Search Engine Architecture

12. Stream Processing

Today's Agenda

- Basics of stream processing
- Approximation algorithms
- Architectures for stream processing

What is stream processing?

- Processing on unbounded data set
- Cf. batch - bounded
- Execution engines typically optimized for bounded or unbounded
- Often low-latency / approximations / weak consistency

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

What to do with data streams?

- Network traffic monitoring
- Datacenter telemetry monitoring
- Sensor networks monitoring
- Credit card fraud detection
- Stock market analysis
- Online mining of click streams
- Monitoring social media streams

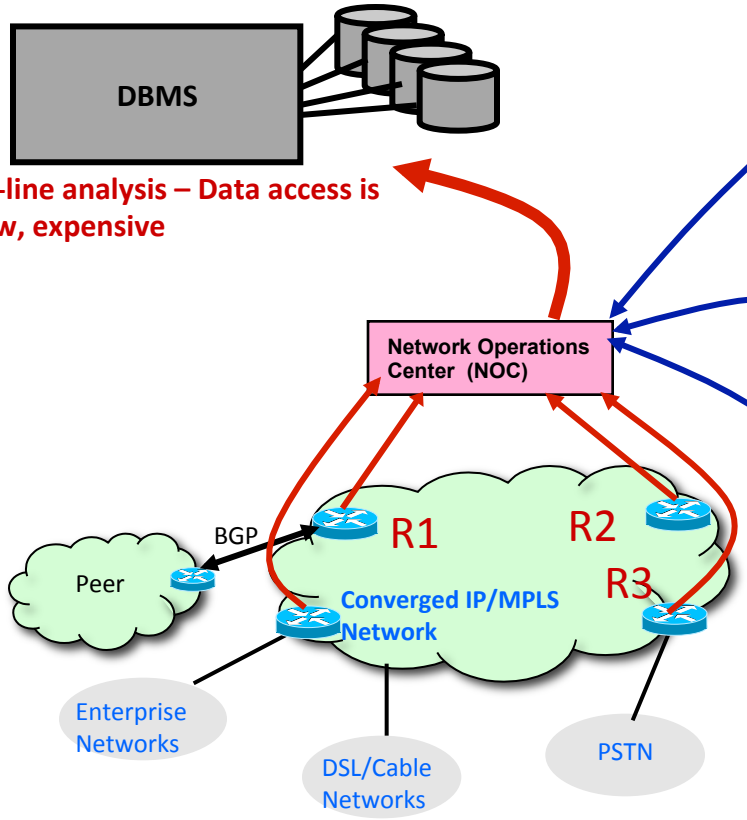
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

What's the scale? Packet data streams

- Single 2 Gb/sec link; say avg. packet size is 50 bytes
  - Number of packets/sec = 5 million
  - Time per packet = 0.2 microseconds
- If we only capture header information per packet: source/destination IP, time, no. of bytes, etc. – at least 10 bytes
  - 50 MB per second
  - 4+ TB per day
  - **Per link!**

What if you wanted to do deep-packet inspection?

Source: Minos Garofalakis, Berkeley CS 286



What are the top (most frequent) 1000 (source, dest) pairs seen by R1 over the last month?

How many distinct (source, dest) pairs have been seen by both R1 and R2 but not R3?

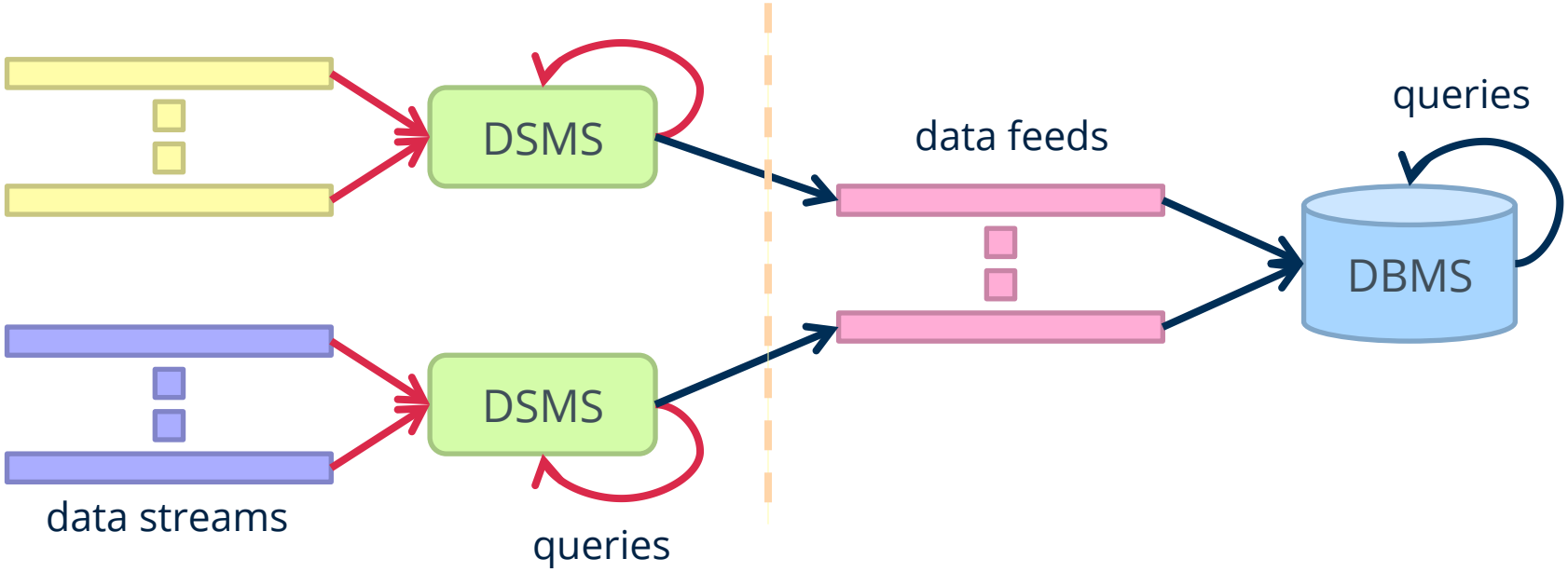
Set-Expression Query

```
SELECT COUNT (R1.source, R1.dest)
FROM  R1, R2
WHERE R1.source = R2.source
```

SQL Join Query

Source: Minos Garofalakis, Berkeley CS 286

Common Architecture



- Data stream management system (DSMS) at observation points
  - Voluminous streams-in, reduced streams-out
- Database management system (DBMS)
  - Outputs of DSMS can be treated as data feeds to databases

Source: Peter Bonz

DBMS vs. DSMS

DBMS

- Model: persistent relations
- Relation: tuple set/bag
- Data update: modifications
- Query: transient
- Query answer: exact
- Query evaluation: arbitrary
- Query plan: fixed

DSMS

- Model: (mostly) transient relations
- Relation: tuple sequence
- Data update: appends
- Query: persistent
- Query answer: approximate
- Query evaluation: one pass
- Query plan: adaptive

Source: Peter Bonz

What makes it hard?

- Intrinsic challenges:
  - Volume
  - Velocity
  - Limited storage
  - Strict latency requirements
  - Out-of-order delivery
- System challenges:
  - Load balancing
  - Unreliable message delivery
  - Fault-tolerance
  - Consistency semantics (lossy, exactly once, at least once, etc.)

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

What exactly do you do?

- "Standard" relational operations:
  - Select
  - Project
  - Transform (i.e., apply custom UDF)
  - Group by
  - Join
  - Aggregations

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Issues of Semantics

- Group by… aggregate
  - When do you stop grouping and start aggregating?
- Joining a stream and a static source
  - Simple lookup
- Joining two streams
  - How long do you wait for the join key in the other stream?
- Joining two streams, group by and aggregation
  - When do you stop joining?

What's the solution?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

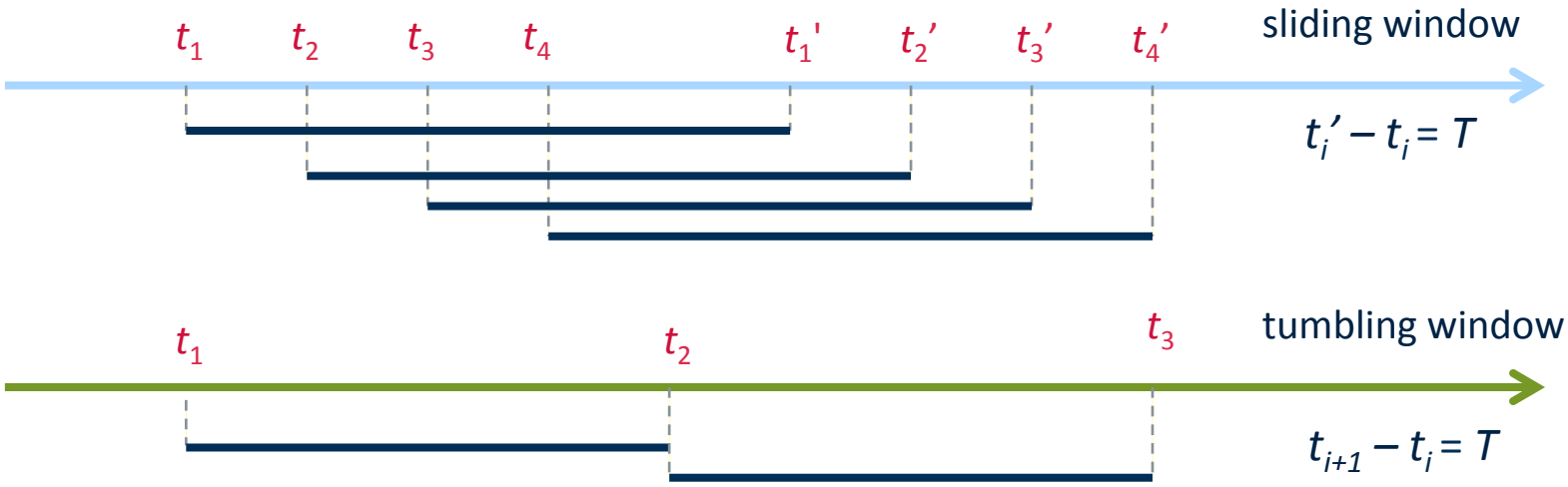
Windows

- Mechanism for extracting finite relations from an infinite stream
- Windows restrict processing scope:
  - Windows based on ordering attributes (e.g., time)
  - Windows based on item (record) counts
  - Windows based on explicit markers (e.g., punctuations)
  - Variants (e.g., some semantic partitioning constraint)

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Windows on Ordering Attributes

- Assumes the existence of an attribute that defines the order of stream elements (e.g., time)
- Let *T* be the window size in units of the ordering attribute



Source: Peter Bonz

Windows on Counts

- Window of size *N* elements (sliding, tumbling) over the stream
- Challenges:
  - Problematic with non-unique timestamps: non-deterministic output
  - Unpredictable window size (and storage requirements)

Source: Peter Bonz

Windows from “Punctuations”

- Application-inserted “end-of-processing”
  - Example: stream of actions… “end of user session”
- Properties
  - Advantage: application-controlled semantics
  - Disadvantage: unpredictable window size (too large or too small)

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Common Techniques

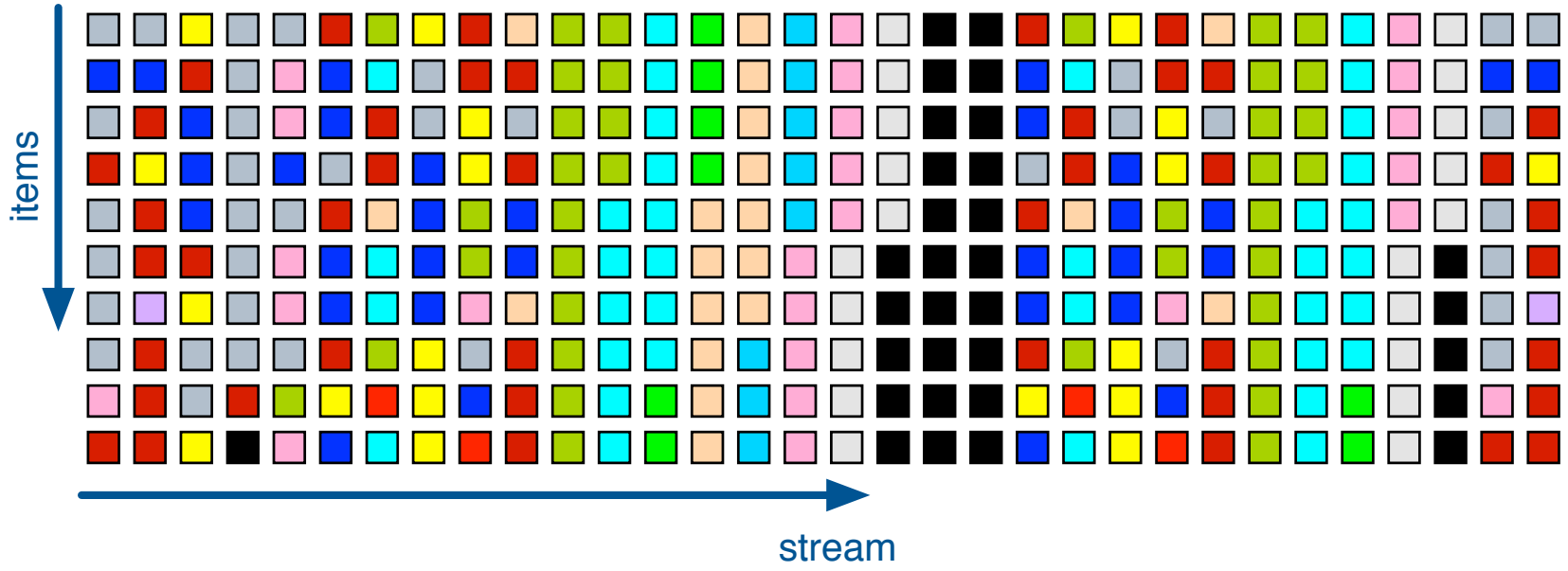
Source: Wikipedia (Forge)

“Hello World” Stream Processing

- Problem:
  - Count the frequency of items in the stream
- Why?
  - Take some action when frequency exceeds a threshold
  - Data mining:
    raw counts → co-occurring counts → association rules

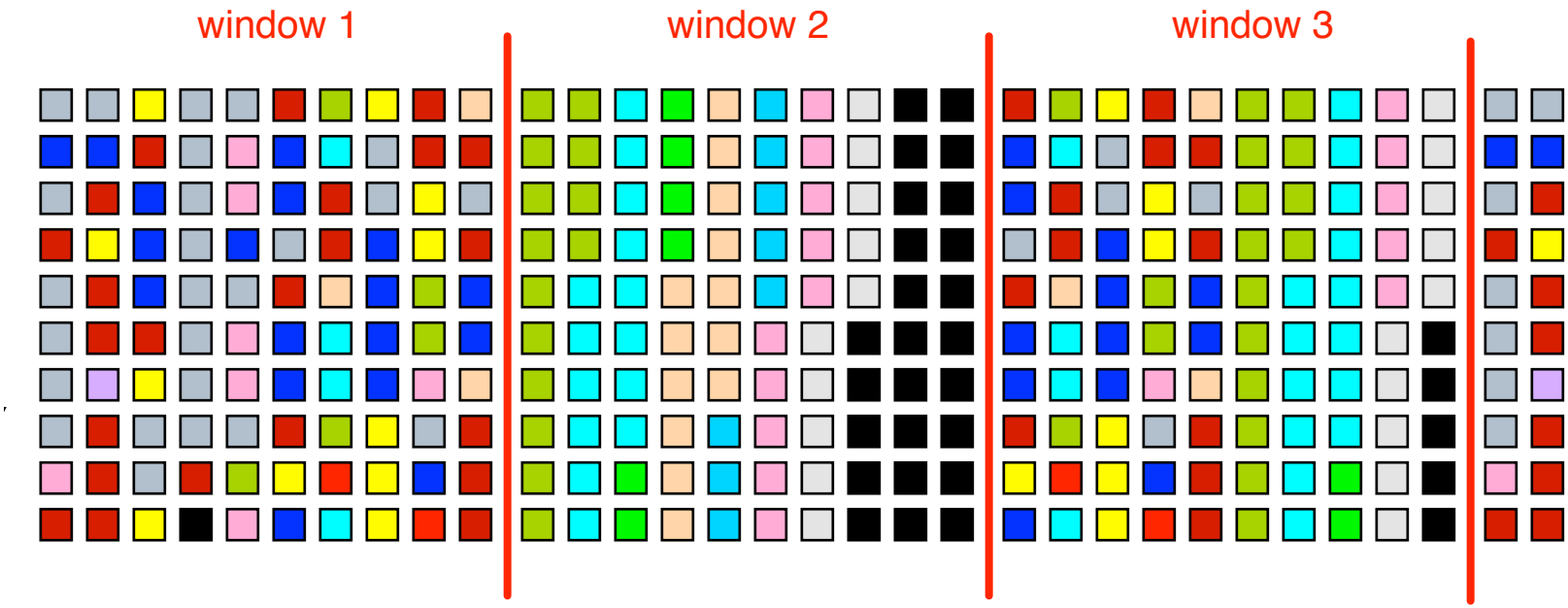
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

The Raw Stream…



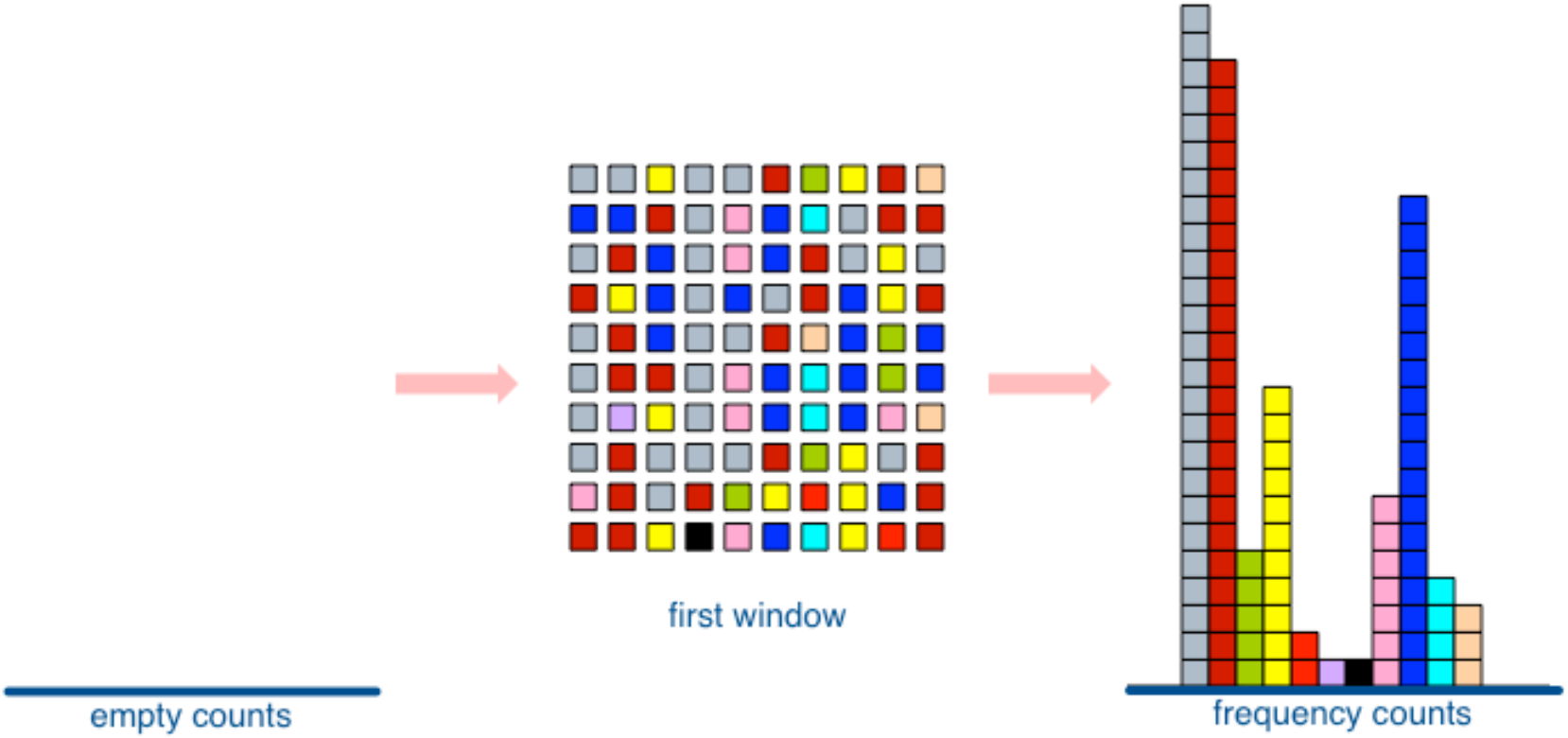
Source: Peter Bonz

Divide Into Windows…



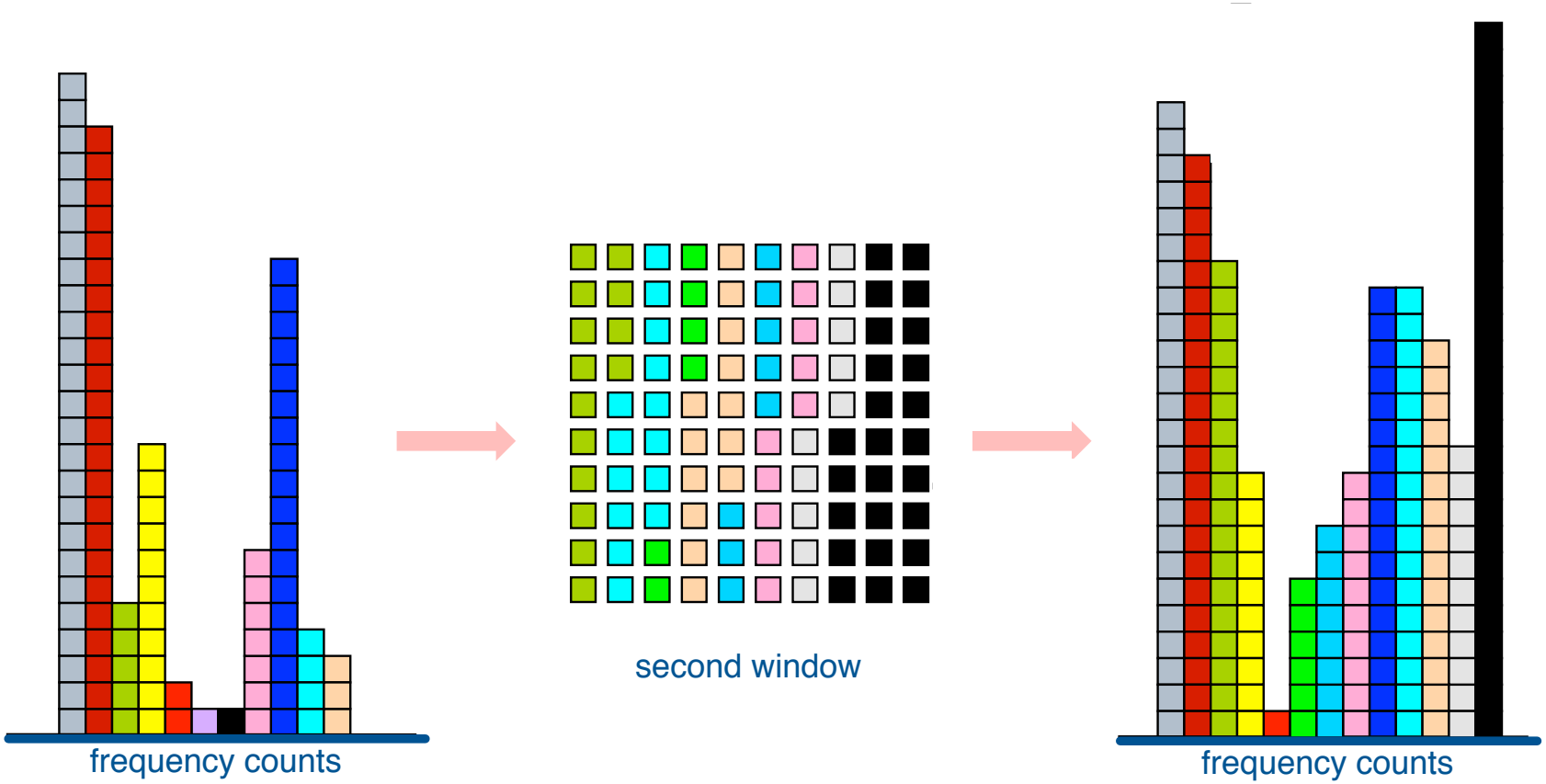
Source: Peter Bonz

First Window



Source: Peter Bonz

Second Window



Source: Peter Bonz

Window Counting

- What's the issue?
- What's the solution?

Lessons learned?
Solutions are approximate (or lossy)

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

General Strategies

- Sampling
- Hashing

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

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
  - …

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

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 discarded: 10/11 × 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 element is replaced: $s/(k+1) \times 1/s = 1/(k + 1)$
  - Probability existing element is not replaced: $k/(k + 1)$
  - Probability each element survives to $(k + 1)$th round:
    $(s/k) \times k/(k + 1) = s/(k + 1)$

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Hashing for Three Common Tasks

- Cardinality estimation — HashSet — HLL counter
  - What's the cardinality of set *S*?
  - How many unique visitors to this page?
- Set membership — HashSet — Bloom Filter
  - Is *x* a member of set *S*?
  - Has this user seen this ad before?
- Frequency estimation — HashMap — CMS
  - How many times have we observed *x*?
  - How many queries has this user issued?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

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 1
  - On expectation, 1/4 of the hash codes will start with 01
  - On expectation, 1/8 of the hash codes will start with 001
  - On expectation, 1/16 of the hash codes will start with 0001
  - …

How do we take advantage of this observation?

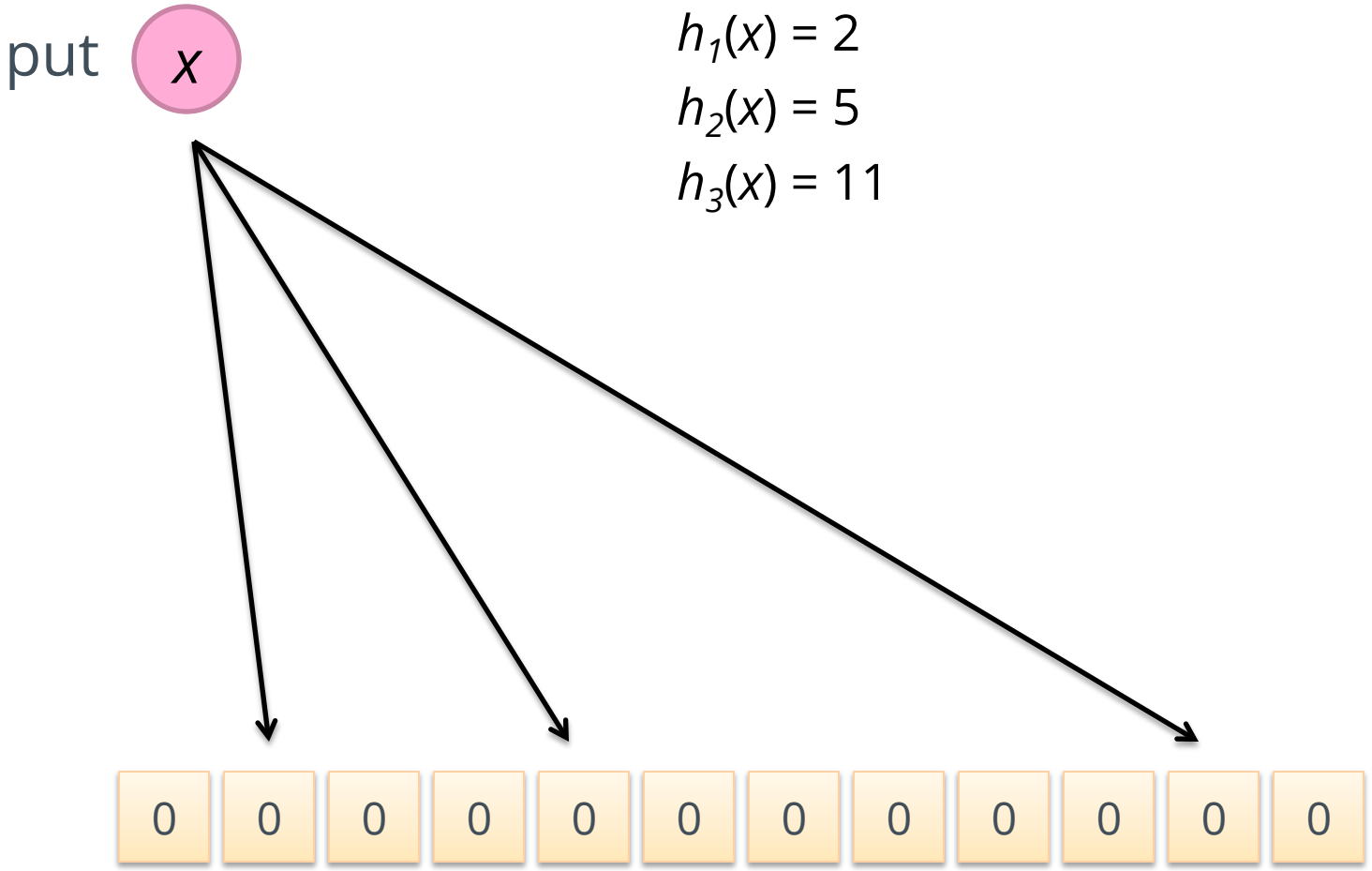
Bloom Filters

- 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: $h_1 \ldots h_k$

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Bloom Filters: put



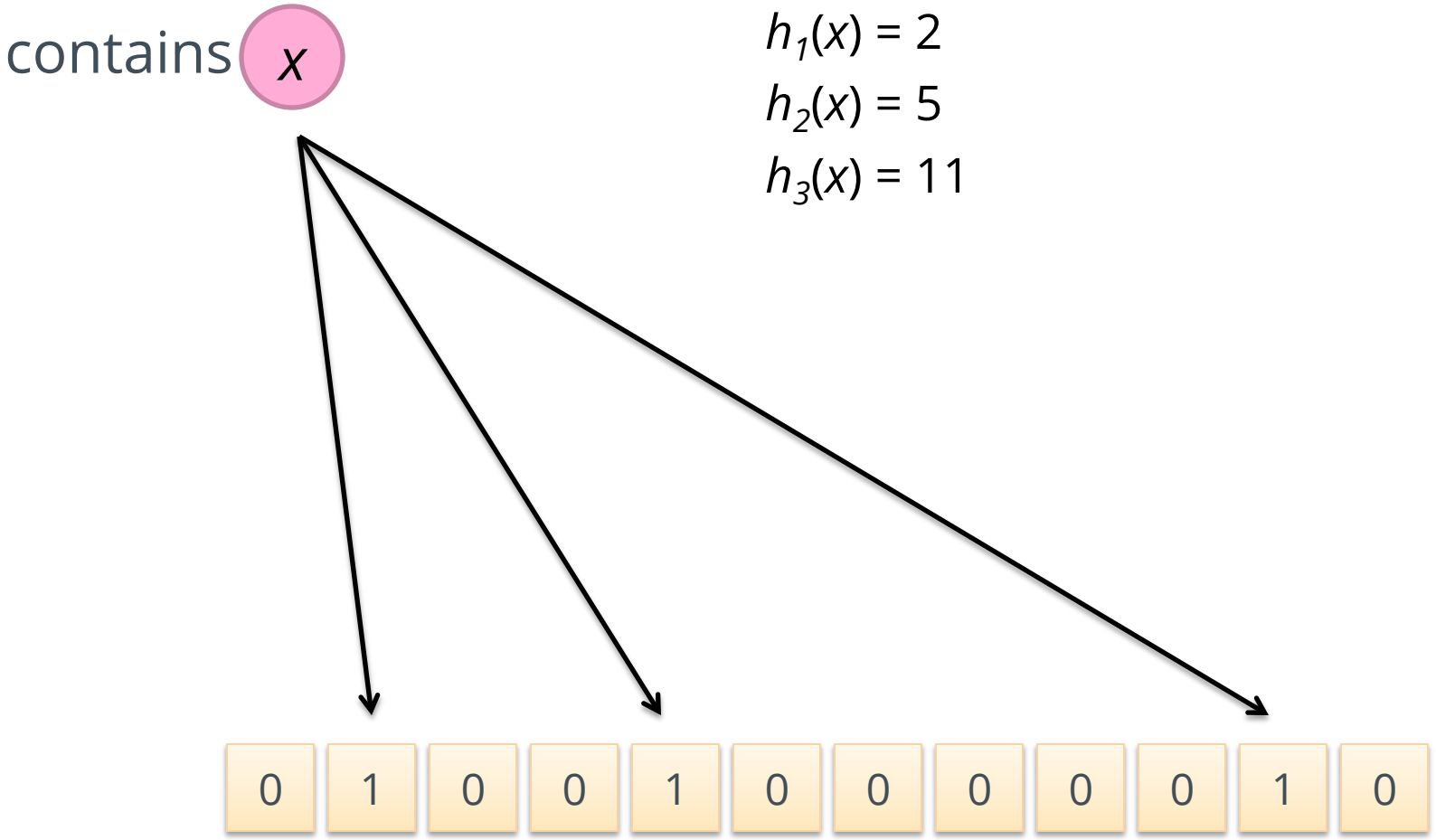
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Bloom Filters: put

put *x*

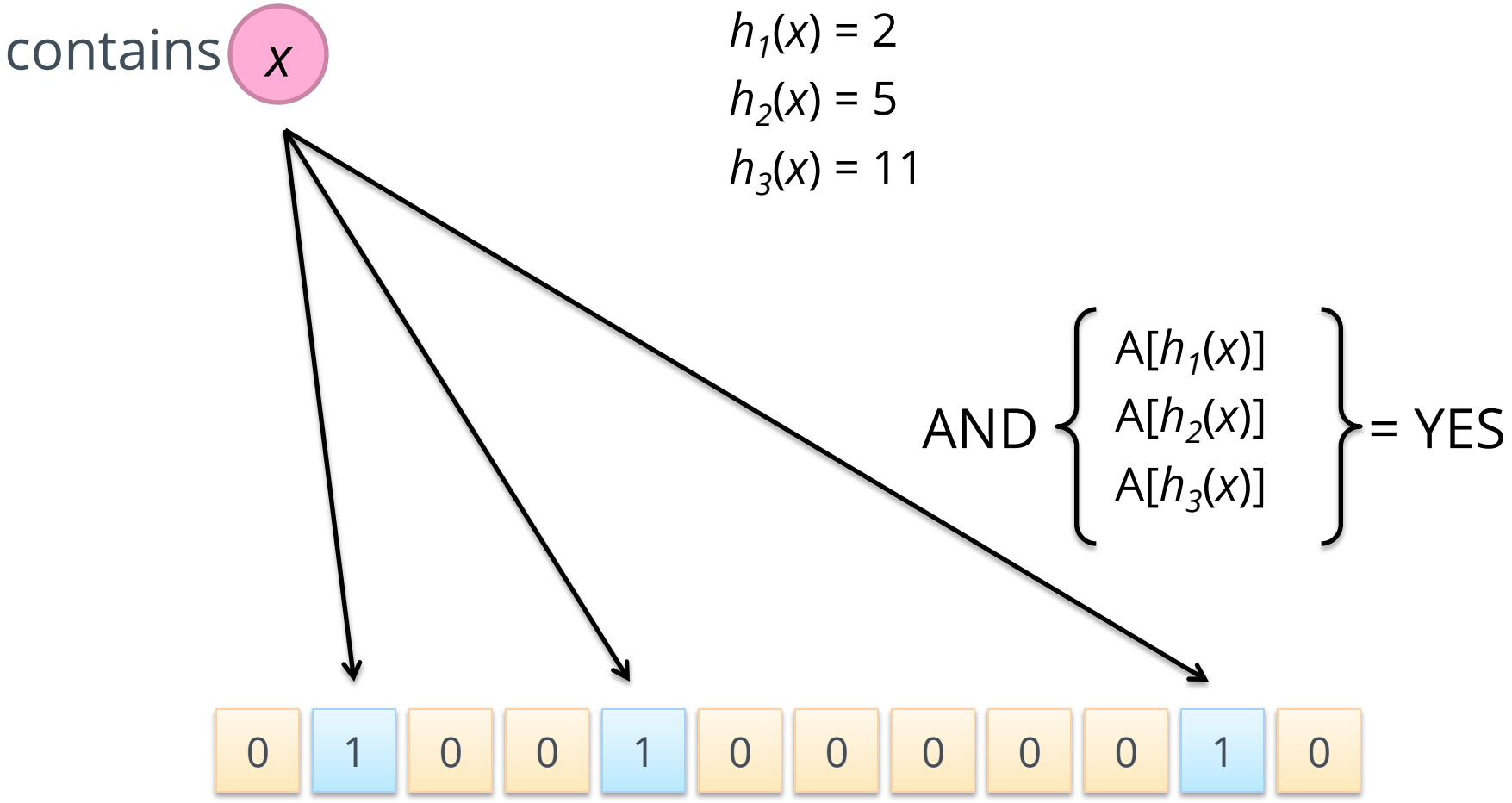
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Bloom Filters: contains



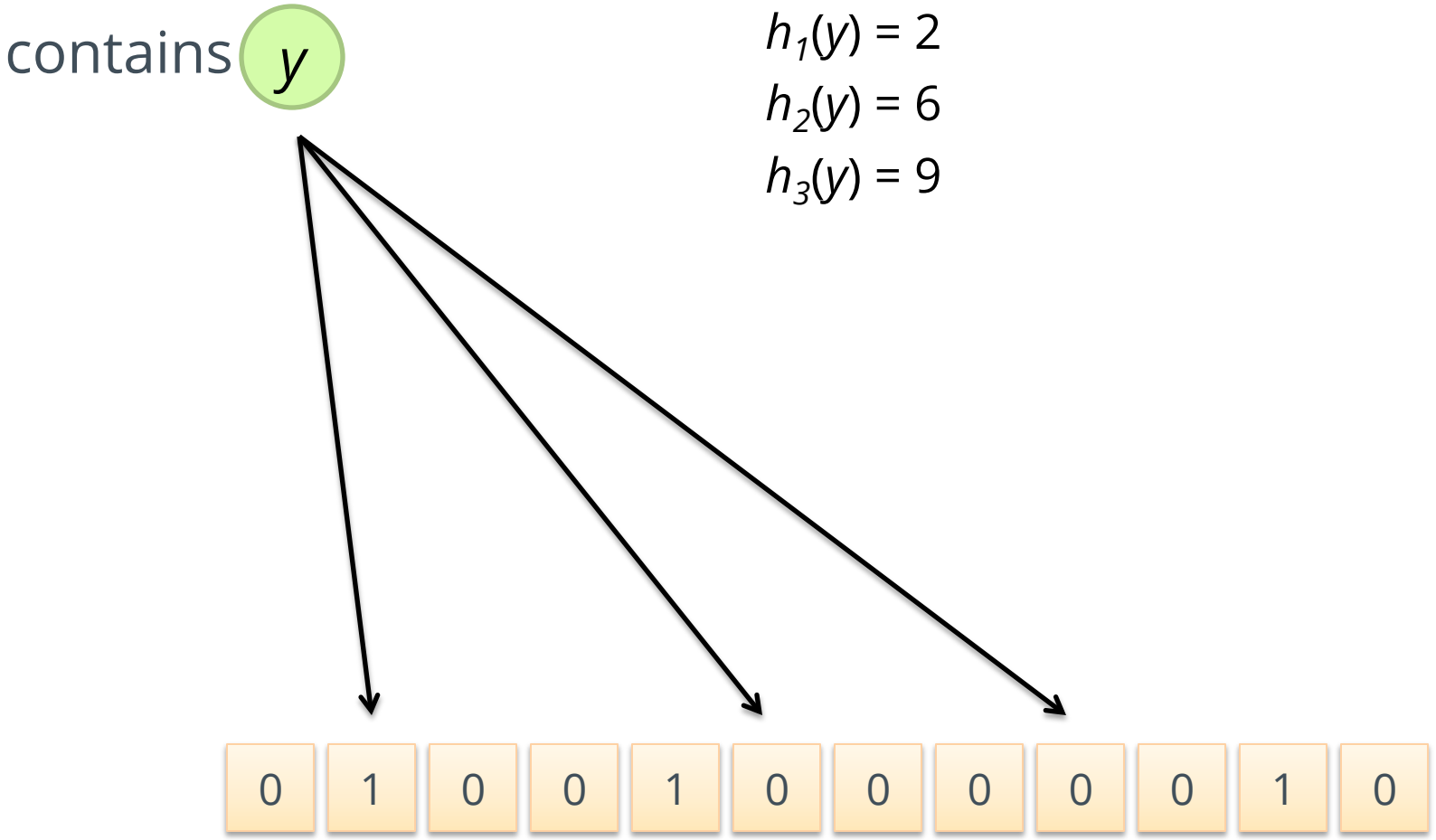
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Bloom Filters: contains



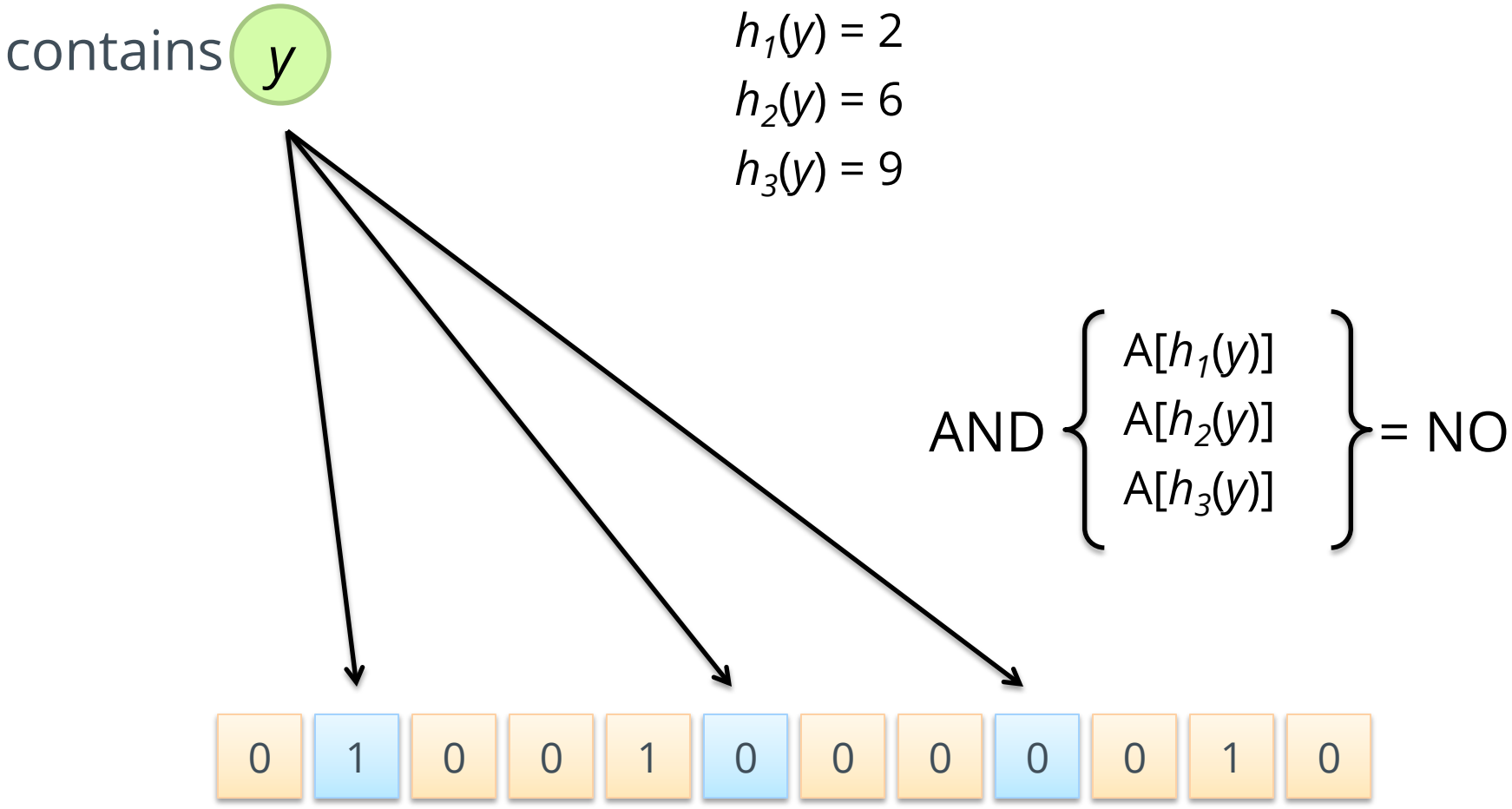
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Bloom Filters: contains



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Bloom Filters: contains



What's going on here?

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

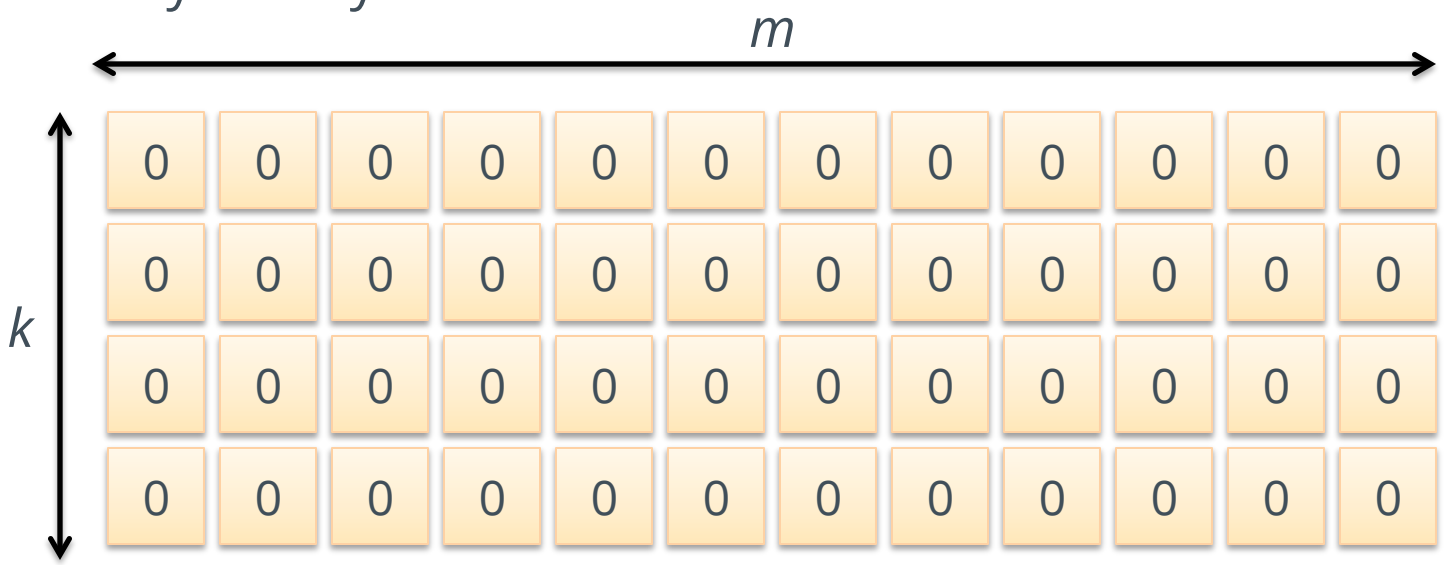
Bloom Filters

- 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*

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

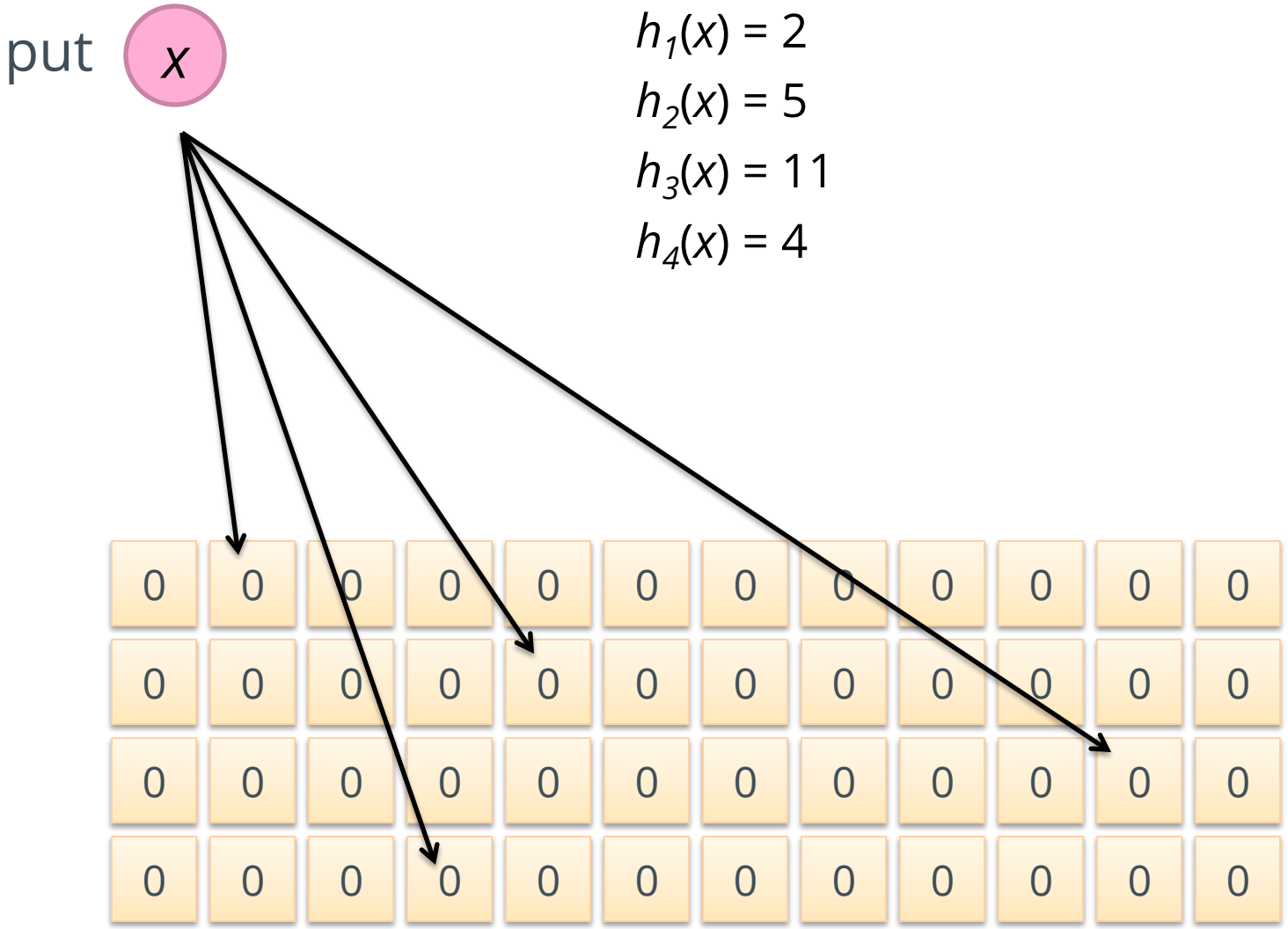
Count-Min Sketches

- Task: frequency estimation
  - put(x) → increment count of x by one
  - get(x) → returns the frequency of x
- Components
  - k hash functions: h_1 … h_k
  - m by k array of counters



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

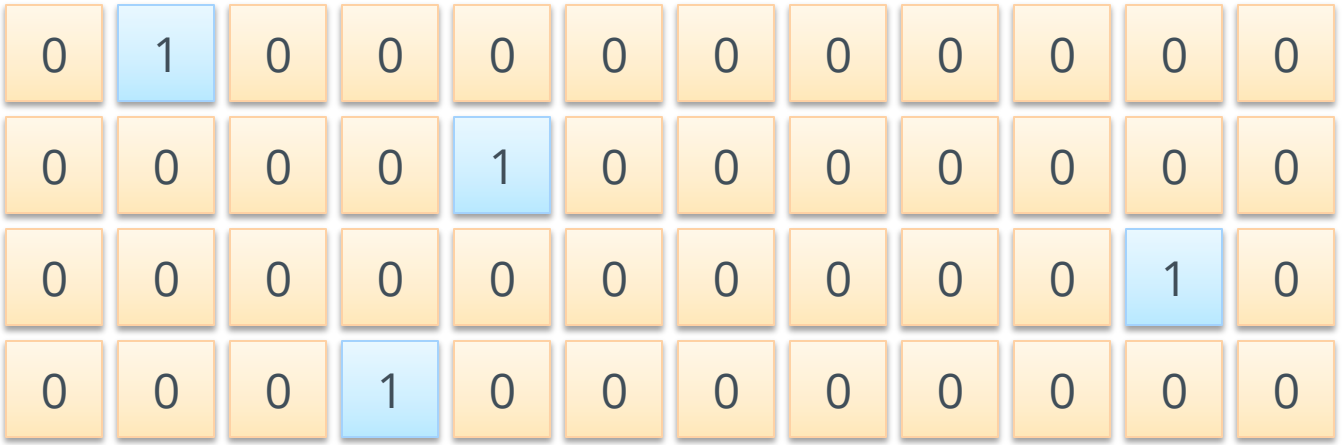
Count-Min Sketches: put



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

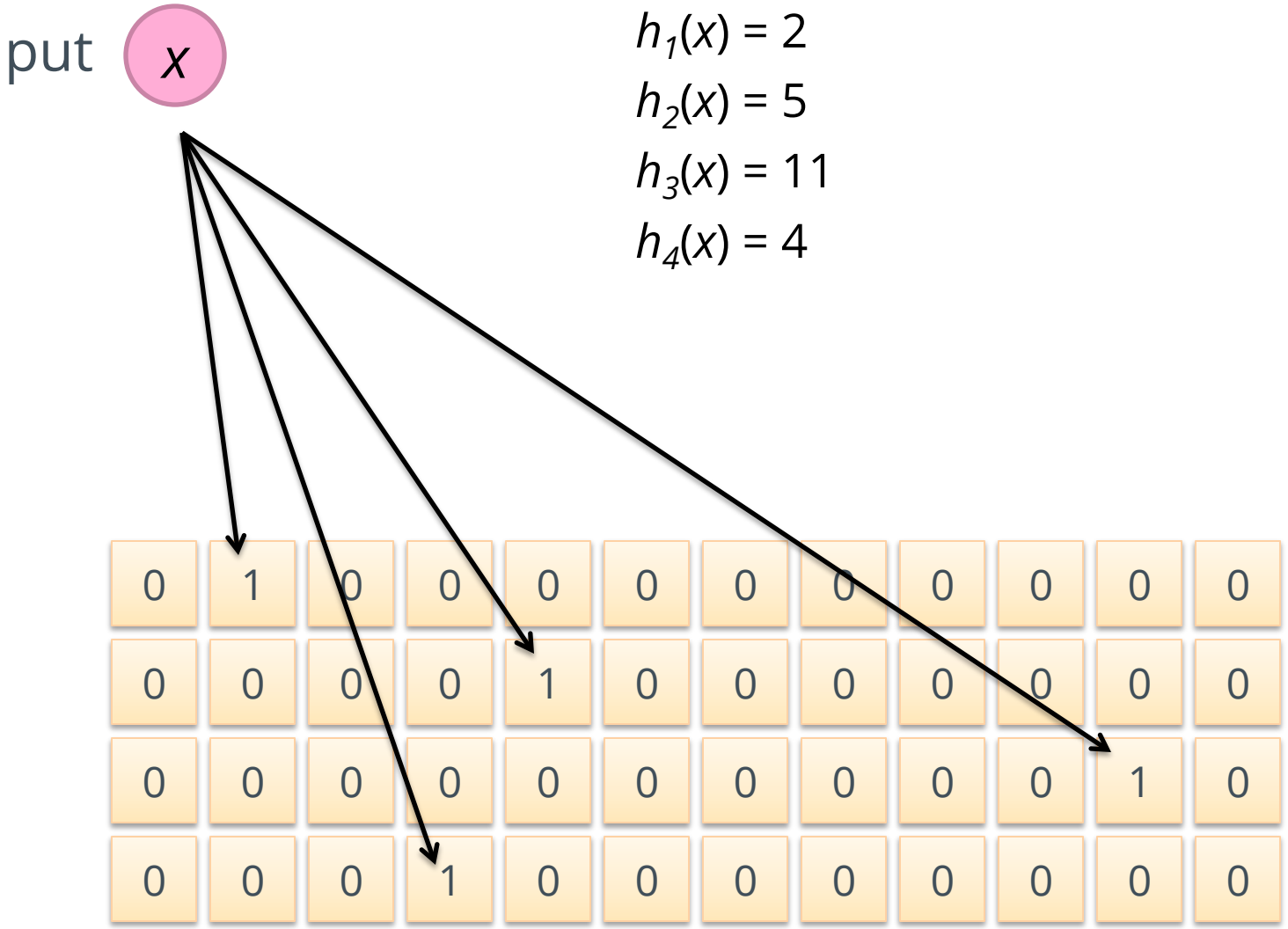
Count-Min Sketches: put

put *x*



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

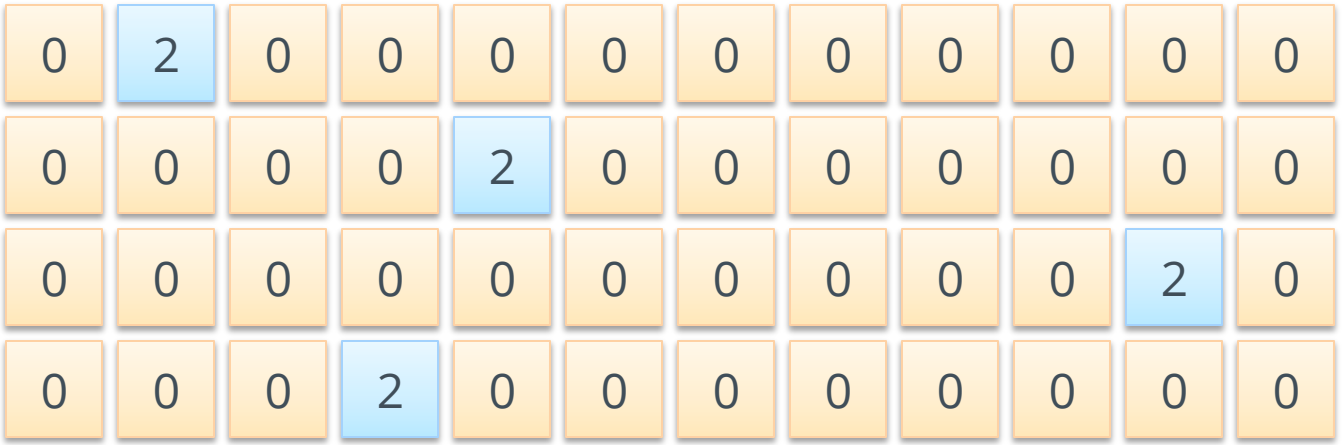
Count-Min Sketches: put



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

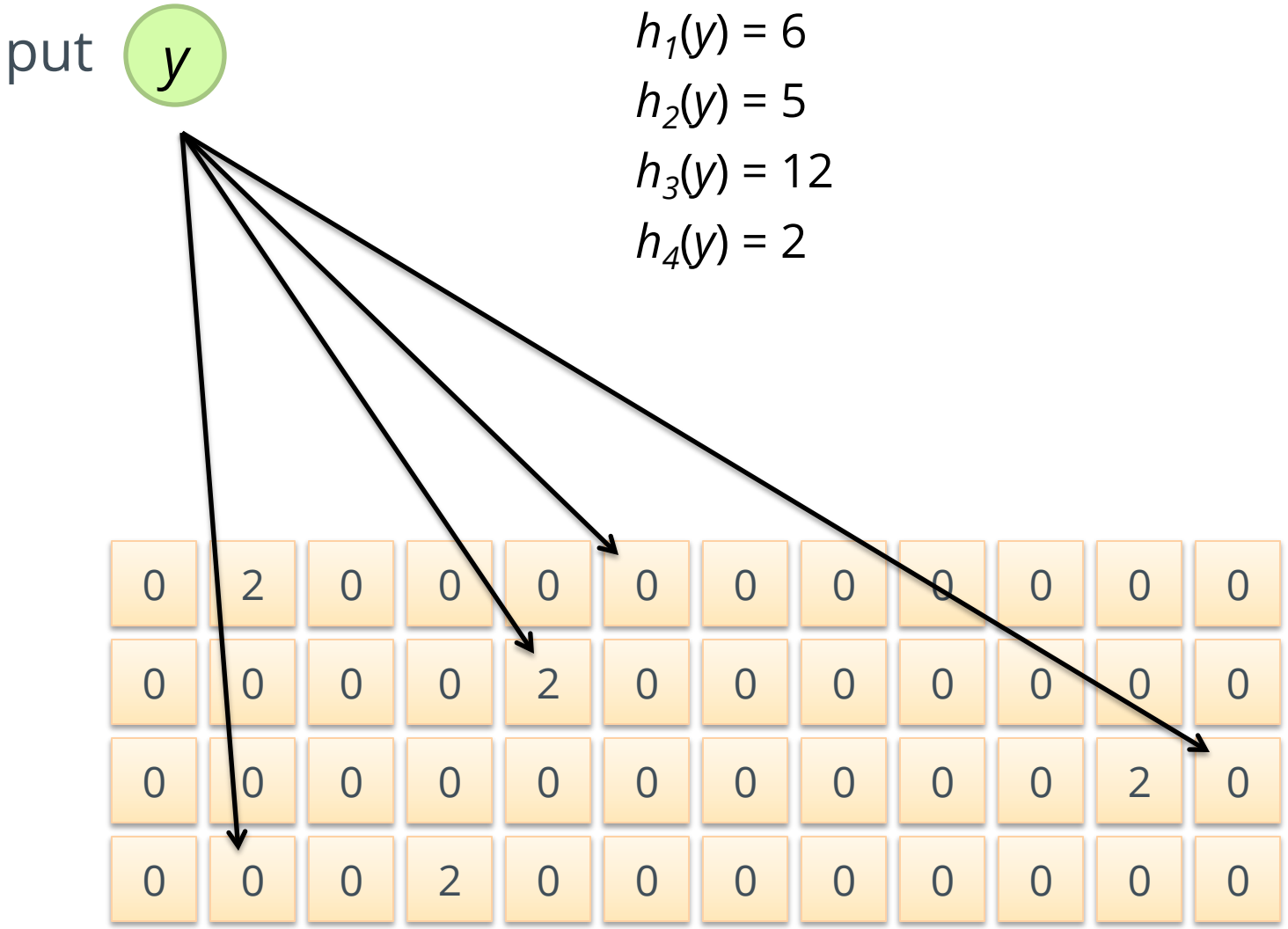
Count-Min Sketches: put

put *x*



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

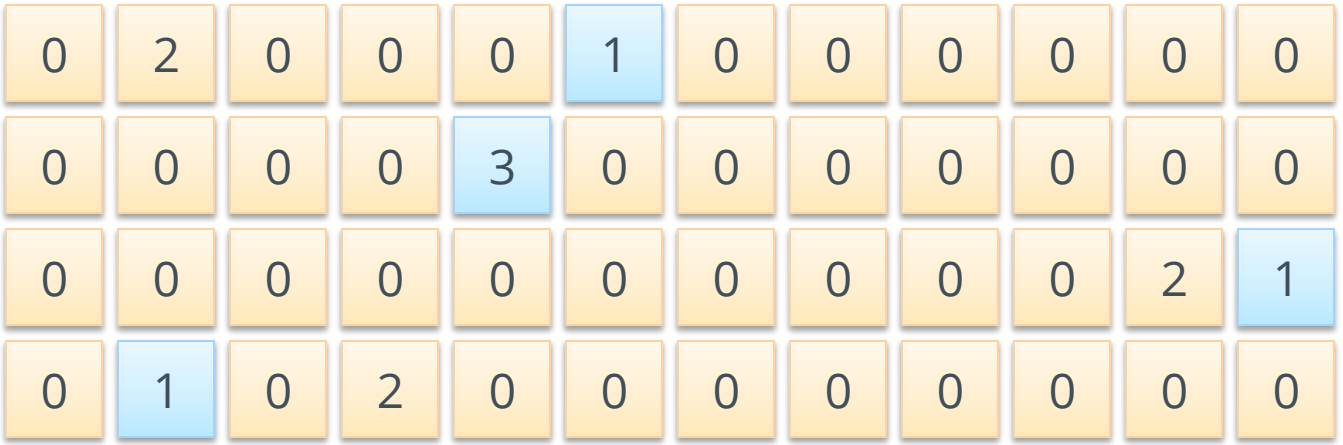
Count-Min Sketches: put



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

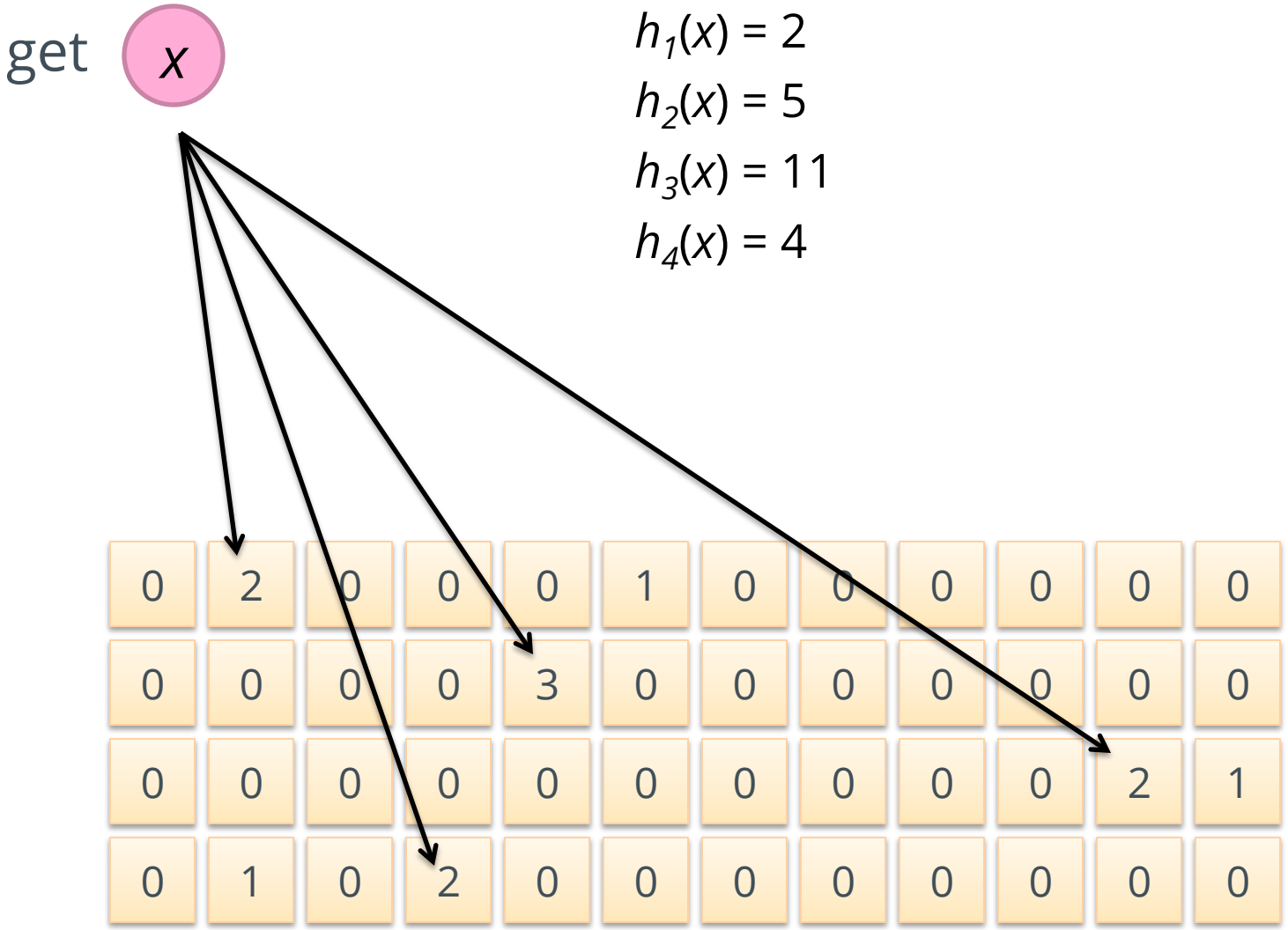
Count-Min Sketches: put

put *y*



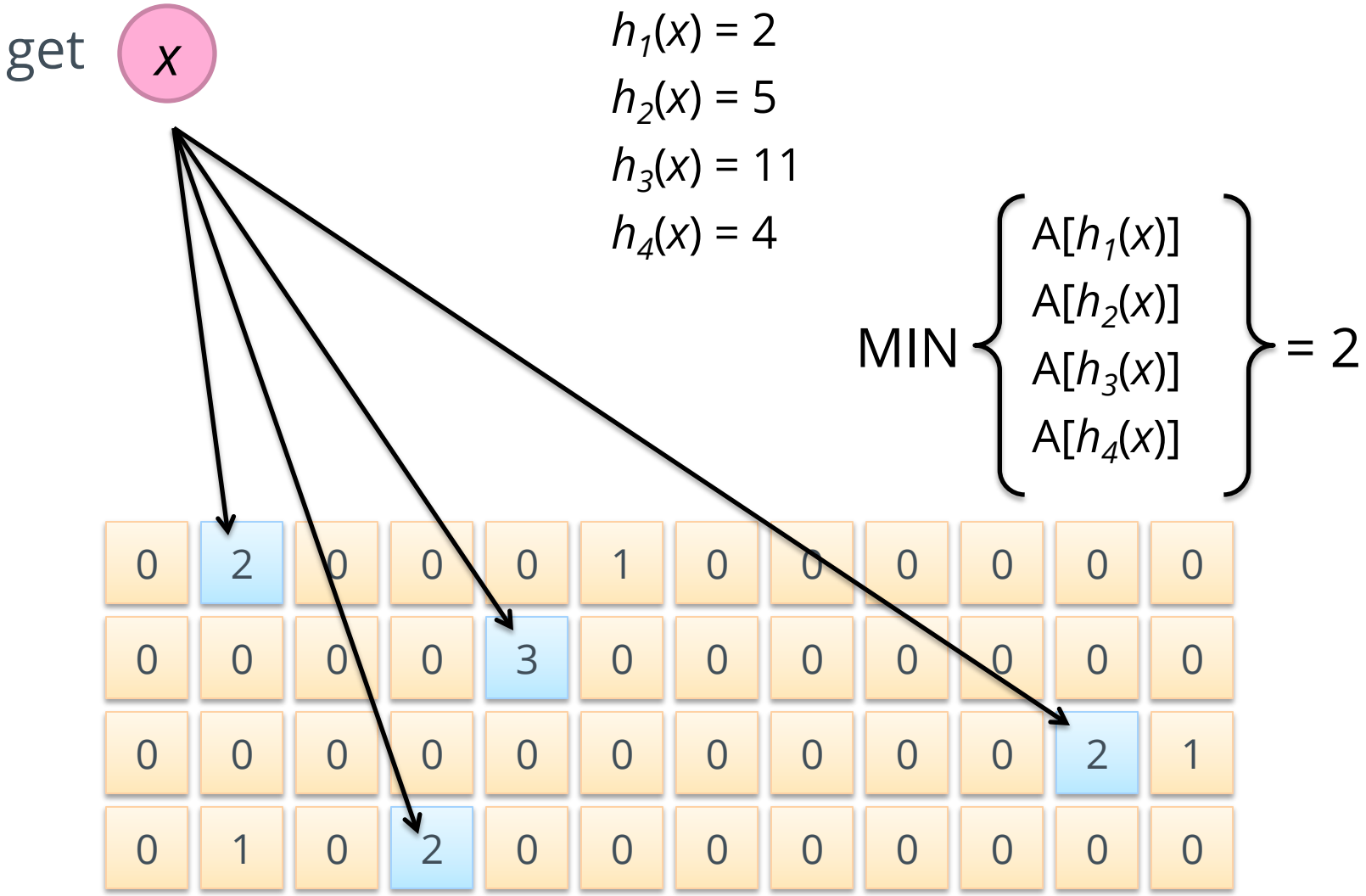
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Count-Min Sketches: get



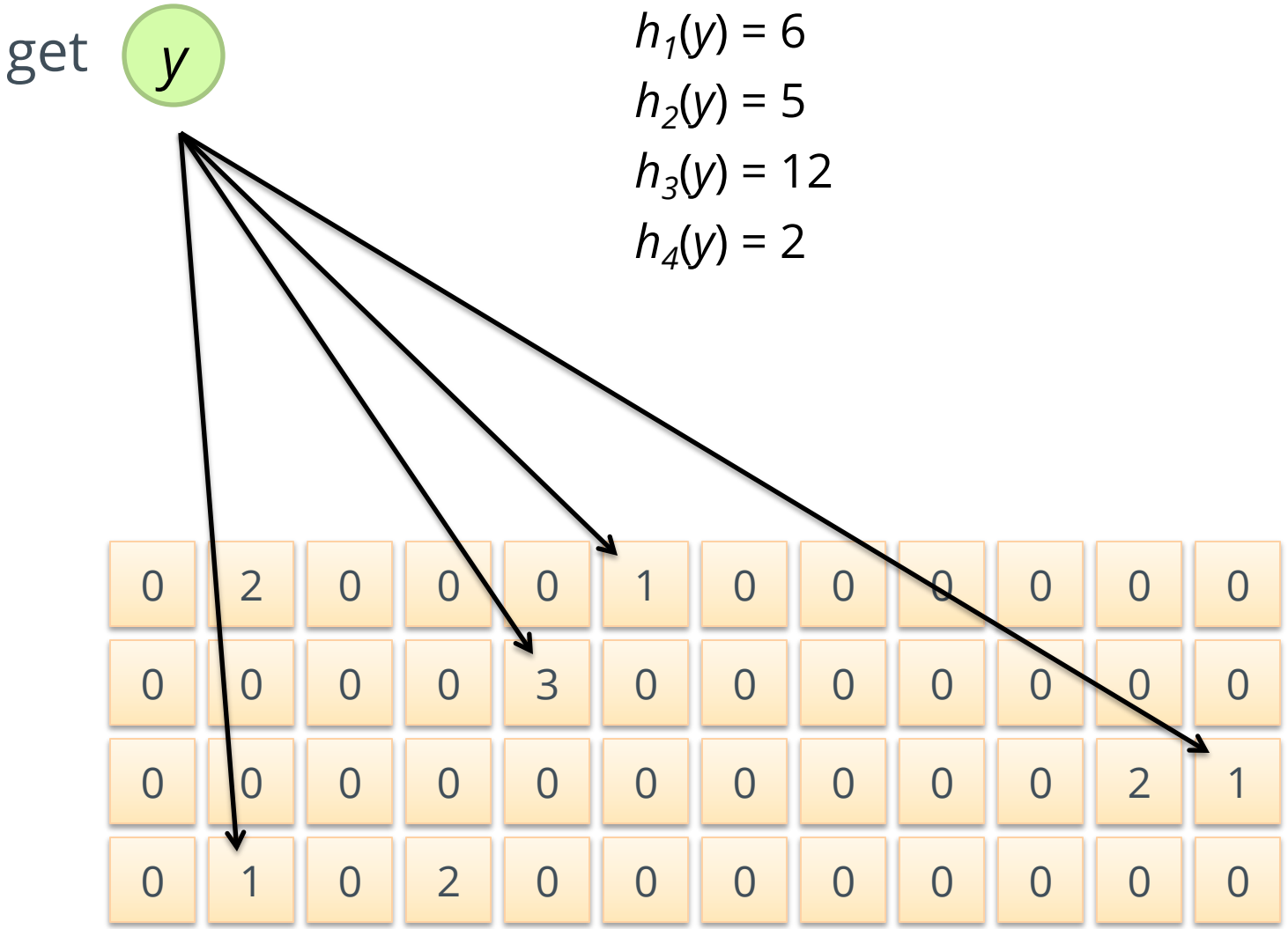
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Count-Min Sketches: get



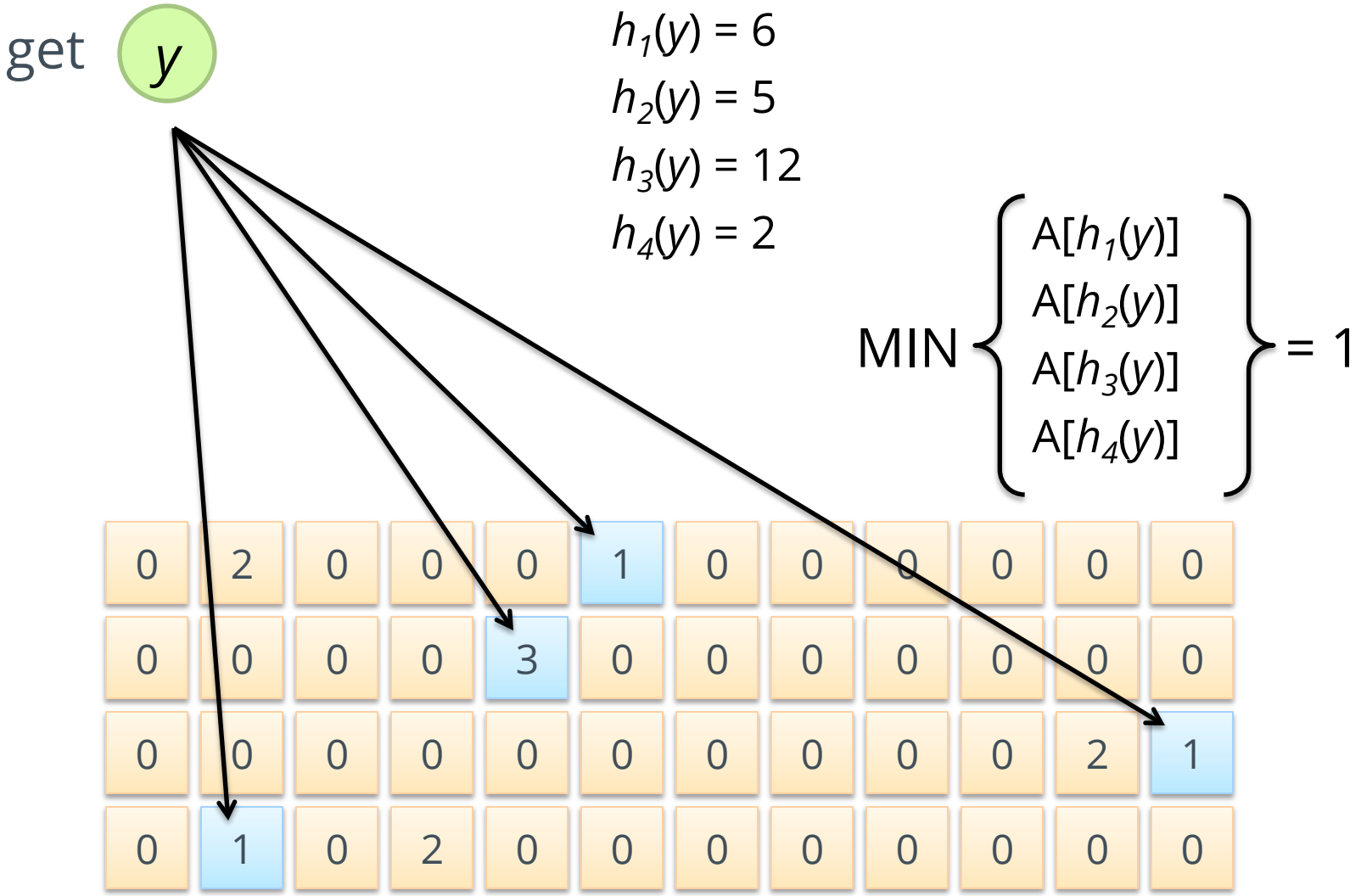
Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Count-Min Sketches: get



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Count-Min Sketches: get



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Count-Min Sketches

- Error properties:
  - 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*, number of hash functions *k*, size of counters

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Minhash

Source: www.flickr.com/photos/rheinitz/6158837748/

Near-Duplicate Detection

- What's the source of the problem?
  - Mirror pages (legit)
  - Spam farms (non-legit)
  - Additional complications (e.g., nav bars)
- Naïve algorithm:
  - Compute cryptographic hash for webpage (e.g., MD5)
  - Insert hash values into a big hash table
  - Compute hash for new webpage: collision implies duplicate
- What's the issue?
- Intuition:
  - Hash function needs to be tolerant of minor differences
  - High similarity implies higher probability of hash collision

Minhash

- Seminal algorithm for near-duplicate detection of webpages
  - Used by AltaVista
  - For details see Broder et al. (1997)
- Setup:
  - Documents (HTML pages) represented by shingles (*n*-grams)
  - Jaccard similarity: dups are pairs with high similarity

Representation

- Sets:
  - A = {e_1, e_3, e_7}
  - B = {e_3, e_5, e_7}
- Can be equivalently expressed as matrices:

Element	A	B
e_1	1	0
e_2	0	0
e_3	1	1
e_4	0	0
e_5	0	1
e_6	0	0
e_7	1	1

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Jaccard Similarity

Element	A	B
e_1	1	0
e_2	0	0
e_3	1	1
e_4	0	0
e_5	0	1
e_6	0	0
e_7	1	1

Let:

M_{00} = # rows where both elements are 0

M_{11} = # rows where both elements are 1

M_{01} = # rows where A=0, B=1

M_{10} = # rows where A=1, B=0

$$\mathrm{J}(A, B) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Minhash

- Computing minhash
  - Start with the matrix representation of the set
  - Randomly permute the rows of the matrix
  - minhash is the first row with a “one”
- Example:

h(A) = e_3 h(B) = e_5

Element	A	B
e_1	1	0
e_2	0	0
e_3	1	1
e_4	0	0
e_5	0	1
e_6	0	0
e_7	1	1

Element	A	B
e_6	0	0
e_2	0	0
e_5	0	1
e_3	1	1
e_7	1	1
e_4	0	0
e_1	1	0

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Minhash and Jaccard

Element	A	B	
e_6	0	0	M_{00}
e_2	0	0	M_{00}
e_5	0	1	M_{01}
e_3	1	1	M_{11}
e_7	1	1	M_{11}
e_4	0	0	M_{00}
e_1	1	0	M_{10}

$$P[h(A) = h(B)] = \mathrm{J}(A, B)$$

$$\frac{M_{11}}{M_{01} + M_{10} + M_{11}} \qquad \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

To Permute or Not to Permute?

- Permutations are expensive
- Interpret the hash value as the permutation
- Only need to keep track of the minimum hash value
  - Can keep track of multiple minhash values at once

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Extracting Similar Pairs (LSH)

- We know: $P[h(A) = h(B)] = \mathrm{J}(A, B)$
- Task: discover all pairs with similarity greater than *s*
- Algorithm:
  - For each object, compute its minhash value
  - Group objects by their hash values
  - Output all pairs within each group
- Analysis:
  - Probability of hit is *s*

Two Minhash Signatures

- Task: discover all pairs with similarity greater than s
- Algorithm:
  - For each object, compute two minhash values and concatenate together into a signature
  - Group objects by their signatures
  - Output all pairs within each group
- Analysis:
  - Probability of hit is s^2

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

k Minhash Signatures

- Task: discover all pairs with similarity greater than *s*
- Algorithm:
  - For each object, compute *k* minhash values and concatenate together into a signature
  - Group objects by their signatures
  - Output all pairs within each group
- Analysis:
  - Probability of hit is s^k

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

n different *k* Minhash Signatures

- Task: discover all pairs with similarity greater than *s*
- Algorithm:
  - For each object, compute *n* sets *k* minhash values
  - For each set, concatenate *k* minhash values together
  - Within each set:
    - Group objects by their signatures
    - Output all pairs within each group
  - De-dup pairs
- Analysis:
  - Probability of misses in all bands is $(1 - s^k)^n$
  - Probability of a hit in at least one band is $1 - (1 - s^k)^n$

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Stream Processing Architectures

Source: Wikipedia (River)

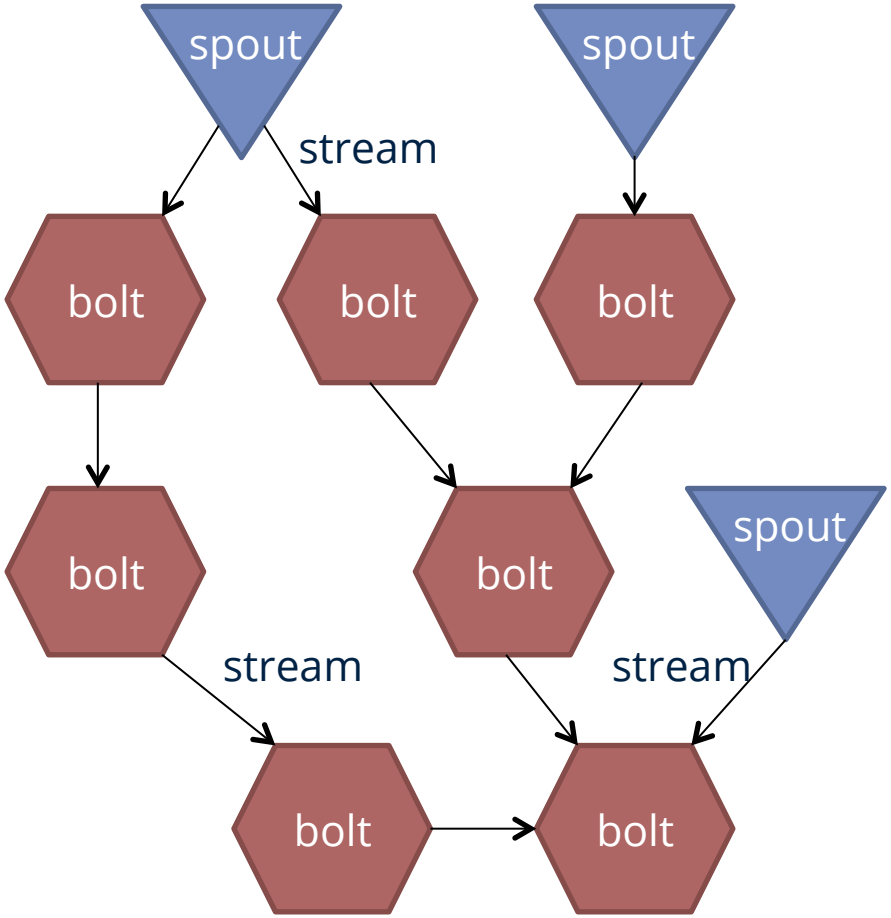
Storm

- Storm topologies = “job”
  - Once started, runs continuously until killed
- A Storm topology is a computation graph
  - Graph contains nodes and edges
  - Nodes hold processing logic (i.e., transformation over its input)
  - Directed edges indicate communication between nodes

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Streams, Spouts, and Bolts

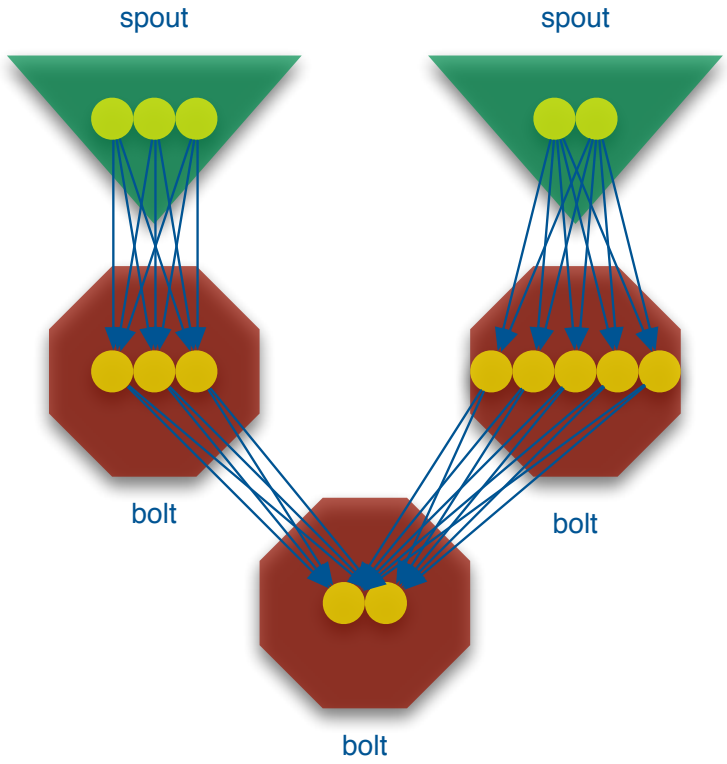
- Streams
  - The basic collection abstraction: an unbounded sequence of tuples
  - Streams are transformed by the processing elements of a topology
- Spouts
  - Stream generators
  - May propagate a single stream to multiple consumers
- Bolts
  - Subscribe to streams
  - Streams transformers
  - Process incoming streams and produce new ones



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Stream Groupings

- Bolts are executed by multiple workers in parallel
- When a bolt emits a tuple, where should it go?
- Stream groupings:
  - Shuffle grouping: round-robin
  - Field grouping: based on data value



Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

Storm: Example

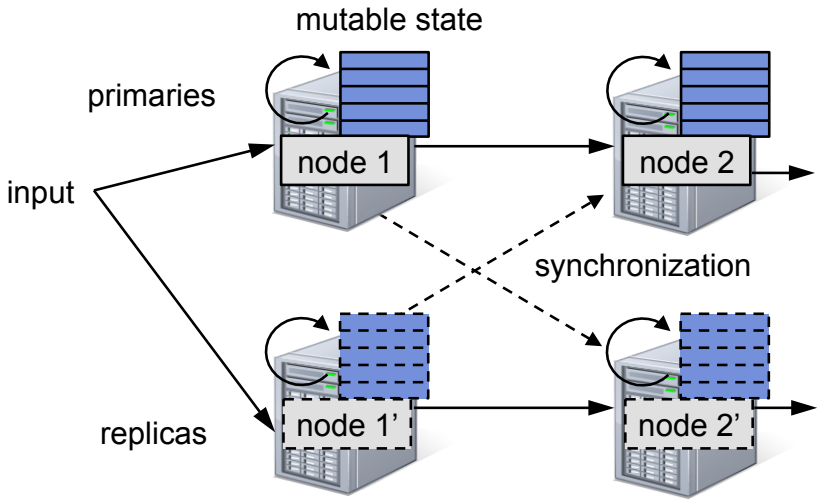
```
// instantiate a new topology
TopologyBuilder builder = new TopologyBuilder();

// set up a new spout with five tasks
builder.setSpout("spout", new RandomSentenceSpout(), 5);

// the sentence splitter bolt with eight tasks
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("spout"); // shuffle grouping for the ouput

// word counter with twelve tasks
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word")); // field grouping

// new configuration
Config conf = new Config();

// set the number of workers for the topology; the 5x8x12=480 tasks
// will be allocated round-robin to the three workers, each task
// running as a separate thread
conf.setNumWorkers(3);

// submit the topology to the cluster
StormSubmitter.submitTopology("word-count", conf, builder.createTopology());
```

Source: Lin et al. Big Data Infrastructure, UMD Spring 2015.

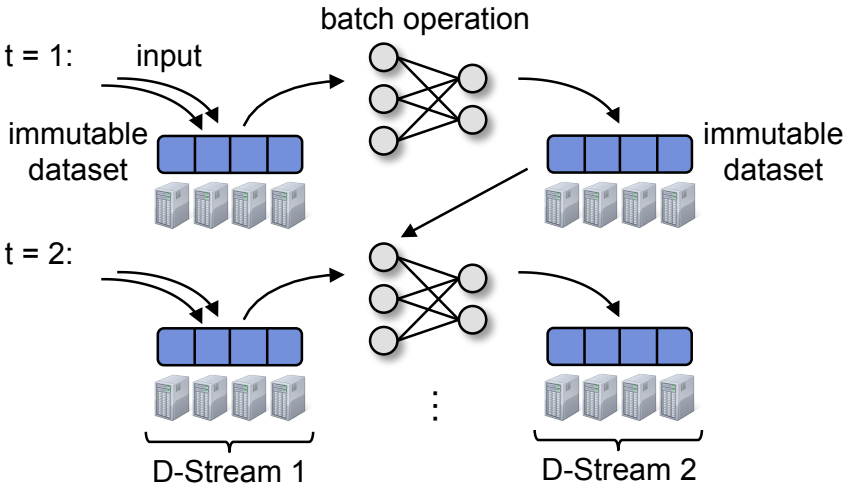
Spark Streaming

Discretized stream processing:

Run a streaming computation as a series of very small, deterministic batch jobs



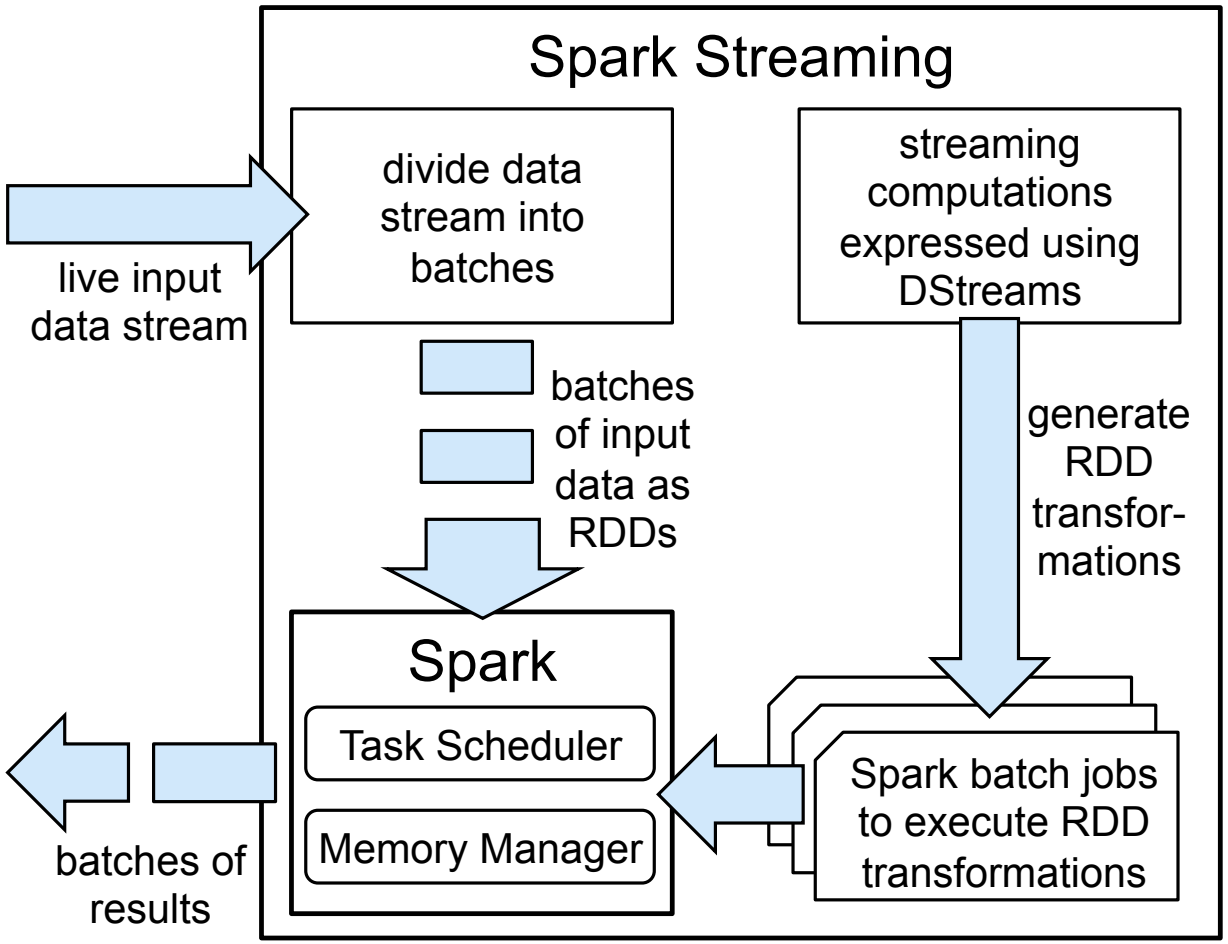
Continuous Operator Model



Discretized Streams

Source: Zaharia et al. (SOSP 2013)

Spark and Spark Streaming



Source: Zaharia et al. (SOSP 2013)

The Dataflow Model

Key Ideas

- Windowing
  - Fixed windows
  - Sliding windows
  - Sessions
- Time domains
  - Event time
  - Processing time
- Triggers

Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Dataflow API

- Easily build pipelines with choice of windowing, time domain, trigger
- Independent of execution engine
  - Batch, micro-batch, or streaming

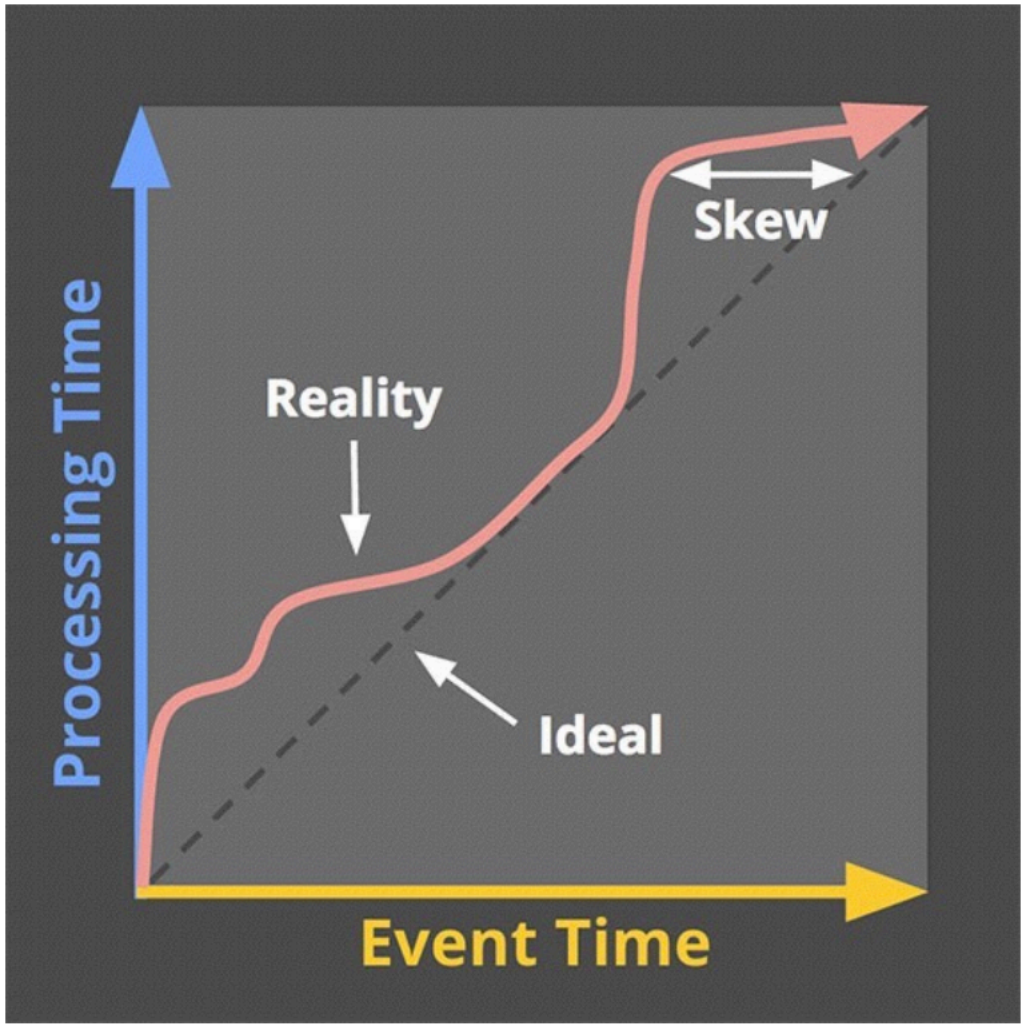
Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Time Domains

- For many applications, windows should be based on “event time” (when the events actually occur)
  - Example: billing advertisers
- Lag, partitions, etc. might cause an event to be processed later than its event time
  - Processing time

Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

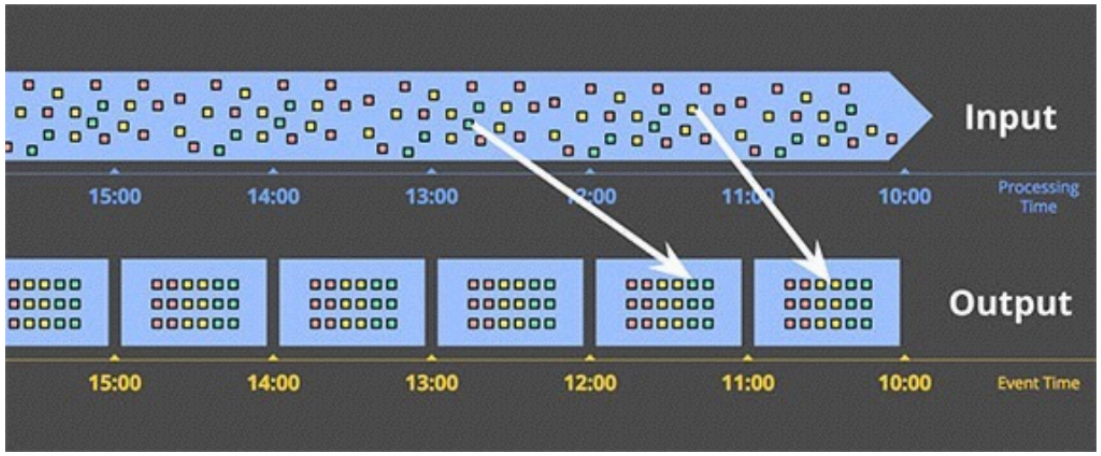
Challenge: Time Skew



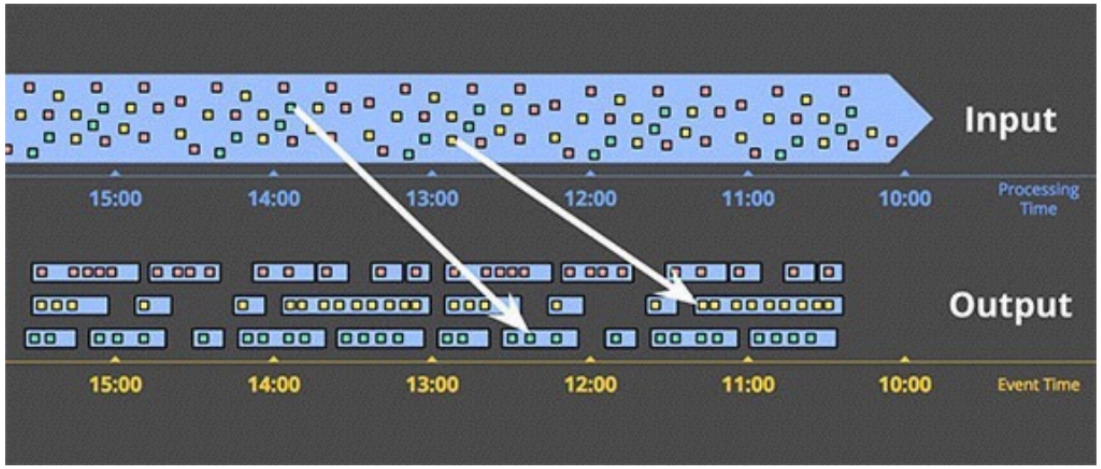
Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Goal: Event-time windows

Fixed windows



Session windows



Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Challenge: Completion

- With event times, how does the system know if it has received all of the data in a window?
- Example: phones might watch YouTube videos (and ads) offline

Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Watermarks

- Heuristics that tell the system when it is likely to have received most of the data in a window
- Based on global progress metrics
- Watermarks are insufficient:
  - Late data might arrive behind the watermark
  - Watermark might be too slow due to one late datum and increase latency for the whole system

Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Incremental Processing

- Difficult to get the single best result from a window
- Instead, let windows produce multiple results (improving incrementally over time)

Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Triggers

- Specify when to output window results
  - At watermark
  - Percentile watermark
  - Every minute, etc.
- Specify how to output results
  - Discard previous window
  - Accumulate
  - Accumulate and retract

Source: Zaharia et al. 6.S897: Large-Scale Systems, 2015.

Questions?