Advanced Networked Systems SS24

Network Monitoring

Prof. Lin Wang, Ph.D.

Computer Networks Group

Paderborn University

https://cs.uni-paderborn.de/cn

Network monitoring tasks

Network monitoring is fundamental in network performance optimization and security

Traffic engineering

Flow size distribution

Anomaly detection (DDoS)

Entropy, traffic changes

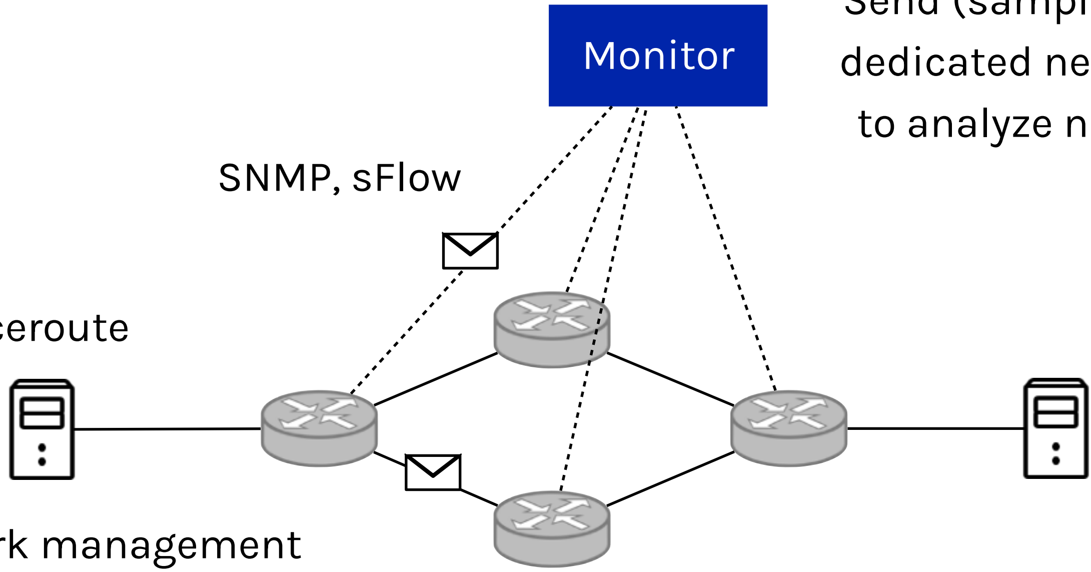
Worm detection

Superspreaders

Accounting

Heavy hitters

Traditional network monitoring



Monitor

Send (sampled) traffic to a dedicated network monitor to analyze network traffic

SNMP, sFlow

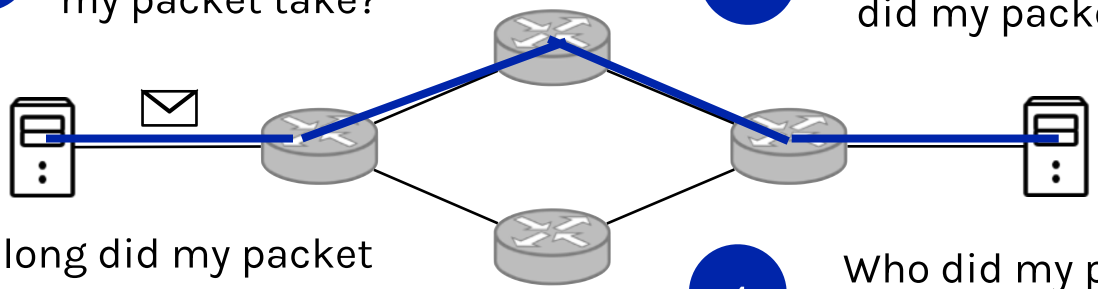
ping, traceroute

Use data plane network management tools like ping/traceroute to diagnose network issues

Per-packet network monitoring

I visited switch 1 @720ns, switch 7 @1.8us

In switch 1, I followed rules 39 and 102. In switch 9…



1 Which path did my packet take?

2 Which rules on the switch did my packet follow?

3 How long did my packet queue at each switch?

4 Who did my packet share the queue with?

Delay: 100ns, 300ns, 10200ns…

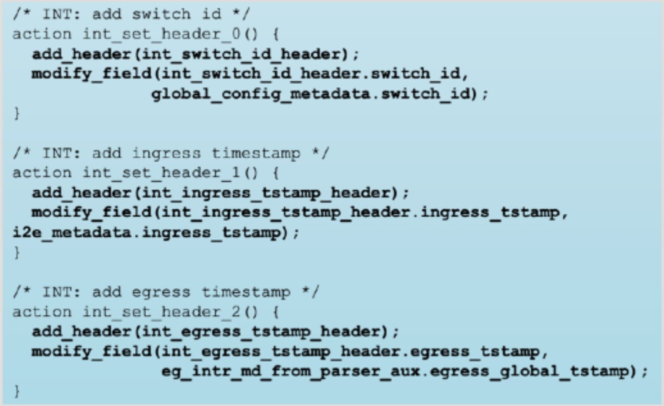
Flow 1: src1->dst1, Flow 2: src2->dst2…

How can we obtain such per-packet information in real time?

In-band network telemetry (INT) with programmable data plane

Leverage the programmability of switches to insert monitoring information in the packet header along the network path

Use P4 to implement logic on switches to insert the switch ID, the ingress timestamp, the egress time stamp, and queue information in the packet header.

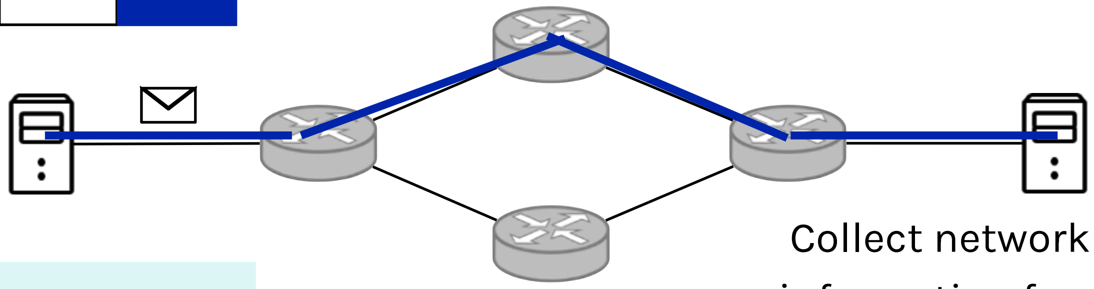


```
/* INT: add switch id */
action int_set_header_0() {
  add_header(int_switch_id_header);
  modify_field(int_switch_id_header.switch_id,
               global_config_metadata.switch_id);
}

/* INT: add ingress timestamp */
action int_set_header_1() {
  add_header(int_ingress_tstamp_header);
  modify_field(int_ingress_tstamp_header.ingress_tstamp,
i2e_metadata.ingress_tstamp);
}

/* INT: add egress timestamp */
action int_set_header_2() {
  add_header(int_egress_tstamp_header);
  modify_field(int_egress_tstamp_header.egress_tstamp,
                eg_intr_md_from_parser_aux.egress_global_tstamp);
}
```

ETH	IP	TCP	INT



Collect network monitoring information from the packet header at the receiver side

Can we monitor the network directly in the data plane?

Learning objectives

What **data structures** we typically use for network monitoring?

How to perform **heavy hitter detection** in the programmable data plane?

What data structures are typically used for network monitoring?

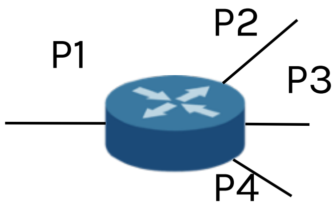
Membership detection

130.83.164.11
130.83.165.12
130.83.165.24
...

Access Control List (ACL)

Decides if an IP address is in the block list

240.0.0.5 → {P1,P3}
240.0.0.6 → {P1,P2}
240.0.0.7 → {P2,P3}
240.0.0.8 → {P1,P2,P3}



IP Multicast

Decides if a router port should replicate a packet

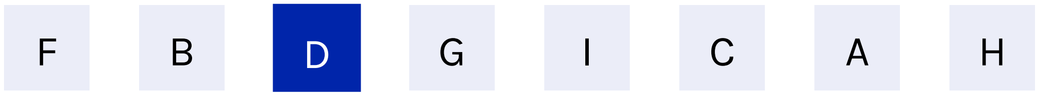
10.0.2.10 → S1
10.0.3.10 → S2
10.0.4.10 → S3

Load Balancer

Decides if a source IP has been assigned to a server

Trivial solutions

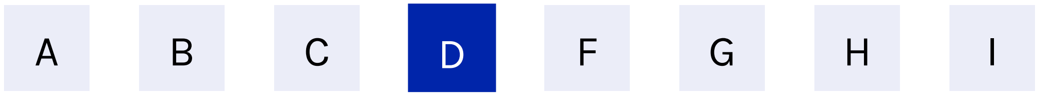
Unordered list:



F B D G I C A H

Linear search: $O(n)$ time where n is the number of elements

Ordered list:



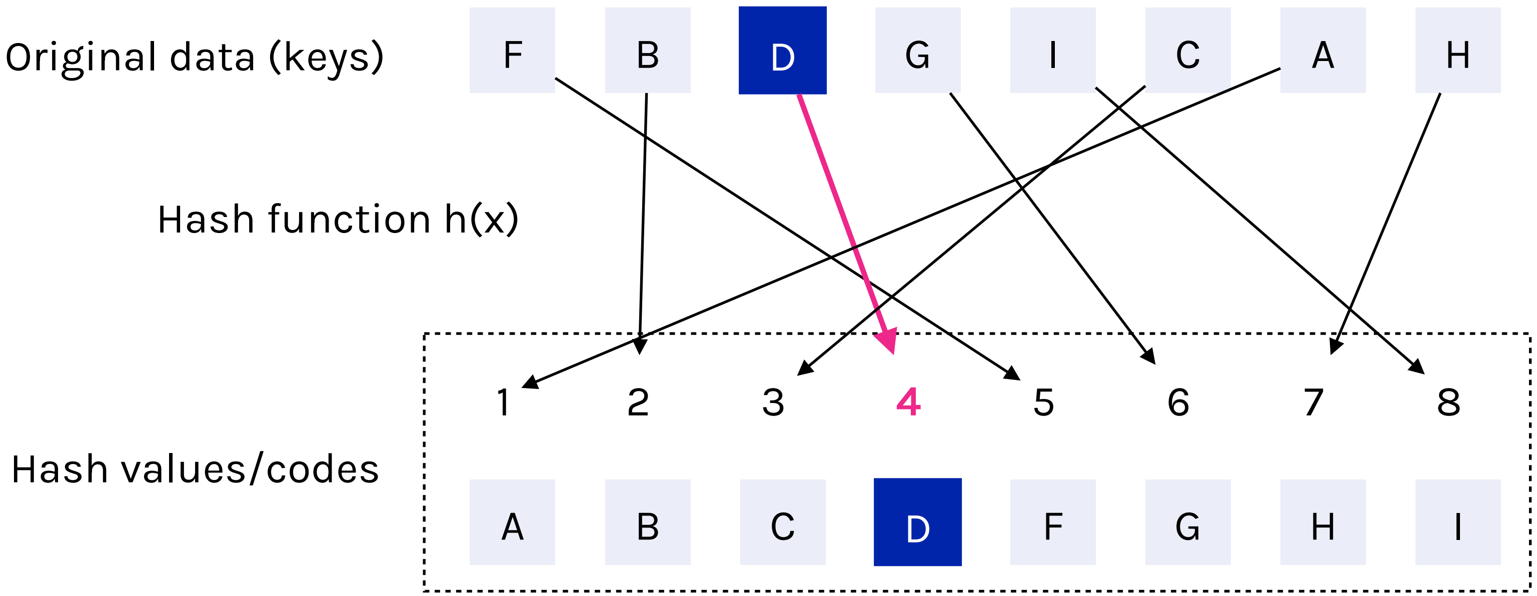
A B C D F G H I

Binary search: $O(\log n)$ time where n is the number of elements

Can we achieve constant time $O(1)$ search?

Hashing

Mapping data (of arbitrary size) to fixed-size values (indices here) with a function, sometimes also called scattered storage addressing



Original data (keys)

F B D G I C A H

Hash function h(x)

Hash values/codes

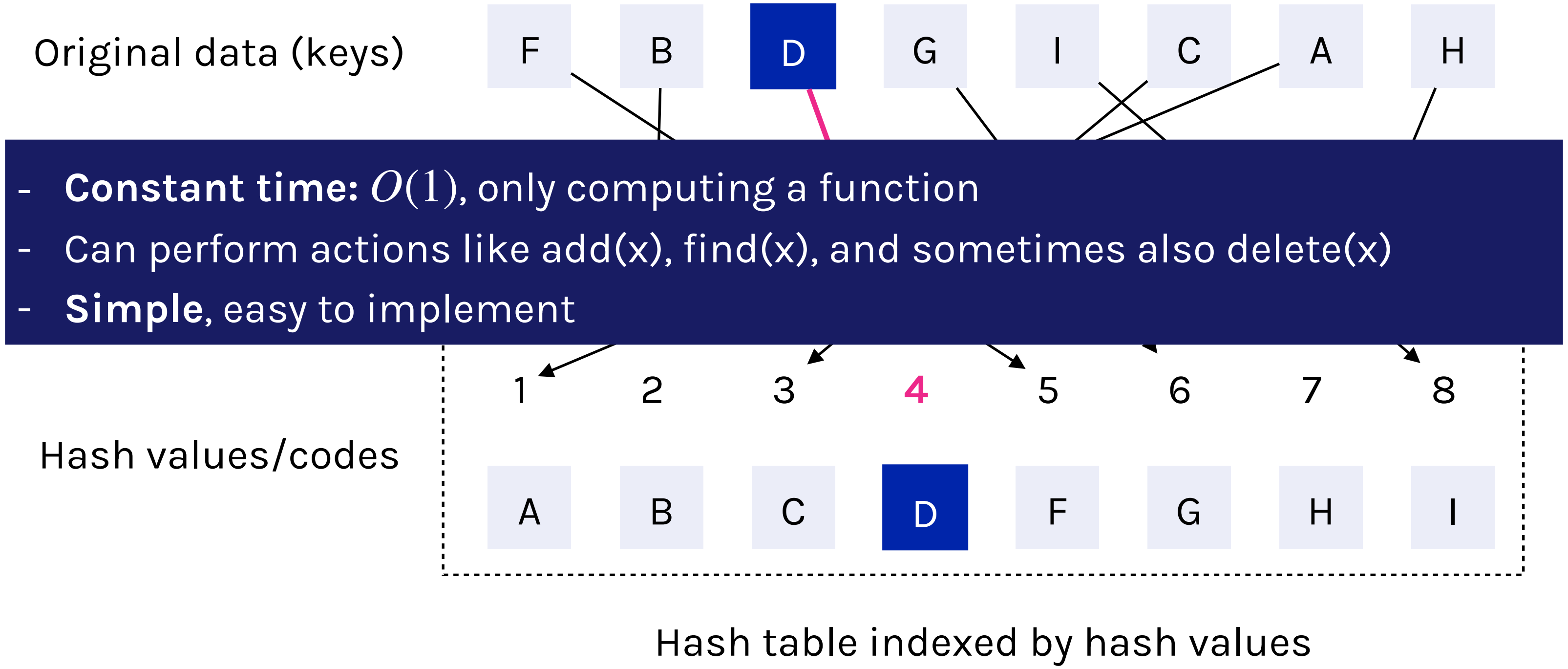
1 2 3 4 5 6 7 8

A B C D F G H I

Hash table indexed by hash values

Hashing

Mapping data (of arbitrary size) to fixed-size values (indices here) with a function, sometimes also called scattered storage addressing



- **Constant time:** $O(1)$, only computing a function
- Can perform actions like add(x), find(x), and sometimes also delete(x)
- **Simple**, easy to implement

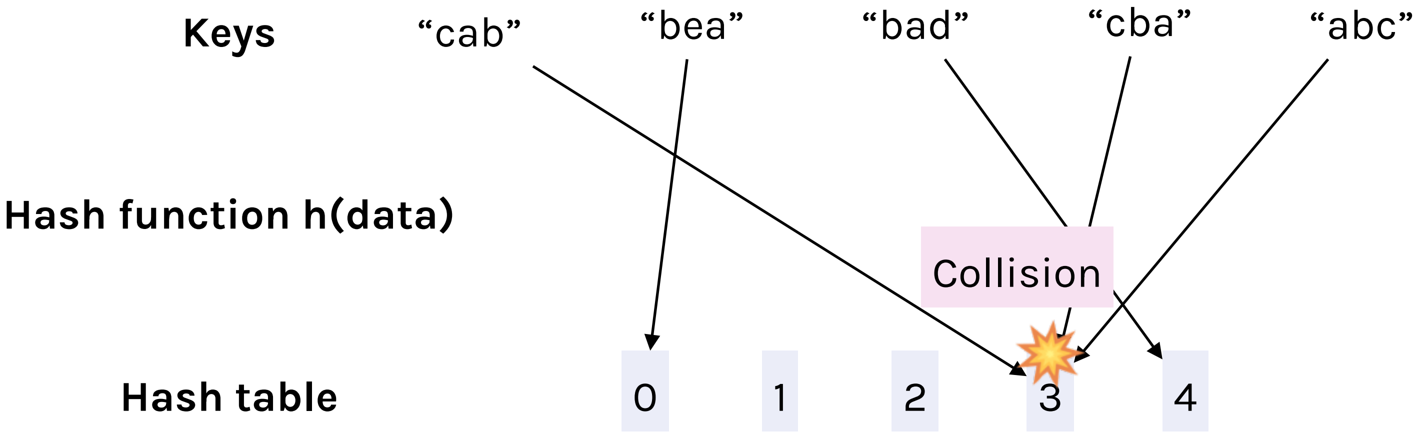
Hash collision

Describes the case where multiple data entries are mapped to the same hash value

Let a = 0, b = 1, c = 2, …

Hash function: h(data) = (∑ characters) mod table_size

table_size: size of the hash table



How can we solve or mitigate this issue?

Properties of good hash functions

Must return numbers: {0,..., table_size}

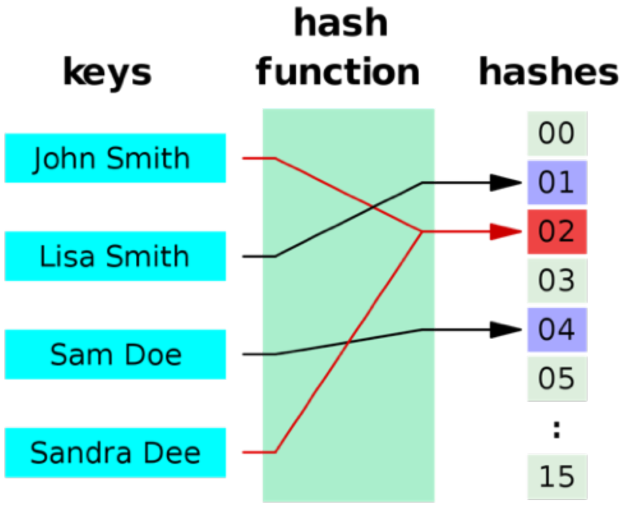
Must be deterministic: always returns the same value for the same key

Should be efficiently computable: $O(1)$ time

Should not waste space unnecessarily:

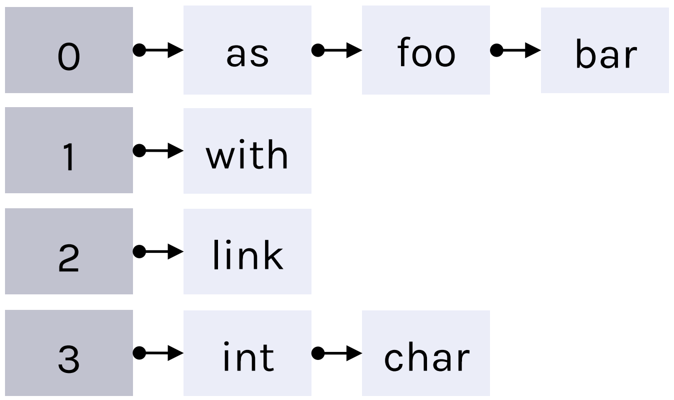
- For every index, there is at least one key that hashes to it
- Load factor lambda = (# of keys) / table_size

Should minimize collisions: keys are nicely spread out

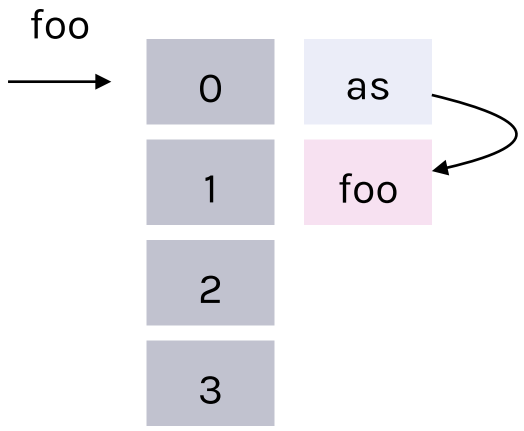


Handling hash collisions

Designing a data structure that can resolve hash collisions



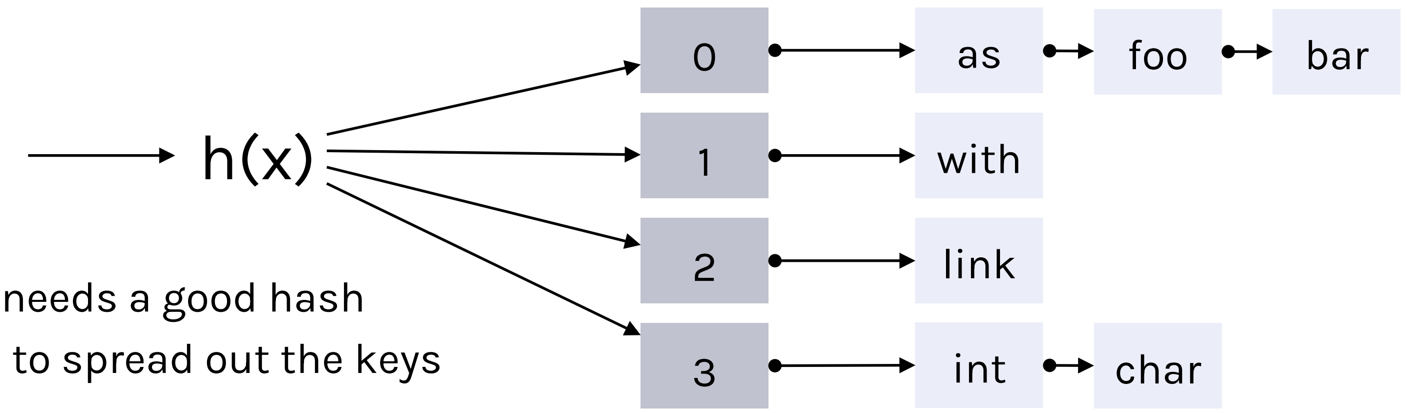
Separate chaining



Open addressing (linear/quadratic probing/cuckoo hashing)

Separate chaining

Creating a list of keys that map to the same hash value



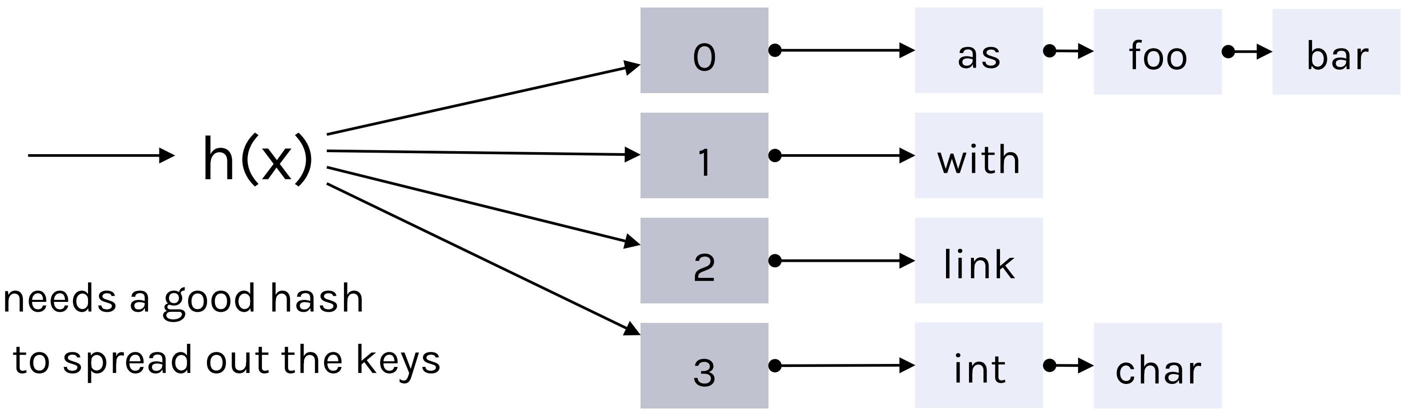
Still needs a good hash function to spread out the keys

A list of keys maintained in a linked list for each hash value

What are the consequences to the hashing performance?

Separate chaining

Creating a list of keys that map to the same hash value

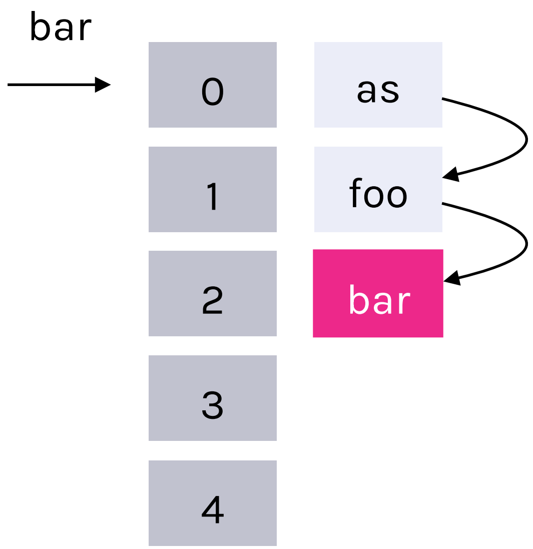


Still needs a good hash function to spread out the keys

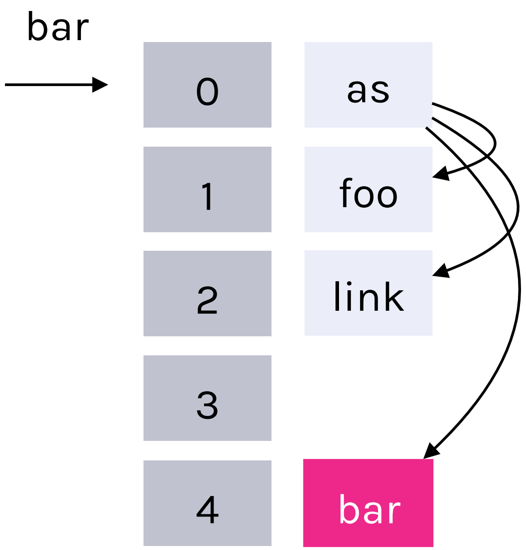
A list of keys maintained in a linked list for each hash value

Lookup time: average case $O(N/\text{table\_size})$, worse case $O(N)$
(N is the total number of keys)

Open addressing



Linear probing (offset = 1, 2, 3,…)

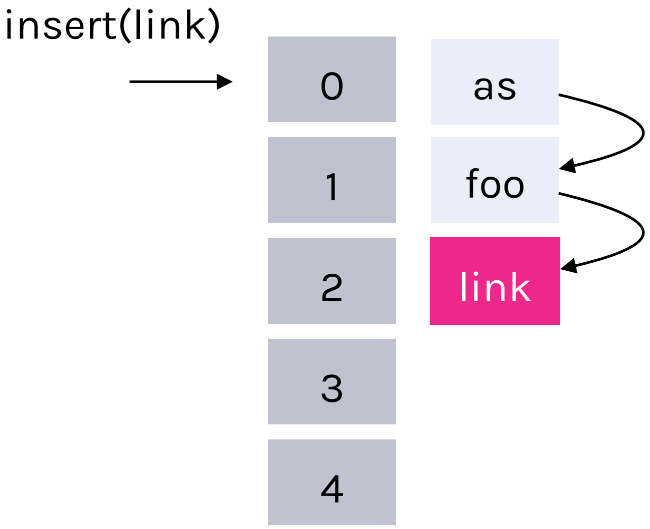


Quadratic probing (offset = 1, 4, 9,…)

Open addressing: linear probing

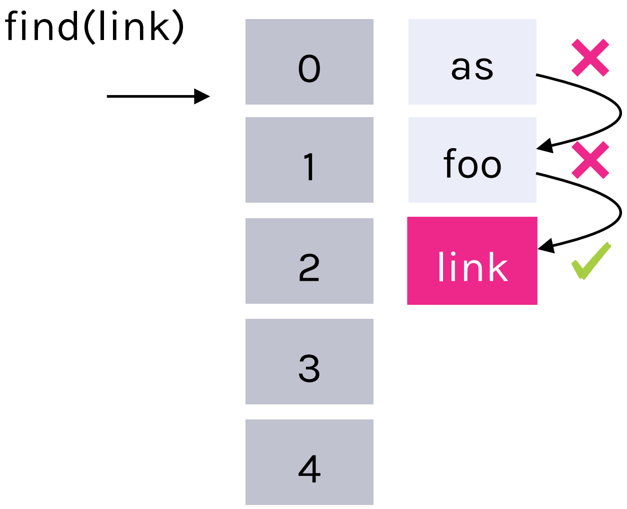
Probing with a linear offset: 1, 2, 3,…

Insertion



Upon collision, inset(x) finds the first slot after $h(x)$ that is empty and inserts x in that slot

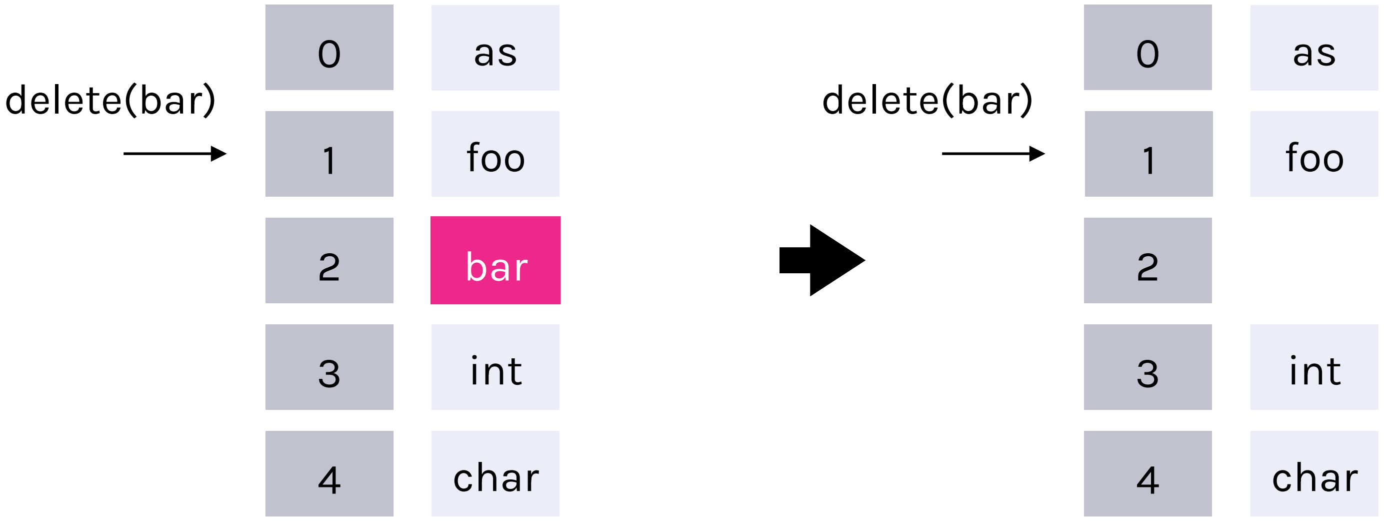
Lookup



Keep checking from $h(x)$ until x is found in the hash table; does not exist if hitting an empty slot before x is found

How to handle delete(x) operations?

Handling deletion operations in linear probing



delete(bar)

0	as
1	foo
2	bar
3	int
4	char

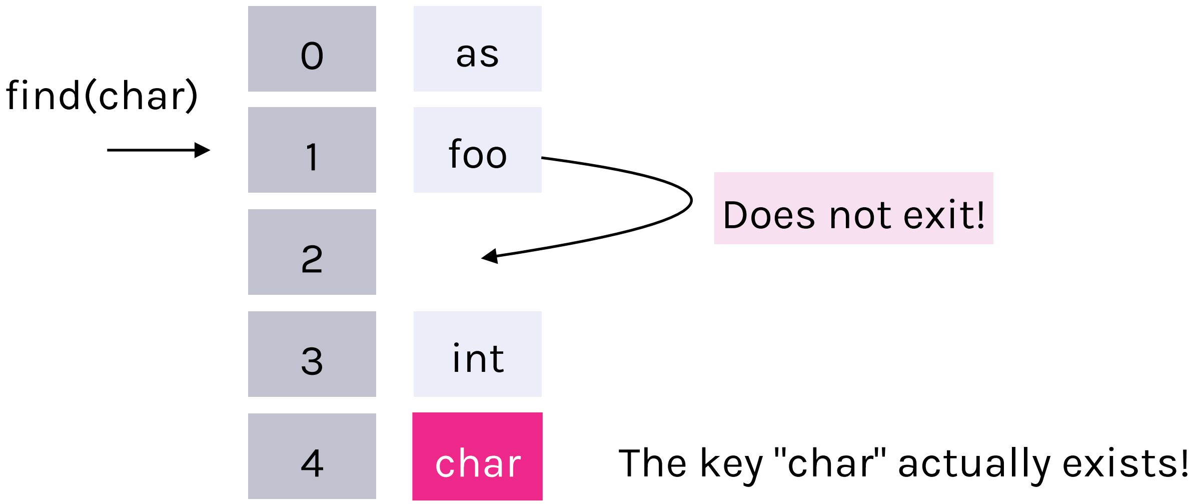
delete(bar)

0	as
1	foo
2	
3	int
4	char

Is this correct?

Handling deletion operations in linear probing

Assume h(char) = 1



find(char)

0	as
1	foo
2	
3	int
4	char

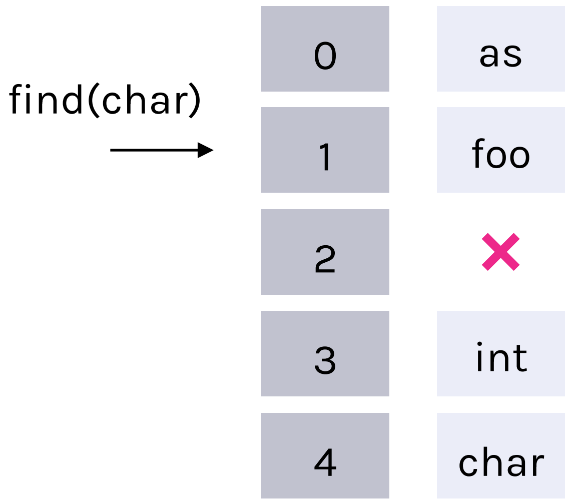
Does not exit!

The key "char" actually exists!

Problem: there are dependencies in locating the different keys in the hash table

Handling deletion operations in linear probing

Assume h(char) = 1

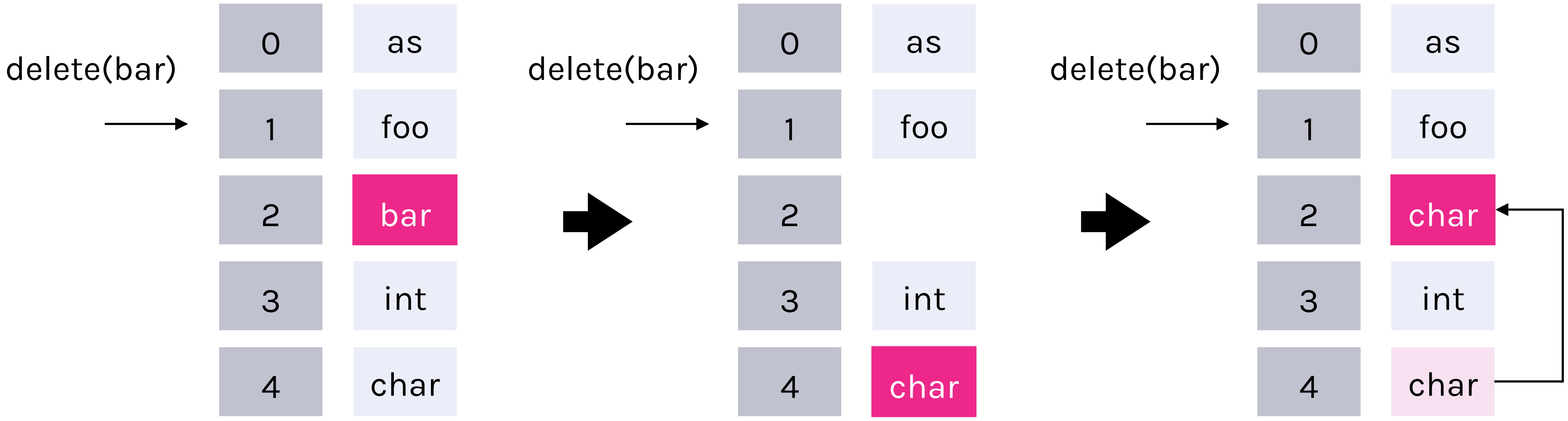


find(char)

0	as
1	foo
2	✗
3	int
4	char

Maintain a flag of “deleted” for the emptied slots; adds in lookup time overhead

Handling deletion operations in linear probing



delete(bar)

0	as
1	foo
2	bar
3	int
4	char

Probe linearly to find the slot containing the target

delete(bar)

0	as
1	foo
2	
3	int
4	char

Delete the target; keep probing and find a key that is **movable** to the empty slot

delete(bar)

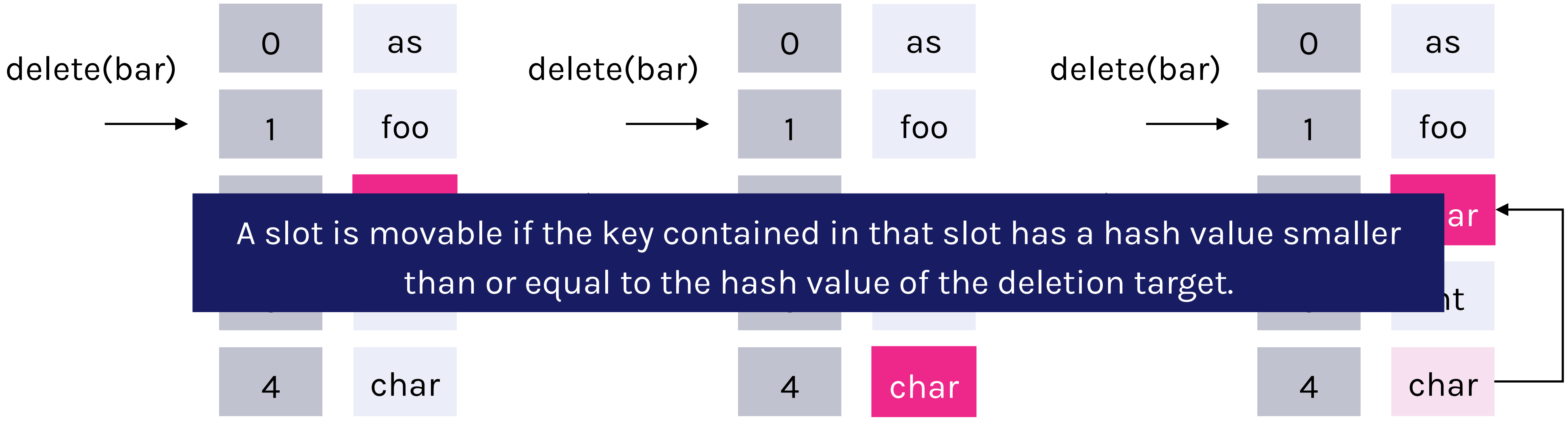
0	as
1	foo
2	char
3	int
4	char

Move the found key to the empty slot

Repeat the process until an empty slot is hit

What defined a slot movable?

Handling deletion operations in linear probing



A slot is movable if the key contained in that slot has a hash value smaller than or equal to the hash value of the deletion target.

Probe linearly to find the slot containing the target

Delete the target; keep probing and find a key that is **movable** to the empty slot

Move the found key to the empty slot

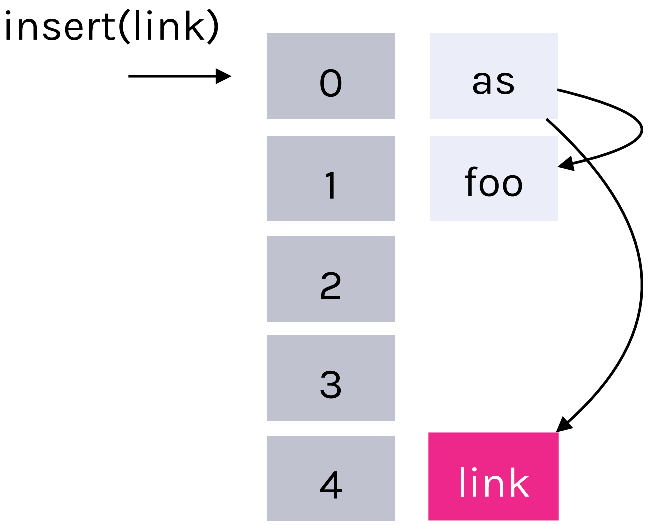
Repeat the process until an empty slot is hit

What defined a slot movable?

Open addressing: quadratic probing

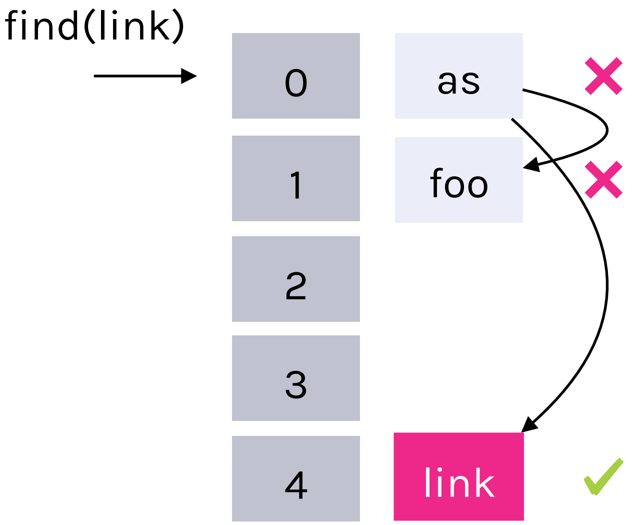
Probing with a quadratic offset: 1, 4, 9,…

Insertion



Upon collision, inset(x) finds the first slot after $h(x)$ that is empty with a quadratic offset and inserts x in that slot

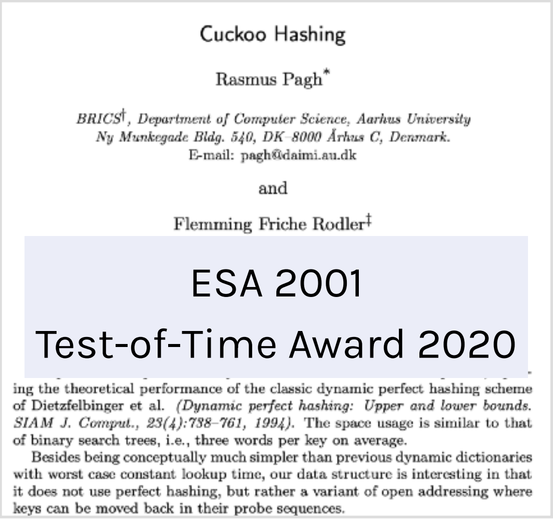
Lookup



Keep checking from $h(x)$ with a quadratic offset until x is found in the hash table; does not exist if hitting an empty slot before x is found

Open addressing: cuckoo hashing

Pushing other keys to a different location upon collisions



Cuckoo Hashing

Rasmus Pagh[*]

BRICS[†], Department of Computer Science, Aarhus University
Ny Munkegade Bldg. 540, DK–8000 Århus C, Denmark.
E-mail: pagh@daimi.au.dk

and

Flemming Friche Rodler[‡]

ESA 2001

Test-of-Time Award 2020

ing the theoretical performance of the classic dynamic perfect hashing scheme of Dietzfelbinger et al. *(Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput., 23(4):738–761, 1994).* The space usage is similar to that of binary search trees, i.e., three words per key on average.

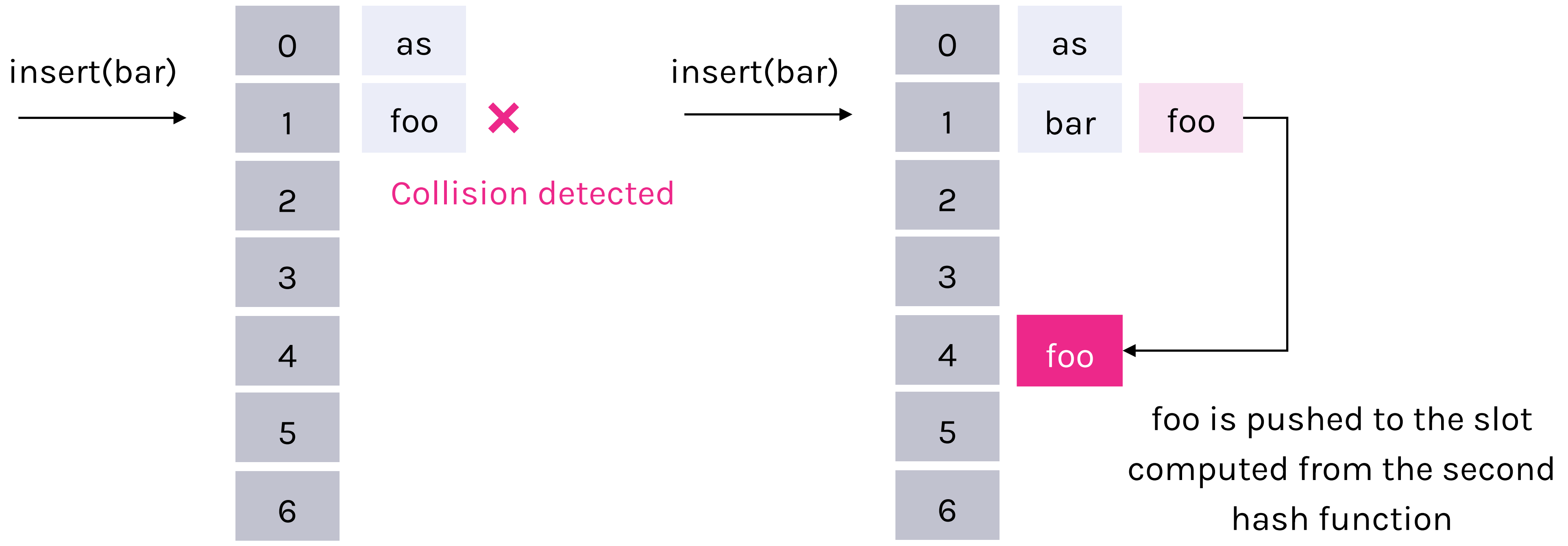
Besides being conceptually much simpler than previous dynamic dictionaries with worst case constant lookup time, our data structure is interesting in that it does not use perfect hashing, but rather a variant of open addressing where keys can be moved back in their probe sequences.

The name is derived from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches.

Cuckoo hashing

Using two hash functions to generate two possible slots for each key

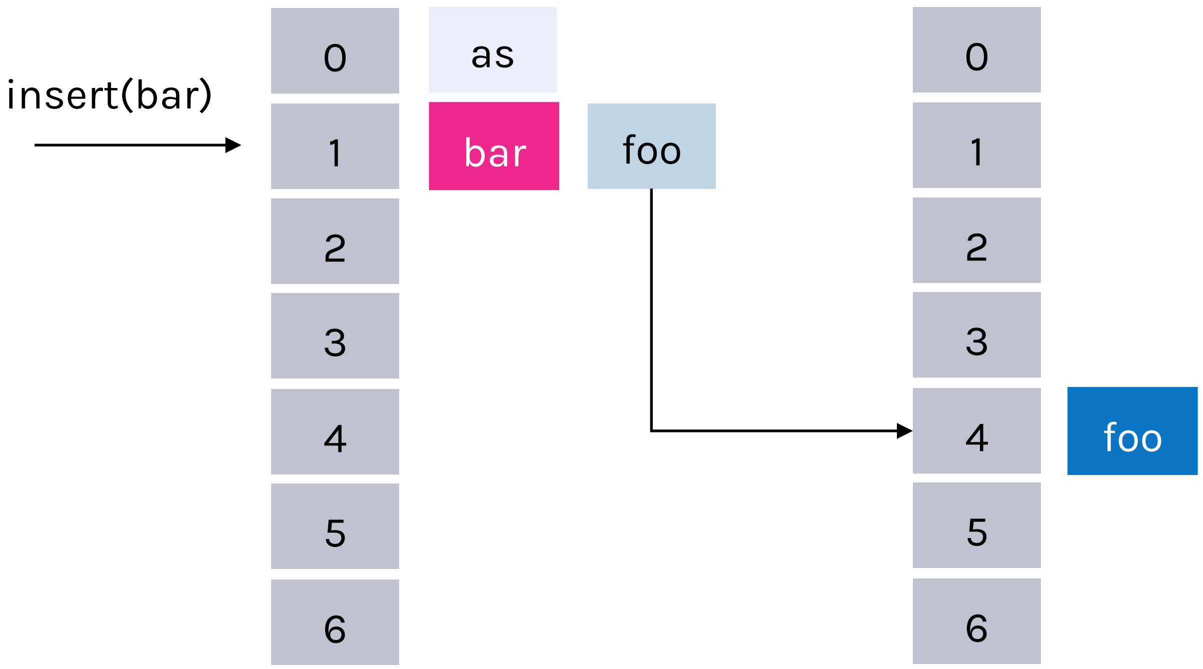
h1(foo) = 1, h2(foo) = 4, h1(bar) = 1, h2(bar) = 5



Cuckoo hashing implementation

Typically using two separate hash tables, each indexed by one hash function

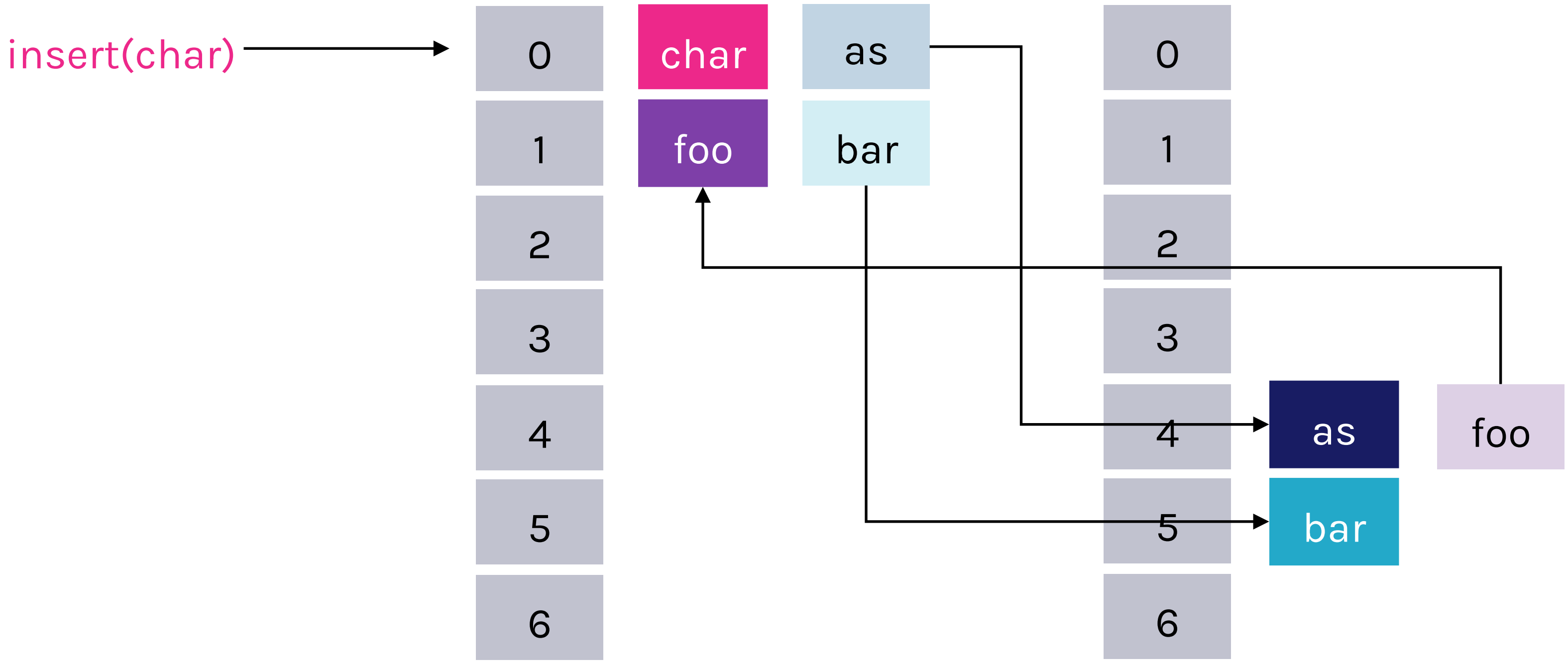
h1(foo) = 1, h2(foo) = 4, h1(as) = 0, h2(as) = 4, h1(bar) = 1, h2(bar) = 5



Cuckoo hashing operations

Insertion takes more time than lookup and deletion

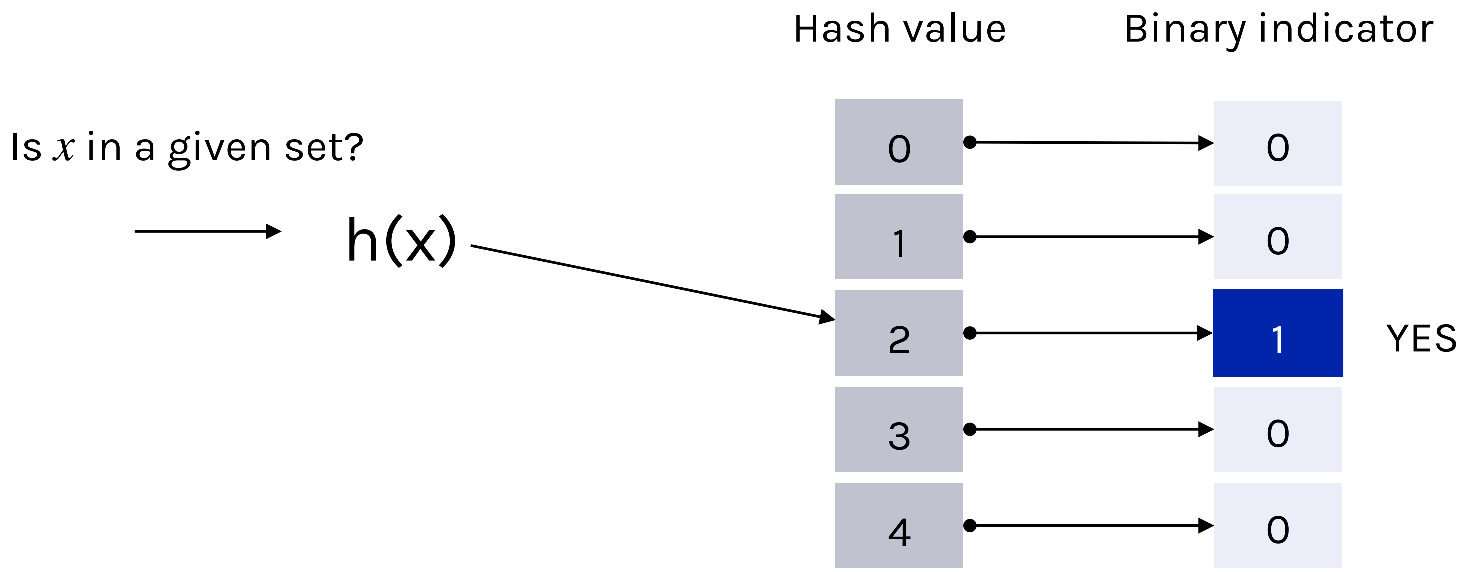
h1(foo) = 1, h2(foo) = 4, h1(as) = 0, h2(as) = 4, h1(bar) = 1, h2(bar) = 5, h1(char) = 0, h2(char) = 2



Insertion time worse case $O(N)$, lookup time $O(1)$, deletion time $O(1)$

Membership determination with hashing

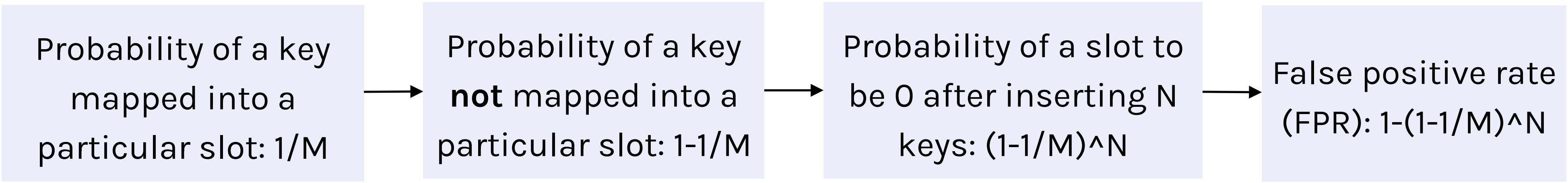
Assume we do not have enough space to store all the keys, but we want to answer membership determination queries



Set the binary indicator to 1 at insertion; return true if the binary indicator is 1 at lookup.

False positive rate analysis

Assume we have in total N keys and we use a hash table of M slots



Probability of a key mapped into a particular slot: 1/M

Probability of a key **not** mapped into a particular slot: 1-1/M

Probability of a slot to be 0 after inserting N keys: (1-1/M)^N

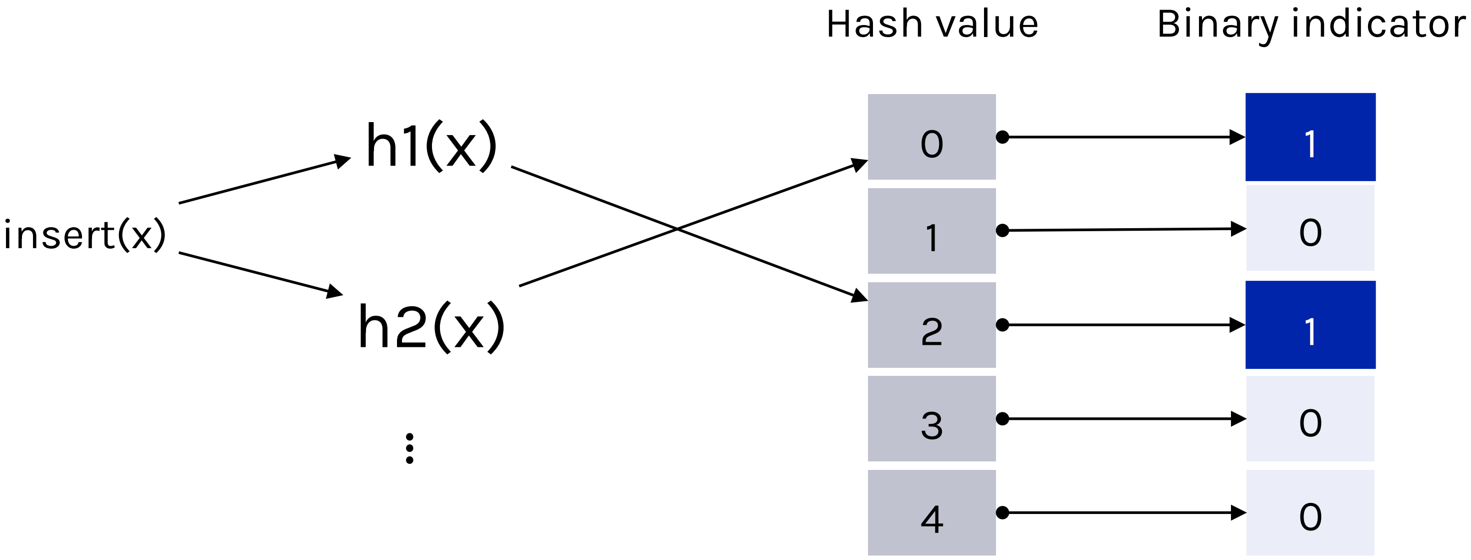
False positive rate (FPR): 1-(1-1/M)^N

# of keys	# of slots	FPR
1000	10,000	9.5%
1000	100,000	1%

Roughly 100x number of slots is required to have an FPR lower than 1%.

Bloom filter

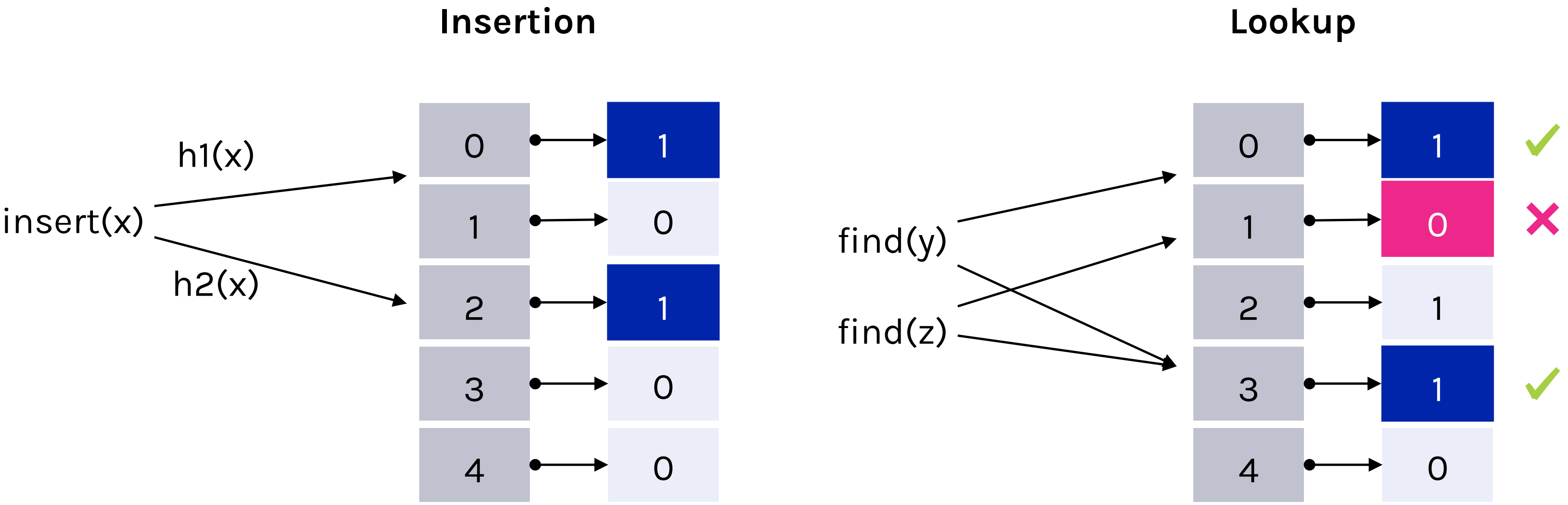
Typically using multiple hash functions to lower collision rate



Hash value	Binary indicator
0	1
1	0
2	1
3	0
4	0

Bloom filter: insertion and lookup

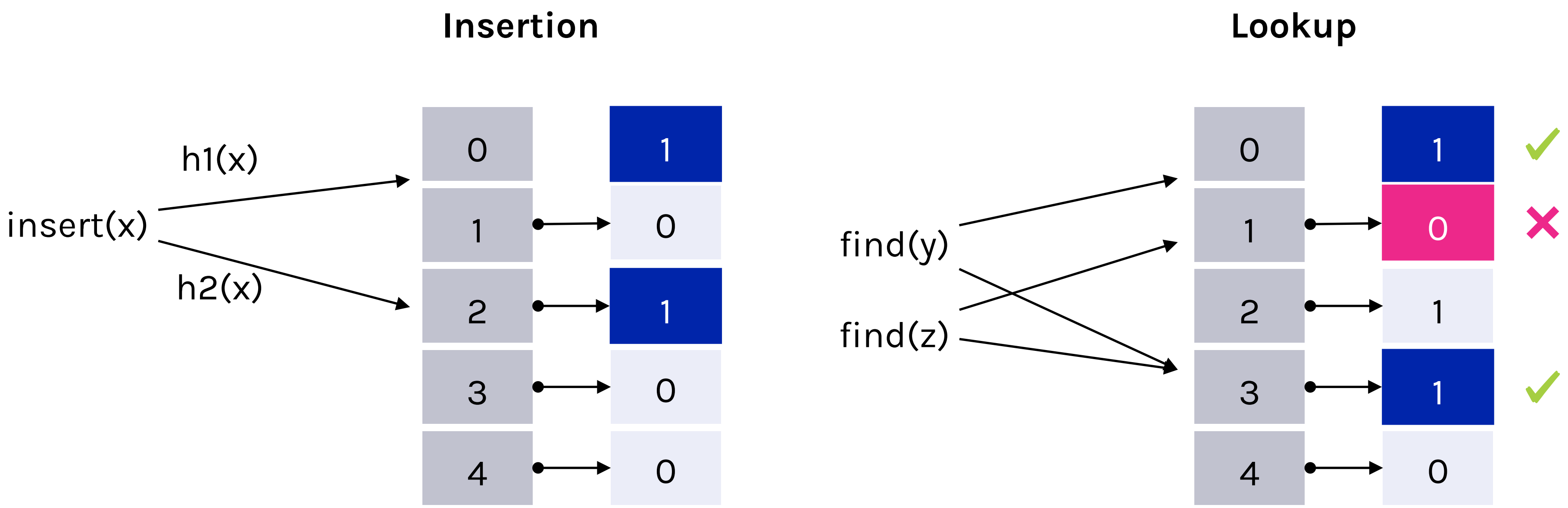
Setting the binary indicators corresponding to the hash values from the input to 1 if 0



Can we delete a key from the Bloom filter?

Bloom filter: insertion and lookup

Setting the binary indicators corresponding to the hash values from the input to 1 if 0



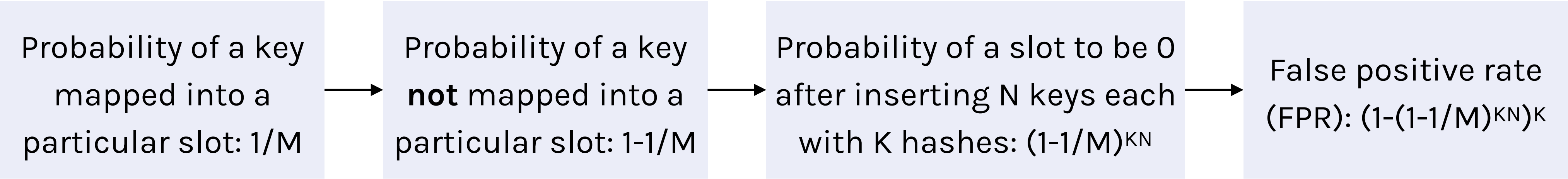
Insertion

Lookup

A basic Bloom filter does not support deletion since the indicators may be shared by other keys.

False positive rate analysis

Assume we have N keys and we use a Bloom filter of M slots with K hash functions



Probability of a key mapped into a particular slot: 1/M

Probability of a key **not** mapped into a particular slot: 1-1/M

Probability of a slot to be 0 after inserting N keys each with K hashes: $(1-1/M)^{KN}$

False positive rate (FPR): $(1-(1-1/M)^{KN})^{K}$

# of keys	# of slots	# of hash functions	FPR
1000	10,000	7	0.82%
1000	100,000	7	≈0%

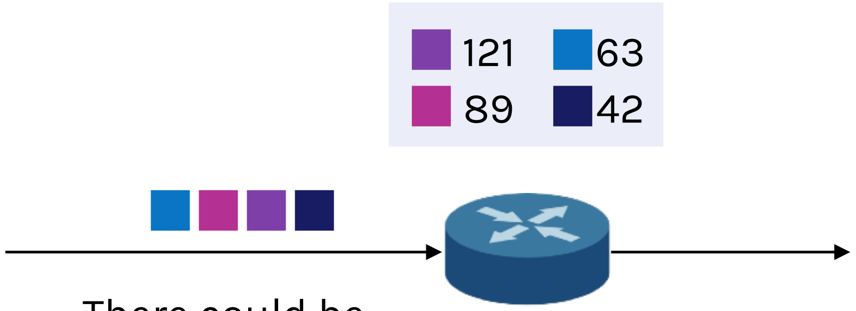
Consumes almost 10x less space than the single-hash case, but requires slightly more computation for the operations.

How to efficiently count the occurrences for a large set of elements?

Example: heavy hitter detection

Detecting the top-K flows (in terms of traffic volume, #packets) that have passed through a given router

A flow is defined by a 5-tuple: <src_ip, dst_ip, src_port, dst_port, protocol>

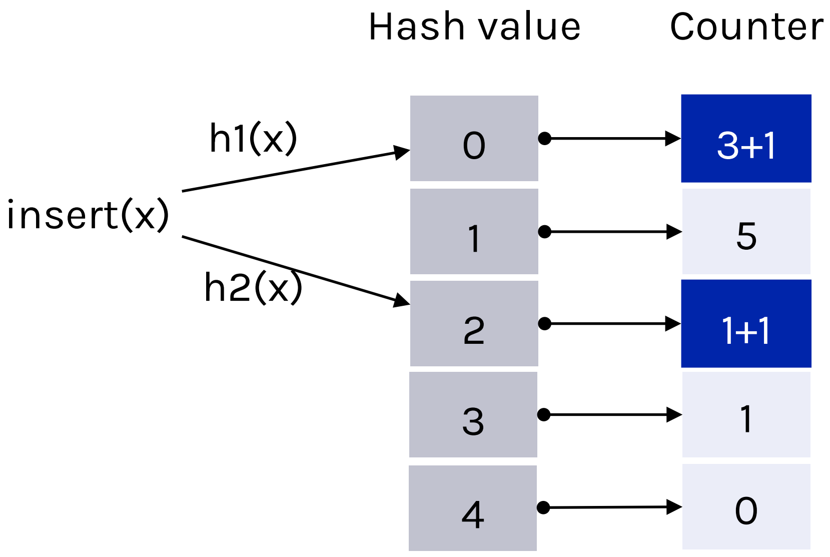


There could be 1000s of flows

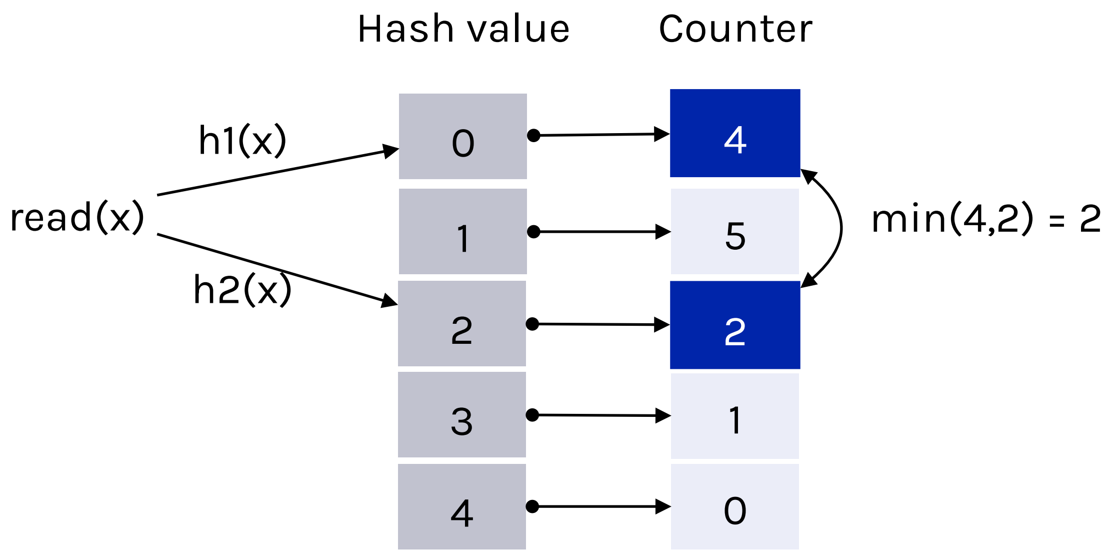
Routers are resource-limited, so creating **counters** for each separate flow is not scalable.

Counting Bloom filter

Extension to Bloom filter that can count the occurrences of keys



Increment the counters corresponding to the hash values

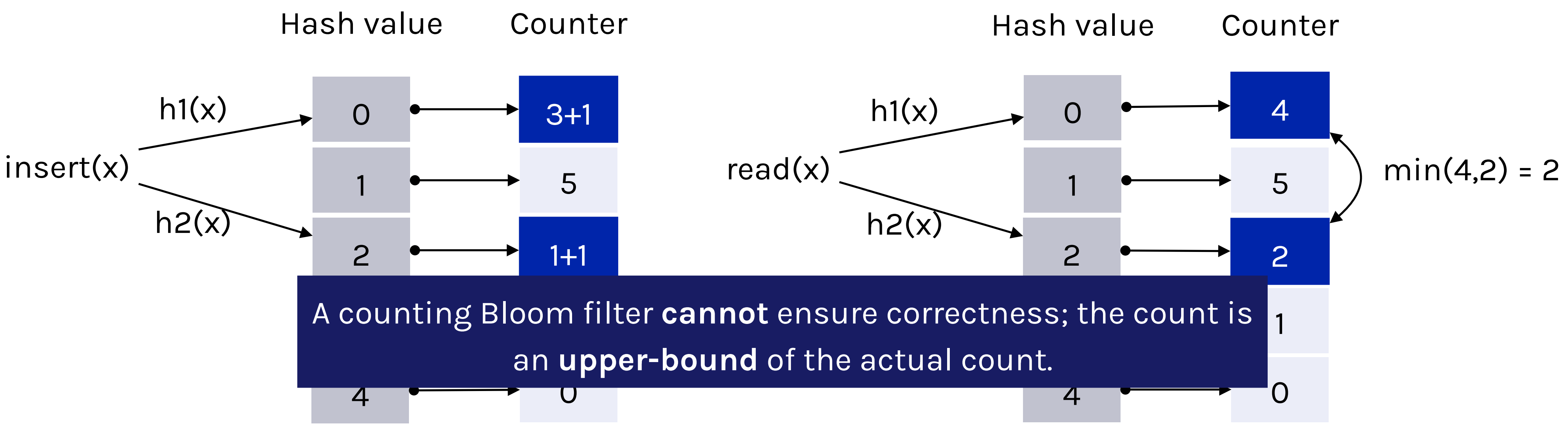


Lookup the counters corresponding to the hash values with the minimum count

Is the count always correct? If not, what guarantees do we have?

Counting Bloom filter

Extension to Bloom filter that can count the occurrences of keys



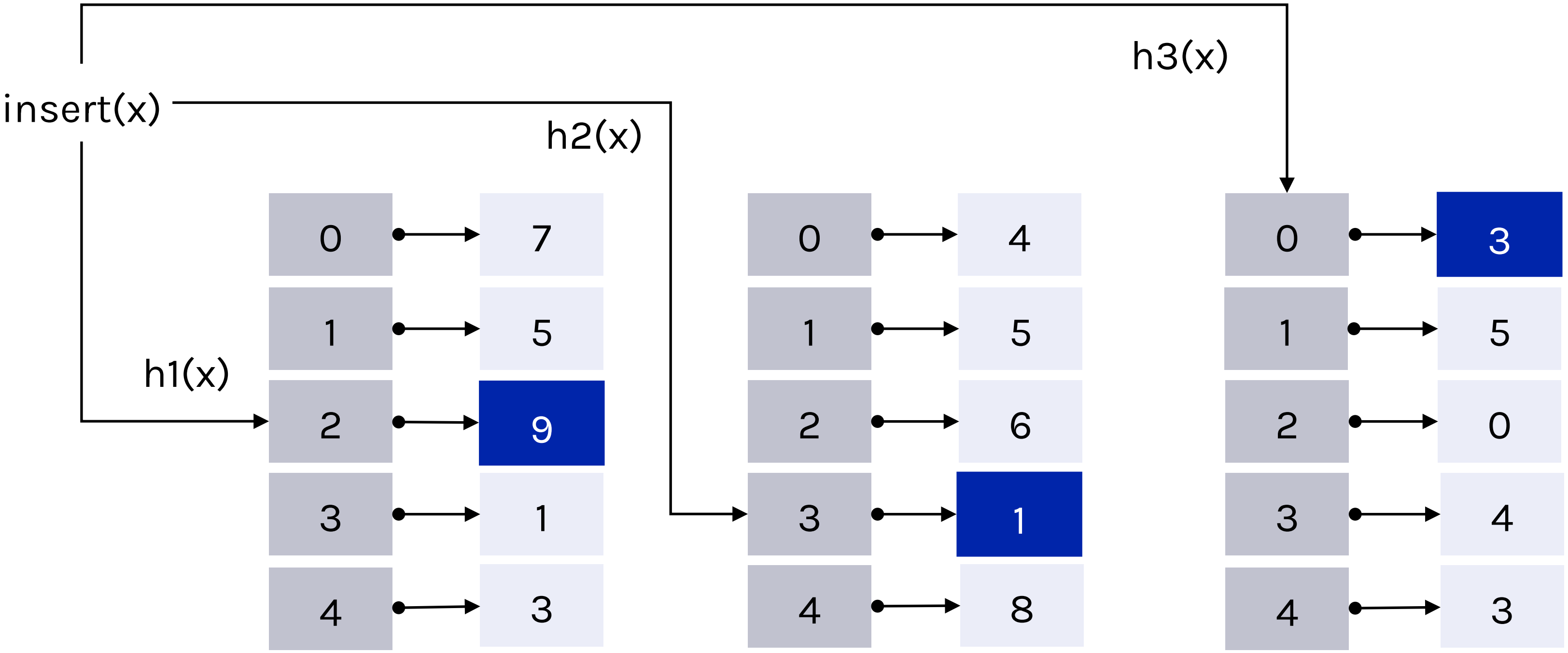
A counting Bloom filter **cannot** ensure correctness; the count is an **upper-bound** of the actual count.

Increment the counters corresponding to the hash values

Lookup the counters corresponding to the hash values with the minimum count

Count-min sketch

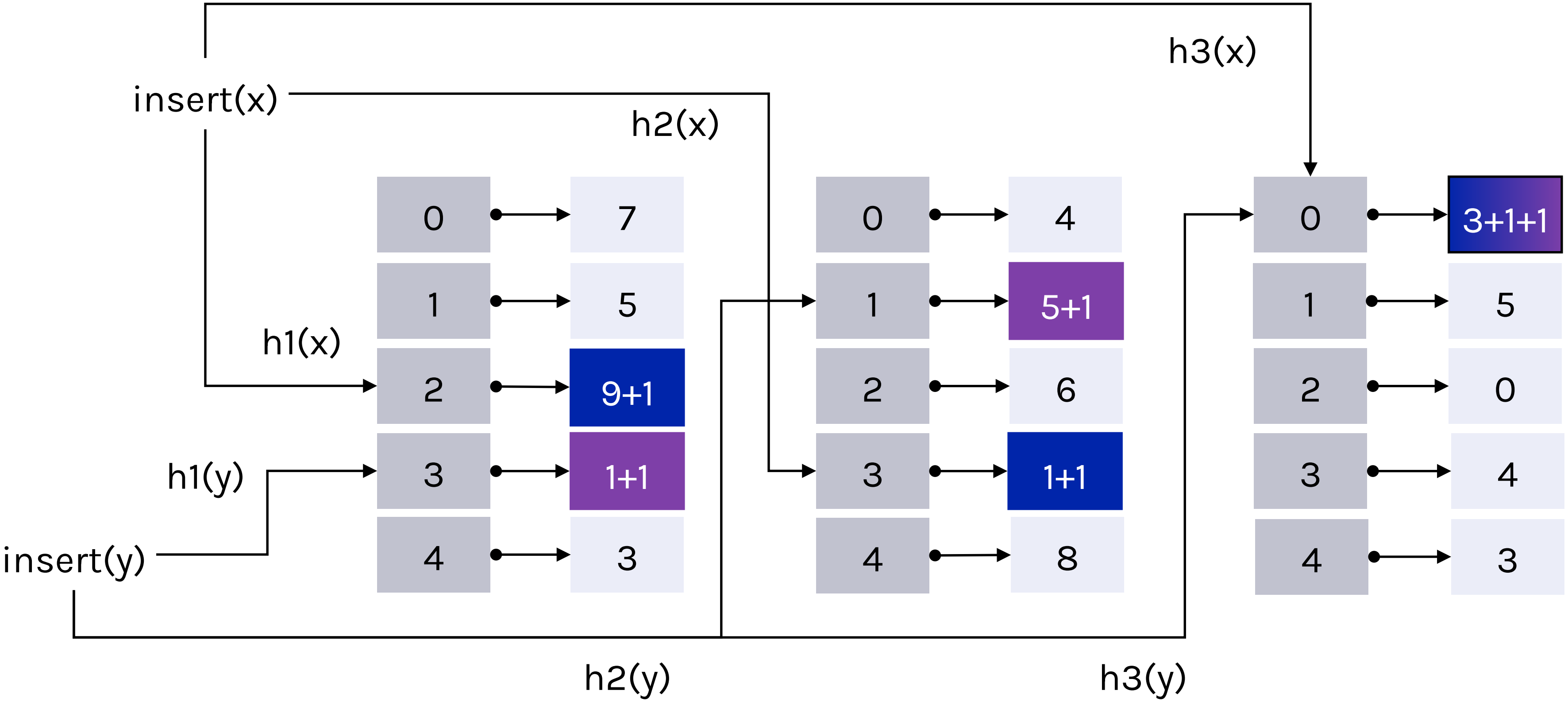
A slight improvement to the counting Bloom filter



Three hash functions are performed, each mapped to an array of counters (hash tables).

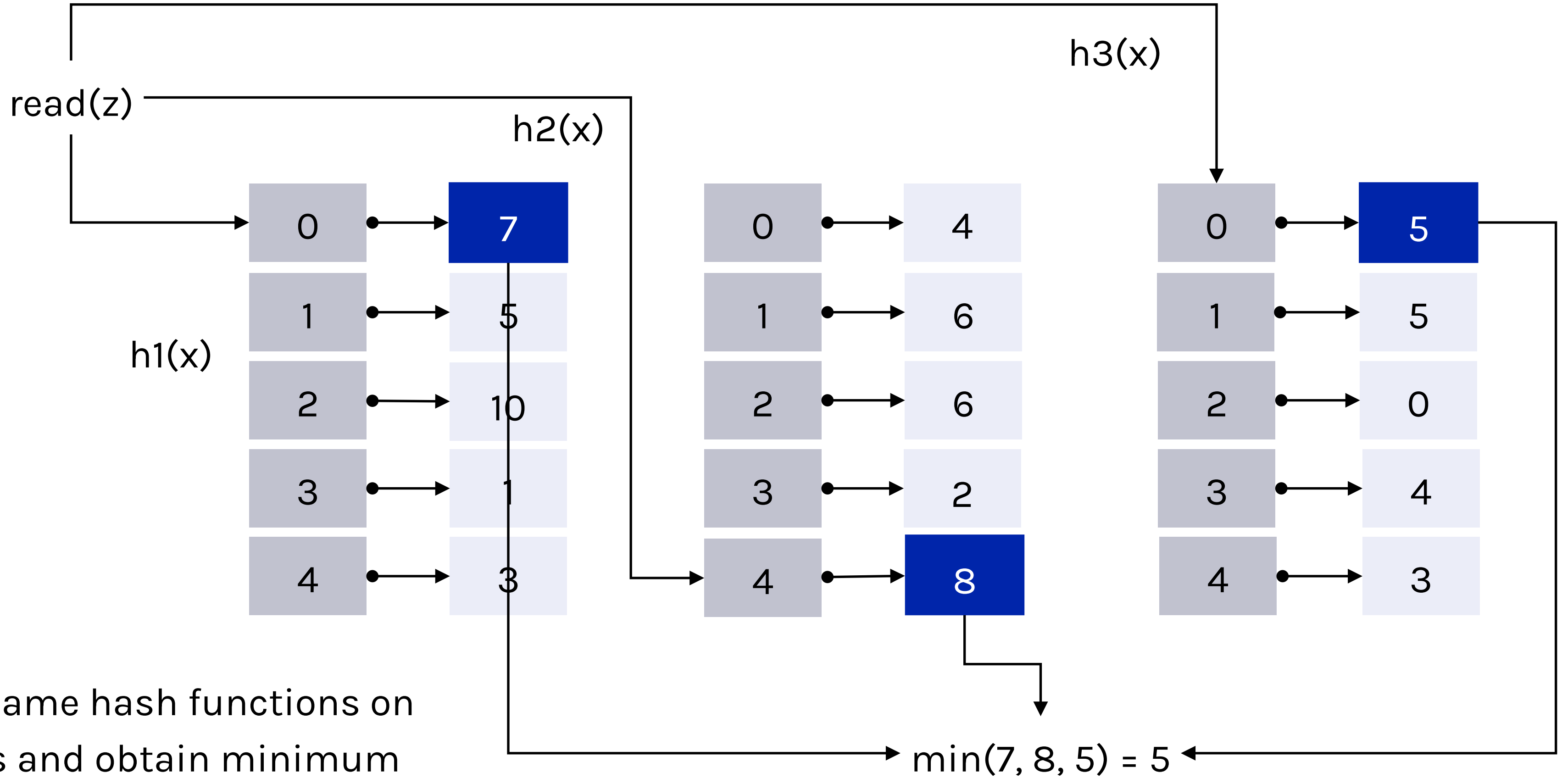
Count-min sketch

Incrementing the counters for the computed hash values



Count-min sketch

How to read the count from the count-min sketch?

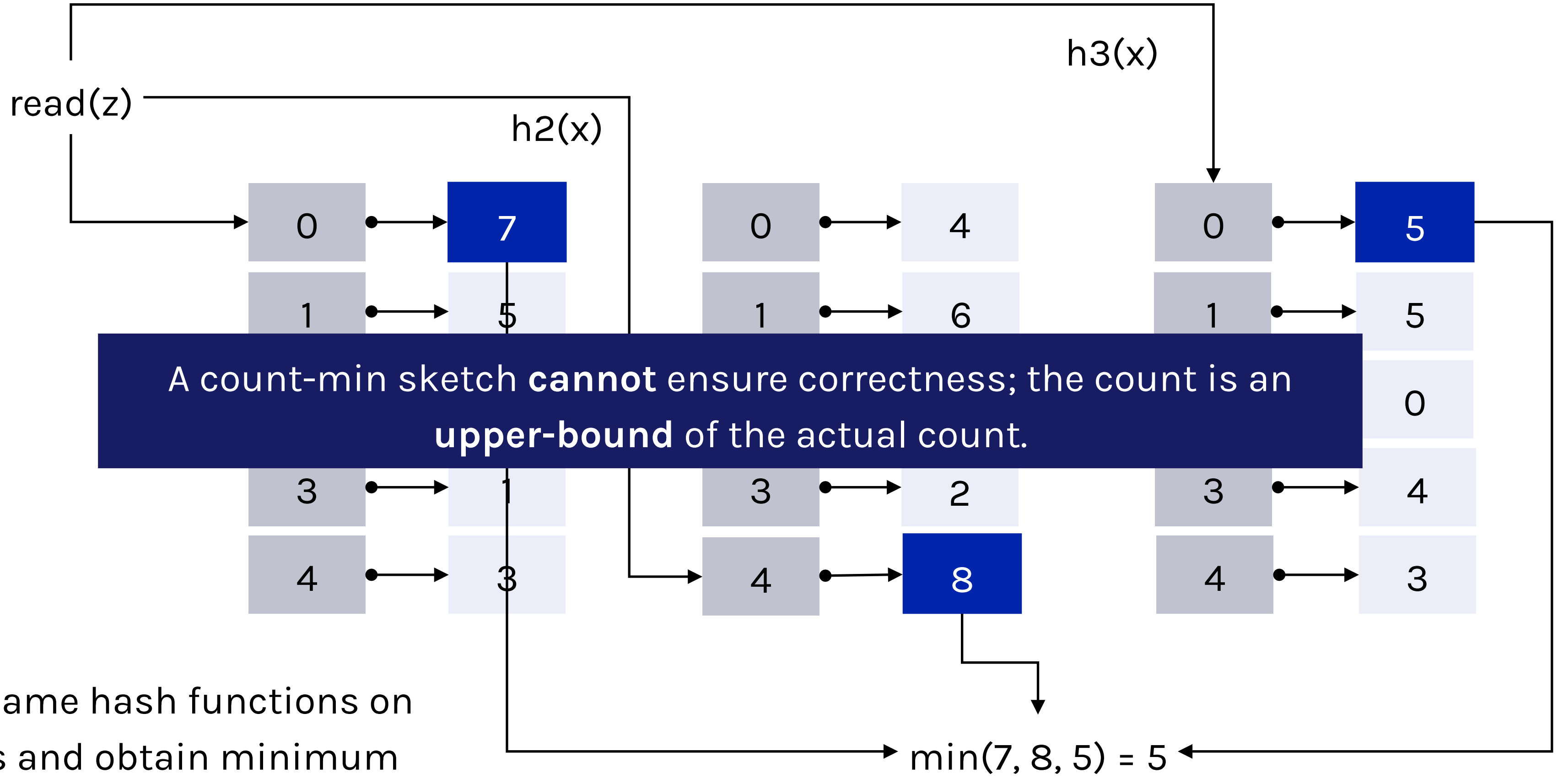


0	7	0	4	0	5
1	5	1	6	1	5
2	10	2	6	2	0
3	1	3	2	3	4
4	3	4	8	4	3

Perform the same hash functions on all the arrays and obtain minimum of all the counters as output

Count-min sketch

How to read the count from the count-min sketch?



A count-min sketch **cannot** ensure correctness; the count is an **upper-bound** of the actual count.

Perform the same hash functions on all the arrays and obtain minimum of all the counters as output

How to perform heavy hitter detection in programmable data plane?

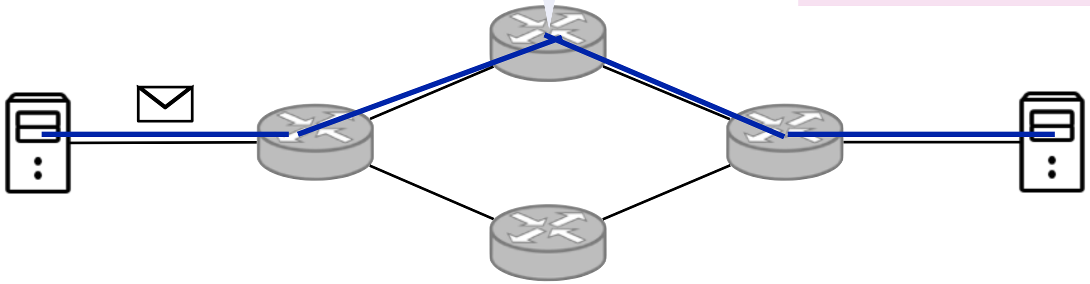
Heavy hitters

Network flows that are larger (in number of packets or bytes) than a fraction t of the total packets seen on the link or the top k flows by size

Flow is defined by combinations of packet header fields, e.g., 5-tuple (src_ip, dst_ip, src_port, dst_port, protocol).

Flow-1: 1750
Flow-2: 1320
Flow-3: 800
…

Number of flows could be tens of thousands and higher



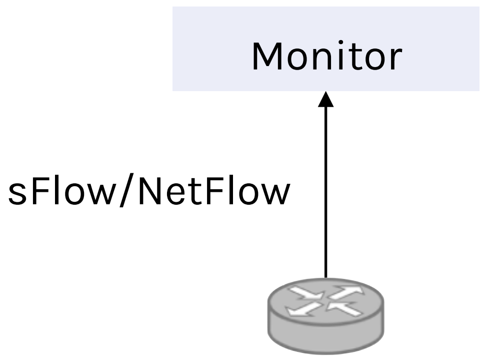
Challenge: finer-grained flows → larger size and number of keys → more bits to represent the key and more entries to track

Design goals and constraints

Accuracy: false positives (reporting a non-heavy flow as heavy), false negatives (not reporting a heavy flow), error in estimating the sizes of heavy flows

Overhead: total amount of memory for the data structure, the number of matching stages uses in the switch pipeline

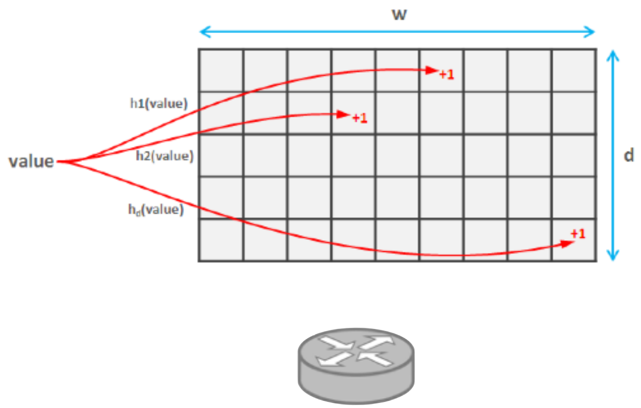
Existing solutions



Monitor

sFlow/NetFlow

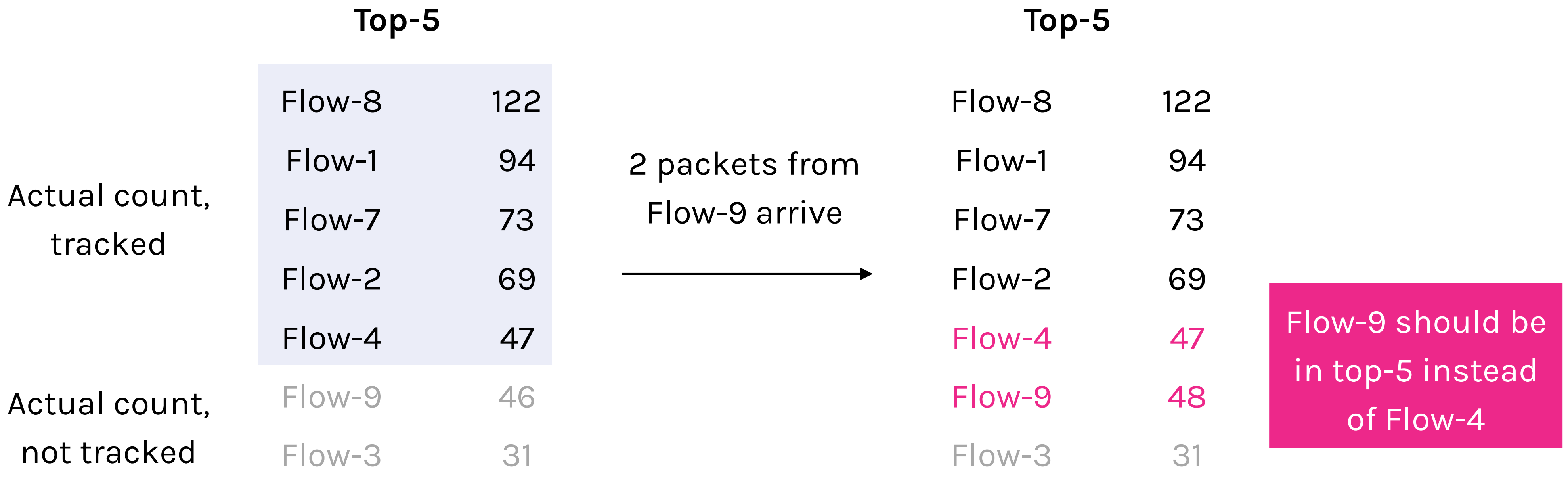
Packet sampling: use aggressive flow sampling range (1% or 0.01%) → **low accuracy**



Streaming algorithms: use count-min / count sketches → **does not track flow entities**

Can we simply use O(k) counters?

Assume we aim to obtain the top-k heavy flows



	Top-5	
Actual count, tracked	Flow-8	122
	Flow-1	94
	Flow-7	73
	Flow-2	69
	Flow-4	47
Actual count, not tracked	Flow-9	46
	Flow-3	31

2 packets from Flow-9 arrive →

Top-5	
Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-4	47
Flow-9	48
Flow-3	31

Flow-9 should be in top-5 instead of Flow-4

The space-saving algorithm

A counter-based algorithm that uses O(k) counters to track k heavy flows

Top-5

Actual count, tracked

Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-4	47

1 packet from Flow-7 arrives →

Flow-8	122
Flow-1	94
Flow-7	**73+1**
Flow-2	69
Flow-4	47

1 packet from **Flow-9** arrives →

Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-9	**47+1**

Properties of the space-saving algorithm

Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-4	47
Flow-9	46
Flow-3	31

Property 1: no flow counter in the table is ever underestimated, i.e., c_j <= val_j

Property 2: the minimum value in the table val_r is an upper bound on the overestimation error of any counter, e.g., val_j <= c_j + val_r.

Property 3: any flow with true count higher than the average table count, i.e., c_j >= C/m >= val_min will always be present in the table (C is the total packet count added into the table, m is the number of entries in the table)

Ahmed Metwally, Divyakant Agrawal, Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. International Conference on Database Theory (ICDT), 2005.

Implementing the space-saving algorithm on switches

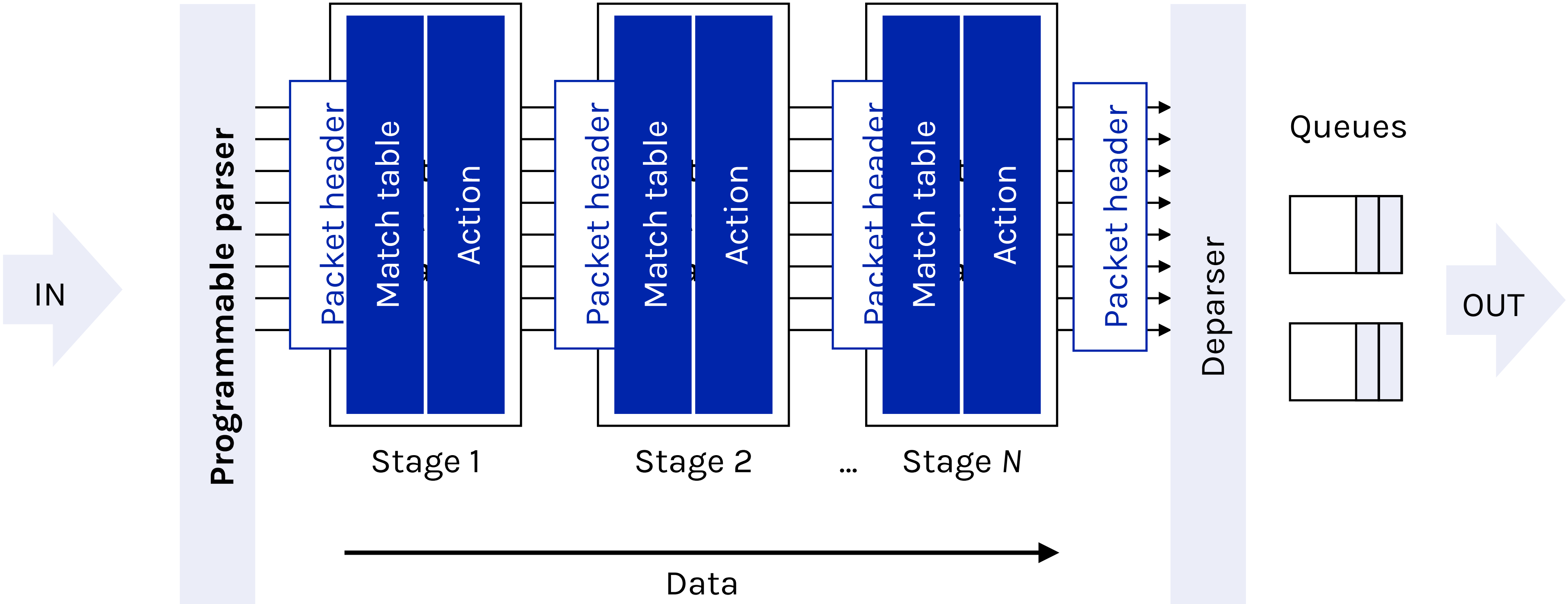
Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-4	47
Flow-9	46
Flow-3	31

If the flow has appeared in the table: hash to the flow key and increment the corresponding counter.

If the flow is not contained in the table: find the minimum counter in the table, replace the key with the current flow key, and increment the counter

How to find the minimum counter in the table?

Recall the RMT architecture



Implementation challenges

Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-4	47

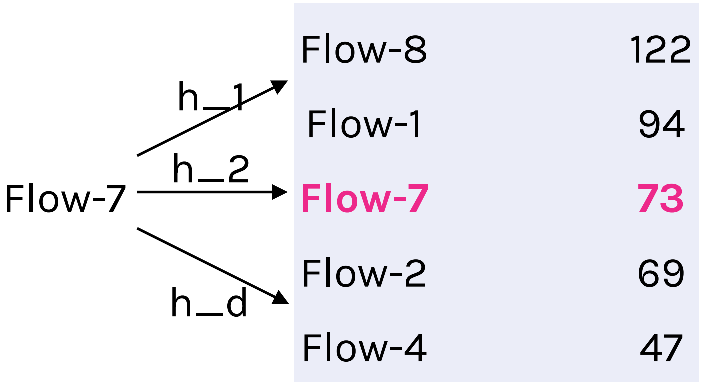
If the flow has appeared in the table: hash to the flow key and increment the corresponding counter.

If the flow is not contained in the table: find the minimum counter in the table, replace the key with the current flow key, and increment the counter

Sorted linked list or priority queue → hard to maintain on switches

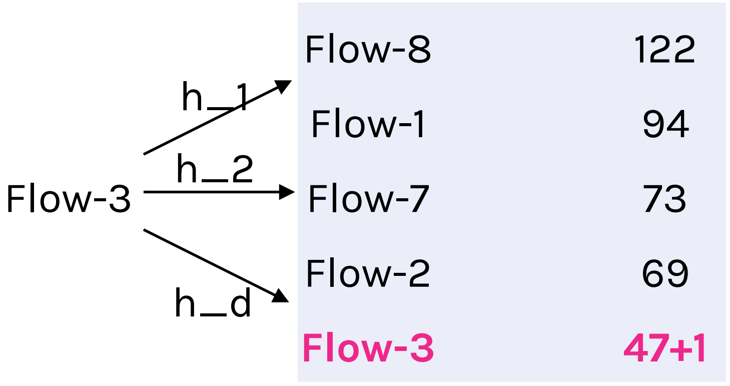
Read k locations, and write back to one location → multiple memory access

Optimization with sampling



Flow-8	122
Flow-1	94
Flow-7	**73**
Flow-2	69
Flow-4	47

If the flow key appears in one of the hashed locations, increment the corresponding counter.



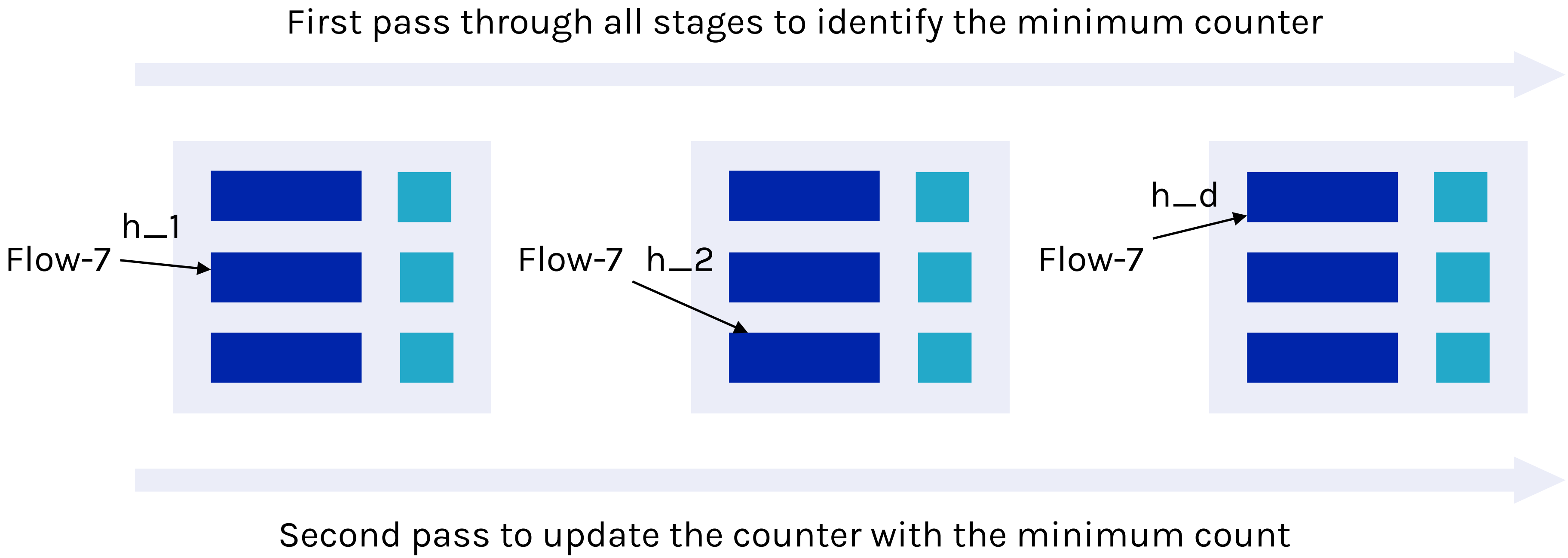
Flow-8	122
Flow-1	94
Flow-7	73
Flow-2	69
Flow-3	**47+1**

Otherwise, choose the smallest counter among the d positions, and replace the key and increment the counter.

Number of memory reads: d, number of memory writes: 1

Optimization with multi-stages

Split the counter table into d stages and read only once per stage



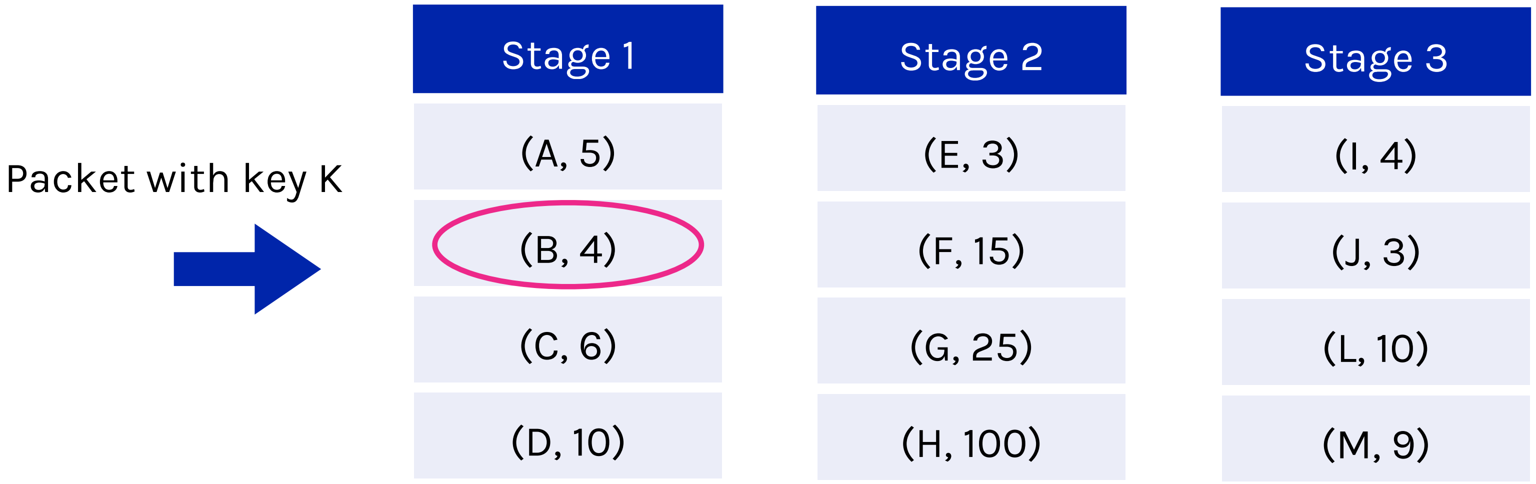
First pass through all stages to identify the minimum counter

Second pass to update the counter with the minimum count

Second pass → packet recirculation for every packet → the bandwidth is halved

HashPipe: feed-forward packet processing

Two key ideas: tracking a rolling minimum and always inserting in the first stage



Packet with key K

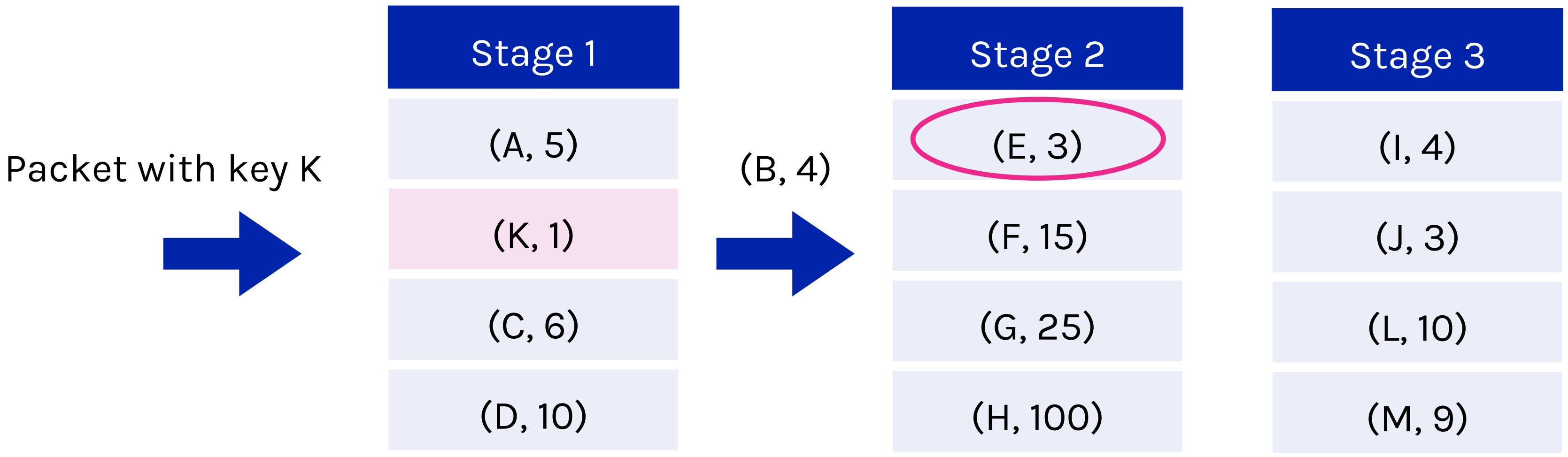
Stage 1	Stage 2	Stage 3
(A, 5)	(E, 3)	(I, 4)
(B, 4)	(F, 15)	(J, 3)
(C, 6)	(G, 25)	(L, 10)
(D, 10)	(H, 100)	(M, 9)

First stage: if key K is a match (or the slot is empty), increment the counter and finish processing; otherwise, always insert the new key with count 1 at the hashed location and carry the old one with the metadata to the next stage

Always insert in the first stage ensures that some duplicate keys can be merged in later stages

HashPipe: feed-forward packet processing

Two key ideas: tracking a rolling minimum and always inserting in the first stage



Packet with key K

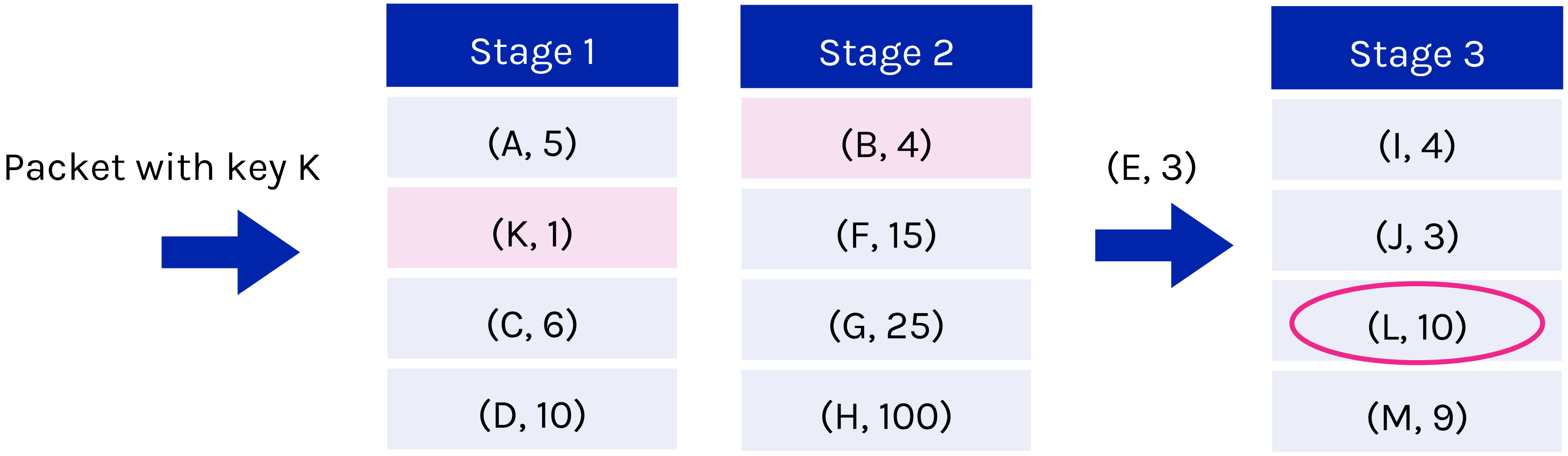
(B, 4)

Stage 1	Stage 2	Stage 3
(A, 5)	(E, 3)	(I, 4)
(K, 1)	(F, 15)	(J, 3)
(C, 6)	(G, 25)	(L, 10)
(D, 10)	(H, 100)	(M, 9)

Later stages: compare the counter at the hashed position (with the key from the metadata) and the counter from the metadata, replace the key-counter in the table if the one carried in the metadata is larger

HashPipe: feed-forward packet processing

Two key ideas: tracking a rolling minimum and always inserting in the first stage



Packet with key K

Stage 1	Stage 2	Stage 3
(A, 5)	(B, 4)	(I, 4)
(K, 1)	(F, 15)	(J, 3)
(C, 6)	(G, 25)	(L, 10)
(D, 10)	(H, 100)	(M, 9)

(E, 3)

Later stages: compare the counter at the hashed position (with the key from the metadata) and the counter from the metadata, replace the key-counter in the table if the one carried in the metadata is larger

HashPipe: feed-forward packet processing

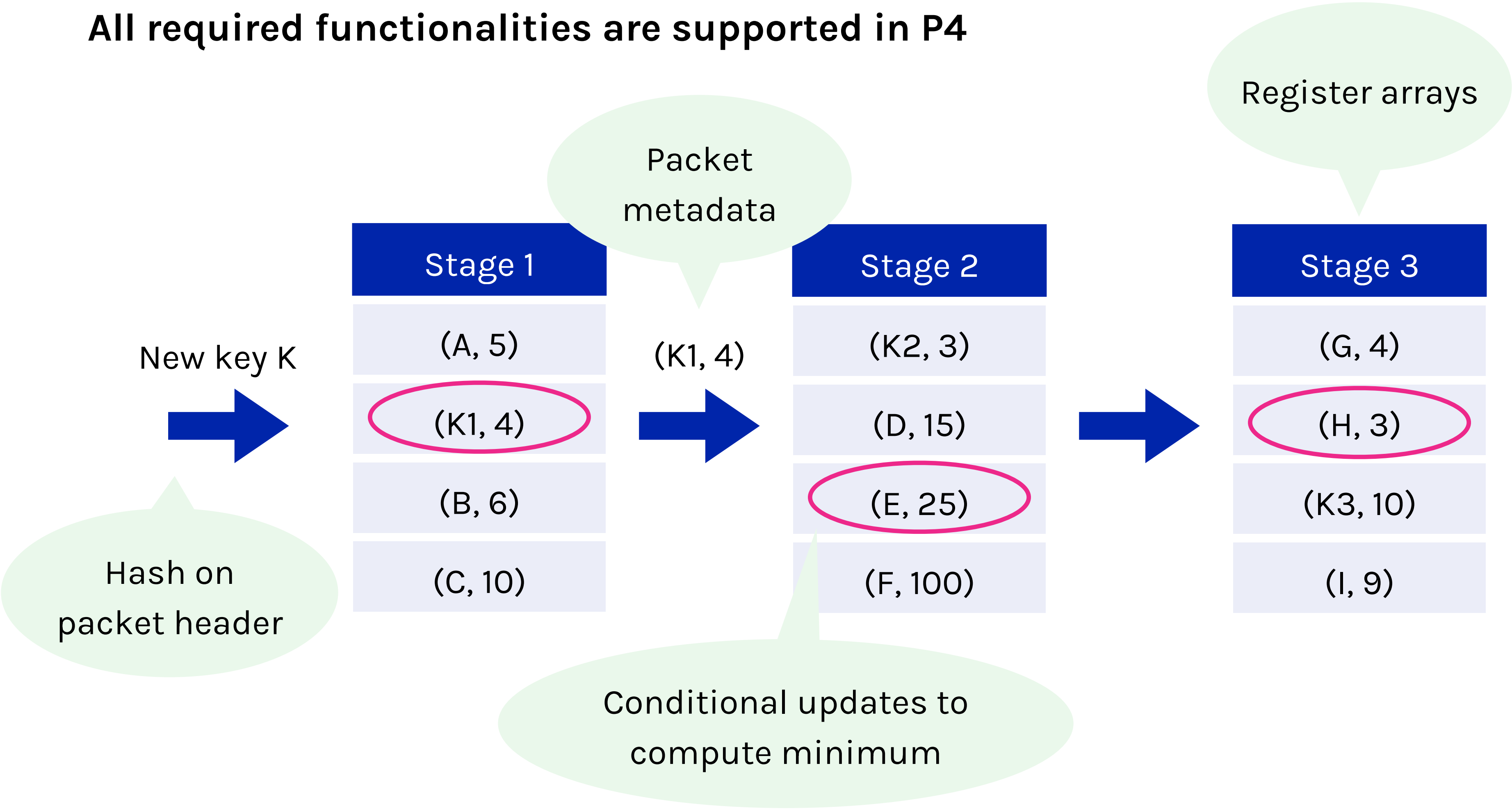
Two key ideas: tracking a rolling minimum and always inserting in the first stage

Stage 1	Stage 2	Stage 3
(A, 5)	(B, 4)	(I, 4)
(K, 1)	(F, 15)	(J, 3)
(C, 6)	(G, 25)	(L, 10)
(D, 10)	(H, 100)	(M, 9)

Last stage: evict a relatively small flow

HashPipe implemented in P4

All required functionalities are supported in P4



New key K

(K1, 4)

Stage 1	Stage 2	Stage 3
(A, 5)	(K2, 3)	(G, 4)
(K1, 4)	(D, 15)	(H, 3)
(B, 6)	(E, 25)	(K3, 10)
(C, 10)	(F, 100)	(I, 9)

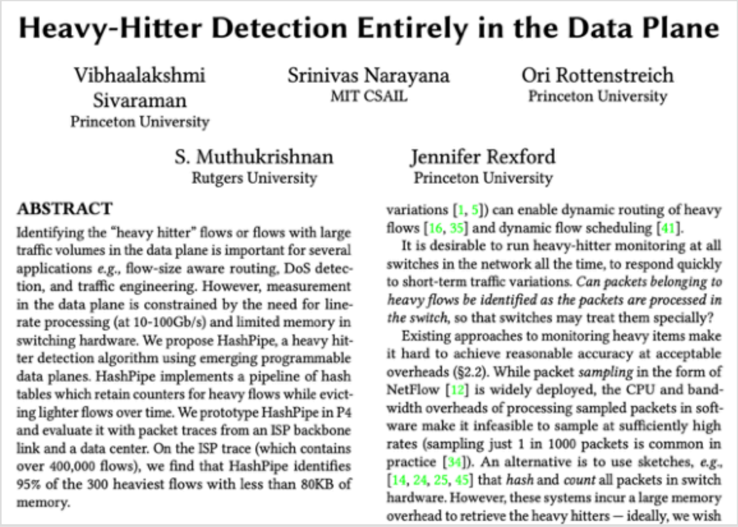
Hash on packet header

Packet metadata

Register arrays

Conditional updates to compute minimum

Sketch-based network monitoring



Heavy-Hitter Detection Entirely in the Data Plane

Vibhaalakshmi Sivaraman
Princeton University

Srinivas Narayana
MIT CSAIL

Ori Rottenstreich
Princeton University

S. Muthukrishnan
Rutgers University

Jennifer Rexford
Princeton University

ABSTRACT

Identifying the "heavy hitter" flows or flows with large traffic volumes in the data plane is important for several applications *e.g.*, flow-size aware routing, DoS detection, and traffic engineering. However, measurement in the data plane is constrained by the need for line-rate processing (at 10-100Gb/s) and limited memory in switching hardware. We propose HashPipe, a heavy hitter detection algorithm using emerging programmable data planes. HashPipe implements a pipeline of hash tables which retain counters for heavy flows while evicting lighter flows over time. We prototype HashPipe in P4 and evaluate it with packet traces from an ISP backbone link and a data center. On the ISP trace (which contains over 400,000 flows), we find that HashPipe identifies 95% of the 300 heaviest flows with less than 80KB of memory.

variations [1, 5]) can enable dynamic routing of heavy flows [16, 35] and dynamic flow scheduling [41].

It is desirable to run heavy-hitter monitoring at all switches in the network all the time, to respond quickly to short-term traffic variations. *Can packets belonging to heavy flows be identified as the packets are processed in the switch*, so that switches may treat them specially?

Existing approaches to monitoring heavy items make it hard to achieve reasonable accuracy at acceptable overheads (§2.2). While packet *sampling* in the form of NetFlow [12] is widely deployed, the CPU and bandwidth overheads of processing sampled packets in software make it infeasible to sample at sufficiently high rates (sampling just 1 in 1000 packets is common in practice [34]). An alternative is to use sketches, *e.g.*, [14, 24, 25, 45] that *hash* and *count* all packets in switch hardware. However, these systems incur a large memory overhead to retrieve the heavy hitters — ideally, we wish

One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon

Zaoxing Liu[†], Antonis Manousis[⋆], Gregory Vorsanger[†], Vyas Sekar[⋆], Vladimir Braverman[†]
[†] Johns Hopkins University [⋆] Carnegie Mellon University

ABSTRACT

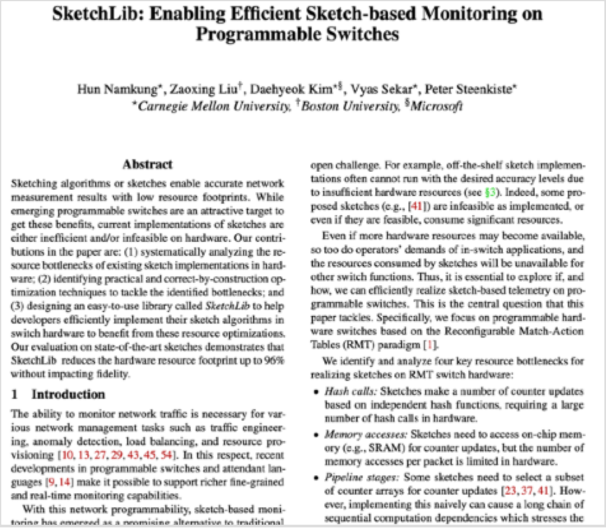
Network management requires accurate estimates of metrics for many applications including traffic engineering (e.g., heavy hitters), anomaly detection (e.g., entropy of source addresses), and security (e.g., DDoS detection). Obtaining accurate estimates given router CPU and memory constraints is a challenging problem. Existing approaches fall in one of two undesirable extremes: (1) low fidelity general-purpose approaches such as sampling, or (2) high fidelity but complex algorithms customized to specific application-level metrics. Ideally, a solution should be both general (i.e., supports many applications) and provide accuracy comparable to custom algorithms. This paper presents *UnivMon*, a framework for flow monitoring which leverages recent theoretical advances and demonstrates that it is possible to achieve both generality and high accuracy. UnivMon uses an application-agnostic data plane monitoring primitive; different (and possibly unforeseen) estimation algorithms run in the control plane, and use the statistics from the data plane to compute application-level metrics. We present a proof-of-concept implementation of UnivMon using P4 and develop simple coordination techniques to provide a "one-big-switch" abstraction for network-wide monitoring. We evaluate the effectiveness of UnivMon using a range of trace-driven evaluations and show that it offers comparable (and sometimes better) accuracy relative to custom sketching solutions across a range of monitoring tasks.

1 Introduction

Network management is multi-faceted and encompasses a range of tasks including traffic engineering [11, 32], attack and anomaly detection [49], and forensic analysis [46]. Each such management task requires accurate and timely statistics on different application-level metrics of interest; e.g., the flow size distribution [37], heavy hitters [10], entropy measures [38,50], or detecting changes in traffic patterns [44].

At a high level, there are two classes of techniques to estimate these metrics of interest. The first class of approaches relies on *generic flow monitoring*, typically with some form of packet sampling (e.g., NetFlow [25]). While generic flow monitoring is good for coarse-grained visibility, prior work has shown that it provides low accuracy for more fine-grained metrics [30, 31, 43]. These well-known limitations of sampling motivated an alternative class of techniques based on *sketching* or *streaming* algorithms. Here, custom online algorithms and data structures are designed for specific metrics of interest that can yield provable resource-accuracy tradeoffs (e.g., [17, 18, 20, 31, 36, 38, 43]).

While the body of work in data streaming and sketching has made significant contributions, we argue that this trajectory of crafting special-purpose algorithms is untenable in the long term. As the number of monitoring tasks grows, this entails significant investment in algorithm design and hardware support for new metrics of interest. While recent tools like OpenSketch [47] and SCREAM [41] provide libraries to



SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches

Hun Namkung[*], Zaoxing Liu[†], Daehyeok Kim[*§], Vyas Sekar[*], Peter Steenkiste[*]
[*]*Carnegie Mellon University,* [†]*Boston University,* [§]*Microsoft*

Abstract

Sketching algorithms or sketches enable accurate network measurement results with low resource footprints. While emerging programmable switches are an attractive target to get these benefits, current implementations of sketches are either inefficient and/or infeasible on hardware. Our contributions in the paper are: (1) systematically analyzing the resource bottlenecks of existing sketch implementations in hardware; (2) identifying practical and correct-by-construction optimization techniques to tackle the identified bottlenecks; and (3) designing an easy-to-use library called *SketchLib* to help developers efficiently implement their sketch algorithms in switch hardware to benefit from these resource optimizations. Our evaluation on state-of-the-art sketches demonstrates that SketchLib reduces the hardware resource footprint up to 96% without impacting fidelity.

1 Introduction

The ability to monitor network traffic is necessary for various network management tasks such as traffic engineering, anomaly detection, load balancing, and resource provisioning [10, 13, 27, 29, 43, 45, 54]. In this respect, recent developments in programmable switches and attendant languages [9, 14] make it possible to support richer fine-grained and real-time monitoring capabilities.

With this network programmability, sketch-based monitoring has emerged as a promising alternative to traditional

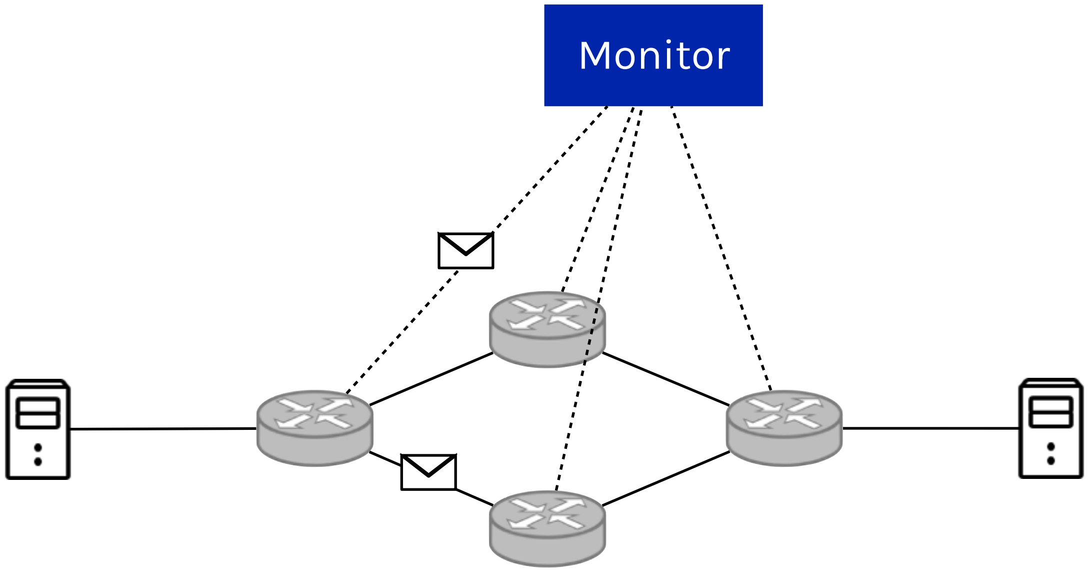
open challenge. For example, off-the-shelf sketch implementations often cannot run with the desired accuracy levels due to insufficient hardware resources (see §3). Indeed, some proposed sketches (e.g., [41]) are infeasible as implemented, or even if they are feasible, consume significant resources.

Even if more hardware resources may become available, so too do operators' demands of in-switch applications, and the resources consumed by sketches will be unavailable for other switch functions. Thus, it is essential to explore if, and how, we can efficiently realize sketch-based telemetry on programmable switches. This is the central question that this paper tackles. Specifically, we focus on programmable hardware switches based on the Reconfigurable Match-Action Tables (RMT) paradigm [1].

We identify and analyze four key resource bottlenecks for realizing sketches on RMT switch hardware:

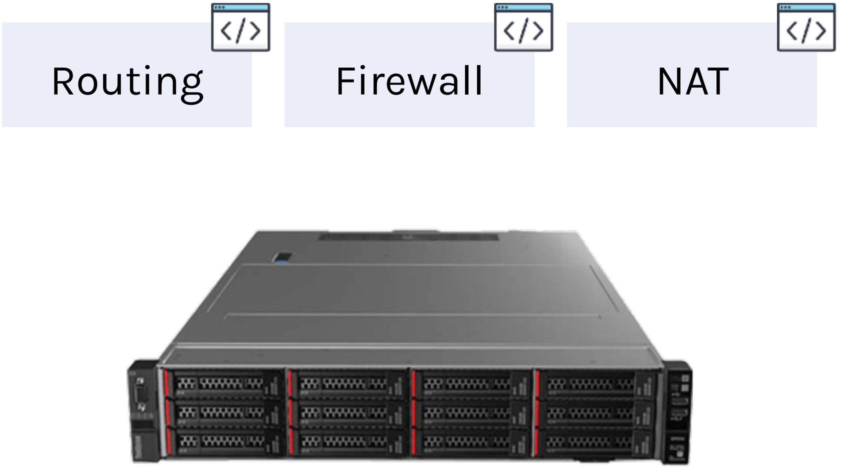
- *Hash calls:* Sketches make a number of counter updates based on independent hash functions, requiring a large number of hash calls in hardware.
- *Memory accesses:* Sketches need to access on-chip memory (e.g., SRAM) for counter updates, but the number of memory accesses per packet is limited in hardware.
- *Pipeline stages:* Some sketches need to select a subset of counter arrays for counter updates [23, 37, 41]. However, implementing this naively can cause a long chain of sequential computation dependencies which stresses the

Summary



Network monitoring: typical data structures for network monitoring, heavy hitter detection in programmable data plane

Next time: network function virtualization



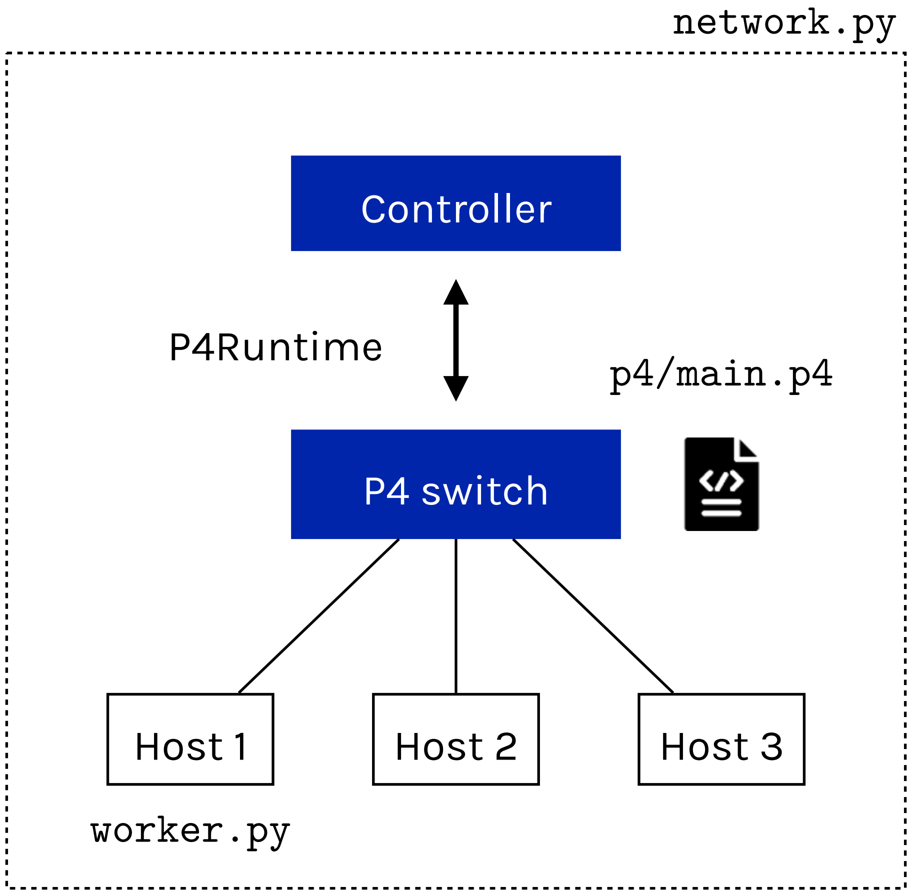
Routing

Firewall

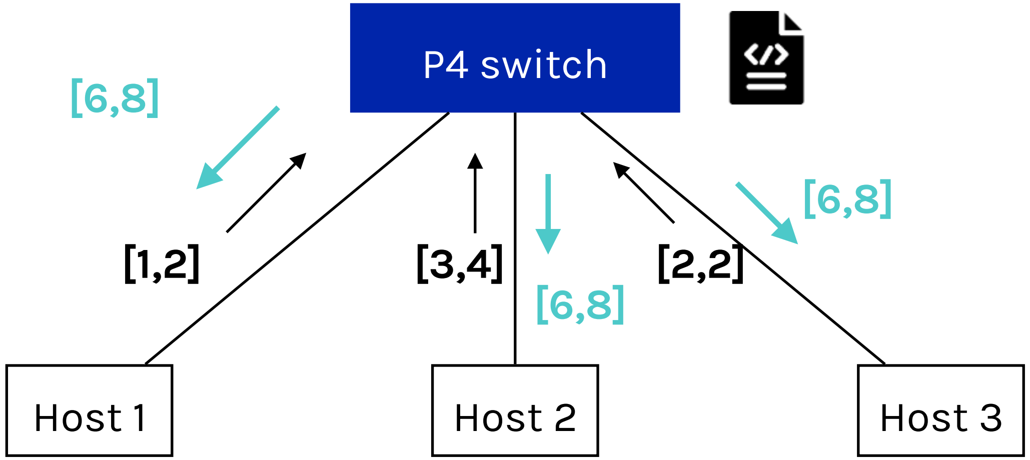
NAT

How to implement network functions in software running on commodity servers?

Lab5 introduction



Lab5 introduction



Vectors	[**1,2**,1,3,2,3,1,1]	[**3,4**,2,3,1,4,2,3]	[**2,2**,1,1,3,2,1,3]
Agg. results	[6,8,0,0,0,0,0,0]	[6,8,0,0,0,0,0,0]	[6,8,0,0,0,0,0,0]

Three levels: (1) Ethernet frames, (2) UDP sockets, (3) UDP sockets with reliability

Call for SHKs (TAs and RAs)

SHKs for teaching

- WS24/25: Computer Networks
- SS25: Advanced Networked Systems
- One year contract, 6.5 hours per week
- Tasks: handling exercises + Q&A
- Requirements
    - Interests in networking
    - Good grades in CN and ANS
    - Reliable

SHKs for research

- Both short-term and long-term projects
- Topics
    - In-network aggregation for ML
    - TinyML: LLM on tiny devices
- Tasks
    - Lab testbed setup
    - Experiments
    - Your own ideas/research

Email your CV + transcripts (lin.wang@upb.de)