# Data-Intensive Distributed Computing

CS 431/631 451/651 (Fall 2019)

## Part 9: Real-Time Data Analytics (2/2)

November 28, 2019

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451

APACHE kafka®

Slides from Michael G. Noll, Verisign

# Kafka?

Apache Kafka

A high-throughput distributed messaging system.

- http://kafka.apache.org/
- Originated at LinkedIn, open sourced in early 2011
- Implemented in Scala, some Java

# Kafka adoption and use cases

- **LinkedIn:** activity streams, operational metrics, data bus
  - 400 nodes, 18k topics, 220B msg/day (peak 3.2M msg/s), May 2014
- **Netflix**: real-time monitoring and event processing
- **Twitter**: as part of their Storm real-time data pipelines
- **Spotify**: log delivery (from 4h down to 10s), Hadoop
- **Loggly**: log collection and processing
- **Mozilla**: telemetry data
- Airbnb, Cisco, Uber, …

https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

# How fast is Kafka?

- **“Up to 2 million writes/sec on 3 cheap machines”**
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated

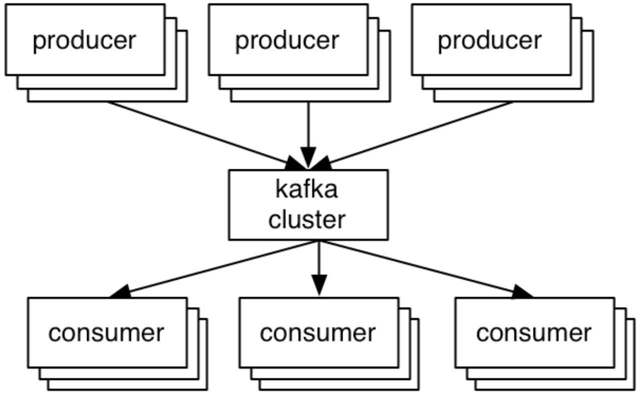
# Why is Kafka so fast?

- Fast **writes**:
  - While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM.
- Fast **reads**:
  - Very efficient to transfer data from page cache to a network **socket**
  - Linux: **`sendfile()`** system call
- Combination of the two = fast Kafka!
  - Example (Operations): On a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks as they will be serving data entirely from cache.

http://kafka.apache.org/documentation.html#persistence

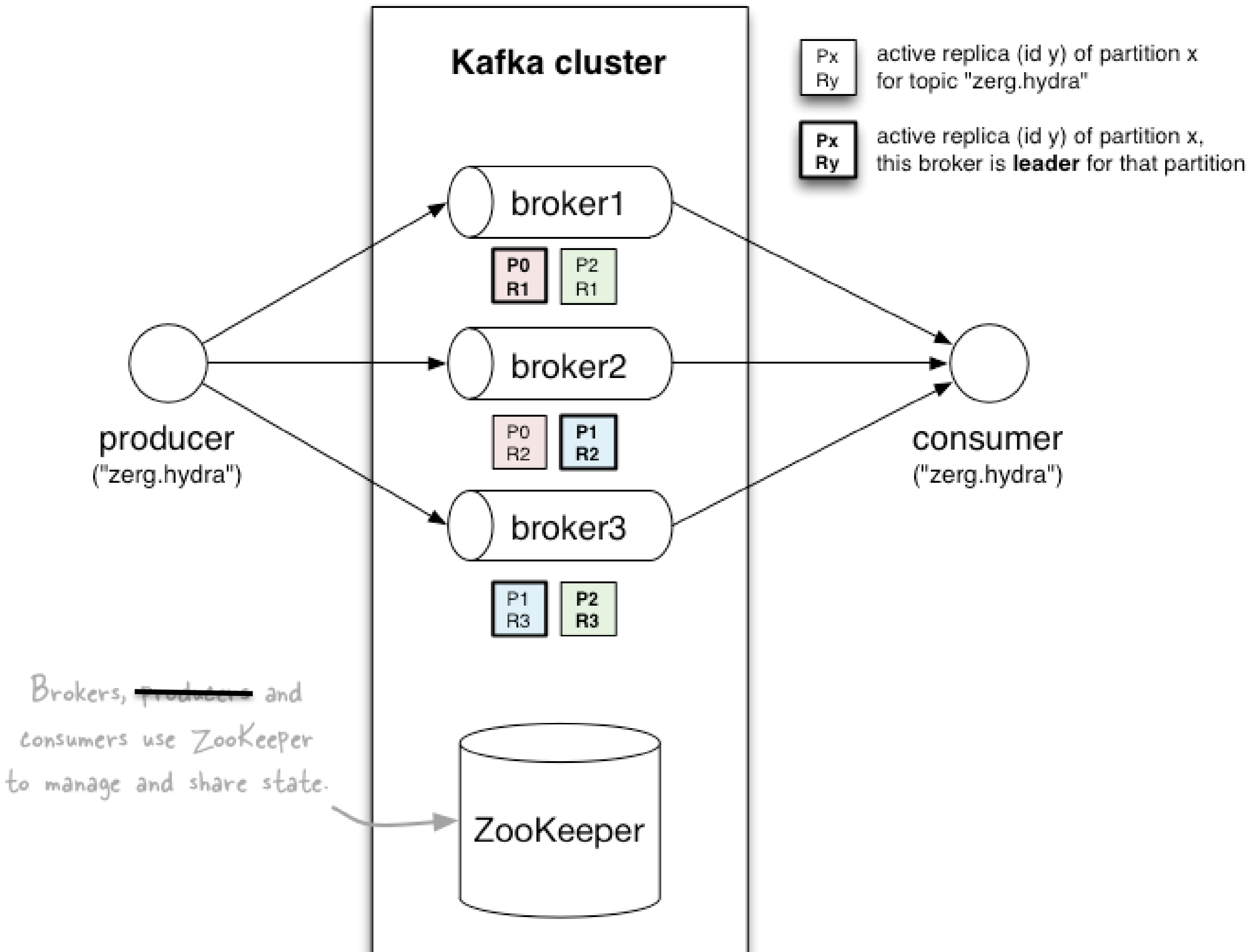
# A first look

- The who is who
  - **Producers** write data to **brokers**.
  - **Consumers** read data from **brokers**.
  - All this is distributed.




- The data
  - Data is stored in **topics**.
  - **Topics** are split into **partitions**, which are **replicated**.

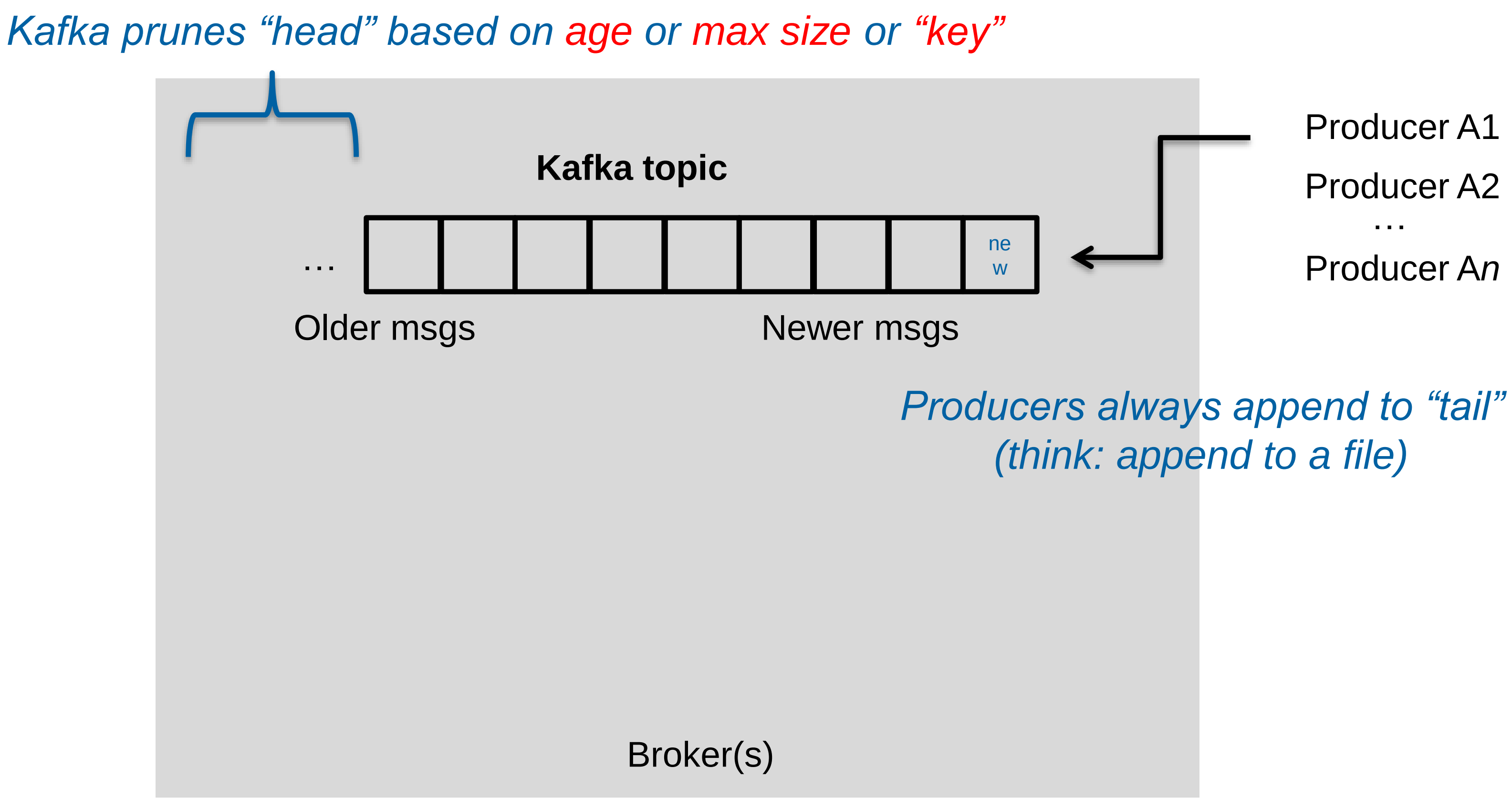
# A first look



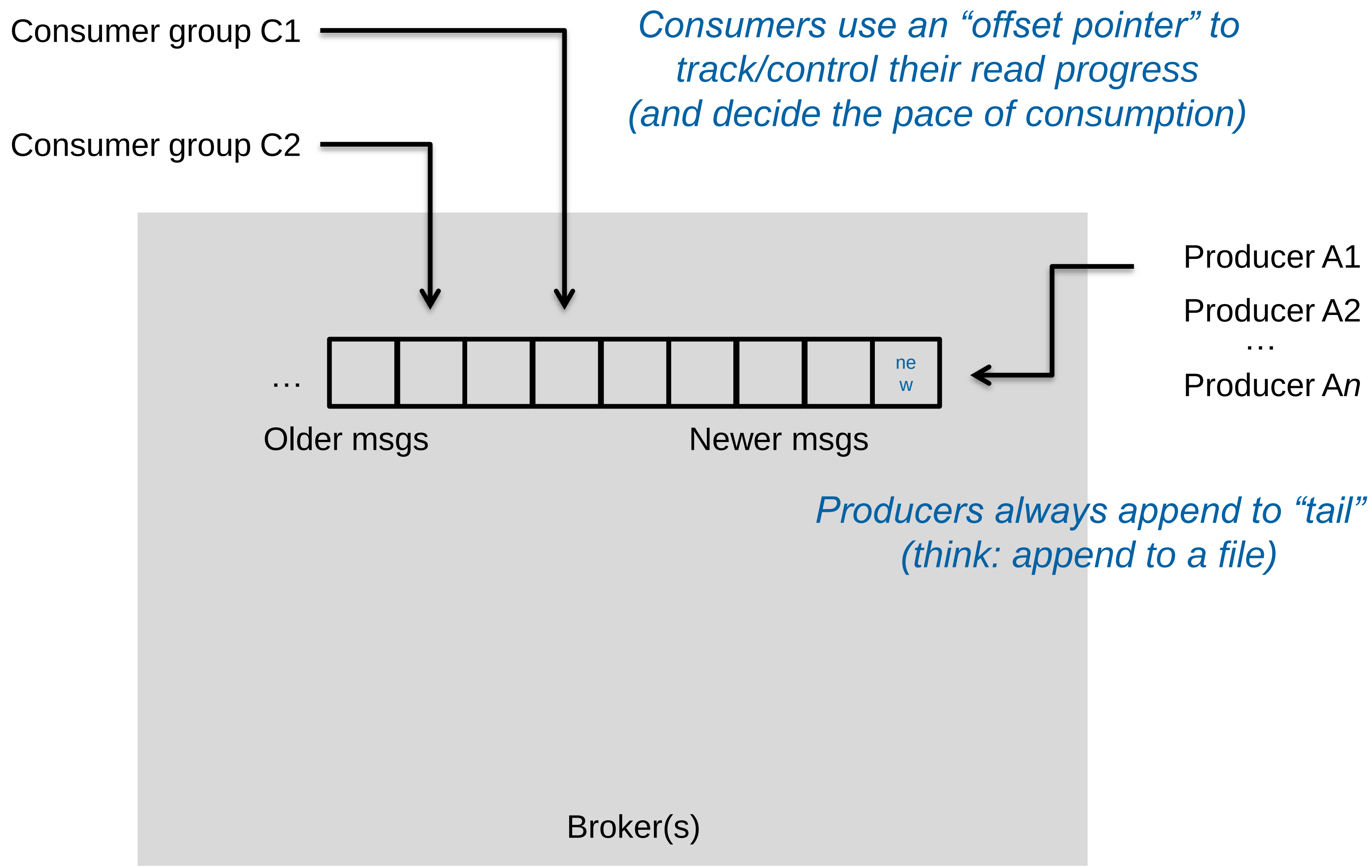

http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

# Topics

- **Topic:** feed name to which messages are published
  - Example: "zerg.hydra"

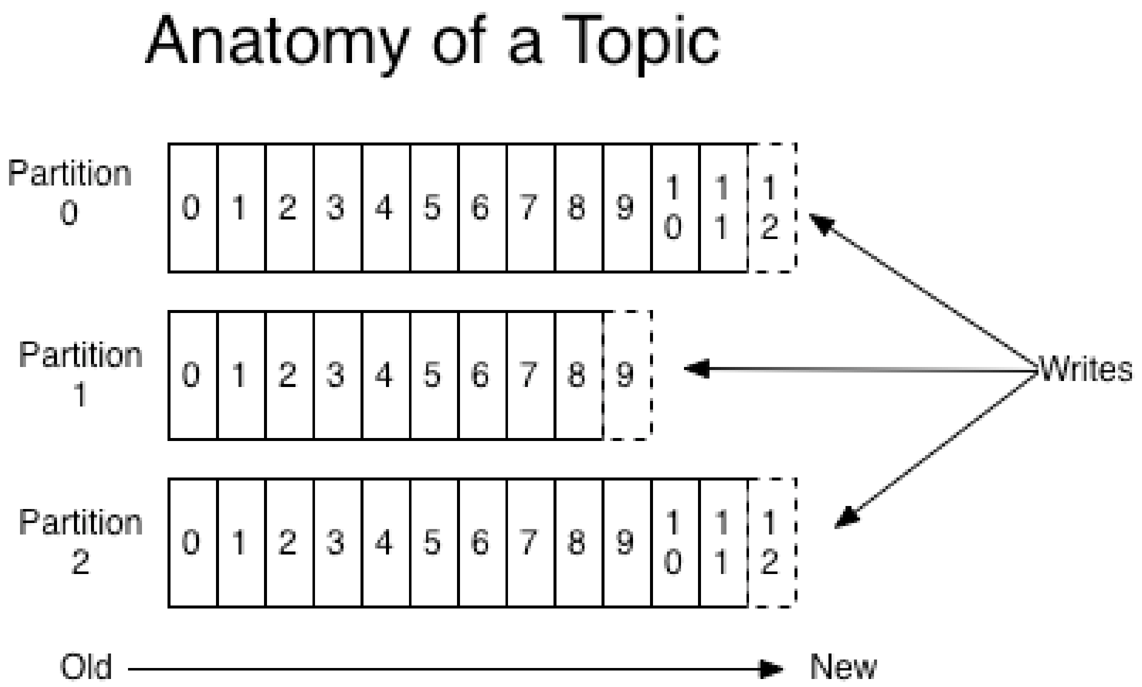


# Topics


Consumer group C1
Consumer group C2
Consumers use an "offset pointer" to track/control their read progress (and decide the pace of consumption)
Producer A1
Producer A2
…
Producer An
…
new
Older msgs
Newer msgs
Producers always append to "tail" (think: append to a file)
Broker(s)

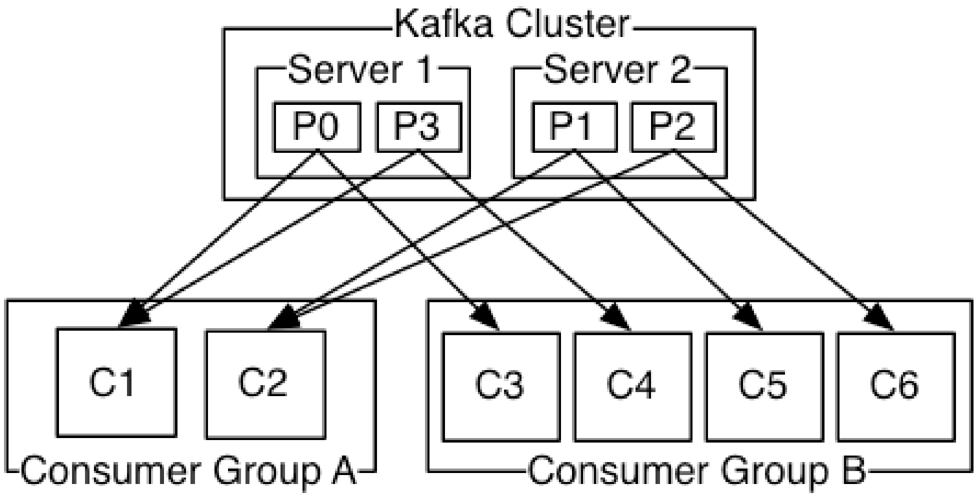
# Partitions

- A topic consists of **partitions.**
- Partition: **ordered + immutable** sequence of messages that is continually appended to



# Partitions

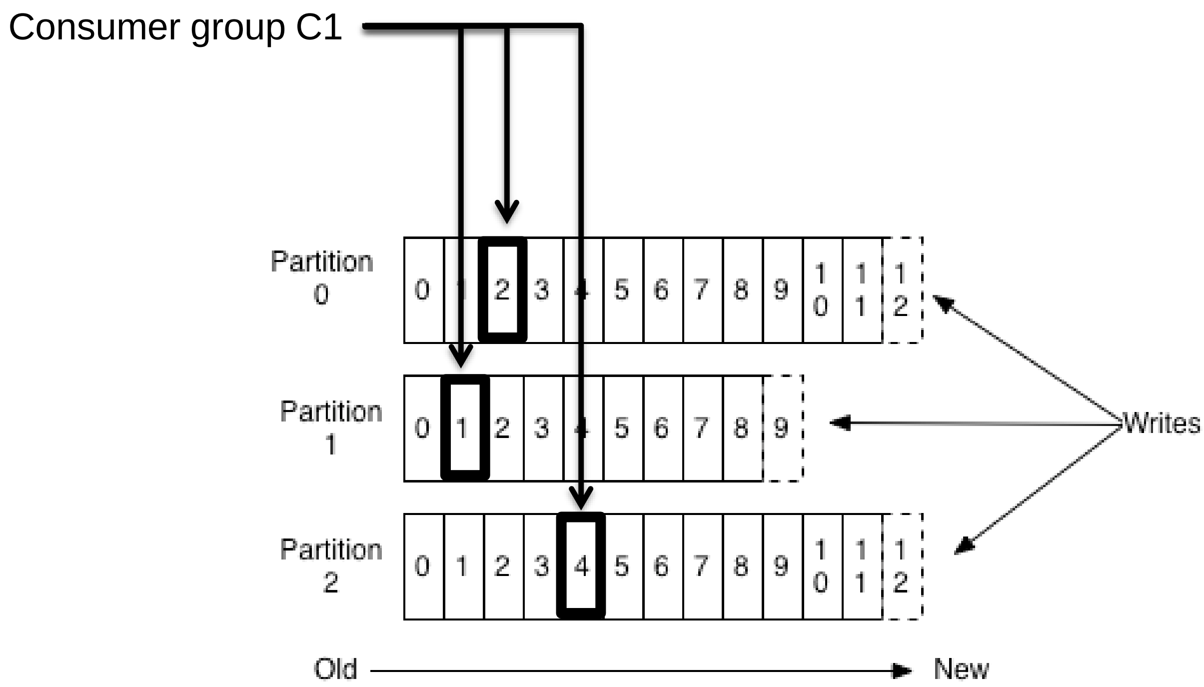
- #partitions of a topic is configurable
- #partitions determines **max** consumer (group) parallelism
  -




  - Consumer group A, with 2 consumers, reads from a 4-partition topic
  - Consumer group B, with 4 consumers, reads from the same topic

# Partition offsets

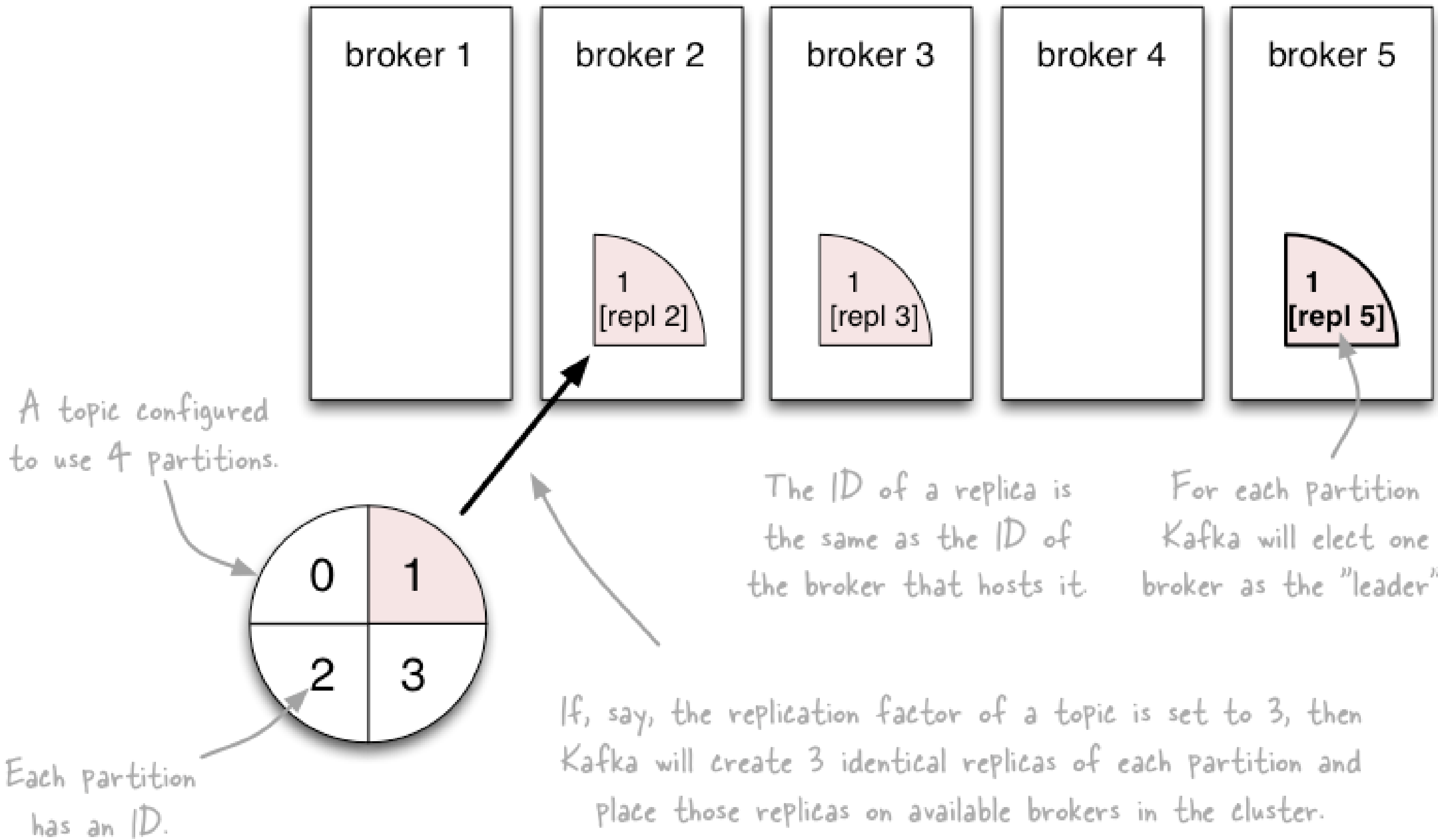
- **Offset**: messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via *(offset, partition, topic)* tuples



# Replicas of a partition

- **Replicas:** “backups” of a partition
  - They exist solely to prevent data loss.
  - Replicas are never read from, never written to.
    - They do NOT help to increase producer or consumer parallelism!
  - Kafka tolerates *(numReplicas - 1)* dead brokers before losing data
    - LinkedIn: numReplicas == 2 → 1 broker can die

# Topics vs. Partitions vs. Replicas




http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/

# Writing data to Kafka

# Writing data to Kafka

- You use Kafka "producers" to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.

- A simple example producer:

```
Properties props = new Properties();
props.put("metadata.broker.list", "...");
ProducerConfig config = new ProducerConfig(props);

Producer p = new Producer(ProducerConfig config);
KeyedMessage<K, V> msg = ...; // cf. later slides
p.send(KeyedMessage<K,V> message);
```

# Producers

- Two types of producers: “async” and “sync”

```
Properties props = new Properties();
props.put("producer.type", "async");
ProducerConfig config = new ProducerConfig(props);
```

  - Same API and configuration, but slightly different semantics.
  - What applies to a sync producer almost always applies to async, too.
  - Async producer is preferred when you want higher throughput.

# Producers

- Two aspects worth mentioning because they significantly influence Kafka performance:

    1. Message acking
    2. Batching of messages

# 1) Message acking

- Background:
  - In Kafka, a message is considered *committed* when "any required" replica for that partition have applied it to their data log.
  - Message acking is about conveying this "Yes, committed!" information back from the brokers to the producer client.
  - Exact meaning of "any required" is defined by `request.required.acks`.

- Only **producers** must configure acking
  - Exact behavior is configured via **`request.required.acks`**, which determines when a produce request is considered completed.
  - Allows you to trade **latency (speed)** <-> **durability (data safety)**.
  - Consumers: Acking and how you configured it on the side of producers do not matter to consumers because only committed messages are ever given out to consumers. They don't need to worry about potentially seeing a message that could be lost if the leader fails.

# 1) Message acking

better latency

better durability

- Typical values of `request.required.acks`
  - **`0`**: producer never waits for an ack from the broker.
    - Gives **the lowest latency** but the weakest durability guarantees.
  - **`1`**: producer gets an ack after the leader replica has received the data.
    - Gives better durability as the we wait until the lead broker acks the request.  Only msgs that were written to the now-dead leader but not yet replicated will be lost.
  - **`-1`**: producer gets an ack after *all* replicas have received the data.
    - Gives **the best durability** as Kafka guarantees that no data will be lost as long as at least one replica remains.

# 2) Batching of messages

- Batching improves throughput
  - Tradeoff is data loss if client dies before pending messages have been sent.

- You have two options to "batch" messages:

1. Use `send(`**`listOfMessages`**`)`.

```
producer.send(List<KeyedMessage<K,V>> messages);
```

   - Sync producer: will send this list ("batch") of messages *right now*. Blocks!
   - Async producer: will send this list of messages in background "as usual", i.e. according to batch-related configuration settings. Does not block!

2. Use `send(`**`singleMessage`**`)` with async producer.

```
producer.send(KeyedMessage<K,V> message);
```

   - For async the behavior is the same as `send(listOfMessages)`.

# Reading data from Kafka

# Reading data from Kafka

- You use Kafka “consumers” to write data to Kafka brokers.
  - Available for JVM (Java, Scala), C/C++, Python, Ruby, etc.

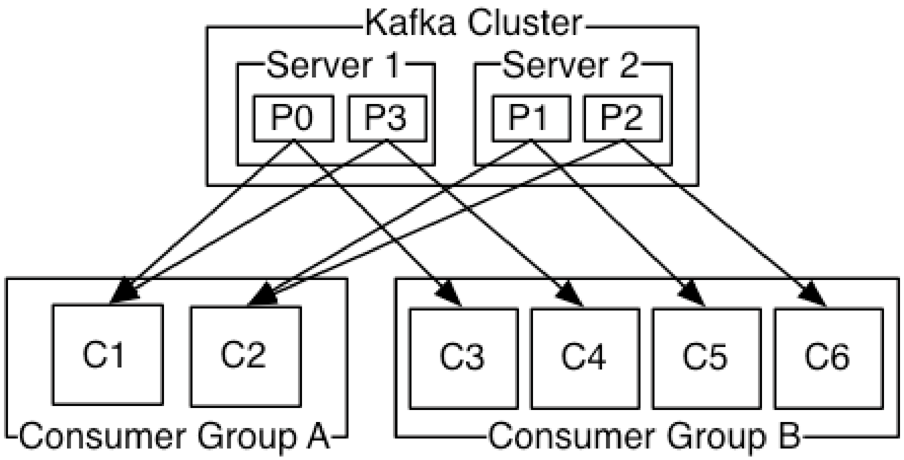
# Reading data from Kafka

- Consumers *pull* from Kafka (there's no push)
  - Allows consumers to control their pace of consumption.
  - Allows to design downstream apps for **average** load, not peak load

- Consumers are responsible to track their read positions aka "offsets"

# Reading data from Kafka

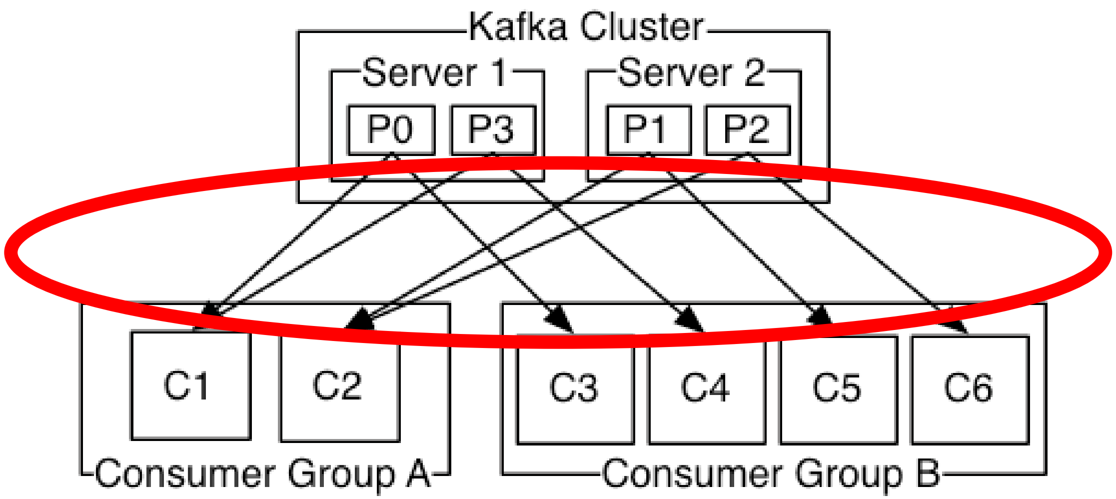
- Consumer "groups"
  - Allows multi-threaded and/or multi-machine consumption from Kafka topics.
  - Consumers "join" a group by using the same `group.id`
  - Kafka guarantees a message is only ever read by a single consumer in a group.
    - Kafka assigns the partitions of a topic to the consumers in a group so that each partition is consumed by exactly one consumer in the group.
    - Maximum parallelism of a consumer group: **#consumers** (in the group) <= **#partitions**



# Guarantees when reading data from Kafka

- A message is only ever read by a single consumer in a group.
- A consumer sees messages in the order they were stored in the log.
- The order of messages is only guaranteed within a partition.

# Rebalancing: how consumers meet brokers




- The assignment of brokers – via the partitions of a topic – to consumers is quite **important**, and it is **dynamic** at run-time.

# probabilistic data structures for Big data and streaming

# Streams Processing Challenges

Inherent challenges

Latency requirements

Space bounds

System challenges

Bursty behavior and load balancing

Out-of-order message delivery and non-determinism

Consistency semantics (at most once, exactly once, at least once)

# Algorithmic Solutions

Throw away data

Sampling

Accepting some approximations

Hashing

# Reservoir Sampling

Task: select *s* elements from a stream of size *N* with uniform probability

*N* can be very very large

We might not even know what *N* is! (infinite stream)

Solution: Reservoir sampling

Store first *s* elements

For the *k*-th element thereafter, keep with probability *s/k*

(randomly discard an existing element)

Example: *s* = 10

Keep first 10 elements

11th element: keep with 10/11

12th element: keep with 10/12

…

# Reservoir Sampling: How does it work?

Example: $s = 10$

Keep first 10 elements

11th element: keep with 10/11

If we decide to keep it: sampled uniformly by definition

probability existing item is discarded: $10/11 \times 1/10 = 1/11$

probability existing item survives: 10/11

General case: at the $(k + 1)$th element

Probability of selecting each item up until now is $s/k$

Probability existing item is discarded: $s/(k+1) \times 1/s = 1/(k + 1)$

Probability existing item survives: $k/(k + 1)$

Probability each item survives to $(k + 1)$th round:

$(s/k) \times k/(k + 1) = s/(k + 1)$

# Hashing for Three Common Tasks

Cardinality estimation

What's the cardinality of set *S*?

How many unique visitors to this page?

HashSet HLL counter

Set membership

Is *x* a member of set *S*?

Has this user seen this ad before?

HashSet Bloom Filter

Frequency estimation

How many times have we observed *x*?

How many queries has this user issued?

HashMap CMS

# HyperLogLog Counter

Task: cardinality estimation of set

size() → number of unique elements in the set

Observation: hash each item and examine the hash code

On expectation, 1/2 of the hash codes will start with 0

On expectation, 1/4 of the hash codes will start with 00

On expectation, 1/8 of the hash codes will start with 000

On expectation, 1/16 of the hash codes will start with 0000

…

How do we take advantage of this observation?

# Bloom Filters

Task: keep track of set membership

put($x$) → insert $x$ into the set

contains($x$) → yes if $x$ is a member of the set

Components

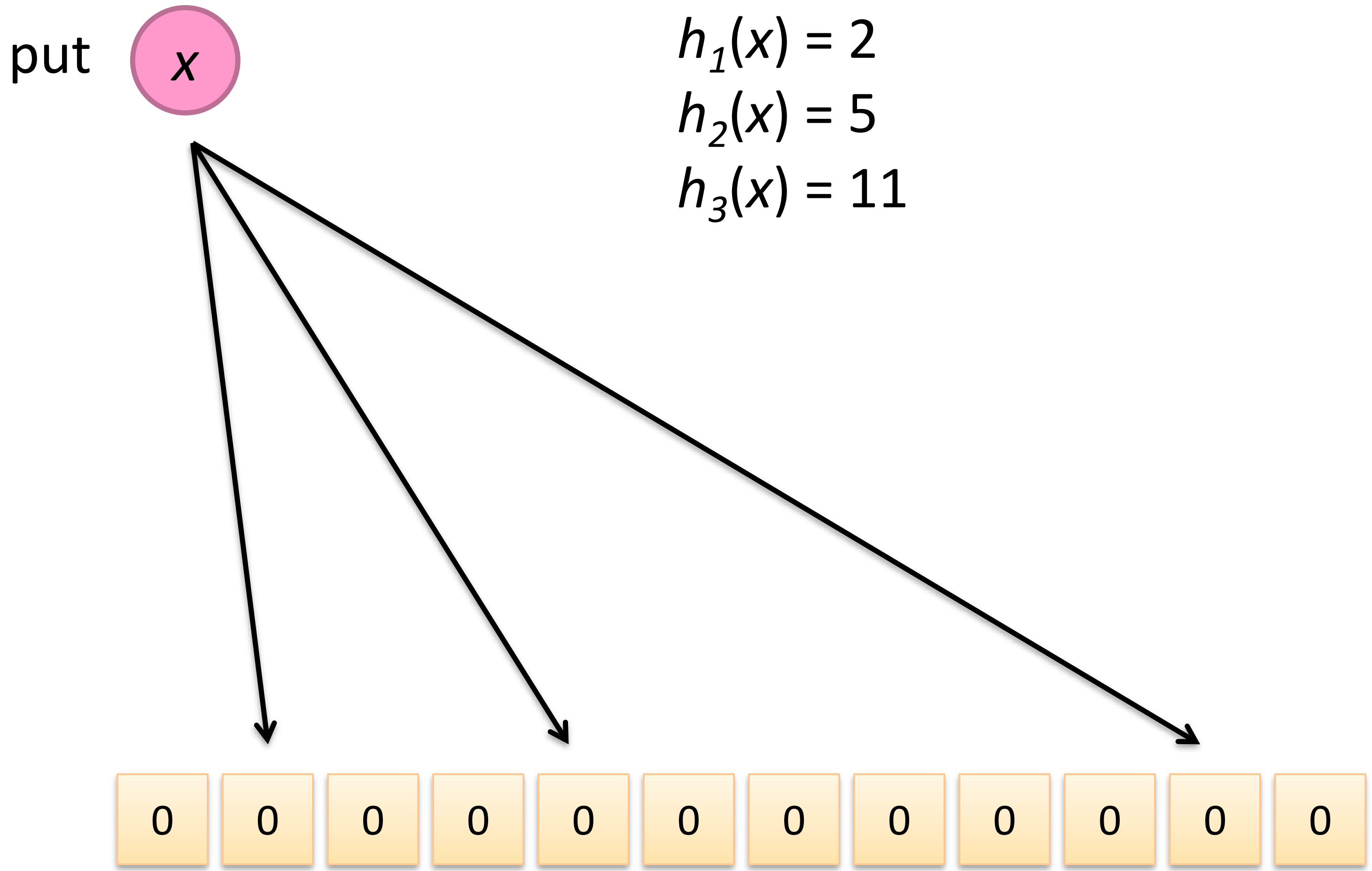
$m$-bit bit vector

$k$ hash functions: $h_1$ … $h_k$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Bloom Filters: put

$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$



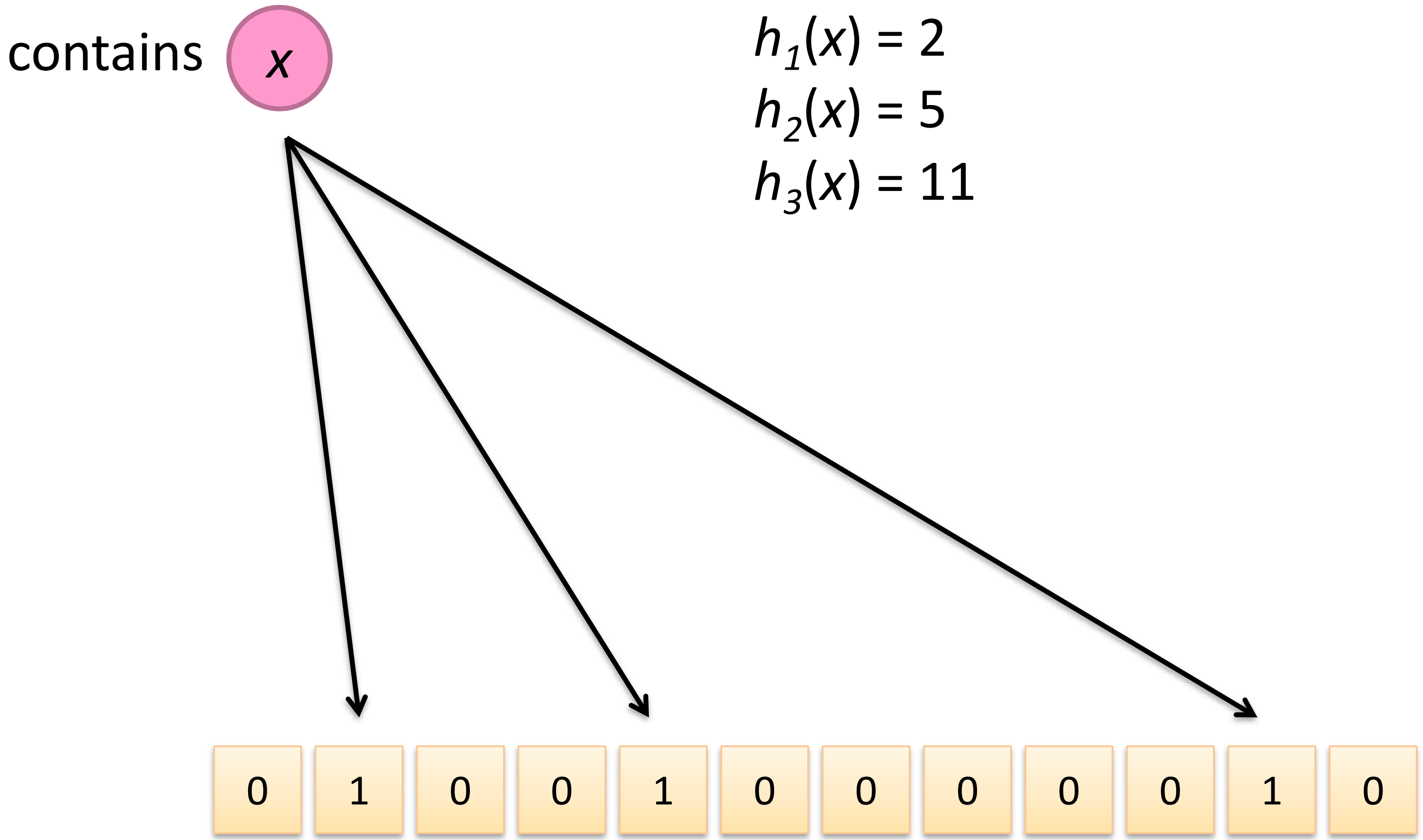
# Bloom Filters: put

put *x*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

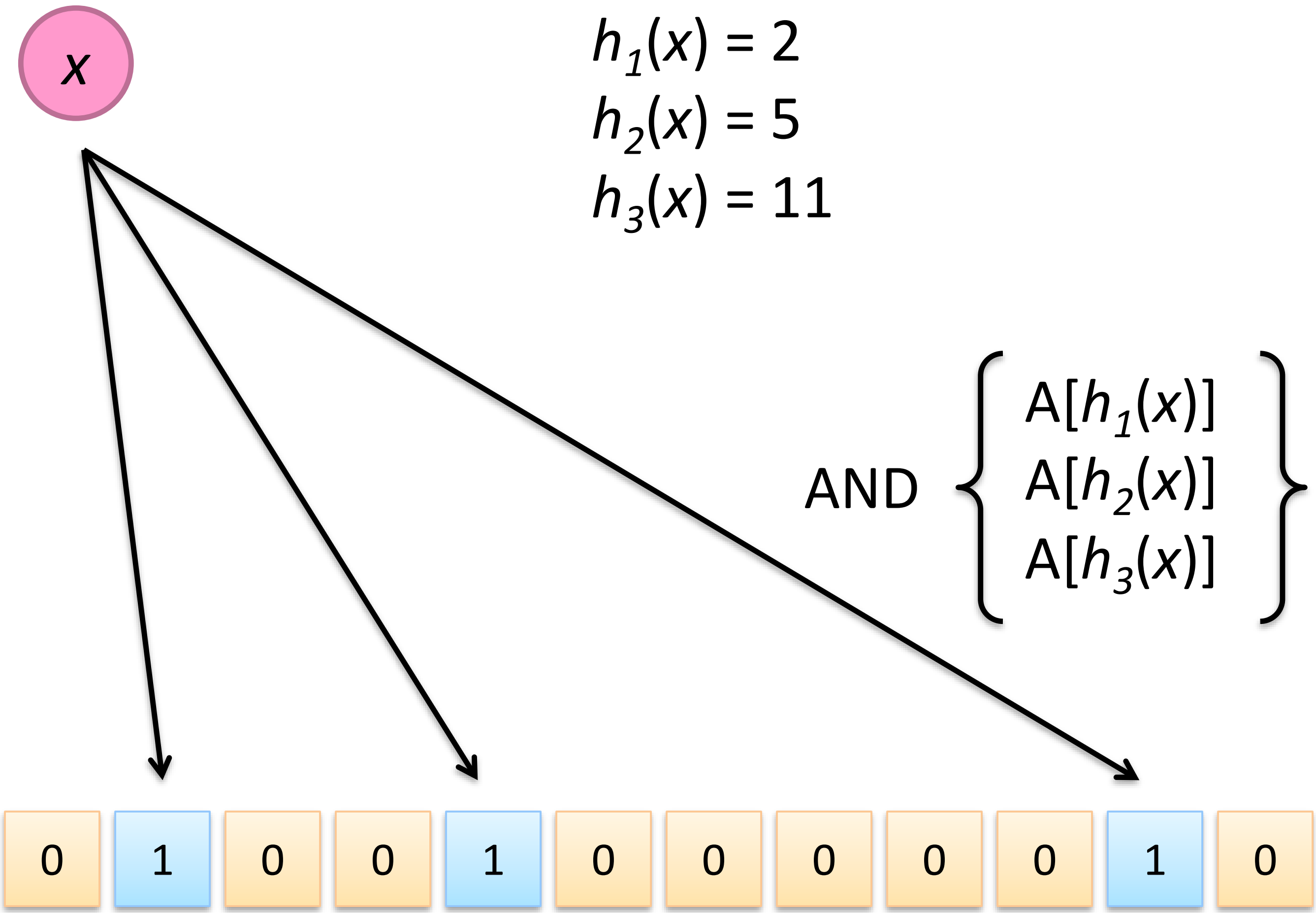
# Bloom Filters: contains

$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$



# Bloom Filters: contains

contains

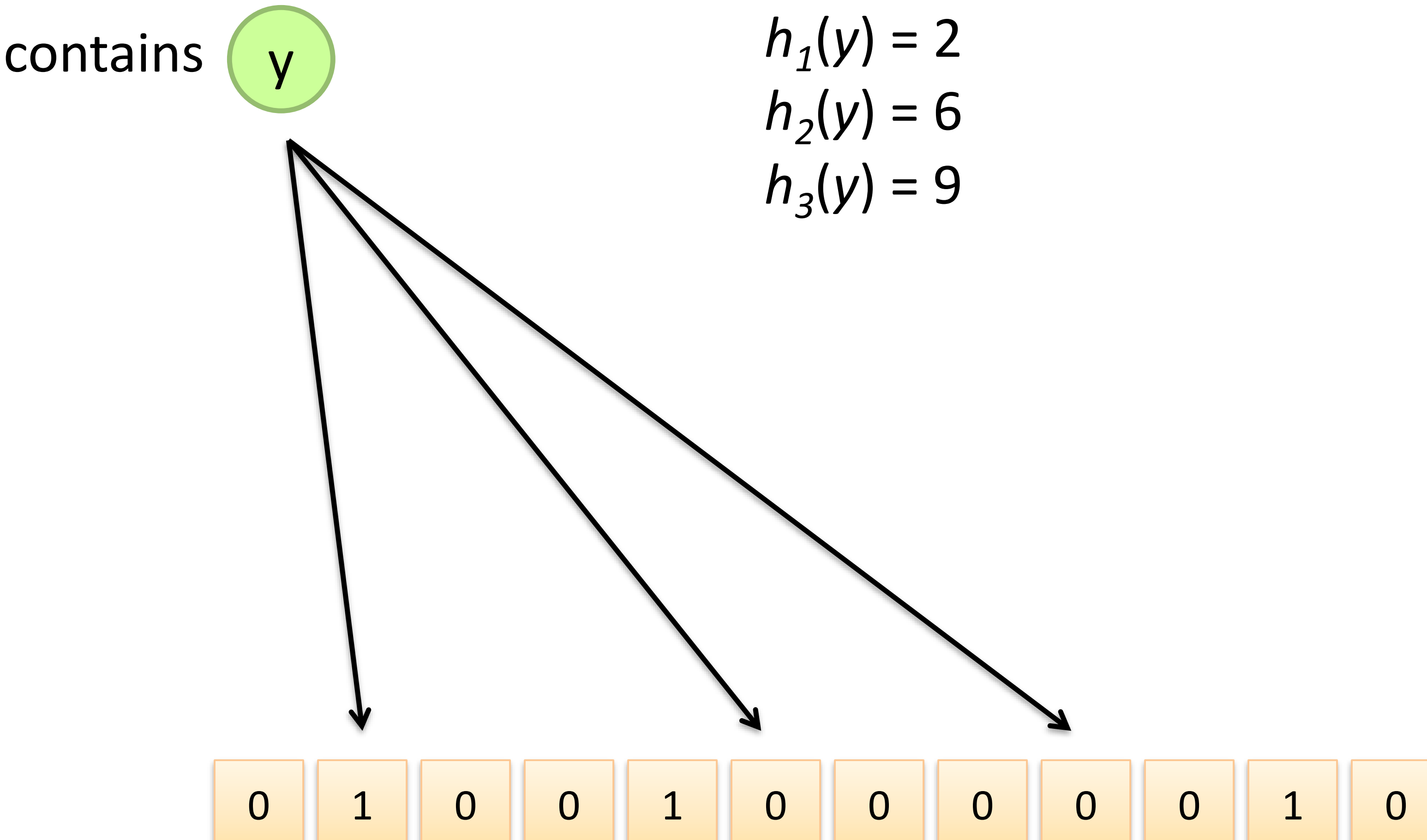



$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$

$$\text{AND}\left\{\begin{array}{l} A[h_1(x)] \\ A[h_2(x)] \\ A[h_3(x)] \end{array}\right\} = \text{YES}$$

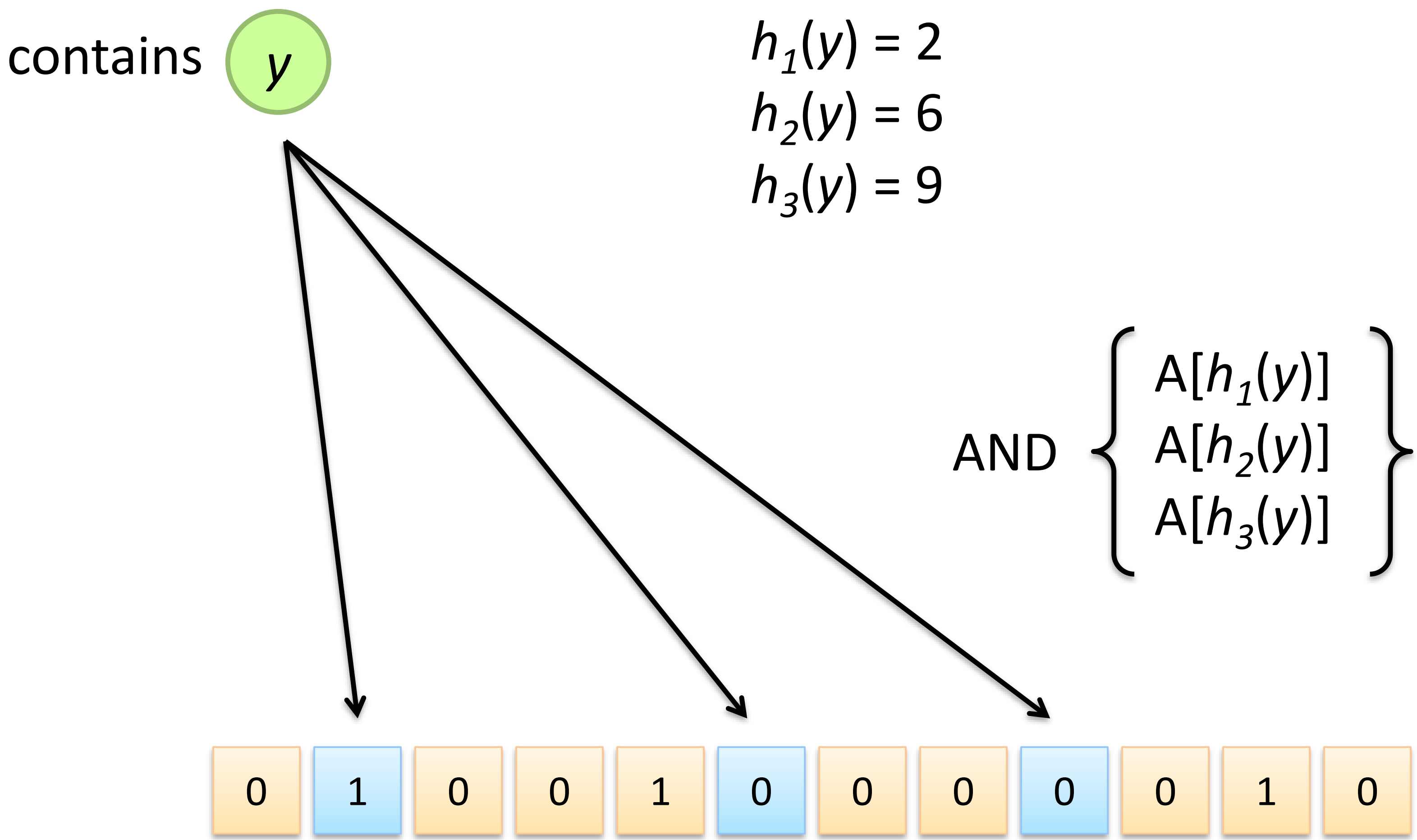
# Bloom Filters: contains

$h_1(y) = 2$
$h_2(y) = 6$
$h_3(y) = 9$



# Bloom Filters: contains

$h_1(y) = 2$
$h_2(y) = 6$
$h_3(y) = 9$




$$\text{AND} \left\{ \begin{array}{l} A[h_1(y)] \\ A[h_2(y)] \\ A[h_3(y)] \end{array} \right\} = \text{NO}$$

What's going on here?

# Bloom Filters

Error properties: contains(*x*)

False positives possible

No false negatives

Usage

Constraints: capacity, error probability

Tunable parameters: size of bit vector *m*, number of hash functions *k*

# Count-Min Sketches

Task: frequency estimation

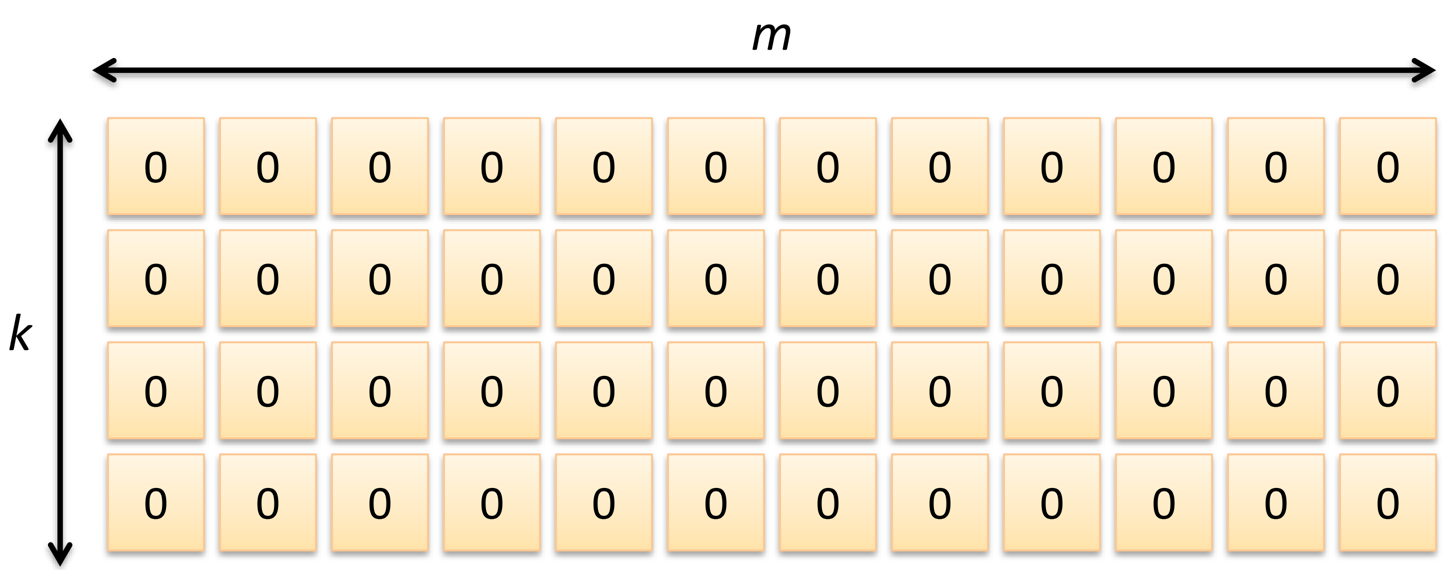
put($x$) → increment count of $x$ by one

get($x$) → returns the frequency of $x$

Components

$m$ by $k$ array of counters

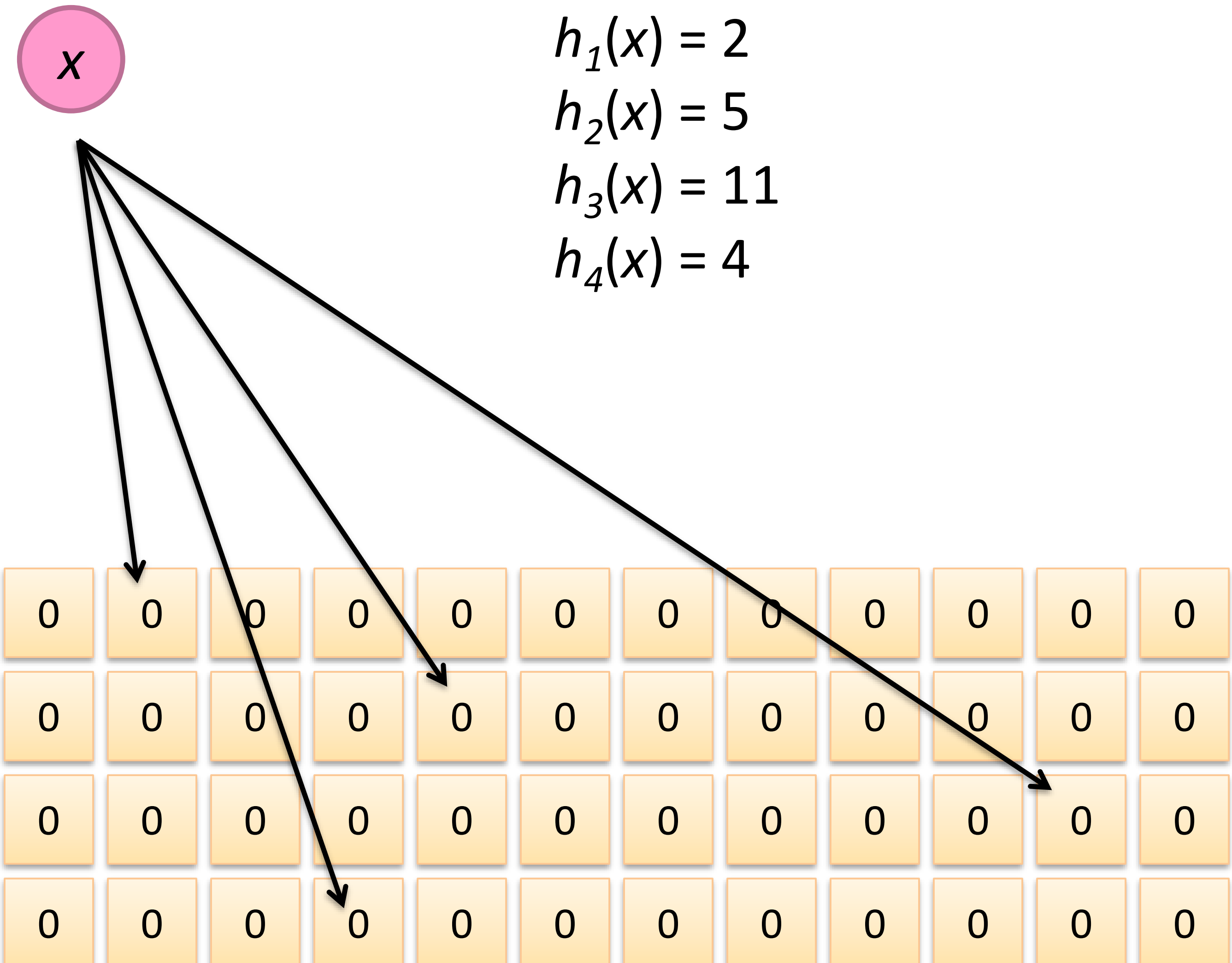
$k$ hash functions: $h_1$ … $h_k$



# Count-Min Sketches: put

put

$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$
$h_4(x) = 4$



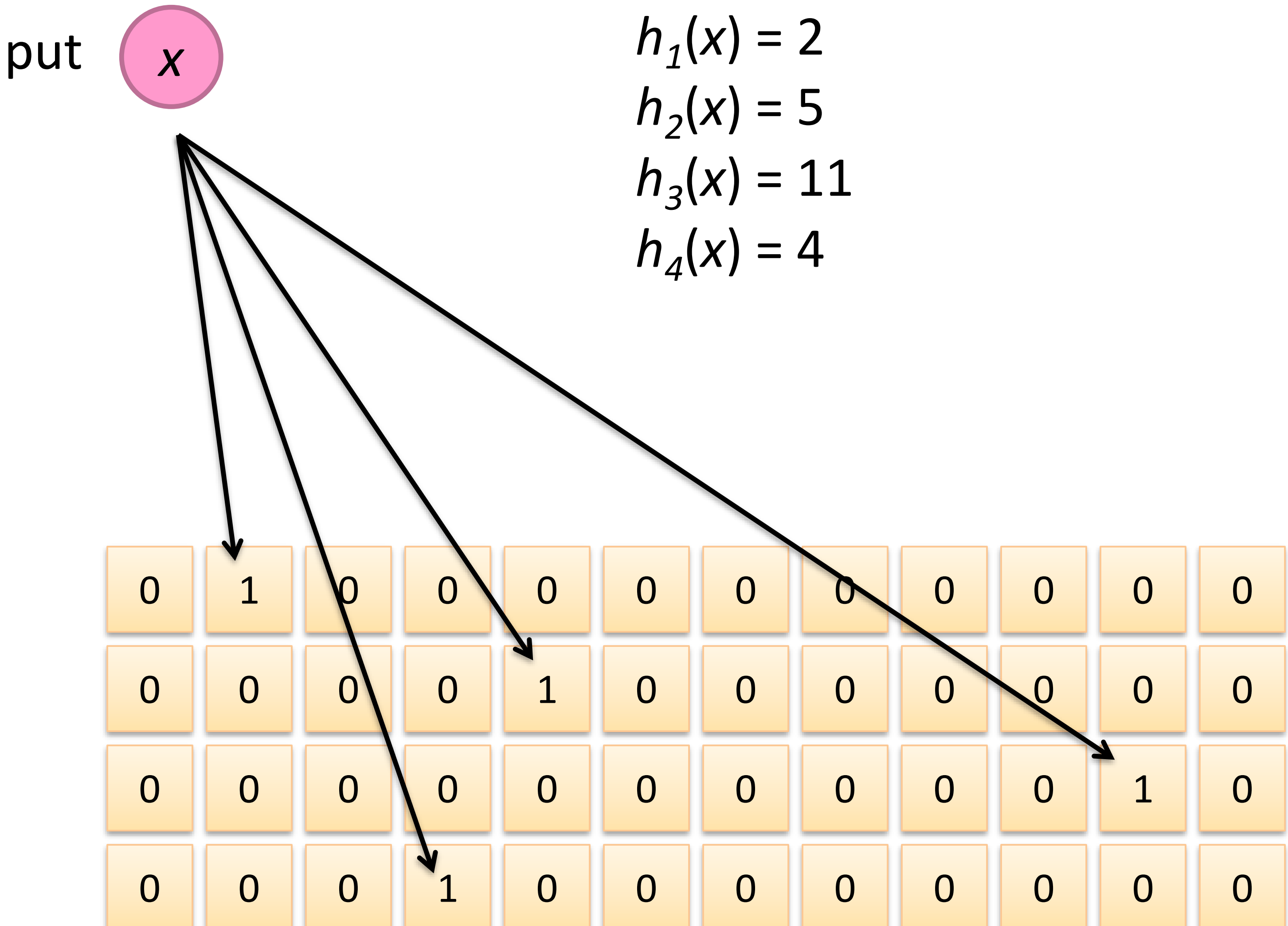
# Count-Min Sketches: put

put $x$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Count-Min Sketches: put

$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$
$h_4(x) = 4$



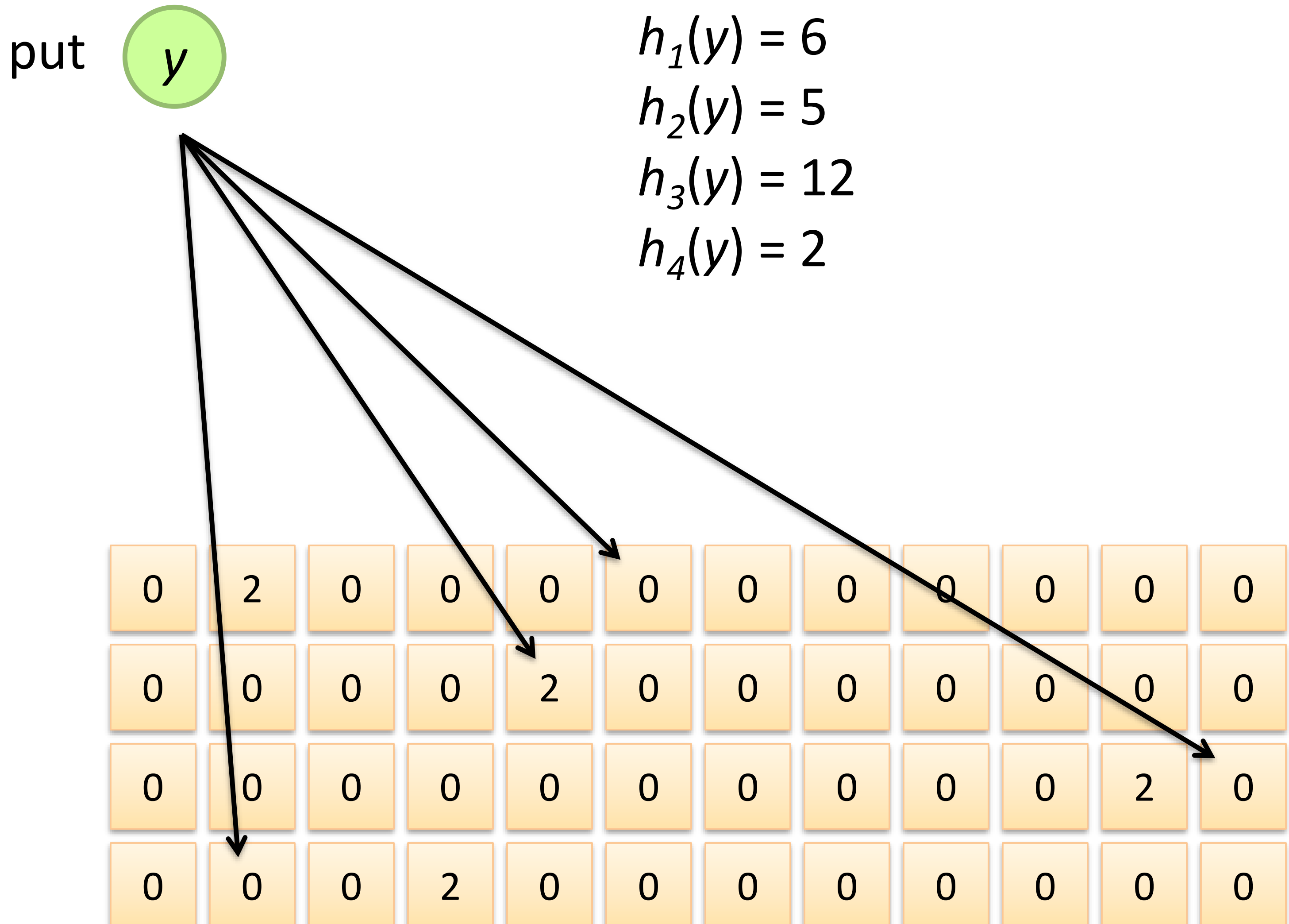
# Count-Min Sketches: put

put *x*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Count-Min Sketches: put

$h_1(y) = 6$
$h_2(y) = 5$
$h_3(y) = 12$
$h_4(y) = 2$

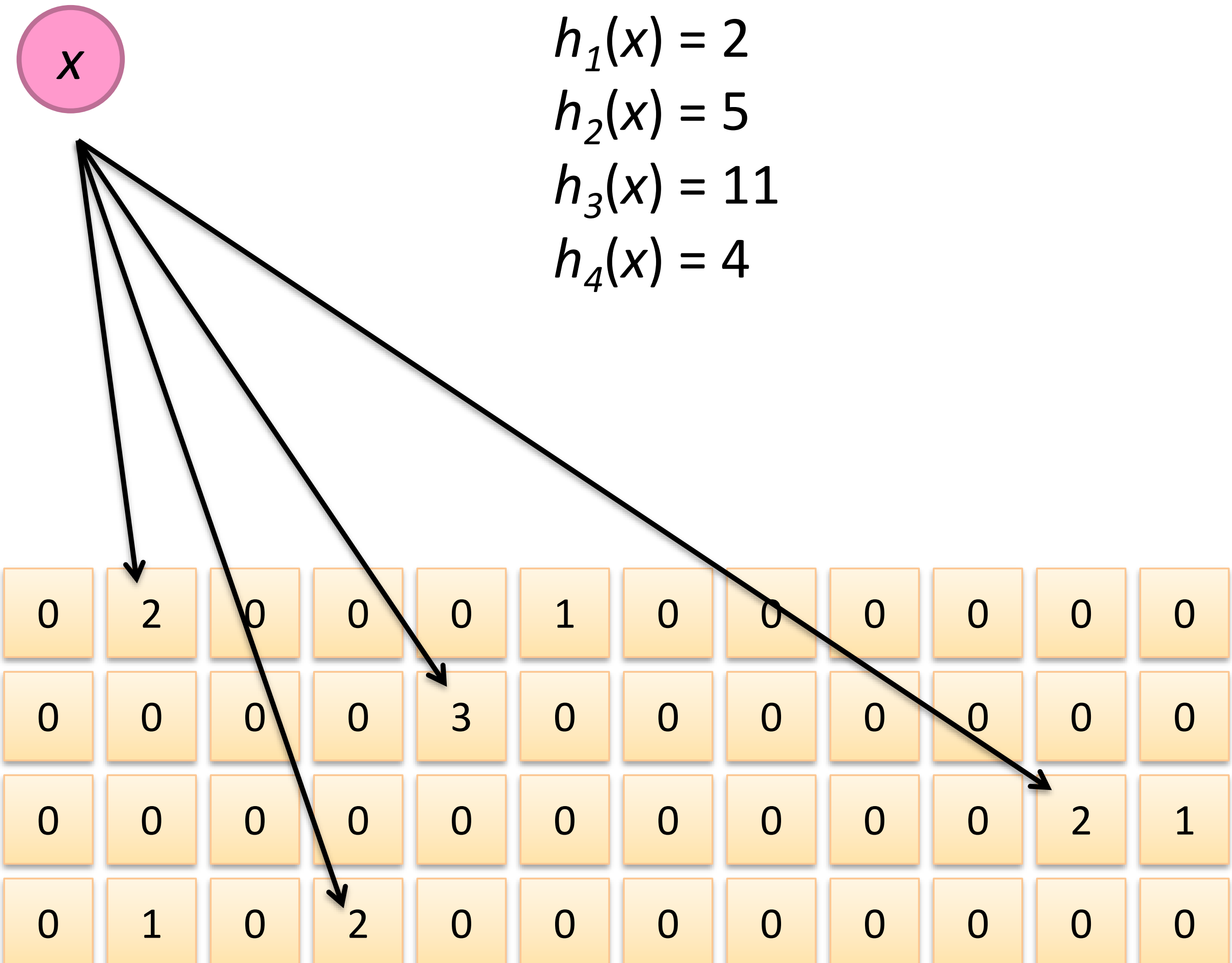


# Count-Min Sketches: put

put y

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Count-Min Sketches: get

get 

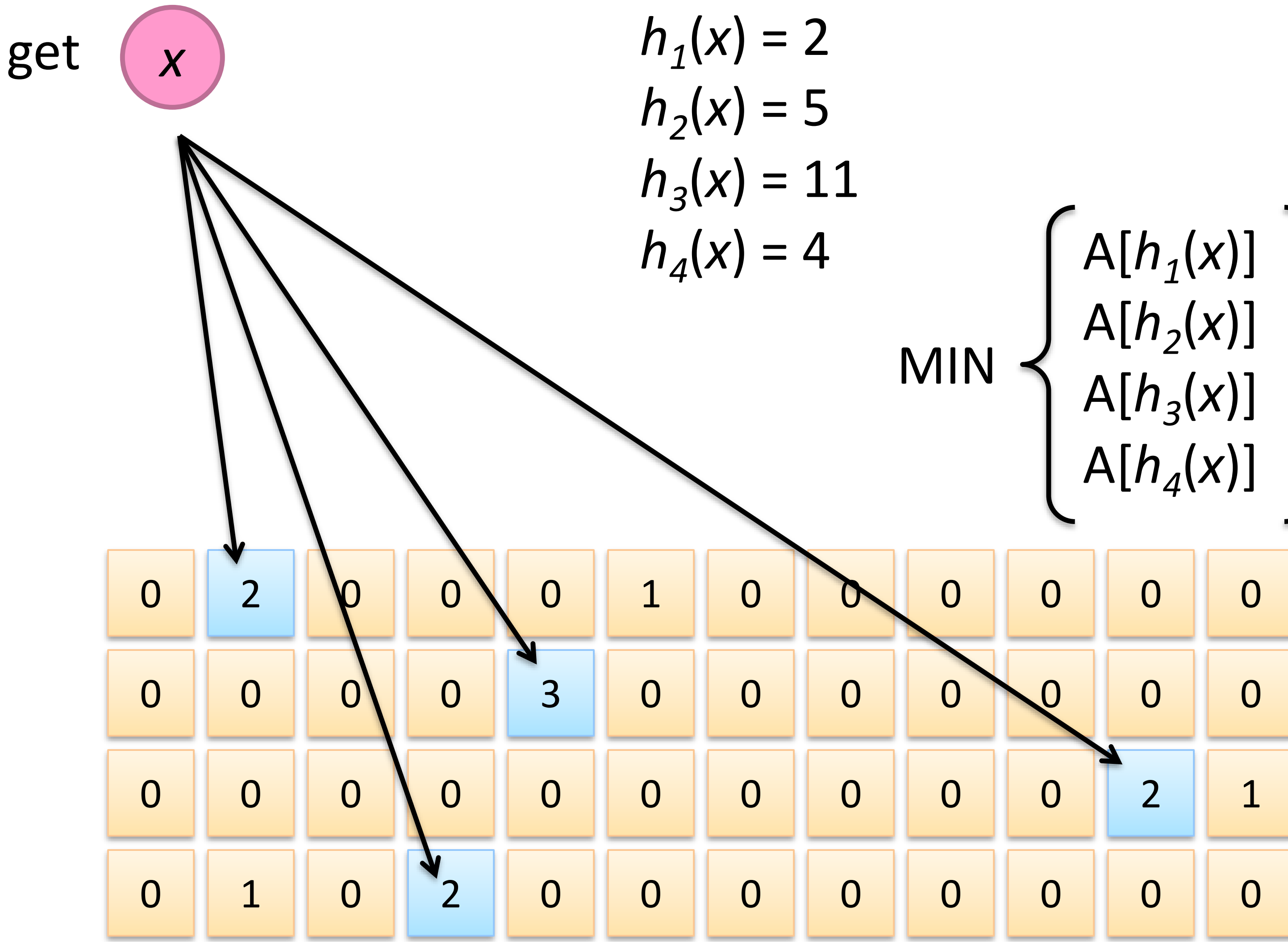

$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$
$h_4(x) = 4$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Count-Min Sketches: get

$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$
$h_4(x) = 4$

$$\text{MIN}\left\{\begin{array}{l} A[h_1(x)] \\ A[h_2(x)] \\ A[h_3(x)] \\ A[h_4(x)] \end{array}\right\} = 2$$



# Count-Min Sketches: get

get *y*

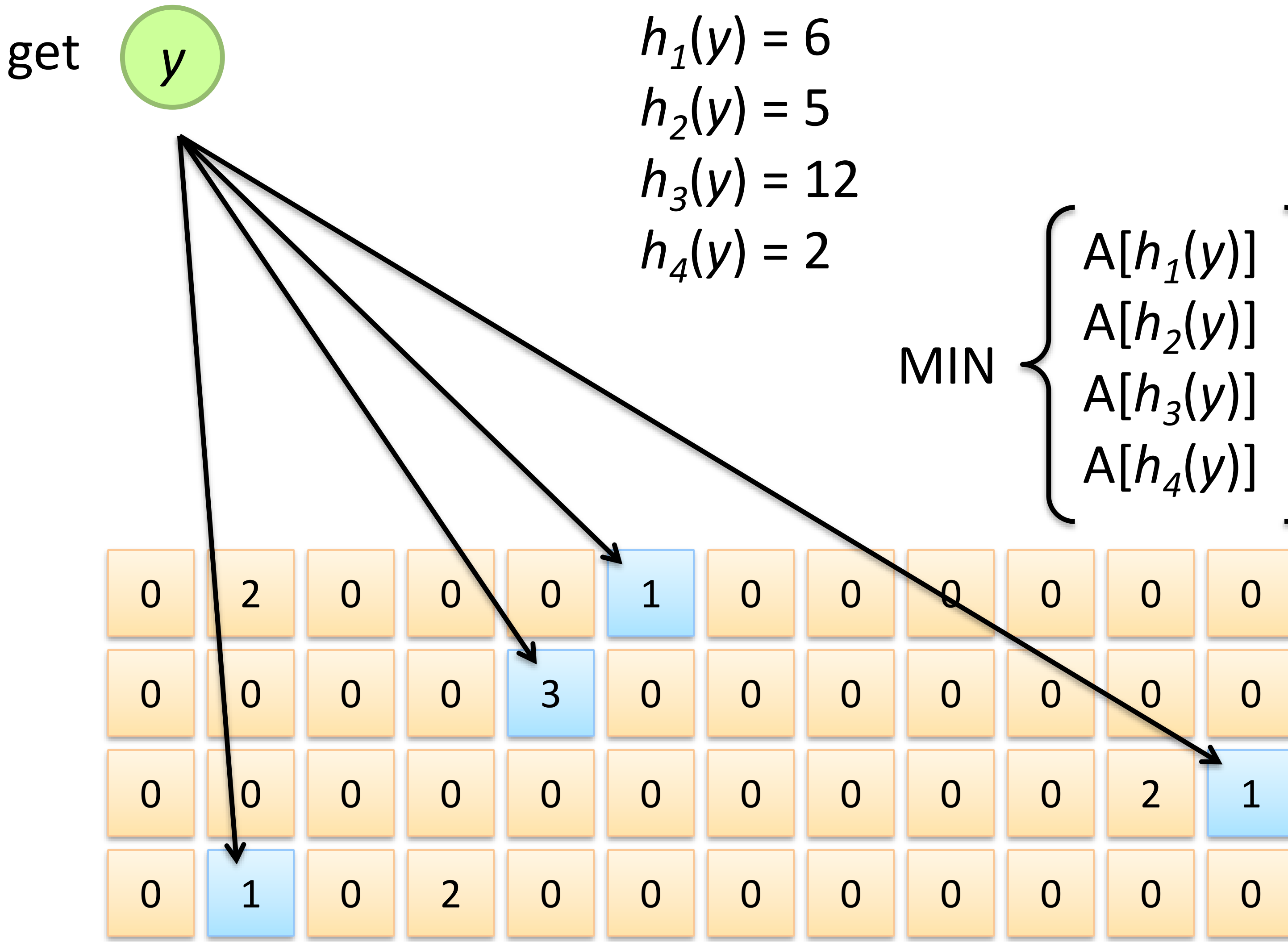
$h_1(y) = 6$
$h_2(y) = 5$
$h_3(y) = 12$
$h_4(y) = 2$

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Count-Min Sketches: get

$h_1(y) = 6$
$h_2(y) = 5$
$h_3(y) = 12$
$h_4(y) = 2$

$$\text{MIN}\begin{Bmatrix} A[h_1(y)] \\ A[h_2(y)] \\ A[h_3(y)] \\ A[h_4(y)] \end{Bmatrix} = 1$$



# Count-Min Sketches

Error properties: get($x$)

Reasonable estimation of heavy-hitters

Frequent over-estimation of tail

Usage

Constraints: number of distinct events, distribution of events, error bounds

Tunable parameters: number of counters $m$ and hash functions $k$, size of counters

# Hashing for Three Common Tasks

Cardinality estimation

What's the cardinality of set *S*?

How many unique visitors to this page?

HashSet HLL counter

Set membership

Is *x* a member of set *S*?

Has this user seen this ad before?

HashSet Bloom Filter

Frequency estimation

How many times have we observed *x*?

How many queries has this user issued?

HashMap CMS