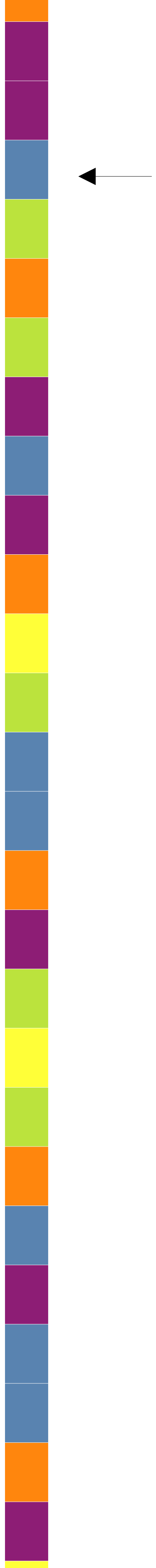
You are looking at a stream of data.

In networking, we usually talk about **streams of packets**,

but these questions apply to other domains as well,

e.g. **search engines and databases**.



You are looking at a stream of data (packets).

There are many questions you might ask:

Is a certain element (e.g. ip address) in the stream?

How frequently does an element appear?

How many distinct elements are in the stream?

What are the most frequent elements?

You are looking at a stream of data (packets).

There are many questions you might ask:

Is a certain element (e.g. ip address) in the stream?

How frequently does an element appear?

How many distinct elements are in the stream?

What are the most frequent elements?

In P4, these questions are difficult to answer.

You are looking at a stream of data (packets).

Today, I'll show you how set membership and frequency queries can be realized in P4.

PART 1

Is a certain element (e.g. ip address) in the stream?

→ Bloom filter

PART 2

How frequently does an element appear?

→ CountMin Sketch, Count Sketch, ...

part 1: **set membership queries with Bloom filters**

Is a certain element (e.g. ip address) in the stream?

(slides by Thomas Holterbach)

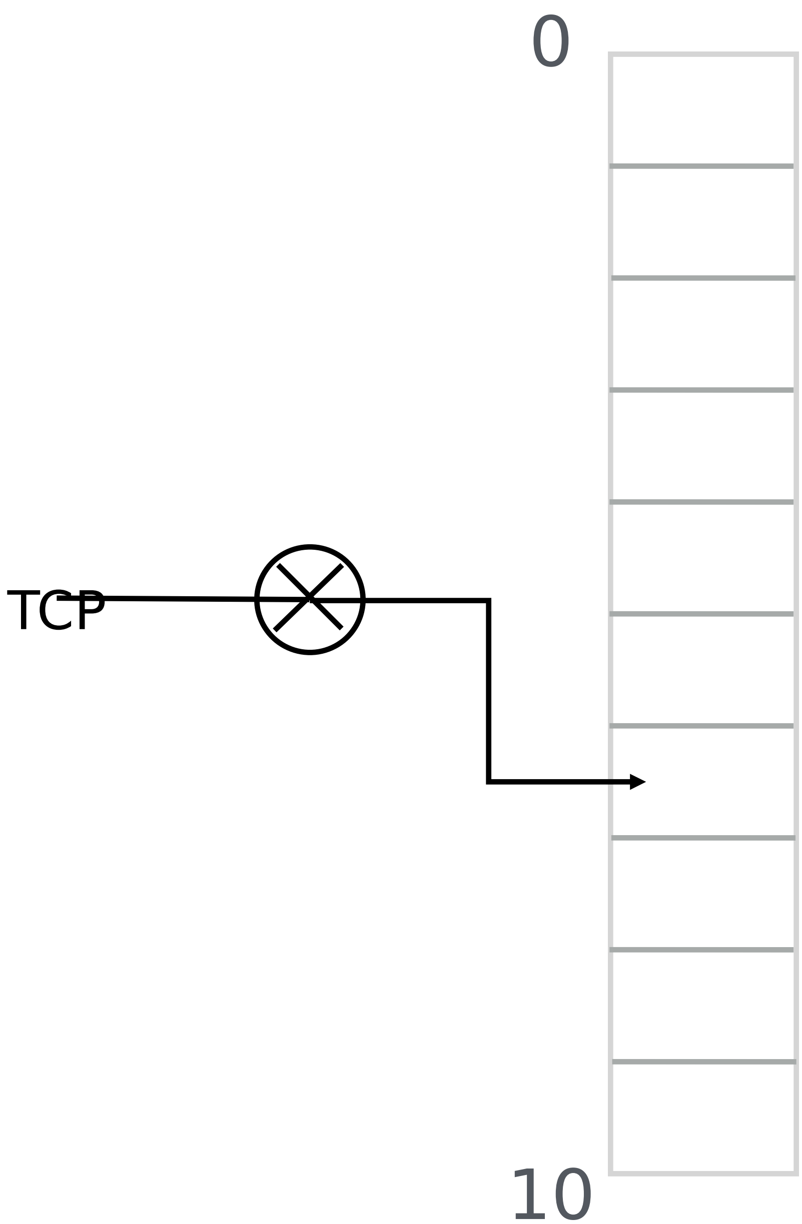
There are two common strategies to implement a set

	strategy #1	strategy #2
output	Deterministic	
number of required operations	Probabilistic	

Intuitive implementation of a set

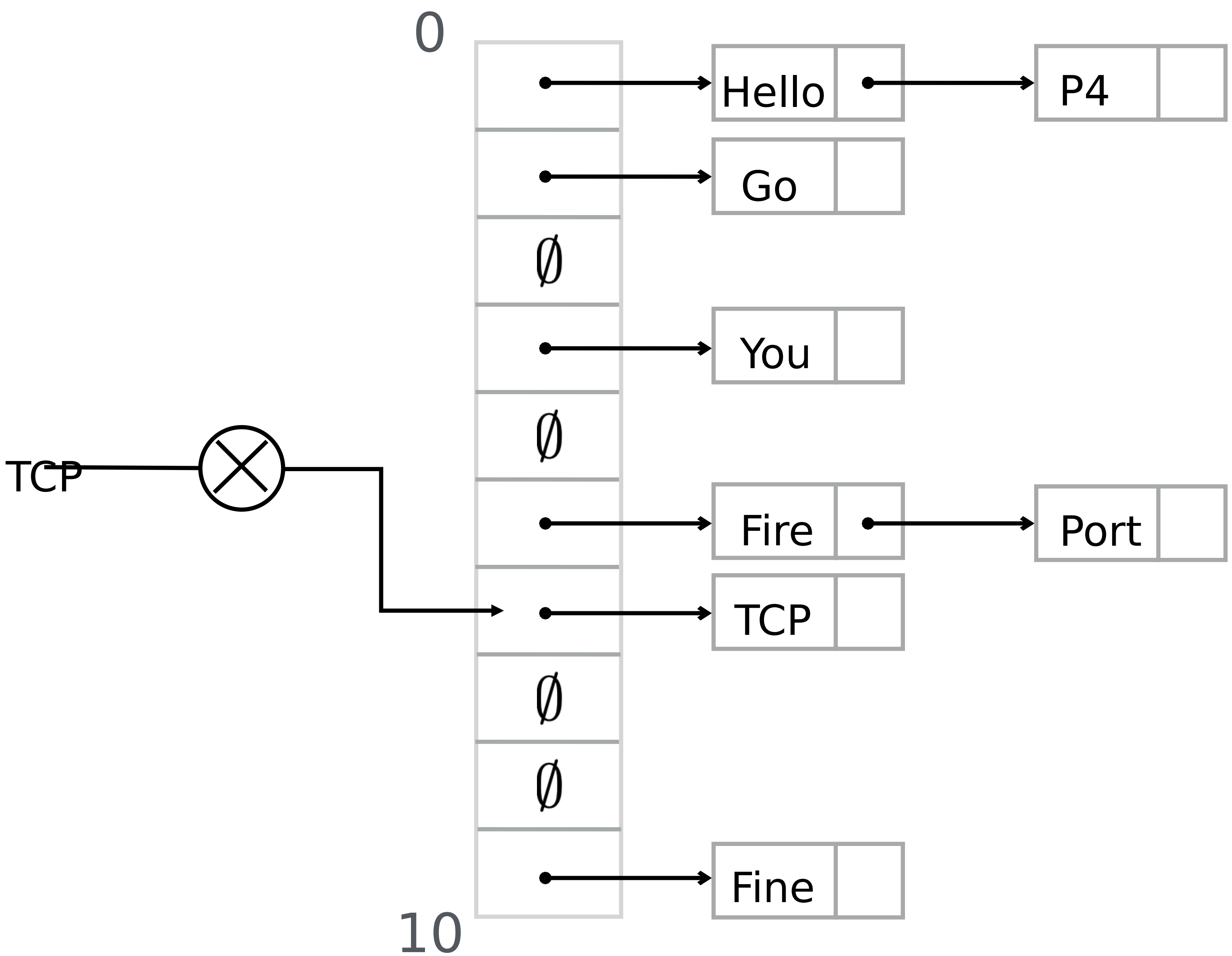
Intuitive implementation of a set

Separate-chaining



Intuitive implementation of a set

Separate-chaining



Intuitive implementation of a set

Separate-chaining

N elements and M cells

	list size
average	N/M
worse-case	N

Intuitive implementation of a set

Separate-chaining

Pros: accurate and fast in the average case

Con: only works in hardware if there is a low number of elements (e.g. < 100)

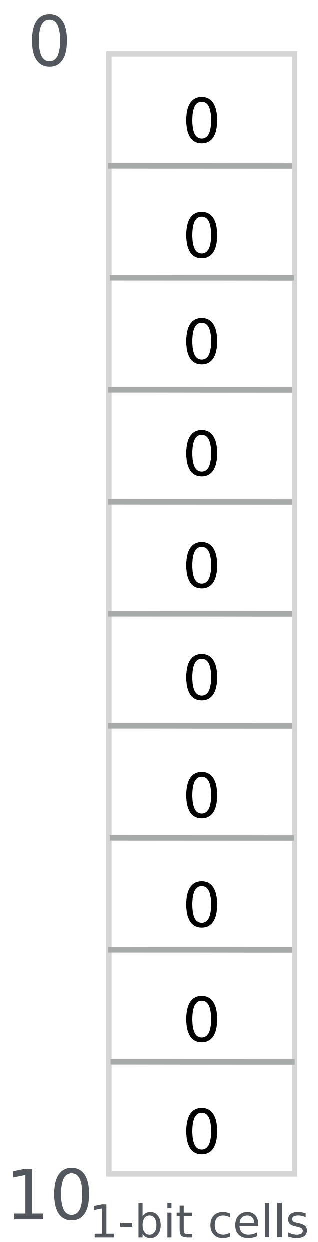
There are two common strategies

to implement a set

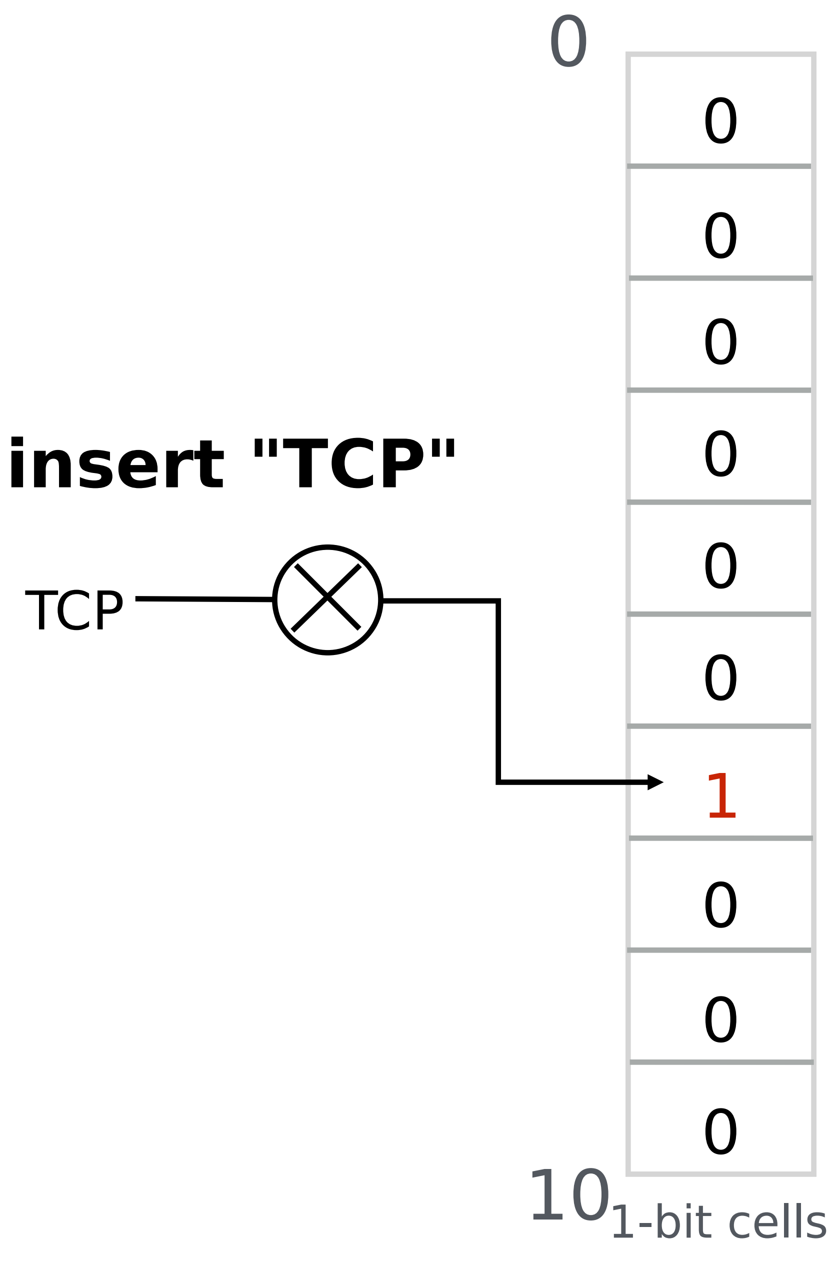
	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

'probabilistic data structures'

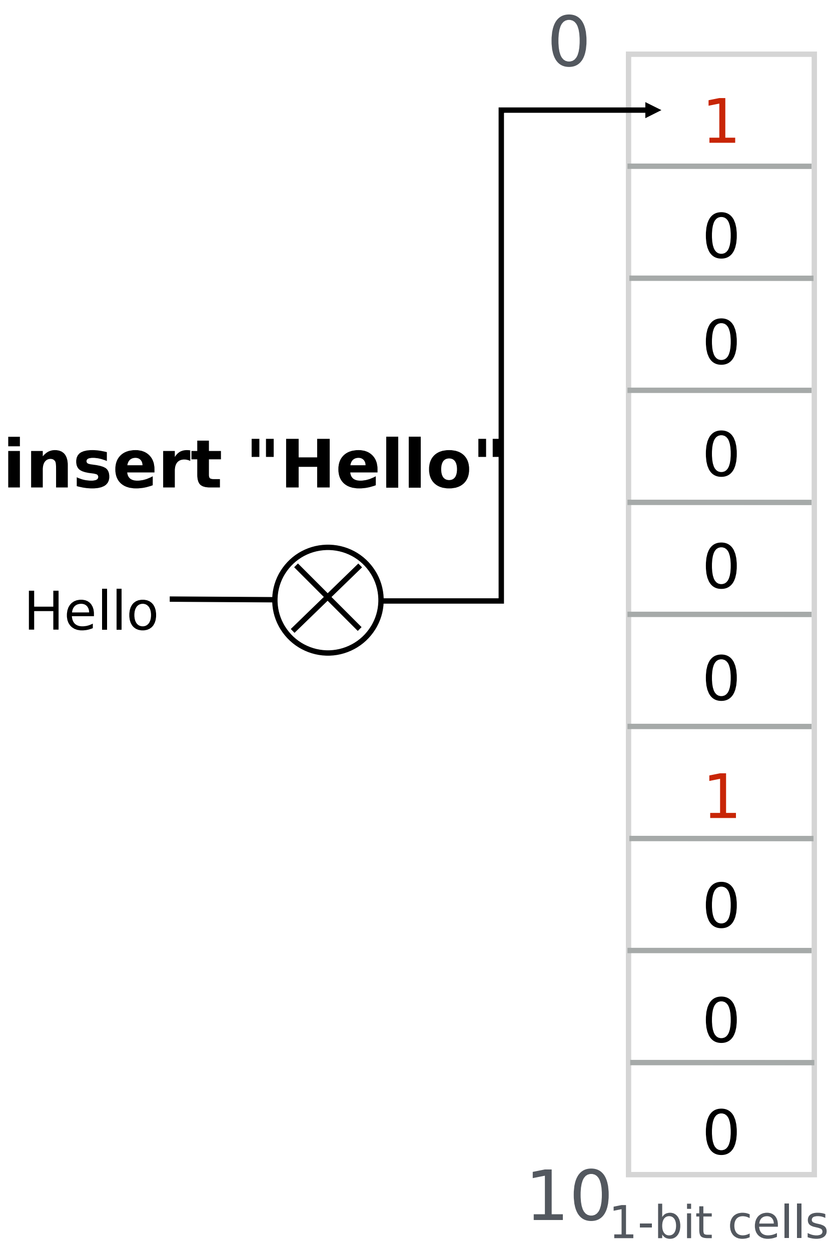
A simple approach for insertions and membership queries



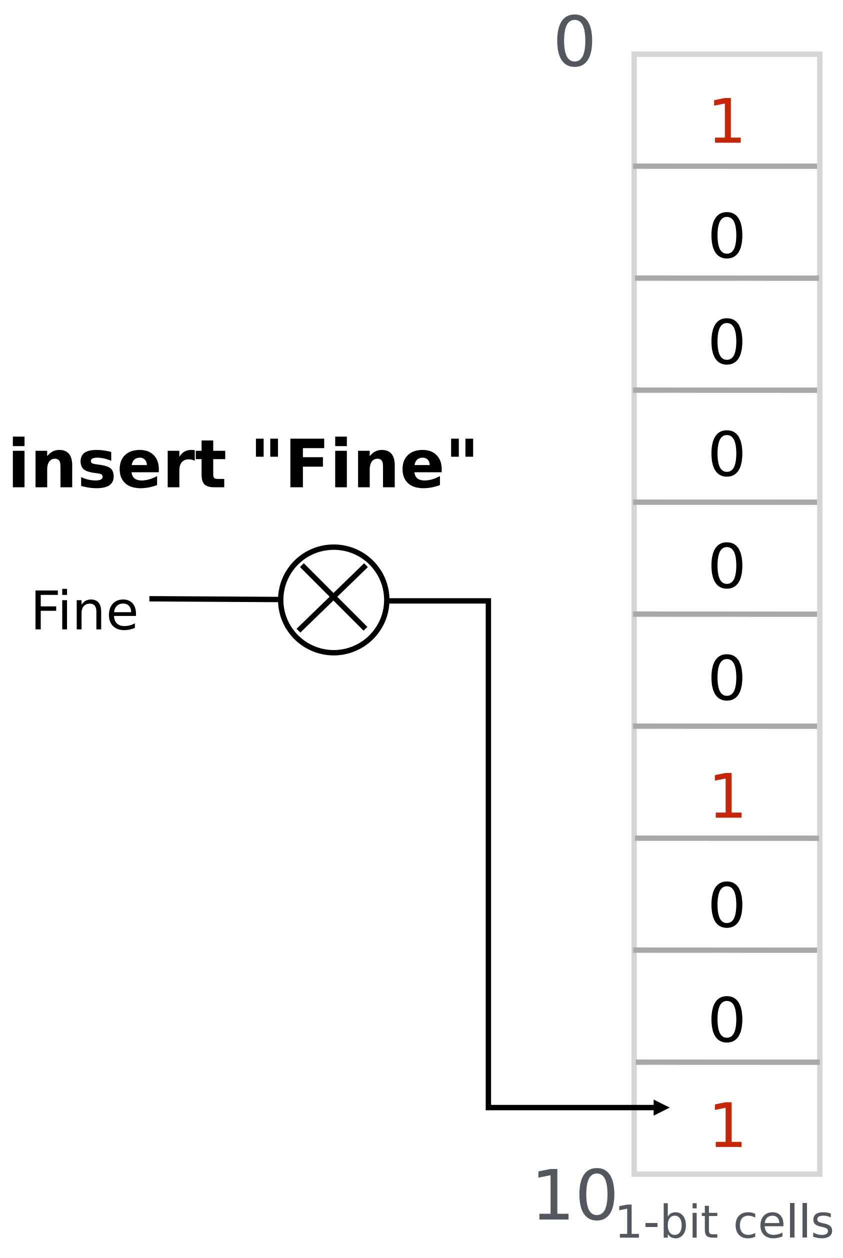
A simple approach for insertions and membership queries



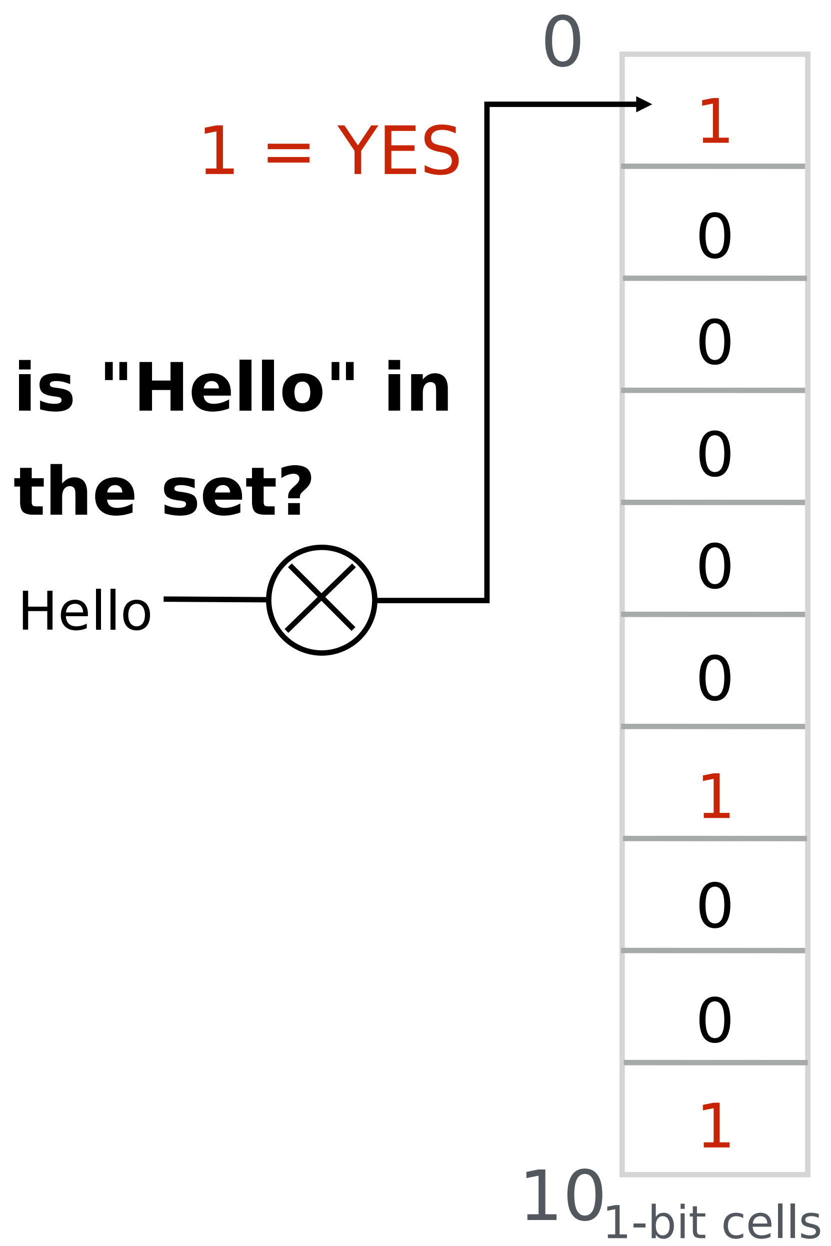
A simple approach for insertions and membership queries



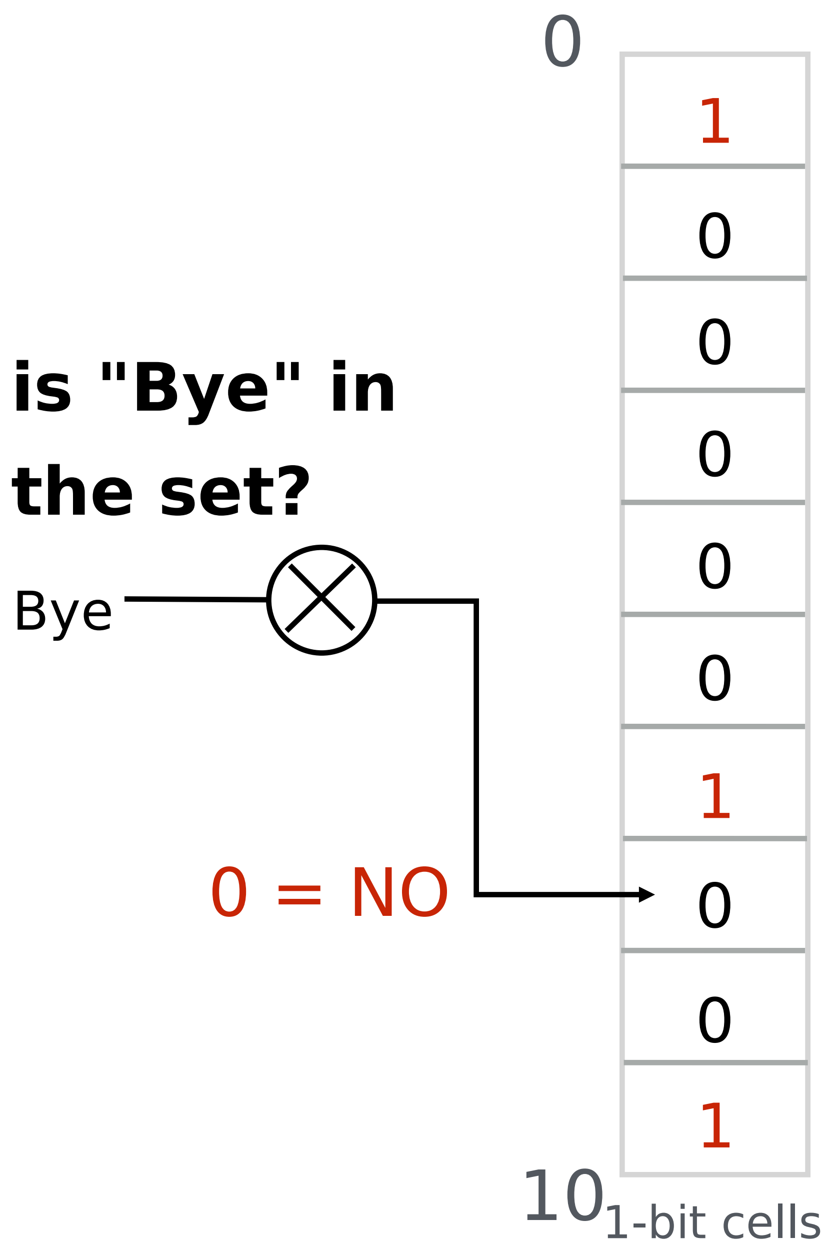
A simple approach for insertions and membership queries



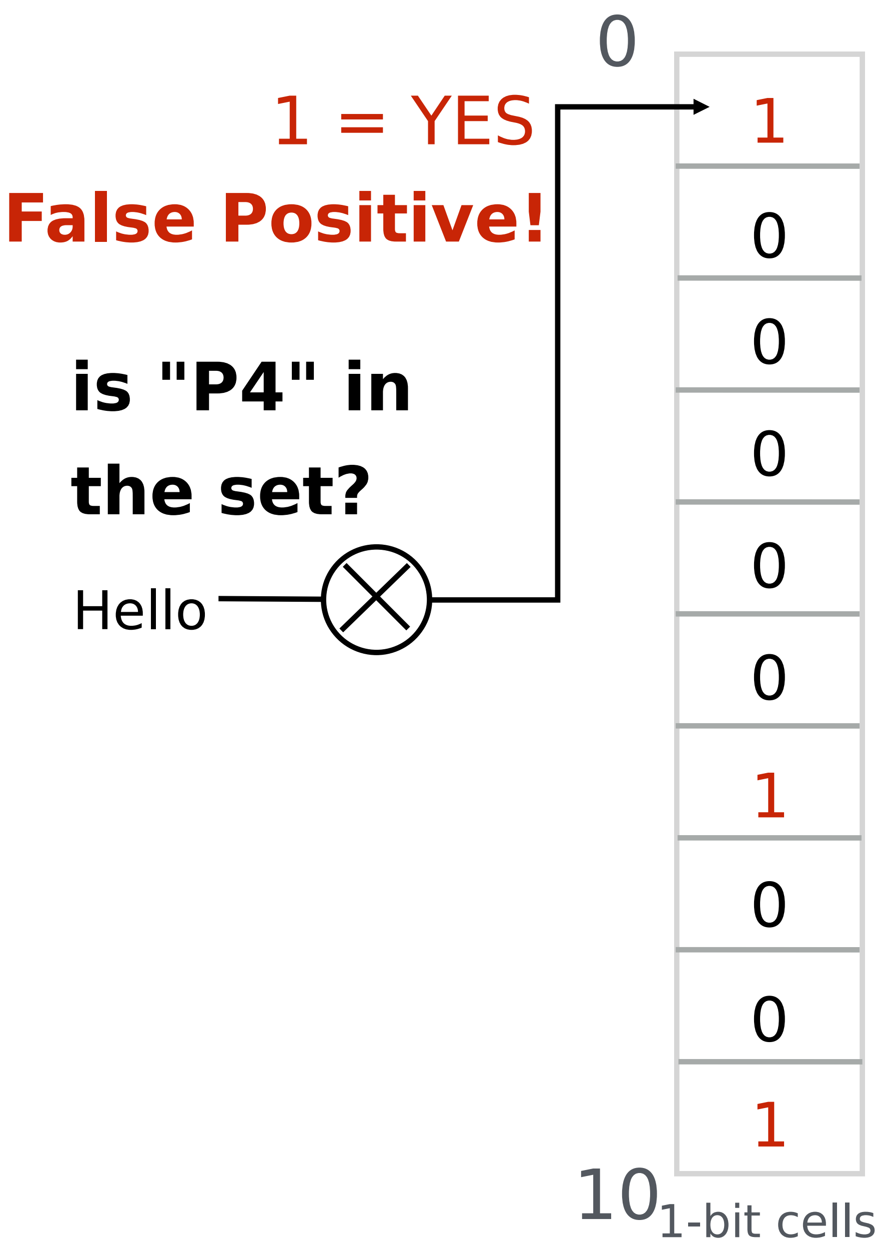
A simple approach for insertions and membership queries



A simple approach for insertions and membership queries



A simple approach for insertions and membership queries



A simple approach for insertions and membership queries

N elements and M cells

probability of an element to be mapped into a particular cell	$\frac{1}{M}$
probability of an element not to be mapped into a particular cell	$1 - \frac{1}{M}$
probability of a cell to be 0	$(1 - \frac{1}{M})^N$
false positive rate (FPR)	$1 - (1 - \frac{1}{M})^N$
false negative rate	0

A simple approach for insertions and membership queries

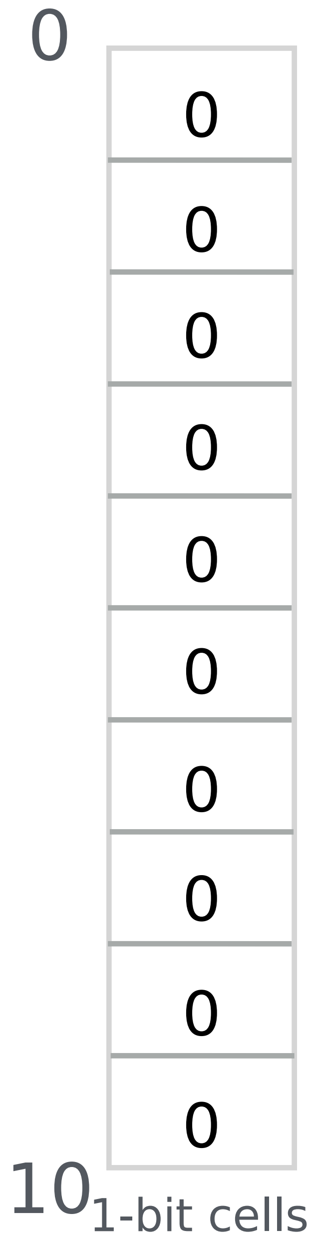
# of elements	# of cells	FPR
1000	10000	9.5%
1000	100000	1%

A simple approach for insertions and membership queries

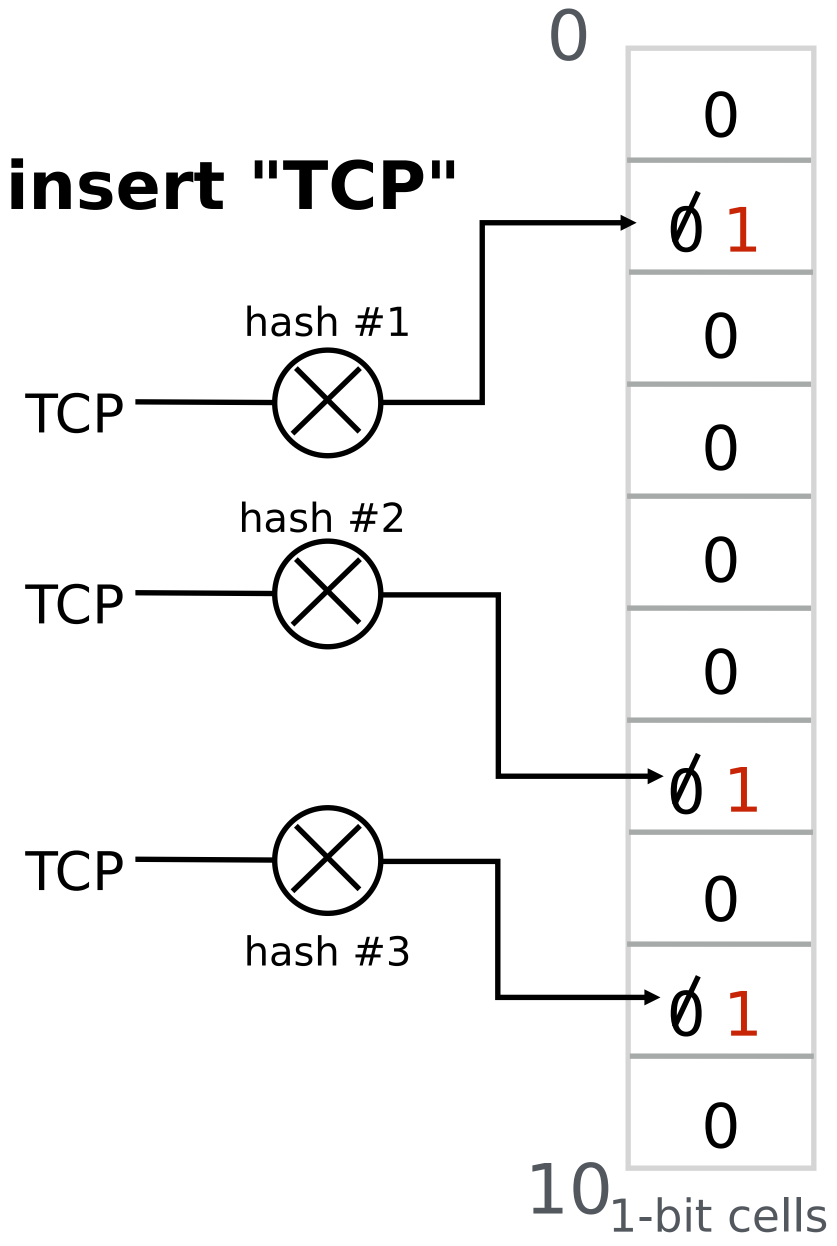
Pros: simple and only one operation per insertion or query

Con: roughly 100x more cells are required than the number of element we want to store for a 1% false positive rate

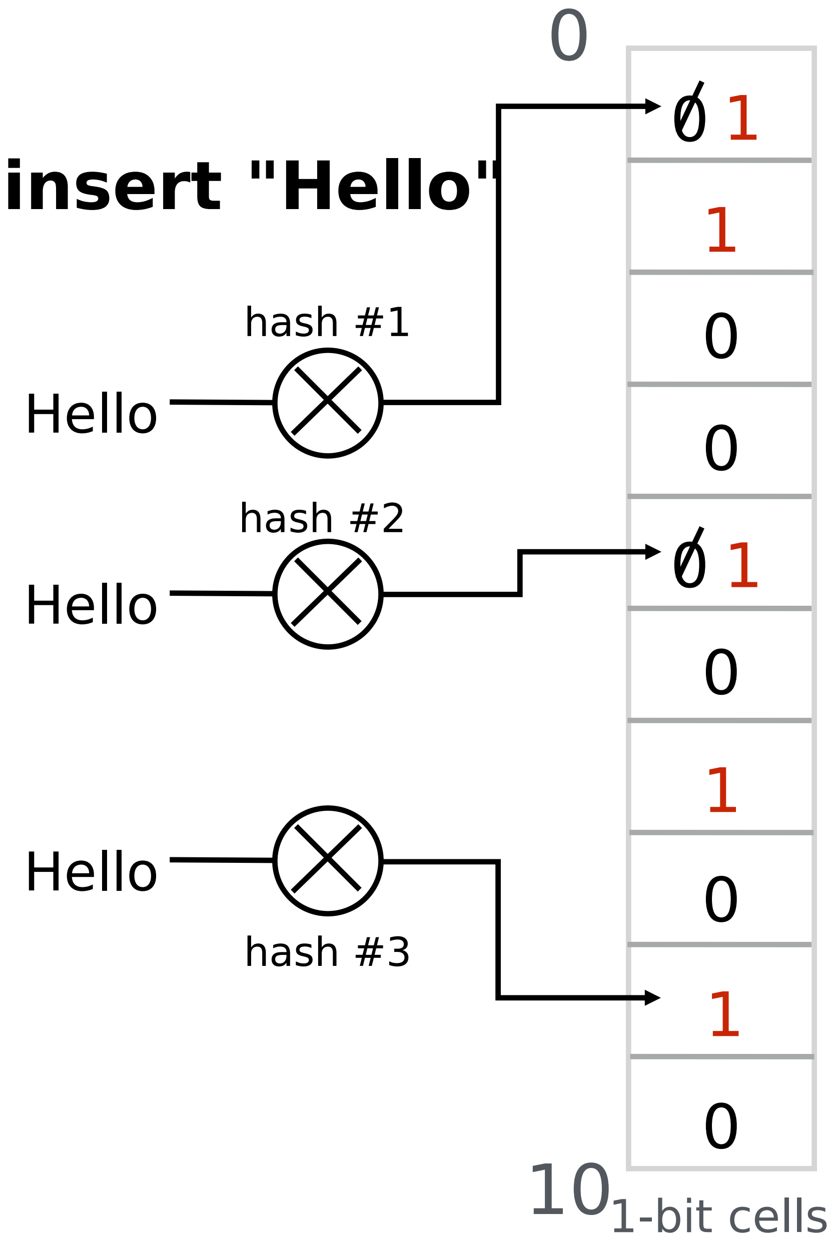
Bloom Filters: a more memory-efficient approach for insertions and membership queries



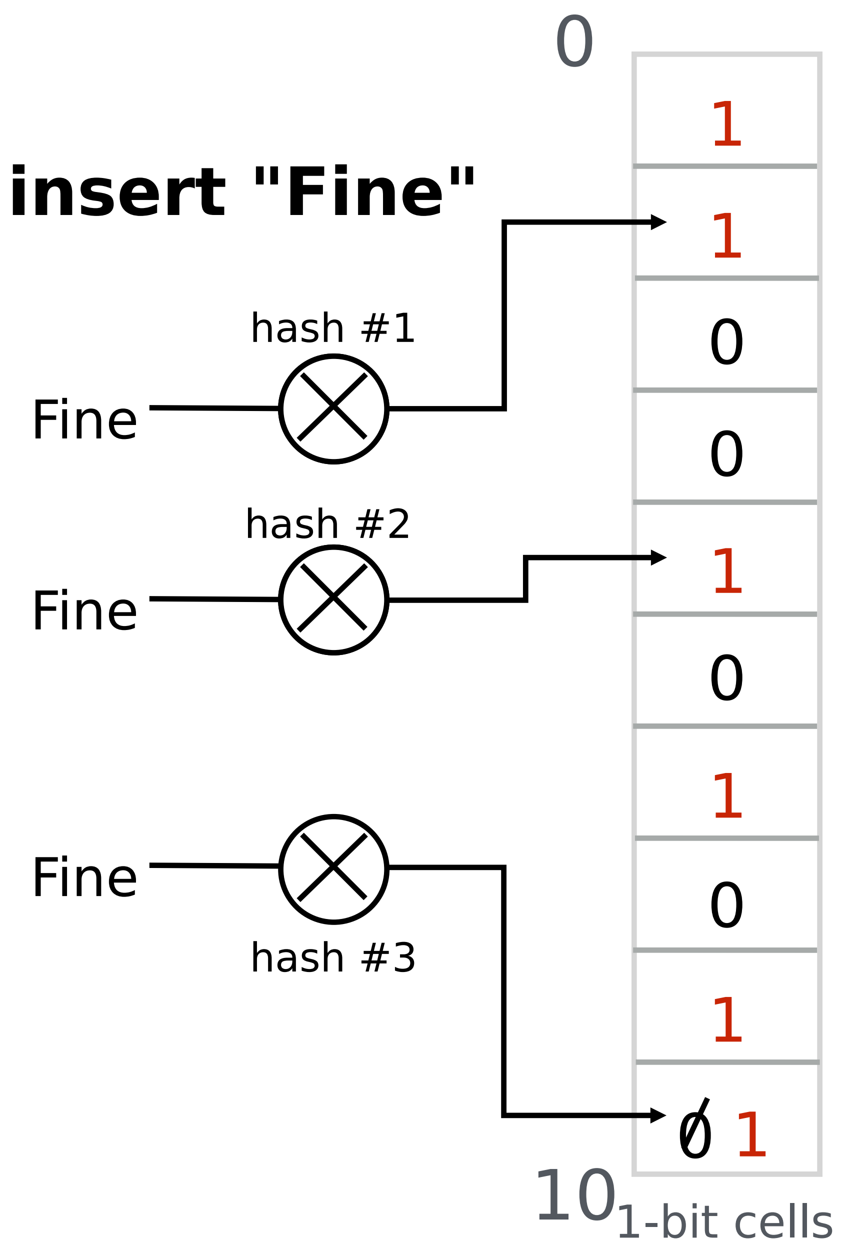
Bloom Filters: a more memory-efficient approach for insertions and membership queries



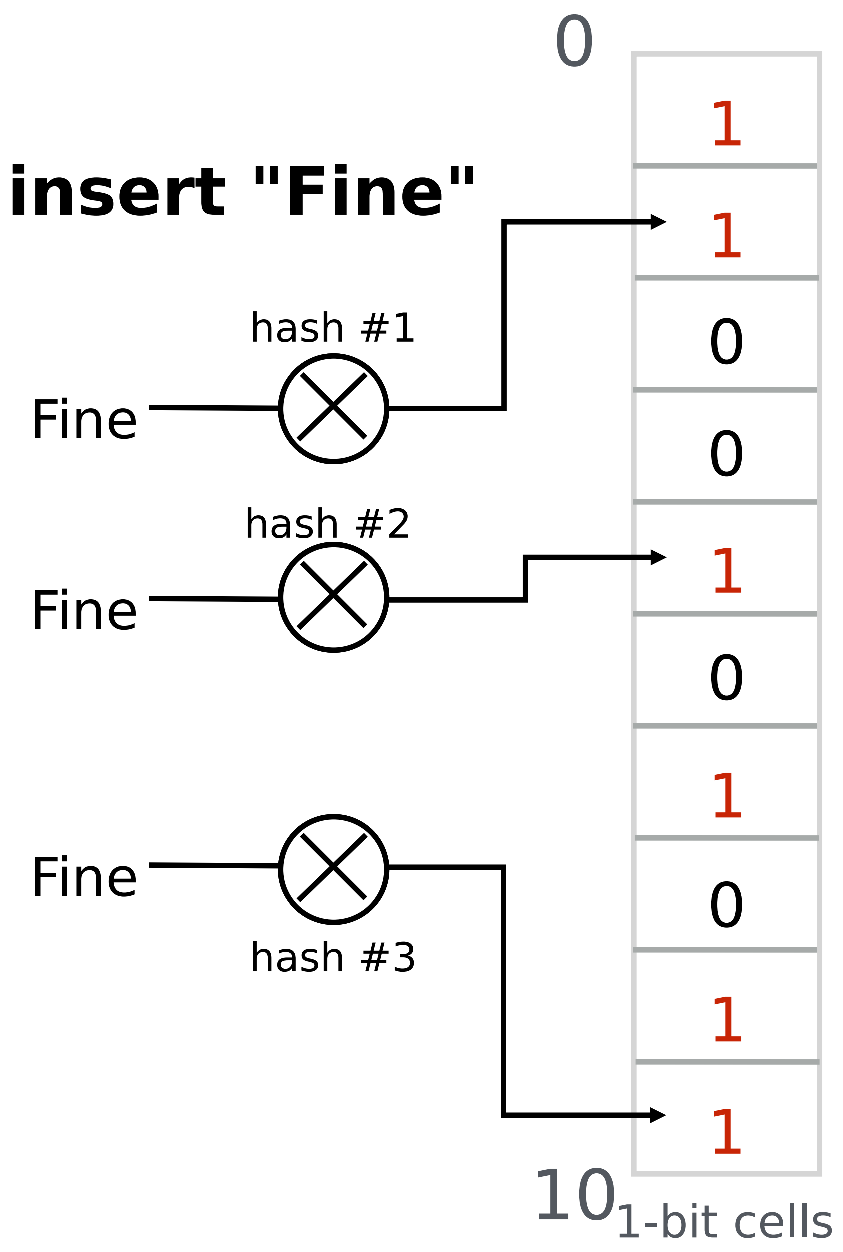
Bloom Filters: a more memory-efficient approach for insertions and membership queries



Bloom Filters: a more memory-efficient approach for insertions and membership queries



Bloom Filters: a more memory-efficient approach for insertions and membership queries



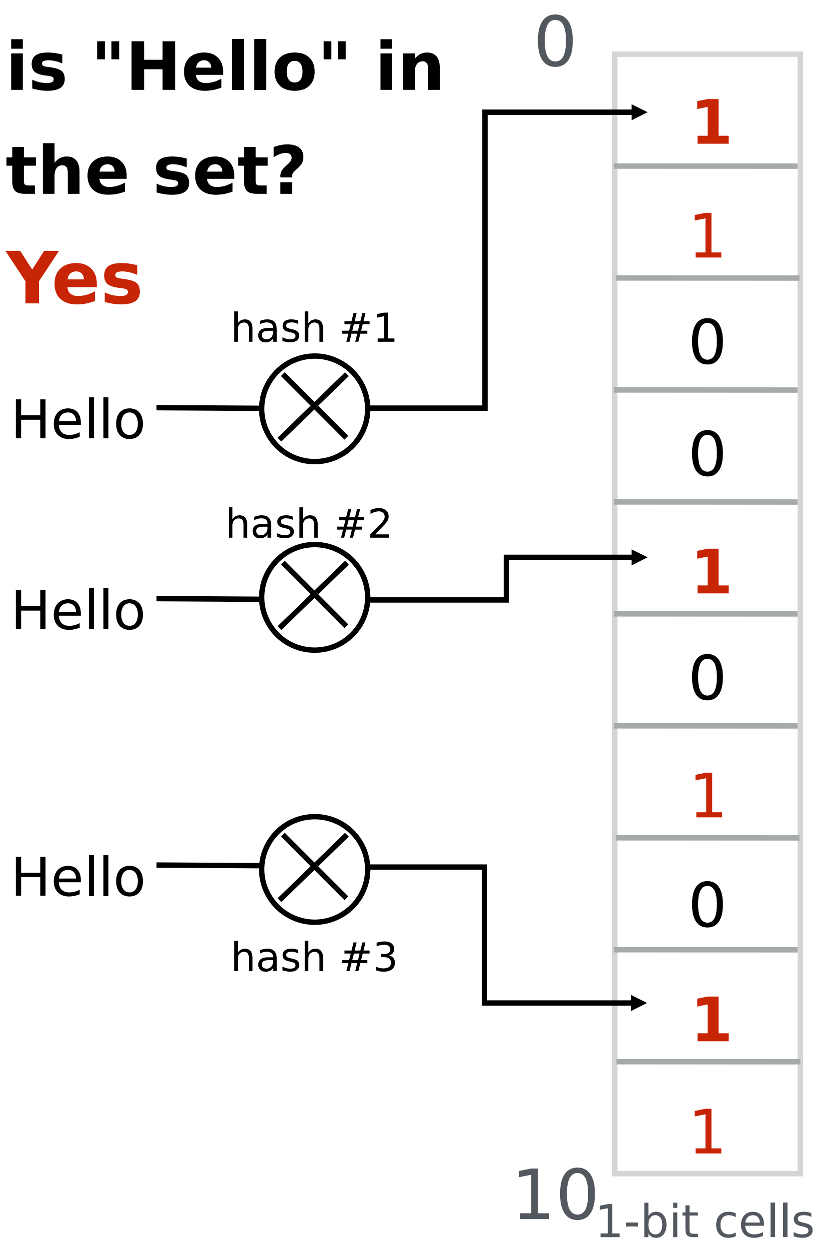
An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

is "Hello" in the set?

Yes

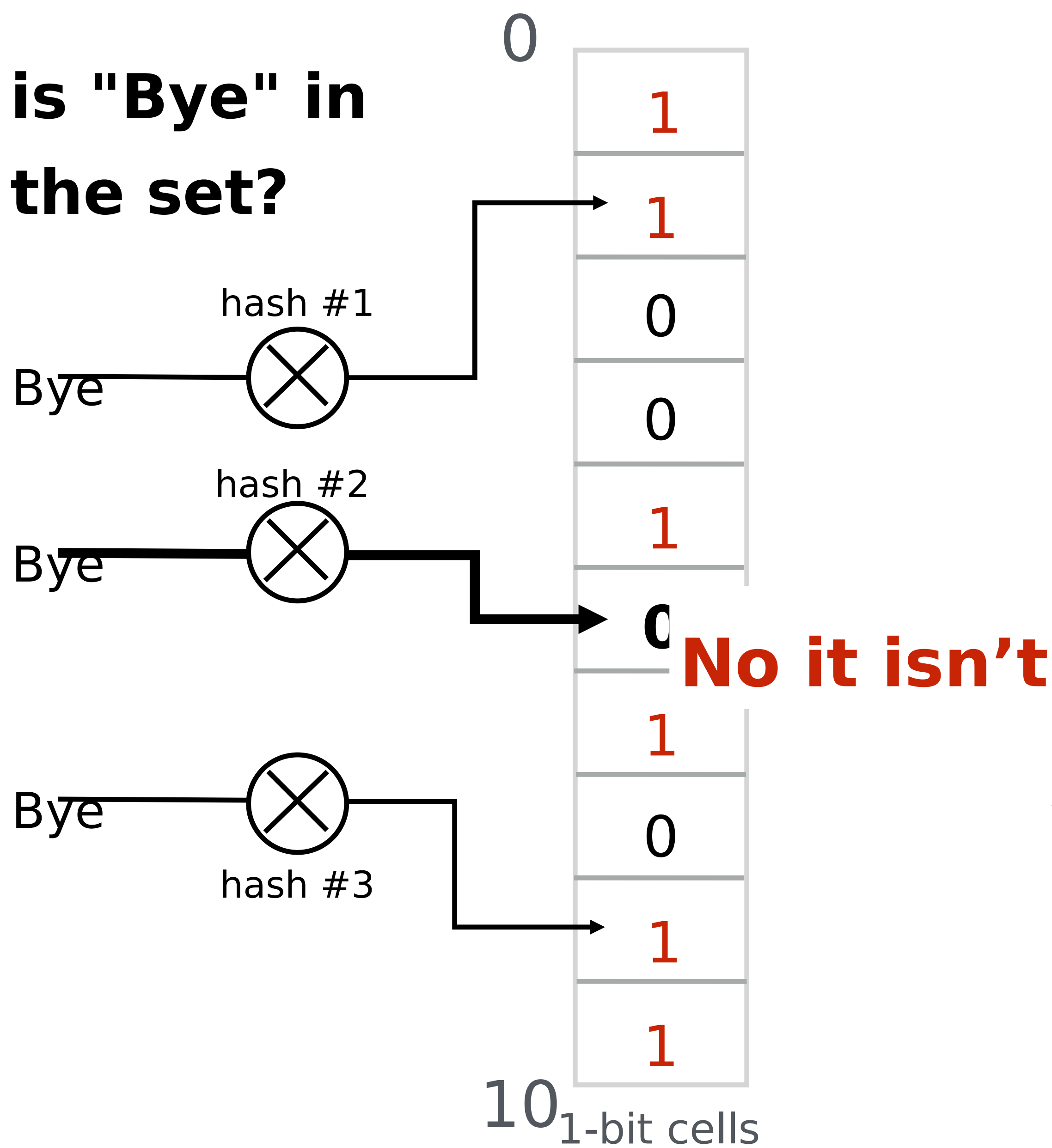


An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

is "Bye" in the set?



An element is considered in the set if **all** the hash values map to a cell with 1

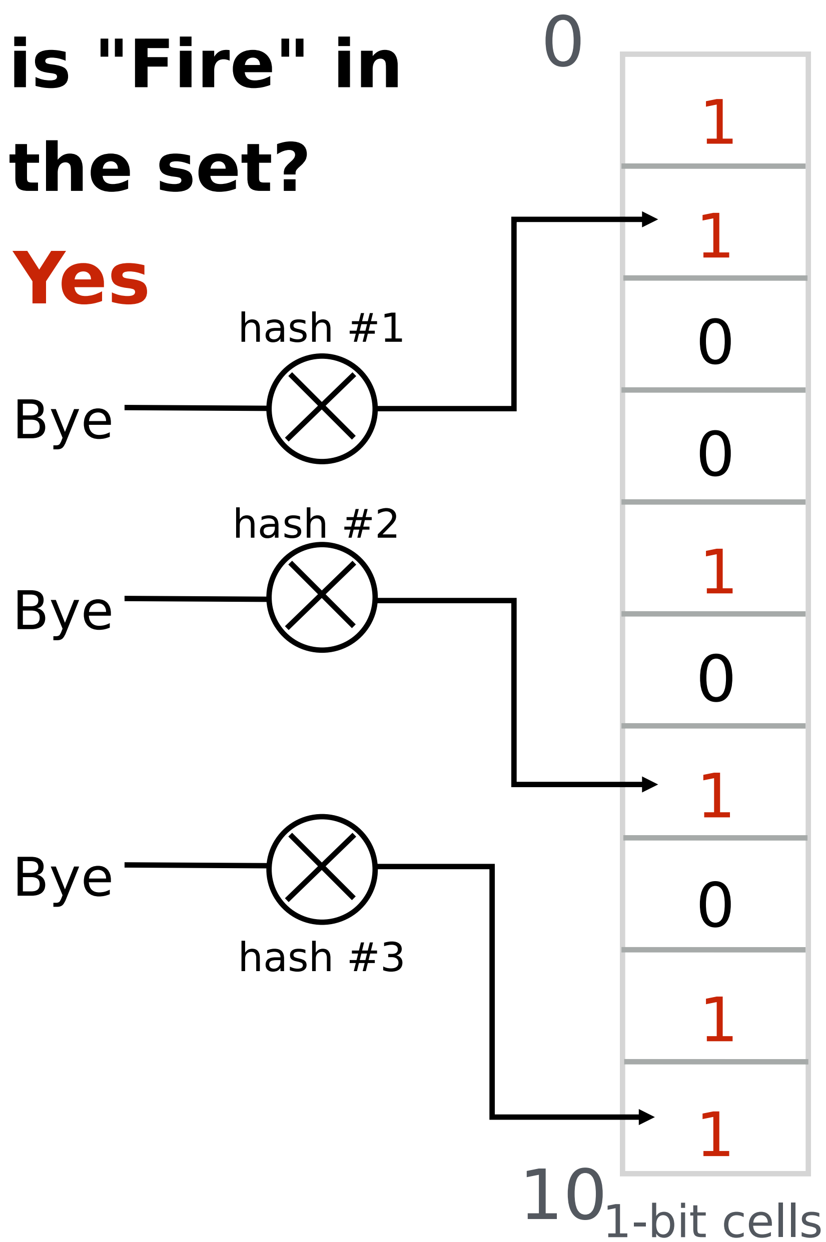
An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

False Positive!

is "Fire" in the set?

Yes



An element is considered in the set if **all** the hash values map to a cell with 1

An element is not in the set if **at least** one hash value maps to a cell with 0

Bloom Filters: a more memory-efficient approach for insertions and membership queries

N elements, M cells and K *independent* hash functions

probability that one hash function returns the index of a particular cell

$$\frac{1}{M}$$

probability that one hash function does not return the index of a particular cell

$$1 - \frac{1}{M}$$

probability of a cell to be 0

$$(1 - \frac{1}{M})^{KN}$$

false positive rate

$$(1 - (1 - \frac{1}{M})^{KN})^{K}$$

false negative rate

$$0$$

Bloom Filters: a more memory-efficient approach for insertions and membership queries

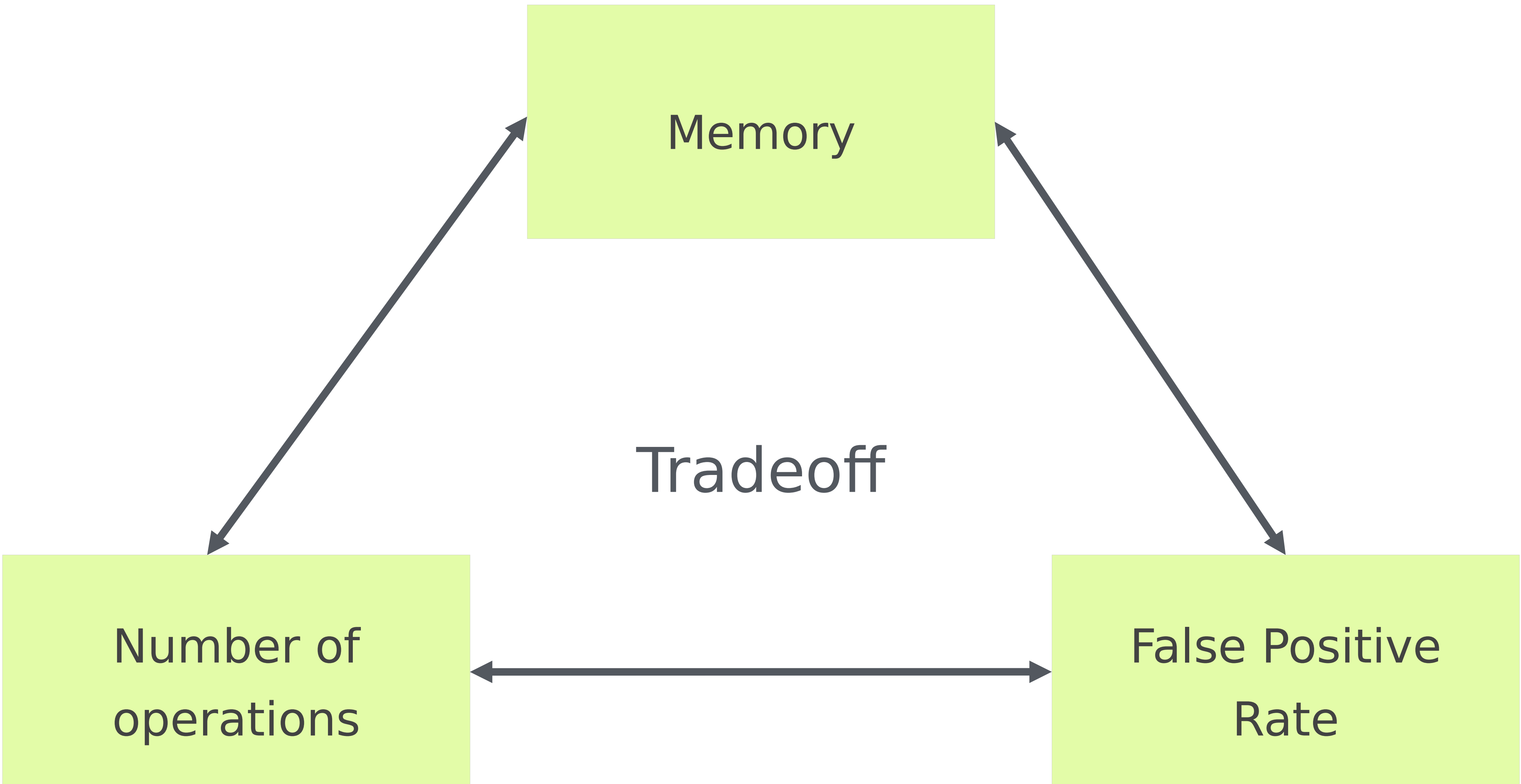
# of elements	# of cells	# hash functions	FPR
1000	10000	7	0.82%
1000	100000	7	$\approx$ 0%

Bloom Filters: a more memory-efficient approach for insertions and membership queries

Pro: consumes roughly 10x less memory than the simple approach

Con: Requires slightly more operations than the simple approach (7 hashes instead of just 1)

Dimension your Bloom Filter



Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate

Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate

asymptotic approx.

$$FP = (1 - (1 - \frac{1}{M})^{KN})^K \boxed{\approx (1 - e^{-KN/M})^K}$$

with calculus you can

dimension your bloom filter

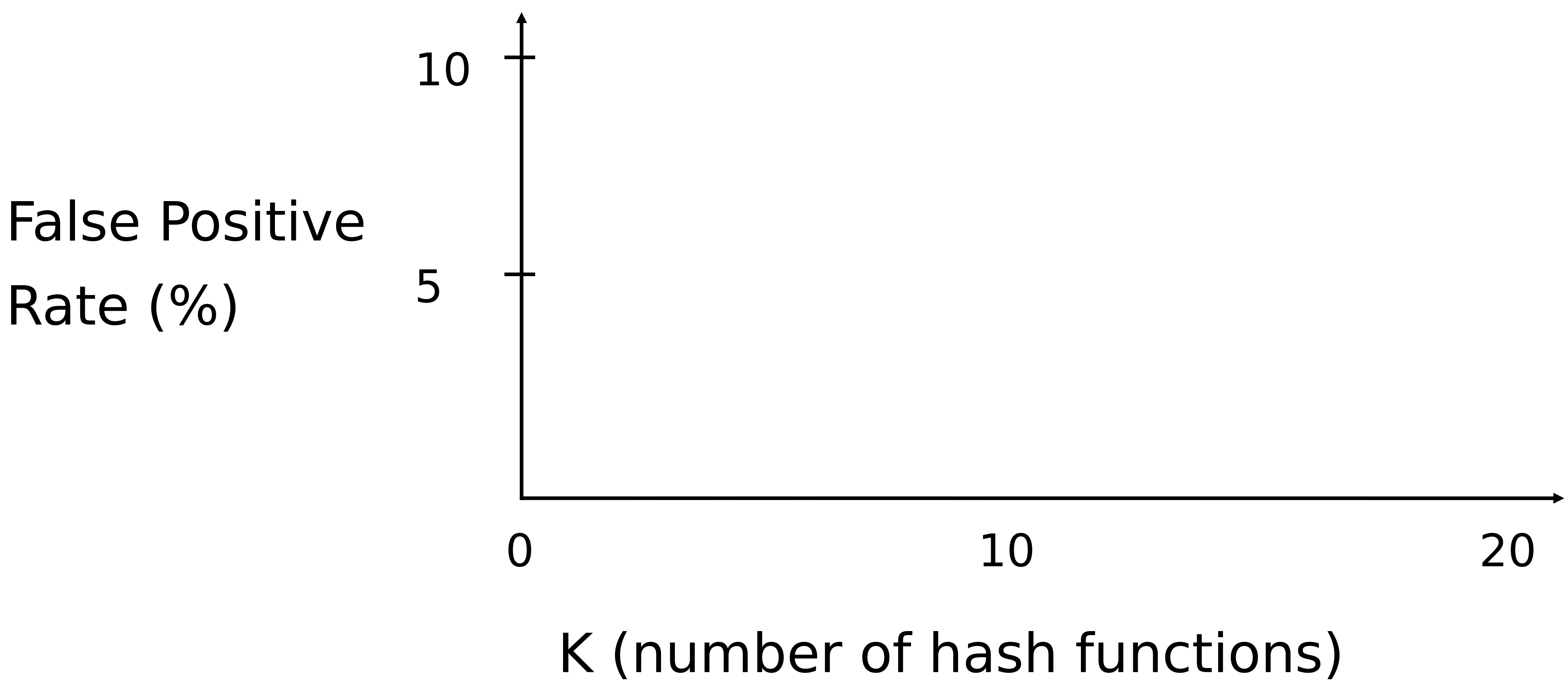
Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate



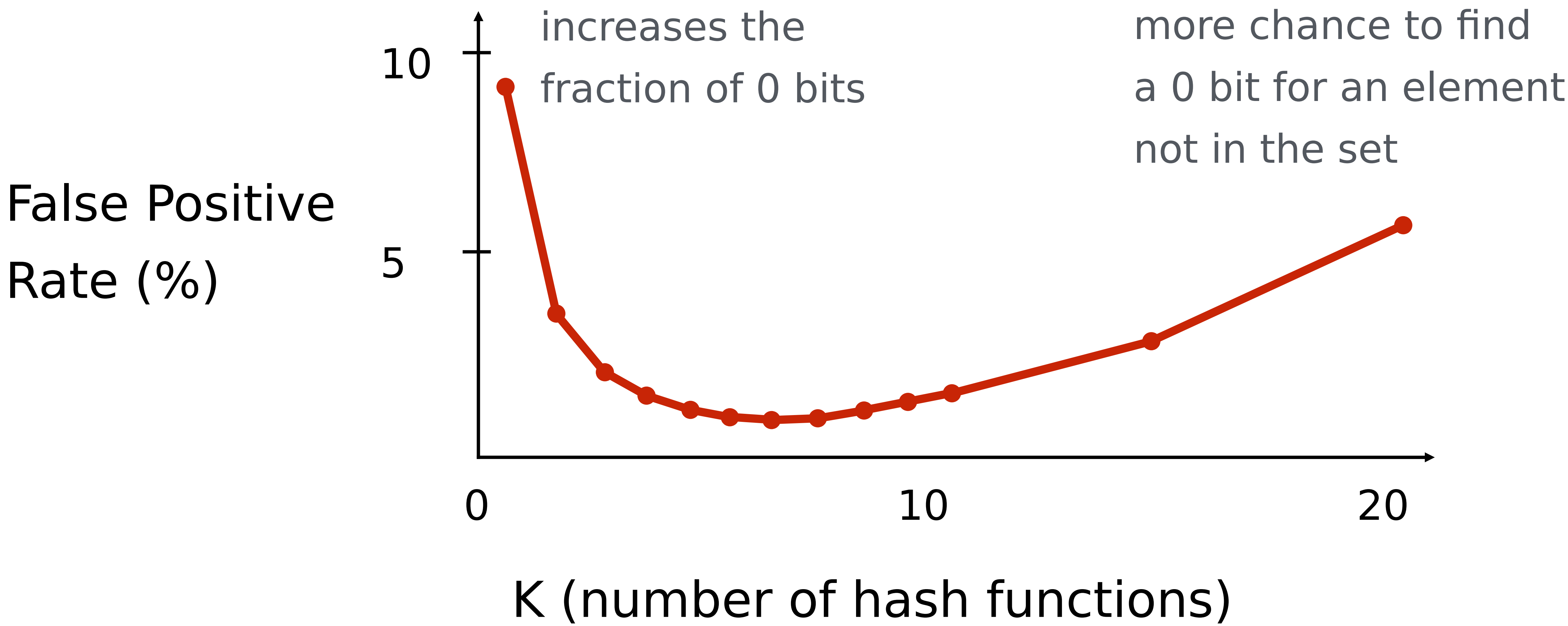
Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate



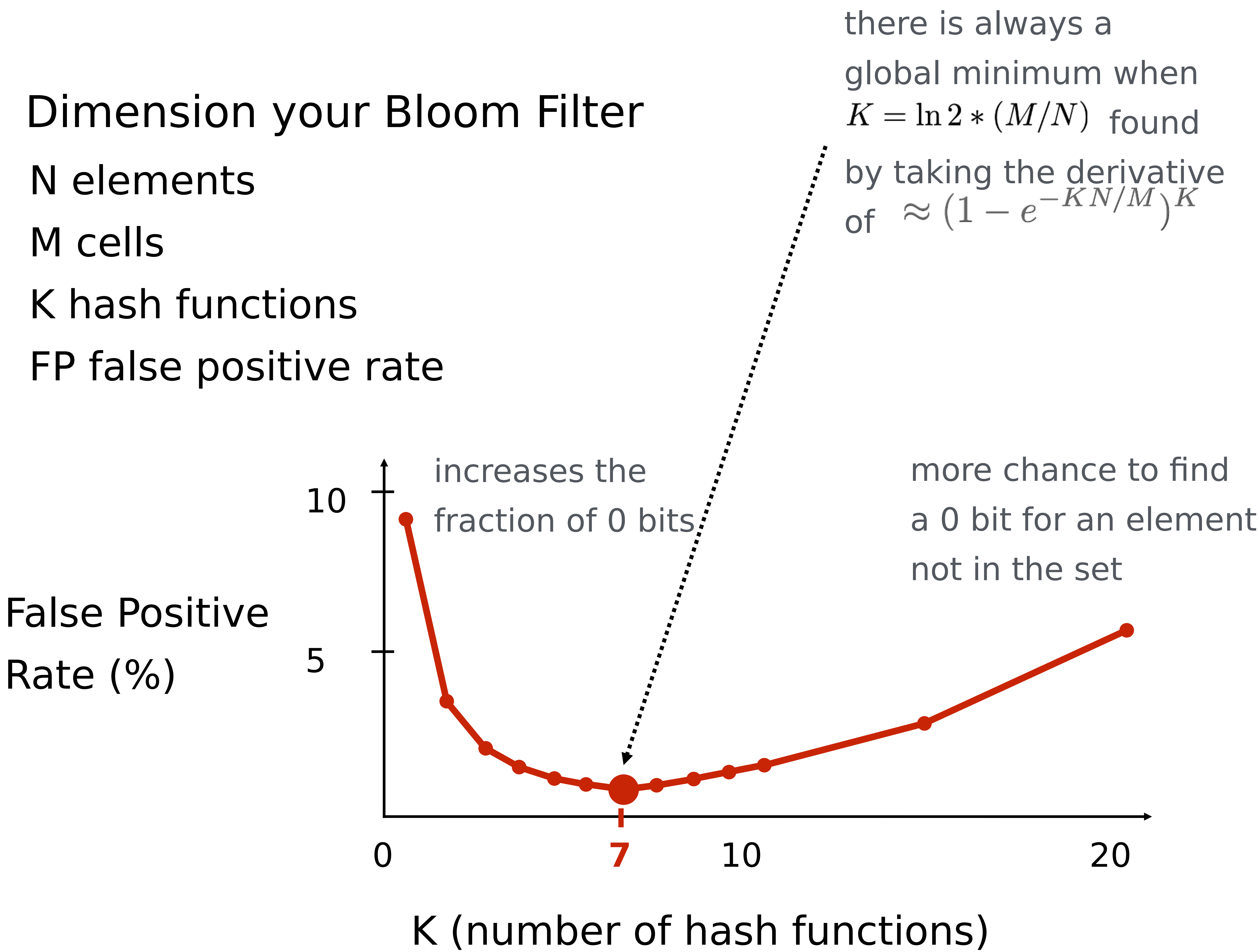
Dimension your Bloom Filter

N elements

M cells

K hash functions

FP false positive rate



Implementation of a Bloom Filter in $P4_{16}$

You will have to use hash functions

v1model

```
enum HashAlgorithm {
    crc32,
    crc32_custom,
    crc16,
    s,
    random,
    identity,
    csum16,
    xor16
}
```

```
extern void hash<O, T, D, M>(out O result,
  in HashAlgorithm algo, in T base, in D data, in M
max);
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Implementation of a Bloom Filter in $P4_{16}$

You will have to use hash functions, as well as registers

v1model

```
extern register<T> {

    register(bit<32> size);

    void read(out T result, in bit<32> index);
    void write(in bit<32> index, in T value);
}
```

more info https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

       if (meta.to_insert == 1) {
         bloom_filter.write(meta.index1, 1);
         bloom_filter.write(meta.index2, 1);
       }
       if (meta.to_query == 1) {
         bloom_filter.read(meta.query1, meta.index1);
         bloom_filter.read(meta.query2, meta.index2);

         if (meta.query1 == 0 || meta.query2 == 0) {
           meta.is_stored = 0;
         }
         else {
           meta.is_stored = 1;
         }
       }
    }
  }
```

Implementation of a Bloom Filter in $P4_{16}$

with 2 hash functions

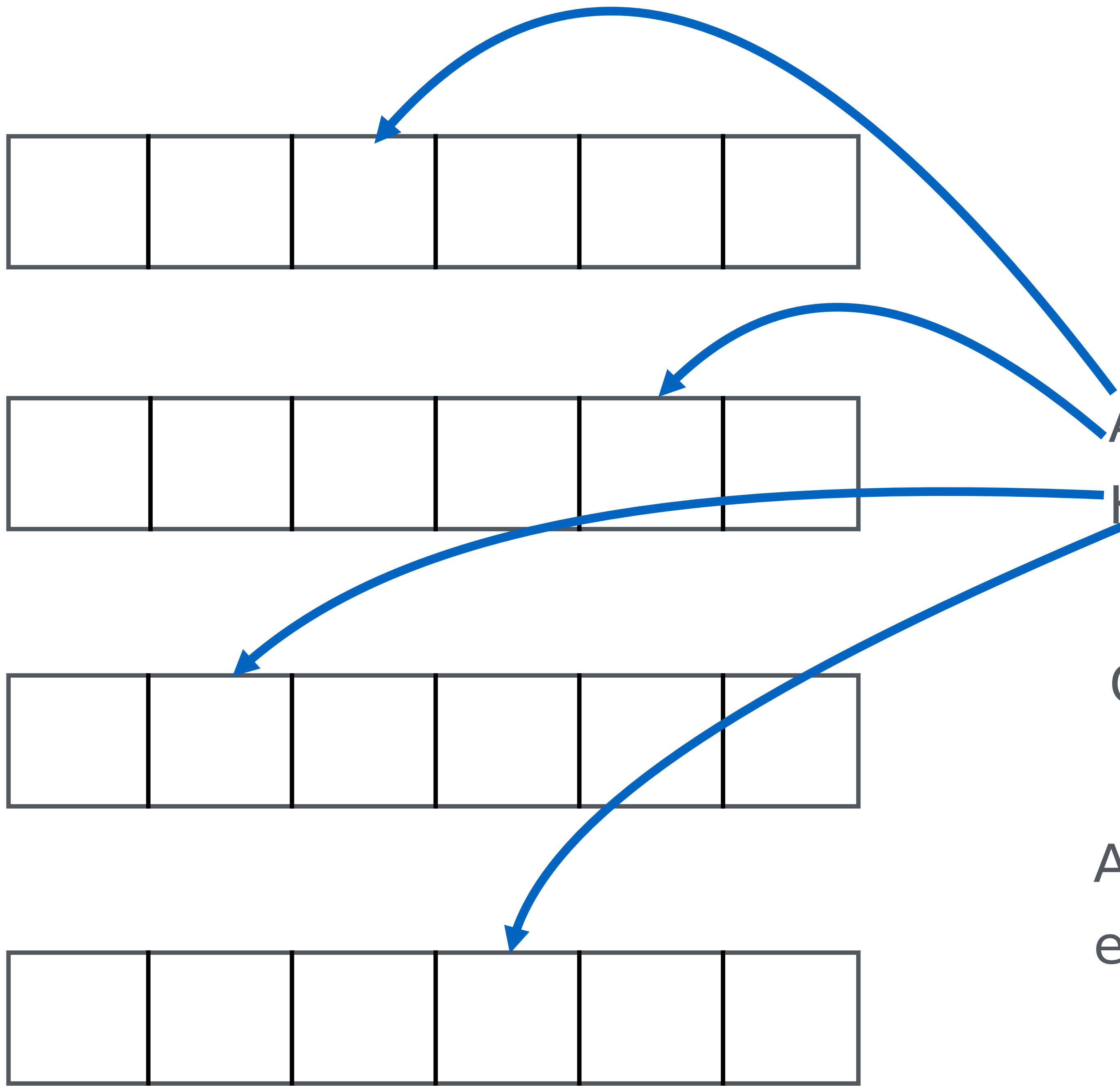
```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

       if (meta.to_insert == 1) {
         bloom_filter.write(meta.index1, 1);
         bloom_filter.write(meta.index2, 1);
       }
       if (meta.to_query == 1) {
         bloom_filter.read(meta.query1, meta.index1);
         bloom_filter.read(meta.query2, meta.index2);

         if (meta.query1 == 0 || meta.query2 == 0) {
           meta.is_stored = 0;
         }
         else {
           meta.is_stored = 1;
         }
       }
    }
  }
```

Everything in bold red must be adapted for your program

Depending on the hardware limitations, splitting the bloom filter might be required



M cells are split into M/K disjoint groups

An element is hashed to K cells, one in each group

One hash function per group

About same performance, may be easier to implement or parallelize

Because deletions are not possible, the controller may need to regularly reset the bloom filters

Resetting a bloom filter takes some time during which it is not usable

Common trick: use two bloom filters and use one when the controller resets the other one

Bloom filters may be extended to allow deletions and to list the filter content.

If you are curious, check out the extended slides for:

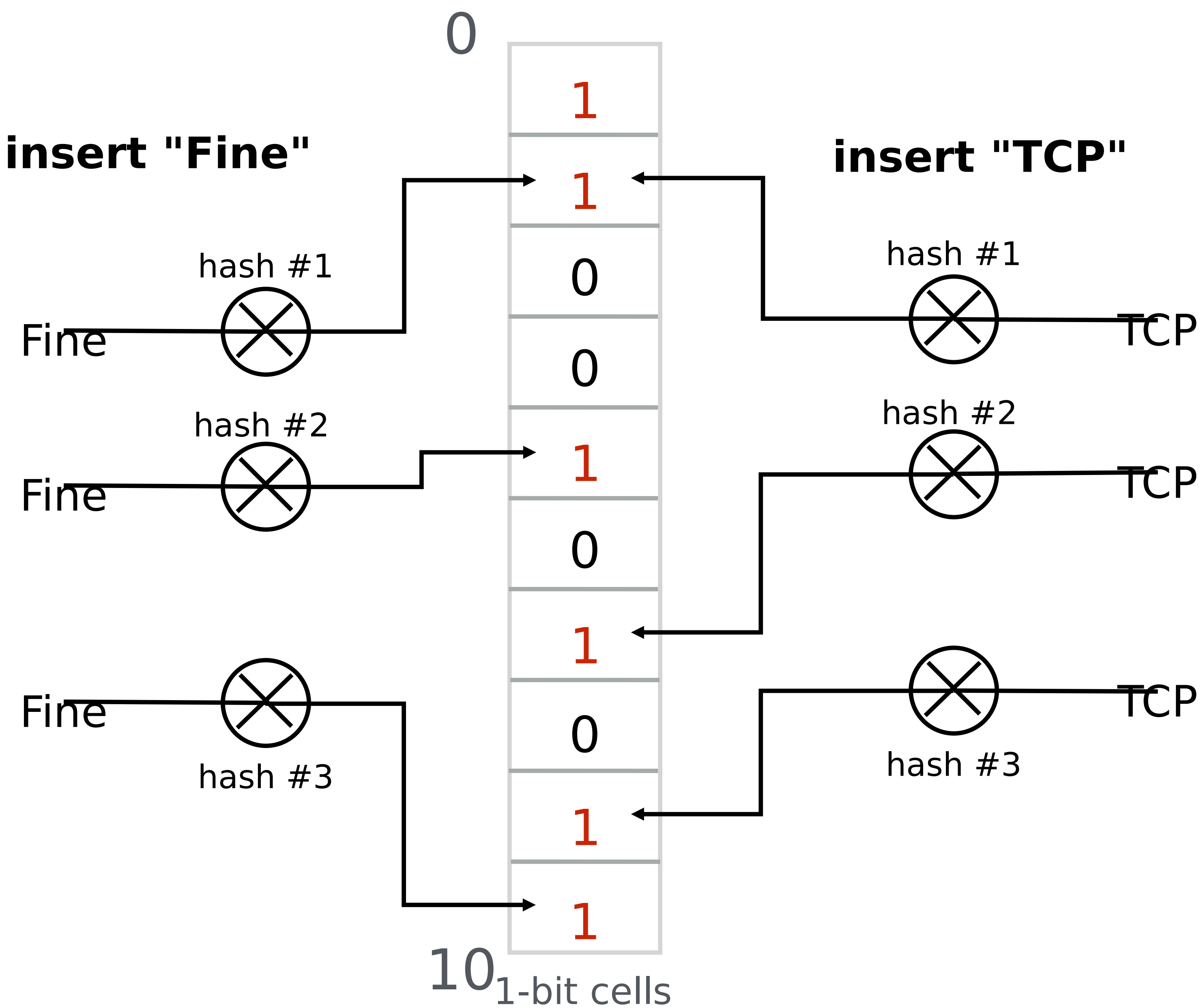
- counting Bloom filters (allow deletions)
- invertible Bloom filters (allow to list content)

So far we have seen how to do insertions and membership queries

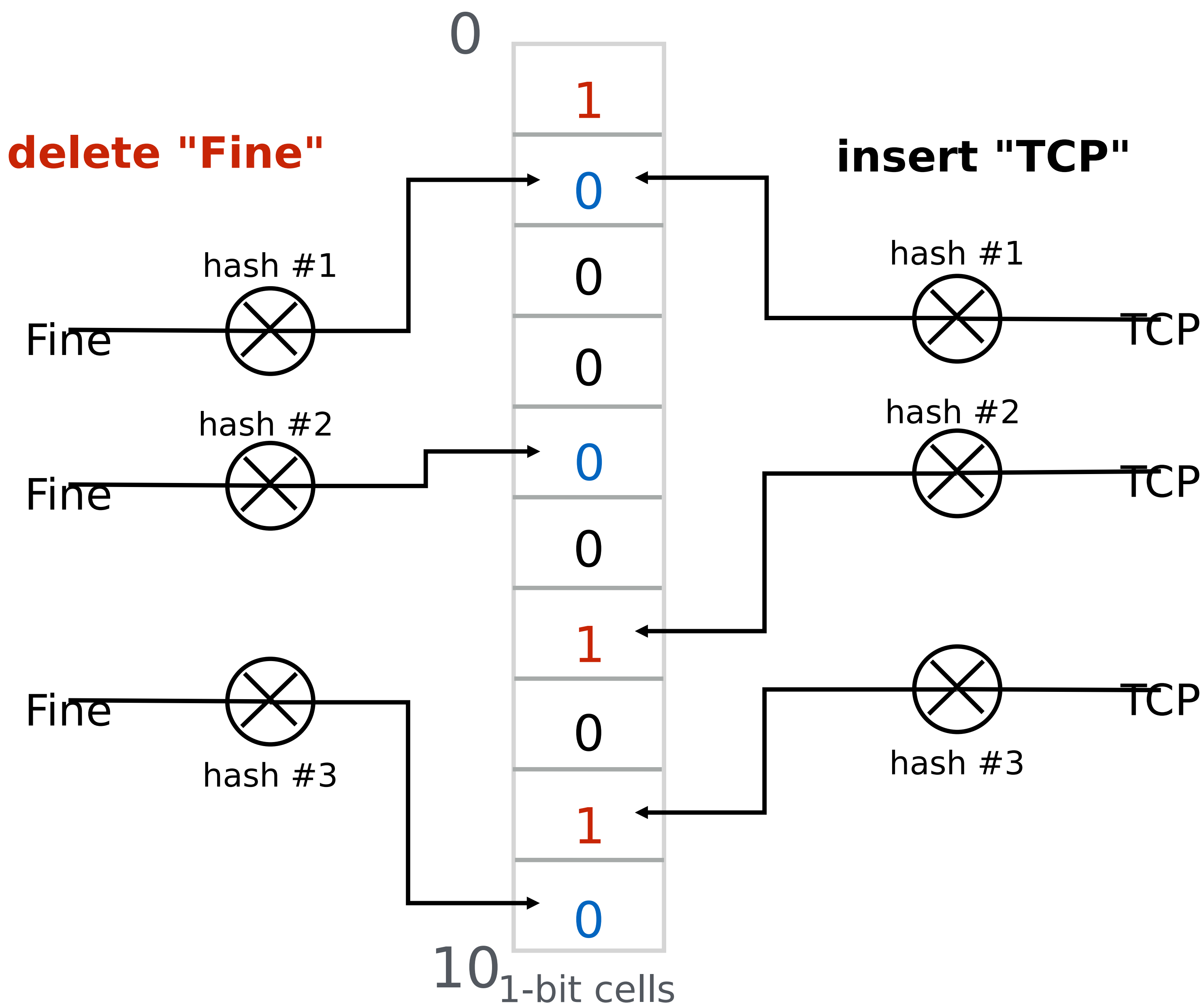
	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

Bloom Filters

However Bloom Filters do not handle deletions



However Bloom Filters do not handle deletions



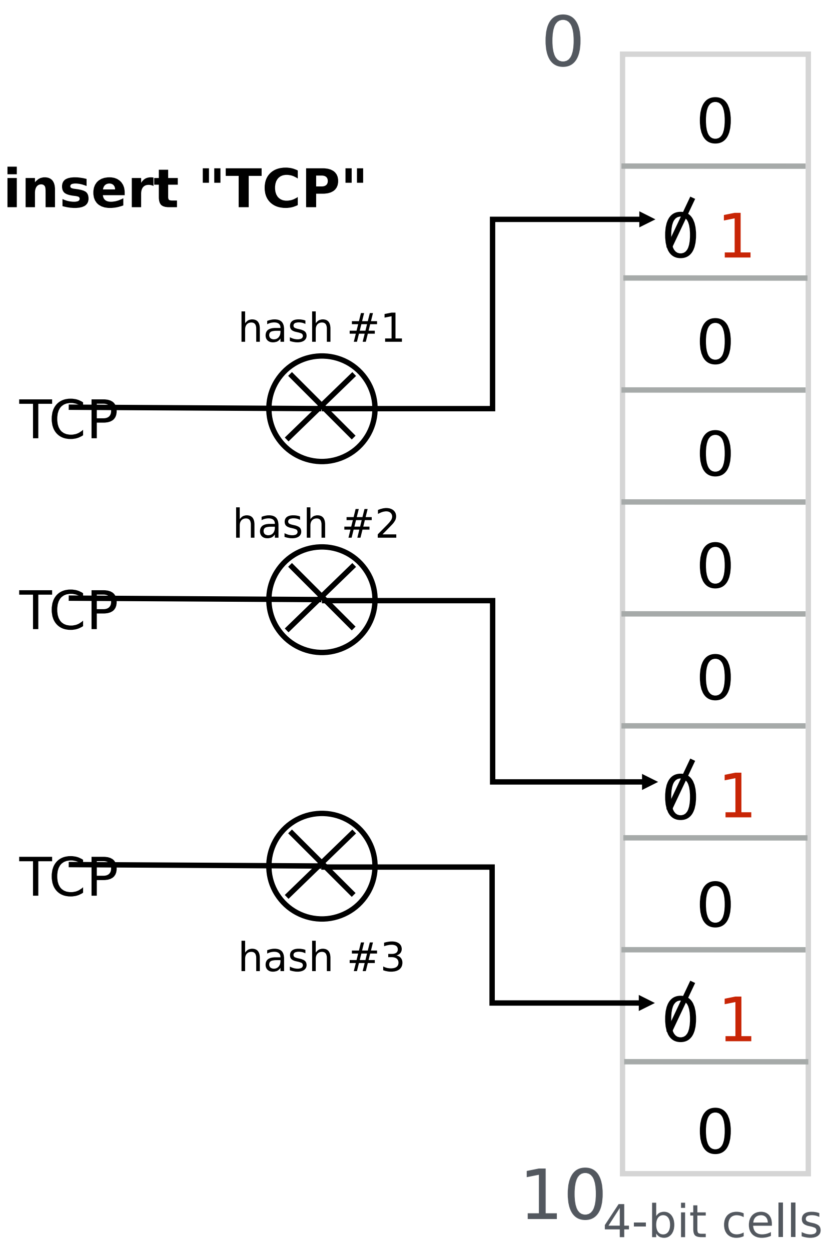
If deleting an element means resetting 1s to 0s, then deleting "Fine" also deletes "TCP"

But we can easily extend them to handle deletions

This extended version is called a Counting Bloom Filter

But we can easily extend them to handle deletions

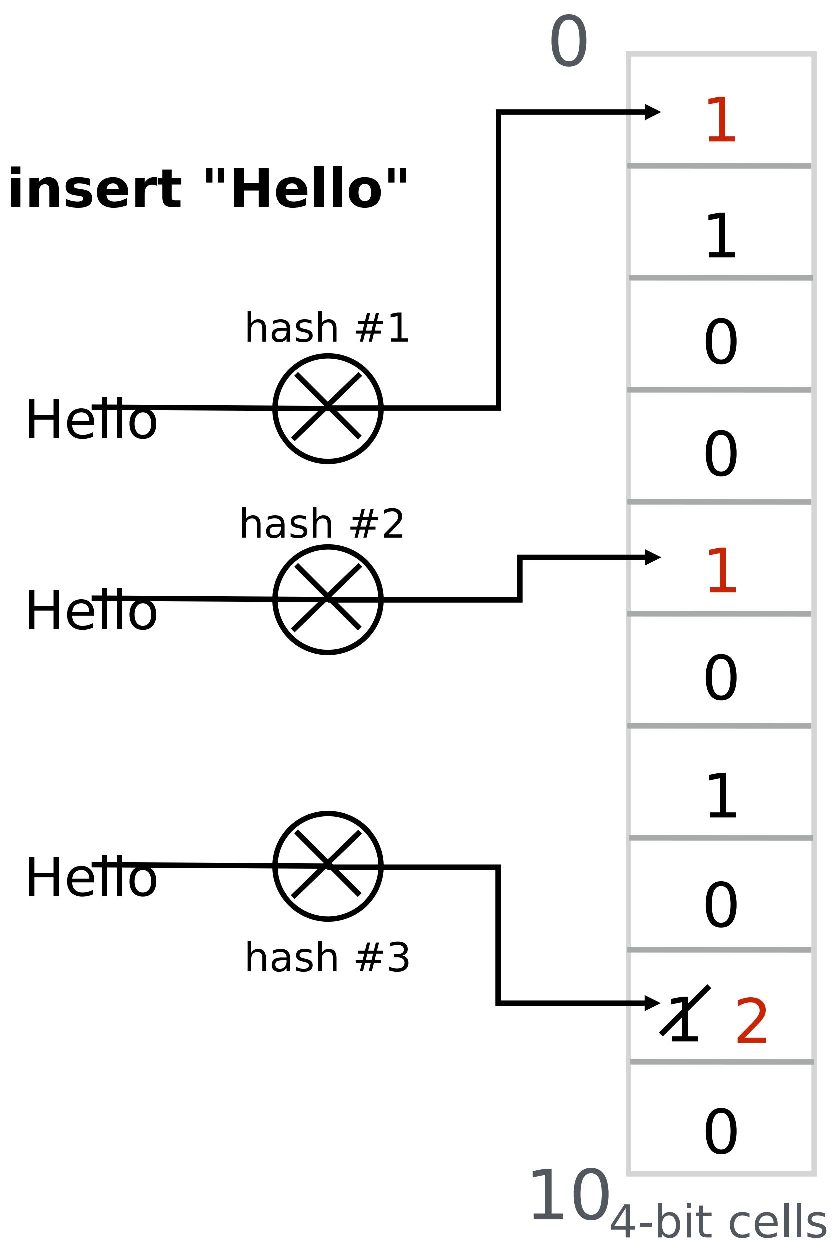
This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

But we can easily extend them to handle deletions

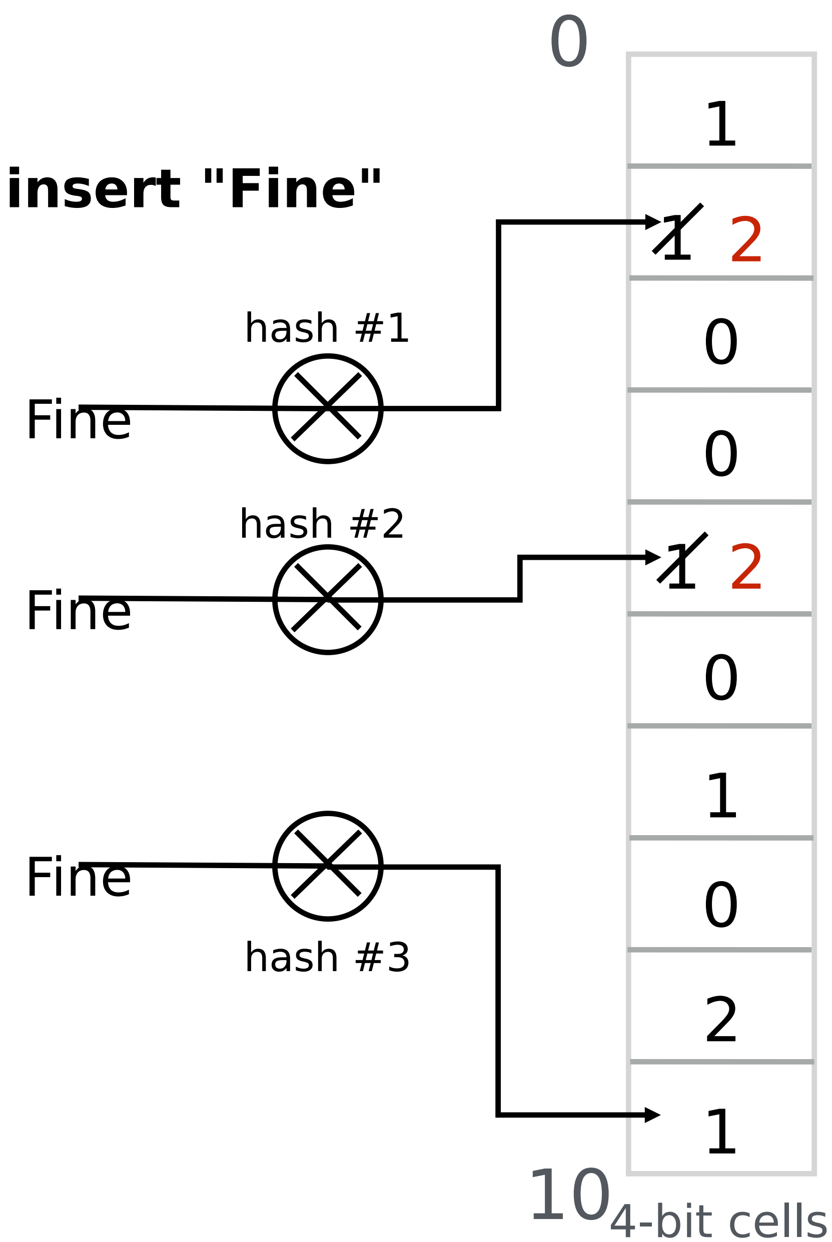
This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

But we can easily extend them to handle deletions

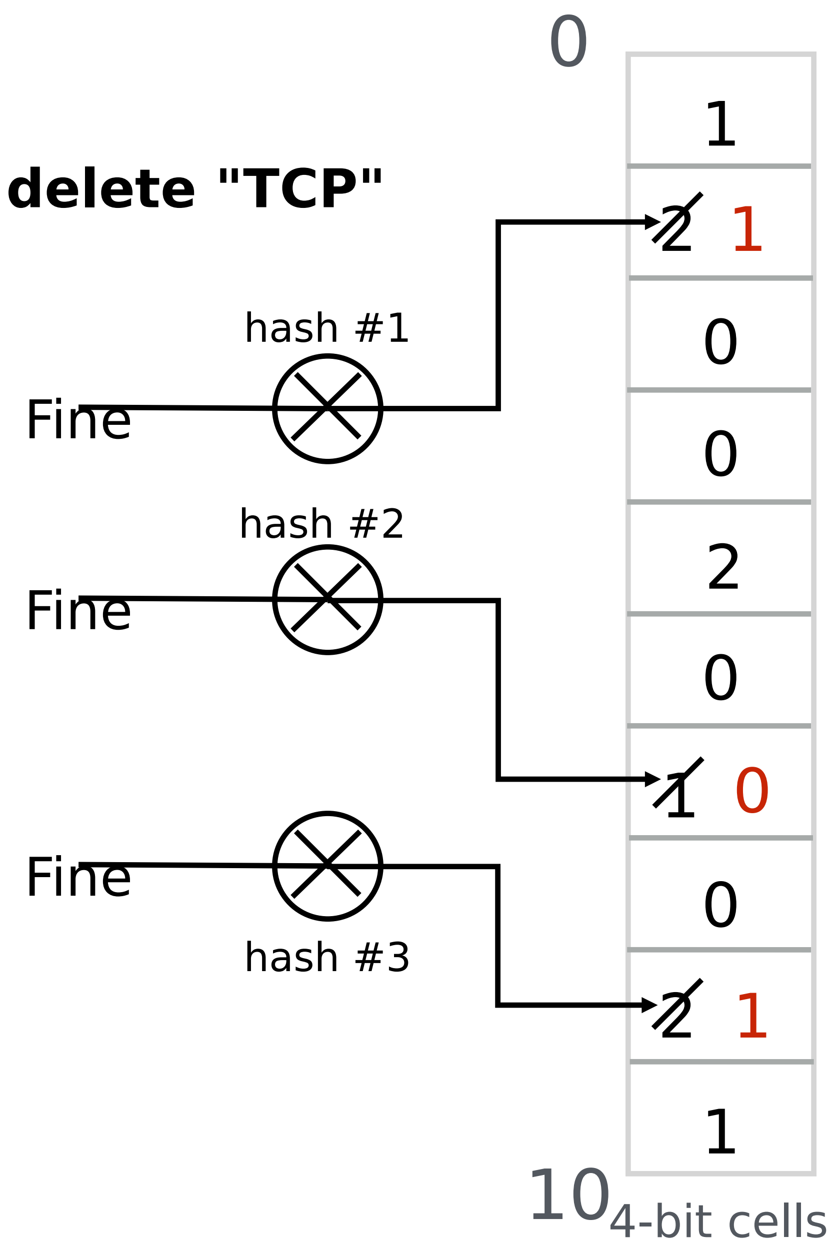
This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

But we can easily extend them to handle deletions

This extended version is called a Counting Bloom Filter

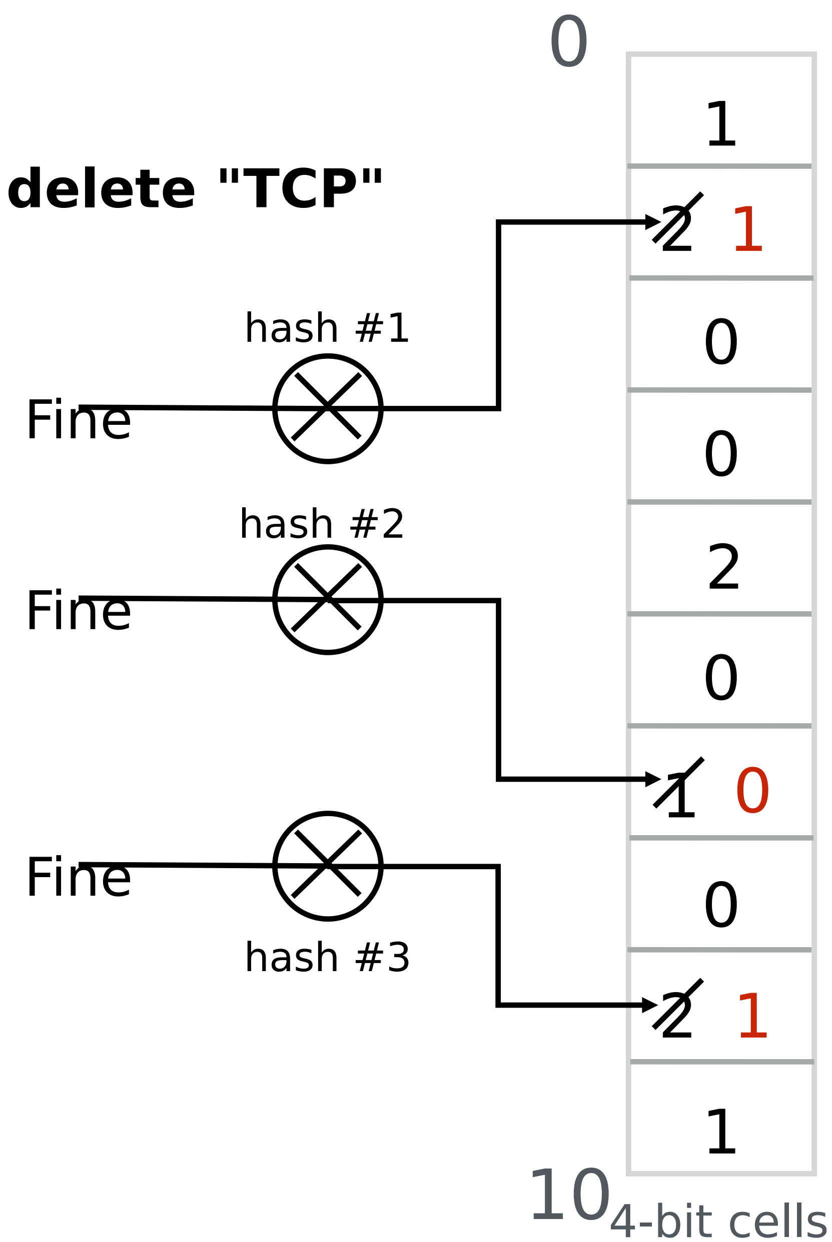


To add an element, increment the corresponding counters

To delete an element, decrement the corresponding counters

But we can easily extend them to handle deletions

This extended version is called a Counting Bloom Filter



To add an element, increment the corresponding counters

To delete an element, decrement the corresponding counters

All of our prior analysis for standard bloom filters applies to counting bloom filters

Counting Bloom Filters do handle deletions
at the price of using more memory

Counting Bloom Filters do handle deletions
at the price of using more memory

Counters must be large enough to avoid overflow

If a counter eventually overflows, the filter may yield false negatives

Counting Bloom Filters do handle deletions
at the price of using more memory

Counters must be large enough to avoid overflow

If a counter eventually overflows, the filter may yield false negatives

Poisson approximation suggests 4 bits/counter

The average load (i.e., $\frac{NK}{M}$) is $\ln 2$ assuming $K = \ln 2 * (M/N)$

With N=10000 and M=80000 the probability that some counter overflows if we use b-bit counters is at most

$M * Pr(Poisson(\ln 2) \geq 2^b) = 1.78\text{e}{-11}$

Implementation of a Counting Bloom Filter in $P4_{16}$

with 2 hash functions

Add a new element

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

       // Add a new element if not yet in the set
       bloom_filter.read(meta.query1, meta.index1);
       bloom_filter.read(meta.query2, meta.index2);

       if (meta.query1 == 0 || meta.query2 == 0) {
         bloom_filter.write(meta.index1, meta.query1 + 1);
         bloom_filter.write(meta.index2, meta.query2 + 1);
       }
    }
  }
```

Implementation of a Counting Bloom Filter in $P4_{16}$

with 2 hash functions

Delete an element

```
control MyIngress(…) {
    register register<bit<1>>(NB_CELLS) bloom_filter;
    apply {
       hash(meta.index1, HashAlgorithm.my_hash1, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);
       hash(meta.index2, HashAlgorithm.my_hash2, 0,
         {meta.dstPrefix, packet.ip.srcIP}, NB_CELLS);

       // Delete a element only if it is in the set
       bloom_filter.read(meta.query1, meta.index1);
       bloom_filter.read(meta.query2, meta.index2);

       if (meta.query1 > 0 && meta.query2 > 0) {
         bloom_filter.write(meta.index1, meta.query1 - 1);
         bloom_filter.write(meta.index2, meta.query2 - 1);
       }
    }
  }
```

So far we have seen how to do insertions, deletions and membership queries

	strategy #1	strategy #2
output	Deterministic	Probabilistic
number of required operations	Probabilistic	Deterministic

Bloom Filters

Counting Bloom Filters

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Each cell contains three fields

count which counts the number of entries mapped to this cell

keySum which is the sum of all the keys mapped to this cell

valueSum which is the sum of all the values mapped to this cell

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Add a new key-value pair
(assuming it is not in the set yet)

For each hash function
hash the key to find the index

Then at this index
increment the count by one
add key to keySum
add value to valueSum

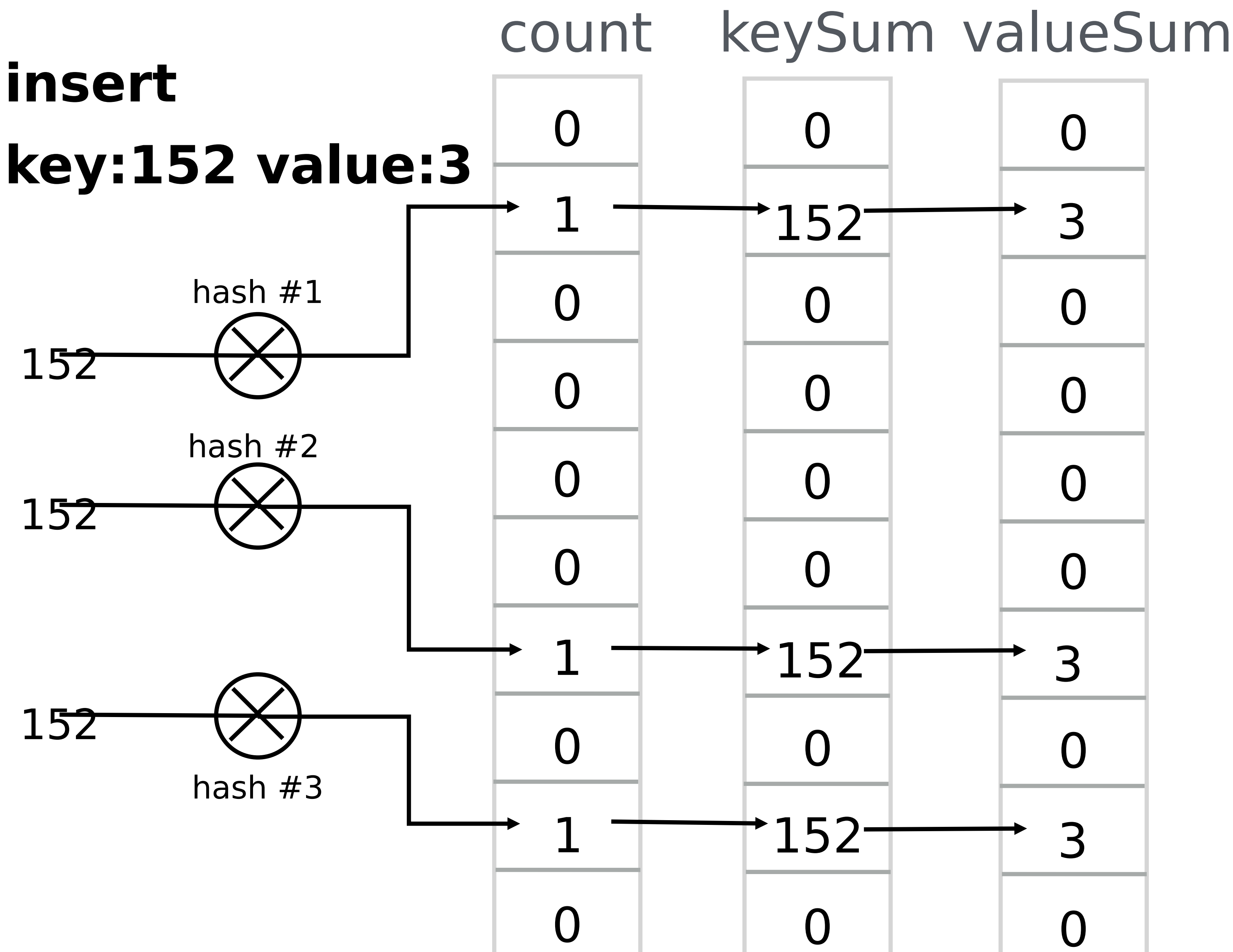
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Delete a key-value pair
(assuming it is in the set)

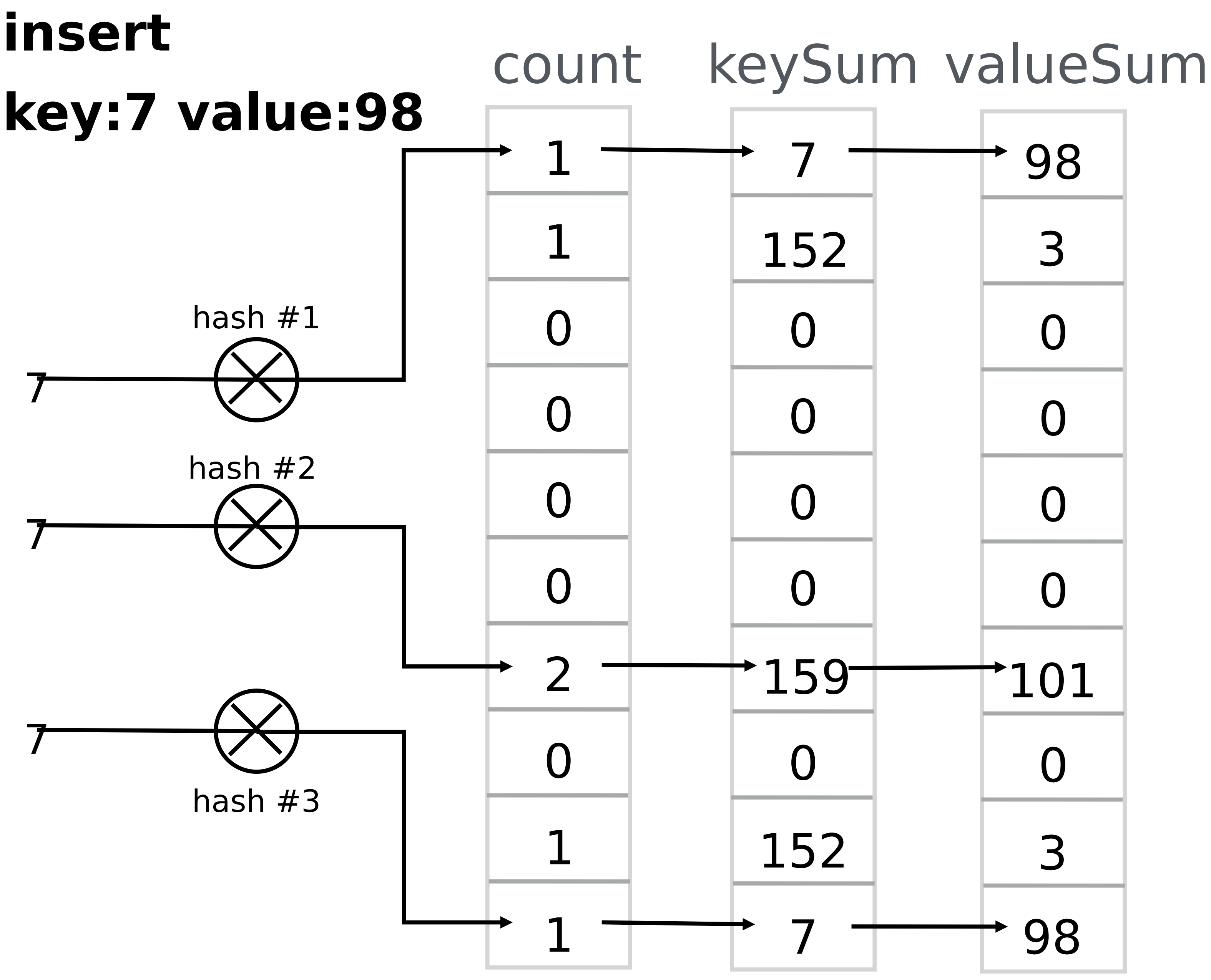
For each hash function
hash the key to find the index

Then at this index
subtract one to the count
subtract key to keySum
subtract value to valueSum

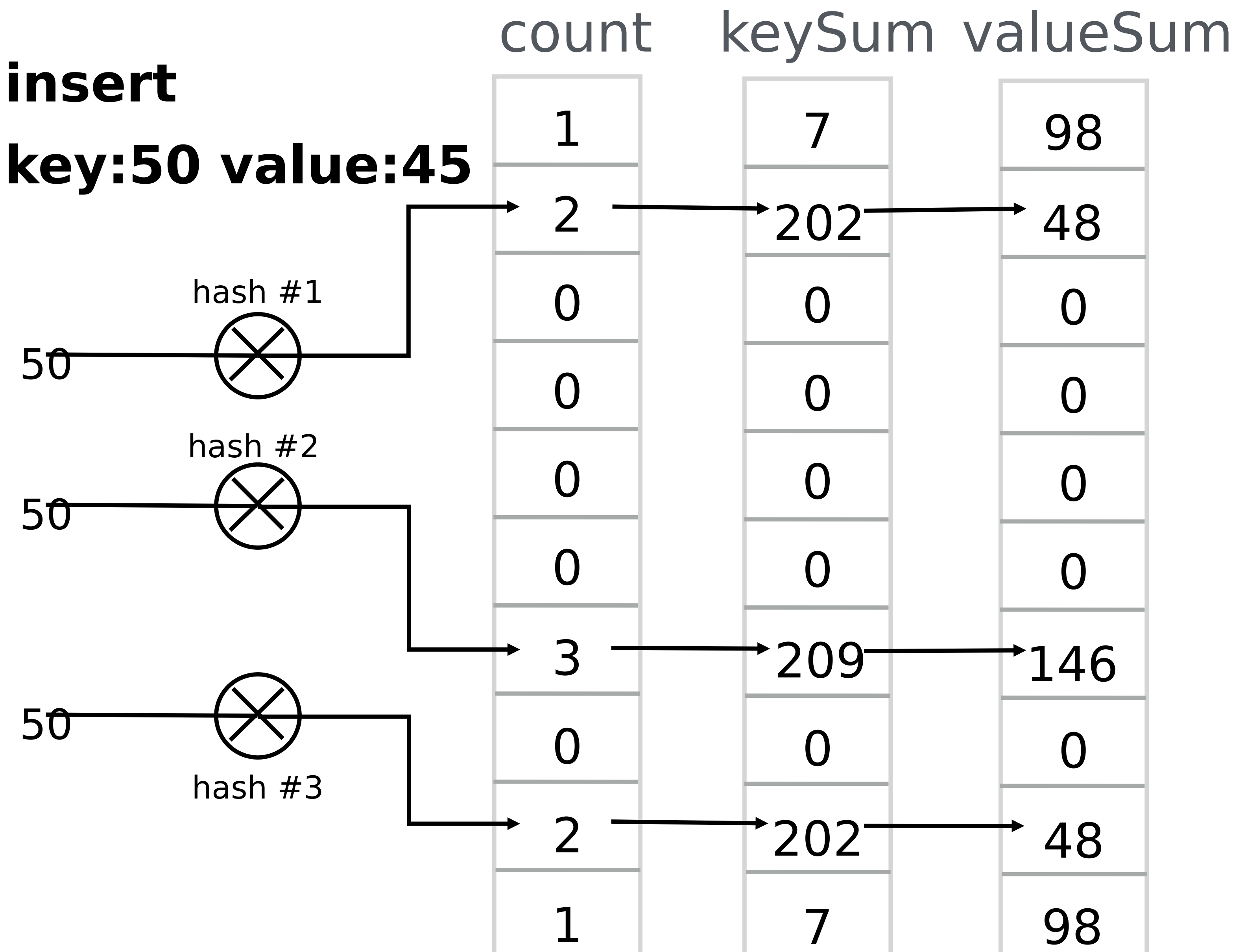
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

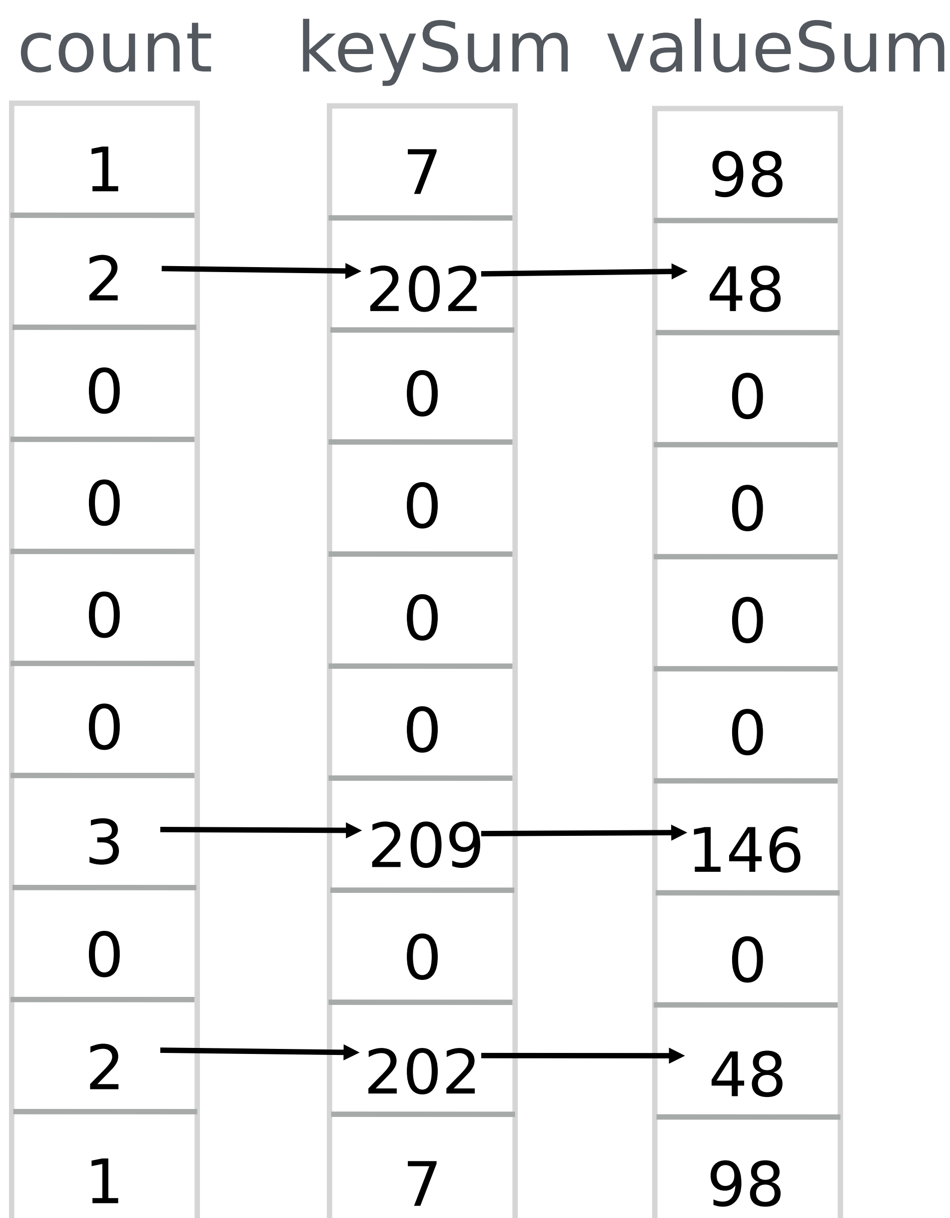


Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

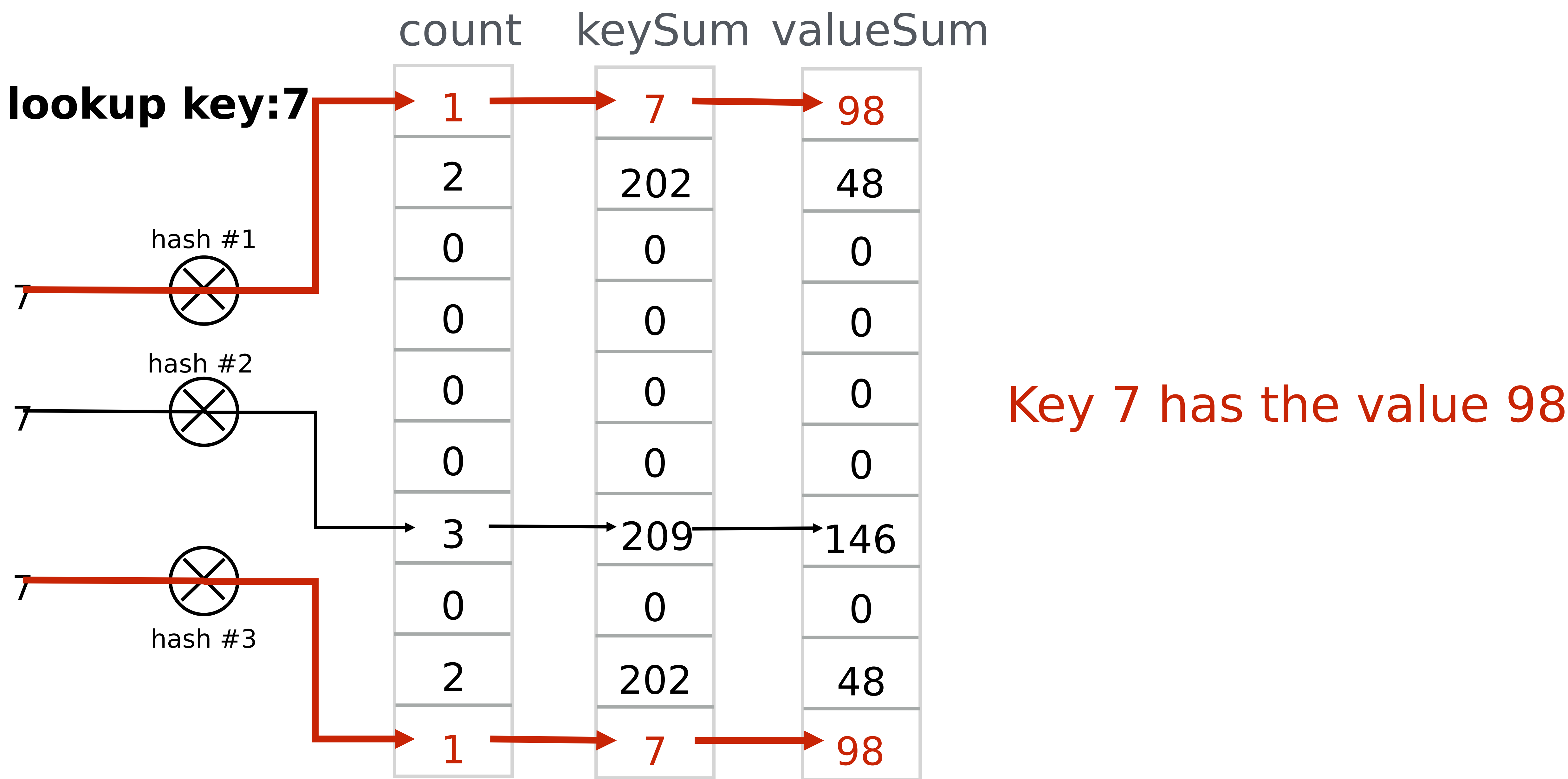
Key-value pair **lookup**

The value of a key can be found if the key is associated to **at least** one cell with a count = 1

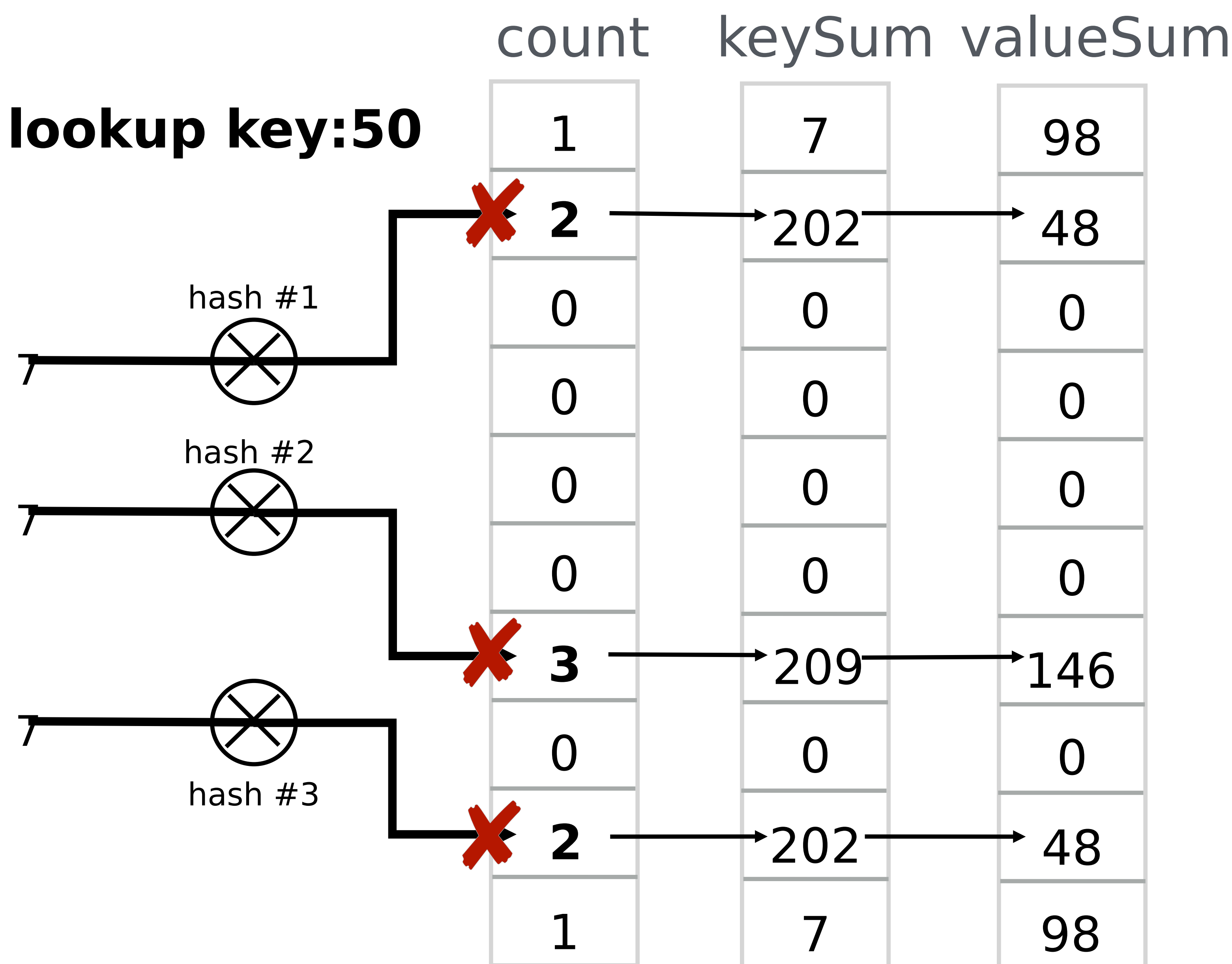
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



The value for the key 50 can’t be found

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Listing the IBLT

While there is an index for which count = 1
Find the corresponding key-value pair and return it
Delete the corresponding key-value pair

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

Listing the IBLT

While there is an index for which count = 1
Find the corresponding key-value pair and return it
Delete the corresponding key-value pair

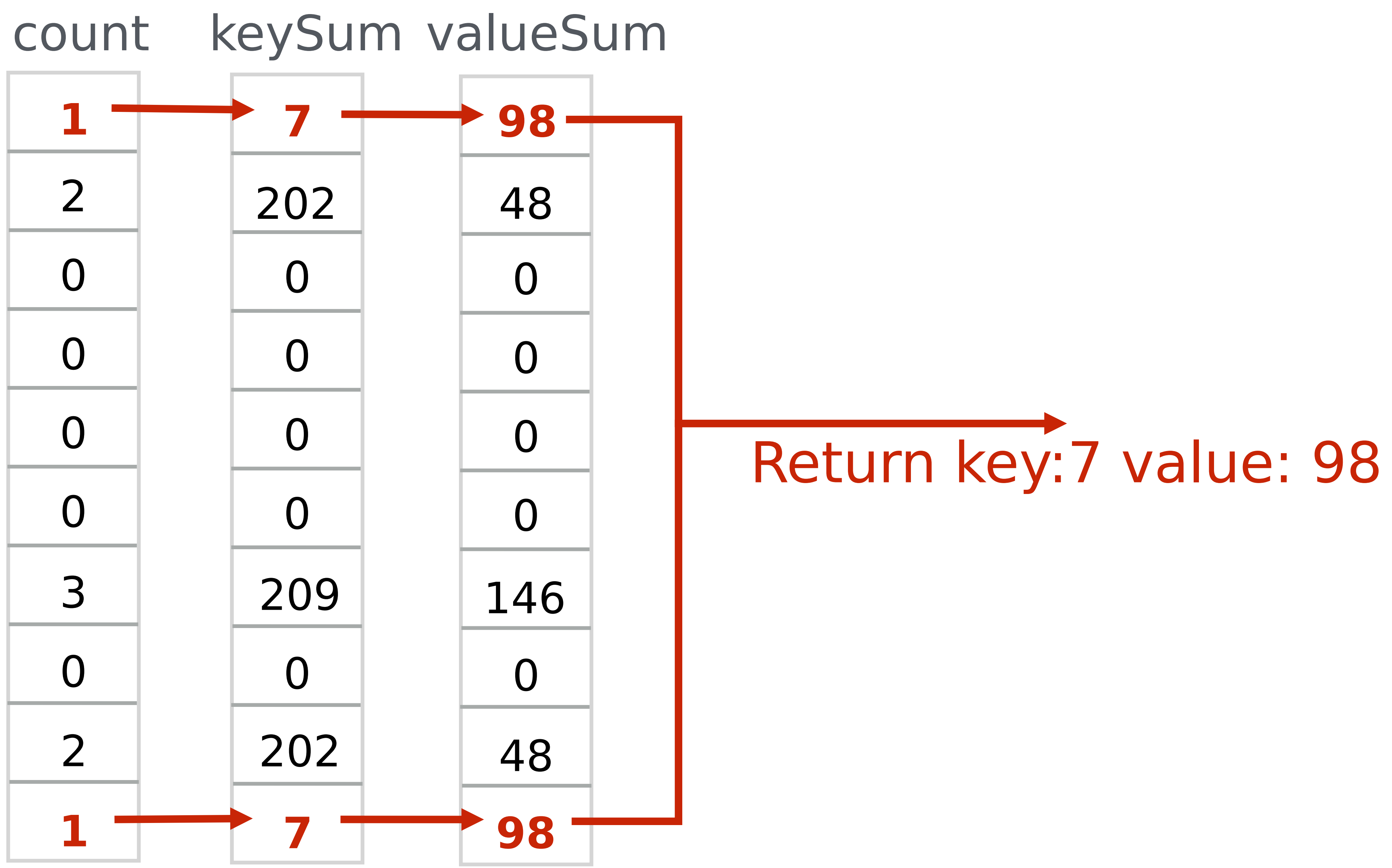
Unless the number of iterations is very low, loops can't be implemented in hardware

The listing is done by the controller

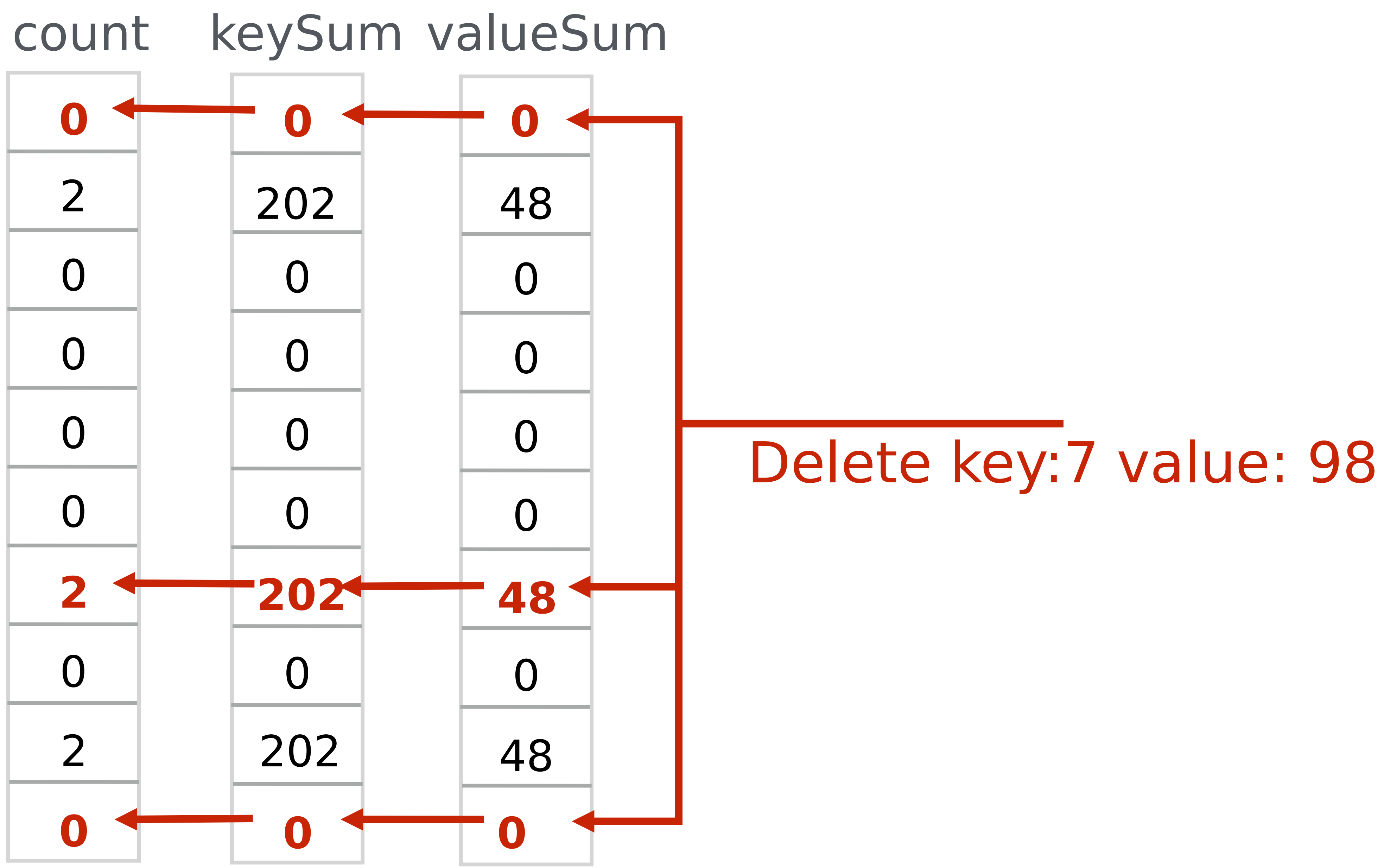
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

count	keySum	valueSum
1	7	98
2	202	48
0	0	0
0	0	0
0	0	0
0	0	0
3	209	146
0	0	0
2	202	48
1	7	98

Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing



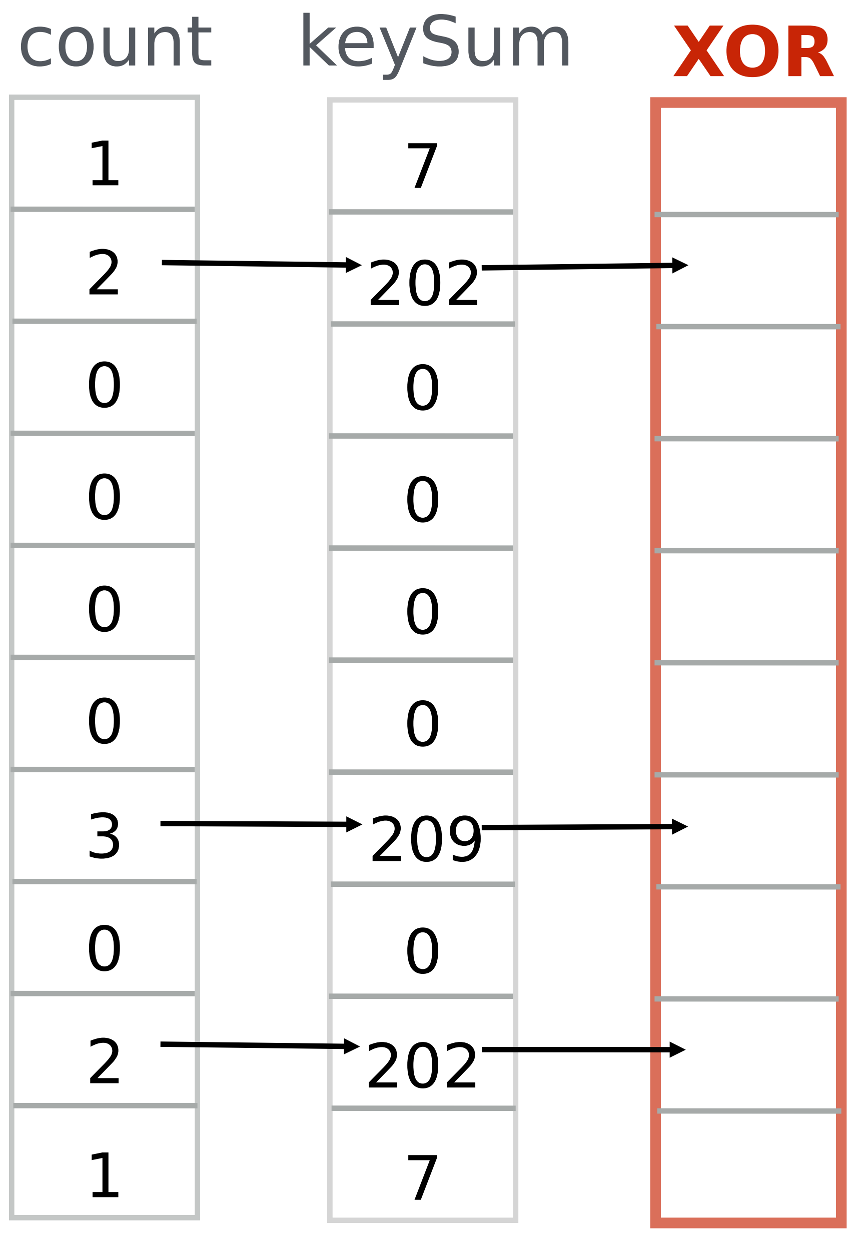
Invertible Bloom Lookup Tables (IBLT) stores key-value pairs and allows for lookups and a complete listing

count	keySum	valueSum
0	0	0
2	202	48
0	0	0
0	0	0
0	0	0
0	0	0
2	202	48
0	0	0
2	202	48
0	0	0

In this example, a complete listing is not possible

In many settings, we can use XORs in place of sums

For example to avoid overflow issues



Bloom filters are **probabilistic data structures** for **set membership queries**. For more info, see:

Space/Time Trade-offs in Hash Coding with Allowable Errors. Burton H. Bloom. 1970.

Network Applications of Bloom Filters: A Survey. Andrei Broder and Michael Mitzenmacher. 2004.

Invertible Bloom Lookup Tables. Michael T. Goodrich and Michael Mitzenmacher. 2015.

FlowRadar: A Better NetFlow for Data Centers Yuliang Li et al. NSDI 2016.

You are looking at a stream of data (packets).

Today, I'll show you how set membership and frequency queries can be realized in P4.

PART 1

Is a certain element (e.g. ip address) in the stream?

→ Bloom filter

PART 2

How frequently does an element appear?

→ CountMin Sketch, Count Sketch, ...

part 2: **counting with sketches**

How frequently does an element appear?

(slides by yours truly)

We are going to look at **frequencies**,

i.e. **how often** an element occurs in a data stream.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

vector of frequencies (counts)

of all ***distinct elements*** $\boldsymbol{x}_i$

We are going to look at **frequencies**,

i.e. **how often** an element occurs in a data stream.

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

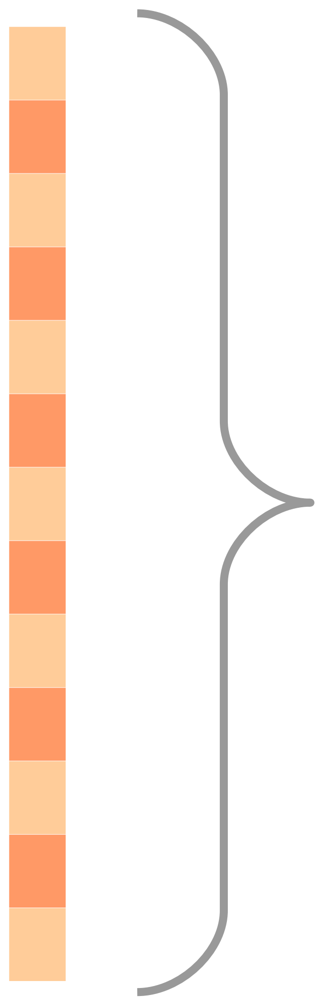
vector of frequencies (counts)

of all ***distinct elements*** *x_i*

e.g. flows, ip addresses, ...

In the worst case, an algorithm providing **exact frequencies** requires **linear space**.

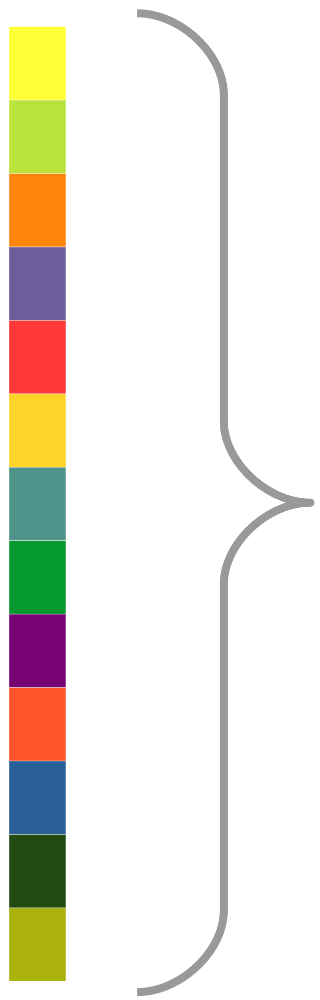
In the worst case, an algorithm providing **exact frequencies** requires **linear space**.



Data Stream

n elements in total

In the worst case, an algorithm providing **exact frequencies** requires **linear space**.



Data Stream

n elements in total

→ **n distinct elements**

(in the worst case)

In the worst case, an algorithm providing **exact frequencies** requires **linear space**.

Data Stream

n elements in total

→ **n distinct elements**

(in the worst case)

→ **n counters** required? :(

Probabilistic datastructures can help again!

Bloom Filters

quickly “filter” only those elements that might be in the set

More efficient by allowing false positives.

Probabilistic datastructures can help again!

Bloom Filters

quickly “filter” only those elements that might be in the set

More efficient by allowing false positives.

Sketches

provide a approximate frequencies of elements in a data stream.

More efficient by allowing mis-counting.

A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

Notation reminder:
vector of frequencies (counts)
of all **distinct elements** $\mathbf{x_i}$

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

$$Pr\left[\ \underset{\text{estimated frequency}}{\hat{x}_i} - \underset{\text{true frequency}}{x_i} \geq \varepsilon \underset{\text{sum of frequencies}}{\|\mathbf{x}\|_1}\ \right] \leq \delta$$

The estimation error ***exceeds*** $\varepsilon\|\mathbf{x}\|_1$

with a ***probability smaller than*** δ

$$Pr\left[\ \underset{\text{estimated frequency}}{\hat{x}_i} - \underset{\text{true frequency}}{x_i} \ \ge\ \varepsilon \overset{\text{relative to L1 norm}}{\underset{\text{sum of frequencies}}{\|\mathbf{x}\|_1}}\ \right] \le \delta$$

The estimation error ***exceeds*** $\varepsilon \|\mathbf{x}\|_1$

with a ***probability smaller than*** δ

$$Pr\left[\ \hat{x}_i \ - \ x_i \ \geq \ \varepsilon \left\|\mathbf{x}\right\|_1 \right] \leq \delta$$

estimated frequency *true frequency* sum *of frequencies*

$$Let \quad \left\|\mathbf{x}\right\|_1 = 10000\,, \quad \varepsilon = 0.01\,, \quad \delta = 0.05$$

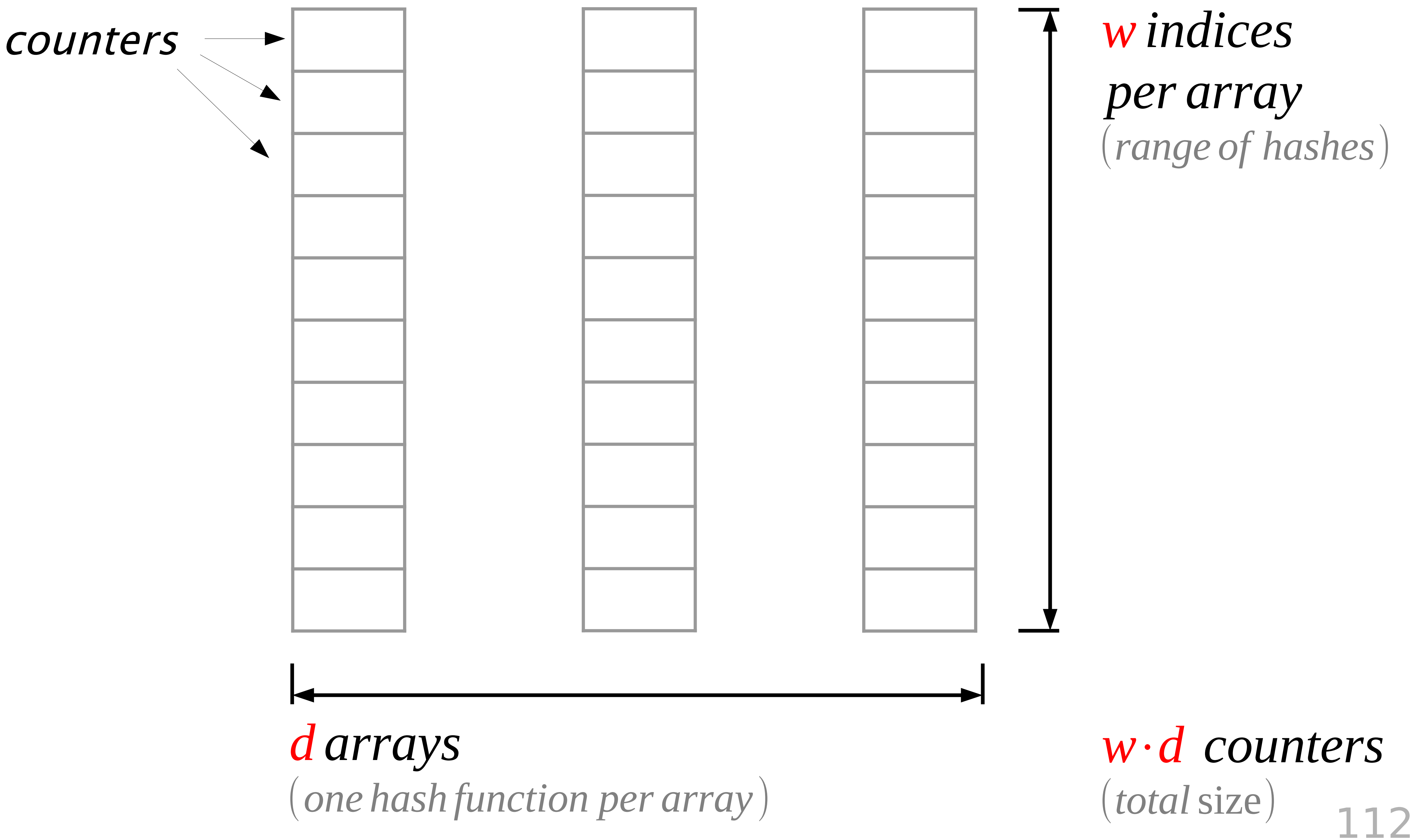
*Then, after counting **10000 elements in total**,*

*the probability for **any estimate** to be*

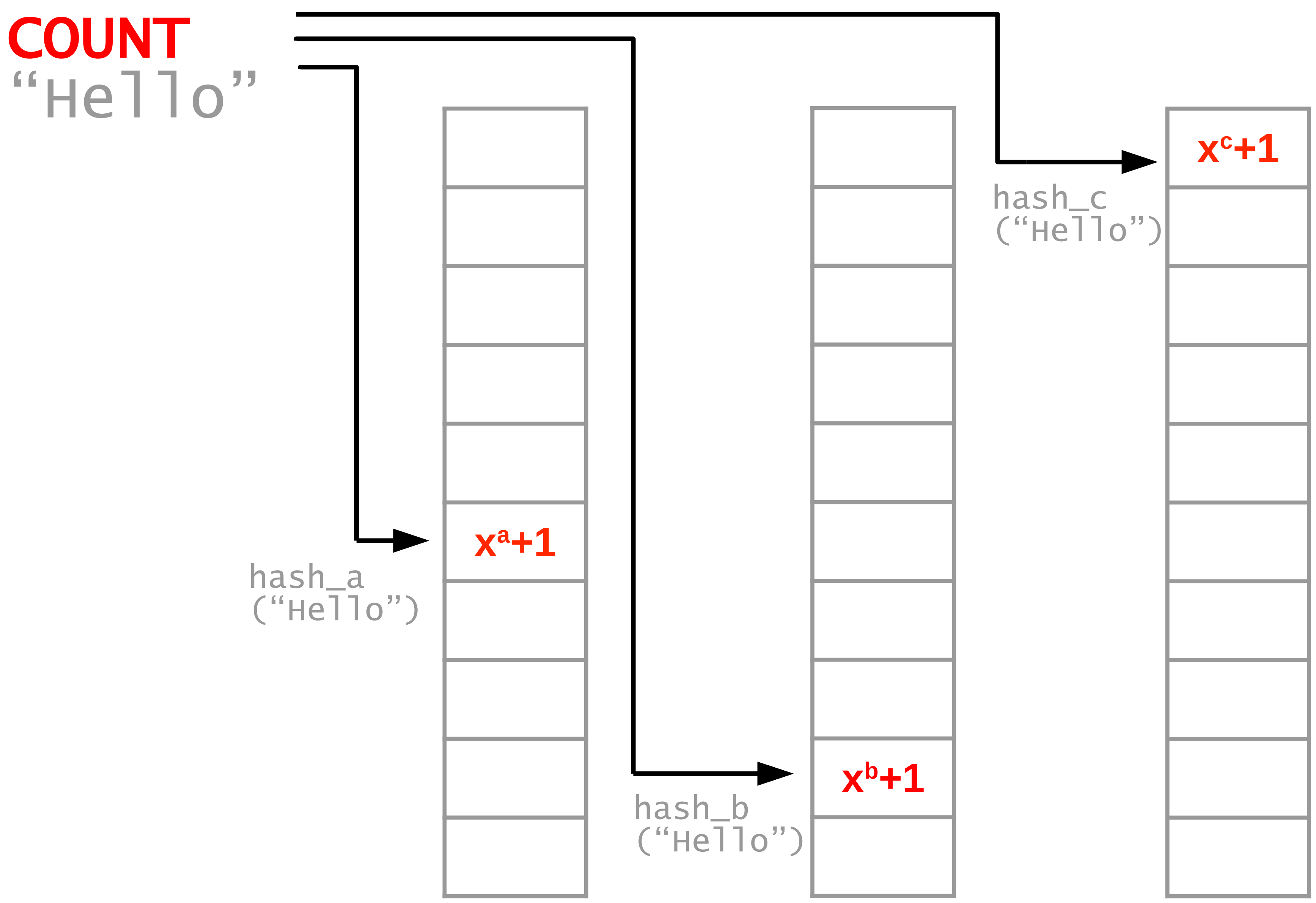
*off by **more than 100** is **less than 5%.***

A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

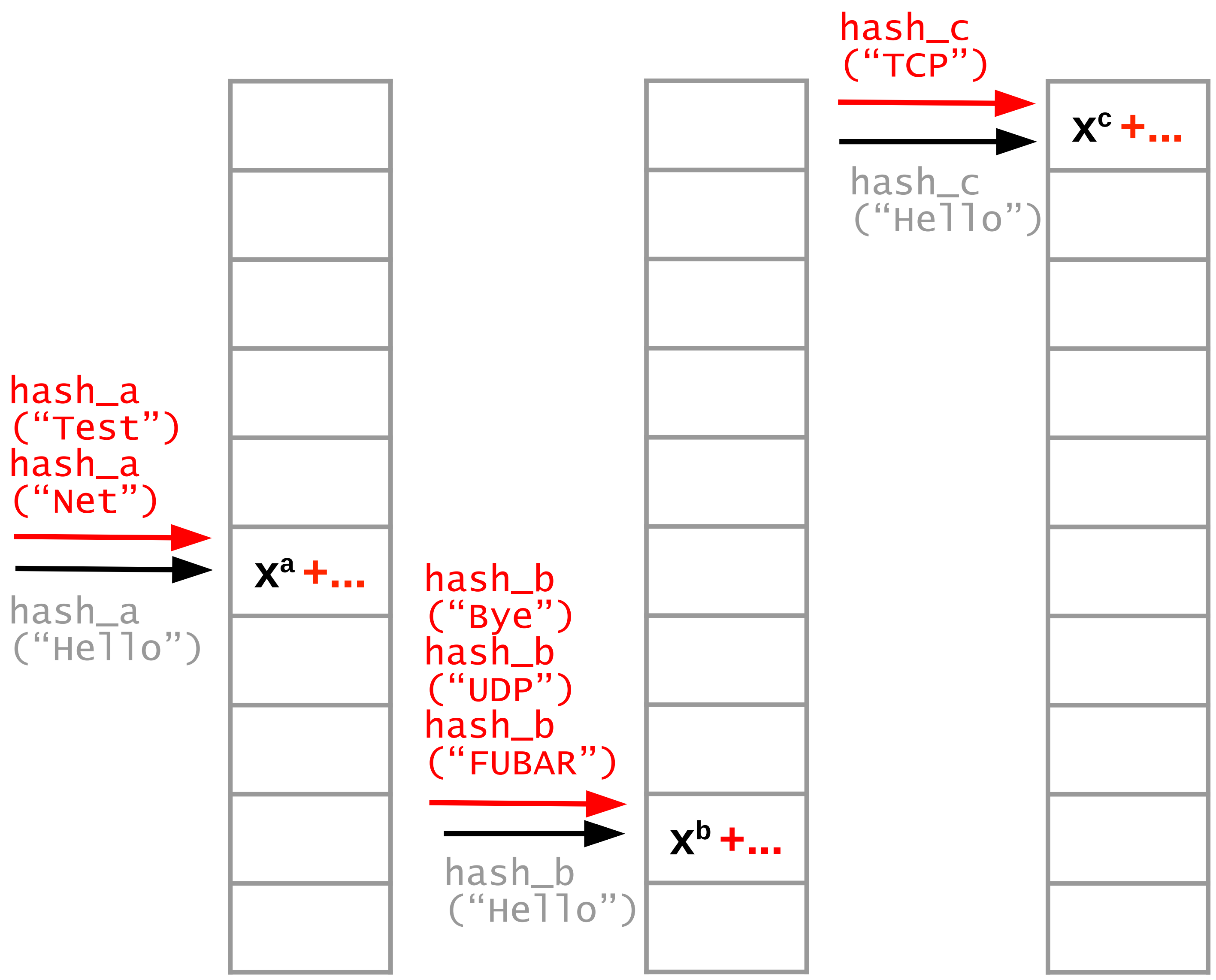
A **CountMin** Sketch uses multiple arrays and hashes.



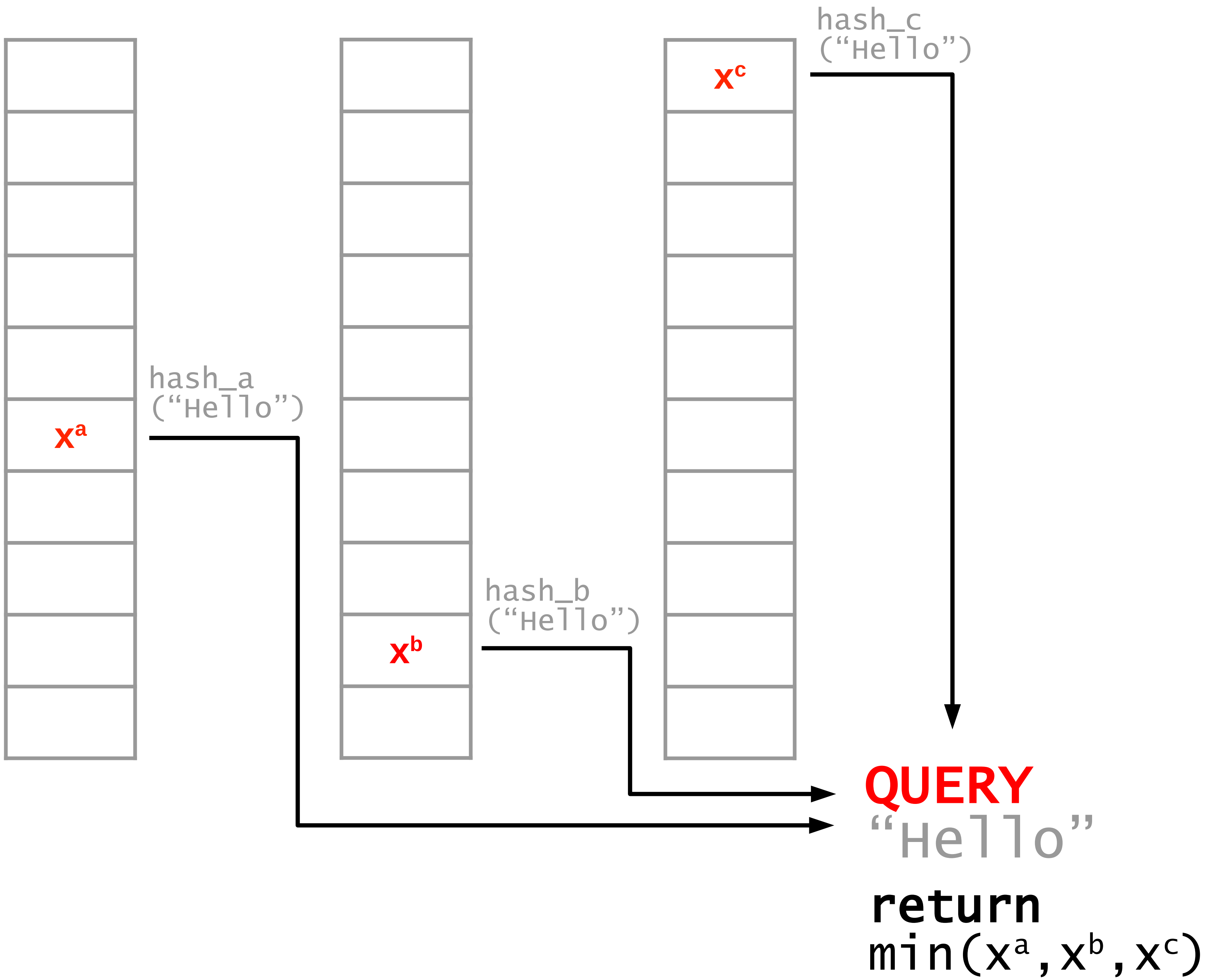
To count, increment **all** hash-indexed fields by 1.



Hash collisions cause **over-counting.**



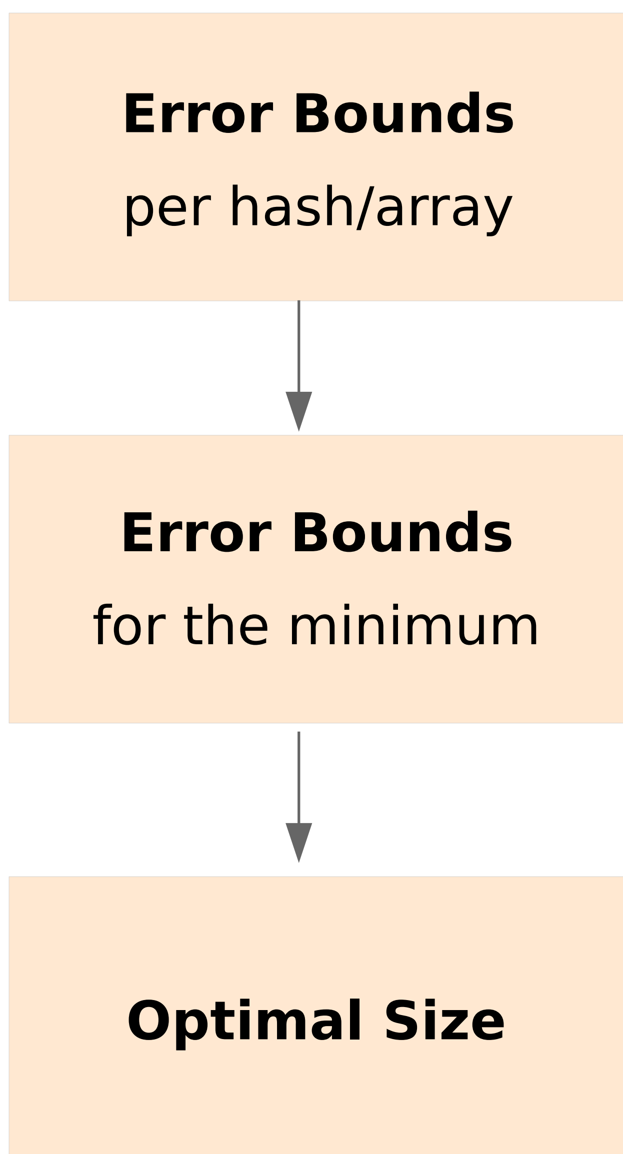
Returning the **minimum value** minimizes the error.



A **CountMin sketch** uses the same principles as a counting bloom filter, but is designed to have **provable L1 error bounds** for frequency queries.

$$Pr\left[\ \underbrace{\hat{x}_i}_{\textit{estimated frequency}} - \underbrace{x_i}_{\textit{true frequency}} \geq \varepsilon \underbrace{\|\boldsymbol{x}\|_1}_{\text{sum of } \textit{frequencies}}\ \right] \leq \delta$$

Understanding the error bounds allows **dimensioning** the sketch optimally.



Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\hat{x}_i = \min_{h \in h_1 .. h_d} \hat{x}_i^h$$

estimated frequency

estimate for specific hash

The error bounds can be derived with **Markov's Inequality**

Error Bounds
per hash/array

↓

Error Bounds
for the minimum

↓

Optimal Size

$$\Pr\left[\,X \geq c \cdot E\left[\,X\,\right]\right] \leq \frac{1}{c}$$

wikipedia.org/wiki/Markov's_inequality

The error bounds can be derived with **Markov’s Inequality**

Error Bounds
per hash/array

Error Bounds
for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

wikipedia.org/wiki/Markov's_inequality

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h = x_i + \sum_{x_j \neq x_i} x_j 1_h(x_i, x_j)$$

true frequency

over-counting from hash collisions

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h = \quad x_i \quad + \quad \underbrace{\sum_{x_j \neq x_i} x_j 1_h(x_i, x_j)}_{\text{hash collision}}$$

hash collision

$$= \begin{cases} 1, & if\ h(x_i) = h(x_j) \\ 0, & otherwise \end{cases}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i = \sum_{x_j \neq x_i} x_j 1_h(x_i, x_j)$$

estimation error

over-counting from hash collisions

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i \;=\; \sum_{x_j \neq x_i} x_j \, 1_h\left(x_i, x_j\right)$$

$$E\left[\hat{x}_i^h - x_i\right] \;=\; E\left[\sum_{x_j \neq x_i} x_j \, 1_h\left(x_i, x_j\right)\right]$$

We treat the **data as a constant** and the **hash as a random function** with certain properties.

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i = \sum_{x_j \neq x_i} x_j \, 1_h(x_i, x_j)$$

$$E\left[\hat{x}_i^h - x_i\right] = E\left[\sum_{x_j \neq x_i} x_j \, 1_h(x_i, x_j)\right]$$

random

constant

wikipedia.org/wiki/Universal_hashing

We treat the **data as a constant** and the **hash as a random function** with certain properties.

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i = \sum_{x_j \neq x_i} x_j \, 1_h\left(x_i, x_j\right)$$

$$E\left[\hat{x}_i^h - x_i\right] = \sum_{x_j \neq x_i} x_j \, E\left[1_h\left(x_i, x_j\right)\right]$$

wikipedia.org/wiki/Universal_hashing

We treat the **data as a constant** and the **hash as a random function** with certain properties.

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i = \sum_{x_j \neq x_i} x_j \, 1_h(x_i, x_j)$$

$$E\left[\hat{x}_i^h - x_i\right] = \sum_{x_j \neq x_i} x_j \underbrace{E\left[1_h(x_i, x_j)\right]}_{\leq \frac{1}{w}}$$

wikipedia.org/wiki/Universal_hashing

We treat the **data as a constant** and the **hash as a random function** with certain properties.

Error Bounds
per hash/array

↓

Error Bounds
for the minimum

↓

Optimal Size

$$\Pr\left[\hat{x}_i^{h} - x_i \geq c \cdot E\left[\hat{x}_i^{h} - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^{h} - x_i = \sum_{x_j \neq x_i} x_j \, 1_h\left(x_i, x_j\right)$$

$$\mathrm{E}\left[\hat{x}_i^{h} - x_i\right] \leq \sum_{x_j \neq x_i} x_j \frac{1}{w}$$

wikipedia.org/wiki/Universal_hashing

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i = \sum_{x_j \neq x_i} x_j 1_h(x_i, x_j)$$

$$\mathrm{E}\left[\hat{x}_i^h - x_i\right] \leq \sum_{x_j \neq x_i} x_j \frac{1}{w} \leq \sum_{x_j} x_j \frac{1}{w}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot E\left[\hat{x}_i^h - x_i\right]\right] \leq \frac{1}{c}$$

$$\hat{x}_i^h - x_i = \sum_{x_j \neq x_i} x_j \, 1_h(x_i, x_j)$$

$$\mathrm{E}\left[\hat{x}_i^h - x_i\right] \leq \sum_{x_j \neq x_i} x_j \frac{1}{w} \leq \|\boldsymbol{x}\|_1 \frac{1}{w}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq c \cdot \underbrace{E\left[\hat{x}_i^h - x_i\right]}_{\leq \frac{1}{w}\|\boldsymbol{x}\|_1}\right] \leq \frac{1}{c}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq \frac{c}{w}\|\boldsymbol{x}\|_1\right] \leq \frac{1}{c}$$

Error Bounds

per hash/array

↓

Error Bounds

for the minimum

↓

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq \underbrace{\varepsilon^h}_{\frac{c}{w}} \|\boldsymbol{x}\|_1\right] \leq \underbrace{\delta^h}_{\frac{1}{c}}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^{h} - x_i \geq \underbrace{\varepsilon^{h}}_{\frac{c}{w}} \|\boldsymbol{x}\|_1\right] \leq \underbrace{\delta^{h}}_{\frac{1}{c}}$$

*The **estimate for each hash** has a well defined **L1 error bound**.*

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$\Pr\left[\hat{x}_i^h - x_i \geq \underbrace{\varepsilon^h}_{\frac{c}{w}} \|\mathbf{x}\|_1\right] \leq \underbrace{\delta^h}_{\frac{1}{c}}$$

The ***estimate for each hash*** *has a well defined* ***L1 error bound****.*

What about the minimum?

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$Pr\left[\hat{x}_i - x_i \geq \frac{c}{w}\|\boldsymbol{x}\|_1\right] \leq \textbf{?}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$Pr\left[\ \underbrace{\min_{h \in h_1 .. h_d} \hat{x}_i^h}_{\hat{x}_i} - x_i \geq \frac{c}{w} \|\boldsymbol{x}\|_1 \right] \leq \ ?$$

Multiple hash functions work like **independent trials**.

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$Pr\,[\underbrace{\min_{h \in h_1 .. h_d} \hat{x}_i^h}_{\hat{x}_i} - x_i \geq \frac{c}{w} \|\boldsymbol{x}\|_1] \;\leq\; \boldsymbol{?}$$

$$\Leftrightarrow$$

$$\prod_{h \in h_1 .. h_d} Pr\,[\hat{x}_i^h - x_i \geq \frac{c}{w} \|\boldsymbol{x}\|_1] \;\leq\; \boldsymbol{?}$$

Error Bounds

per hash/array

↓

Error Bounds

for the minimum

↓

Optimal Size

$$Pr\,[\underbrace{\min_{h \in h_1 .. h_d} \hat{x}_i^{\,h}}_{\hat{x}_i} - x_i \geq \frac{c}{w} \|\boldsymbol{x}\|_1] \quad \leq \quad \boldsymbol{?}$$

$$\Leftrightarrow$$

$$\prod_{h \in h_1 .. h_d} \underbrace{Pr\,[\hat{x}_i^{\,h} - x_i \geq \frac{c}{w} \|\boldsymbol{x}\|_1]}_{\leq \frac{1}{c}} \quad \leq \quad \boldsymbol{?}$$

error bound per hash

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$Pr\,[\underbrace{\min_{h \in h_1 .. h_d} \hat{x}_i^{\,h}}_{\hat{x}_i} - x_i \geq \frac{c}{w}\,\|\boldsymbol{x}\|_1] \quad \leq \quad ?$$

$$\Leftrightarrow$$

$$\prod_{h \in h_1 .. h_d} \underbrace{Pr\,[\hat{x}_i^{\,h} - x_i \geq \frac{c}{w}\,\|\boldsymbol{x}\|_1]}_{\leq \frac{1}{c}} \quad \leq \quad \frac{1}{c^d}$$

Error Bounds

per hash/array

↓

Error Bounds

for the minimum

↓

Optimal Size

$$Pr\left[\underbrace{\min_{h \in h_1 .. h_d} \hat{x}_i^h}_{\hat{x}_i} - x_i \geq \frac{c}{w} \|\mathbf{x}\|_1\right] \leq \frac{\mathbf{1}}{c^d}$$

Error Bounds

per hash/array

Error Bounds

for the minimum

Optimal Size

$$Pr\left[\hat{x}_i - x_i \geq \frac{c}{w}\|\boldsymbol{x}\|_1\right] \leq \frac{1}{c^d}$$

Error Bounds
per hash/array

Error Bounds
for the minimum

Optimal Size

$$Pr\left[\hat{x}_i - x_i \geq \underbrace{\varepsilon}_{\frac{c}{w}} \left\|\boldsymbol{x}\right\|_1\right] \leq \underbrace{\delta}_{\frac{1}{c^d}}$$

We have proven the error bounds!

But what about the constant c?

For **every c**, there is a pair (d, w) achieving the error bound and confidence (ε, δ).

Error Bounds
per hash/array

↓

Error Bounds
for the minimum

↓

Optimal Size

$$\varepsilon = \frac{c}{w} \quad \Rightarrow \quad w = \left\lceil \frac{c}{\varepsilon} \right\rceil \qquad \textit{(hash range)}$$

$$\delta = \frac{1}{c^d} \quad \Rightarrow \quad d = \left\lceil \log_c \frac{1}{\delta} \right\rceil \qquad \textit{(\#hashes)}$$

Choosing c=e **minimizes** the total **number of counters**.

Error Bounds

per hash/array

↓

Error Bounds

for the minimum

↓

Optimal Size

$$\varepsilon = \frac{e}{w} \quad \Rightarrow \quad w = \left\lceil \frac{e}{\varepsilon} \right\rceil \qquad \text{(hash range)}$$

$$\delta = \frac{1}{e^d} \quad \Rightarrow \quad d = \left\lceil \ln \frac{1}{\delta} \right\rceil \qquad \text{(\#hashes)}$$

$$d \cdot w = \frac{c}{\varepsilon} \log_c \frac{1}{\delta} \overset{minimize}{=} \frac{e}{\varepsilon} \ln \frac{1}{\delta}$$

A **CountMin** sketch recipe

Error Bounds
per hash/array

↓

Error Bounds
for the minimum

↓

Optimal Size

Given ε, δ ***, choosing***

$$w = \left\lceil \frac{e}{\varepsilon} \right\rceil$$ *(hash range)*

$$d = \left\lceil \ln \frac{1}{\delta} \right\rceil$$ *(#hashes)*

*requires the **minimum number of counters** s.t. the CountMin Sketch can guarantee that*

$$\hat{x}_i - x_i \geq \varepsilon \, \|\boldsymbol{x}\|_1$$

with a probability less than δ

A **CountMin** sketch recipe

(see extended slides for derivation)

Error Bounds

per hash/array

↓

Error Bounds

for the minimum

↓

Optimal Size

Given ε, δ **, choosing**

$$w = \left\lceil \frac{e}{\varepsilon} \right\rceil$$

(hash range)

$$d = \left\lceil \ln \frac{1}{\delta} \right\rceil$$

(#hashes)

requires the **minimum number of counters** s.t. the CountMin Sketch can guarantee that

$$\hat{x}_i - x_i \geq \varepsilon \left\| \boldsymbol{x} \right\|_1$$

$$\textit{with a probability less than } \delta$$

A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

***CountMin** sketch recipe*

Choose $d = \left\lceil \ln \frac{1}{\delta} \right\rceil, w = \left\lceil \frac{e}{\varepsilon} \right\rceil$

Then $\hat{x}_i - x_i \geq \varepsilon \, \|\boldsymbol{x}\|_1$ *with a probability less than* δ

A **CountMin sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L1 error bounds** for frequency queries.

→ only one design out of many!

A **Count sketch** uses the same principles as a counting bloom filter, but is **designed** to have **provable L2 error bounds** for frequency queries.

The Count sketch uses **additional hashing** to give **L2 error bounds**, but requires more **resources**.

CountMin sketch

$h_1, \ldots, h_d$: U → {1, …, w}

COUNT x_i:

for h in $h_1, \ldots, h_d$:

$Reg_h[h(x_i)] + 1$

QUERY x_i:

return $\min_{h \text{ in } h1, \ldots, hd}($

$Reg_h[h(x_i)]$

)

The Count sketch uses **additional hashing** to give **L2 error bounds**, but requires more **resources**.

CountMin sketch	Count sketch
$h_1, \ldots, h_d$: $U \to \{1, \ldots, w\}$	$h_1, \ldots, h_d$: $U \to \{1, \ldots, w\}$
	g: $U \to \{+1, -1\}$
COUNT x_i:	**COUNT x_i:**
for h in $h_1, \ldots, h_d$:	for h in $h_1, \ldots, h_d$:
$Reg_h[h(x_i)]$ + 1	$Reg_h[h(x_i)]$ + $g(x_i)$
QUERY x_i:	**QUERY x_i:**
return $\min_{h \text{ in } h1, \ldots, hd}($	return $\text{median}_{h \text{ in } h1, \ldots, hd}($
$Reg_h[h(x_i)]$	$Reg_h[h(x_i)] * g(x_i)$
)	)

The Count sketch uses **additional hashing** to give **L2 error bounds**, but requires more **resources**.

***CountMin** sketch recipe*

Choose $d = \left\lceil \ln \frac{1}{\delta} \right\rceil,\ w = \left\lceil \frac{e}{\varepsilon} \right\rceil$

Then $\hat{x}_i - x_i \geq \varepsilon \|\boldsymbol{x}\|_1$ *with a probability less than* δ

The Count sketch uses **additional hashing** to give **L2 error bounds**, but requires more **resources**.

***CountMin** sketch recipe*

Choose $d = \left\lceil \ln \frac{1}{\delta} \right\rceil, w = \left\lceil \frac{e}{\varepsilon} \right\rceil$

Then $\hat{x}_i - x_i \geq \varepsilon \|\boldsymbol{x}\|_1$ *with a probability less than* δ

***Count** sketch recipe*

Choose $d = \left\lceil \ln \frac{1}{\delta} \right\rceil, w = \left\lceil \frac{e}{\boldsymbol{\varepsilon}^2} \right\rceil$

Then $\hat{x}_i - x_i \geq \varepsilon \|\boldsymbol{x}\|_2$ *with a probability less than* δ

Sketches **are the new black**

OpenSketch

NSDI ‘13

[source]

Software Defined Traffic Measurement with OpenSketch

Minlan Yu[†] Lavanya Jose[*] Rui Miao[†]
[†] University of Southern California [*] Princeton University

UnivMon

SIGCOMM ‘16

[source]

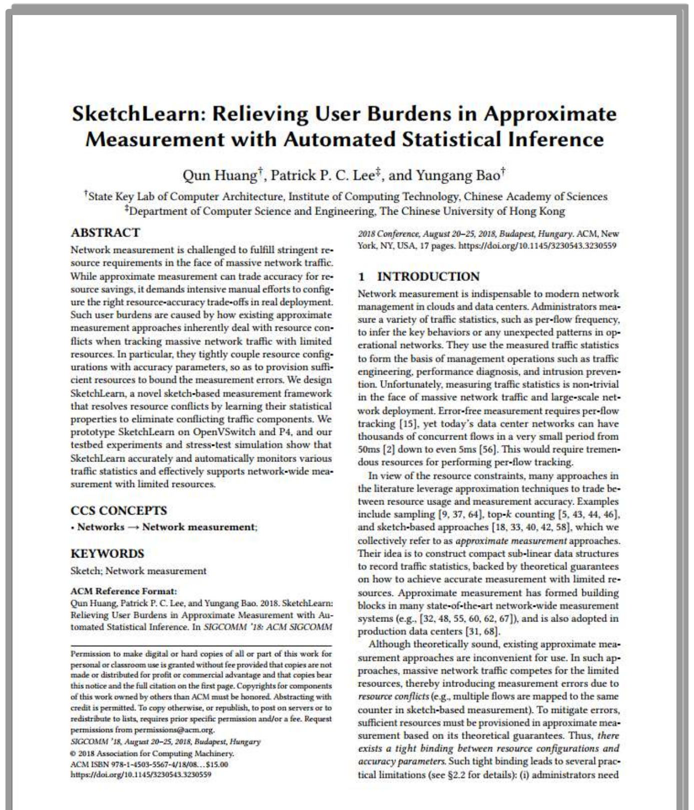
One Sketch to Rule Them All:
Rethinking Network Flow Monitoring with UnivMon

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

SketchLearn

SIGCOMM ‘18

[source]



SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference

Qun Huang[†], Patrick P. C. Lee[‡], and Yungang Bao[†]
[†] State Key Lab of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[‡] Department of Computer Science and Engineering, The Chinese University of Hong Kong

Sketches **are the new black**

LightGuardian

NSDI ‘21

[source]


LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets

Nearly-Zero-Error

NSDI ‘21

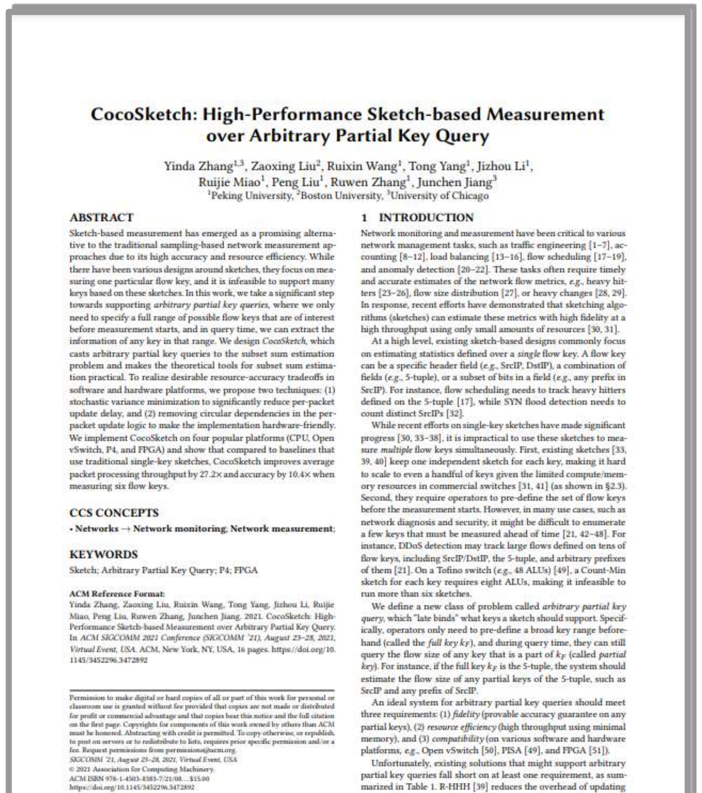
[source]


Toward Nearly-Zero-Error Sketching via Compressive Sensing

CocoSketch

SIGCOMM ‘21

[source]


CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$Pr\left[\hat{x}_i - x_i \geq \varepsilon \|\boldsymbol{x}\|_1\right] \leq \delta$$

estimation error

relative to sum of all elements

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$\textit{Let}\quad \varepsilon = 0.01,\quad \|\boldsymbol{x}\|_1 = 10000 \quad \left(\Rightarrow \varepsilon \cdot \|\boldsymbol{x}\|_1 = 100\right)$$

$$\textit{Assume two flows}\quad x_a,\quad x_b,$$

$$\textit{with}\quad \|x_a\|_1 = 1000,\quad \|x_b\|_1 = 50$$

low frequency

high frequency

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$Let \quad \varepsilon = 0.01, \quad \|\mathbf{x}\|_1 = 10000 \quad \left(\Rightarrow \varepsilon \cdot \|\mathbf{x}\|_1 = 100\right)$$

$$Assume\ two\ flows \quad x_a\,, \quad x_b\,,$$

$$with \quad \|x_a\|_1 = 1000, \quad \|x_b\|_1 = 50$$

Error relative to **stream size**: 1%

Sketches have limitations. They **compute statistical summaries** and favor elements with **high frequency**.

$$Let \quad \varepsilon = 0.01 \,, \quad \|\boldsymbol{x}\|_1 = 10000 \quad (\Rightarrow \varepsilon \cdot \|\boldsymbol{x}\|_1 = 100)$$

$$Assume\ two\ flows \quad x_a \,, \quad x_b \,,$$

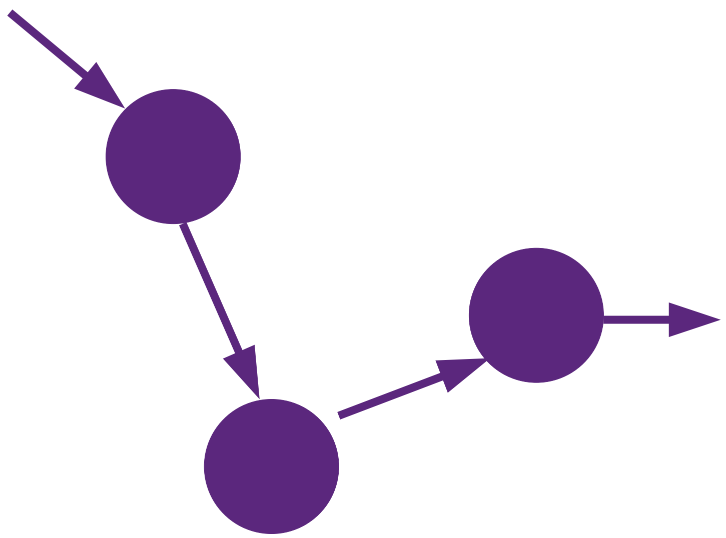
$$with \quad \|x_a\|_1 = 1000 \,, \quad \|x_b\|_1 = 50$$

Error relative to **stream size**: 1%

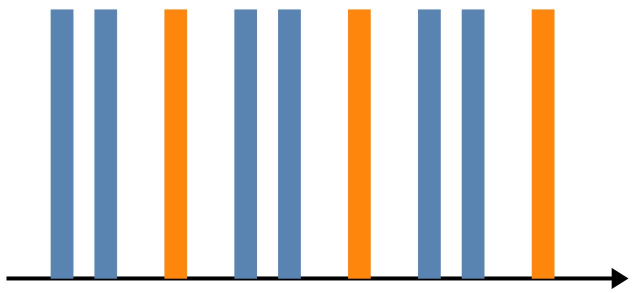
flow size: x_a: 10%, x_b: **200%**

Other problems a sketch **can't handle**:

causality



patterns



rare things

You are looking at a stream of data (packets). Today, I'll show you how set membership and frequency queries can be realized in P4.

PART 1

Is a certain element (e.g. ip address) in the stream?

→ Bloom filter

PART 2

How frequently does an element appear?

→ CountMin Sketch, Count Sketch, ...

TAKEAWAY

Probabilistic data structures provide **trade-offs between resources and error**, and **provable guarantees** to rely on.