SONATA: Query-Driven Network Telemetry

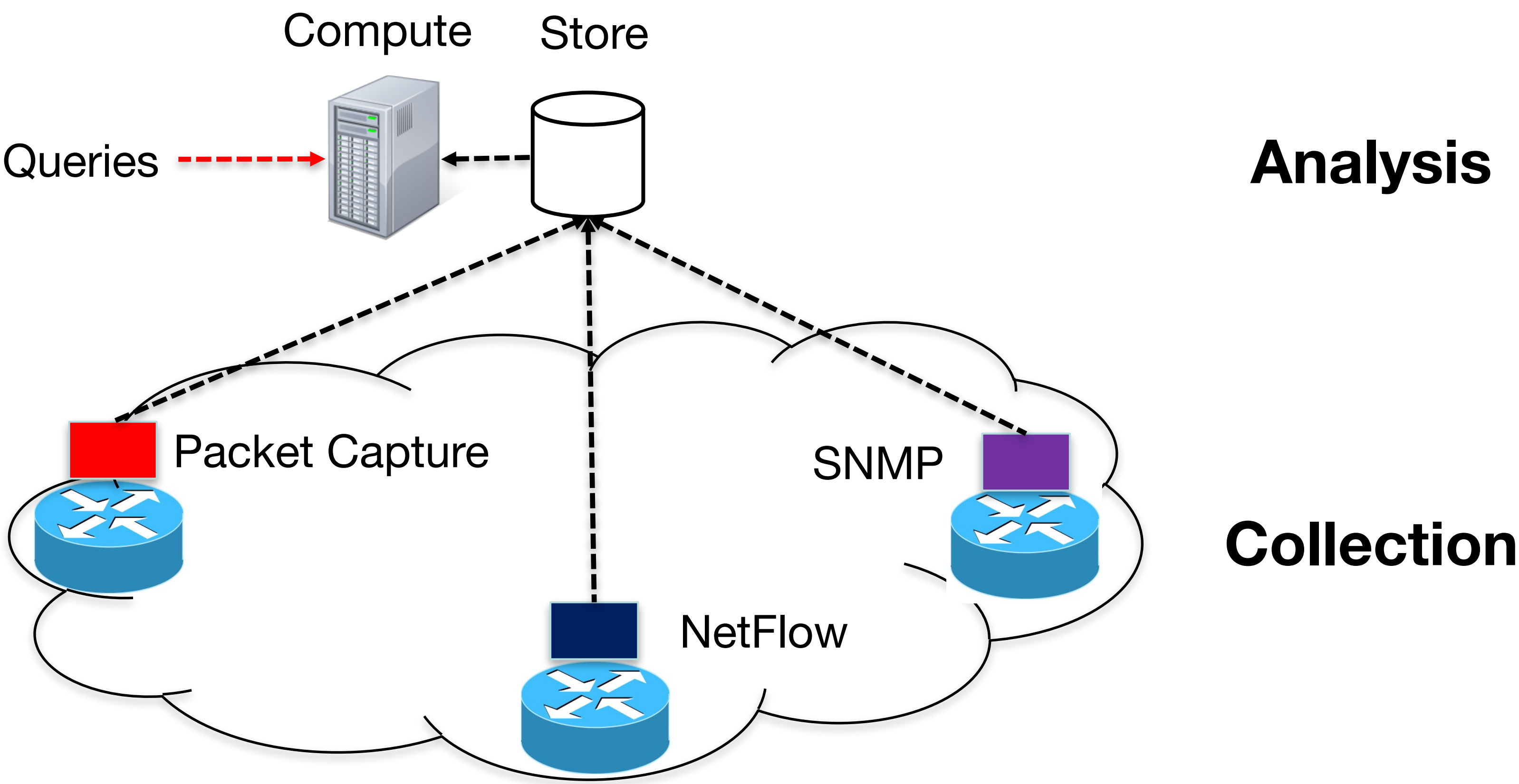
Arpit Gupta

Princeton University

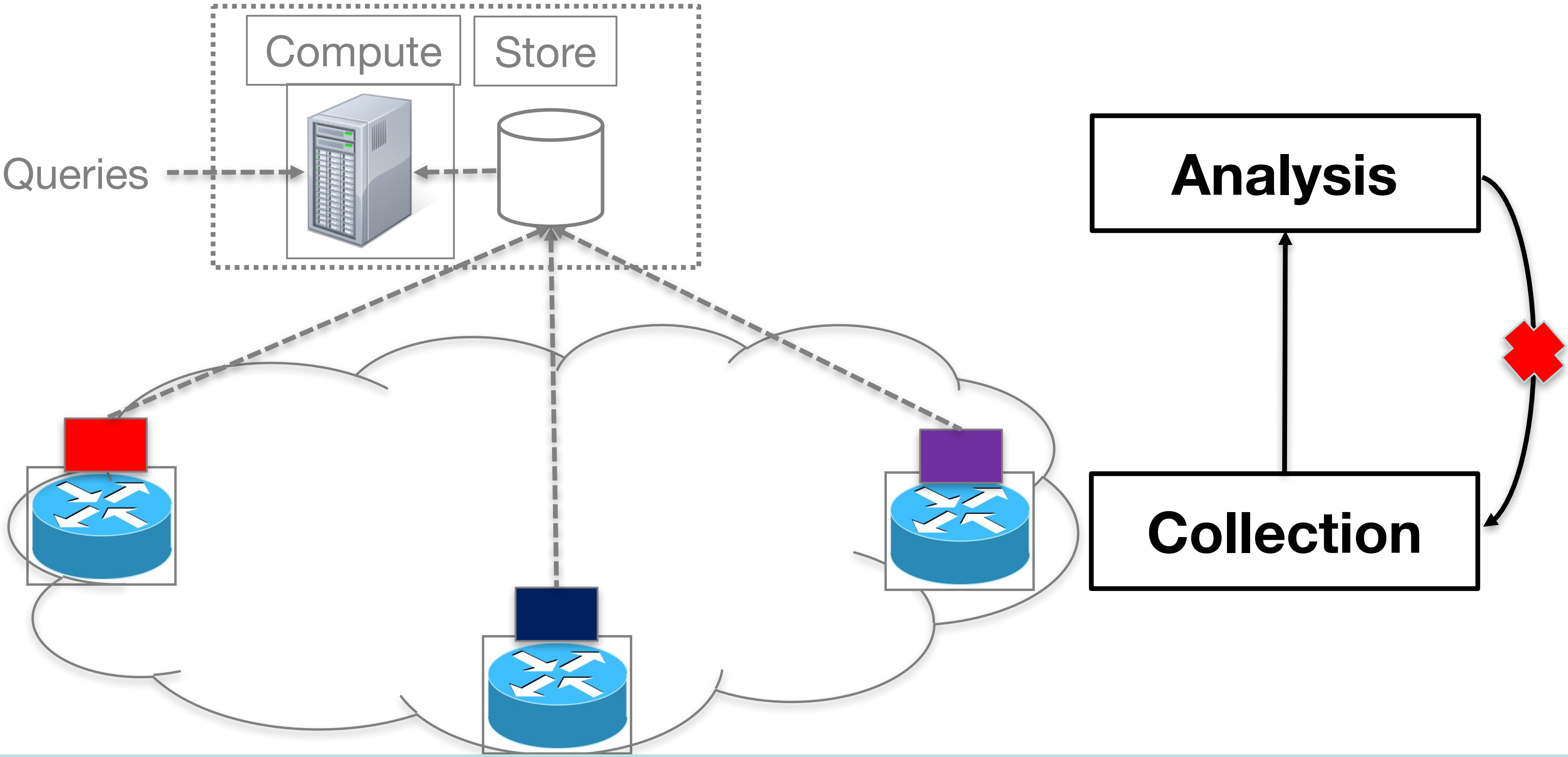
Rob Harrison, *Ankita Pawar, Marco Canini,*

Nick Feamster, Jennifer Rexford, Walter Willinger

Existing Telemetry Systems



Existing Telemetry Systems



Existing Systems are Query-Agnostic!

Problems with Status Quo

- ***Expressiveness***
  - Configure collection & analysis stages separately
  - Static (and often coarse) data collection
  - Brittle analysis setup---specific to collection tools

- ***Scalability***

  Hard to scale query execution as:

  - Traffic Volume increases and/or

Network Telemetry Systems should be **Expressive** & **Scalable**

Idea 1: Declarative Query Interface

- ***Extensible Packet-As-Tuple Abstraction***

  Treat packets as tuples carrying header, payload, and meta fields

- ***Expressive Dataflow Operators***
  – Most telemetry applications
    - Collect aggr. statistics over subset of traffic
    - Join results of one analysis with the other

  – Express them as declarative queries composed of dataflow operators, e.g. **map**, **reduce**, **filter**, **join** etc.

Example Queries

Detecting Newly Opened TCP Connections

Detect hosts for which the number of newly opened TCP connections exceeds threshold (Th)

```
victimIPs = pktStream
            .filter(p => p.tcp.flag == SYN)
            .map(p => (p.dstIP, 1))
            .reduce(keys=(dstIP,), sum)
            .filter((dstIP, count) => count > Th)
            .map((dstIP, count) => dstIP)
```

Collect aggr. stats over subset of traffic

Example Queries

Detecting Traffic Anomalies

Detect hosts for which the number of **unique** source IPs sending DNS response messages exceeds threshold (Th)

```
pvictimIPs = pktStream
             .filter(p => p.udp.sport == 53)
             .map(p => (p.dstIP, p.srcIP))
             .distinct()
             .map((dstIP, srcIP) => (dstIP, 1))
```

Apply multiple aggregations over the packet tuple streams

Example Queries

Confirming Reflection Attacks

Detect hosts with **traffic anomalies** that are of type RRSIG

```
victimIPs = pktStream
            .filter(p => p.udp.sport == 53)
            .join(pVictimIPs, key='dstIP')
            .filter(p => p.dns.rr.type == RRSIG)
            .map(p => (p.dstIP, 1))
            .reduce(keys=(dstIP,), sum)
            .filter((dstIP, count) => count > T2)
```

Join results of one analysis with the other

Changing Status Quo

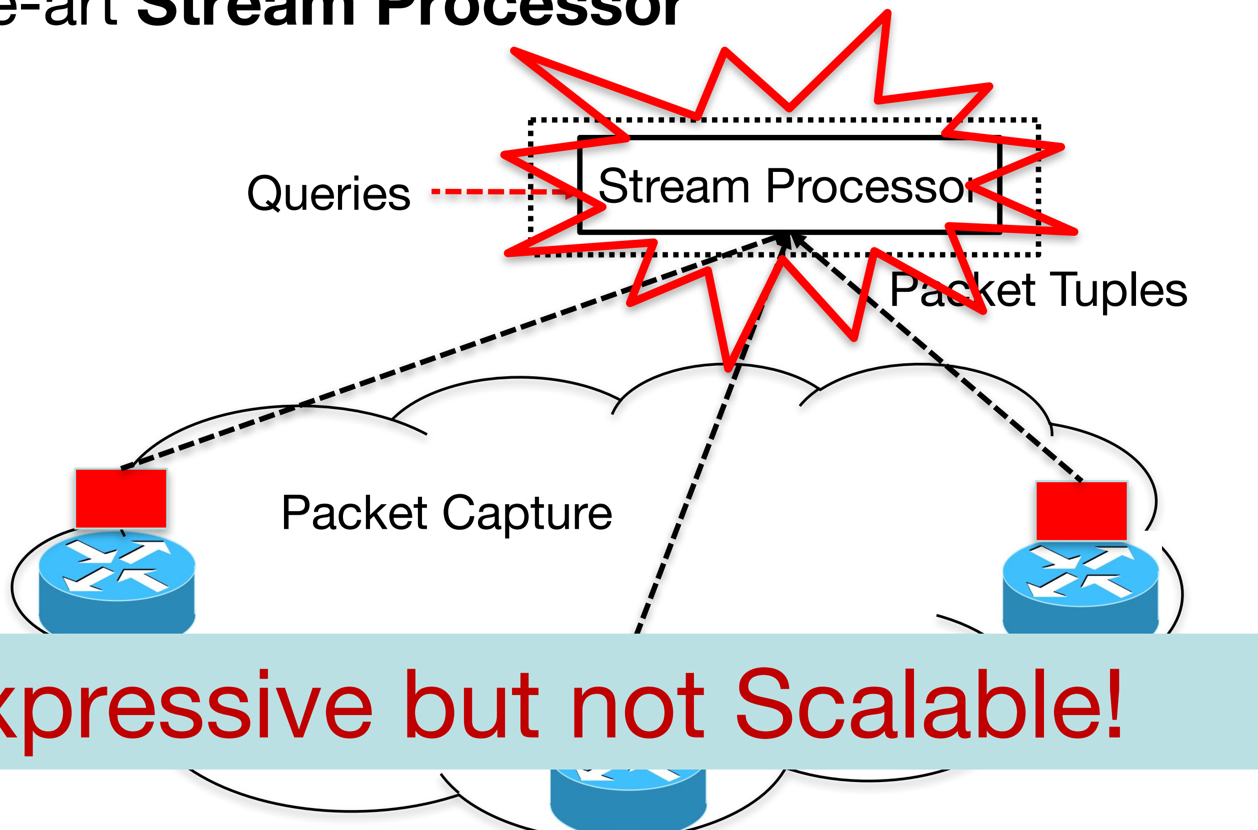
- ***Expressiveness***
  - Express dataflow queries over packet tuples
  - Not tied to low-level (3rd party/platform-specific) APIs
  - Trivial to add new queries and change collection tools

Easier to express network telemetry tasks!

Query Execution

Use Scalable Stream Processors

Process all (or subset of) captured packet tuples using state-of-the-art **Stream Processor**



Idea 2: Query Partitioning

- ***Observation***

  Data plane can process packets at line rate

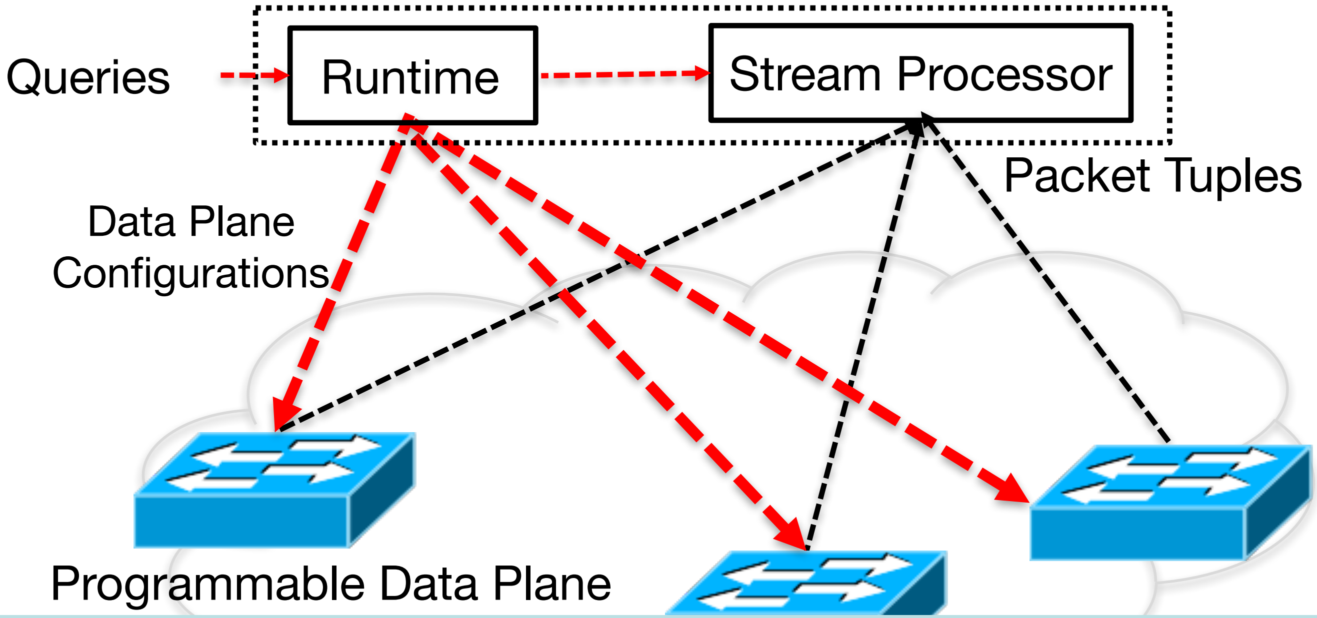
- ***How it works?***

  Execute subset of dataflow operators in the data plane

- ***Trade-off***

  Trades workload at stream processor at the cost of additional resource usage in the data plane

Query Partitioning in Action



Partition Queries b/w
Switches and Stream Processor

Query Partitioning in Action

```
pktStream
.filter(p => p.udp.sport == 53)
.map(p => (p.dstIP, p.srcIP))
.distinct()
.map((dstIP, srcIP) => (dstIP, 1))
.reduce(keys=(dstIP,), sum)
.filter((dstIP, count) => count > Th)
.map((dstIP, count) => dstIP)
```

Traffic Anomaly Query

```
pktStream
.filter(p=>p.srcPort==53)
.map(p=>(p.dstIP,p.srcIP))
.distinct()
```

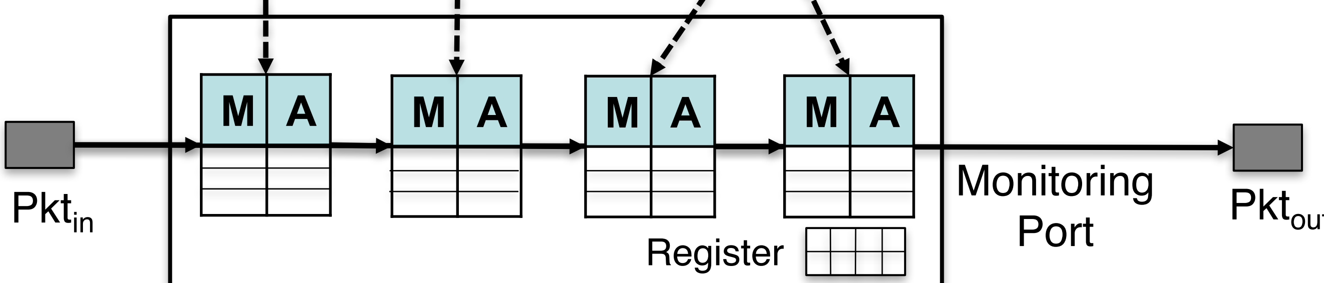
```
.map((dstIP, srcIP)=>(dstIP,1))
.reduce(keys=(dstIP,), sum)
.filter((dstIP,count)=>count>Th)
.map((dstIP, count) => dstIP)
```

Stream Processor

Programmable Data Plane

Compiling Queries for PISA Targets

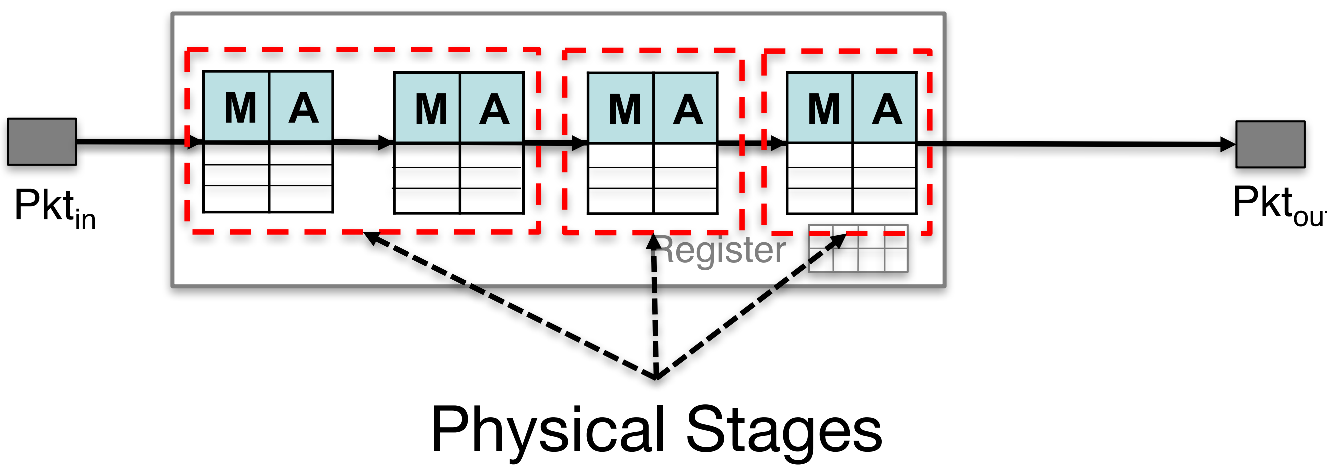
```
pktStream
    .filter(p=>p.udp.sport==53)
          .map(p=>(p.dstIP,p.srcIP))
                .distinct()
```



See Tutorial 2 for details

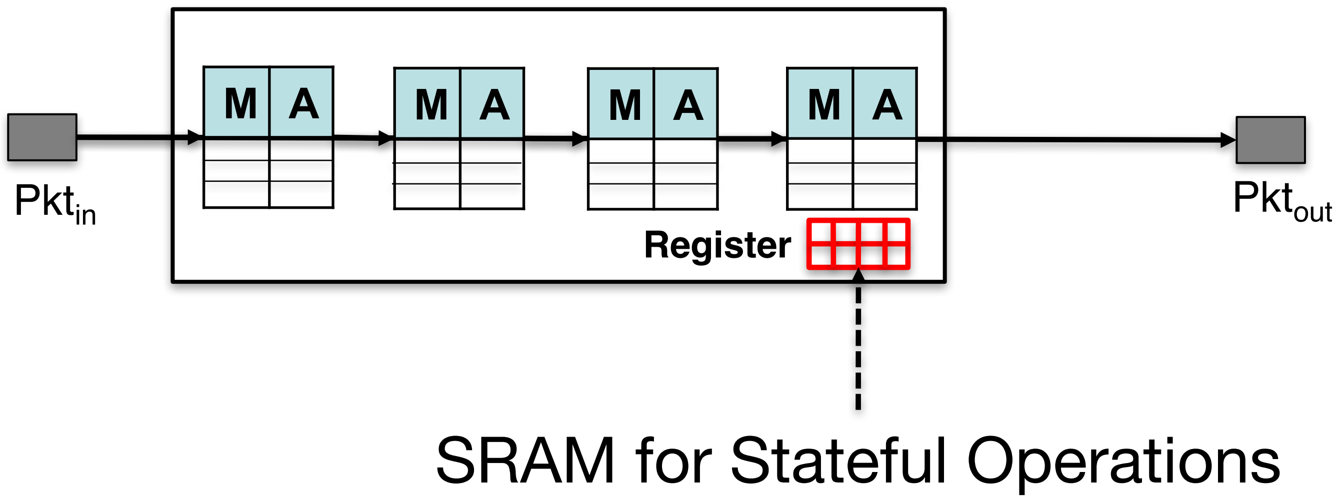
Limited Data-Plane Resources

- *Number of Physical Stages*
- *Number of Actions per Stage*



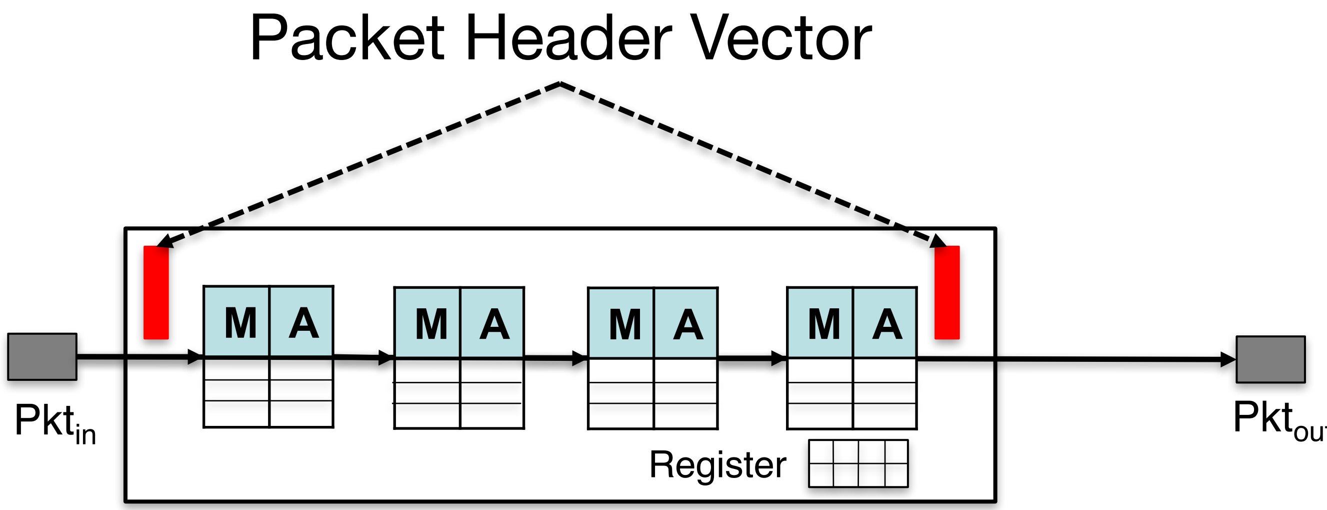
Limited Data-Plane Resources

Available Memory per Stage



Limited Data-Plane Resources

Available State for Metadata fields



Selecting Query (Partitioning) Plans

- ***Given:***

  Queries & Training Data

- ***Objective:***

  Minimize the workload at Stream Processor

- ***Constraints:***
  - Available memory per stage

Solve Query Planning Problem as an ILP

  - Number of actions per stage
  - Total number of stages

Idea 3: Iterative Refinement

- ***Observation***

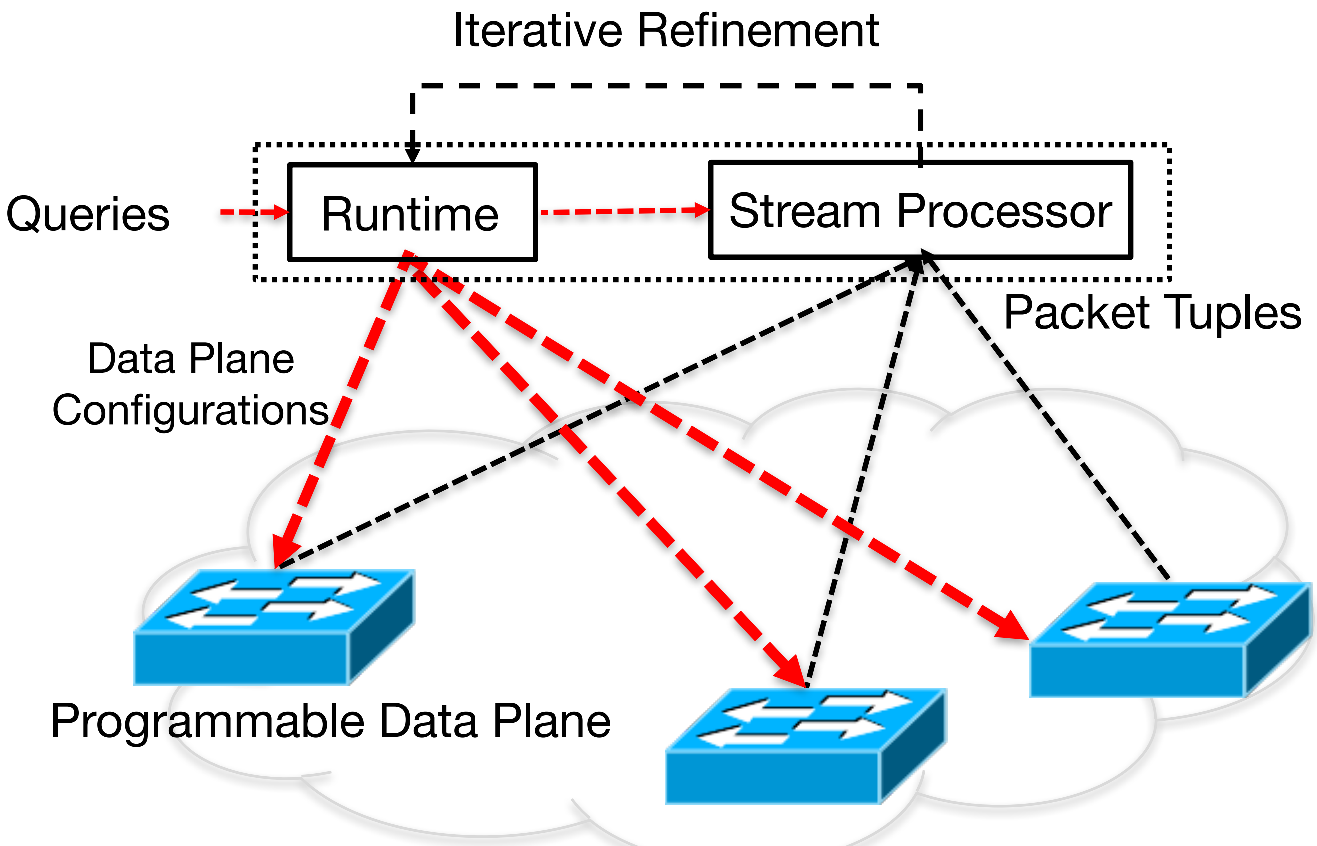
  Tiny fraction of traffic or flows satisfy telemetry queries

- ***How it works?***
  - Execute queries at coarser levels
  - Iteratively zoom-in on interesting traffic over time

- ***Trade-offs***

  Trades workload at stream processor at the cost of additional detection delay

Iterative Refinement in Action



Queries' Output Drives further Processing

Iterative Refinement in Action

Refinement Key = `dstIP`

```
pktStream
.filter(p => p.udp.sport == 53)
.map(p => (p.dstIP, p.srcIP))
.distinct()
.map((dstIP, srcIP) => (dstIP, 1))
.reduce(keys=(dstIP,), sum)
.filter((dstIP, count) => count > Th)
.map((dstIP, count) => dstIP)
```

Traffic Anomaly Query

/8 → /16

```
Q8(W) = pktStream
    .filter(p=>p.udp.sport==53)
    .map(dstIP=>dstIP/8)
    .map(p=>(p.dstIP,p.srcIP))
    …
```

```
Q16(W+1) = pktStream
    .filter(p=>p.udp.sport==53)
    .filter(p=>p.dstIP/8 in Q8(W))
    .map(dstIP=>dstIP/16)
    .map(p=>(p.dstIP,p.srcIP))
    …
```

Query-Driven Network Telemetry!

Time

Quantify Performance Gains

- ***Realistic Workload***
  - Anonymized packet traces from a large ISP
  - Processing 20 M packets per second (~100 Gbps)

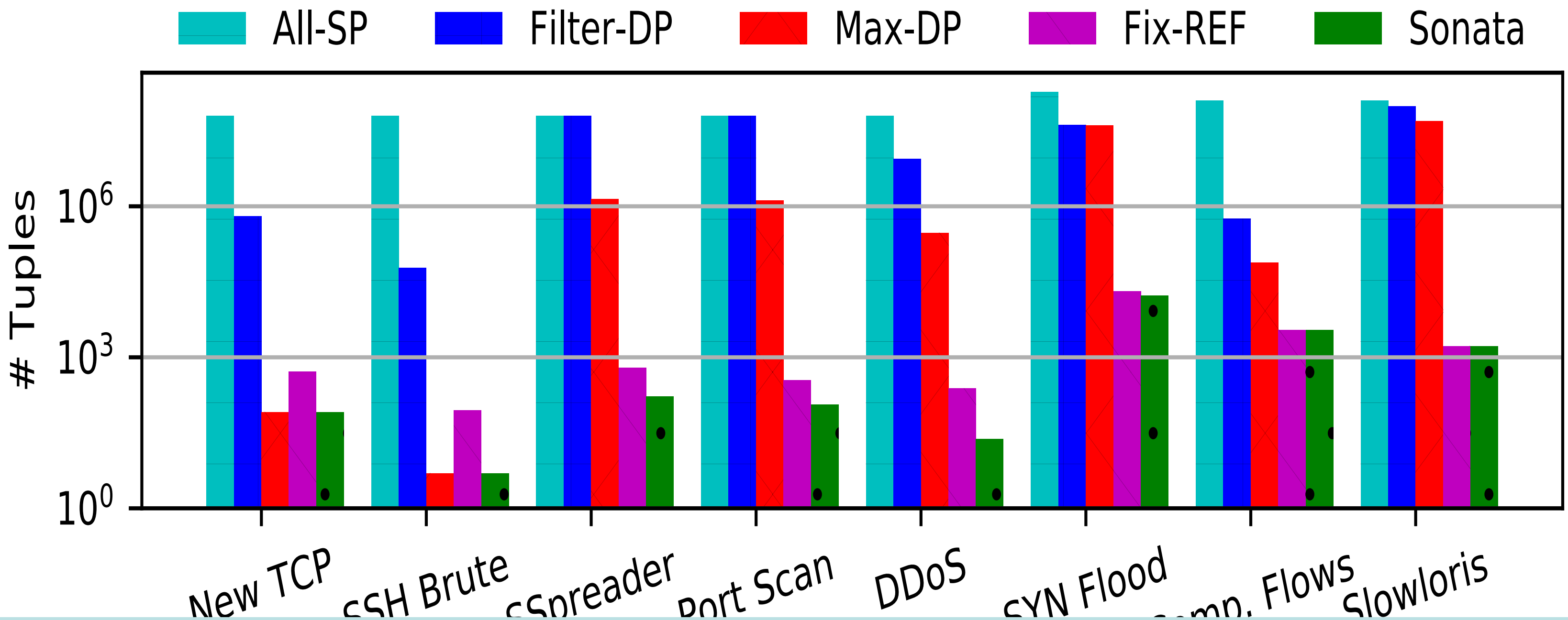
- ***Typical Telemetry Tasks***

  New TCP, SSH Brute, Super Spreader, Port Scan, DDoS, SYN Flood, Completed Flows, Slow Loris, …

- ***Comparisons***

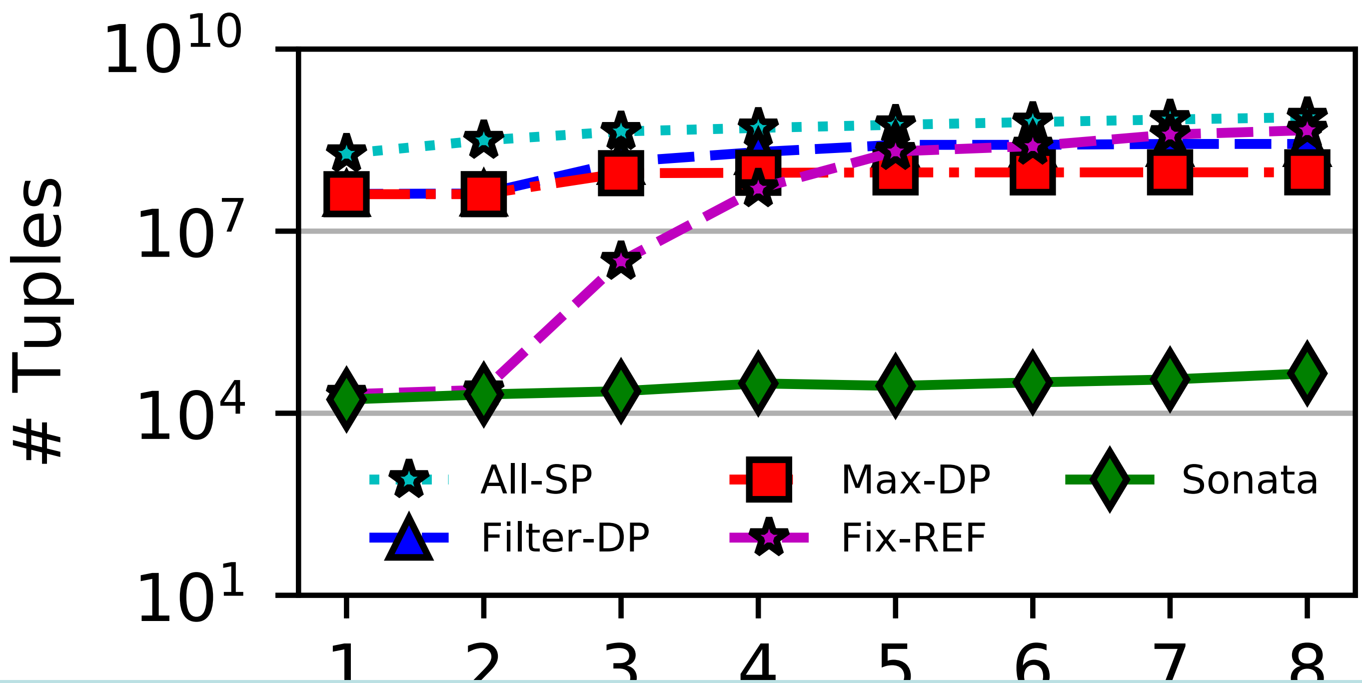
  All-SP, Filter-DP, Max-DP, Fix-REF

Single-Query Performance



Reduces workload at stream processor by up to **seven** orders of magnitude

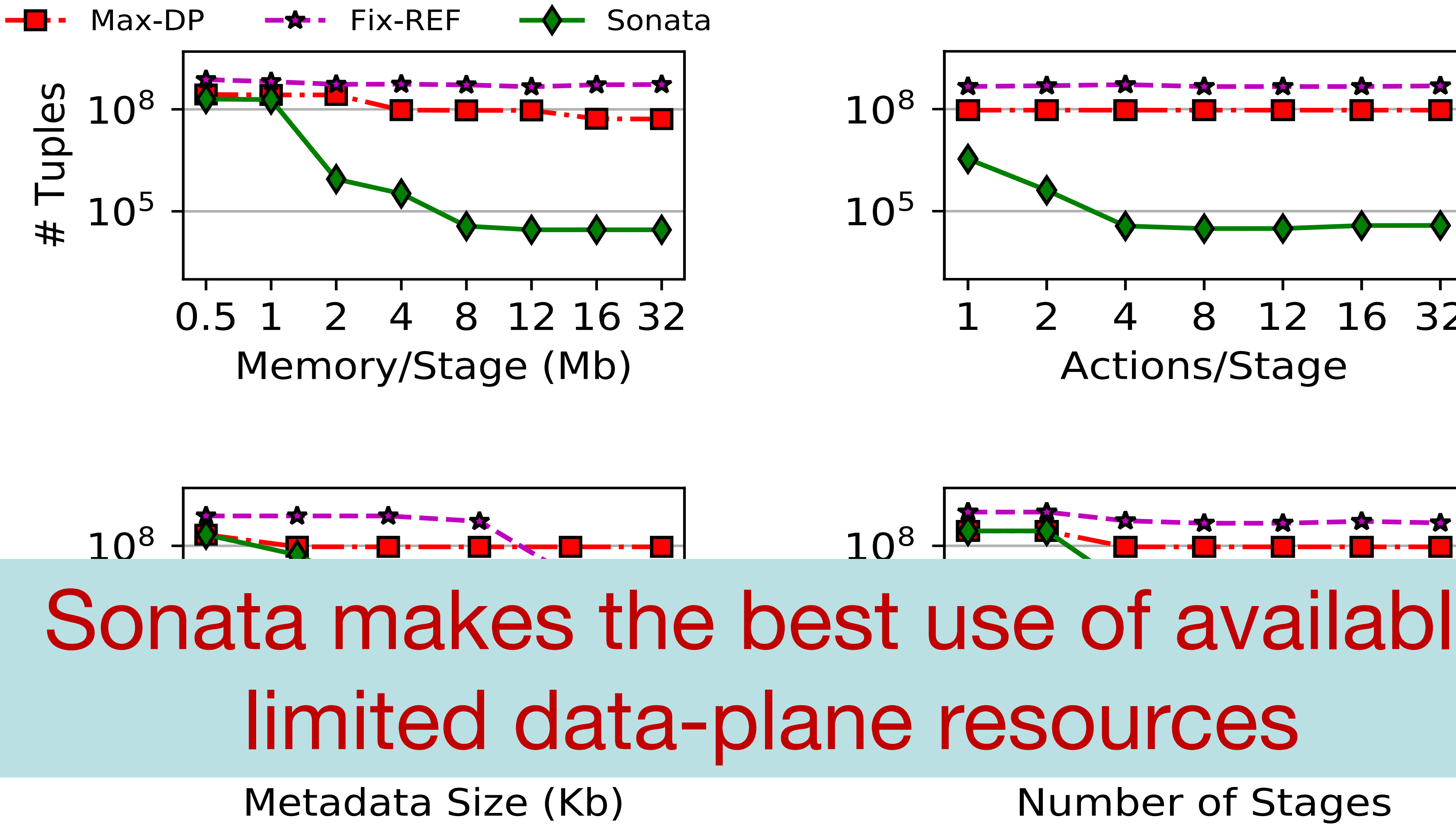
Multi-Query Performance



Reduces workload at stream processor by up to **three** orders of magnitude

Sensitivity Analysis

Data-Plane Resources



Sonata makes the best use of available limited data-plane resources

Changing Status Quo

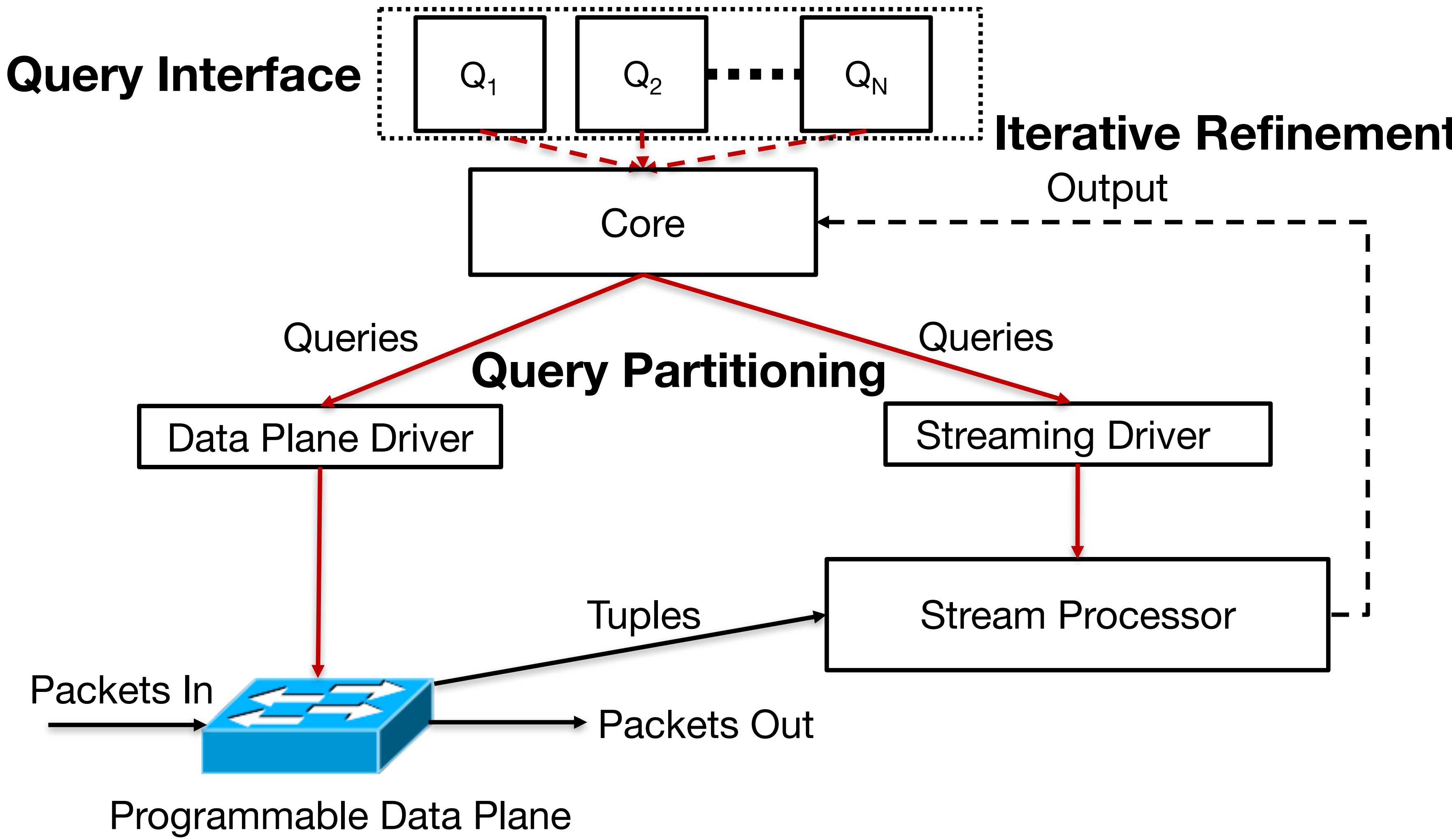
- *Expressiveness*
  - Express Dataflow queries over packet tuples
  - Not worry about how and where the query is executed
  - Adding new queries and collection tools is trivial

- ***Scalability***

  Answers multiple queries for traffic volume as high as 100 Gb/s in real-time

Sonata is **Expressive** & **Scalable**!

Sonata Implementation



More Use Cases

Performance Monitoring

Monitor various performance metrics

```
TCP-Monitoring =
      pktStream
       .map(p => (key, perf-metric))
```

5-tuples,
ingress-egress pairs,
src-dst pairs,
..

nBytes,
loss,
latency,
...

Performance Monitoring

Identify flows for which the traffic volume exceeds threshold (T)

```
Heavy-Hitters =
      pktStream
      .map(p => (p.5-tuple,p.nBytes))
      .reduce(keys=(5-tuple,), sum)
      .filter((5-tuple,bytes) => bytes > T)
      .map((5-tuple,bytes)=> 5-tuple)
```

Use Sonata for Collection & Analysis

Detecting Microbursts

Detect ports for which the total traffic volume exceeds a threshold (T_1)

```
mBursts = pktStream
            .map(p => (p.port, p.nBytes))
            .reduce(keys=(port,), sum)
            .filter((port, bytes) => bytes > T₁)
            .map((port, bytes) => port)
```

Analyzing Microbursts

Analyze which flows contribute to microbursts

```
Top-Contributors = pktStream
      .map(p => (p.port,p.5-tuple,p.nBytes))
      .join(mBursts, key='port')
      .map((port,5-tuple,nBytes)=>(5-tuple,nBytes))
      .reduce(keys=(5-tuples,), sum)
      .filter((5-tuples,bytes) => bytes > T2)
      .map((5-tuples,bytes) => 5-tuples)
```

Future Work

Extend Packet Tuples

```
victimIPs(t) = pktStream(W)
              …
              .filter(p => p.dns.rr.type == RRSIG)
              …
```

- Currently, *dns.rr.type* is parsed in user-space
- Possible to parse it in the data plane itself
- Layers of Interest:
  - DNS
  - SMTP
  - …

Extend Dataflow Operators

- ***Extend existing Operators***
  - Reduce
    - Currently, only *sum* function is supported
    - Implement more complex aggregation functions
  - Join
    - Currently, only *inner join* is supported
    - Implement full outer, Cartesian, left/right inner/outer joins
- ***Add new Operators***
  - Flat Map
  - Sample

Support Network-Wide Queries

- ***Extend Query Interface***
  - Support dataflow queries over **all** packets tuples at **any** location
  - Extract path-related fields, e.g. traversed hops, queue lengths per hop, path latency etc.

- ***Scale Query Execution***
  - Distribute aggregation operations and thresholds along the path
  - Use topological hierarchy for iterative refinement
  - Dynamically route packets for processing

Summary

- SONATA enables **expressive** and **scalable** network telemetry using
  - Declarative Query Interface
  - Query Partitioning
  - Iterative Refinement

- Contribute to Sonata Project
  - 10+ active members and growing
  - GitHub: github.com/Sonata-Princeton/SONATA-DEV

sonata.cs.princeton.edu

Isolating Video Streaming Traffic

Detect hosts that receive DNS response messages from known video streaming services

```
vidFlows = pktStream
        .filter(p => p.udp.sport == 53)
        .map(p => (p.dns.qname, p.dstIP, p.srcIP))
        .join(known_vid_services, key=‘qname’)
        .map(p => (p.dstIP, p.srcIP))
```

*https://github.com/ssundaresan/traffic-analysis

Detecting Bottlenecks

Detect links responsible for performance degradation of video streaming flows

```
BNLinks(t) = pktStream
        .filter(p => p.tcp.sport == 80)
        .map(p => ((p.dstIP, p.srcIP), p.nBytes))
        .join(vidFlows,key=((dstIP, srcIP))
        .reduceByKey(sum)
        .filter(p => p.dataRate> T1)
        .flatmap(p =>(Link(p), 1))
        .reduceByKey(sum)
        .filter(p => p.count > T2)
        .map(p => p.link)
```