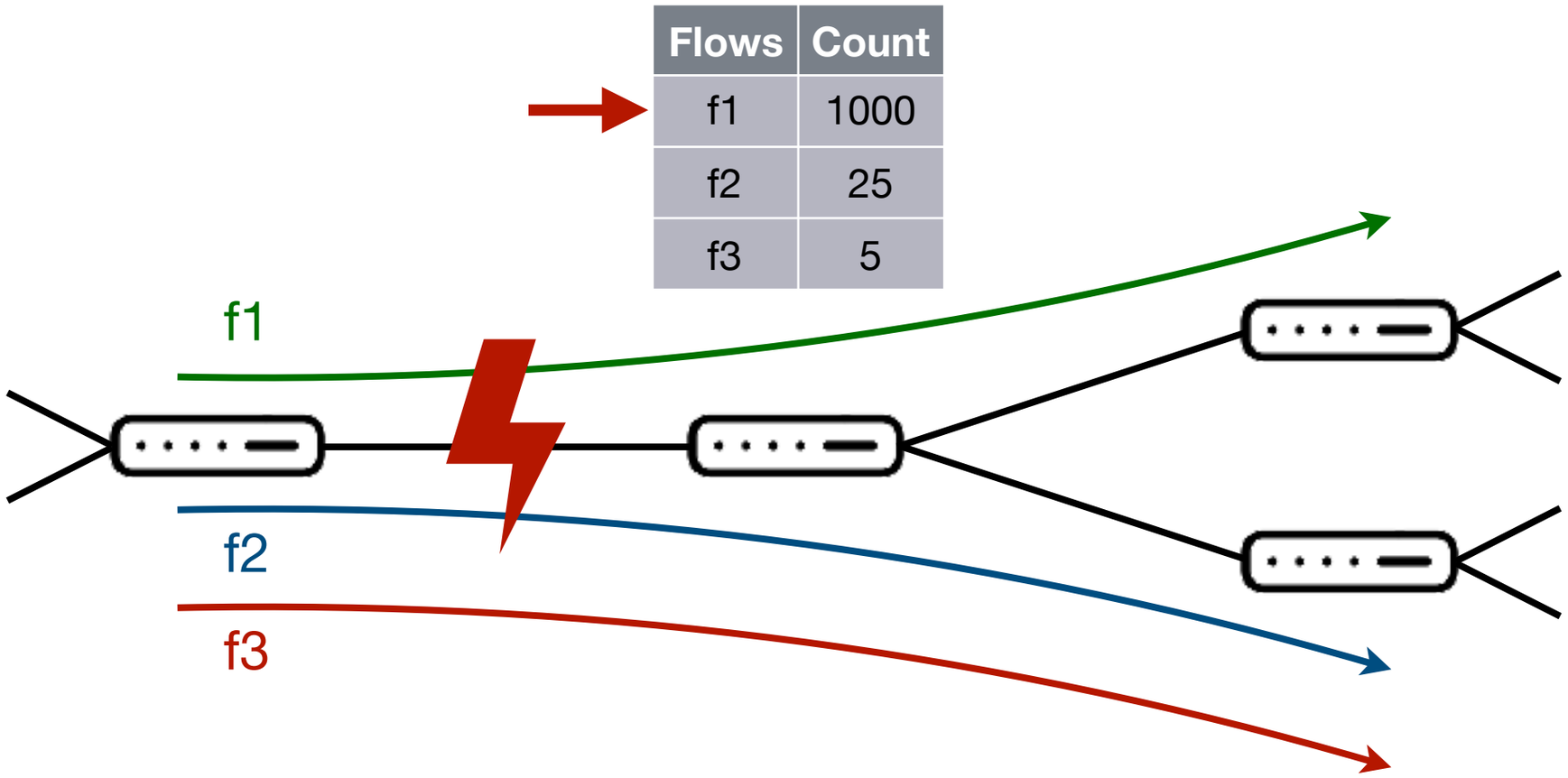
Heavy-Hitter Detection Entirely in the Data Plane

V. Sivaraman, S. Narayana, O. Rottenstreich,
S. Muthukrishnan, J. Rexford

Presented by Benjamin Rothenberger

Heavy Hitters



- Trouble shooting (Anomaly detection, DoS detection)
- Traffic engineering (Flow-size aware routing, Dynamic routing)

Heavy Hitters - Definition

- Flows larger than a fraction *t* of the total packets seen on a link

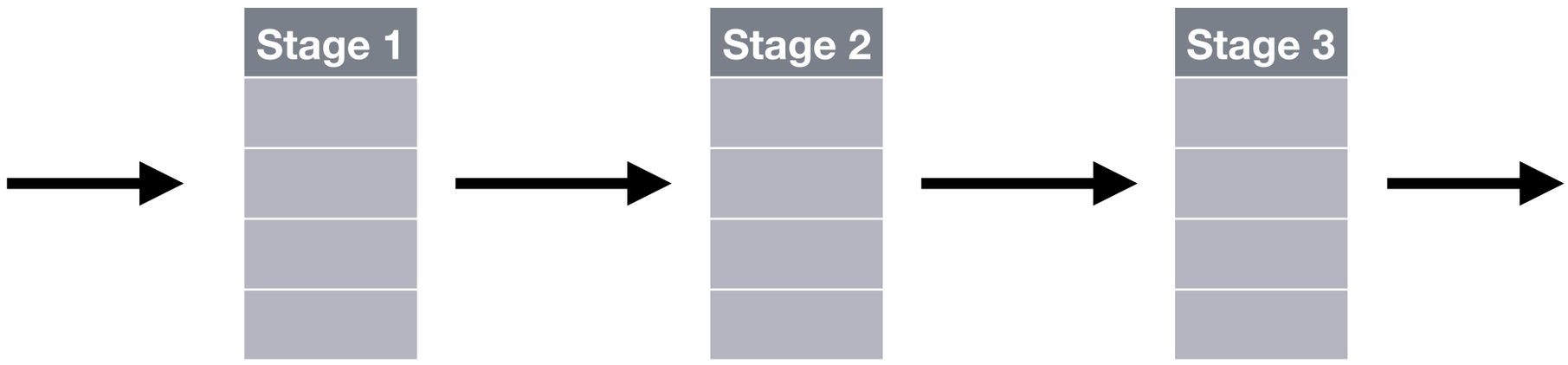
“threshold-*t*” problem

- Top *k* flows by size seen on a link

“top-*k*” problem

Why in the Data Plane?

- Programmable switches with stateful memory
- Basic arithmetic operations on state
- Pipelined operations over multiple states



Challenges

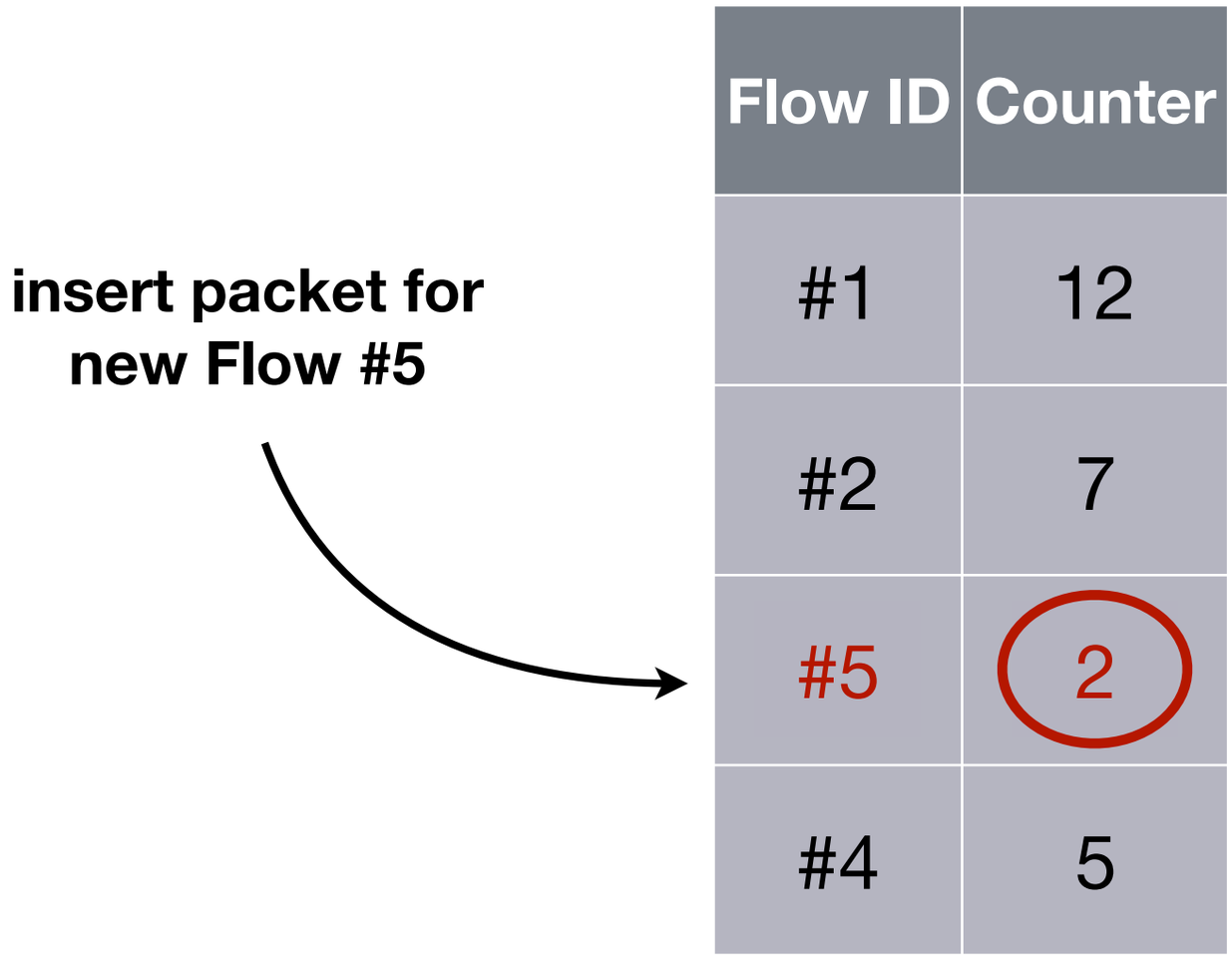
- Processing restricted to data plane
- Limited memory in switching hardware
- Small time budget for processing
- High accuracy of detection
- Line-rate processing

Context

	Sampling-based	Sketching-based	Counting-based
	record a subset of packets by sampling	hash packet & increment counter at hashed location	maintain table of flows and corresponding counts
+	Small memory to track heavy flows	Statistics for all flows in single data structure	Summary structure with heavy flow IDs and counters
−	Underestimates counts for heavy flows	No flow identifier to count association	Occasional updates to multiple counters
	NetFlow NetQRE	SketchVisor	Trumpet PathDump Space-Saving

Space-saving algorithm

- Goal: track the *k* heaviest items using only O(*k*) counters



Space-saving algorithm

- *O(k)* space to store heavy flows
- Provable guarantees on accuracy
- Evict the minimum to insert new flow
- Multiple reads, but exactly one write per packet

Towards HashPipe

- Idea #1: prevent entire table scan by sampling a constant *d* number of values

- Approximate minimum

- Multiple read-modify-write operations in parallel

  Not supported in hardware!

Towards HashPipe

- Idea #2: reduce number of reads to memory by splitting the counter table into *d* disjoint tables
  - Read exactly one slot per table
  - Pipeline memory access
  - Requires multiple passes: find minimum, update counter
  - Possible through recirculation of packet
    needed for every packet? halves bandwidth!

Towards HashPipe

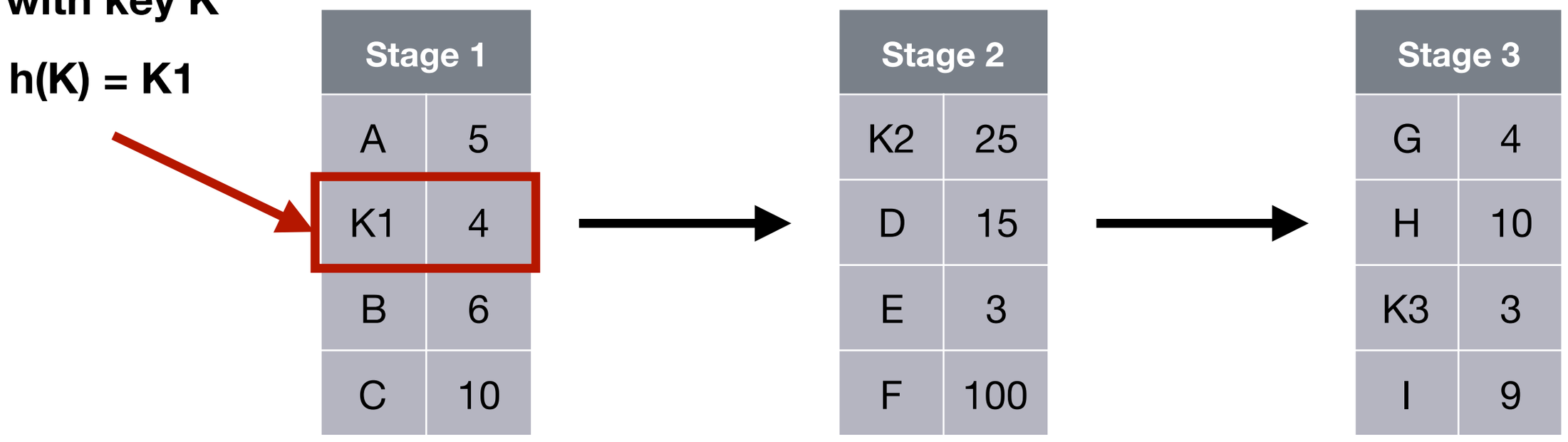
- Idea #3: Feed-forward packet processing and track rolling minimum
  - Track minimum counter value as packet *metadata*
  - Switch hashes into each stage on the *carried key*
  - Always insert in the first stage, evict existing entry into packet metadata

    Duplicate keys!

HashPipe - Example

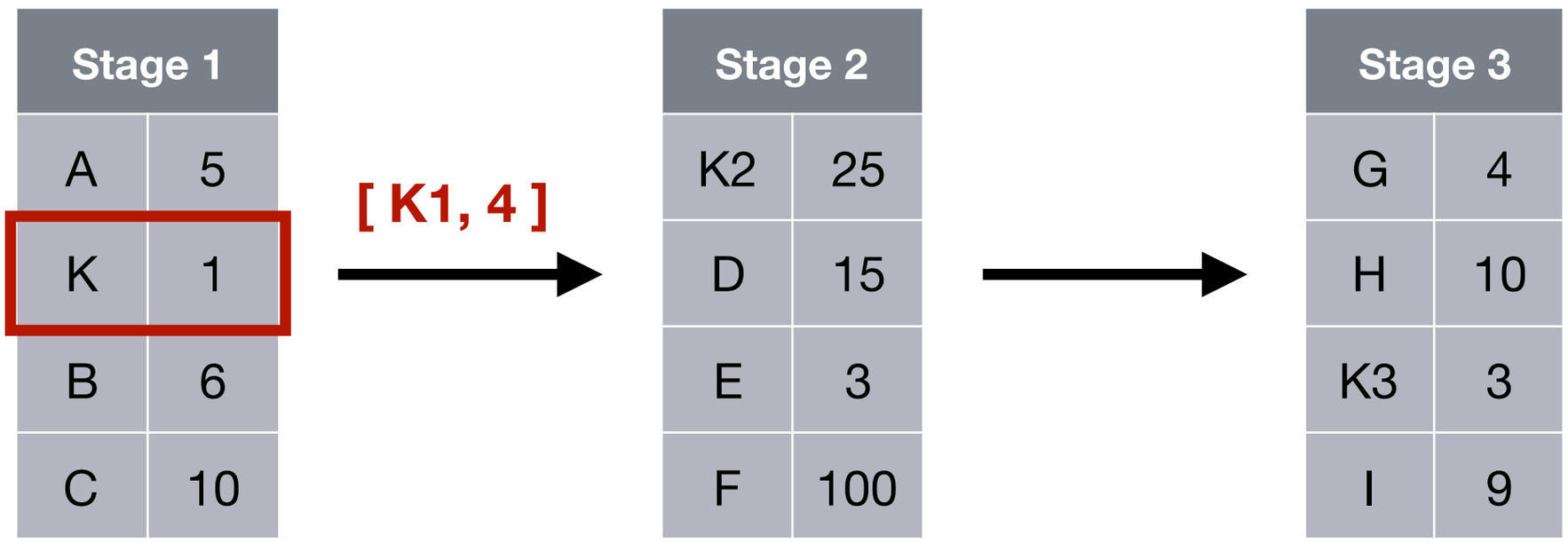
new flow with key K

h(K) = K1



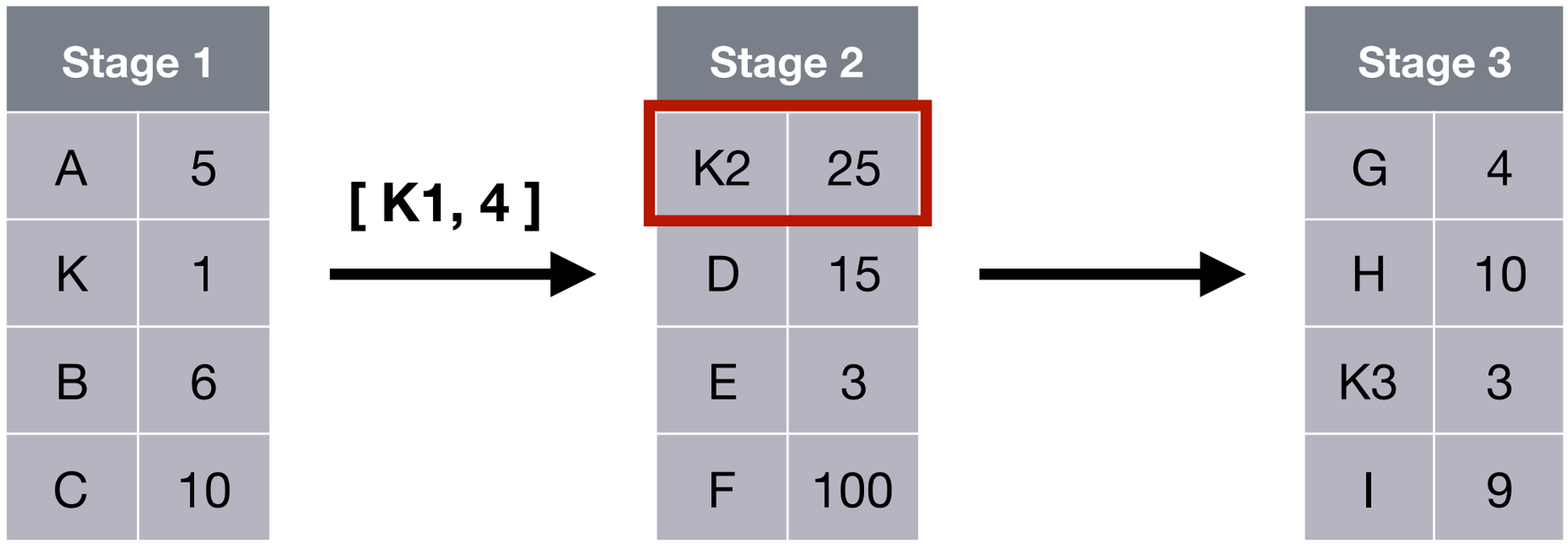
always insert into first stage!

HashPipe - Example



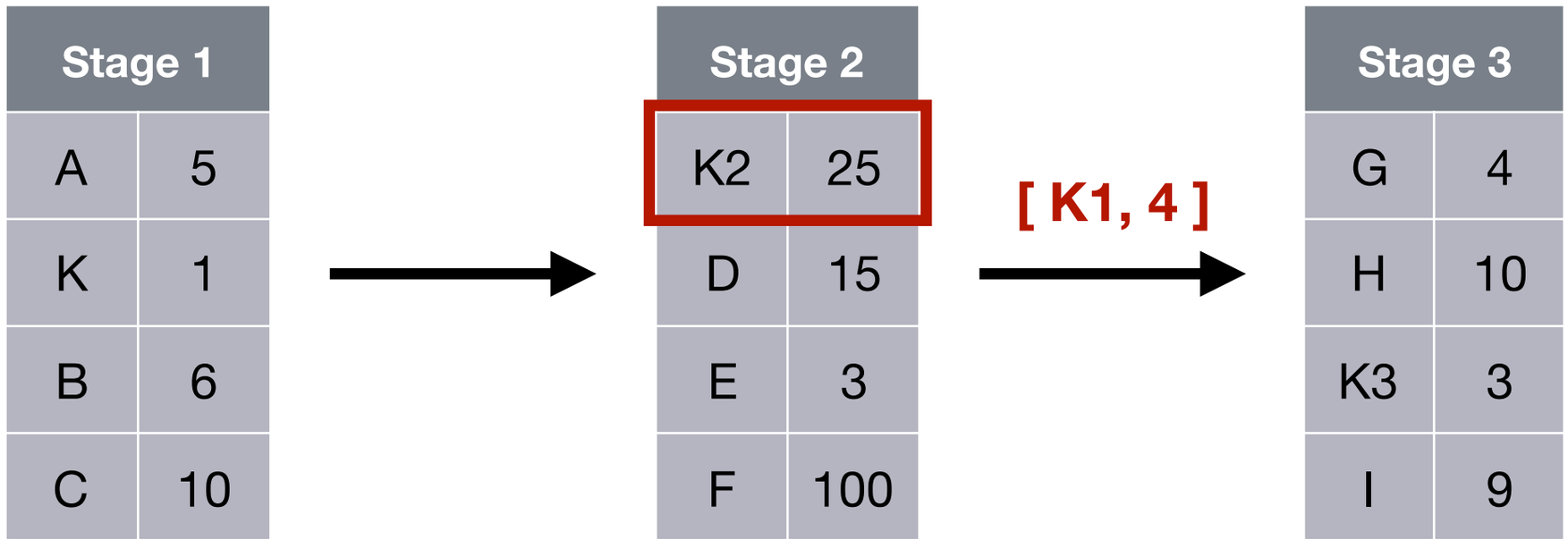
HashPipe - Example

h(K1) = K2
Max(K1, K2) -> K2



HashPipe - Example

h(K1) = K2
Max(K1, K2) -> K2



HashPipe - Example

h(K1) = K3
Max(K1, K3) -> K1

Stage 1	
A	5
K	1
B	6
C	10

→

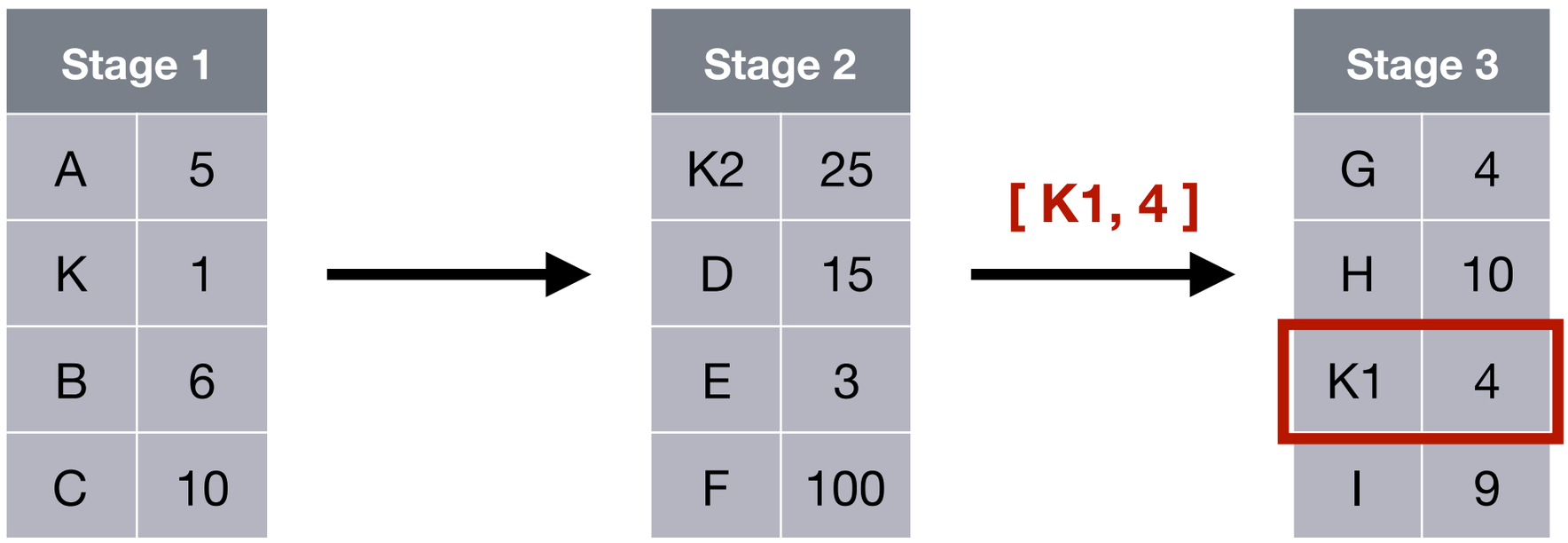
Stage 2	
K2	25
D	15
E	3
F	100

[K1, 4] →

Stage 3	
G	4
H	10
K3	3
I	9

HashPipe - Example

h(K1) = K3
Max(K1, K3) -> K1



Evaluation

- Prototype implementation in P4
- Top-k flows on:
    1. CAIDA traffic traces with 500M packets
    2. Data center trace with 100M packets

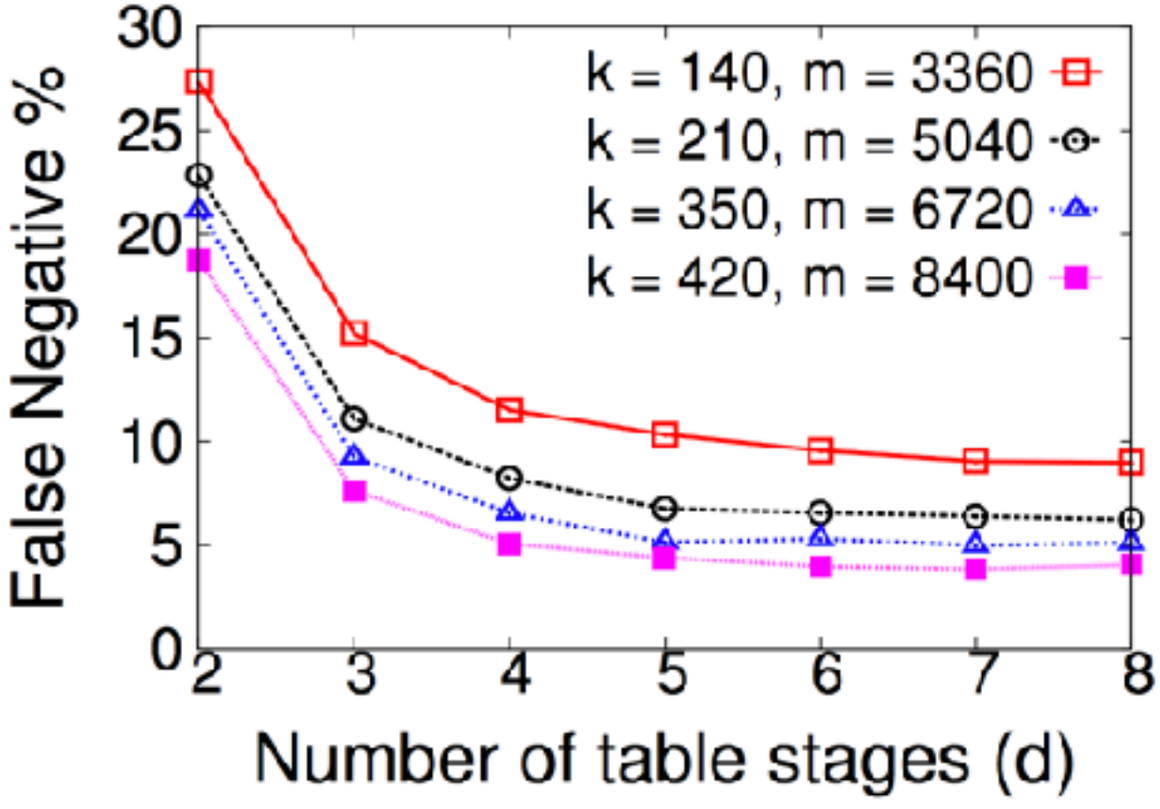
Metrics

- False Positives (FP):
  reporting a non-heavy flow as heavy

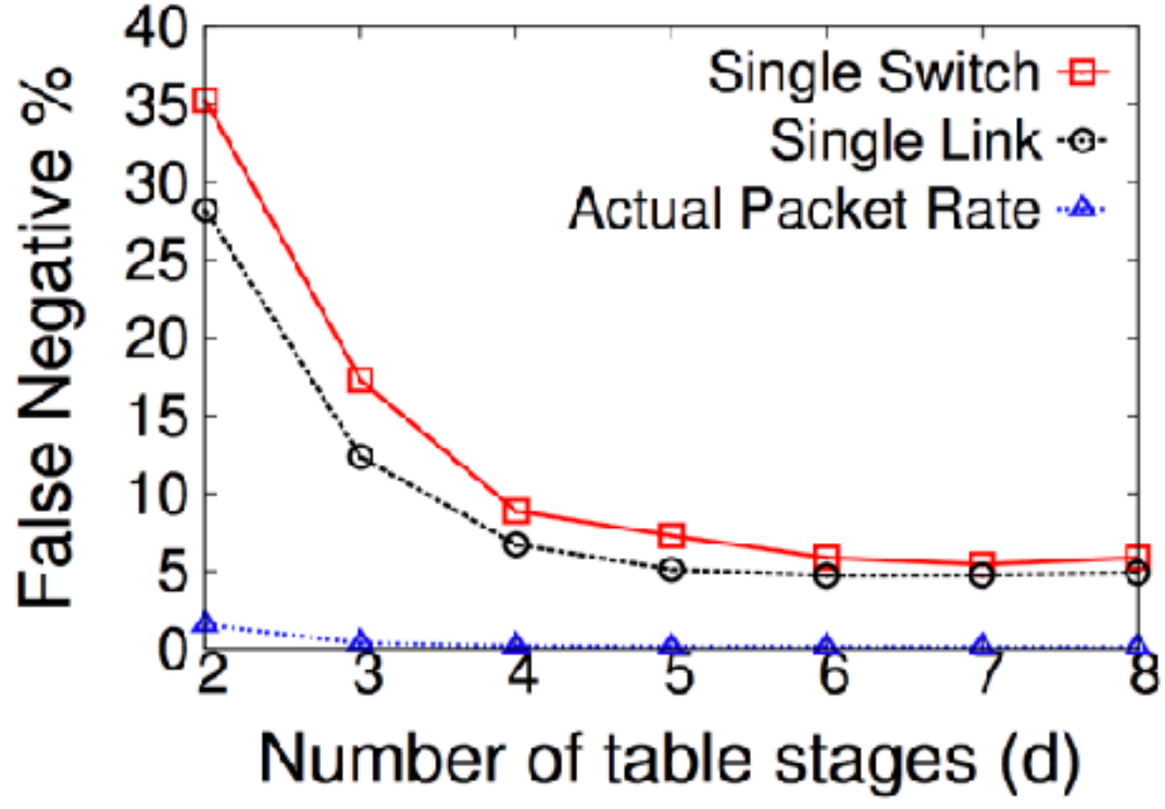
- False Negatives (FN):
  not reporting a heavy flow

- Count Estimation Error (CEE):
  deviation of flow count in data structure vs. actual flow count

How to choose *d*?



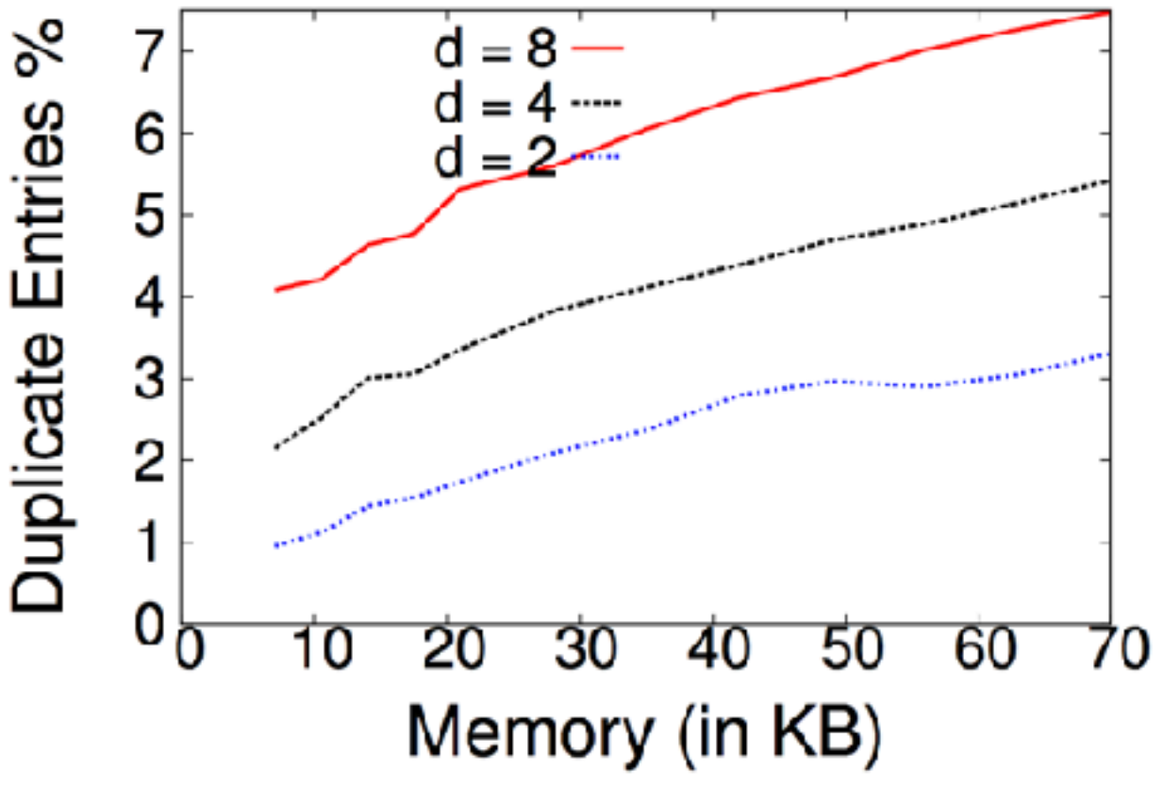
a) Backbone



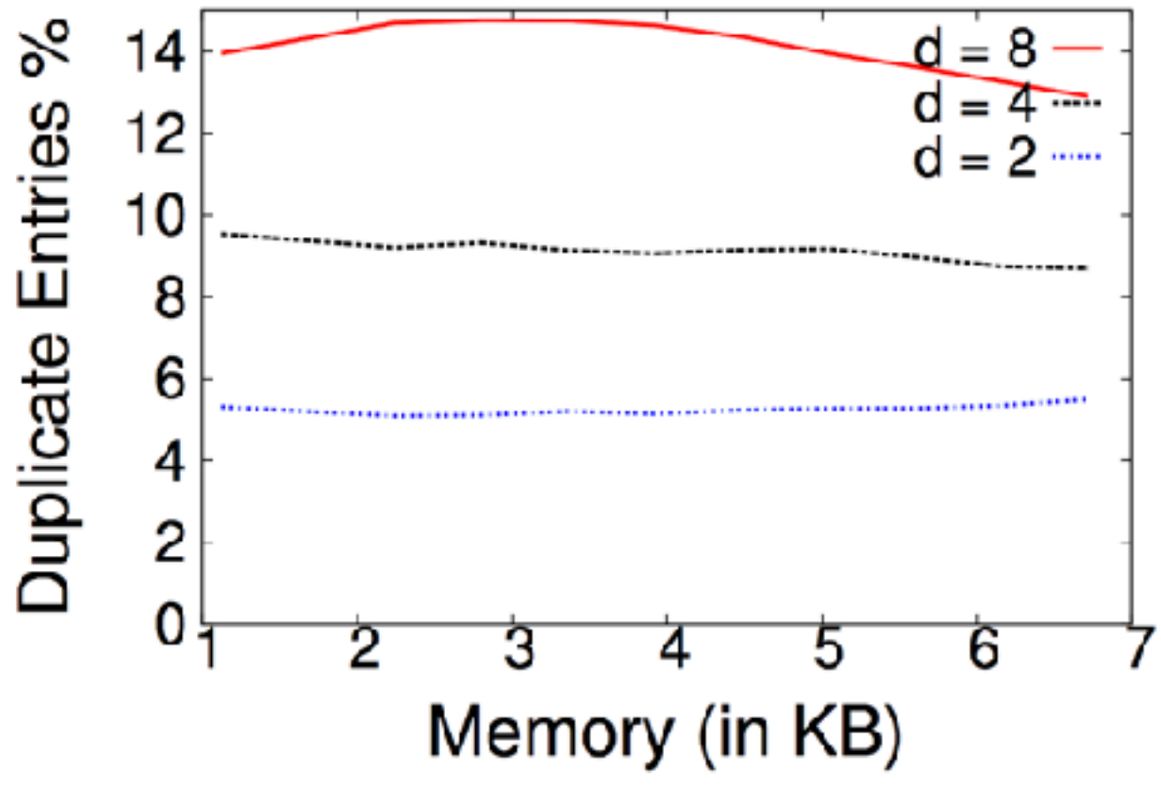
b) DCN

d=6?

Duplicate Keys



a) Backbone



b) DCN

Comments

- Iterative approach to present solution
- Considers underlying hardware architecture for solution
- No modelling / mathematical reasoning about system properties
  (e.g., probability of hash collisions / upper bound on duplicates)
- Solution targets single problem: could it be extended to solve other problems?

Questions / Comments ?

Backup Slides

Space-saving algorithm

Algorithm 1: Space Saving algorithm [28]

```
1  ▷ Table T has m slots, either containing (key_j, val_j)
     at slot j ∈ {1, ..., m}, or empty. Incoming packet
     has key iKey.
2  if ∃ slot j in T with iKey = key_j then
3  |   val_j ← val_j + 1
4  else
5  |   if ∃ empty slot j in T then
6  |   |   (key_j, val_j) ← (iKey, 1)
7  |   else
8  |   |   r ← argmin_{j∈{1,...,m}}(val_j)
9  |   |   (key_r, val_r) ← (iKey, val_r + 1)
10 |   end
11 end
```

if key exists
-> increment counter

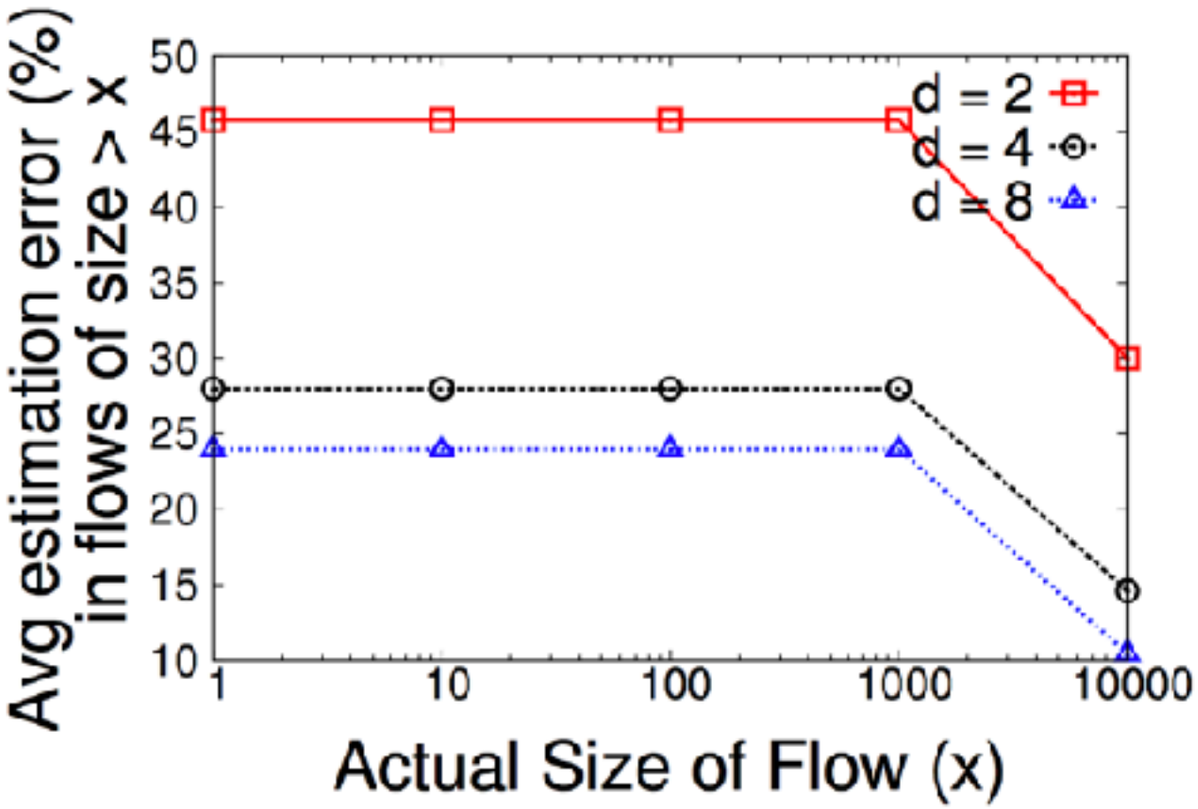
if there exists an empty slot
-> insert key with value 1

find smallest counter
-> increment old value
and store with new key

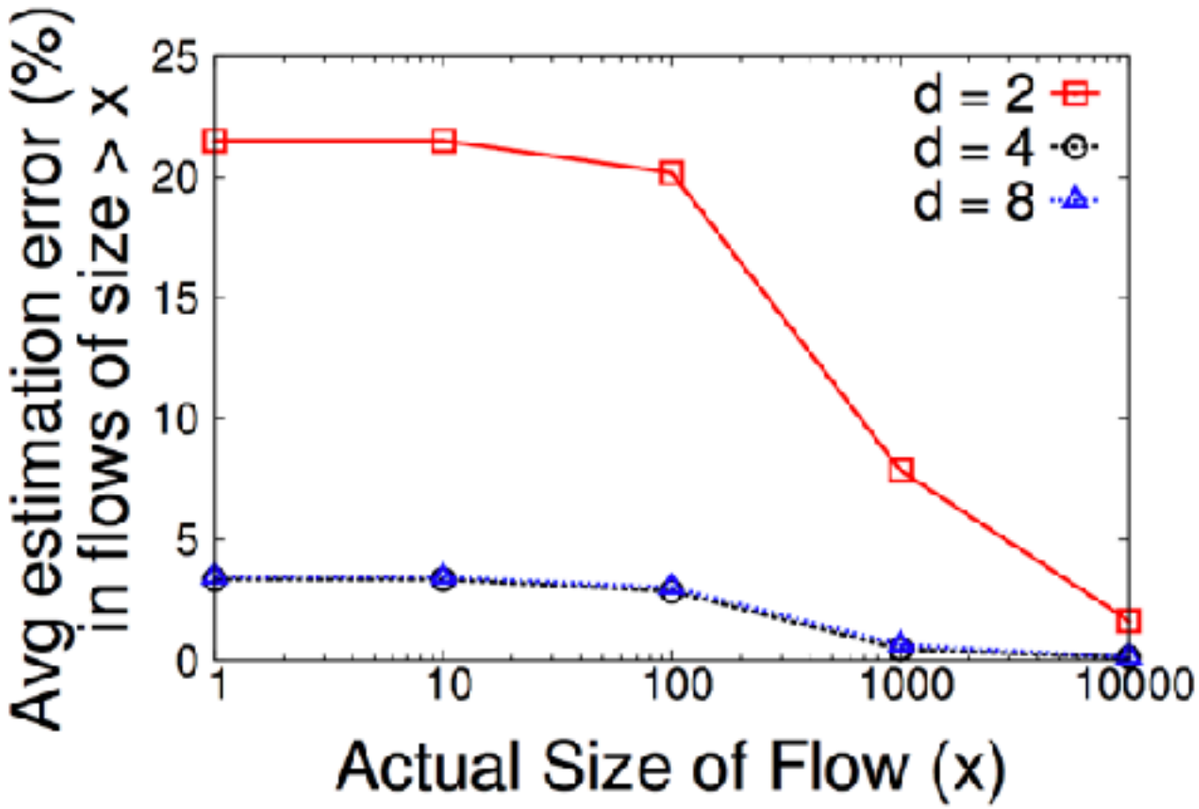
Accuracy Properties

- No flow counter is underestimated
- Minimum value is an upper bound on the overestimation error of any counter
- Any flow with true count higher than the average table count is always present in table

Average Estimation Error

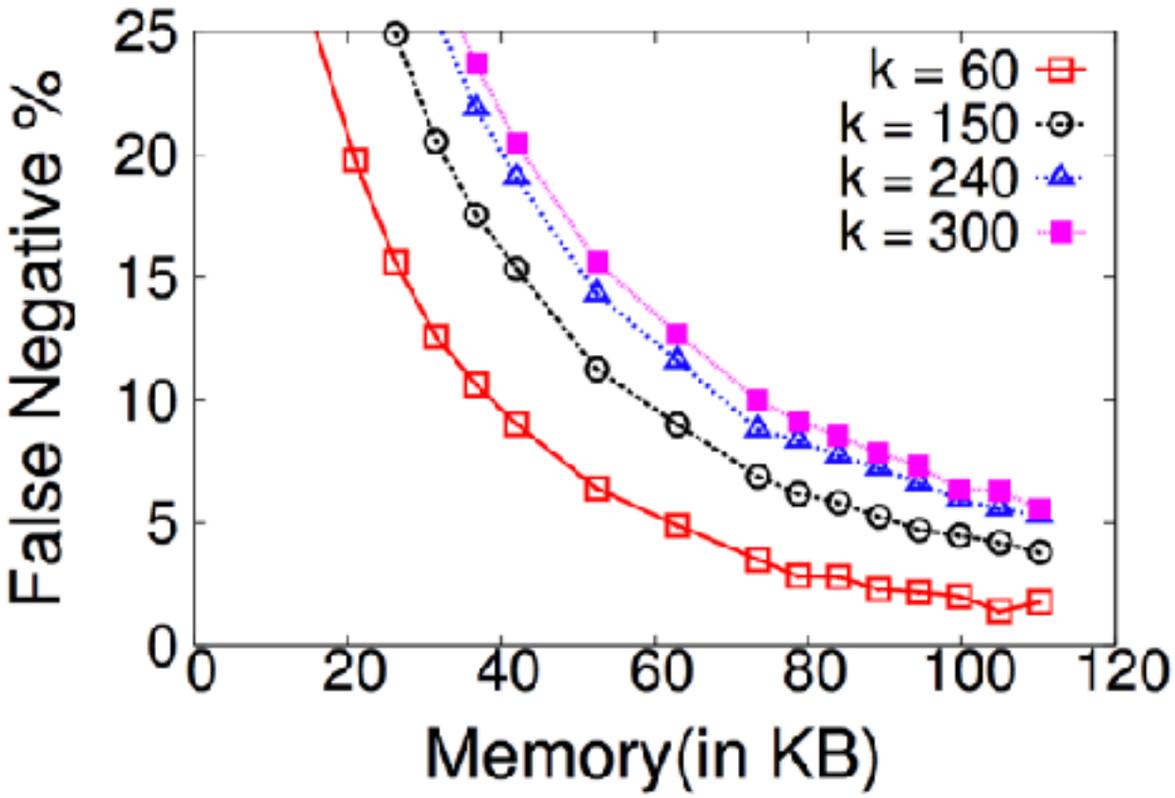


(a) ISP backbone

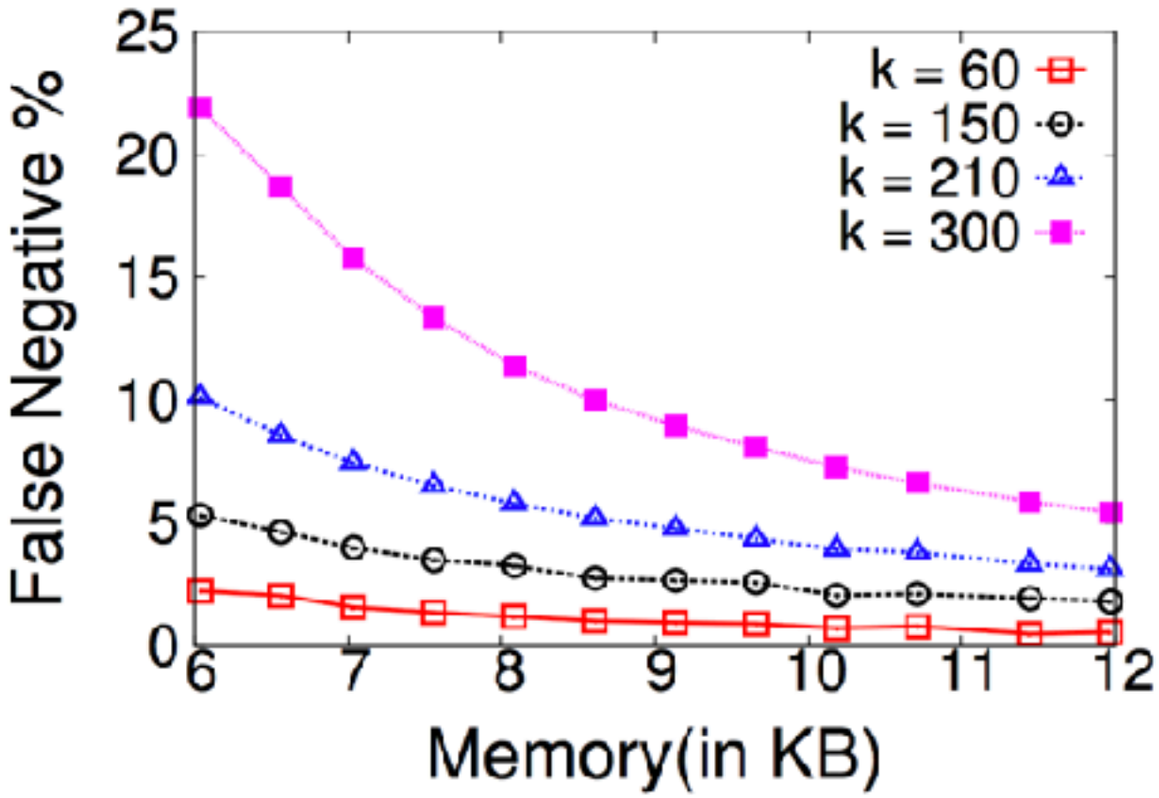


(b) Data center (link)

FN vs. Memory

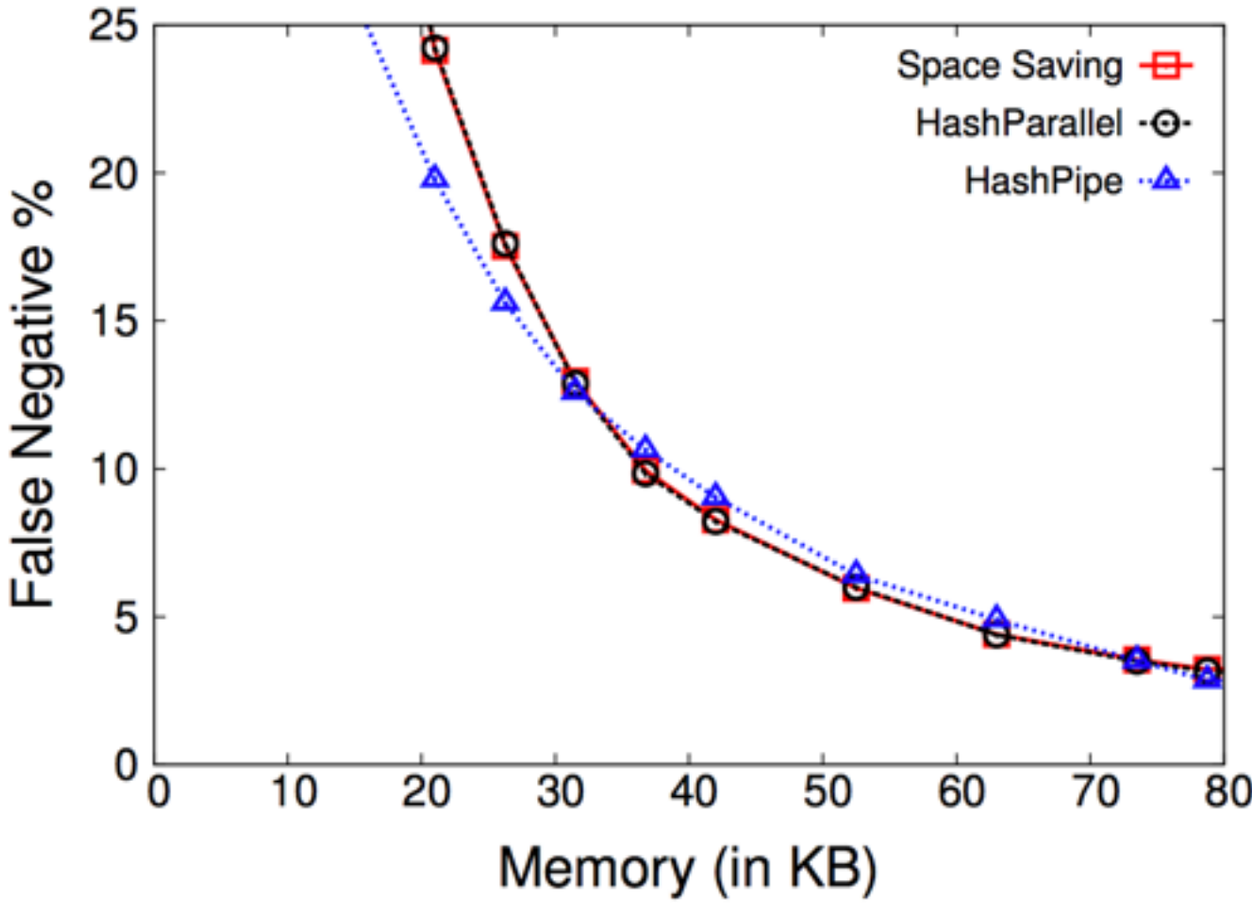


(a) ISP backbone

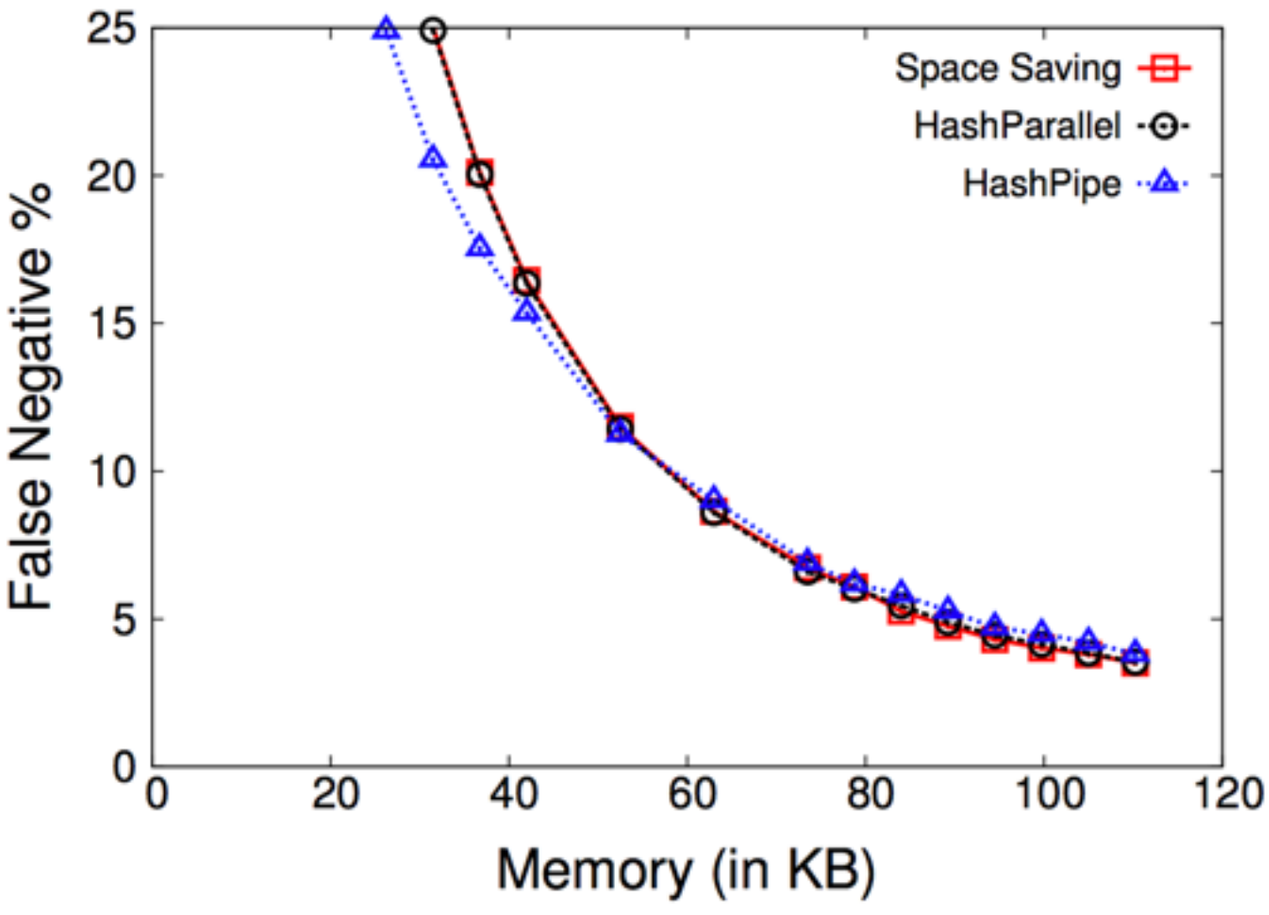


(b) Data center

HashPipe vs. Related Work



k = 60



k = 150