Seattle and Chicago. We evaluate over a subset of this data containing 600 million packets and transferring about 360 GB of data over 10 minutes. This data contains no layer-2 headers or packet payloads, and the layer-3 headers were anonymized with a prefix-preserving algorithm [14].

Query planning. For query planning, we consider a maximum of eight refinement levels for all queries (*i.e.*, $R = \{4, 8, \ldots, 32\}$); additional levels offered only marginal improvements. We replay the packet traces at $20x$ speed to evaluate Sonata on a simulated 100 Gbps workload (*i.e.*, about 20 million packets per second) that might be experienced at a border switch in a large network. We use a time window (W) of three seconds. In general, selecting a shorter time interval is desirable; however, for very short time intervals the overhead of updating the filter rules in the data plane at the end of each window can introduce significant errors. Our choice of three seconds strikes a balance between achieving a tolerable detection delay and minimizing the errors introduced by the data-plane update overhead. Sonata's query planner processed around 60 million packets for each time interval to estimate the number of packet tuples (N) and the register sizes (B). We observed that although the ILP solver could compute near-optimal query plans in 10–20 minutes, the solver typically required several hours to determine the optimal plans. Since running the ILP solver for longer durations had diminishing returns, we selected a time limit of 20 minutes for the ILP solver to report the best (possibly sub-optimal) solution that it found in that period.

Query Plan	Description	Telemetry Systems
All-SP	Mirror all incoming packets to the stream processor	Gigascope[10], OpenSOC[40], NetQRE[57]
Filter-DP	Apply only filter operations on the switch	EverFlow[59]
Max-DP	Execute as many dataflow operations as possible on the switch	Univmon[26], OpenSketch[55]
Fix-REF	Iteratively zoom-in one refinenement level at a time	DREAM[29]

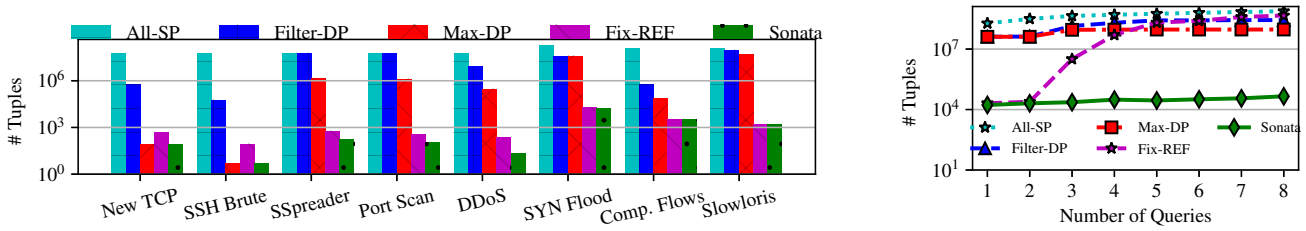
Table 6: *Telemetry systems emulated for evaluation.*

Targets. Since switches have fixed resource constraints, we choose to evaluate Sonata's performance with simulated PISA switches. This approach allows us to parameterize the various resource constraints and to evaluate Sonata's performance over a variety of potential PISA switches. Unless otherwise specified, we present results for a simulated PISA switch with sixteen stages ($S = 16$), eight stateful operators per stage ($A = 8$), and eight Mb of register memory per stage ($B = 8$ Mb). Within each stage, a single stateful operator can use up to four Mb.

Comparisons to existing systems. We compare Sonata's performance to that of four alternative query plans. Each plan is representative of groups of existing systems, such as Gigascope [10], OpenSOC [41], EverFlow [59], OpenSketch [55], and DREAM [29], as shown in Table 6. Rather than instrumenting each of these systems, we emulate them by modifying the constraints on the Sonata's query-planning ILP. For example to emulate the *Fix-REF* plan, we add the constraint $\forall q, r : I_{q,r} = 1$.

6.2 Load on the Stream Processor

We perform trace-driven analysis to quantify how much Sonata reduces the workload on the stream processor. To



(a) *Single-query performance*

(b) *Multi-query performance*

Figure 7: *Reduction in workload on the stream processor running: (a) one query at a time, (2) concurrently running multiple queries.*

enable comparison with prior work, we evaluate the top eight queries from Table 5; these queries process only layer 3 and 4 header fields. *Fix-REF* queries use all eight refinement levels, while Sonata may select a subset of all eight levels in its query plans.

Single query performance. Figure 7a shows that Sonata reduces the workload on the stream processor by as much as seven orders of magnitude. *Filter-DP* is efficient for the SSH brute-force attack query, because this query examines such a small fraction of the traffic. *Filter-DP*'s performance is similar to *All-SP* for queries that must process a larger fraction of traffic, such as detecting Superspreaders [55]. For some queries, such as the SSH brute-force attack, *Max-DP* matches Sonata's performance. In many other cases, large amounts of traffic are sent to the stream processor due to a lack of resources. For example, the Superspreader query exhausts stateful processing resources. *Fix-REF*'s performance matches Sonata's for most cases, but uses up to seven additional windows to detect traffic that satisfies the query.

Multi-query performance. Figure 7b shows how the workload on the stream processor increases with the number of queries. When executing eight queries concurrently, Sonata reduces the workload by three orders of magnitude compared to other query plans. These gains come at the cost of up to three additional time windows to detect traffic that satisfies the query. The performance of *Fix-REF* degrades the most because the available switch resources, such as metadata and stages, are quickly exhausted when supporting a fixed refinement plan for several queries. We have also considered query plans with fewer refinement levels for *Fix-REF* and observed similar trends. For example, when considering just two refinement levels (`dIP/16` and `dIP/32`) for all eight queries, we observed that the load on the stream processor was two orders of magnitude greater than Sonata.

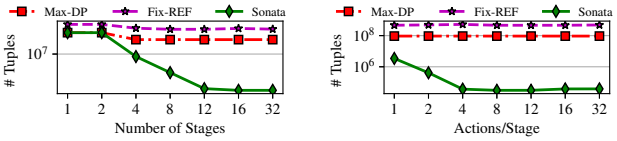
As the number of queries increases, the number of tuples will continue to increase and eventually be similar to *All-SP*. Although Sonata makes the best use of limited resources for a given target, its performance gains are bounded by the available switch resources. It is important to differentiate the limitations on Sonata's performance from the limitations imposed by existing hardware switches. While today's commodity hardware switches can support tens of network monitoring applications, we envision that the next-generation hardware switches will be able to support hundreds if not thousands of concurrent monitoring queries with Sonata.

Effect of switch constraints. We study how switch constraints affect Sonata's ability to reduce the load on the stream processor. To quantify this relationship, we vary one switch constraint at a time for the simulated PISA switch. Figure 8a shows how the workload on the stream processor decreases as the number of stages increases. More stages allow Sonata to consider more levels for dynamic refinement. Additional stages slightly improve the performance of *Fix-REF* as it can now support stateful operations for the queries at finer refinement levels on the switch. We observe similar trends as the number of stateful actions per stage (Figure 8b), memory per stage (Figure 8c), and total metadata size (Figure 8d) increase. As expected, *Max-DP* slightly reduces the load on the stream processor when more memory per stage is available for stateful operations; increasing the total metadata size also allows *Fix-REF* to execute more queries in the switch—reducing the load on the stream processor.

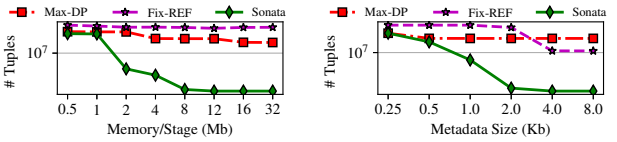
Overhead of dynamic refinement. When running all eight queries concurrently, as many as 200 `filter` table entries are updated after each time window during dynamic refinement. Micro-benchmarking experiments with the Tofino switch [48] show that updating 200 table entries takes about 127 ms, and resetting registers takes about 4 ms. The total update time took 131 ms which is about 5% of the specified window interval ($W = 3$ s).

6.3 Case Study: Tofino Switch

We used Sonata to execute Query 3 with a Tofino switch [48]. We chose this query to highlight how Sonata handles `join` operators and operations over a packet's payload. For this experiment, we built a testbed containing four hosts and a



(a) *Maximum pipeline depth.* **(b)** *Maximum pipeline width.*



(c) *Maximum bits per stage.* **(d)** *Metadata Size.*

Figure 8: *Effect of switch constraints.*

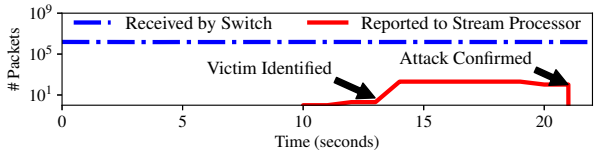


Figure 9: *Detecting Zorro attacks using Tofino switch.*

Tofino switch [48]. Each host has two Intel Xeon E5-2630 v4 10-core processors running at 2.2 Ghz with 128 GB RAM and 10 Gbps NICs. We dedicate two hosts for traffic generation: one sender and one receiver. We assign a third host for the emitter component and a fourth for the remaining runtime, streaming driver, and Spark Streaming [47] components (see Figure 6). The data-plane driver runs on the CPU of the Tofino switch itself. The sender connects to the Tofino switch with two interfaces: one interface to replay CAIDA traces using the Moongen [12] traffic generator at about 1.5 Mpps and another to send attack traffic using Scapy [50]. If we were processing packets at Tofino's maximum rate of 6.5 Tbps, our setup would only need to replace the single instance of Spark Streaming with a cluster that supports the expected data rate.

The attacker starts sending similar-sized telnet packets to a single host (`99.7.0.25`) at time $t = 10$ s. Figure 9 shows the number of packets: (1) received by the switch, and (2) reported to the stream processor on a log scale. Sonata reports only two packet tuples, out of 1.5M pps, to the stream processor to detect the victim in three seconds using two refinement levels: $* \rightarrow 24$ and $24 \rightarrow 32$. At $t = 13$ s, the stream processor starts processing the payload of all telnet packets destined for the victim host, which is only around 100 pps. The attacker gains shell access at $t = 20$ s and sends five packets with the keyword "zorro" in it. Sonata detects the attack at $t = 21$ s, demonstrating its ability to perform real-time telemetry using state-of-the-art hardware switches.

7 RELATED WORK

Network telemetry. Existing telemetry systems that process all packets at the stream processor such as Chimera [7], Gigascope [10], OpenSOC [40], and NetQRE [57] can express a range of queries but can only support lower packet rates because the stream processor ultimately processes all results. These systems also require deploying and configuring a collection infrastructure to capture packets from the data plane for analysis, incurring significant bandwidth overhead. These systems can benefit from horizontally scalable stream processors such as Spark Streaming [58] and Flink [38], but they also face scaling limitations due to packet parsing and cluster coordination [41].

Everflow [59], UnivMon [26], OpenSketch [55], and Marple [34] rely on programmable switches to execute queries entirely in the data plane. These systems can process queries at line rate but can only support queries that can be implemented on switches. Trumpet [31] and Pathdump [46] offload query processing to end-hosts (VMs in data center networks) but not to switches. Our previous work [16] proposed a telemetry system that partitions a single query across a stream processor and switch, but it only considers switches with fixed-function chipsets, and requires the network operators to specify the refinement and partitioning plans manually. In contrast, Sonata supports programmable switches and employs a sophisticated query planner to automatically partition and refine multiple queries.

Query planning. Database research has explored query planning and optimization extensively [4, 32, 36]. Gigascope performs query partitioning to minimize the data transfer from the capture card to the stream processor [10]. Sensor networks have explored the query partitioning problems that are similar to those that Sonata faces [4, 27, 28, 32, 36, 45]. However, these systems face different optimization problems because they typically involve lower traffic rates and involve special-purpose queries. Path Queries [33] and SNAP [3] facilitate network-wide queries that execute across multiple switches; in contrast, Sonata currently only compiles queries to a single switch, but it addresses a complementary set of problems, such as unifying data-plane and stream-processing platforms to support richer queries and partitioning sets of queries across a data-plane switch and a stream processor.

Query-driven dynamic refinement. Autofocus [13], ProgME [56], and DREAM [29], SCREAM [30], MULTOPS [15], and HHH [23] all iteratively zoom in on traffic of interest. These systems either do not apply to streaming data (e.g., ProgME requires multiple passes over the data [56]), they use a static refinement plan for all queries (e.g., HHH zooms in one bit at a time), or they do not satisfy general

queries on network traffic (e.g., MULTOPS is specifically designed for bandwidth attack detection). These approaches all rely on general-purpose CPUs to process the data-plane output, but none of them permit additional parsing, joining, or aggregation at the stream processor, as Sonata does.

8 CONCLUSION

Ensuring that networks are secure and performant requires continually collecting and analyzing data. Sonata makes it easy to do so, by exposing a familiar, unified query interface to operators and building on advances in both stream processing and programmable switches to implement these queries efficiently. Sonata solves an ILP to compute optimal query plans that use available data-plane resources to minimize the traffic sent by the switch to the stream processor. Our experiments using real traffic workloads show that by making the best use of available data-plane resources, Sonata can reduce traffic rates at the stream processor by several orders of magnitude.

Sonata provides a foundation for much future work. First, we are currently extending Sonata to support telemetry applications such as network-wide heavy hitter detection [18] that require observing traffic at multiple locations. Second, we plan to improve on Sonata's query planning by developing new (1) methods to minimize the amount of packet traces required, (2) heuristics for expediting the computation of query plans, and (3) techniques to make the query planning more robust. Finally, our long-term goal is to use Sonata as a building block for closed-loop reaction to network events, in real time and at scale.

REFERENCES

[1] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. Understanding the Mirai botnet. In *USENIX Security Symposium* (2017).

[2] Apache Thrift API. https://thrift.apache.org/.

[3] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. SNAP: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM* (2016).

[4] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD International Conference on Management of Data* (2015).

[5] Assignment 3, COS 561, Princeton University. https://github.com/Sonata-Princeton/SONATA-DEV/tree/tutorial/sonata/tutorials/Tutorial-1.

[6] Bilge, L., Kirda, E., Kruegel, C., and Balduzzi, M. Exposure: Finding malicious domains using passive DNS analysis. In *USENIX Network and Distributed System Security Symposium* (2011).

[7] Borders, K., Springer, J., and Burnside, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium* (2012).

[8] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (July 2014), 87–95.

[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 *ACM SIGCOMM* (2013).

[10] Cranor, C., Johnson, T., Spatscheck, O., and Shkapenyuk, V. Gigascope: A stream database for network applications. In *ACM SIGMOD International Conference on Management of Data* (2003).

[11] The CAIDA UCSD Anonymized Internet Traces 2016-09. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[12] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. Moongen: A scriptable high-speed packet generator. In *ACM Internet Measurement Conference* (2015).

[13] Estan, C., Savage, S., and Varghese, G. Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM* (2003).

[14] Fan, J., Xu, J., Ammar, M. H., and Moon, S. B. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. *Computer Networks* (2004).

[15] Gil, T. M., and Poletto, M. MULTOPS: A data-structure for bandwidth attack detection. In *USENIX Security Symposium* (2001).

[16] Gupta, A., Birkner, R., Canini, M., Feamster, N., MacStoker, C., and Willinger, W. Network Monitoring as a Streaming Analytics Problem. In *ACM HotNets* (2016).

[17] Gurobi Solver. http://www.gurobi.com/.

[18] Harrison, R., Qizhe, C., Gupta, A., and Rexford, J. Network-Wide Heavy Hitter Detection with Commodity Switches. In *ACM Symposium on SDN Research (SOSR)* (2018).

[19] Hira, M., and Wobker, L. J. Improving Network Monitoring and Management with Programmable Data Planes. Blog posting, http://p4.org/p4/inband-network-telemetry/, September 2015.

[20] Izzard, M. The Programmable Switch Chip Consigns Legacy Fixed-Function Chips to the History Books. https://goo.gl/JKWnQc, September 2016.

[21] Javed, M., and Paxson, V. Detecting stealthy, distributed SSH bruteforcing. In *ACM SIGSAC Conference on Computer & Communications Security* (2013), pp. 85–96.

[22] Jose, L., Yan, L., Varghese, G., and McKeown, N. Compiling packet programs to reconfigurable switches. In *USENIX NSDI* (2015).

[23] Jose, L., Yu, M., and Rexford, J. Online measurement of large traffic aggregates on commodity switches. In *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (March 2011).

[24] Jung, J., Paxson, V., Berger, A. W., and Balakrishnan, H. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy* (2004), IEEE, pp. 211–225.

[25] Kührer, M., Hupperich, T., Rossow, C., and Holz, T. Exit from hell? Reducing the impact of amplification DDoS attacks. In *USENIX Security Symposium* (2014).

[26] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM* (2016).

[27] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. TAG: A Tiny Aggregation Service for Ad-hoc Sensor Networks. In *USENIX OSDI* (2002).

[28] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transaction on Database System 30*, 1 (2005).

[29] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Dream: Dynamic resource allocation for software-defined measurement. *ACM SIGCOMM* (2015).

[30] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Scream: Sketch resource allocation for software-defined measurement. In *ACM CoNEXT* (2015).

[31] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM* (2016).

[32] Mullin, J. K. Optimal Semijoins for Distributed Database Systems. *IEEE Transactions on Software Engineering 16*, 5 (1990).

[33] Narayana, S., Arashloo, M. T., Rexford, J., and Walker, D. Compiling path queries. In *USENIX NSDI* (2016).

[34] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM* (2017).

[35] Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., and Rossow, C. IoTPOT: Analysing the rise of IoT compromises. In *USENIX Workshop on Offensive Technology* (2015).

[36] Polychroniou, O., Sen, R., and Ross, K. A. Track join: Distributed joins with minimal network traffic. In *ACM SIGMOD International Conference on Management of Data* (2014).

[37] An update on the Memcached/Redis benchmark. http://oldblog.antirez.com/post/update-on-memcached-redis-benchmark.html.

[38] Apache Flink. http://flink.apache.org/.

[39] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[40] OpenSOC. http://opensoc.github.io/.

[41] OpenSOC Scalability. https://goo.gl/CX2jWr.

[42] The Bro Network Security Monitor. https://www.bro.org/.

[43] The CAIDA Anonymized Internet Traces 2016 Dataset. https://www.caida.org/data/passive/passive_2016_dataset.xml.

[44] Slowloris HTTP DoS. https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris, June 2009.

[45] Srivastava, U., Munagala, K., and Widom, J. Operator Placement for In-Network Stream Query Processing. In *Symposium on Principles of Database Systems* (2005).

[46] Tammana, P., Agarwal, R., and Lee, M. Simplifying datacenter network debugging with PathDump. In *USENIX OSDI* (2016).

[47] Apache Spark. http://spark.apache.org/.

[48] Barefoot's Tofino. https://www.barefootnetworks.com/technology/.

[49] P4 software switch. https://github.com/p4lang/behavioral-model.

[50] Scapy: Python-based interactive packet manipulation program. https://github.com/secdev/scapy/.

[51] SONATA Github. https://github.com/Sonata-Princeton/SONATA-DEV.

[52] Sonata Queries. https://github.com/sonata-queries/sonata-queries.

[53] Vinnakota, B. P4 with the Netronome Server Networking Platform. https://goo.gl/PKQtC7, May 2016.

[54] Wu, Q., Strassner, J., Farrel, A., and Zhang, L. Network telemetry and big data analysis. *Network Working Group Internet-Draft* (2016 (Expired)).

[55] Yu, M., Jose, L., and Miao, R. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI* (2013).

[56] Yuan, L., Chuah, C.-N., and Mohapatra, P. ProgME: Towards Programmable Network Measurement. In *ACM SIGCOMM* (2007).

[57] Yuan, Y., Lin, D., Mishra, A., Marwaha, S., Alur, R., and Loo, B. T. Quantitative Network Monitoring with NetQRE. In *ACM SIGCOMM* (2017).

[58] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM SOSP* (2013).

[59] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., and Zheng, H. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015).