

# Fast and Efficient Lookups via Data-Driven FIB Designs

**Sachin Ashok\***, Aditi Partap\*, Ammar Tahir\*

\* Equal Contribution

1st ACM SIGCOMM Workshop on Future of Internet Routing & Addressing (FIRA), 2022



**Stanford**

# Motivation

## Observation #1 — FIB size growth

- FIBs (forwarding information base) map input IP prefixes to output ports
- Rapidly growing number of hosts on the internet
- FIB sizes likely to increase further
- **But switch memory is limited and expensive ...**

# Motivation

## Observation #2 — FIB look up time

- Link speeds continue to increase
- More look ups required per second!

Goal: We need to achieve fast look ups while keeping memory usage low!

# Existing solutions

Two broad approaches:

## I. Probabilistic data structures

- Bloom Filters
  - Answers membership queries using bit arrays (**constant time look up!**)
  - Uses **much lower memory** at the expense of **false positives** (multiple elements map to same index)
  - Existing designs **don't scale for** large no. of ports
  - e.g., Buffalo [CoNext '09]

# Existing solutions

Two broad approaches:

## 2. Hash functions

- Perfect hashing
  - Key value store of  $n$  elements can be stored in a hash table of equal size (**no collisions!**) and can have look ups performed in **constant time**
  - **Hard to update** on the fly and **memory in-efficient as compared to BFs**
  - e.g., Ludo Hashing (ACM SIGMETRICS '20)

# Problem Statement

Can we use machine learning (e.g., decision trees, NNs) to “mine patterns” and create a more memory efficient yet fast FIB design?

# Let's take a look at some real world data ...

- Dataset: Enterprise network
  - FIB snapshots (spanning multiple years) from 10s of core and 100s of edge routers
  - Example of a FIB entry

```
"py/object": "vendor_fib.FWEntry",  
  "prefix": "w1.x1.y1.z1/n",  
  "action": {  
    "py/tuple": [  
      "w2.x2.y2.z2",  
      "VlanM",  
      null  
    ]  
  }
```

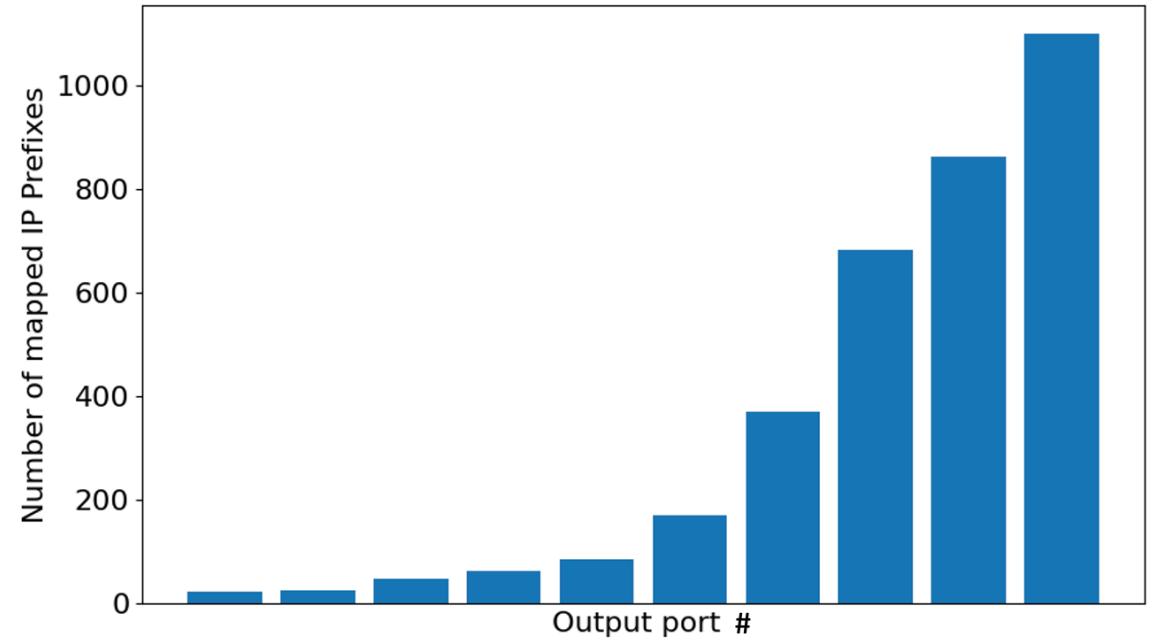
# Lack of learnable patterns

- Tried using ML techniques to predict output port for each packet
- Accuracy < 27% : Lack of learnable patterns!

<b>ML Model</b>	<b>Average Accuracy</b>
Decision Trees	26.5%
Random Forest	26.47%
Decision Trees + AdaBoost	19.13%
Support Vector Machines	25.13%
Linear Regression	25.19%

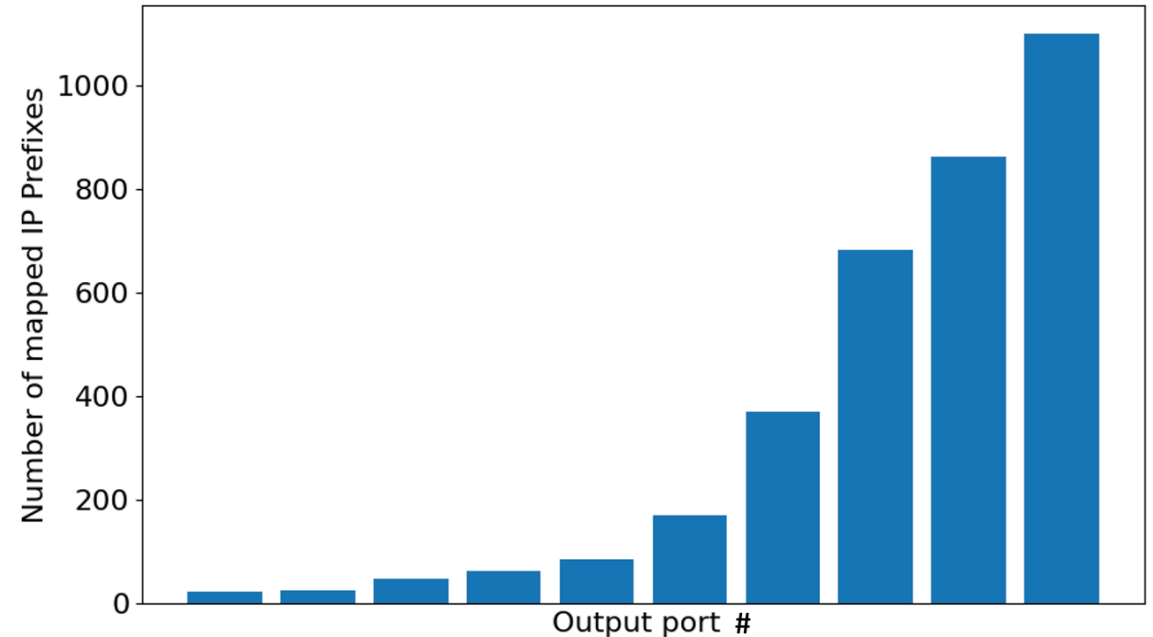
# Heavy Hitter Ports

- Most FIB entries map to a minority of output ports
  - These heavy hitter ports are typically the network's out-facing ports



# Heavy Hitter Ports

- Most FIB entries map to a minority of output ports
  - These heavy hitter ports are typically the network's out-facing ports
- It's easier to learn the classification of IP prefixes to heavy hitter or non-heavy hitter ports



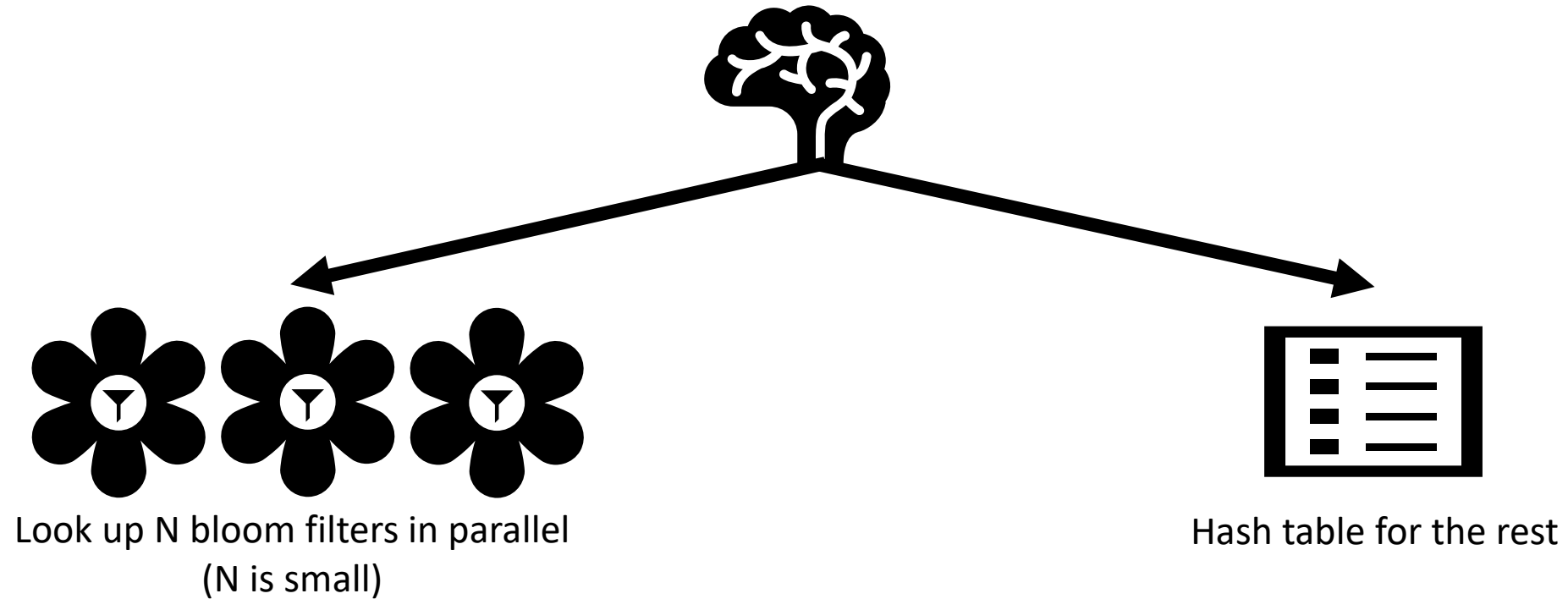
<b>ML Model</b>	<b>Average Accuracy</b>
Decision Trees	86.98%
Random Forest	87.93%
Decision Trees + AdaBoost	93.39%
Support Vector Machines	89.69%
Linear Regression	83.53%

# Heavy Hitter Ports

- Not too many patterns in the data but some observations:
  - Large no. of IP prefixes map to few output ports
    - e.g., 3465 entries → 5 ports
    - Why? possibly packet has to be forwarded to other core routers or outside the network
  - Similarly, many IP prefixes have close to 1-1 mapping to an output port
    - e.g., 4037 entries → 3590 ports
    - Why? possibly packet has to be forwarded to end points
- Can we use these patterns somehow?

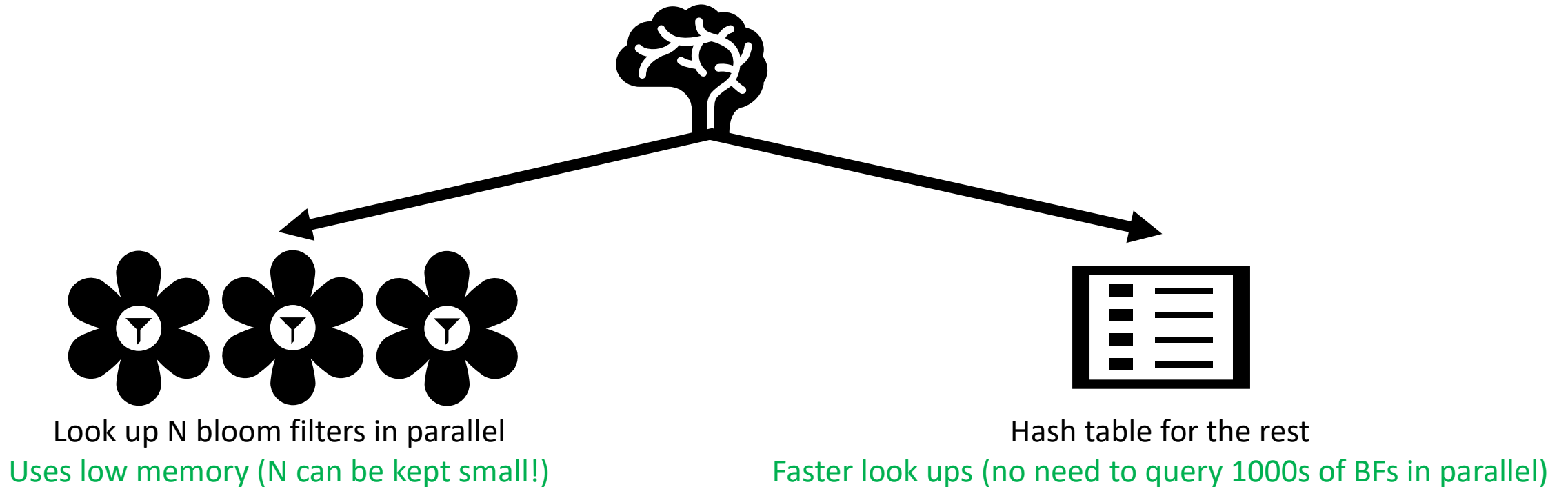
# Our Approach

- Main idea: Use a tiny classifier to classify an input IP prefix as belonging to a heavy hitter output port or not.



# Our Approach

- Main idea: Use a tiny classifier to classify an input IP prefix as belonging to a heavy hitter (HH) output port or not.



# Design choices for Classifier – ML Model

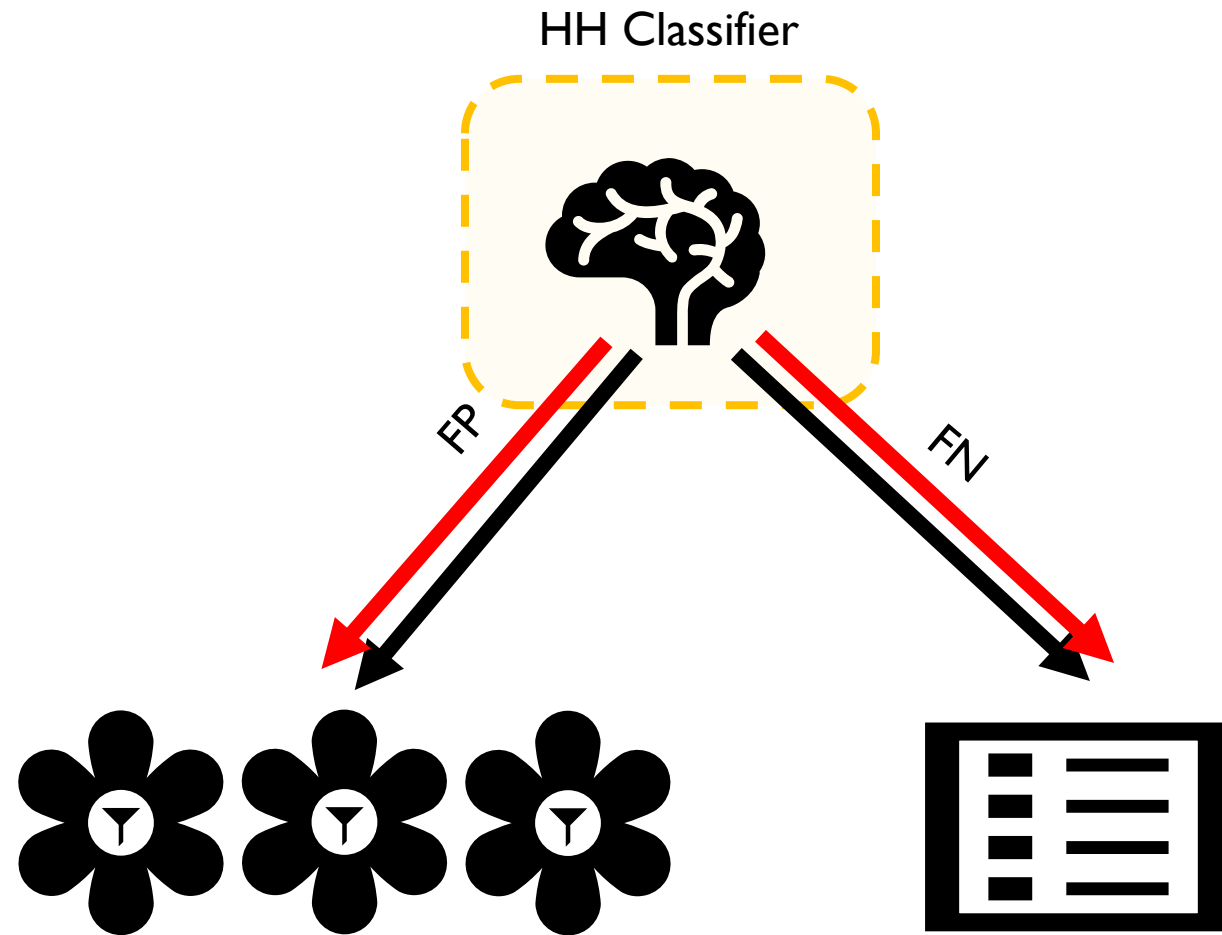
## I. ML model as Heavy Hitter Classifier

### Pros

- Models like decision trees can be much more compact in size

### Cons

- Has false negatives as well as false positives



# Design choices for Classifier – Bloom Filter

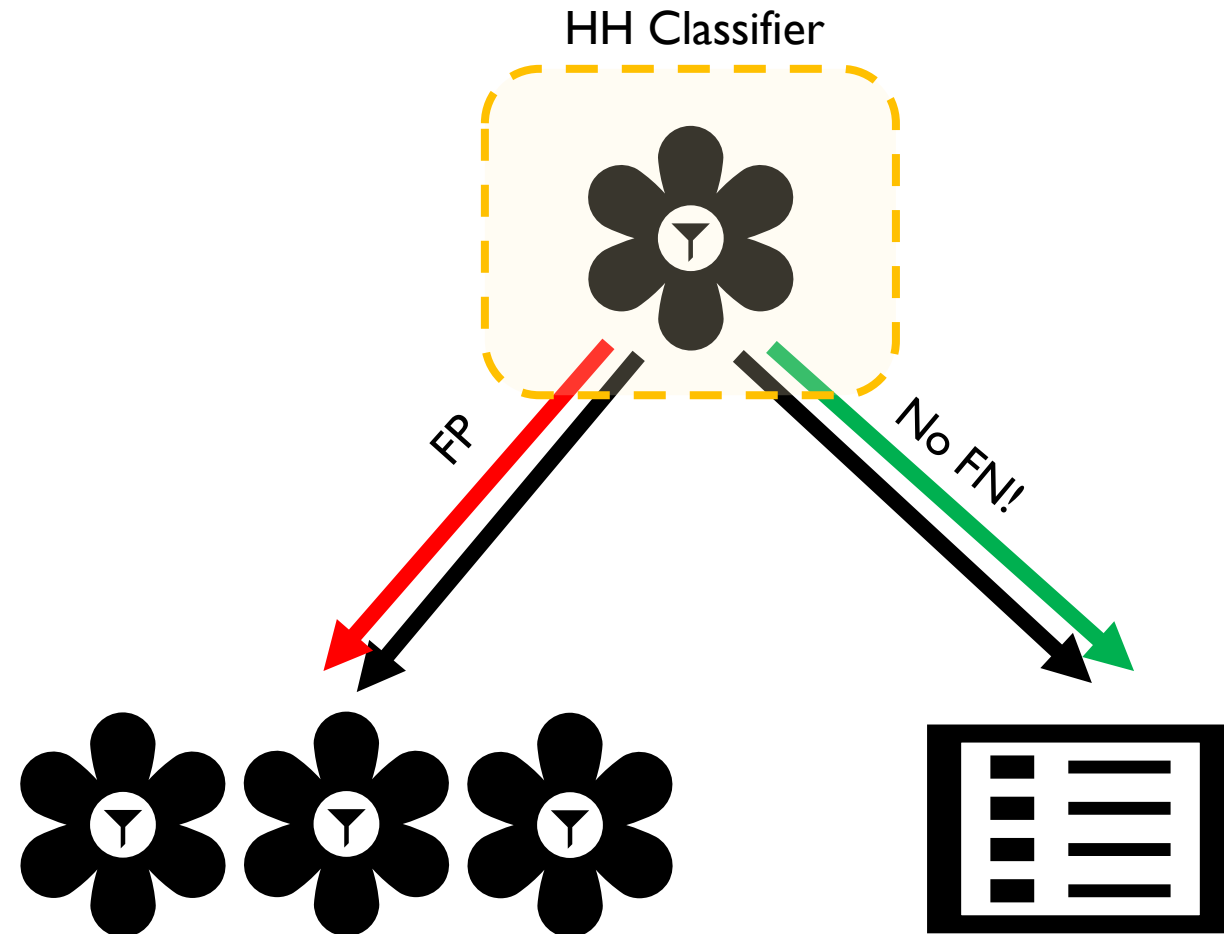
## 2. Single Bloom Filter as the Heavy Hitter Classifier

### Pros

- No false negatives
- Fast

### Cons

- False positive rate depends on size



# Design choices for Classifier – Hybrid

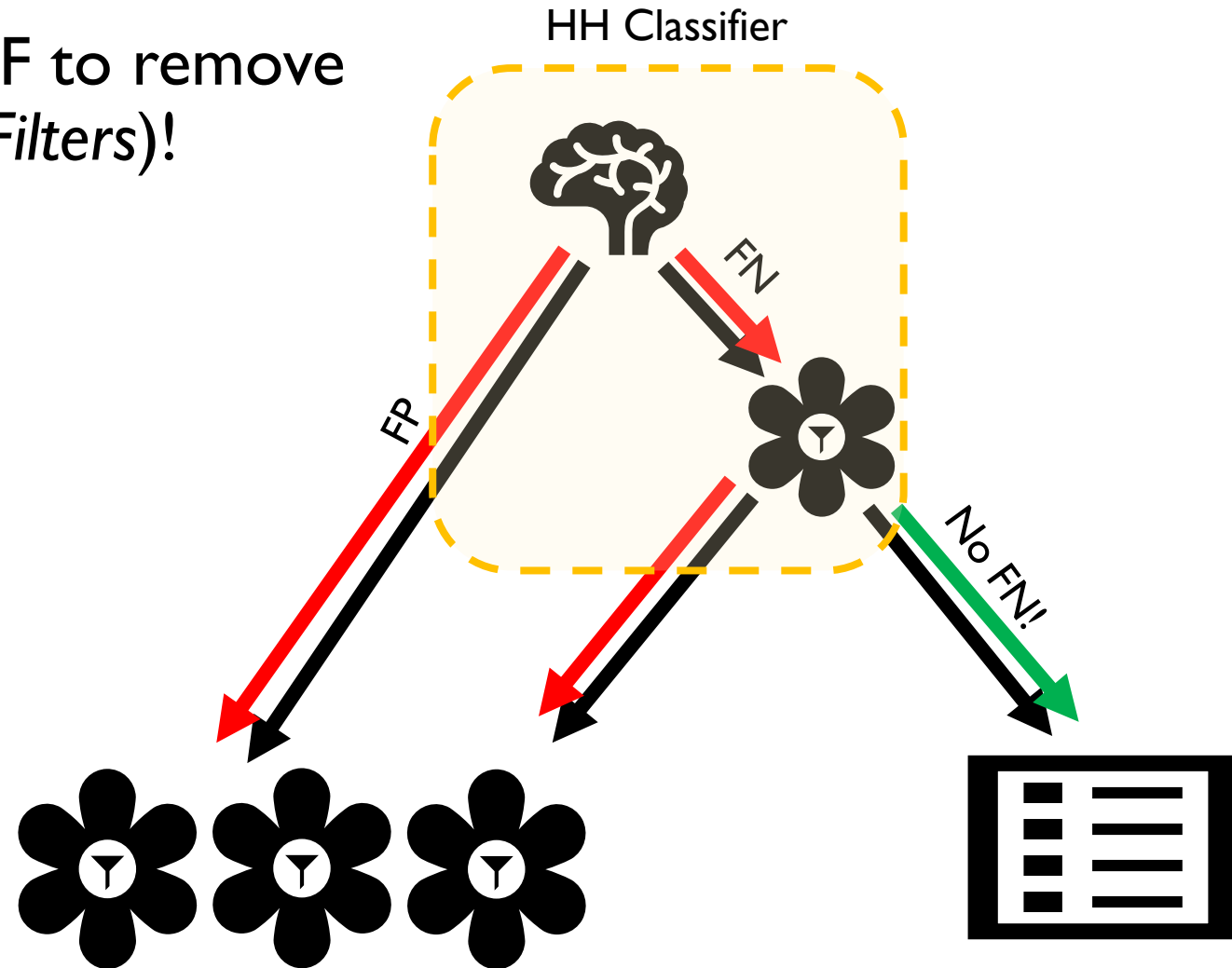
3. Back up ML classifier with a BF to remove false negative (*Learned Bloom Filters*)!

## Pros

- No false negatives
- Smaller size than a BF-based design

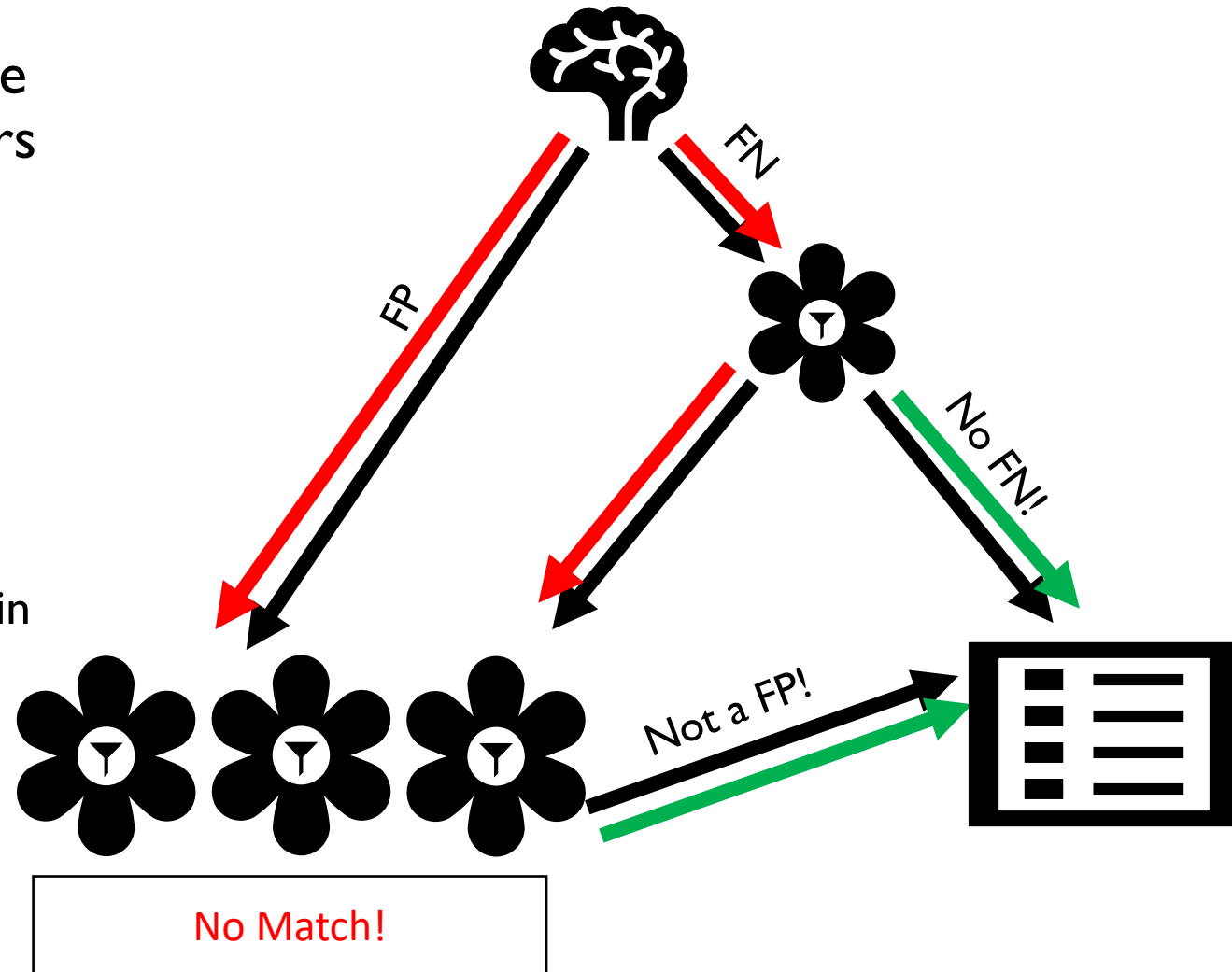
## Cons

- Slightly higher memory and lookup time than standalone ML based design



# Handling False positives

- Classifier's false positive means that we are looking up for a key in bloom filters which does not exist there.
- Solution: If no match in BFs, look up sequentially in HashTable!
  - This does not account for false positives in lower bloom filters!



# Theoretical Formulation

- Goal: minimize error while fitting all Bloom Filters in limited memory.
- Objective Function: Sum of
  - False Positive Rate of Bloom Filters for heavy hitter classification (minimize number of sequential lookups)
  - False Positive Rate of Bloom Filters for each heavy hitter port (minimize number of wrong forwarding decisions)
- Constraint on total memory usage, including space used by
  - ML model and BF (Bloom Filter) for classification
  - LudoHash for all entries mapping to non heavy hitter ports
  - BF for each heavy hitter port

# Performance analysis (setup)

- Implemented in C++, used LudoHash codebase
- Ran experiments on AWS VM, on single core.
- Used real FIB data (more realistic patterns of # of heavy hitters)

# Performance analysis (metrics)

## 1. Memory (theoretical): $m1 + m2 + m3$

- $m1$ : ML model or BF or ML model + BF size
- $m2$ : BF size for heavy hitters
- $m3$ : Ludo size for non-heavy hitters

## 2. Query time

- $T1$ : ML model or BF or ML model + BF
- $T2$ : BF for heavy hitters (|| look up across BFs, pick max time for calculation)
- $T3$ : Ludo query time
- Classifier says Yes  $\rightarrow T1 + T2$ , False +ve  $\rightarrow + T3$
- Classifier says No  $\rightarrow T1 + T3$

## 3. Accuracy

# Performance analysis (dimensions)

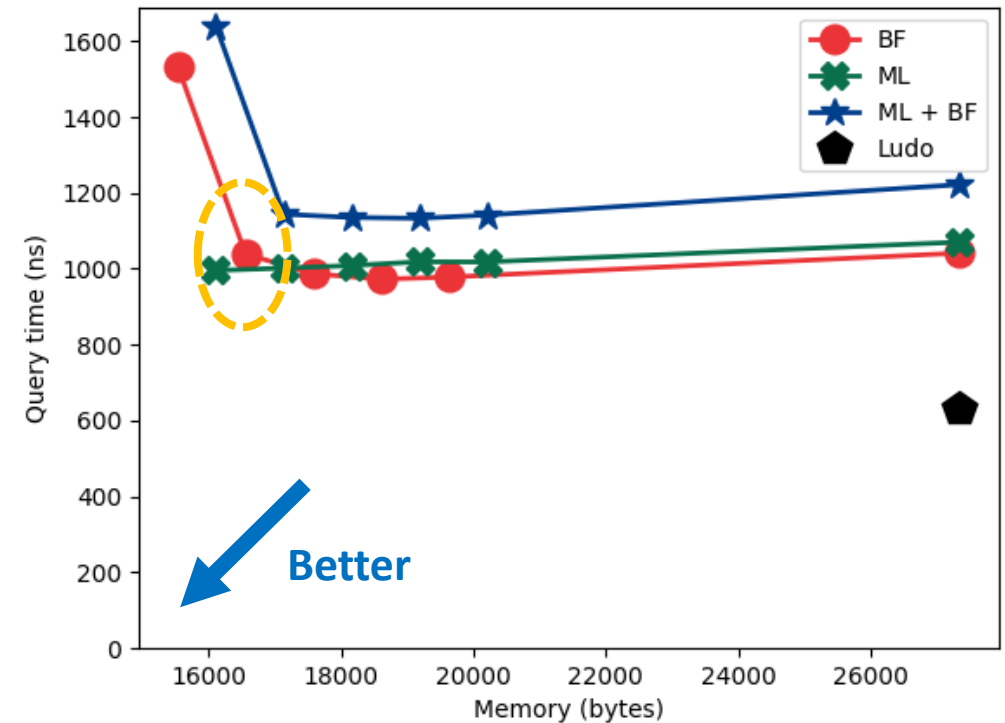
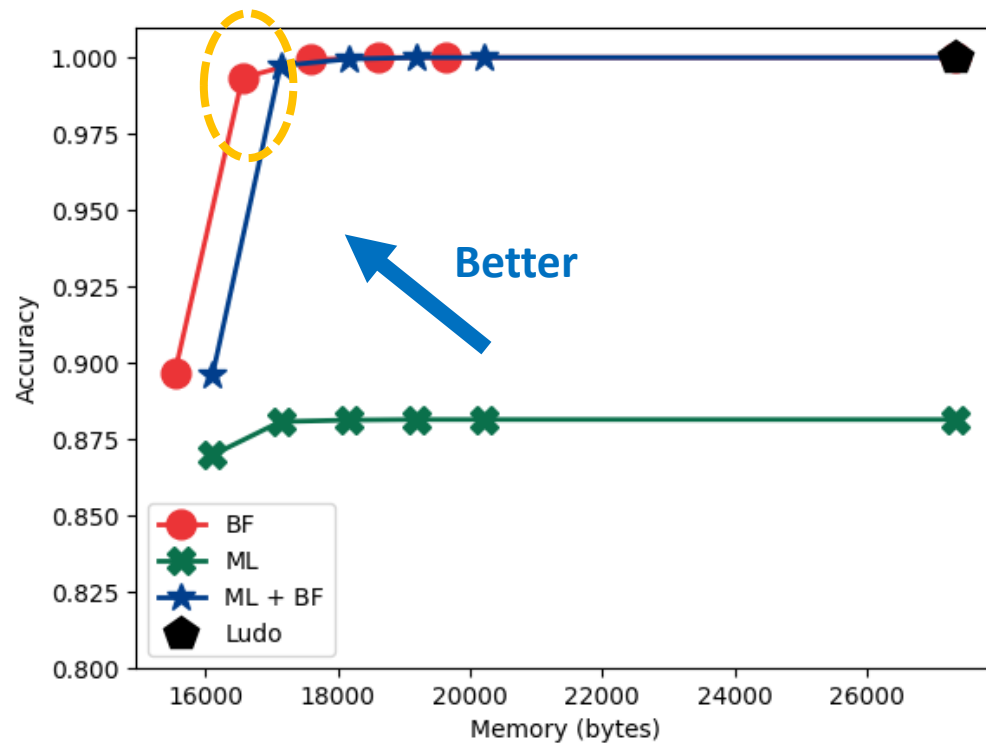
1. Available switch memory
2. Workloads
  1. Uniform
  2. Zipfian

# Performance analysis (results)

1. 47% IPs are heavy hitters
2. Memory gains are strictly limited to < 50%
3. Takeaway

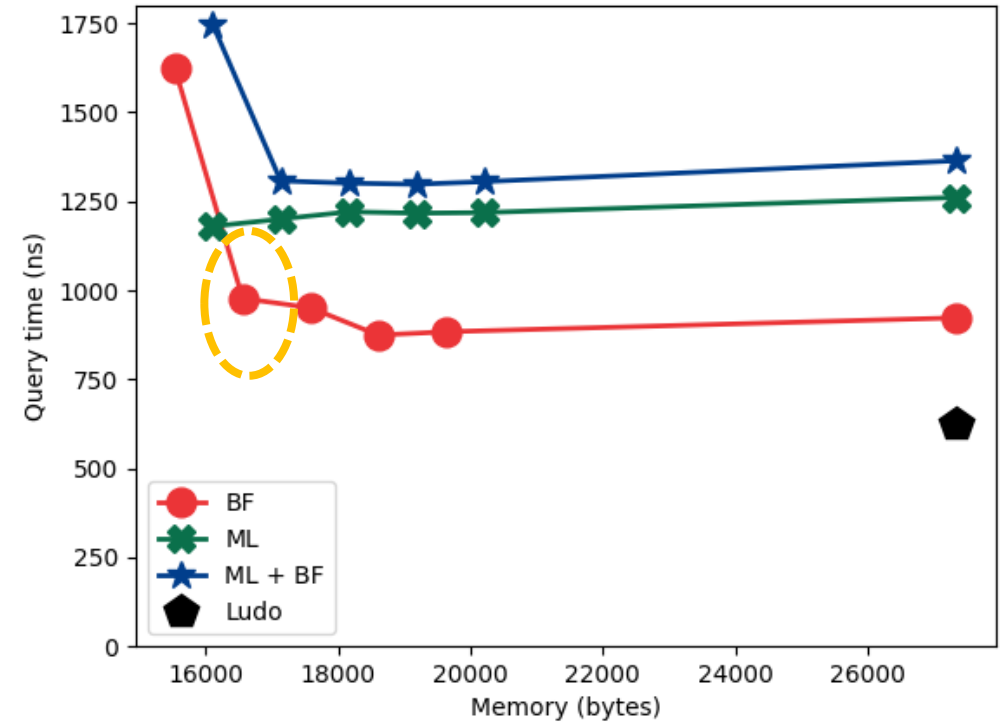
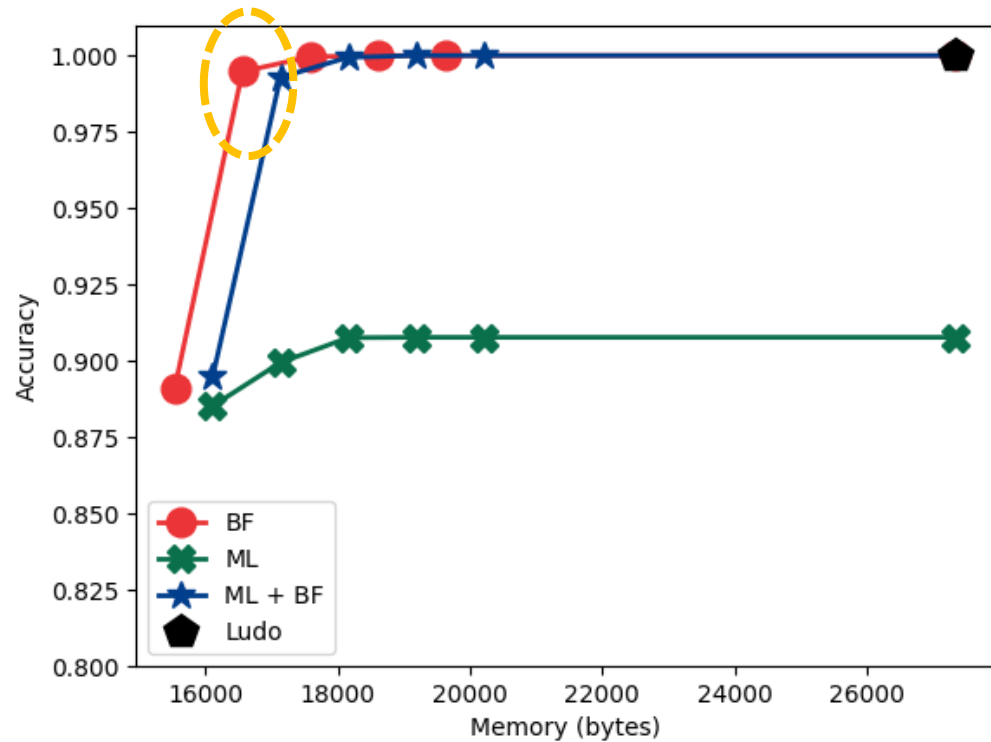
Representative Router 1 (Workload: **Uniform**)

1. Memory: 1.65x lower
2. Query time: 1.5x higher
3. Accuracy: 0.99x



# Performance analysis (results)

Representative Router 1 (Workload: [Zipfian](#))



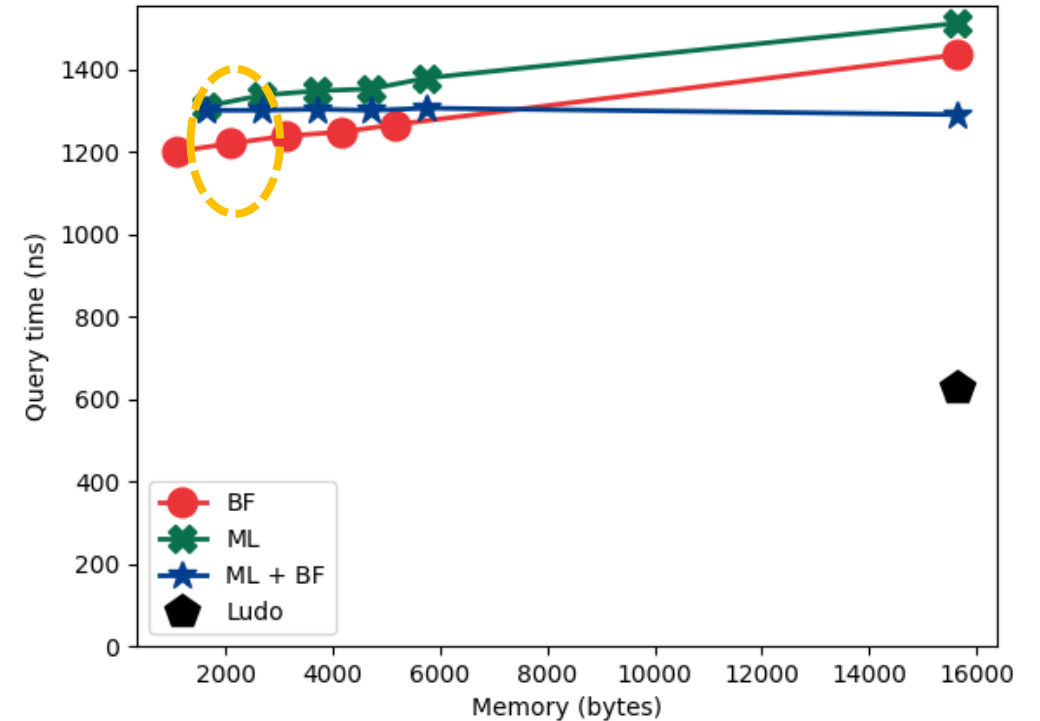
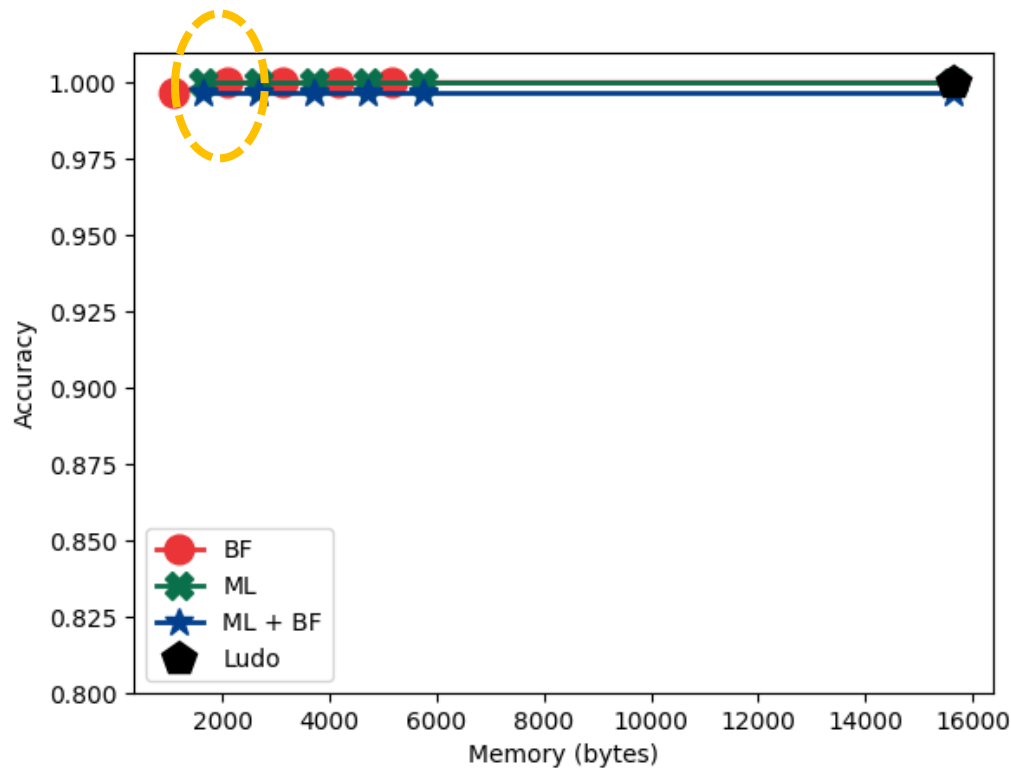
# Performance analysis (results)

1. 90% IPs are heavy hitters
2. Memory gains are strictly limited to < 90%

### 3. Takeaway

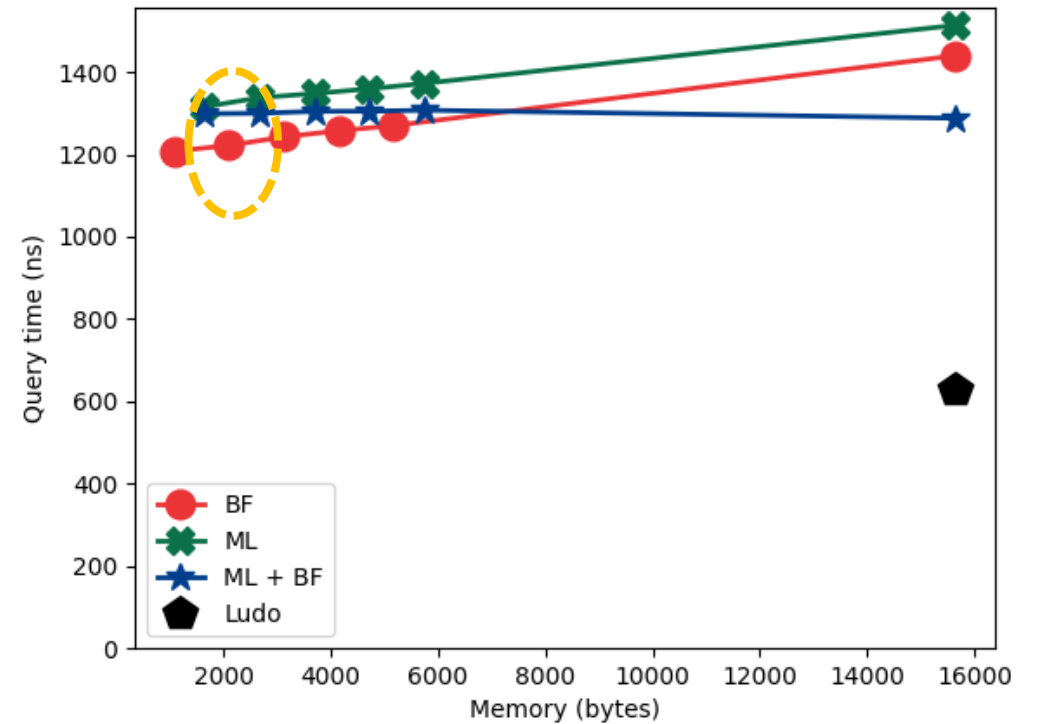
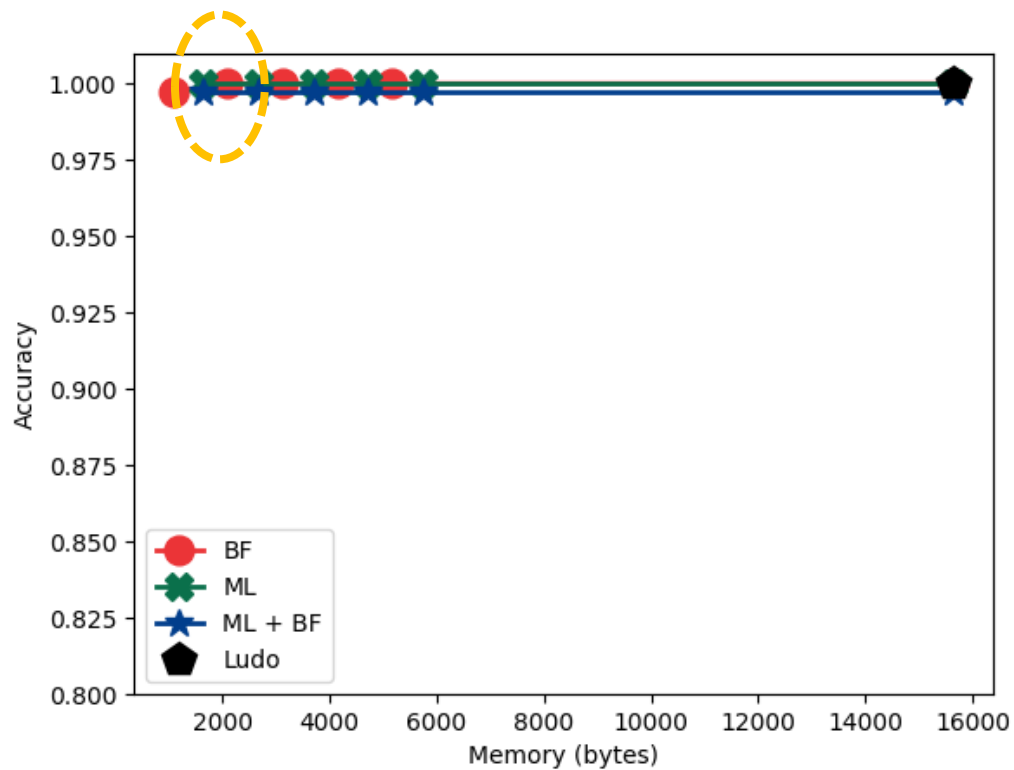
1. Memory: 8x lower
2. Query time: 2x higher
3. Accuracy: 1x

Representative Router 2 (Workload: **Uniform**)



# Performance analysis (results)

Representative Router 2 (Workload: [Zipfian](#))



# Future work

## 1. Design

1. Key set updation
2. Handle longest prefix matching
3. BF to predict non-heavy hitters (reverse problem)

## 2. More evaluation

1. Hardware memory usage & inference time
2. Multi-threaded performance
3. Comparison with more schemes
4. More network datasets

# Conclusion

- Important to achieve **fast look up times** with **limited memory usage** for FIBs.
- Our hybrid design tries to achieve the best of both worlds, currently **saving 2x - 8x memory** while **sacrificing some performance (~1.5x worse)** with **<1% error**.