UNIVERSITY OF WATERLOO

Big Data Infrastructure

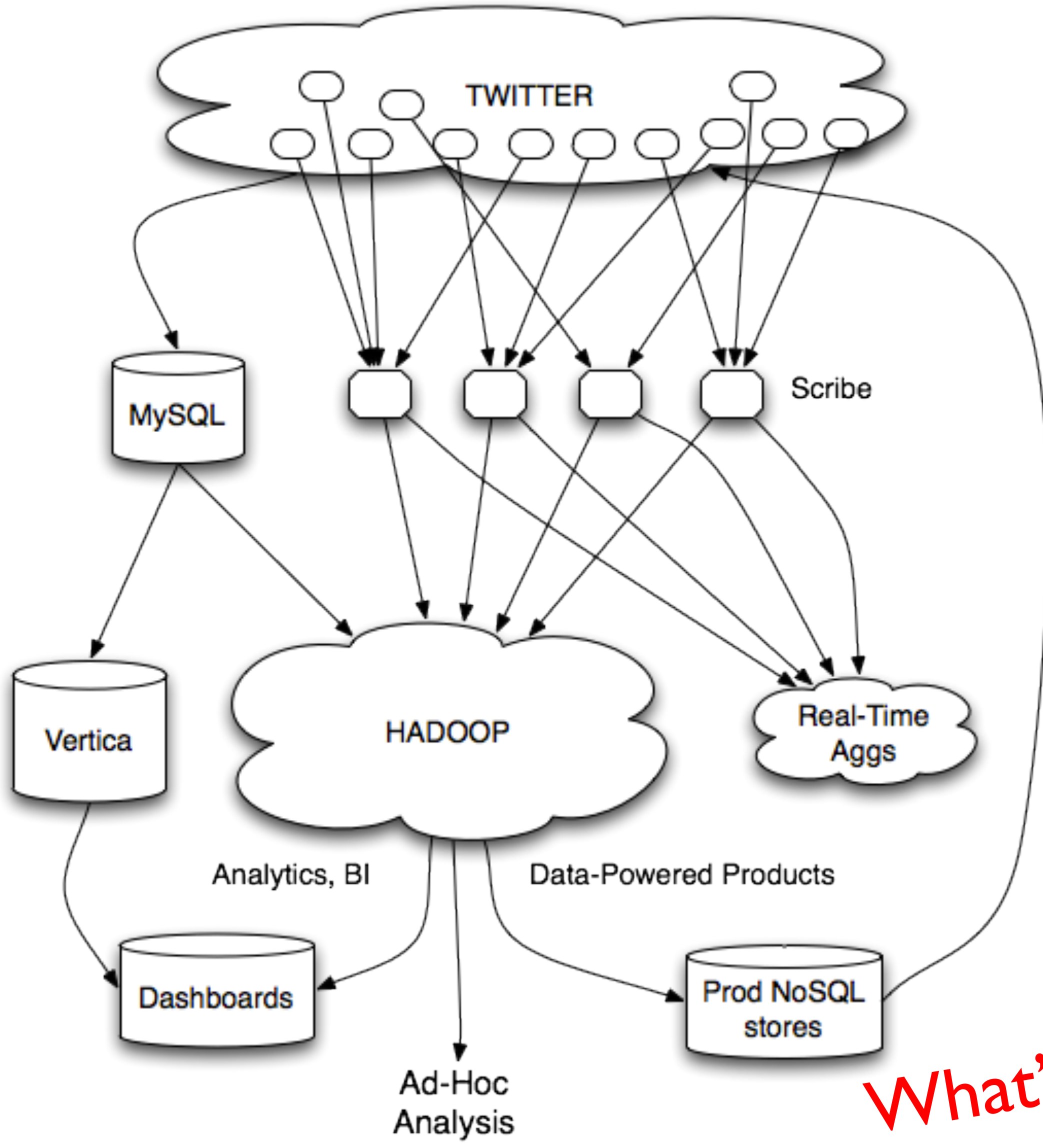
CS 489/698 Big Data Infrastructure (Winter 2016)

Week 12: Real-Time Data Analytics (2/2)

March 31, 2016

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2016w/



What's the issue?

Twitter's data warehousing architecture

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?

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

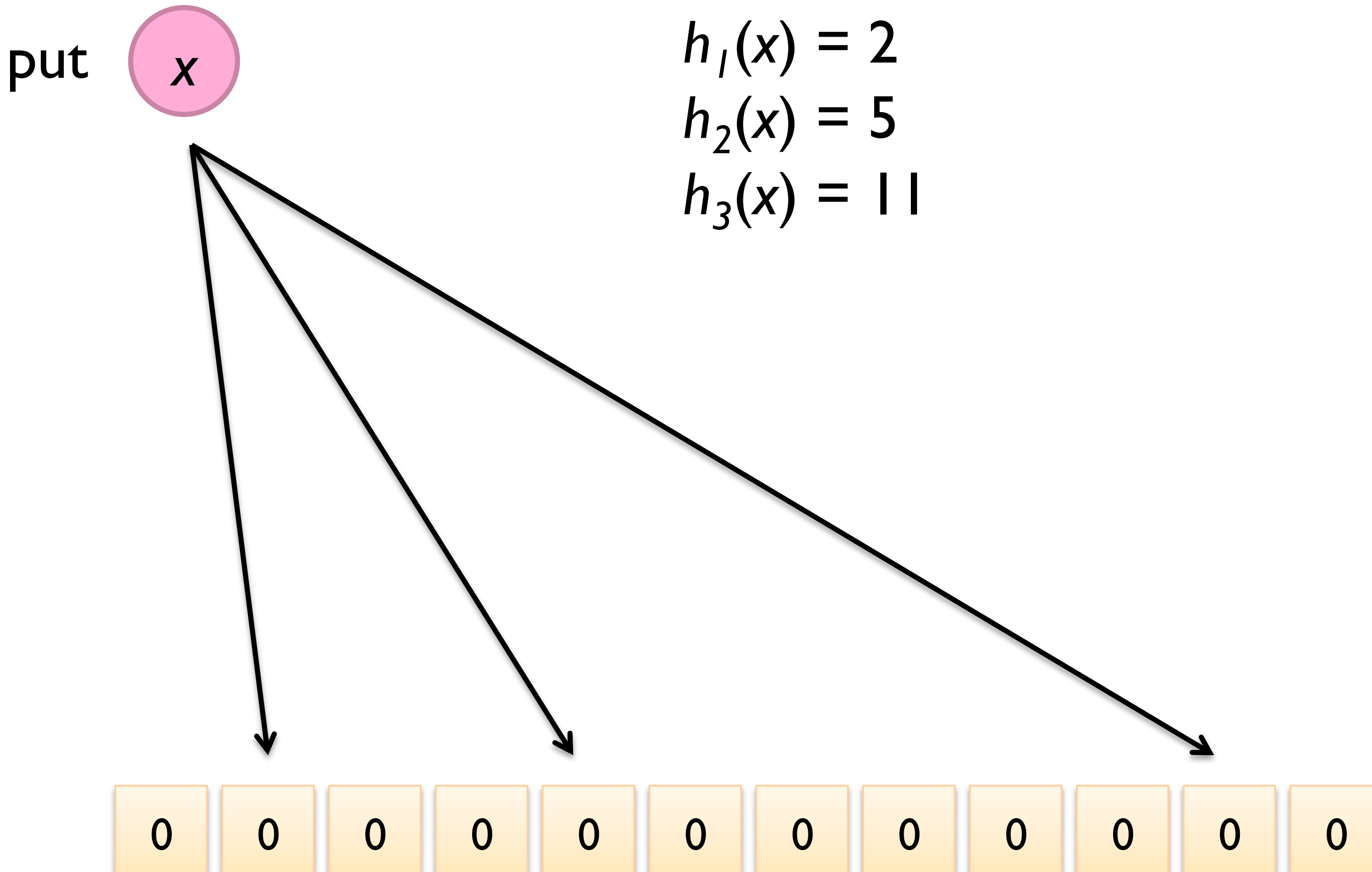
- Task: keep track of set membership
  - put(x) → insert x into the set
  - contains(x) → yes if x is a member of the set
- Components
  - m-bit bit vector

0	0	0	0	0	0	0	0	0	0	0	0

  - k hash functions: $h_1 \ldots h_k$

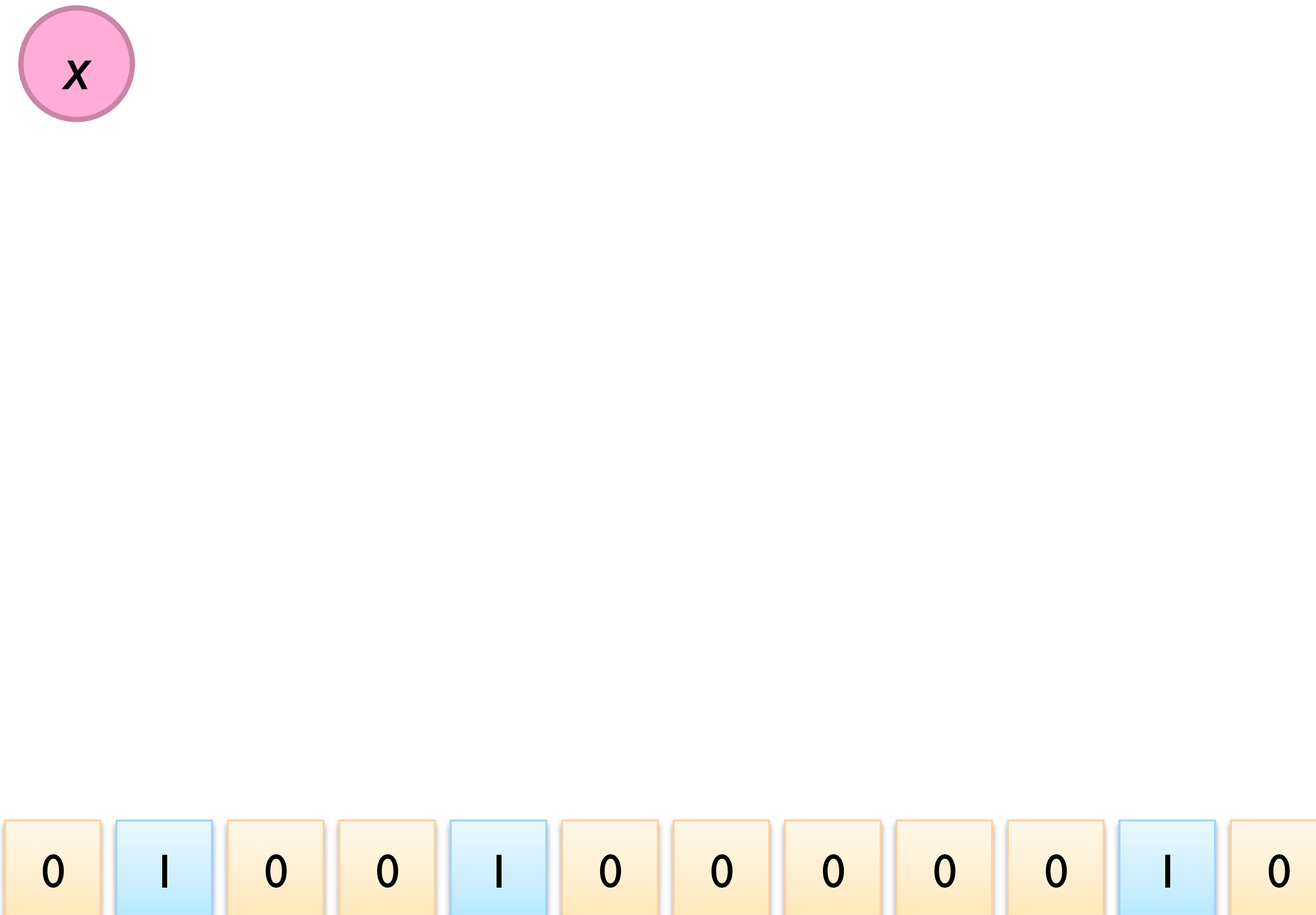
Bloom Filters: put

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



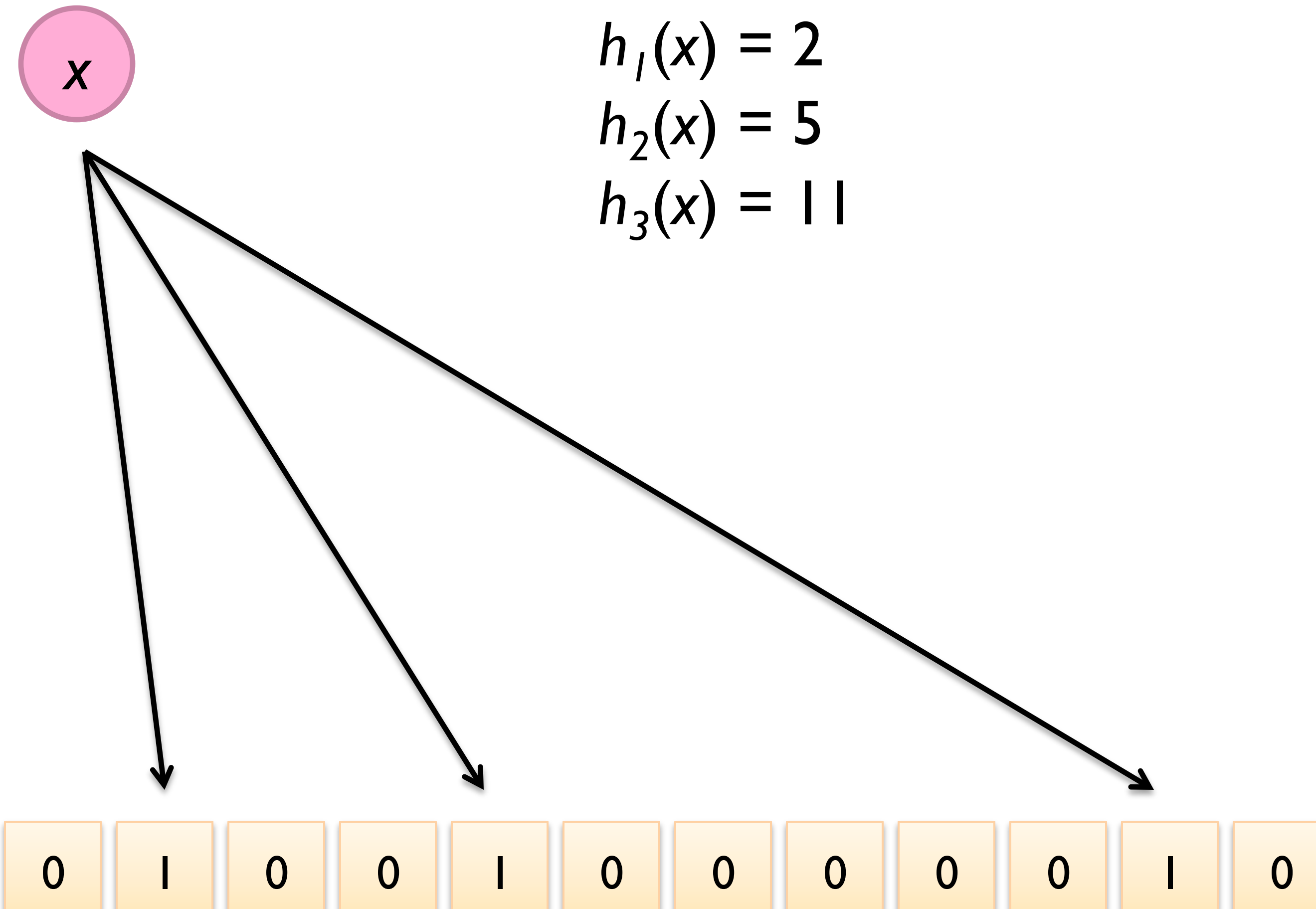
Bloom Filters: put

put



Bloom Filters: contains

contains



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

Bloom Filters: contains

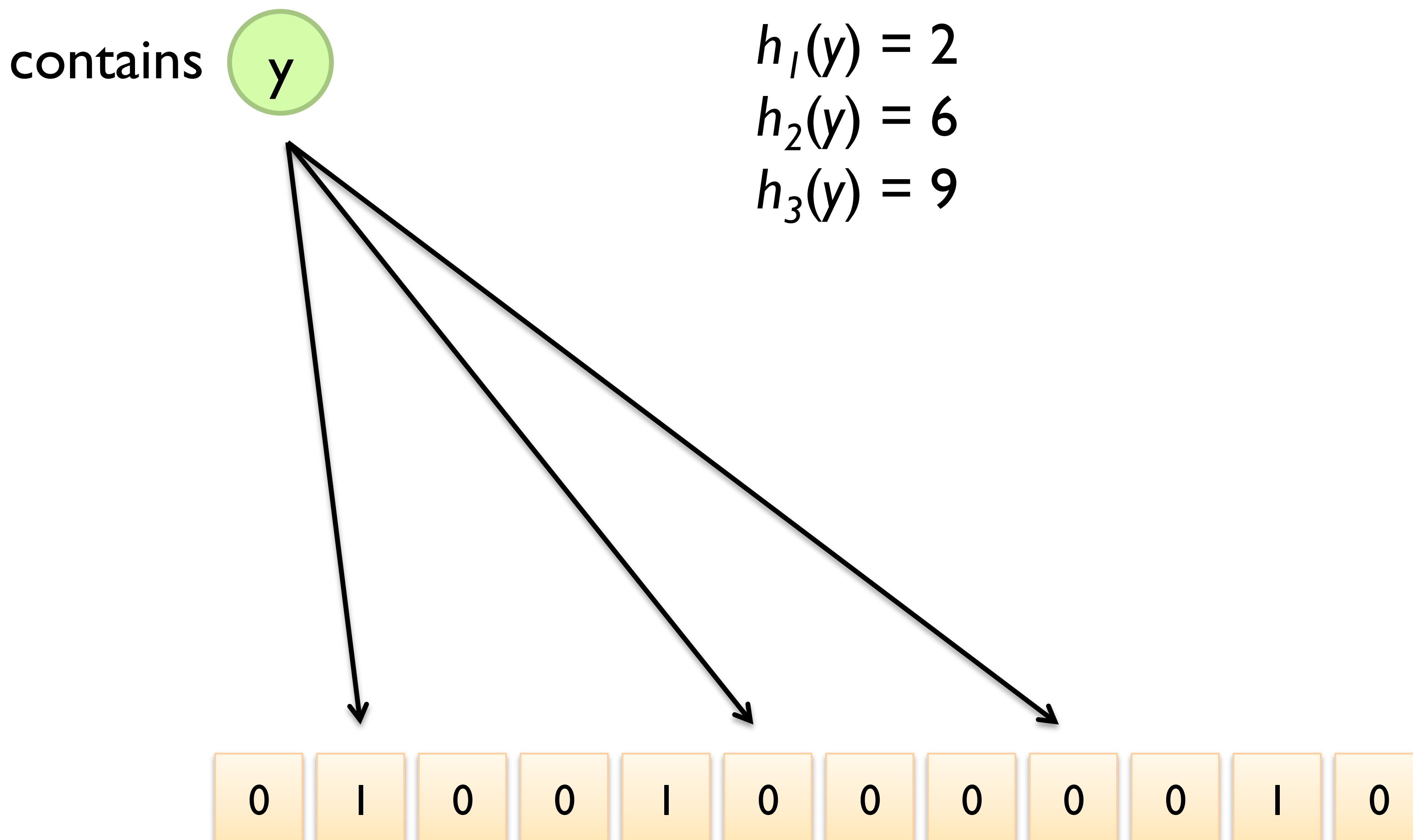
contains x

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

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

0	1	0	0	1	0	0	0	0	0	1	0

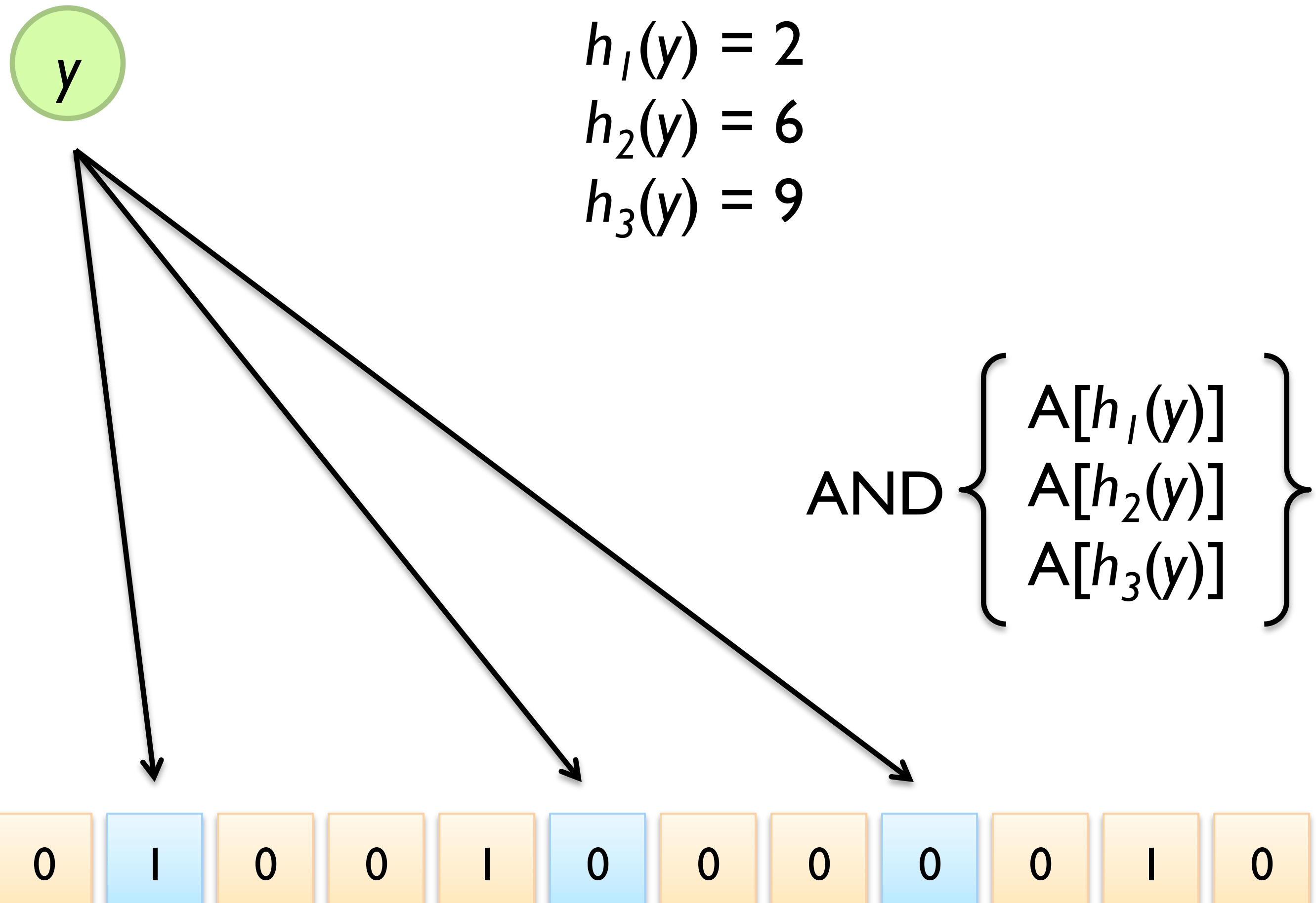
Bloom Filters: contains



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

Bloom Filters: contains

contains



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

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

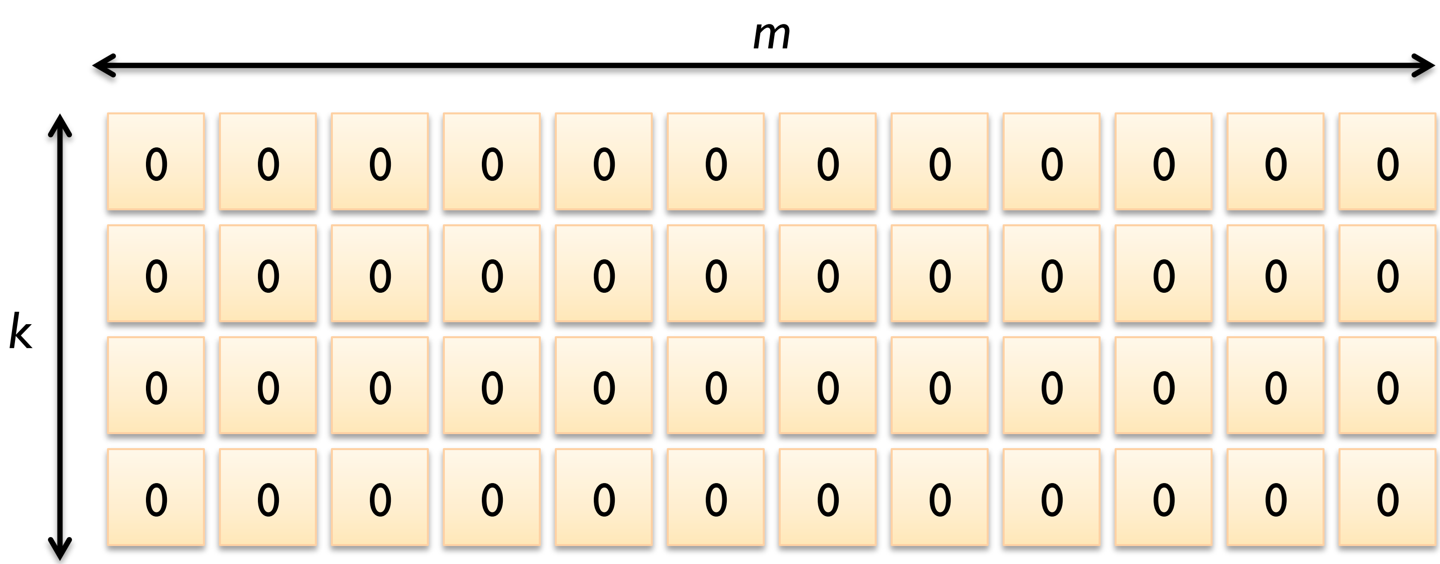
What's going on here?

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

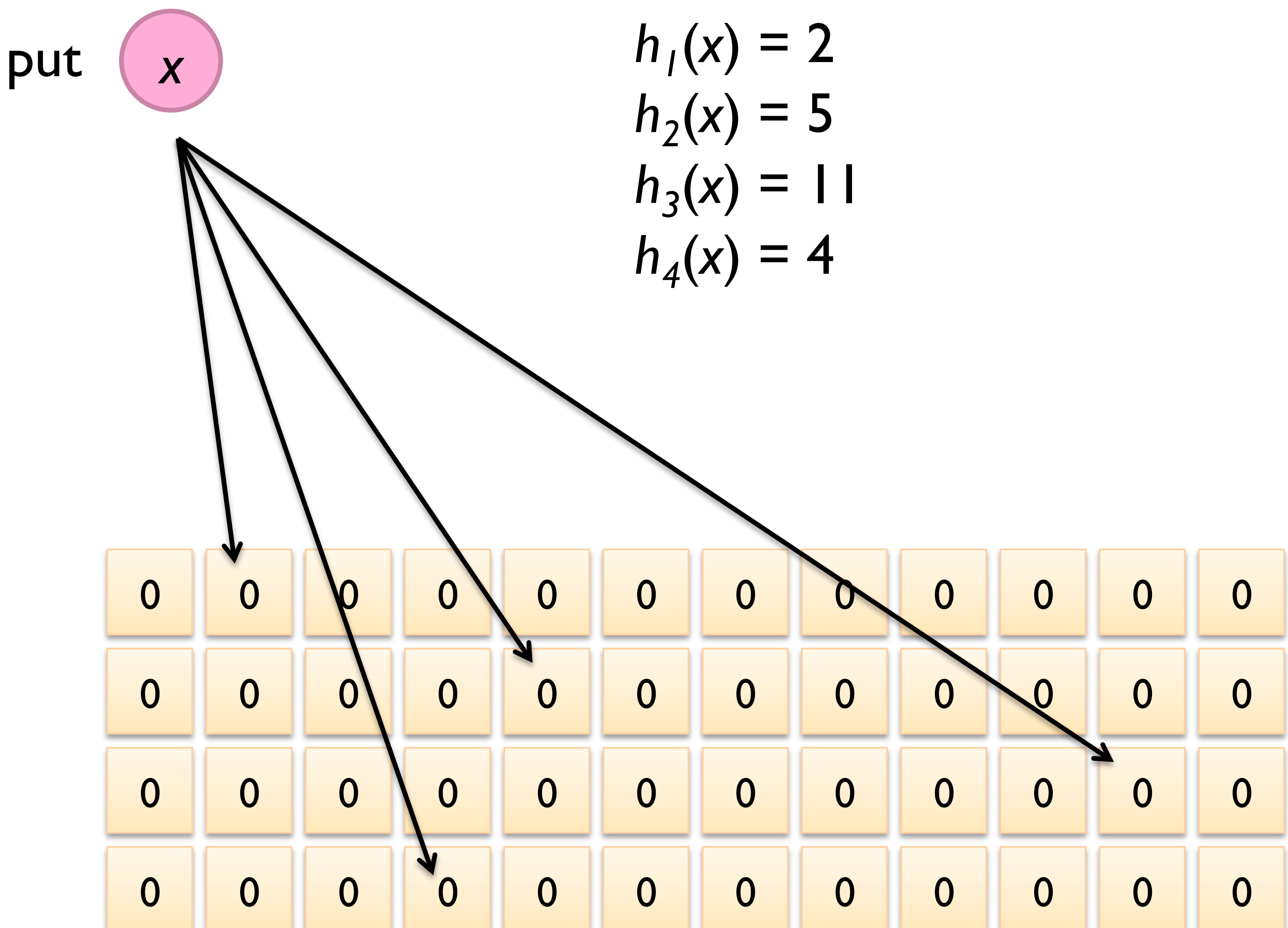
Count-Min Sketches

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



Count-Min Sketches: put

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



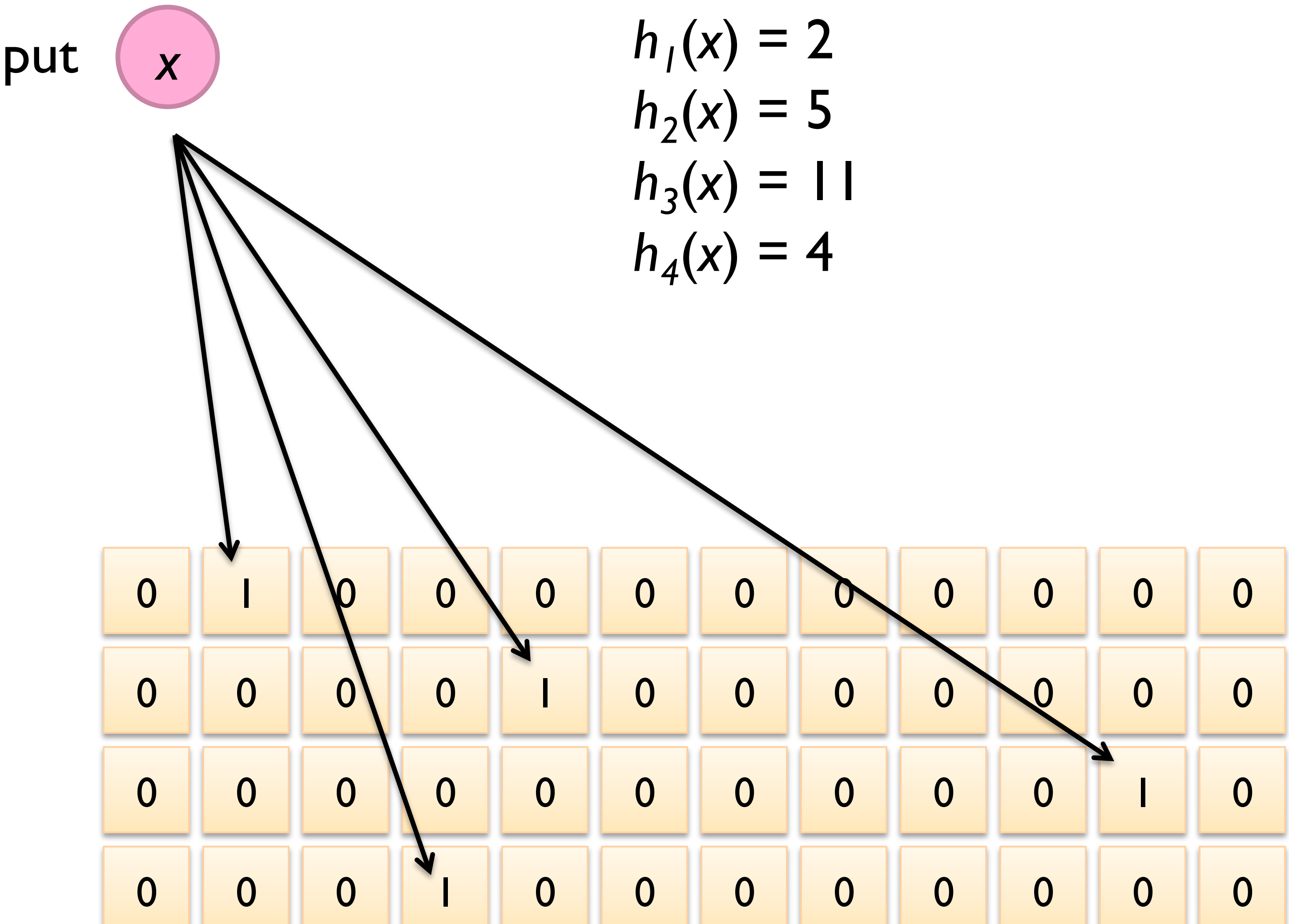
Count-Min Sketches: put

put x

0	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

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



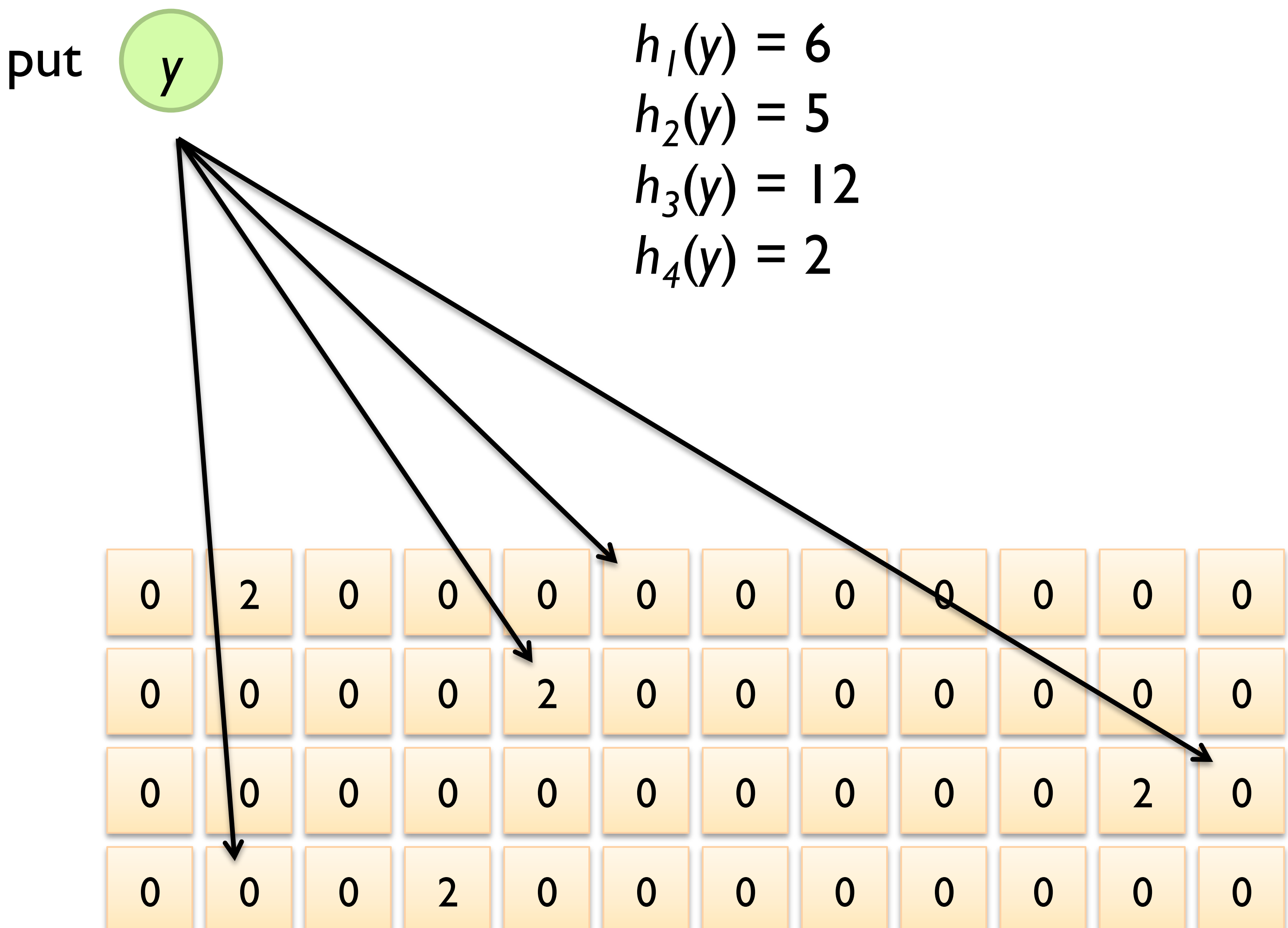
Count-Min Sketches: put

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

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



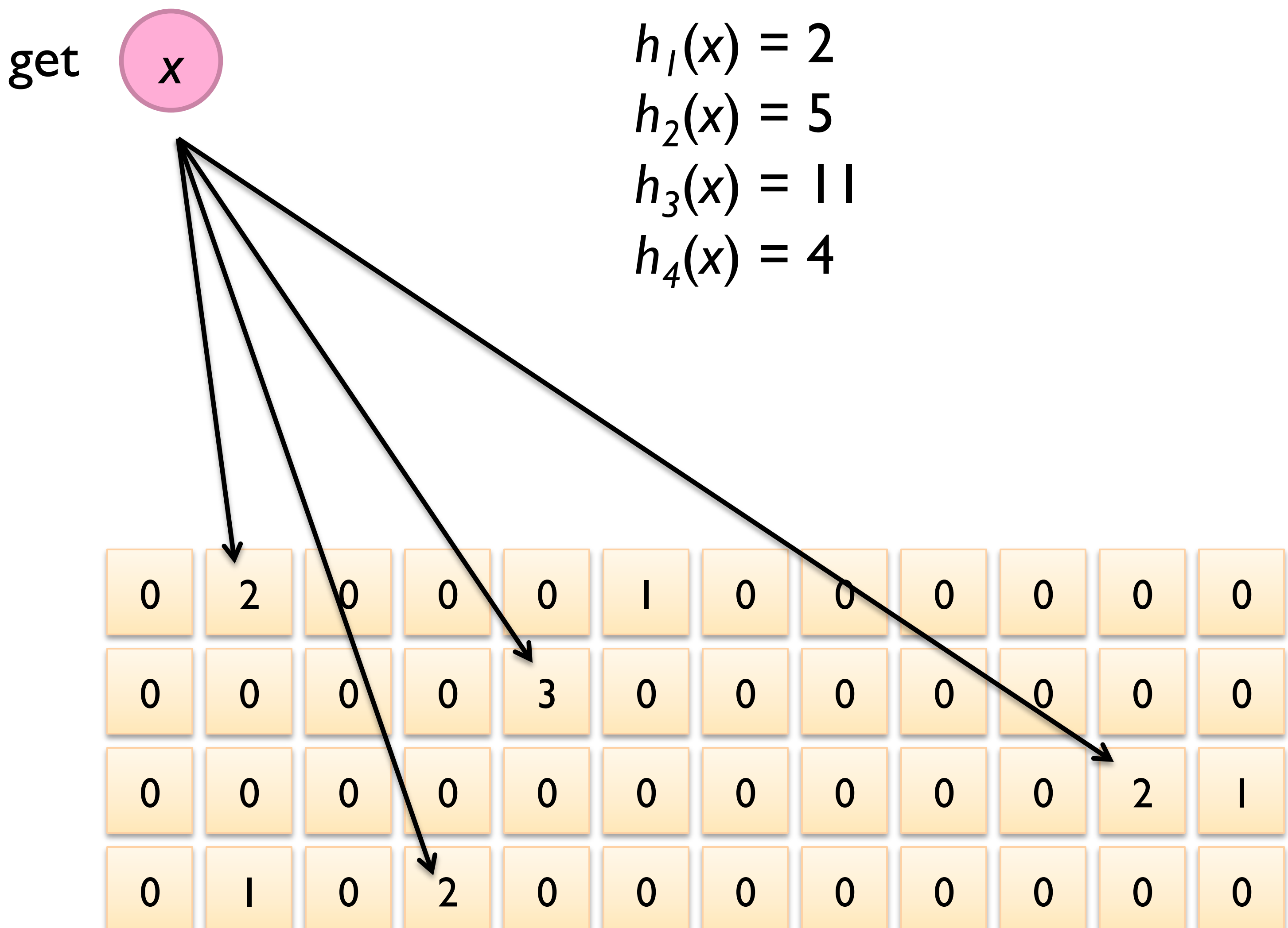
Count-Min Sketches: put

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

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



Count-Min Sketches: get

get x

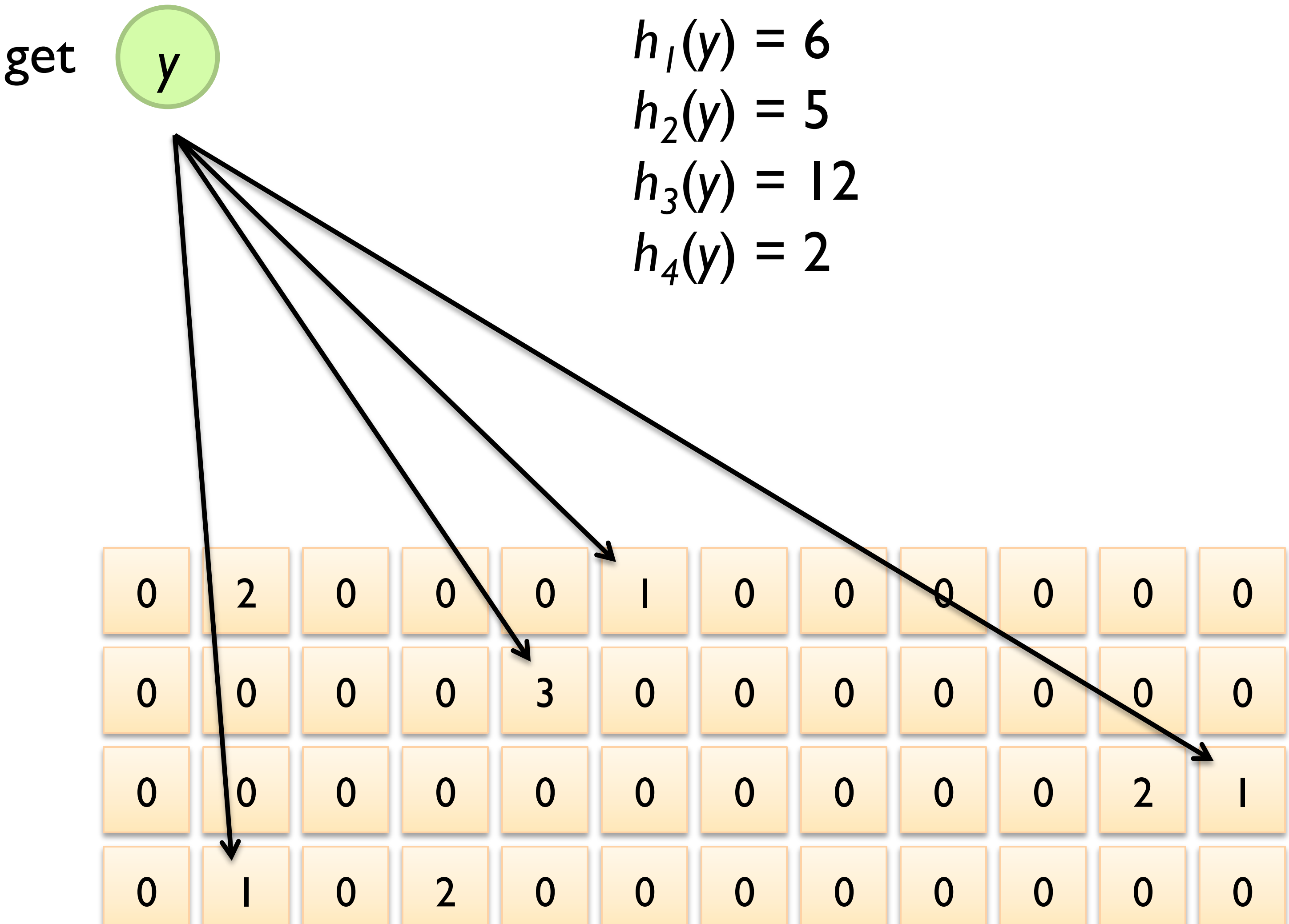
$h_1(x) = 2$
$h_2(x) = 5$
$h_3(x) = 11$
$h_4(x) = 4$

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

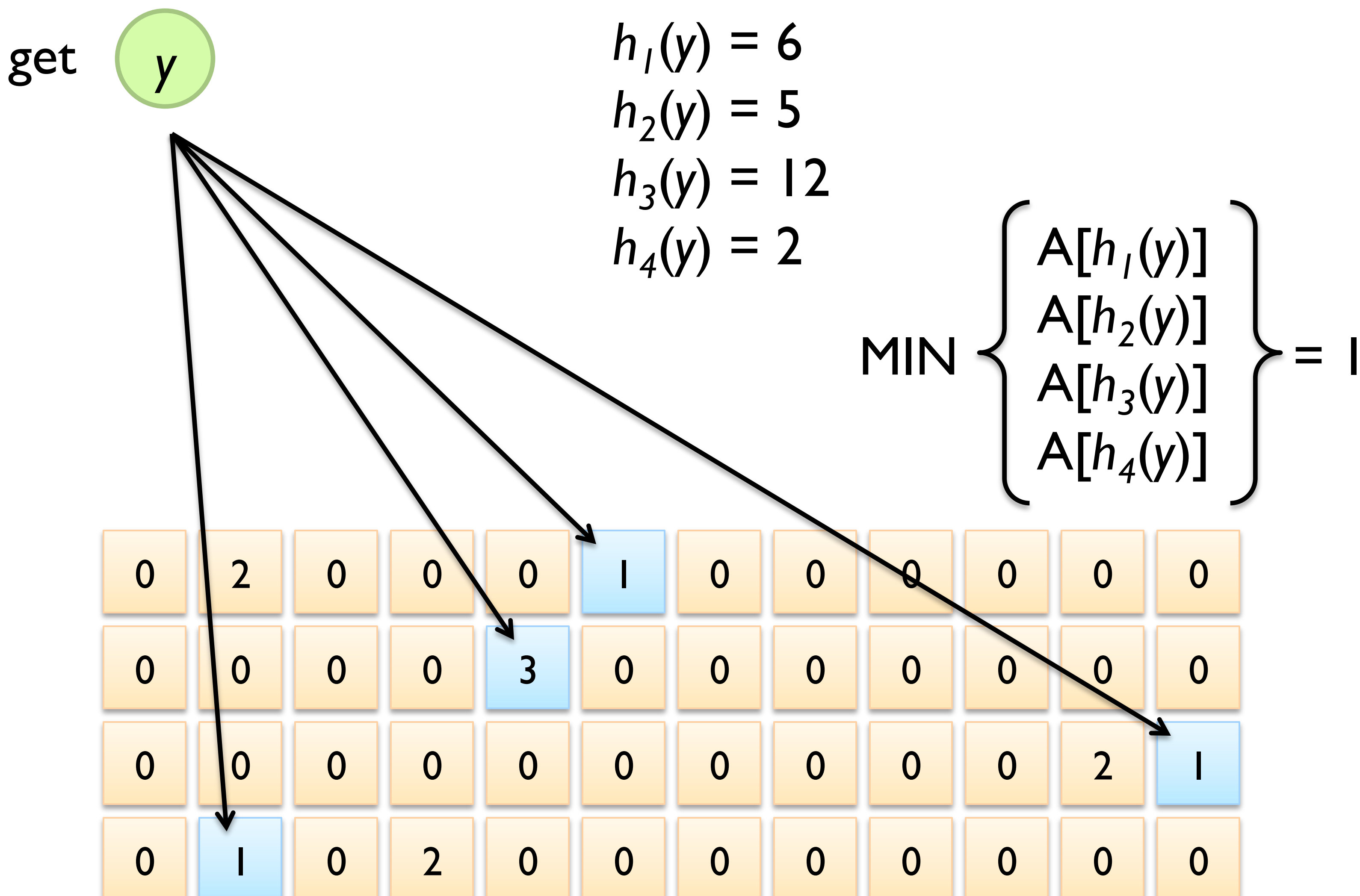
0	2	0	0	0	1	0	0	0	0	0	0
0	0	0	0	3	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	2	1
0	1	0	2	0	0	0	0	0	0	0	0

Count-Min Sketches: get

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



Count-Min Sketches: get



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

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?


Stream Processing Architectures
Source: Wikipedia (River)

Producer/Consumers

How do consumers get data from producers?

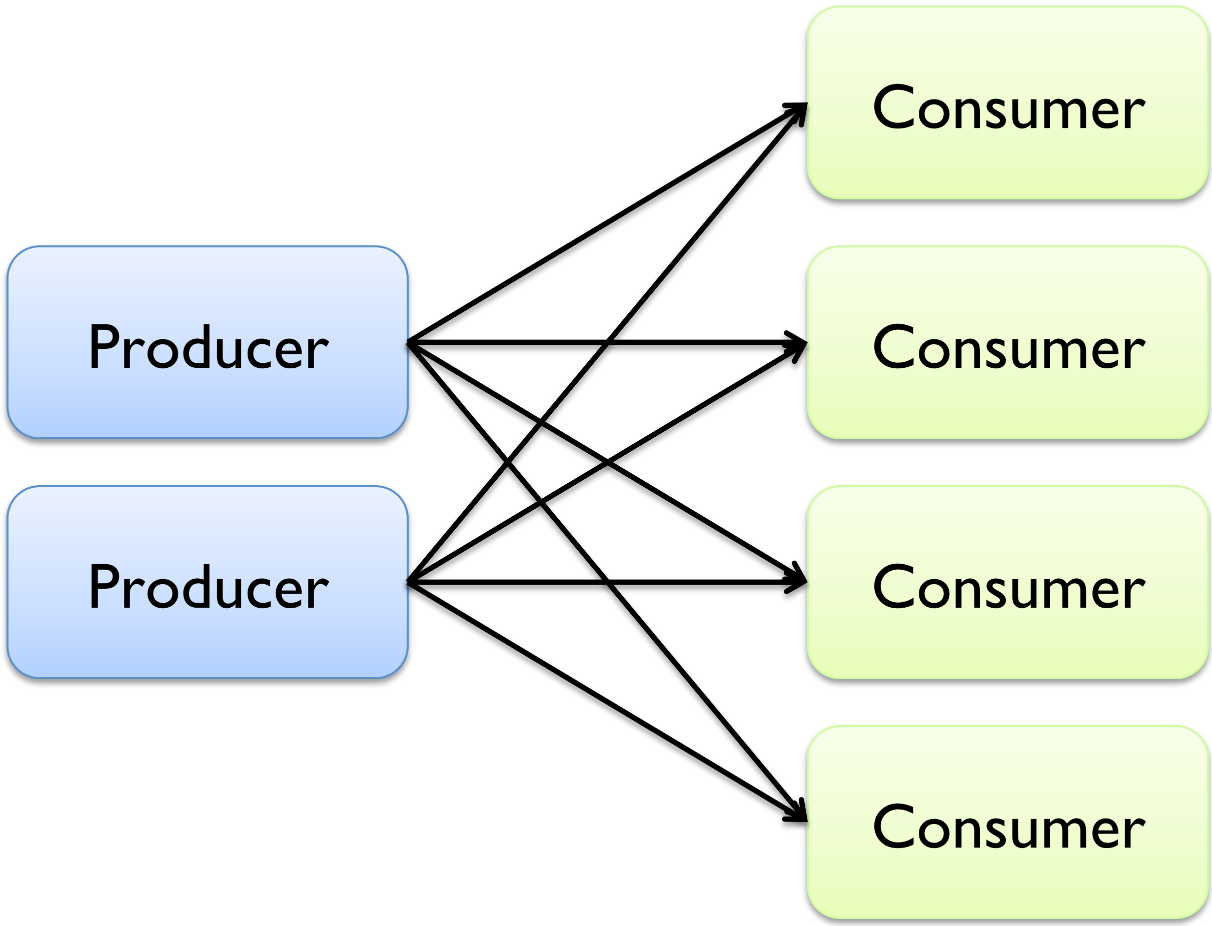
Producer/Consumers

Producer pushes
e.g., callback

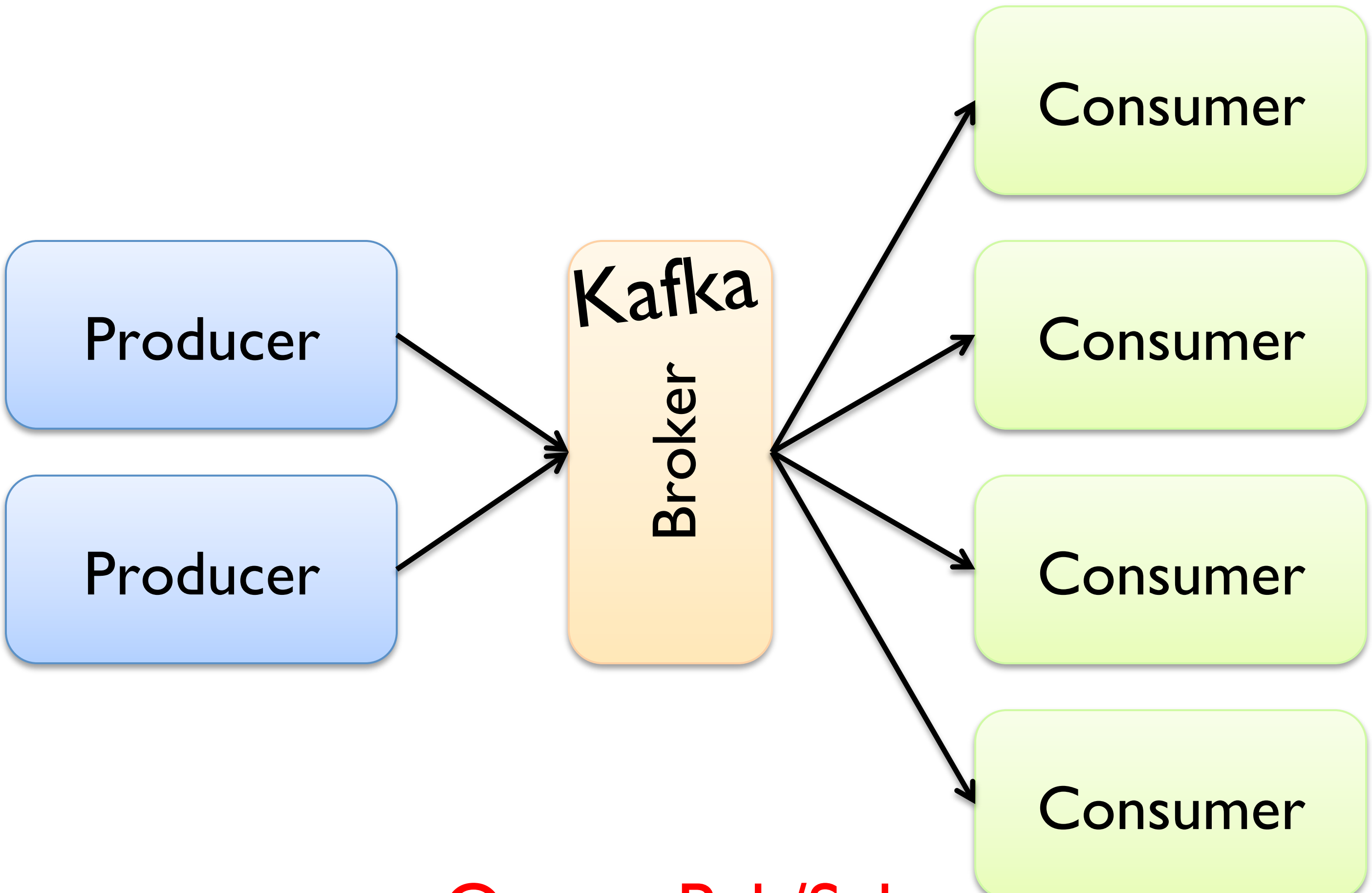
Producer/Consumers

Consumer pulls
e.g., poll, tail

Producer/Consumers



Producer/Consumers



Queue, Pub/Sub

Tuple-at-a-Time Processing

Storm

- Open-source real-time distributed stream processing system
  - Started at BackType
  - BackType acquired by Twitter in 2011
  - Now an Apache project
- Storm aspires to be the Hadoop of real-time processing!

Storm Topologies

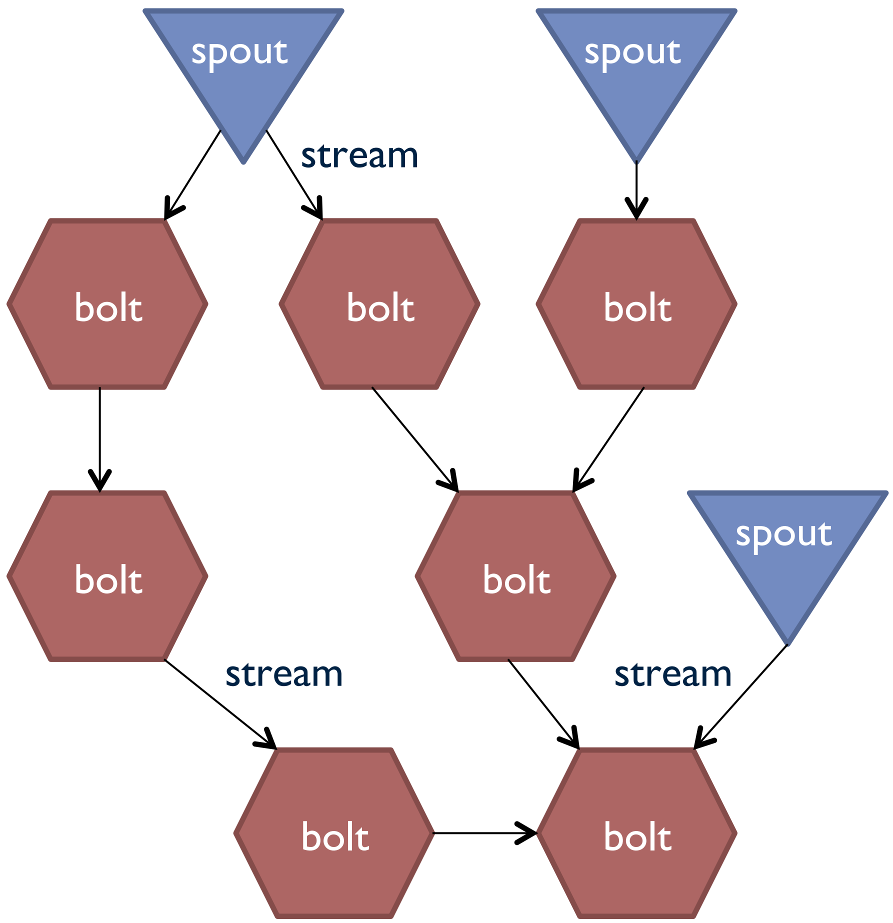
- Storm topologies = “job”
  - Once started, runs continuously until killed
- A Storm topology is a computation graph
  - Graph contains nodes and edges
  - Nodes hold processing logic (i.e., transformation over its input)
  - Directed edges indicate communication between nodes
- Processing semantics:
  - At most once: without acknowledgments
  - At least once: with acknowledgements

Streams, Spouts, and Bolts

- Streams
  - The basic collection abstraction: an unbounded sequence of tuples
  - Streams are transformed by the processing elements of a topology

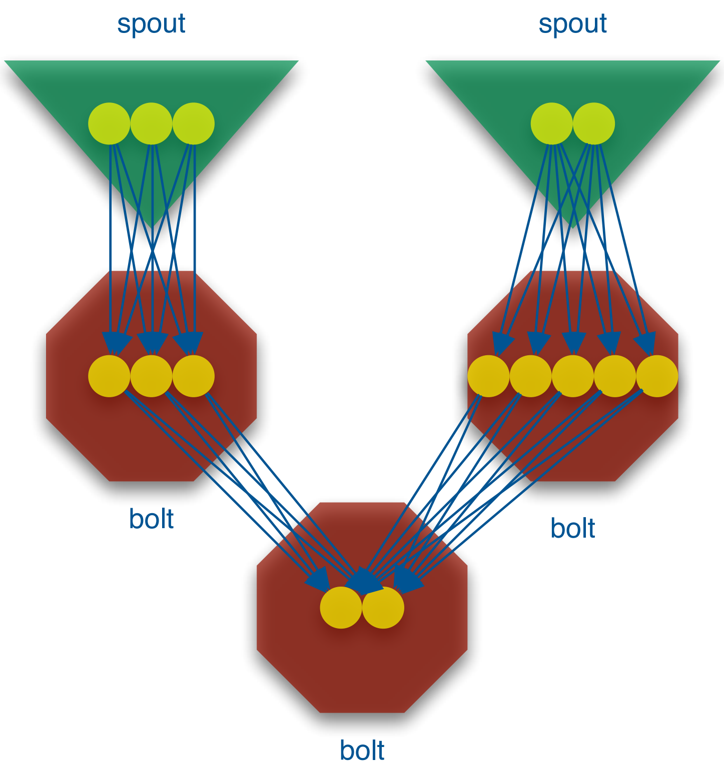
- Spouts
  - Stream generators
  - May propagate a single stream to multiple consumers

- Bolts
  - Subscribe to streams
  - Streams transformers
  - Process incoming streams and produce new ones



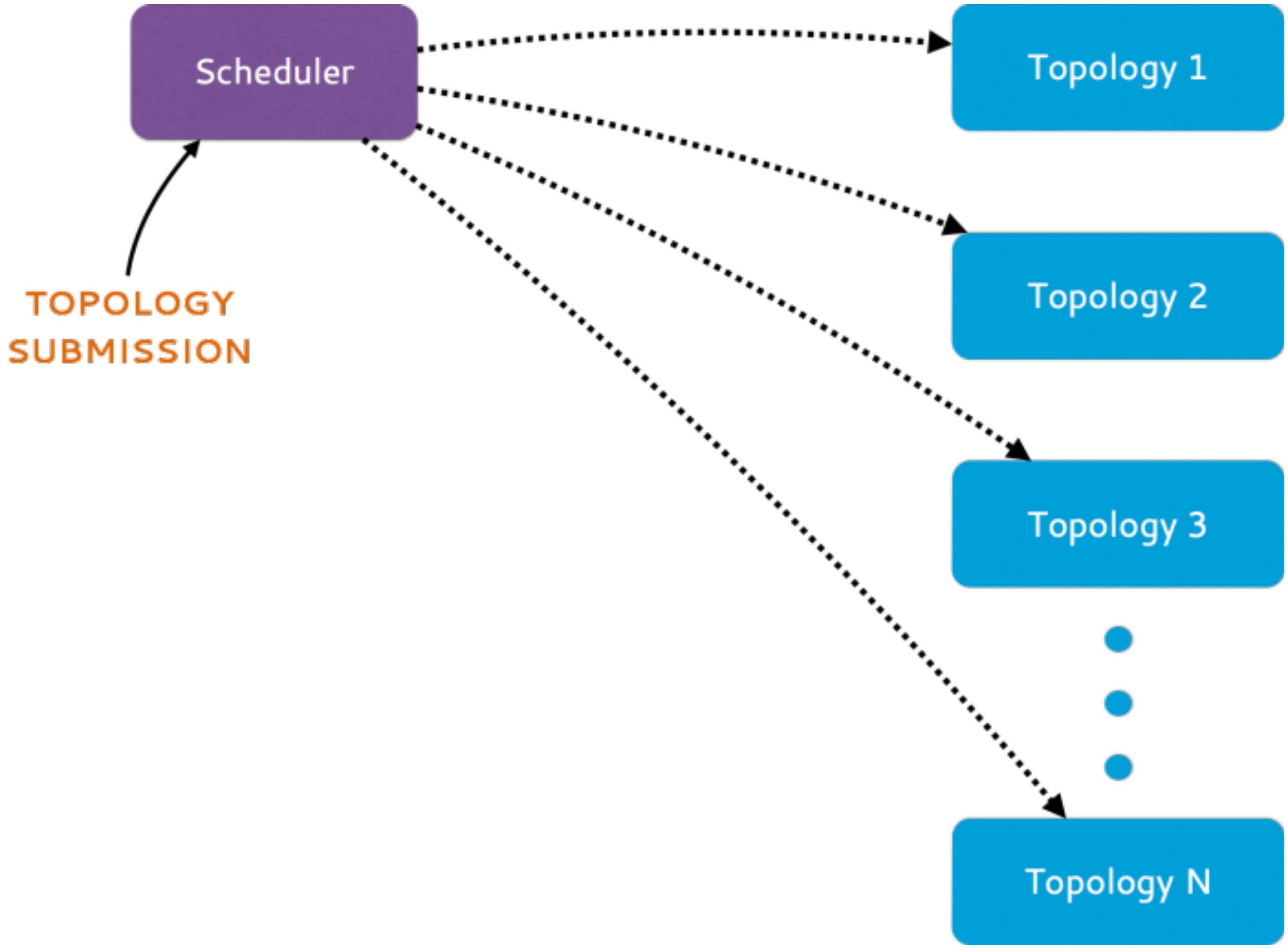
Stream Groupings

- 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

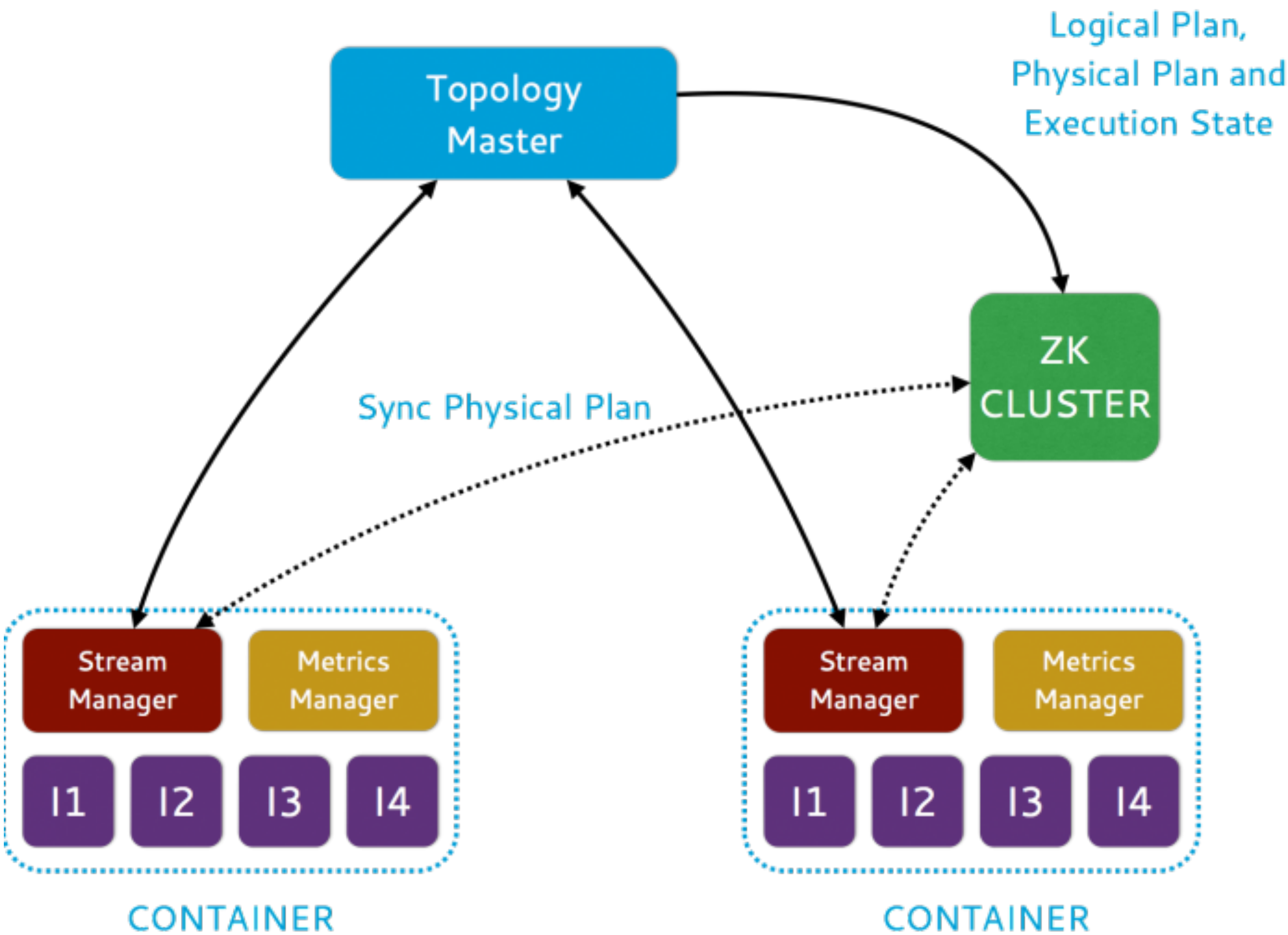


From Storm to Heron

- Heron = API compatible re-implementation of Storm



Source: https://blog.twitter.com/2015/flying-faster-with-twitter-heron



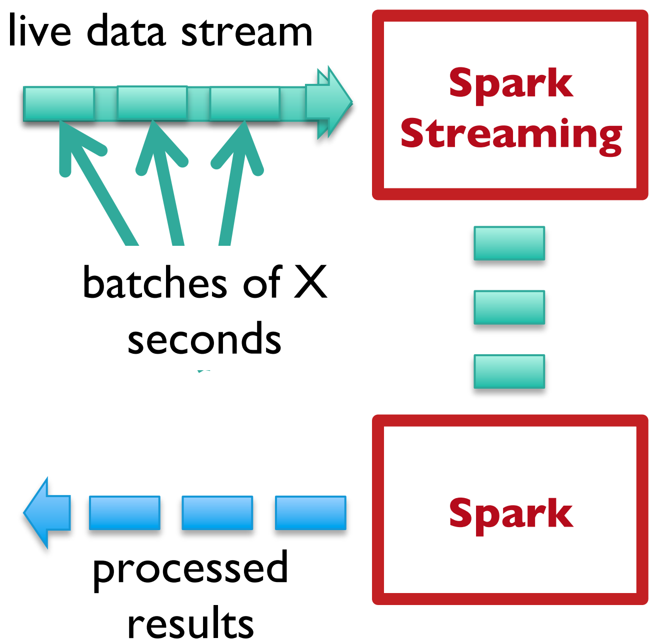
Source: https://blog.twitter.com/2015/flying-faster-with-twitter-heron

Mini-Batch Processing

Discretized Stream Processing

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

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

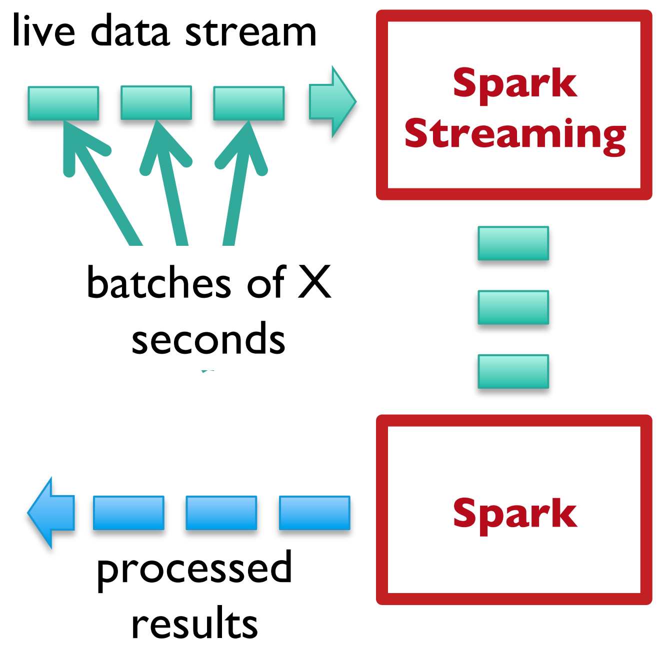


Source: All following Spark Streaming slides by Tathagata Das

Discretized Stream Processing

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

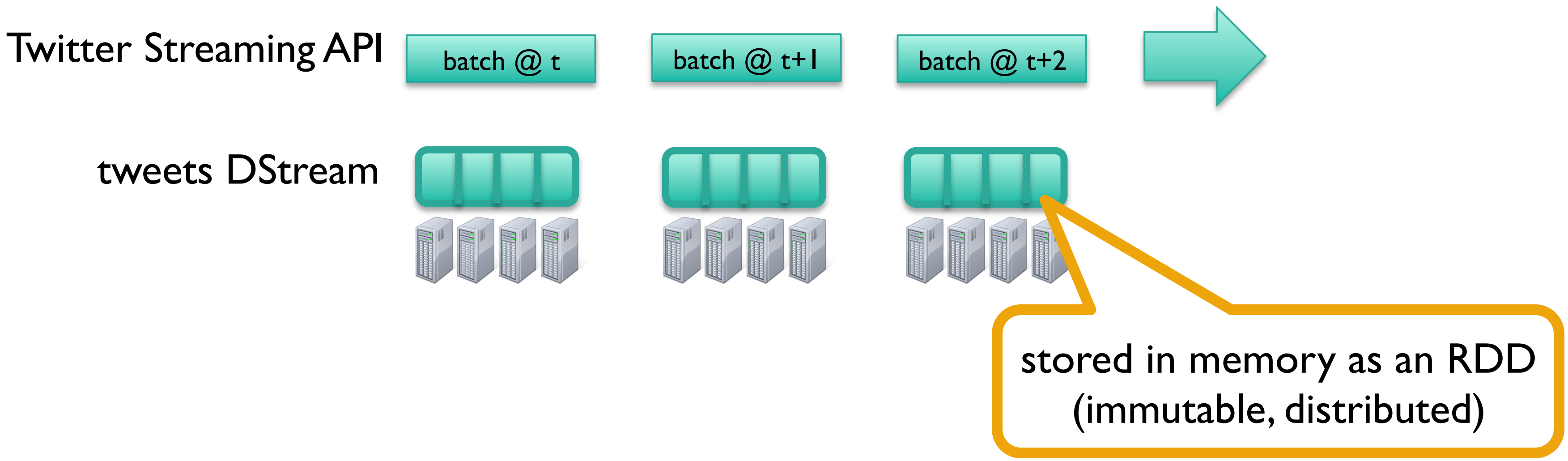
- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

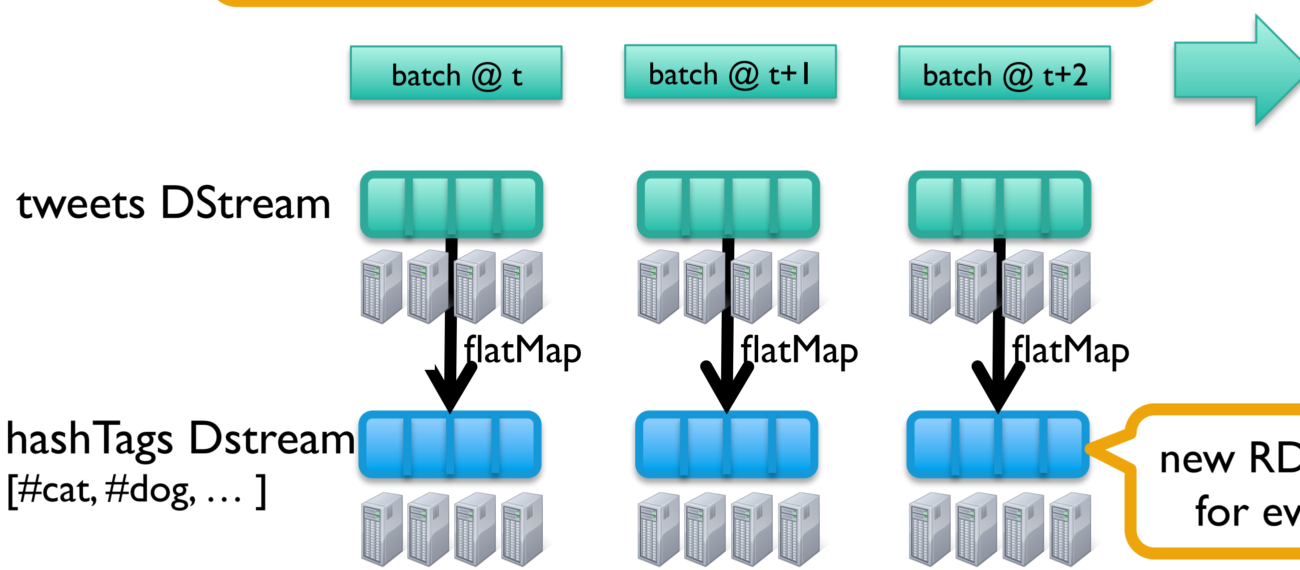


Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

transformation: modify data in one Dstream to create another DStream

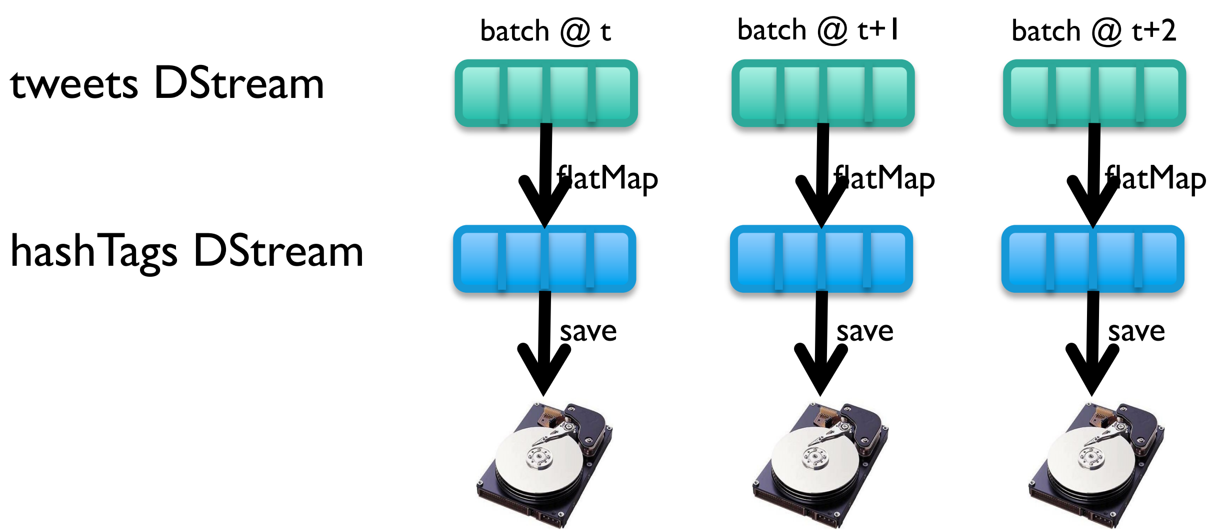


new RDDs created for every batch

Example: Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

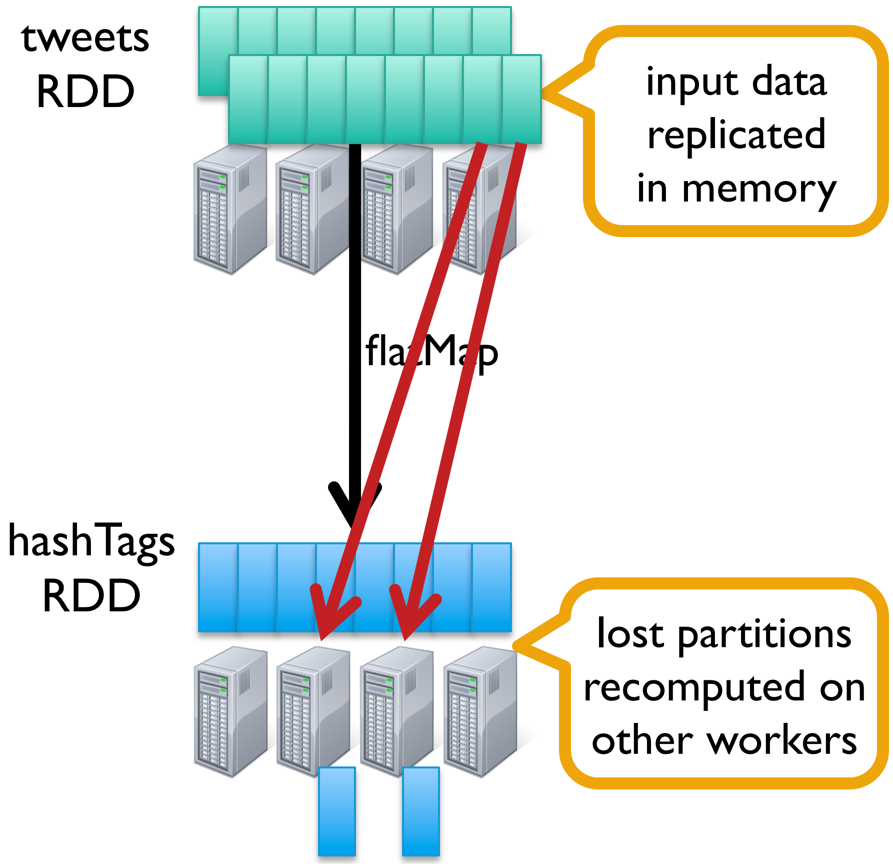
output operation: to push data to external storage



every batch saved to HDFS

Fault-tolerance

- RDDs are remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data



Key concepts

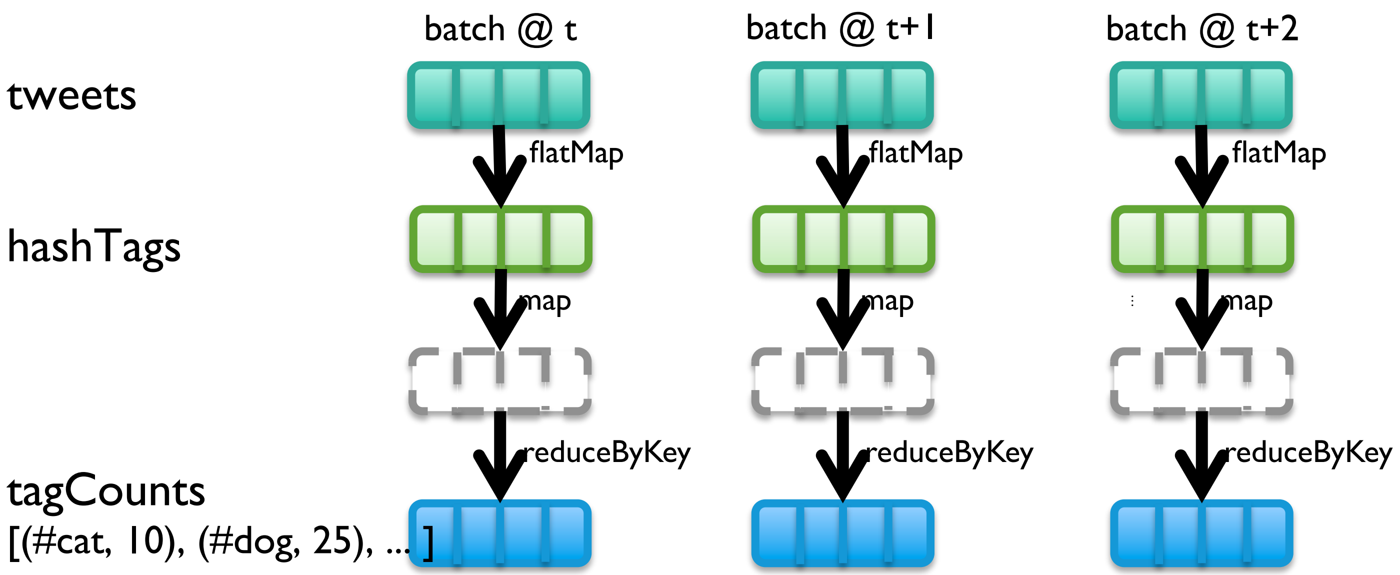
- DStream – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets

- Transformations – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …

- Output Operations – send data to external entity
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Example: Count the hashtags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```



Example: Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

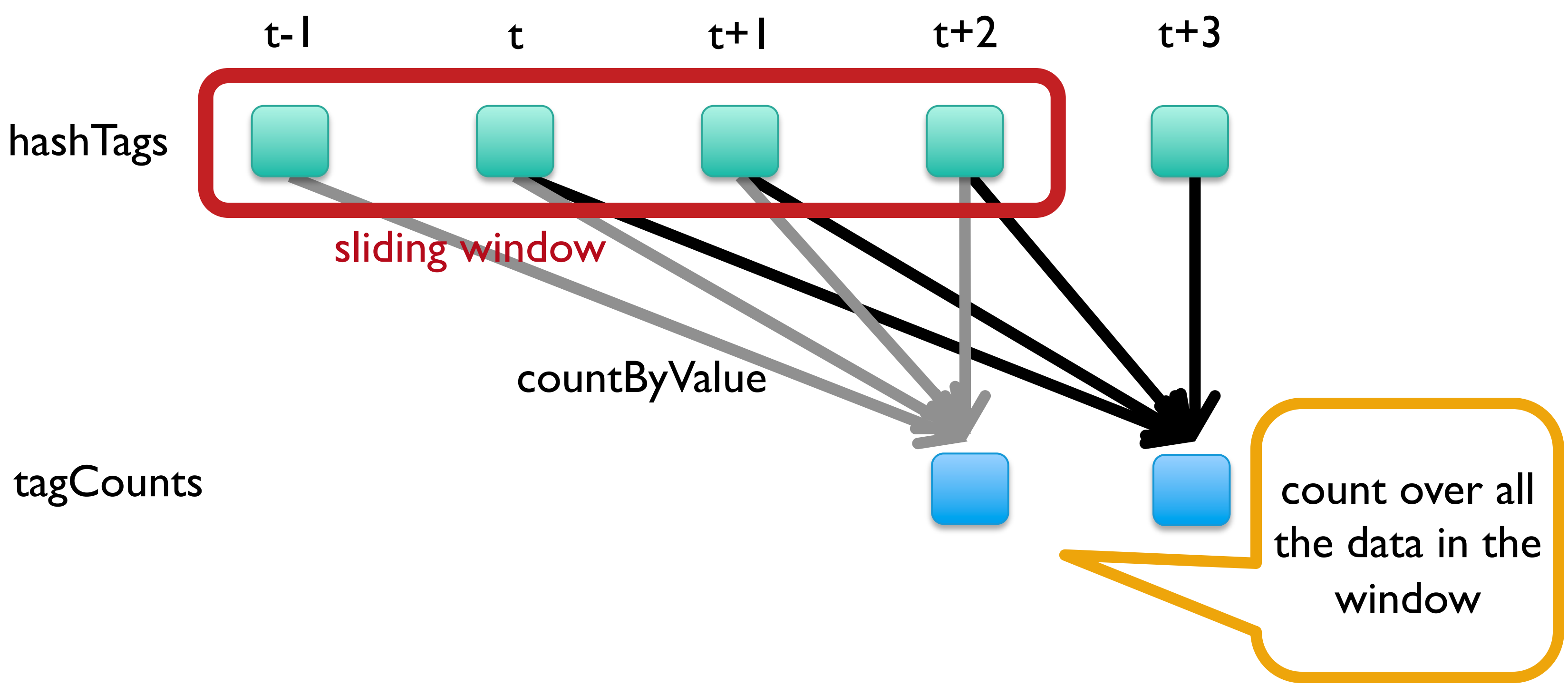
sliding window operation

window length

sliding interval

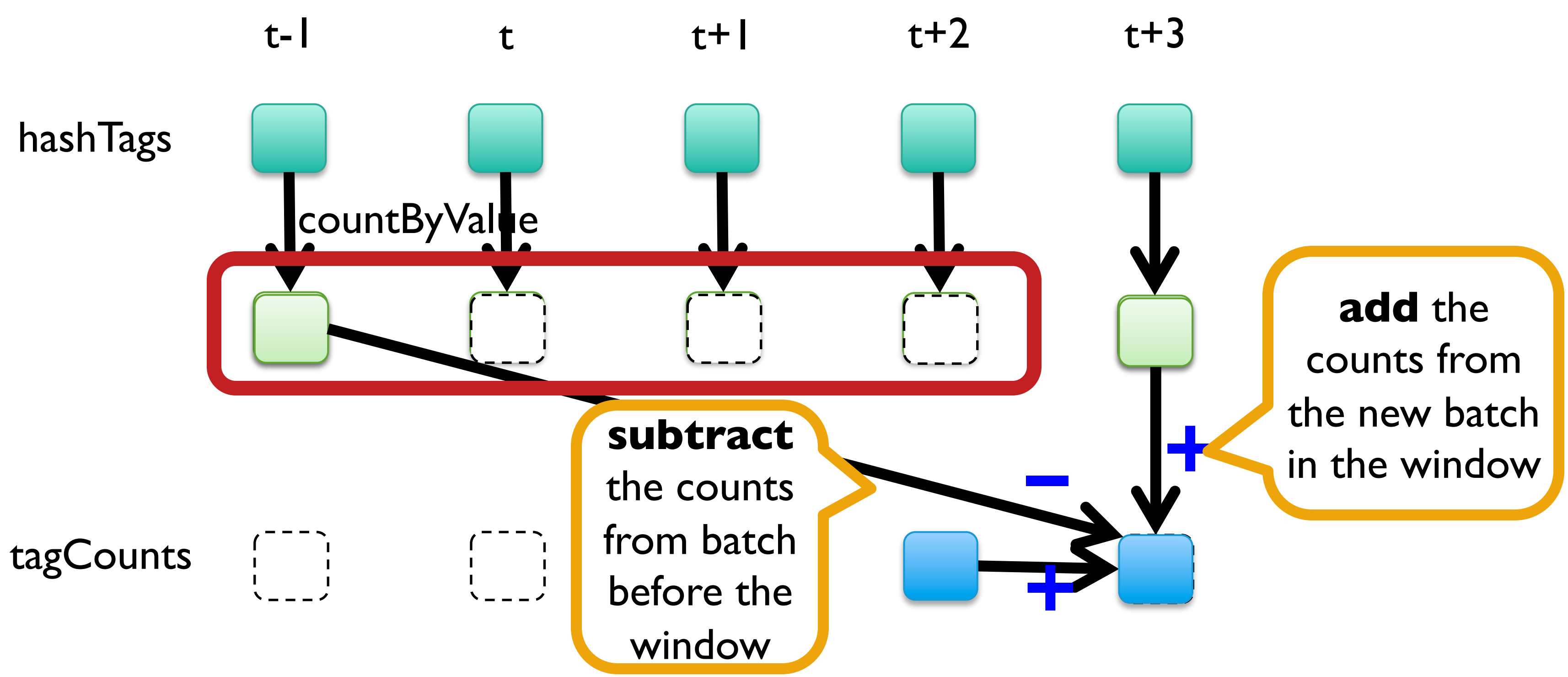
Example: Count the hashtags over last 10 mins

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)
```

Integrating Batch and Online Processing

Summingbird

A domain-specific language (in Scala) designed
to integrate batch and online MapReduce computations

Idea #1: Algebraic structures provide the basis for
seamless integration of batch and online processing

Idea #2: For many tasks, close enough is good enough
Probabilistic data structures as monoids

Batch and Online MapReduce

"map"

```
flatMap[T, U](fn: T => List[U]): List[U]
```

```
map[T, U](fn: T => U): List[U]
```

```
filter[T](fn: T => Boolean): List[T]
```

"reduce"

```
sumByKey
```

Idea #1: Algebraic structures provide the basis for seamless integration of batch and online processing

Semigroup = $(M , \oplus)$

$$\oplus : M \times M \longrightarrow M, \text{ s.t., } \forall m_1, m_2, m_3 \ni M$$

$$(m_1 \oplus m_2) \oplus m_3 = m_1 \oplus (m_2 \oplus m_3)$$

Monoid = Semigroup + identity

$$\varepsilon \text{ s.t., } \varepsilon \oplus m = m \oplus \varepsilon = m, \ \forall m \ni M$$

Commutative Monoid = Monoid + commutativity

$$\forall m_1, m_2 \ni M, m_1 \oplus m_2 = m_2 \oplus m_1$$

Simplest example: integers with + (addition)

Idea #1: Algebraic structures provide the basis for seamless integration of batch and online processing

Summingbird values must be at least semigroups
(most are commutative monoids in practice)

Power of associativity =
You can put the parentheses anywhere!

$(a \oplus b \oplus c \oplus d \oplus e \oplus f)$	Batch = Hadoop
$(((((a \oplus b) \oplus c) \oplus d) \oplus e) \oplus f)$	Online = Storm
$((a \oplus b \oplus c) \oplus (d \oplus e \oplus f))$	Mini-batches

Results are exactly the same!

Summingbird Word Count

```
def wordCount[P <: Platform[P]]
  (source: Producer[P, String],
   store: P#Store[String, Long]) =
   source.flatMap { sentence =>
      toWords(sentence).map(_ -> 1L)
    }.sumByKey(store)
```

where data comes from

where data goes

“map”

“reduce”

Run on Scalding (Cascading/Hadoop)

```
Scalding.run {
  wordCount[Scalding](
    Scalding.source[Tweet]("source_data"),
    Scalding.store[String, Long]("count_out")
  )
}
```

read from HDFS

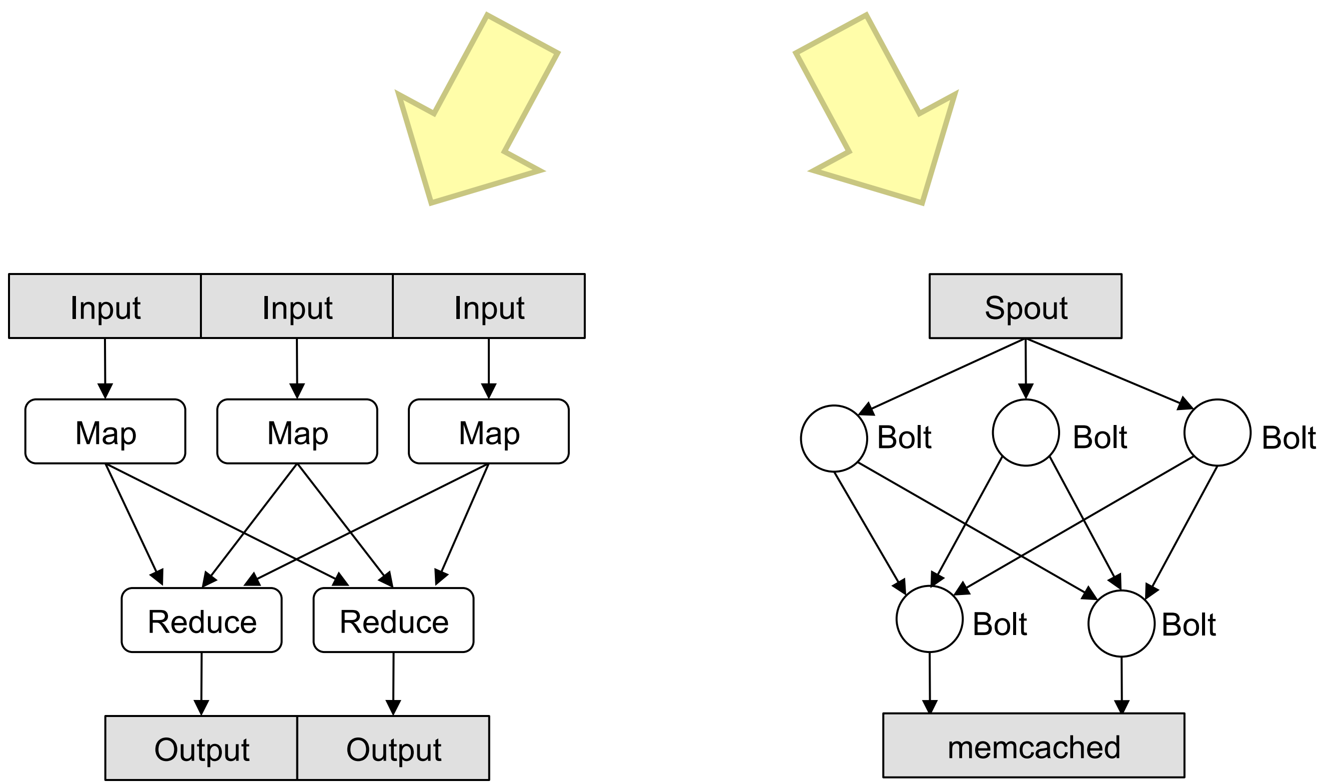
write to HDFS

Run on Storm

```
Storm.run {
  wordCount[Storm](
    new TweetSpout(),
    new MemcacheStore[String, Long]
  )
}
```

read from message queue

write to KV store


Input
Input
Input
Map
Map
Map
Reduce
Reduce
Output
Output
Spout
Bolt
Bolt
Bolt
Bolt
Bolt
memcached

“Boring” monoids

addition, multiplication, max, min

moments (mean, variance, etc.)

sets

tuples of monoids

hashmaps with monoid values

More interesting monoids?

Idea #2: For many tasks, close enough is good enough!

"Interesting" monoids

Bloom filters (set membership)

HyperLogLog counters (cardinality estimation)

Count-min sketches (event counts)

Common features

1. Variations on hashing
2. Bounded error

Cheat sheet

	Exact	Approximate
Set membership	set	Bloom filter
Set cardinality	set	hyperloglog counter
Frequency count	hashmap	count-min sketches

Task: count queries by hour

Exact with hashmaps

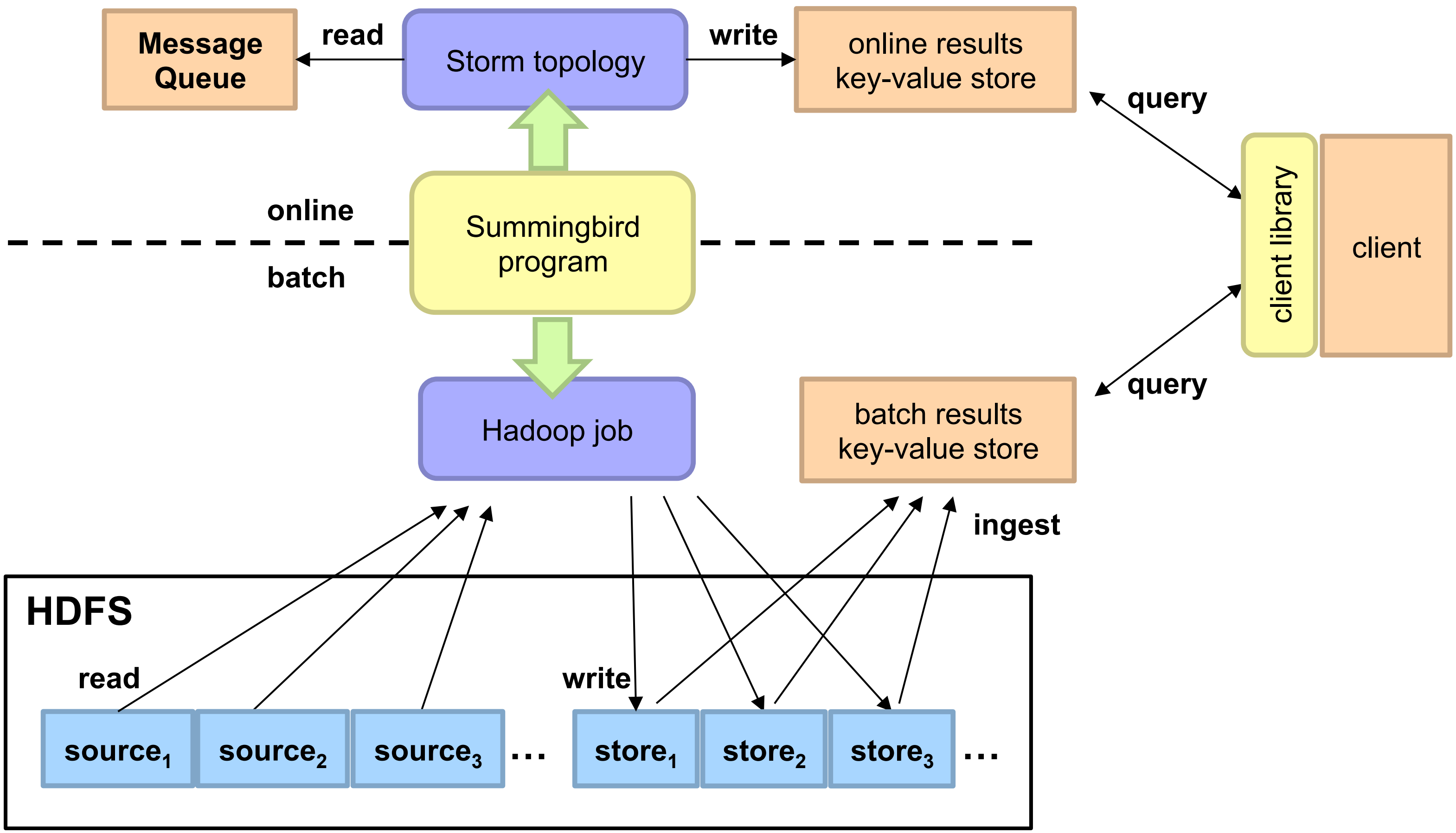
```
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, Map[String, Long]]) =
   source.flatMap { query =>
      (query.getHour, Map(query.getQuery -> 1L))
    }.sumByKey(store)
```

Approximate with CMS

```
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, SketchMap[String, Long]])
  (implicit countMonoid: SketchMapMonoid[String, Long]) =
   source.flatMap { query =>
      (query.getHour,
       countMonoid.create((query.getQuery, 1L)))
    }.sumByKey(store)
```

Hybrid Online/Batch Processing

Example: count historical clicks and clicks in real time



Source: Wikipedia (Japanese rock garden)