

# Traffic Analysis Using Streaming Queries

Mike Fisk  
Los Alamos National Laboratory

*mfisk@lanl.gov*

# Outline

---

- Intro to Continuous Query Systems
  - a.k.a Streaming Databases
  - Relevance to data networks
- Optimizing the evaluation of multiple Boolean queries
  - Counting Algorithm
  - Snort
  - Static Dataflow Optimization
    - Common Subexpression
    - Vector Algorithms
- Performance Comparisons

# Observations

---

- Traffic analysis tools are data-type-specific
  - Flowtools – netflow
  - Snort – pcap
  - Psad – iptables logs
  - ...
- Most analysis systems lack a framework for optimizing rules/queries
  - Reordering boolean expressions
  - Grouping (common sub-expressions)
  - Vector/set operations

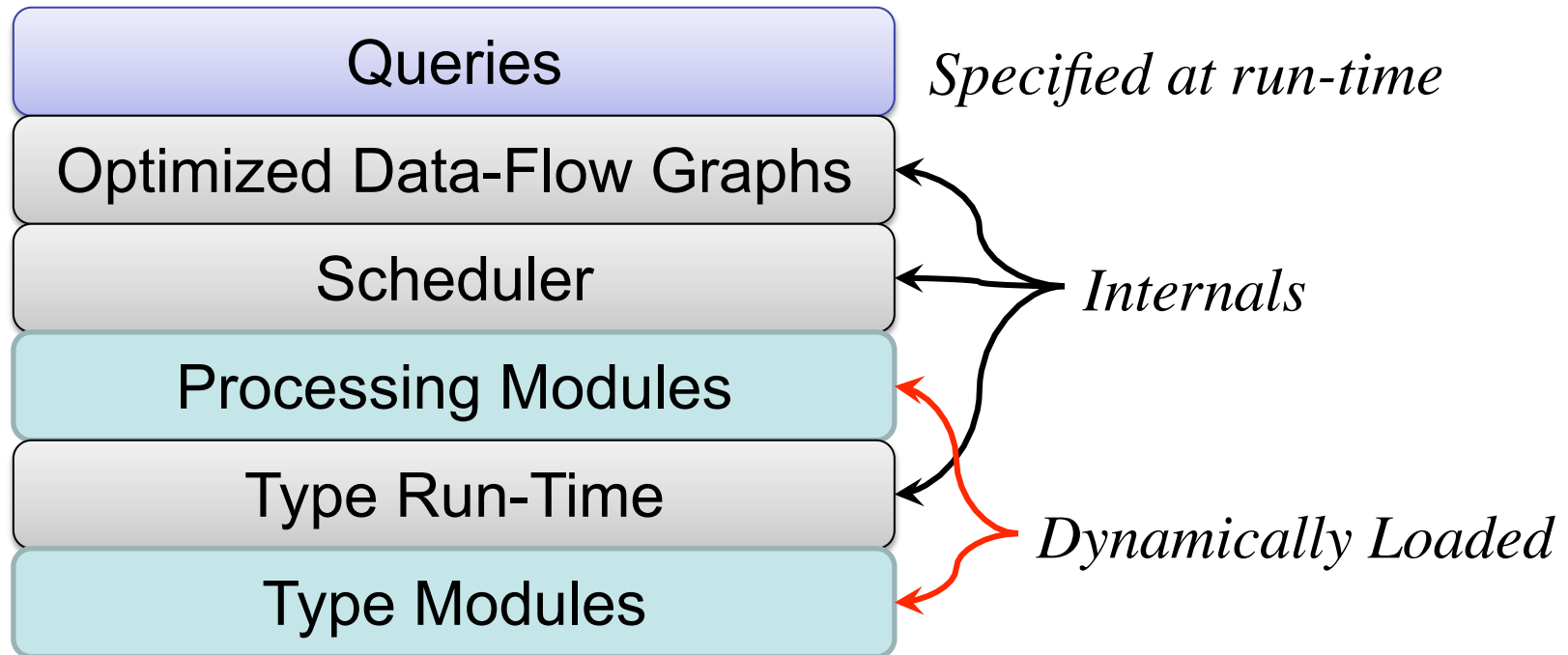
# Continuous Query Systems

---

- Continuous Query systems are to streaming data what Relational Database systems are to stored data
  - Filtering, summarization, aggregation
- Example datasets:
  - Sensor data (temperature, traffic, etc)
  - Stock exchange transactions
  - Packets, flows, logs
- Inefficient and high latency to load data into a traditional database and query periodically.
  - How often could you afford to re-execute the query?
- Example systems:
  - NiagraCQ (Wisc), Telegraph (Berkeley), SMACQ, etc.
  - Commercial: StreamBase, etc.
- Example systems in disguise:
  - Snort, router ACLs, firewall filters, packet classification, egrep

# System for Modular Analysis & Continuous Queries

---



# Type Model

---

- Stream of dynamically & heterogeneously typed objects
  - Each object can have different type
  - Types need not be statically defined in advance
- Objects refer to storage locations
  - Internal to the object, or references into other objects or external memory
- Objects have fields
  - Fields are (indifferently) struct elements, enums, unions, casts, string conversions, etc.
  - Fields are first-class objects
  - Fields can be dynamically attached to objects
- Objects are immutable
  - Enables parallelism without locking

# Type Module Definition

---

- There are no fundamental types
- Pcap packet example

```
struct dts_field_spec dts_type_packet_fields[] = {  
  //Type      Name      Access Function if not fixed  
  { "timeval",  "ts",      NULL }, // Fixed-length, fixed-location  
  { "uint32",   "caplen",  NULL },  
  { "uint32",   "len",      NULL },  
  { "ipproto",  "ipprotocol", dts_pkthdr_get_protocol }, // Function-pointer  
  { "string",   "packet",   dts_pkthdr_get_packet },  
  { "macaddr",  "dstmac",   dts_pkthdr_get_dstmac },  
  { "nuint16",  "ethertype", dts_pkthdr_get_ethertype },  
  { "ip",       "srcip",    dts_pkthdr_get_srcip },  
  ...  
}
```

# SMACQ Processing Modules

---

- Modules are the atoms of query optimization
- Written in C++ or Python
- Take arbitrary flags and arguments
  - Unix command-line style
- Introspection: Can ask runtime to identify downstream invariants
  - When module can do eager pre-filtering (e.g. hardware prefilter on NIC, database query, etc.)
- Event-driven (produce/consume) API
  - Can use “threaded” wrapper if lazy (really co-routines)
- Can embed other query instantiations
  - Can instantiate new scheduler, or share primary (preferred)



# Example Processing Module (Python)

---

Class Dumper:

```
"""Print a few elements of each datum and pass every 5th"""
```

```
def __init__(self, smacq, *args):
```

```
    print ('init', args)
```

```
    self.smacq = smacq    #Save reference to runtime
```

```
    self.buf = []        #List of objects received
```

```
def consume(self, datum):
```

```
    for i in 'srcip', 'dstip', 'ipprotocol', 'len':
```

```
        v = datum[i].value
```

```
        print (i, datum[i].type, type(v), v)
```

```
    self.buf.append(datum)
```

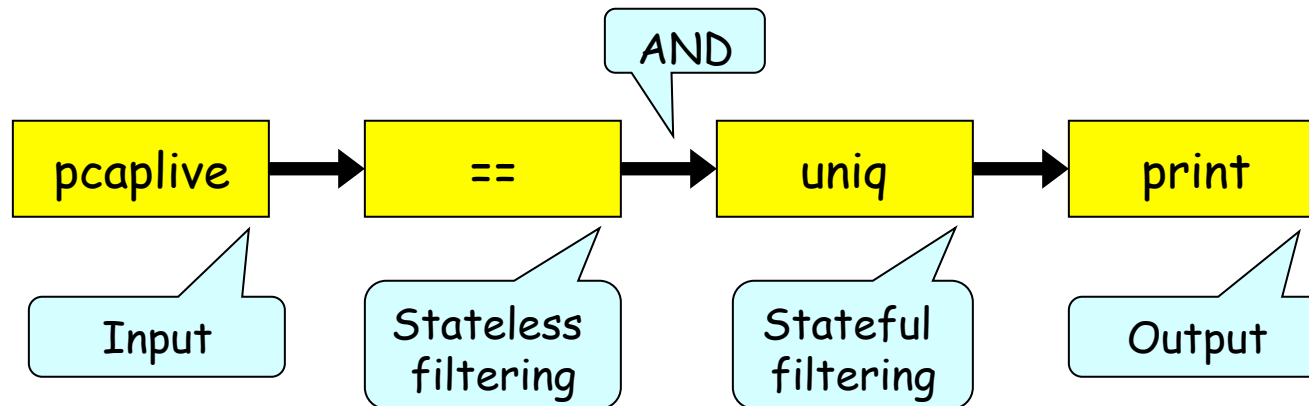
```
    if len(self.buf) == 5:
```

```
        self.smacq.enqueue(datum) # Output object downstream
```

```
        self.buf = []
```

# Query Model: Dataflow Graphs

- Queries are dataflow graphs



- Modules declare algebraic properties:
  - stateless (map), annotation, vector, demux, (associative)*
  - Enables optimization, rewriting, parallelization, map/reduce
- Static optimizer applies all data-flow optimizations permitted by algebraic properties of the involved modules

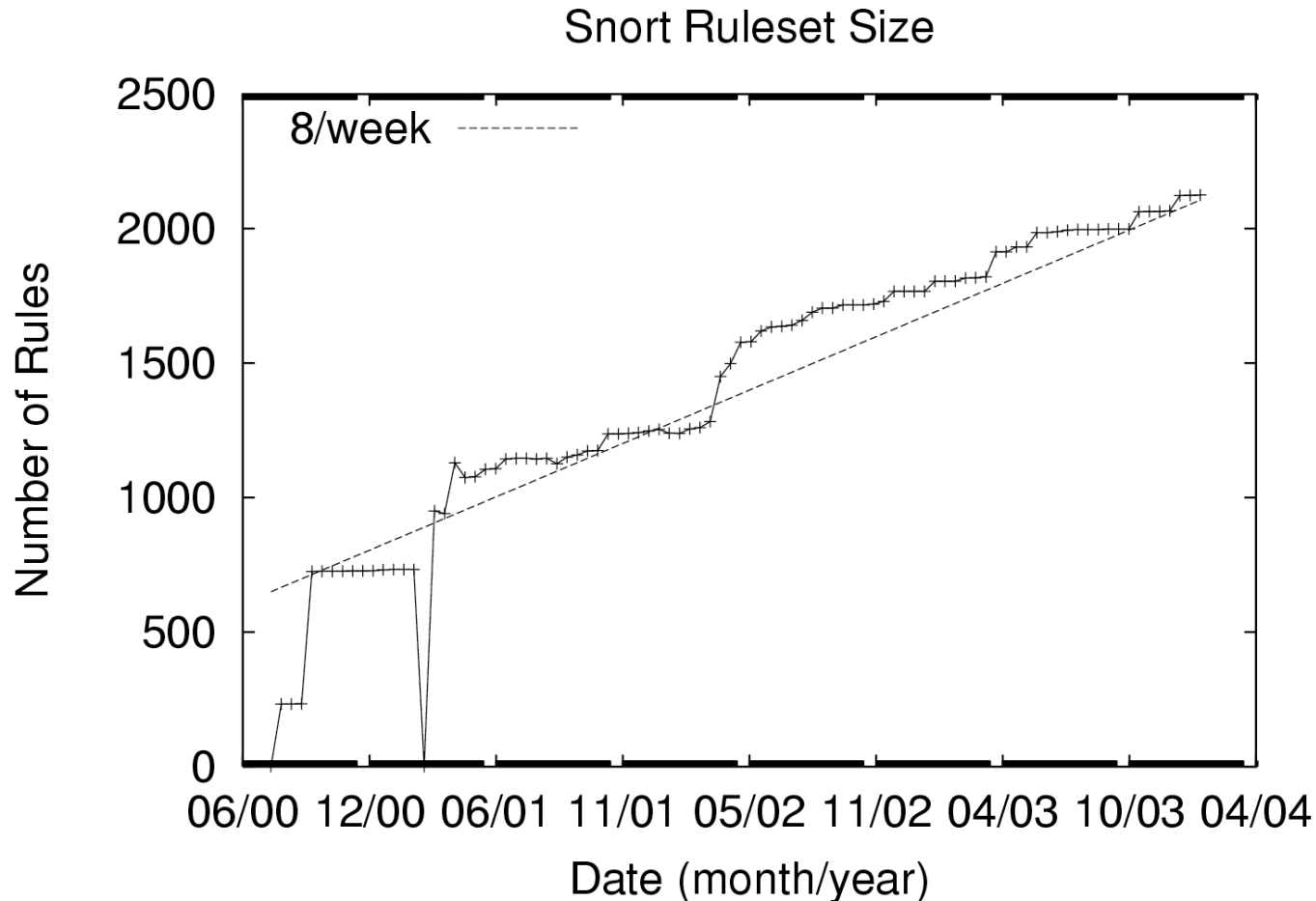
# Optimizing Continuous Queries

---

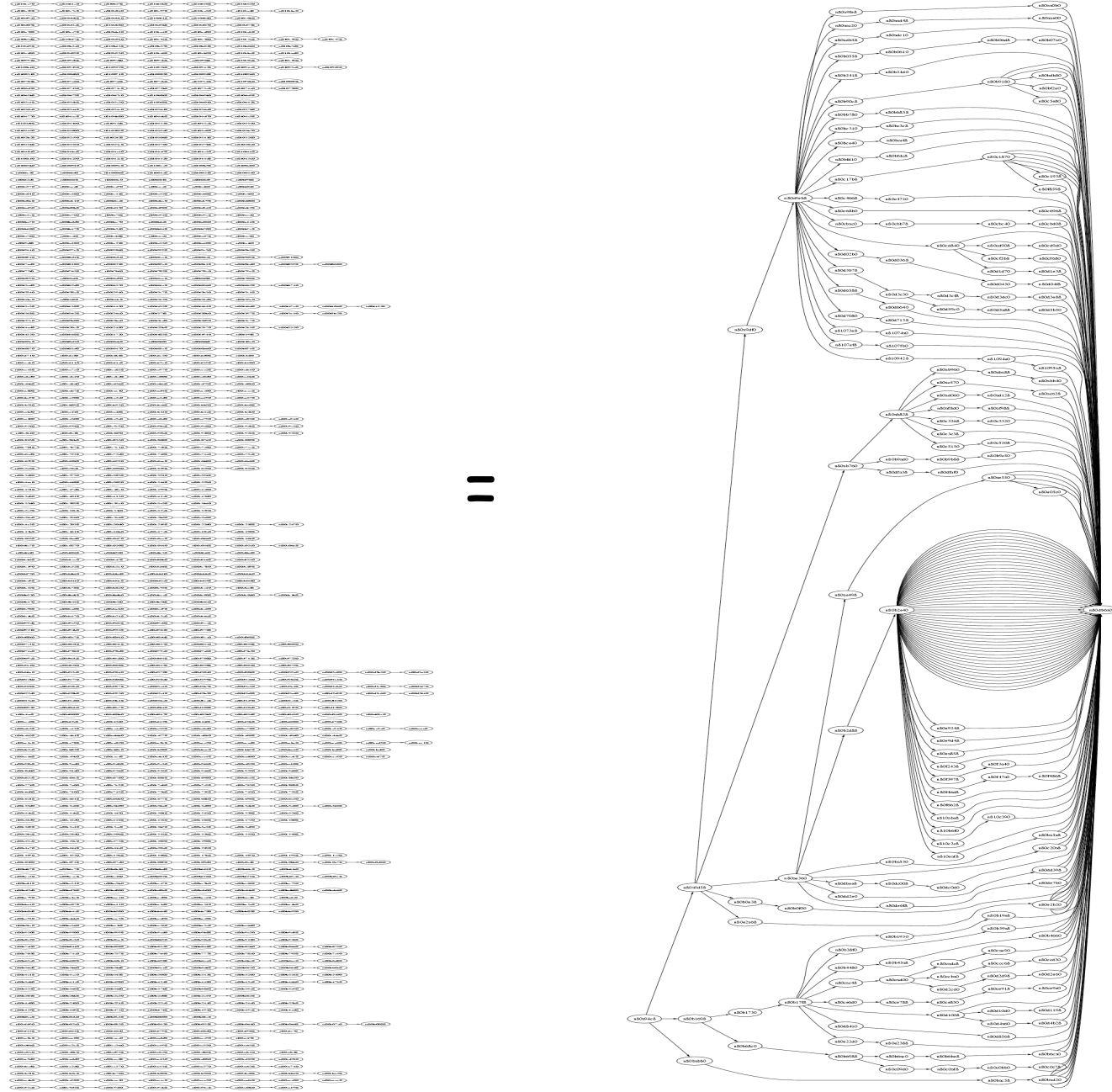
- Traditional database query optimization:
  - Uses data indexes
  - Minimizes individual query times
- Continuous-query optimization:
  - Executing many queries simultaneously
  - Minimize resource consumption per unit of data input
    - Maximize data throughput

# Why is multiple query processing important? Approximately 8 new rules each week

---

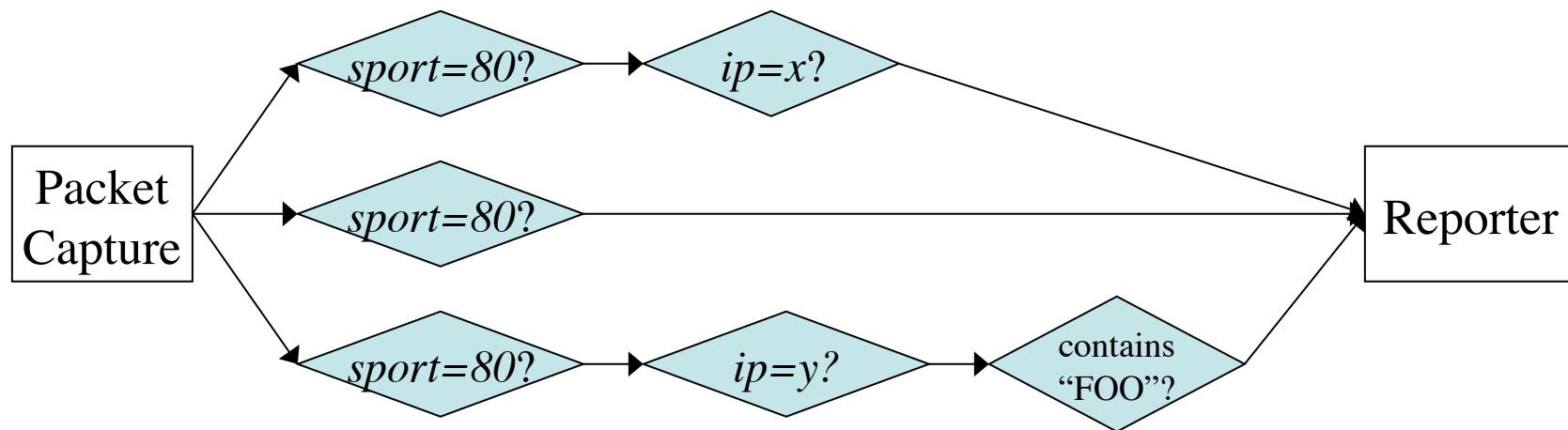


# Optimization of 150 Snort Rules



# Example Queries

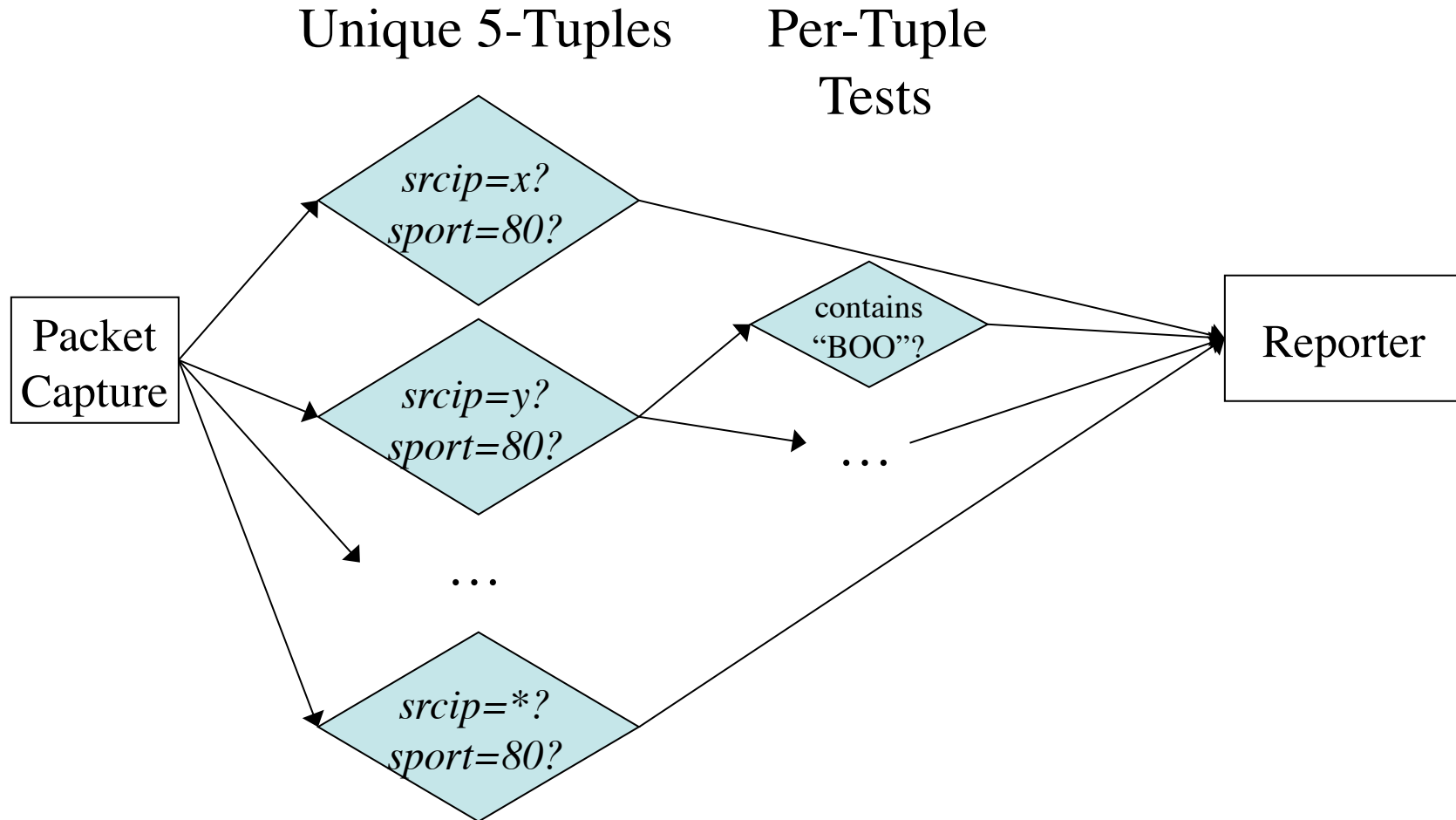
## 6 Tests in 3 Rules



# Snort Approach

[Roesh, LISA 99]

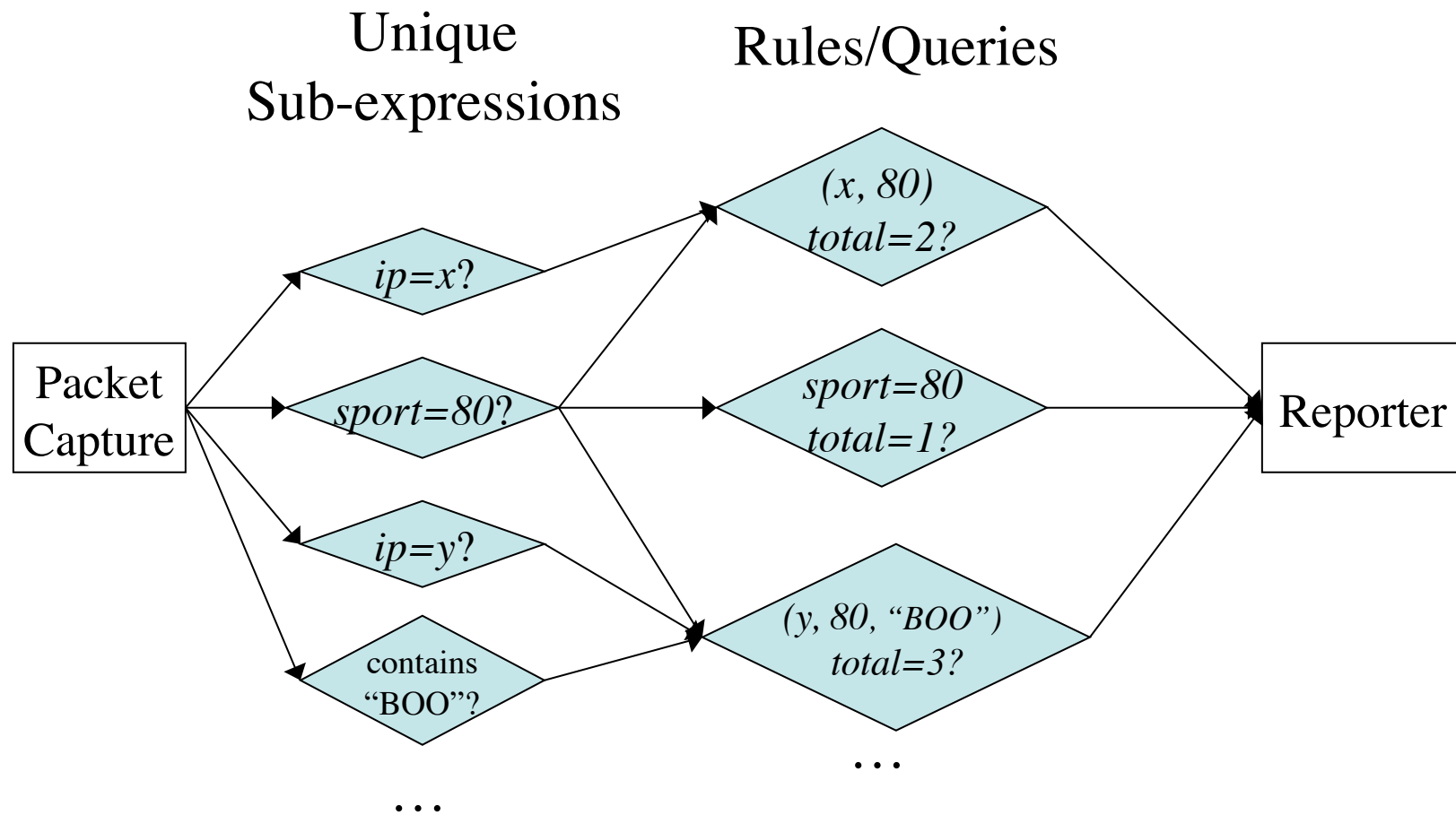
## Example: 6-7 Tests



# Counting Approach

[Carzaniga & Wolf, SIGCOMM 03]

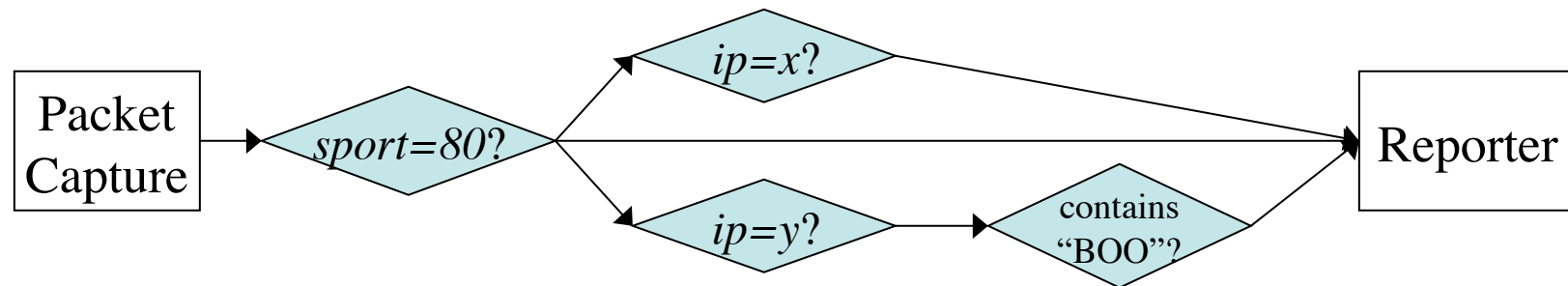
## Example: 7 Tests





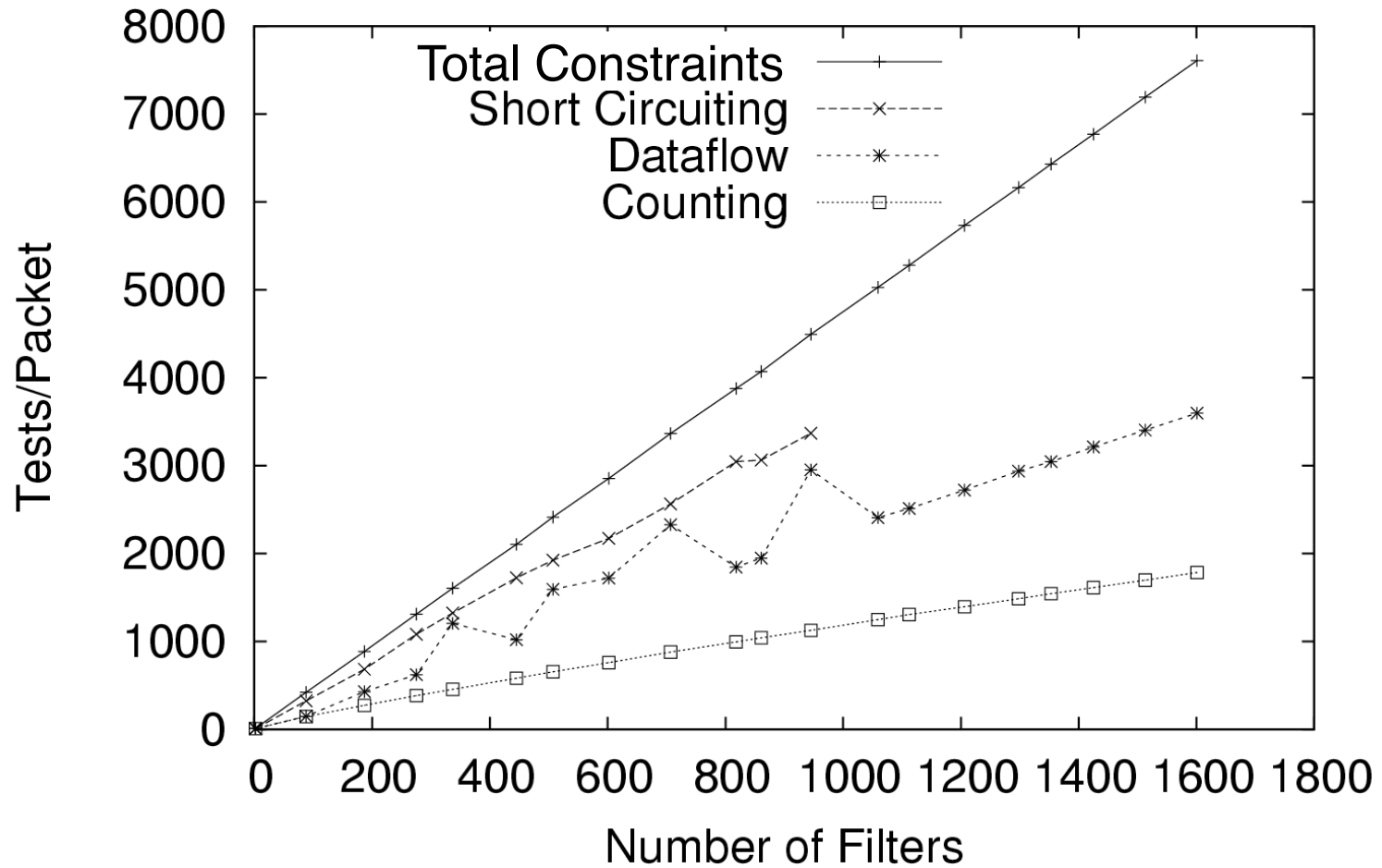
# Data-Flow Approach

## Example: 1-4 Tests



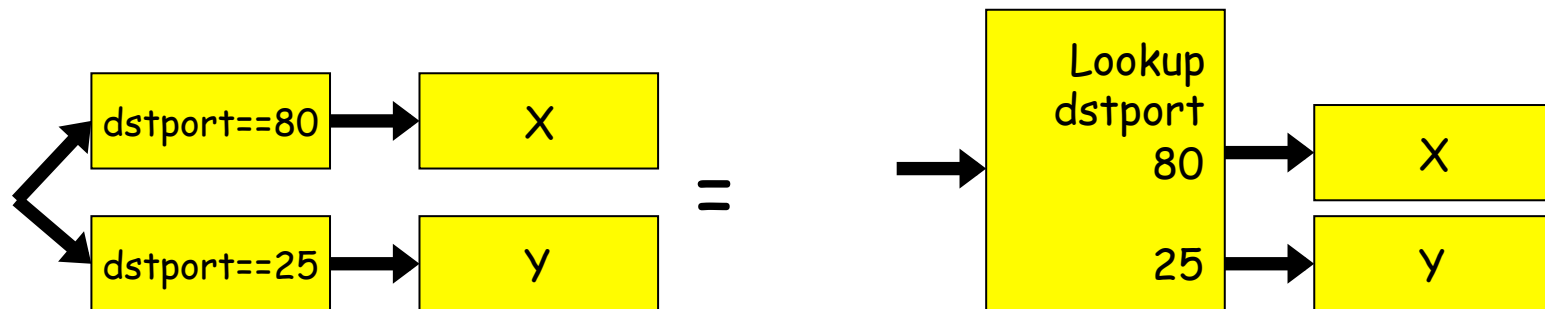
1. Common roots
2. Common leaves
3. Common upstream graphs
4. Common downstream graphs

# Performance Comparison

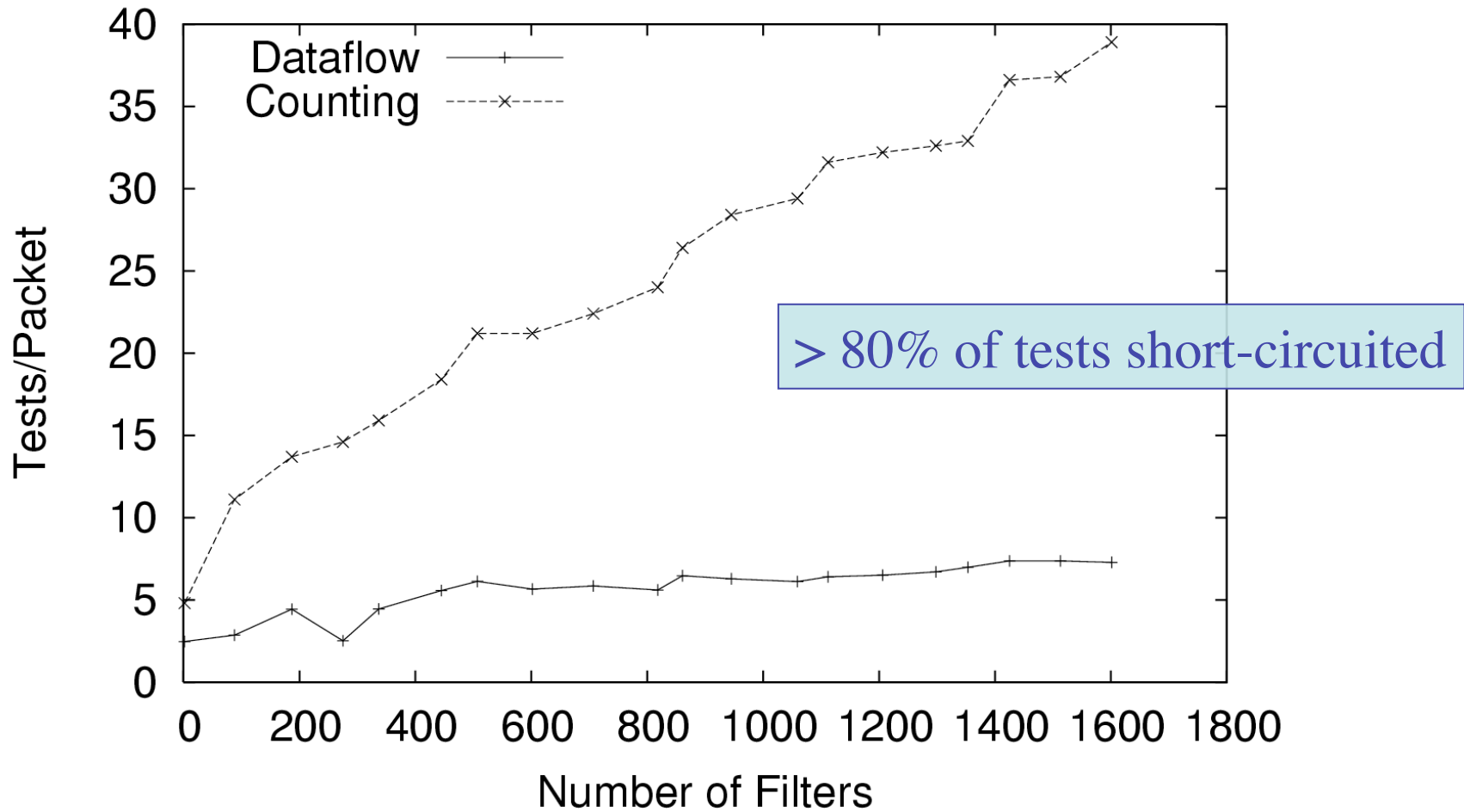


# Vector Functions

- Most optimizations in stream analysis have employed a class of algorithms that can be characterized as *vector functions*:
  - $f(x, v) = f(x, v_1), f(x, v_2), \dots$
  - Vector version is typically  $O(1)$  or  $O(\log n)$  instead of  $O(n)$
- Examples
  - Set of equality tests becomes a single lookup in a hash-table
  - Set of string matches becomes a single DFA to traverse

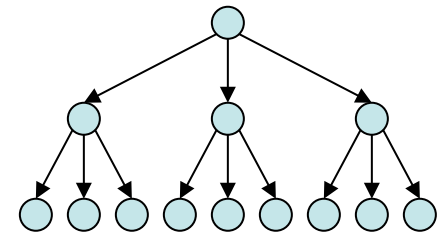


# Performance Comparison with Vector Functions



# Analysis: Why was Counting better only without vectors?

- Assume that each test results in  $p$  more tests
  - $p = \text{fanout} \cdot \text{short-circuiting}$
  - $p \leq \text{fanout}$
  - $0 \leq \text{short-circuiting} \leq 1$
- Assume data-flow of tests is a balanced tree of depth  $d$ 
  - $d$  is an integer  $\geq 1$
- Expected number of evaluations:
$$1 + p + p^2 + p^3 + \dots + p^{d-1} = (1 - p^d) / (1 - p)$$
- Let  $u =$  number of unique tests = Counting's performance
$$s(1 - p^d) / (1 - p) < u \quad \text{if } (d > 1, p < 1)$$
- For IDS test:  $d = 6$ 
  - With Vectors ( $u=39$ ):  $p < 1.7$  is desired. Actual  $p = 1$
  - Without Vectors ( $u=1782$ ):  $p < 4.2$  is desired. Actual  $p = 5.8$



# Supported Query Languages

- SQL style:

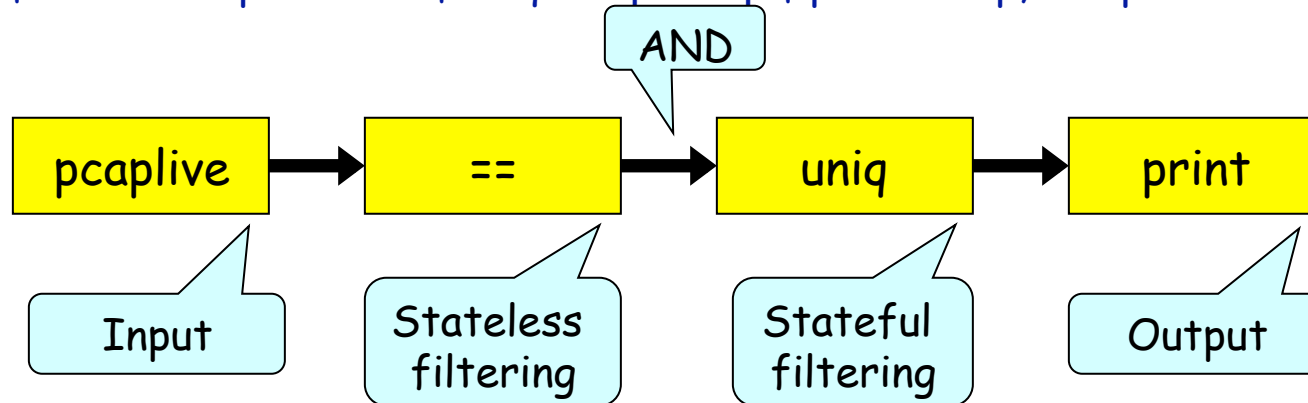
`print srcip, dstip from`

`(cflow where dstport==80 and uniq(srcip, dstip))`

- Misplaced belief that since SQL is well defined, people can just use it
- Deeply nested queries make you wish you were merely nested in s-expressions

- Unix pipe style:

`cflow | where dstport==80 | uniq srcip dstip | print srcip, dstip`



# Supported Query Languages

---

- Datalog

Pairs :- cflow | uniq(srcip dstip)

SrcCount :- count() group by ipprotocol srcport

DstCount :- count() group by ipprotocol dstport

Pdf :- filter(count) | pdf

Print :- sort(-r probability) | print(type ipprotocol port probability)

Pairs | SrcCount | const(-f type src) | Pdf | rename (srcport port) | Print

Pairs | private | DstCount | const(-f type dst) | Pdf | rename (dstport port) | Print

- Clean, allows named subexpressions

# Join Models

---

- DFA module

- Define a state machine where transitions specified as Booleans on new inputs

- SQL style

- Example: print running cross-product

```
print a.ipid b.ipid from
```

```
pcapfile(0325@1112-snort.pcap) a, b where a.ipid != b.ipid
```

- New keyword **UNTIL** defines when state can be removed
  - “**NEW**” refers to newly input data for comparison

- Example: print retransmissions within the same second

```
print expr(b.ts - a.ts) from pcaplive() a until(new.a.ts.sec > a.ts.sec), b until(new)
where b.ts > a.ts and a.srcip == b.srcip and a.srcport == b.srcport
and a.seq == b.seq and a.payload != "" and b.payload != ""
```



# Usage Experience

---

- Online detection & automated response systems
- Ad-hoc queries for forensic analysis and data exploration
- Feature extraction for other software

# Conclusions

---

- Continuous Queries provide a common query syntax, software infrastructure, and optimization framework for traffic analysis
- CQ necessary for streaming applications, sufficient for ad-hoc forensic analysis

Open source at [SMACQ.SF.NET](https://smacq.sf.net)

# Conclusions

Open source at  
**SMACQ.SF.NET**

- Continuous Queries provide a common query syntax, software infrastructure, and optimization framework for traffic analysis
- Two identified strategies for static optimization of multiple queries
  - Remove (Counting) or Reduce (Data-flow) redundant tests
  - Boolean (Data-flow) short-circuiting removes need for some subsequent tests
- Performance Analysis:
  - Counting is preferable when short-circuiting is rare
  - Data-flow out-performs counting when short-circuiting is significant
    - When breadth of graph is reduced with vector functions, actual IDS workload benefits significantly from short-circuiting
- Data-flow approach can also benefit from additional, dynamic reordering of tests to maximize early short-circuiting