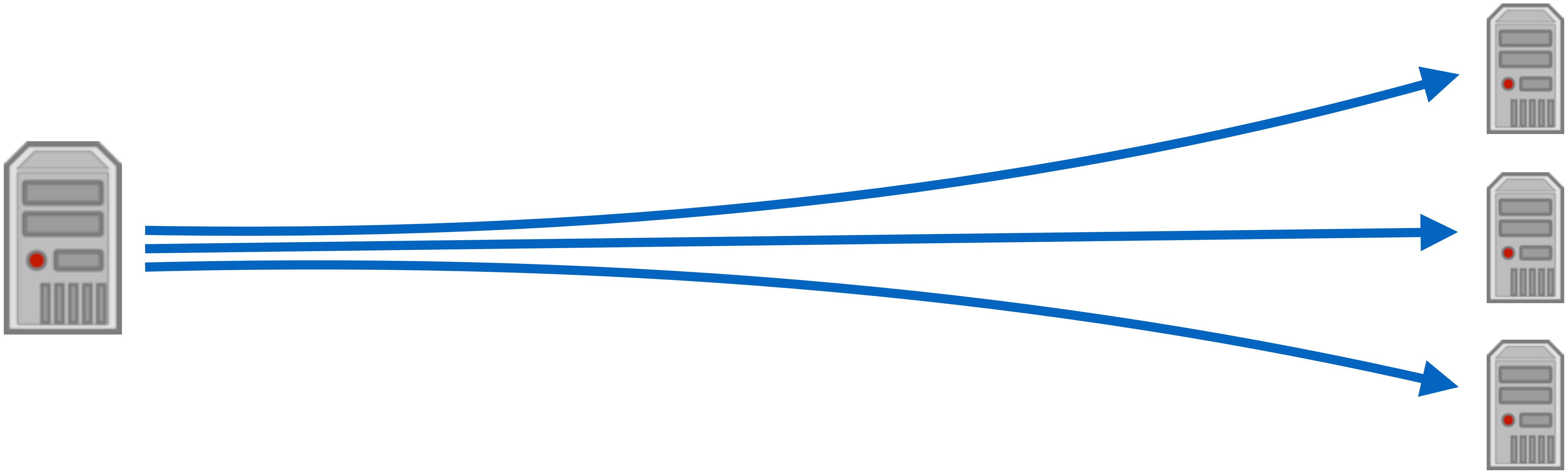
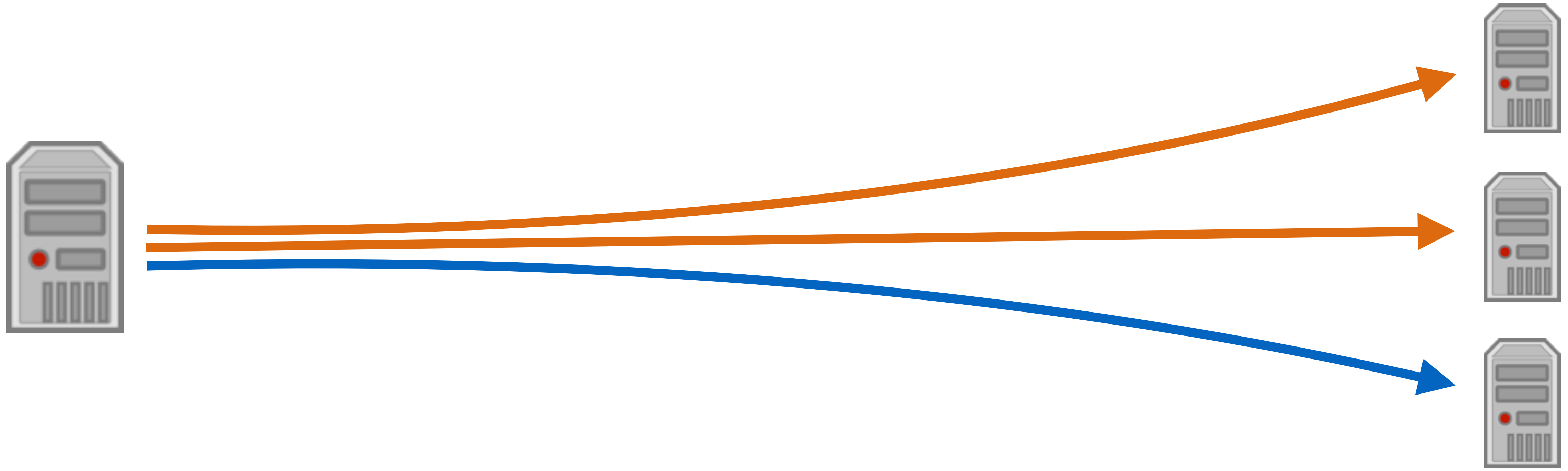
High-Throughput Publish/Subscribe in the Forwarding Plane

Theo Jepsen, Masoud Moushref, Antonio Carzaniga,
Nate Foster, Xiaozhou Li, Milad Sharif, Robert Soulé
Università della Svizzera italiana (USI) and Barefoot Networks

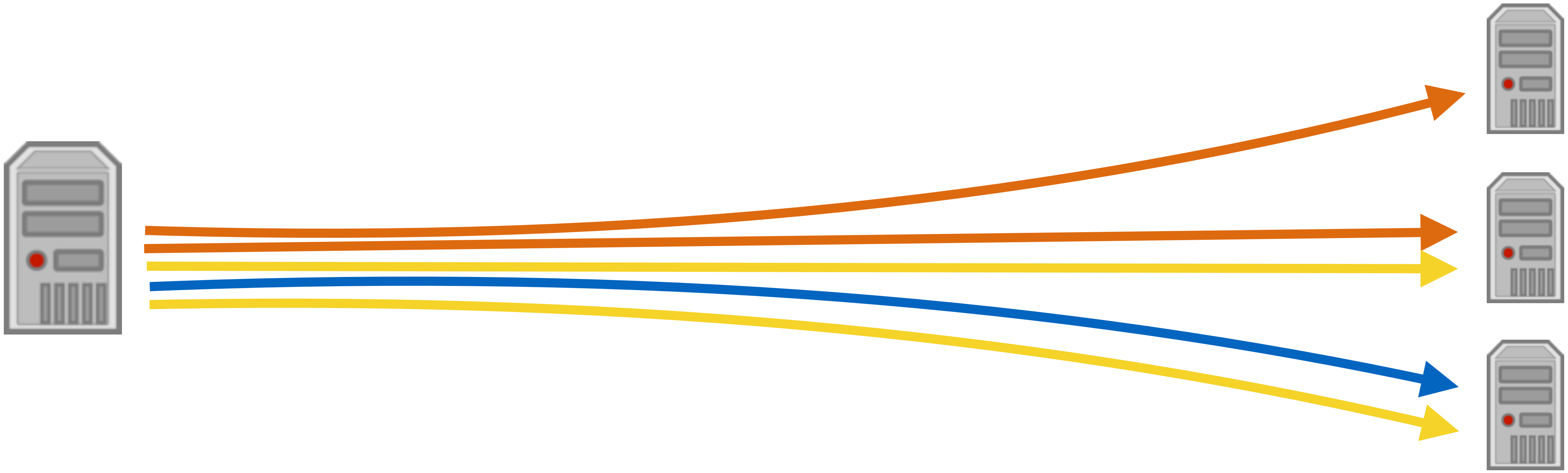
Publish/Subscribe Is Critical to Distributed Applications



Publish/Subscribe Is Critical to Distributed Applications



Publish/Subscribe Is Critical to Distributed Applications



Publish/Subscribe Is Critical to Distributed Applications

Publish/Subscribe Is Critical to Distributed Applications

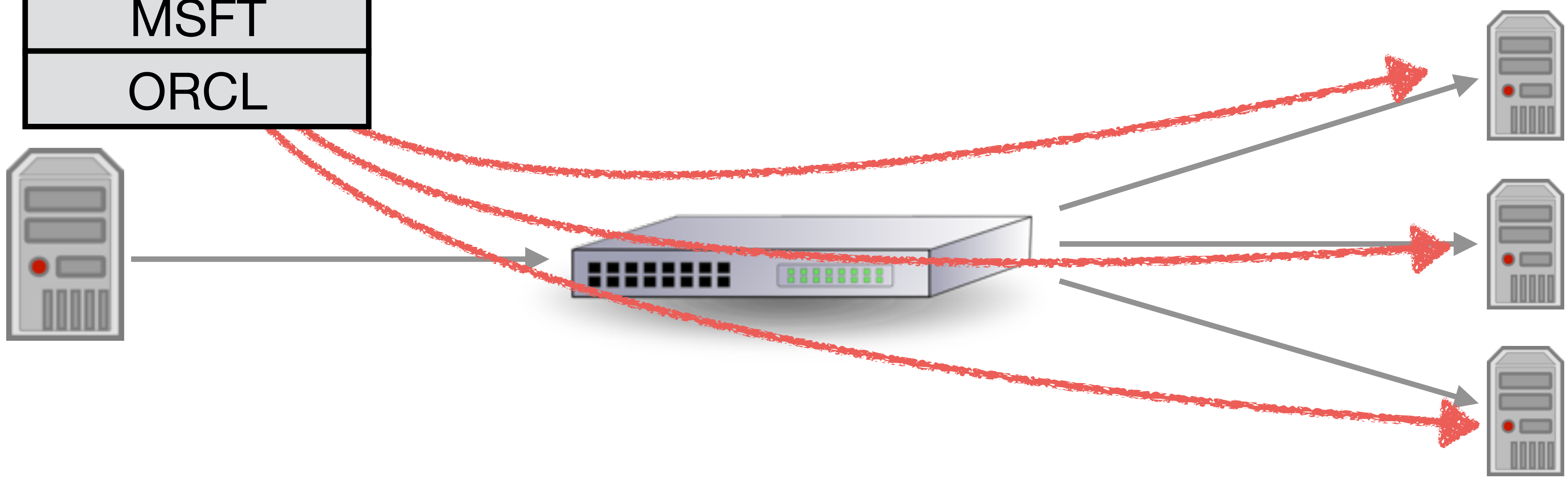
Publish/Subscribe Is Critical to Distributed Applications

Motivating Example: ITCH Market Feed

1 NASDAQ publishes feed:

Ether/IP/UDP
MOLD
GOOGL
MSFT
ORCL

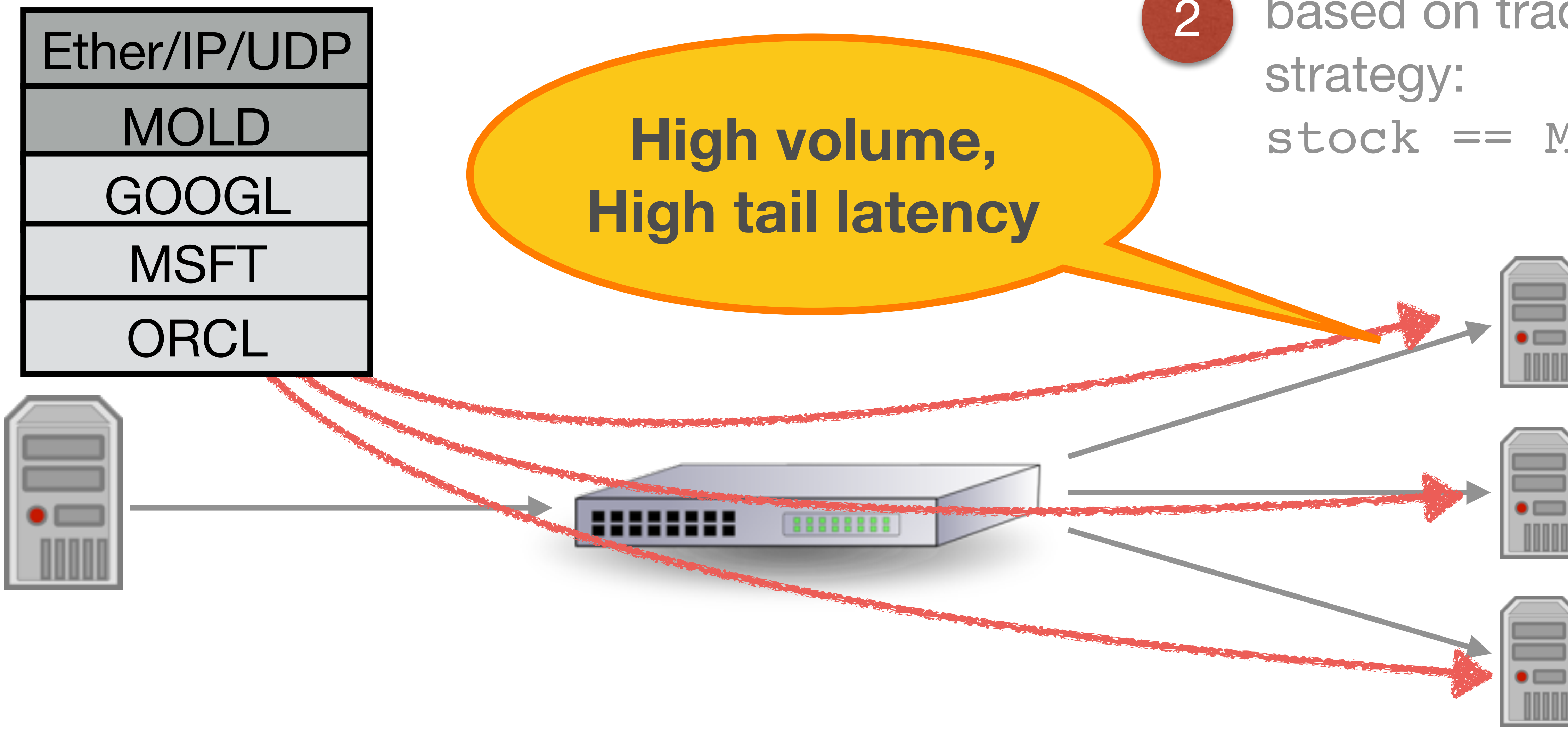
2 End-Points filter based on trading strategy:
`stock == MSFT`



Motivating Example: ITCH Market Feed

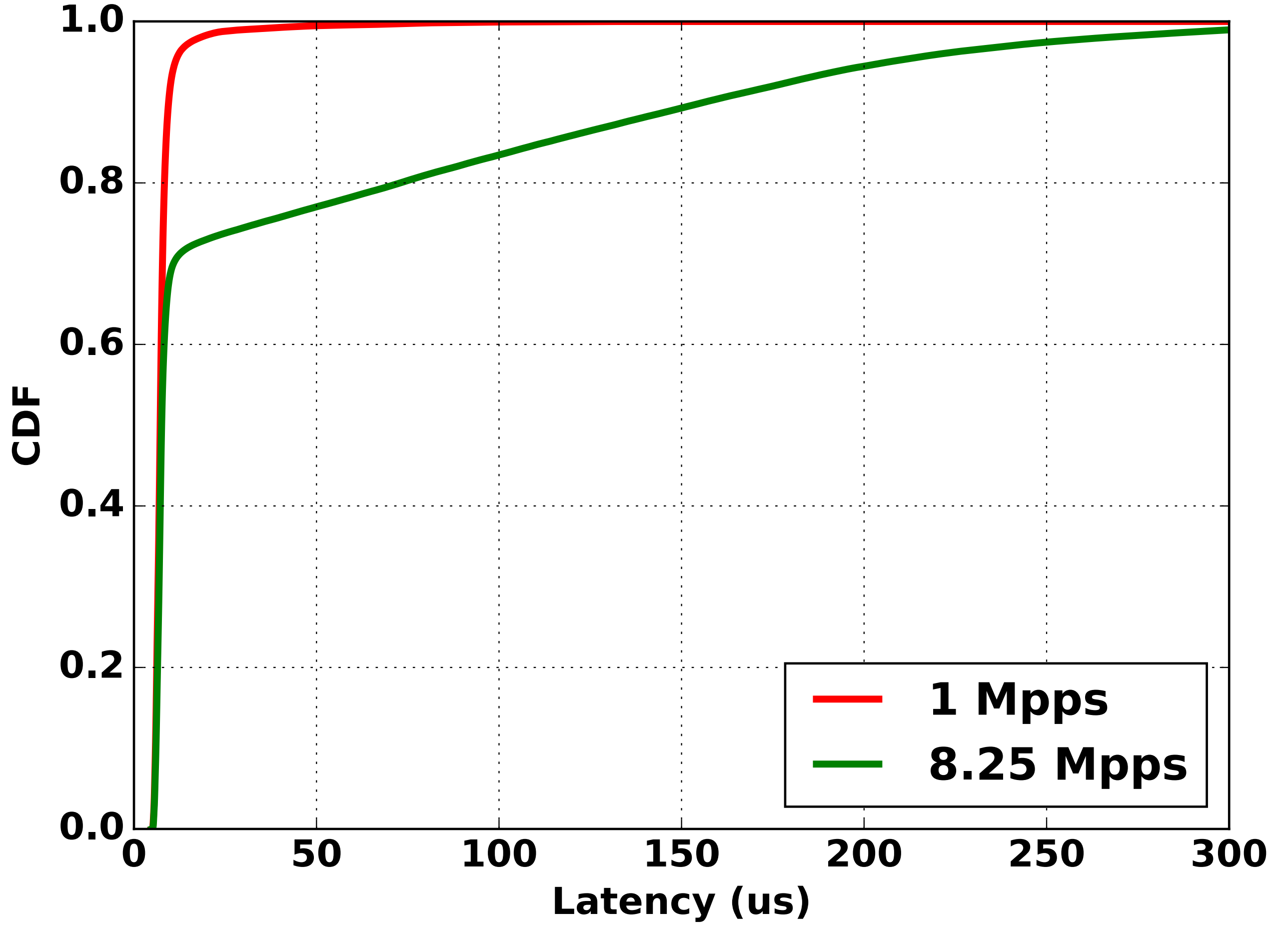
1 NASDAQ publishes feed:

2 End-Points filter based on trading strategy:
`stock == MSFT`



In-software Processing: Multicast + Kernel Bypass

- Unnecessary congestion in the network
- Burden of filtering on hosts leads to queuing
- Highlights need for “in network” solution

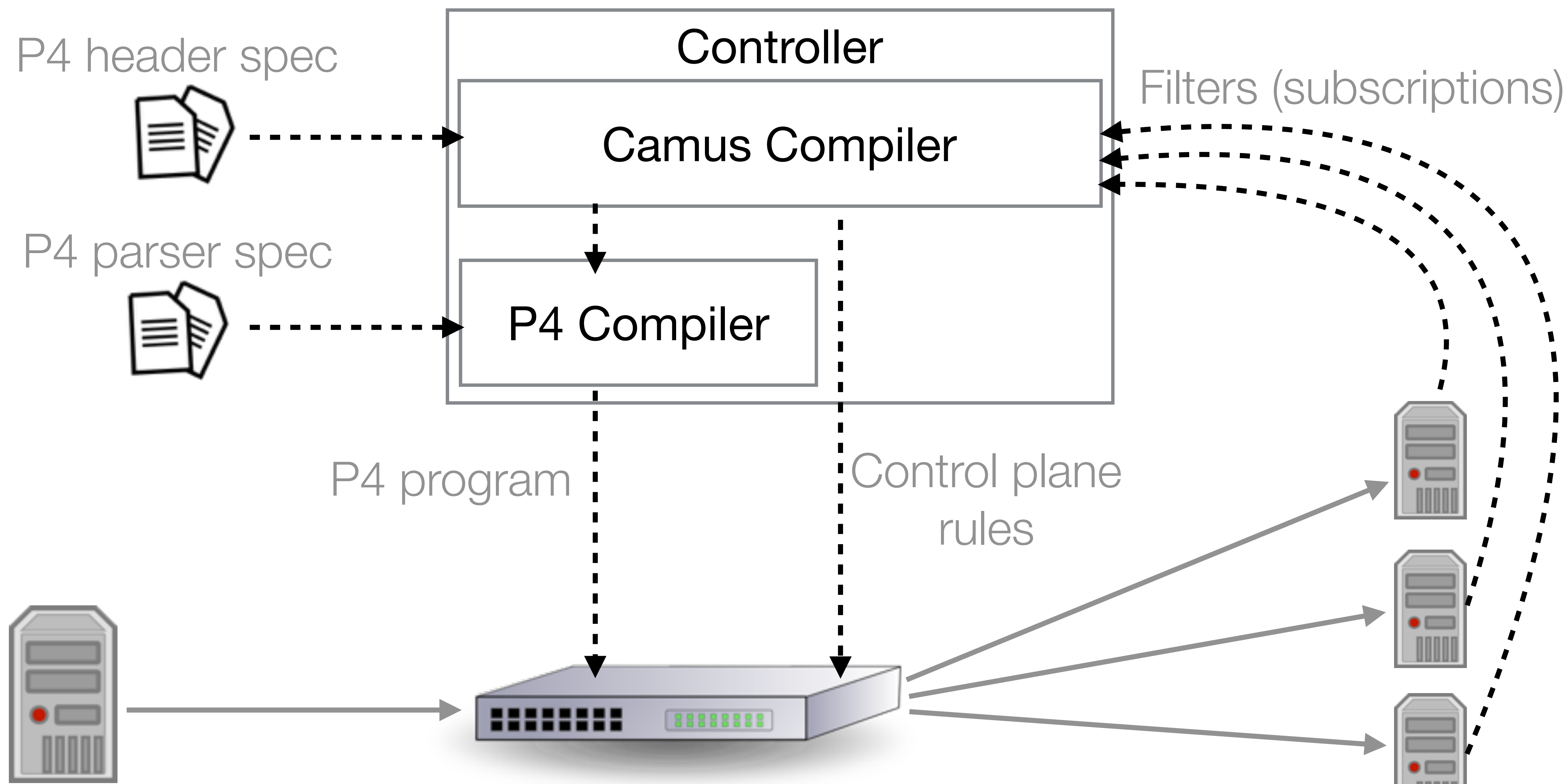


Nasdaq 9/30/17, 0.5% GOOGL

Challenges

- Different applications have different message formats
- Filter content based on expressive conditions
- Deep packets and multiple messages per packet

Camus: Dataplane Pub/Sub



Publisher Interface

- A publisher simply composes and sends packets
- Camus generates application-specific parsing logic
- Parsing logic is static, installed once with Camus

Subscriber Interface

- Filters are boolean formulas of atomic predicates and an action

```
stock == GOOGL : fwd(1)
```

- A forwarding action may be unicast or multicast:

```
stock == GOOGL : fwd(1,2,3)
```

- Rules may be stateful or compute a function:

```
stock == GOOGL ∧ avg(price) > 50 : fwd(1)
```

Compiling Static Pipeline

```
header_type itch_add_order_t {
    fields {
        stock_locate: 16;
        /* … */
        shares: 32;
        stock: 64;
        price: 32;
    }
}
header itch_add_order_t add_order;
@pragma query_field(add_order.shares)
@pragma query_field(add_order.price)
@pragma query_field_exact(add_order.stock)
@pragma query_counter(my_counter, 100, 1024)
```

Compiling Static Pipeline

```
header_type itch_add_order_t {
    fields {
        stock_locate: 16;
        /* … */
        shares: 32;
        stock: 64;
        price: 32;
    }
}
header itch_add_order_t add_order;
@pragma query_field(add_order.shares)
@pragma query_field(add_order.price)
@pragma query_field_exact(add_order.stock)
@pragma query_counter(my_counter, 100, 1024)
```

P4 header for message format

Compiling Static Pipeline

```
header_type itch_add_order_t {
    fields {
        stock_locate: 16;
        /* … */
        shares: 32;
        stock: 64;
        price: 32;
    }
}
header itch_add_order_t add_order;
@pragma query_field(add_order.shares)
@pragma query_field(add_order.price)
@pragma query_field_exact(add_order.stock)
@pragma query_counter(my_counter, 100, 1024)
```

P4 header for message format

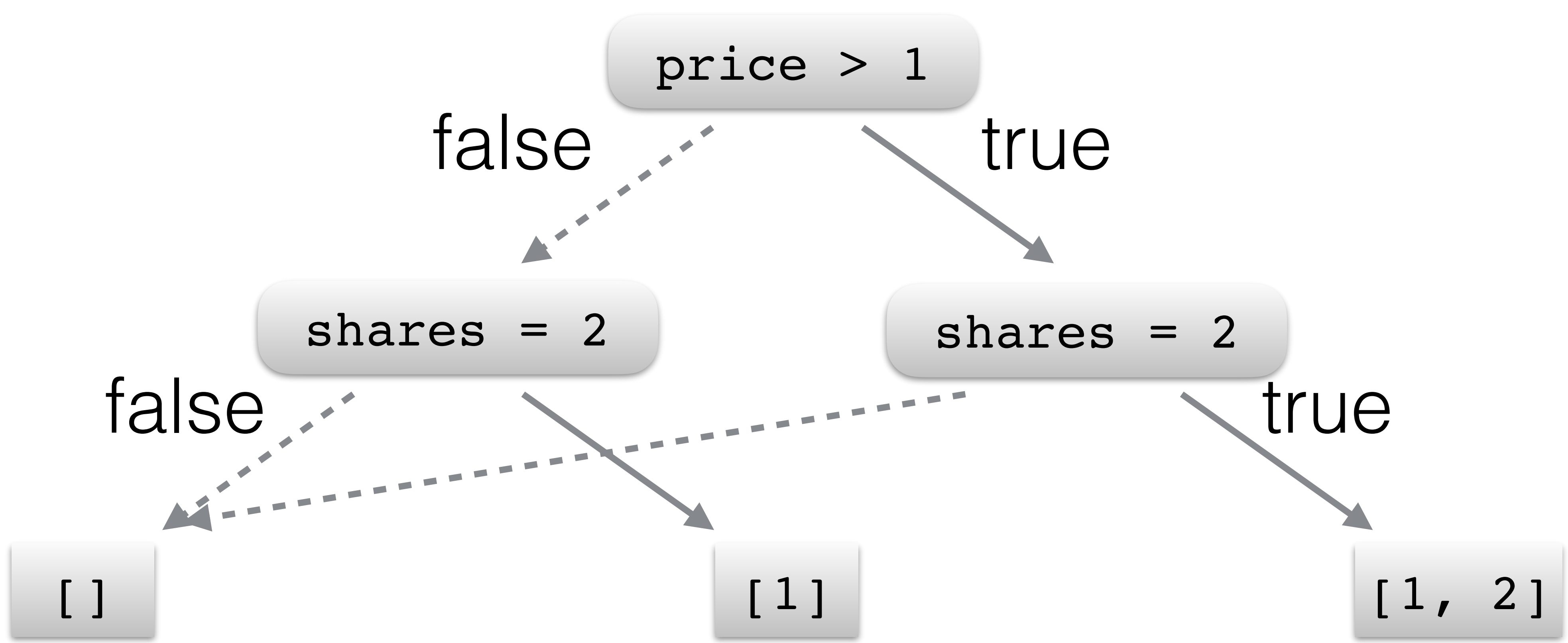
Pragmas for pipeline and state

Compiling Dynamic Filters: Representing Rules with BDDs

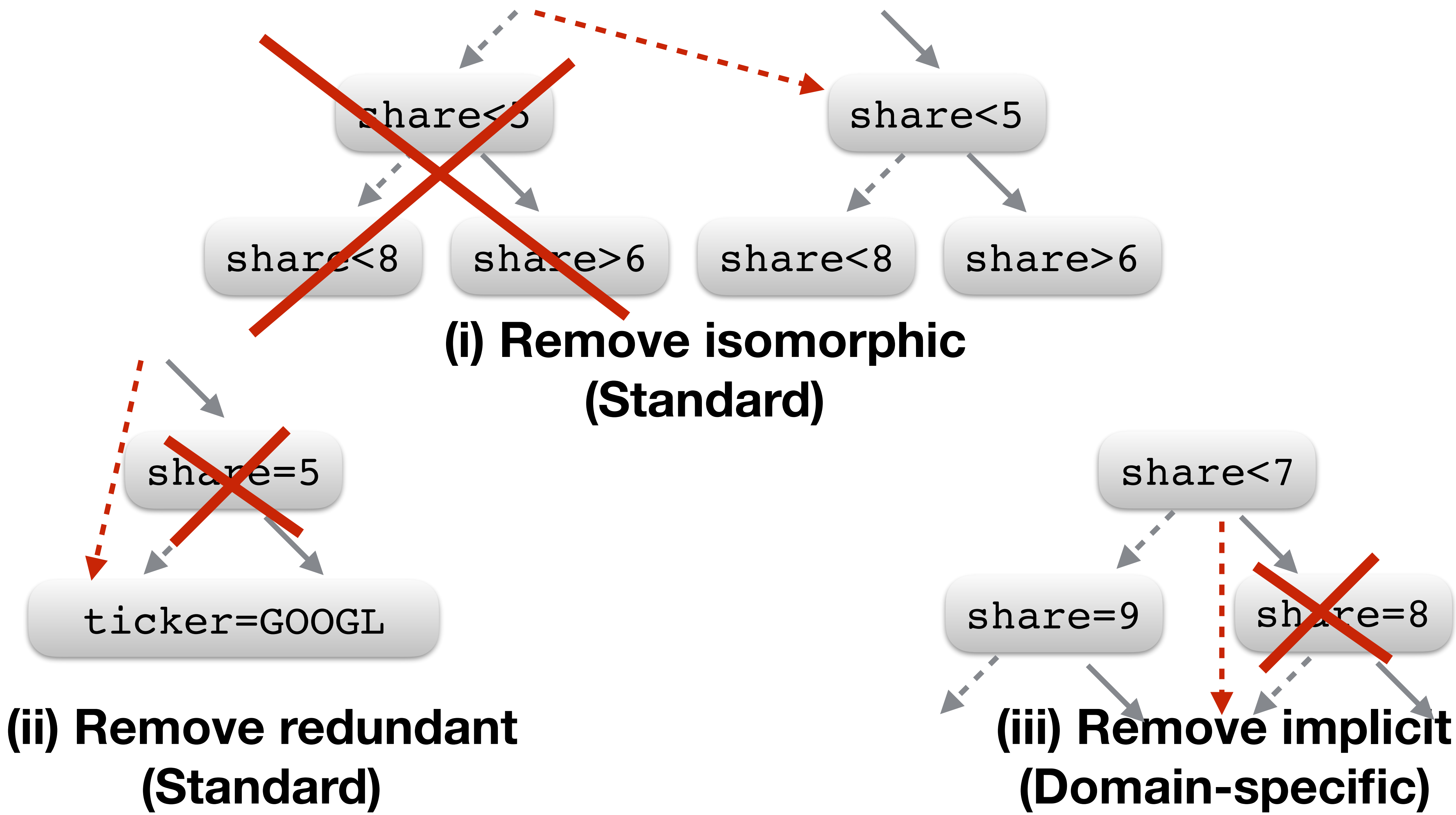
```
shares==2 : fwd(1)
price>1 ∧ shares==2 : fwd(2)
```

Compiling Dynamic Filters: Representing Rules with BDDs

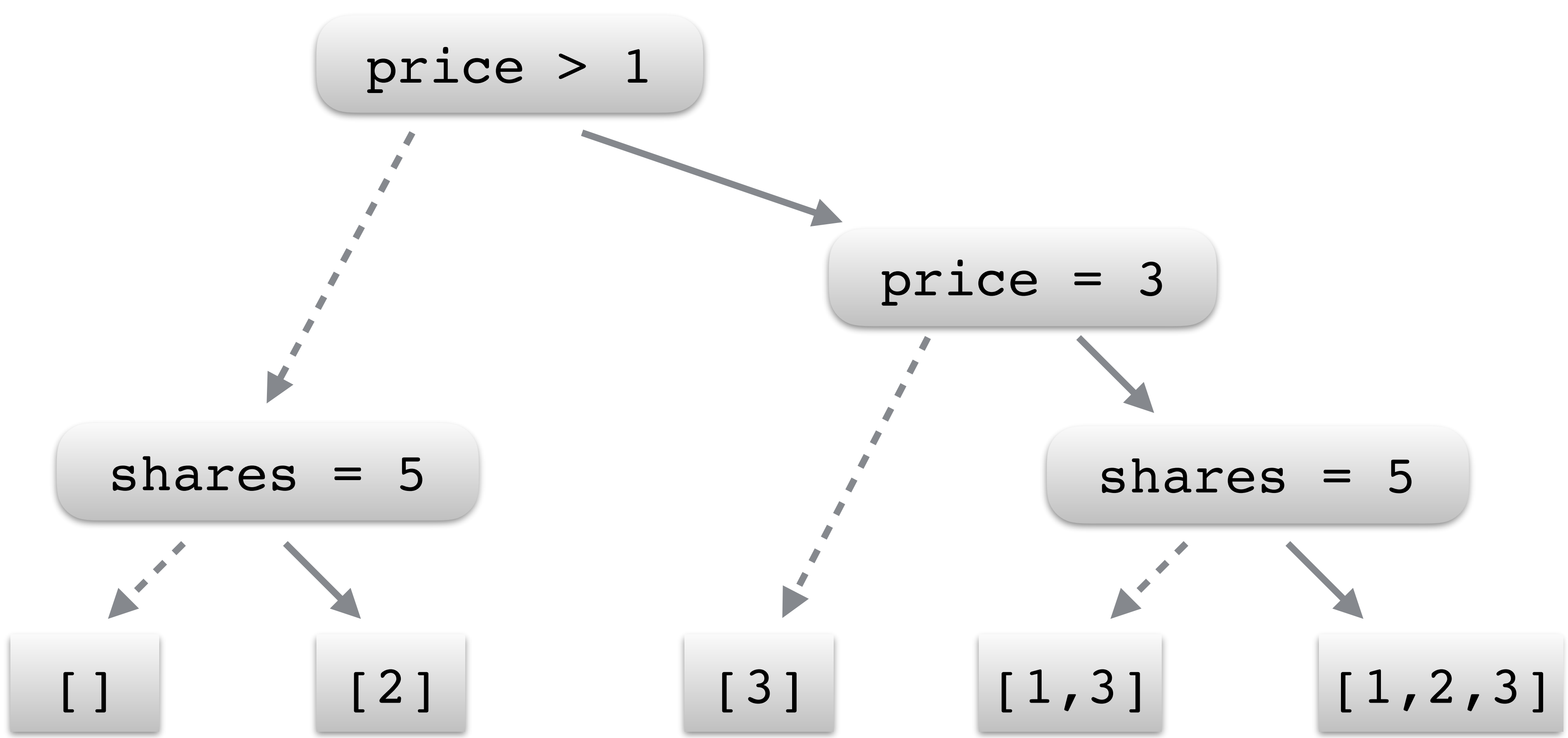
```
shares==2 : fwd(1)
price>1 ∧ shares==2 : fwd(2)
```



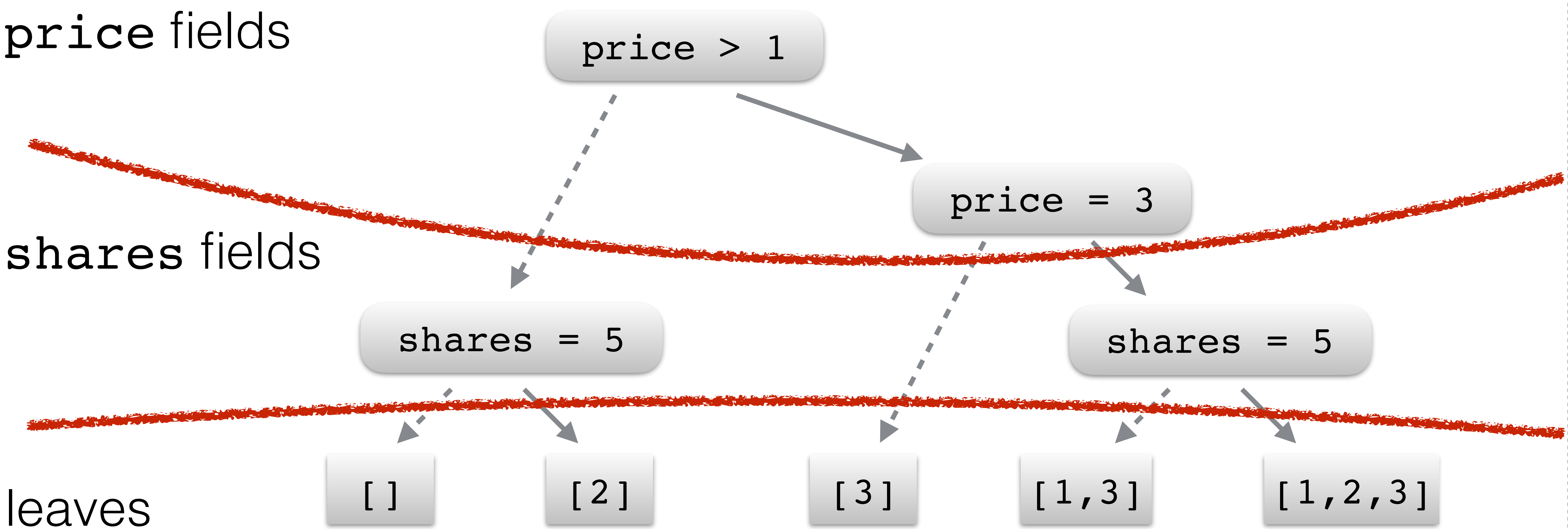
Compiling Dynamic Filters: BDD Reductions



Compiling Dynamic Filters: BDDs to Forwarding Table (1/4)

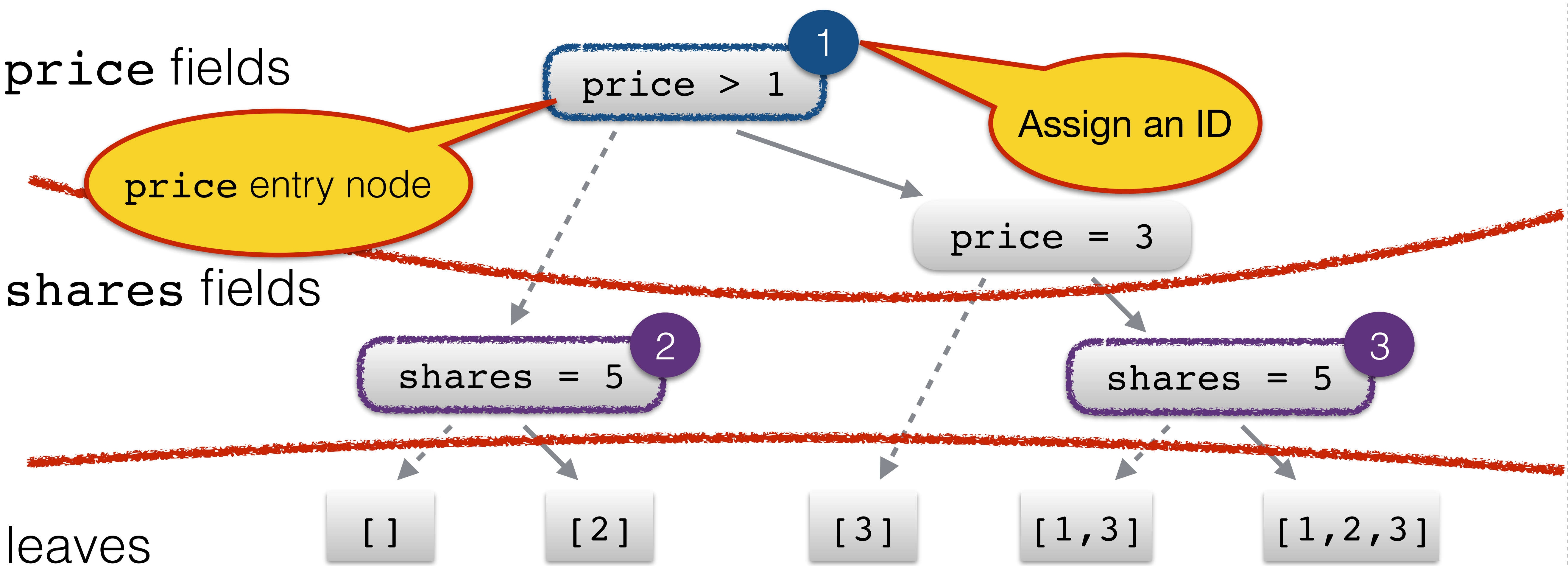


Compiling Dynamic Filters: BDDs to Forwarding Table (1/4)



Partition into sub-graphs by field

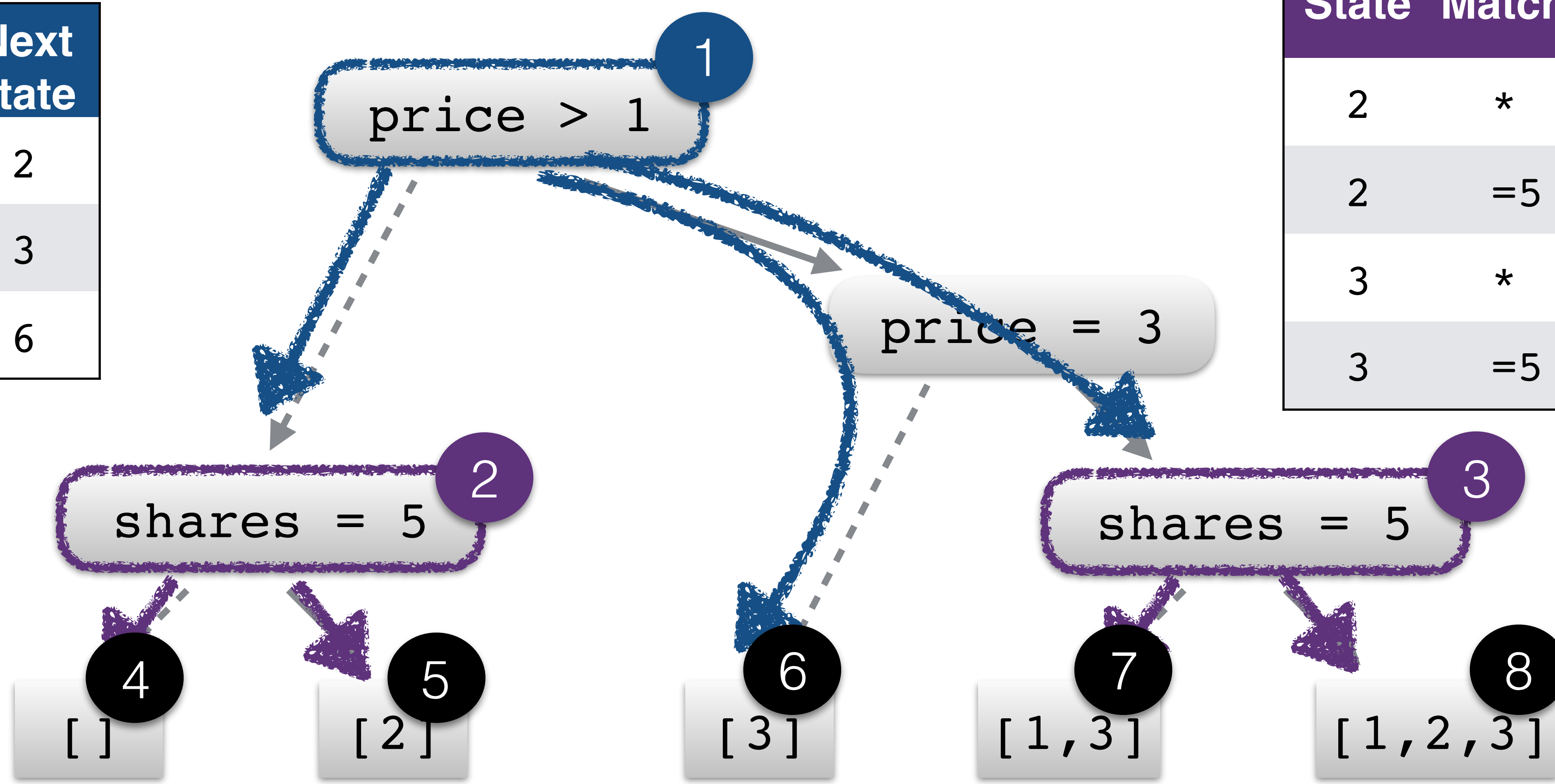
Compiling Dynamic Filters: BDDs to Forwarding Table (2/4)



Identify entry and exit node sets

Compiling Dynamic Filters: BDDs to Forwarding Table (3/4)

State	Match	Next state
1	*	2
1	=3	3
1	>1	6



State	Match	Next state
2	*	4
2	=5	5
3	*	7
3	=5	8

For each path, the tuple (entry ID, match, exit ID) corresponds to an entry in its field's table

Compiling Dynamic Filters: BDDs to Forwarding Table (4/4)

`price` table

State	Match	Next state
1	*	2
1	=3	3
1	>1	6

`shares` table

State	Match	Next state
2	*	4
2	=5	5
3	*	7
3	=5	8

State	Actions
4	[ ]
5	[2]
6	[3]
7	[1,3]
8	[1,2,3]

Encode BDD as finite state machine in the forwarding tables

Multiple Messages Per Packet

- Parsing deep: recirculate packet and advance index
- Routing multiple messages: prune unwanted messages at egress

Evaluation

Compiler Efficiency

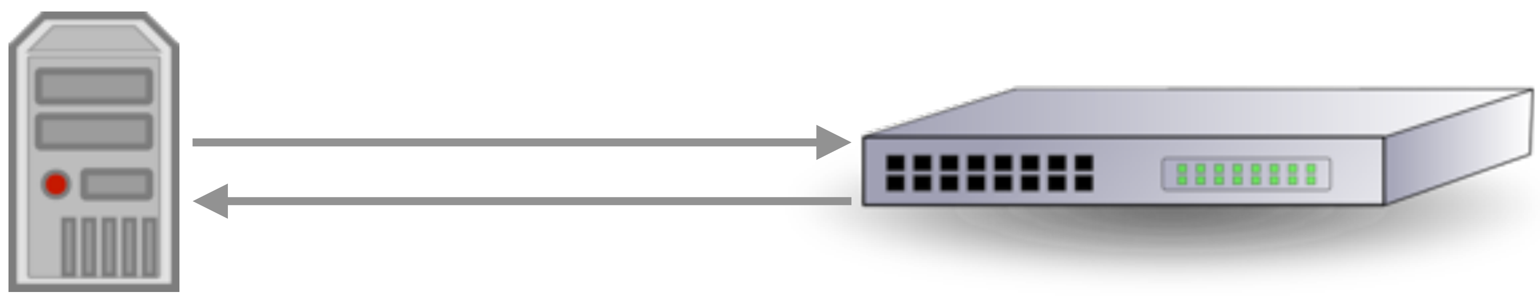
- Used synthetic workload generator to create queries of the form:

```
stock = S ∧ price > P: fwd(H)
```

- Can fit O(100K) queries in switch memory!
- Compiling 100K subscriptions required 21,401 table entries and 198 multicast groups

Experiment: In-Network ITCH Filtering

1 Publisher sends feed (add order)



3 Client calculates latency

2 Switch filters for: stock = “GOOGL”

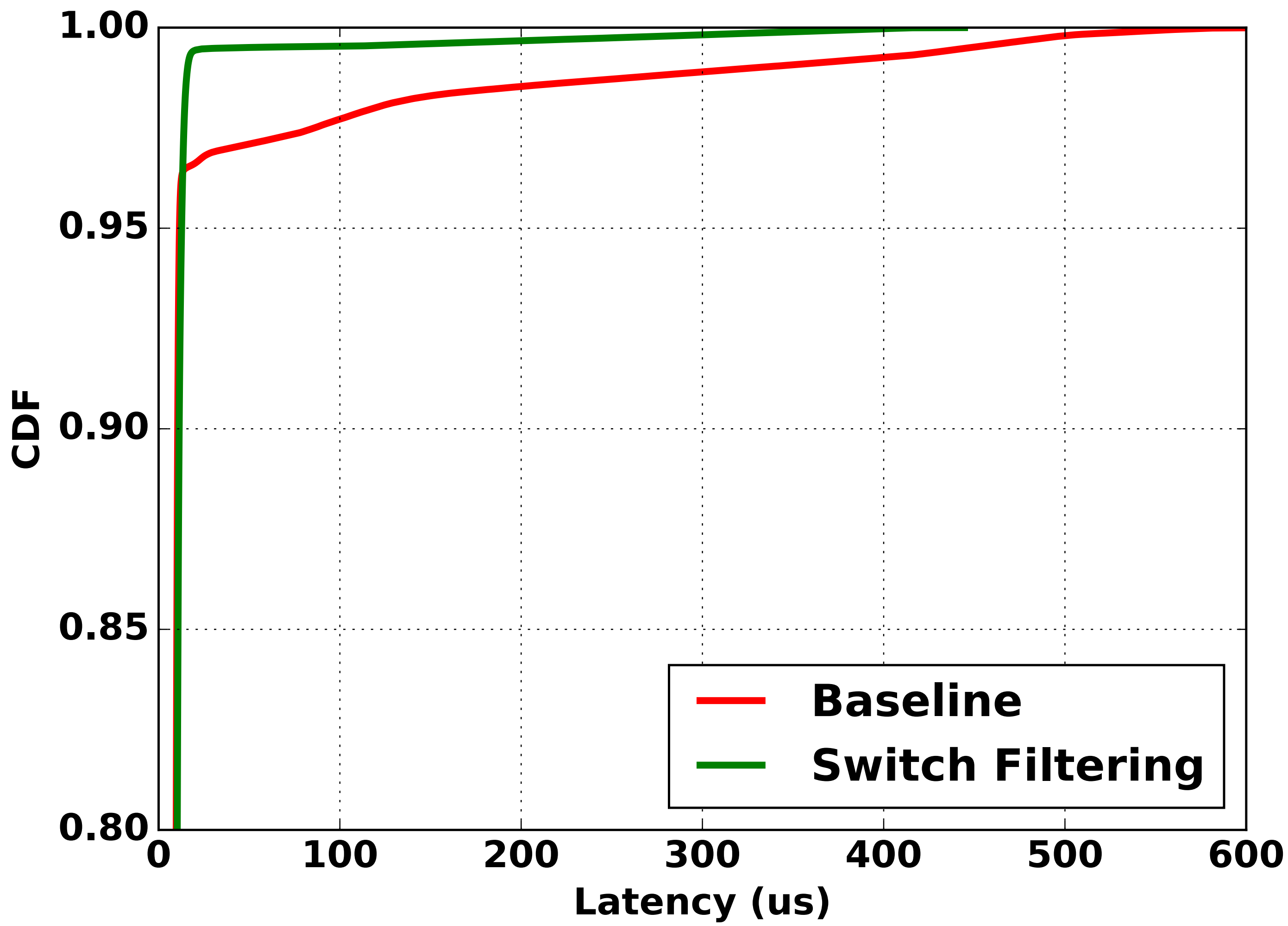
- **Machine has 2 x 25 GB/s NICs**
- **Forward: switch forwards packets; queries evaluated in software**
- **Filter: switch evaluates queries**

Workloads

Workload	Messages per packet	% GOOGL
Synthetic	1-12 (Zipf dist.)	1%
Synthetic (worst case)	Exactly 12	100%
Nasdaq sample 08302017	Exactly 1	0.1%

Synthetic Workload CDF of Latency

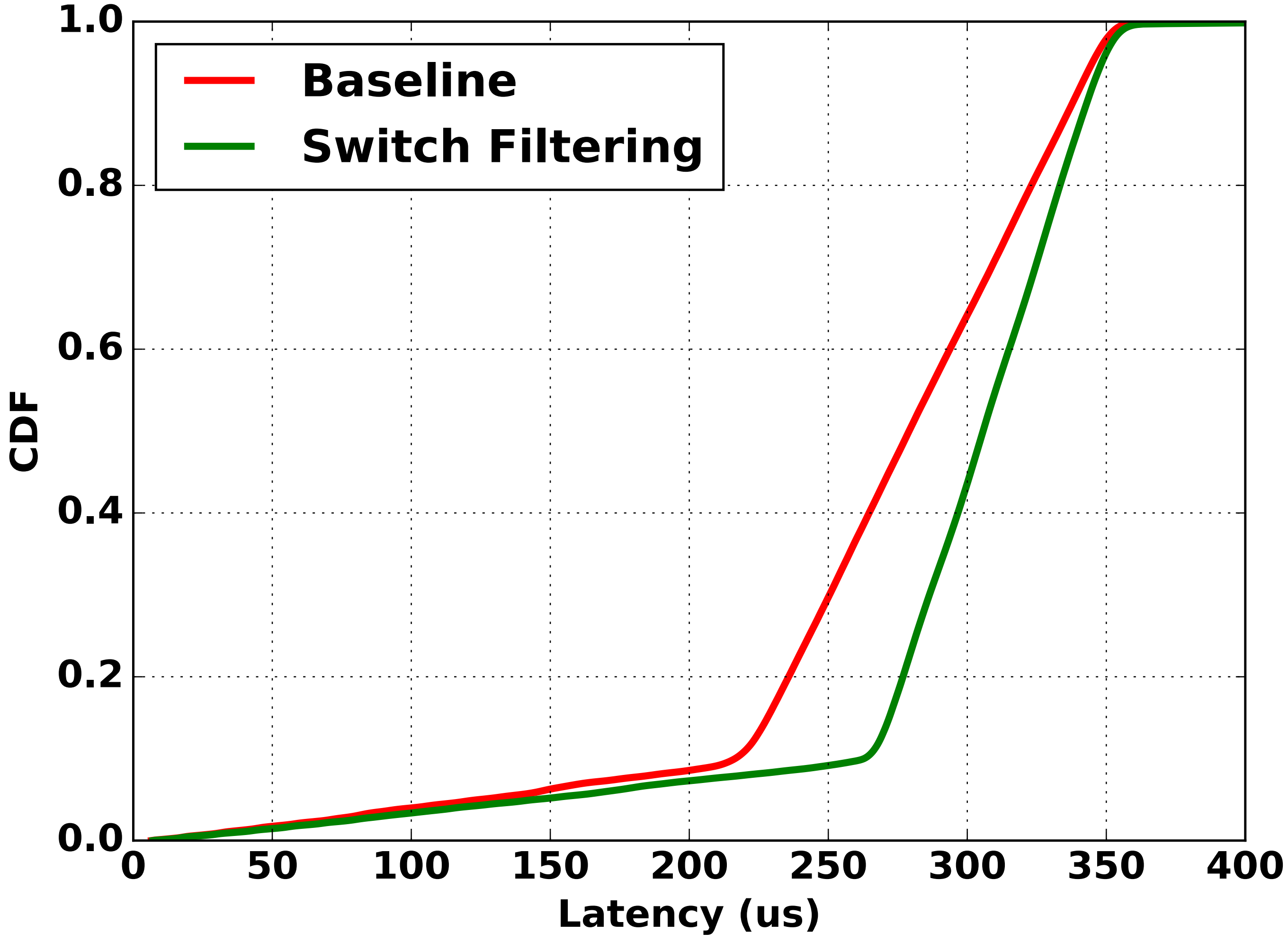
1% GOOGL, 1-12 messages / packet



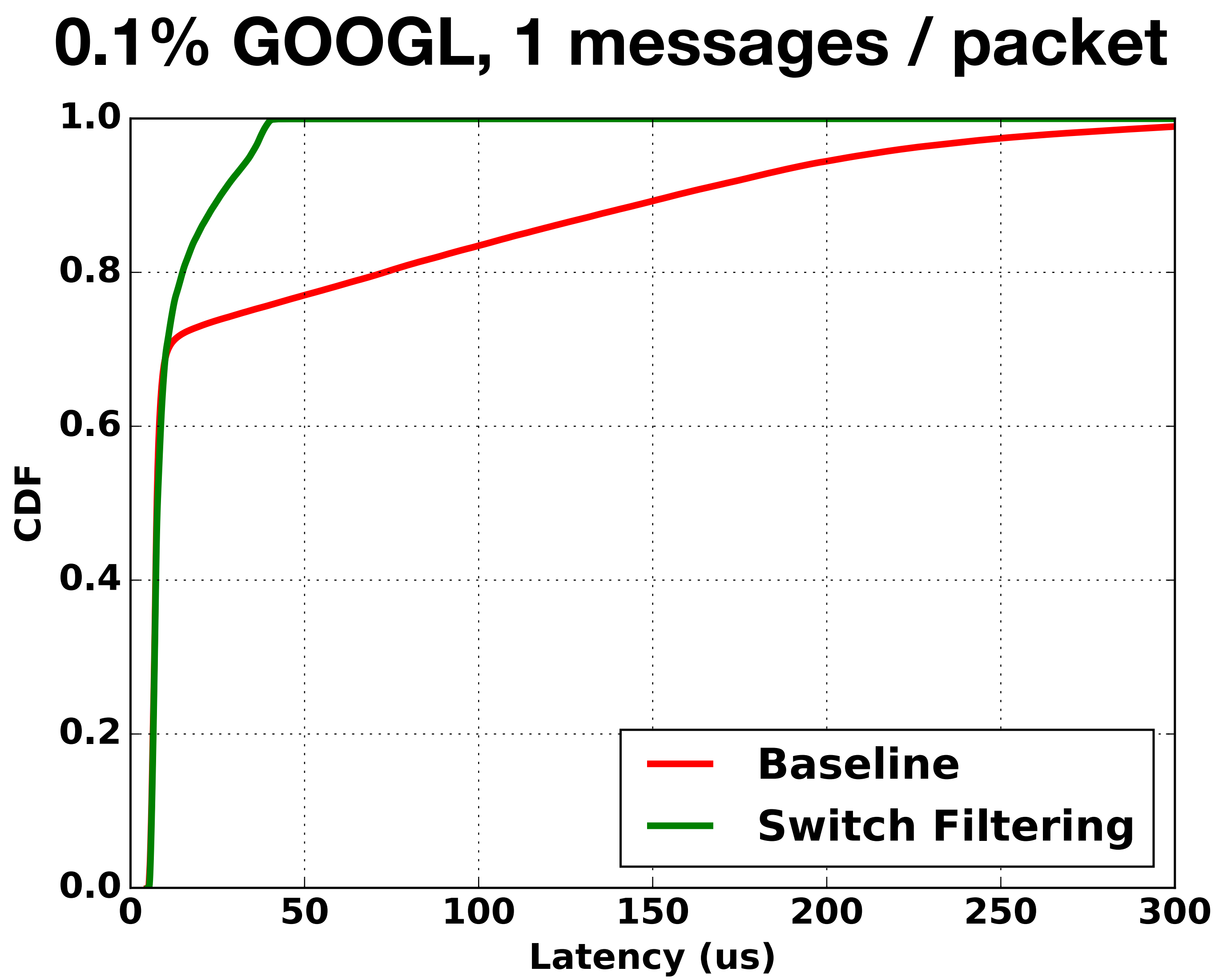
With Camus, 99% finish under 20us
Without Camus, 99% finish under 500us

Worst-Case Workload CDF of Latency

100% GOOGL, 12 messages / packet



NASDAQ Workload (8/30/17) CDF of Latency



With Camus, 100% finish under 100us
Without Camus, 84% finish under 100us

Conclusion

- Camus is a pub/sub service implemented on programmable network ASICs
- Uses a novel BDD-based algorithm to translate predicates into P4 tables that can support O(100K) expressions
- Increases system flexibility and reduces latency for clients

http://inf.usi.ch/phd/jepsen/