UNIVERSITY OF WATERLOO

Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2017)

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

March 30, 2016

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

These slides are available at http://lintool.github.io/bigdata-2017w/


Home
Connect
Discover
Me
sochi
#Sochi2014
#SochiProblems
Sochi
#SochiFail
Mishne et al. Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. SIGMOD 2013.
@lintool
TWEETS 1,647
FOLLOWING 253
FOLLOWERS 6,565
Compose new Tweet...
Who to follow · Refresh · View all
plotly @plotlygraphs
Follow
Promoted
Brad Anderson @boorad
Followed by Florian Leibert ...
Follow
Sheila Morrissey @sheilaMorr
Follow
Popular accounts · Find friends
Trends · Change
#Olympics
Promoted
Ukraine
#ConfessYourUnpopularOpinion
Venny
#PremioLoNuestro
Tweets
Promoted by Cloudera
Expand
Retweeted by Nitin Madnani
Clinton Paquin @clintonpaqui
View conversation
The Hill @thehill · 1h
View summary
Retweeted by Alex Feinberg
Popehat @Popehat · 10h
Expand
Reply Retweet Favorite More
The Hill @thehill · 1h
Boehner: I'd rather kill myself than raise the minimum wage trib.al/jZIkEus by @mollyhooper and @BobCusack
View summary
Reply Retweet Favorite More
CNN Breaking News @cnnbrk · 1h
Ukrainian Pres. says he has begun work on 3 key opposition demands: New elections, return to old constitution, formation of a unity gov's.
Expand
Reply Retweet Favorite More
Sochi 2014 @Sochi2014
Sochi Olympics 2014 @2014Sochi
Игры Сочи 2014 @sochi2014_ru
Sochi Problems @SochiProblem
NYT Olympics @SochiNYT
Sochi Problems @SochiProblems
Search all people for sochi

Initial Implementation

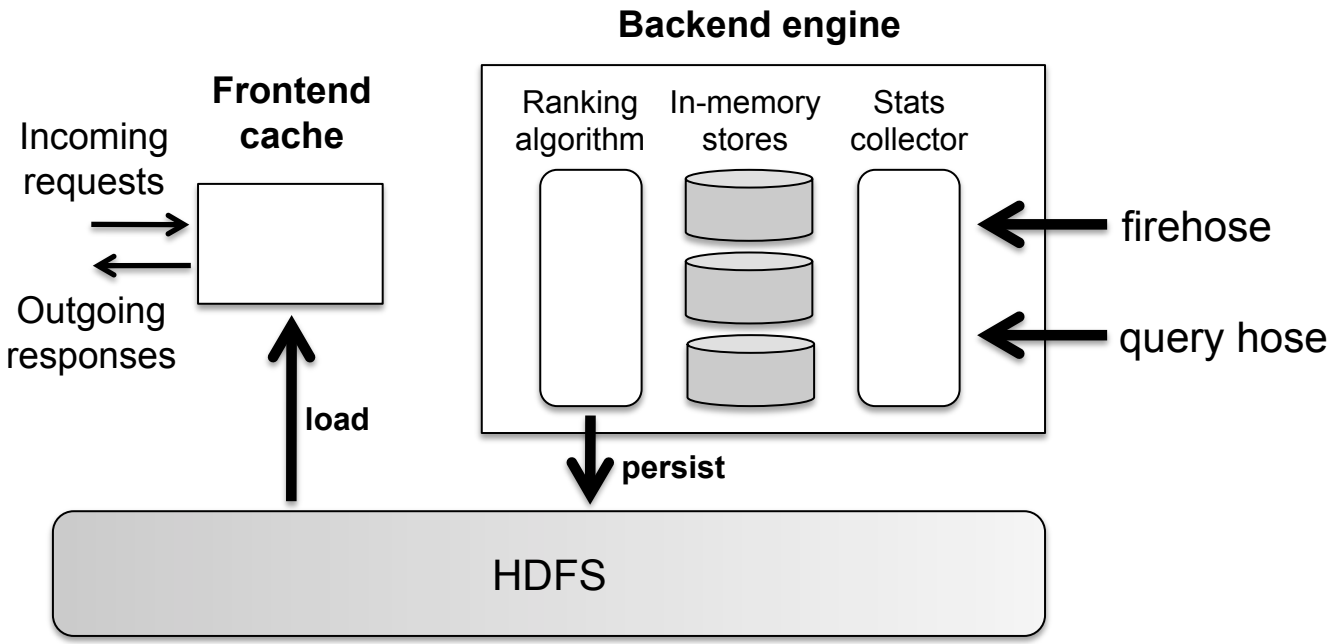
Algorithm: Co-occurrences within query sessions
Implementation: Pig scripts over query logs on HDFS

Problem: Query suggestions were several hours old!

Why?
Log collection lag
Hadoop scheduling lag
Hadoop job latencies

We need real-time processing!

Solution?



Can we do better than one-off custom architectures?

Source: Wikipedia (River)

Producer/Consumers

How do consumers get data from producers?

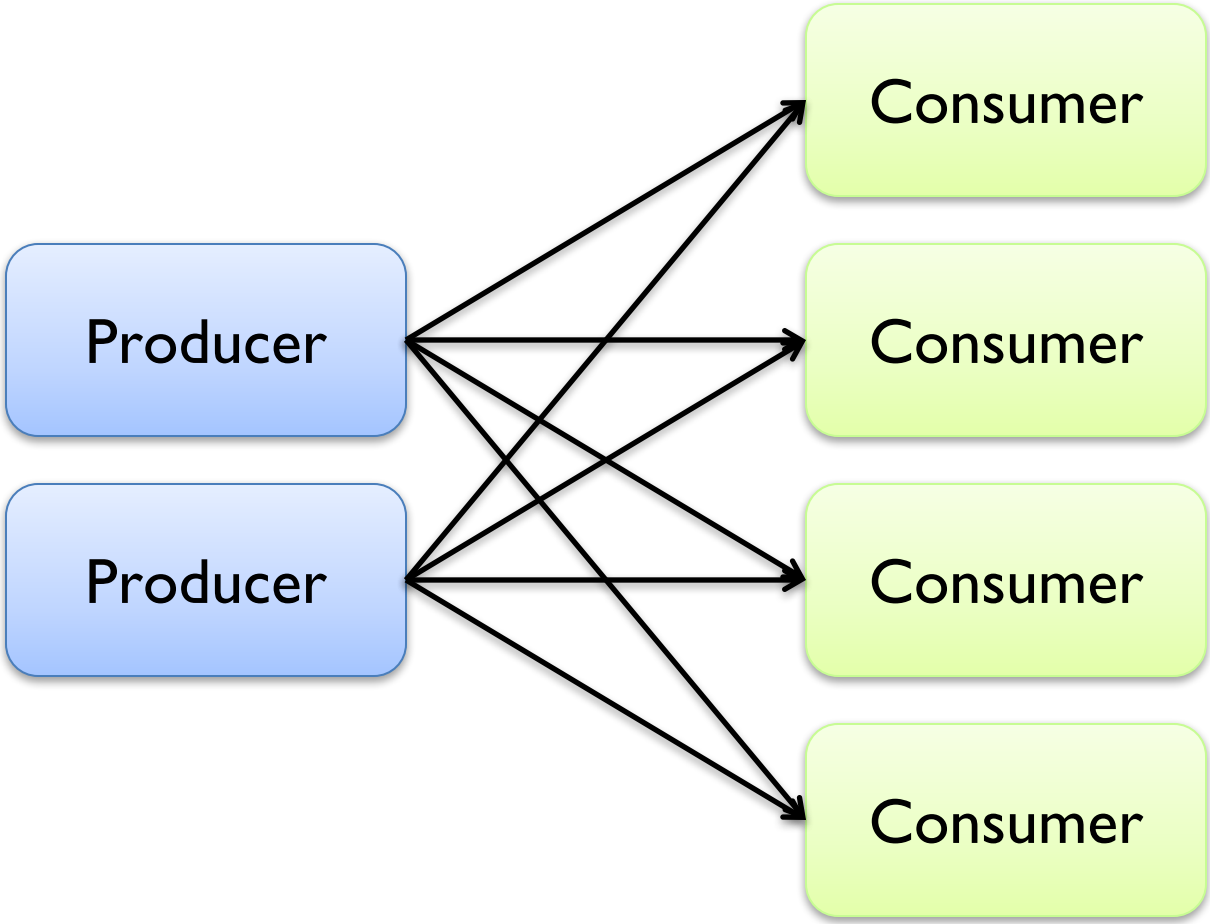
Producer/Consumers

Producer pushes
e.g., callback

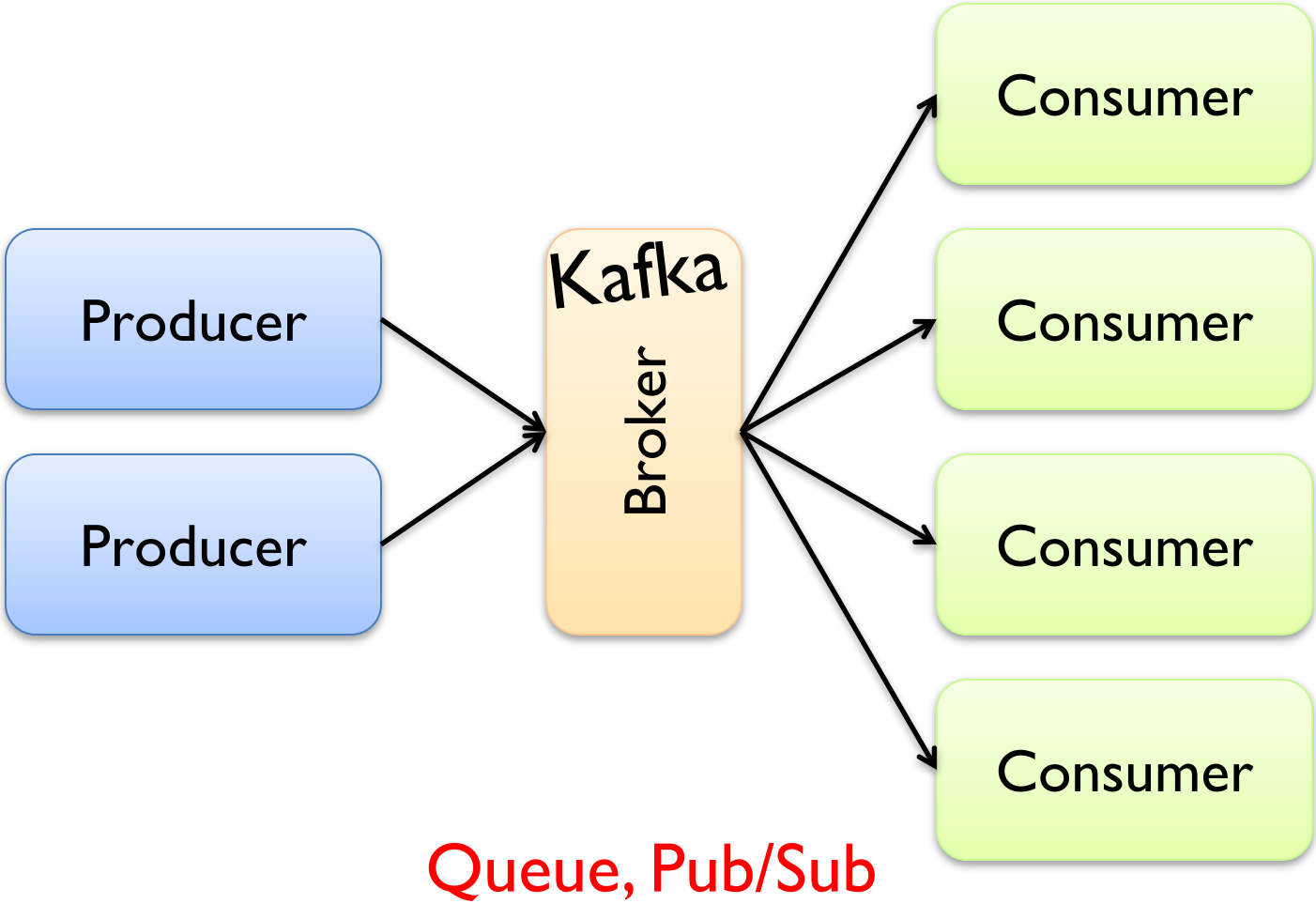
Producer/Consumers

Consumer pulls
e.g., poll, tail

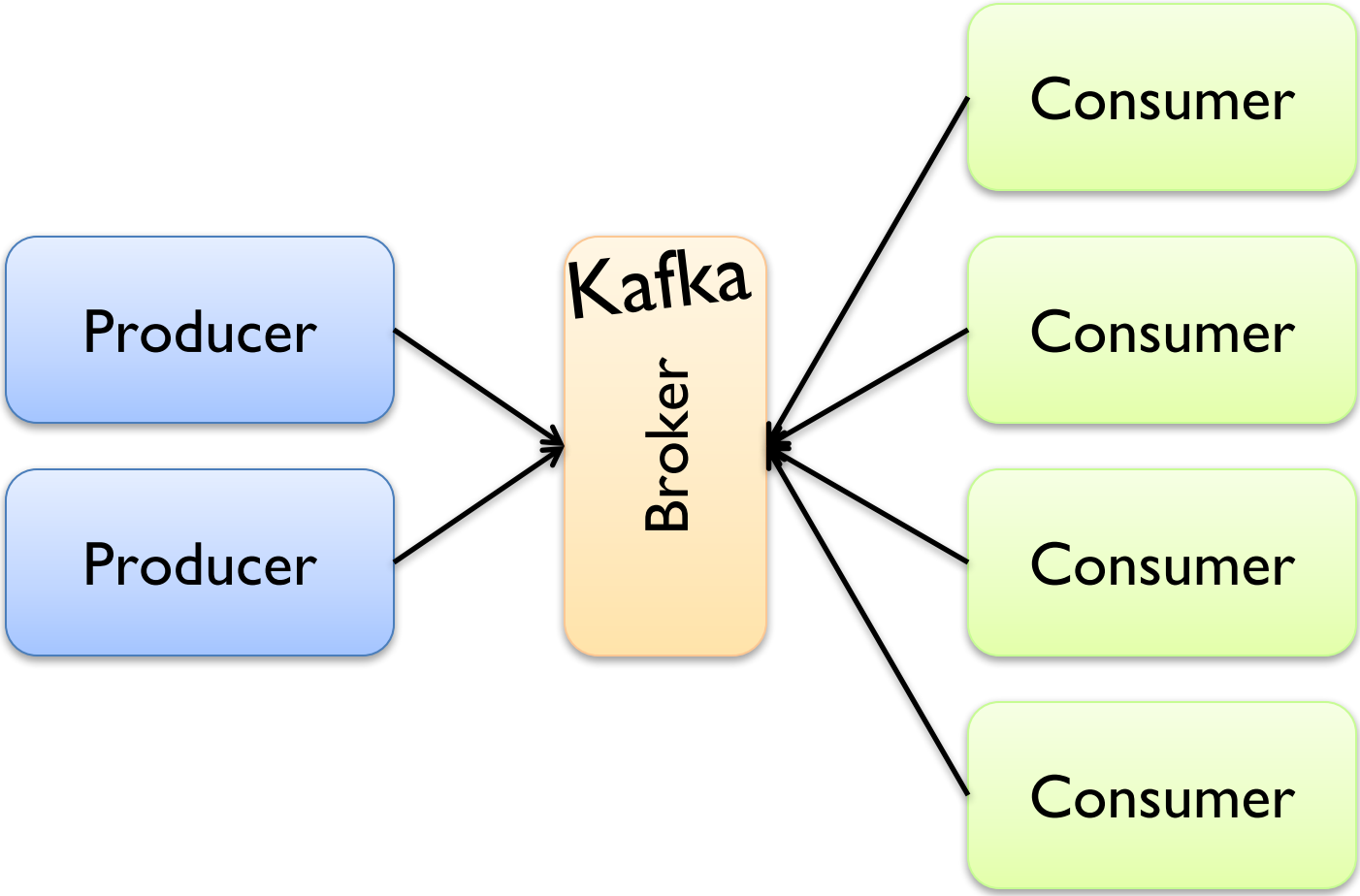
Producer/Consumers



Producer/Consumers



Producer/Consumers



Source: Wikipedia (River)

Storm/Heron

Storm: real-time distributed stream processing system

Started at BackType

BackType acquired by Twitter in 2011

Now an Apache project

Heron: API compatible re-implementation of Storm

Introduced by Twitter in 2015

Open-sourced in 2016

Want real-time stream processing?
I got your back.

I've got the most intuitive
implementation: a computation graph!

APACHE
STORM™

Distributed • Resilient • Real-time

Topologies

Storm topologies = “job”

Once started, runs continuously until killed

A topology is a computation graph

Graph contains vertices and edges

Vertices hold processing logic

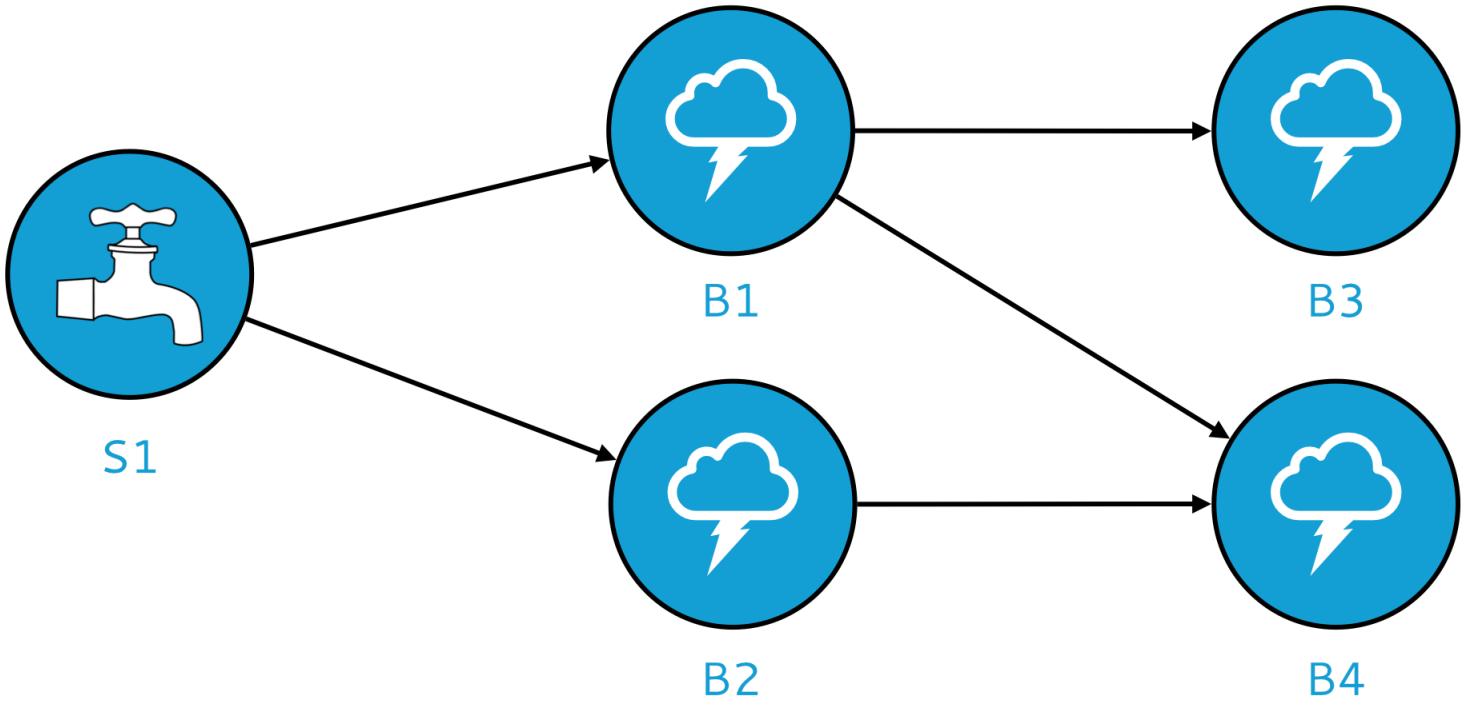
Directed edges indicate communication between vertices

Processing semantics

At most once: without acknowledgments

At least once: with acknowledgements

Spouts and Bolts: Logical Plan



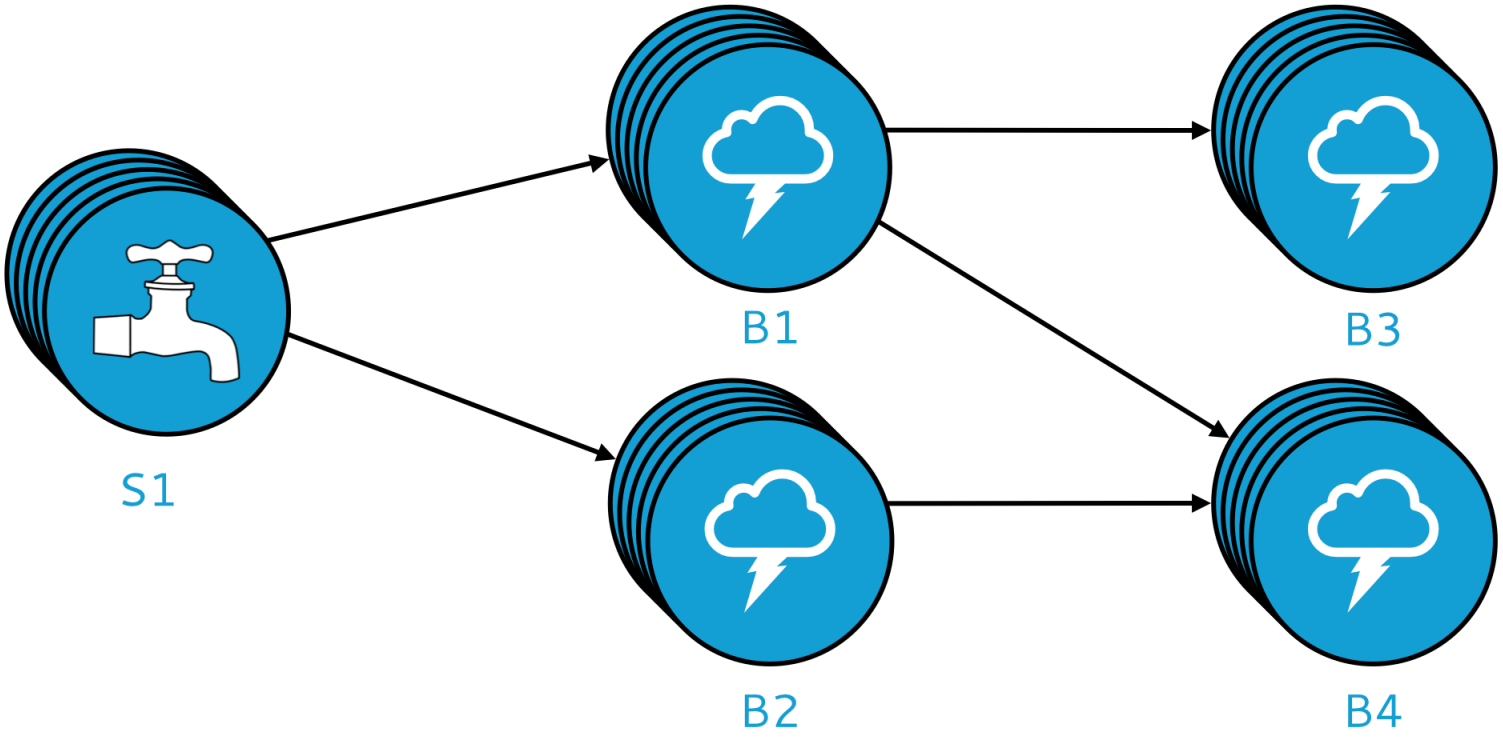
Components

Tuples: data that flow through the topology

Spouts: responsible for emitting tuples

Bolts: responsible for processing tuples

Spouts and Bolts: Physical Plan



Physical plan specifies execution details

Parallelism: how many instances of bolts and spouts to run

Placement of bolts/spouts on machines

…

Stream Groupings

Bolts are executed by multiple instances in parallel

User-specified as part of the topology

When a bolt emits a tuple, where should it go?

Answer: Grouping strategy

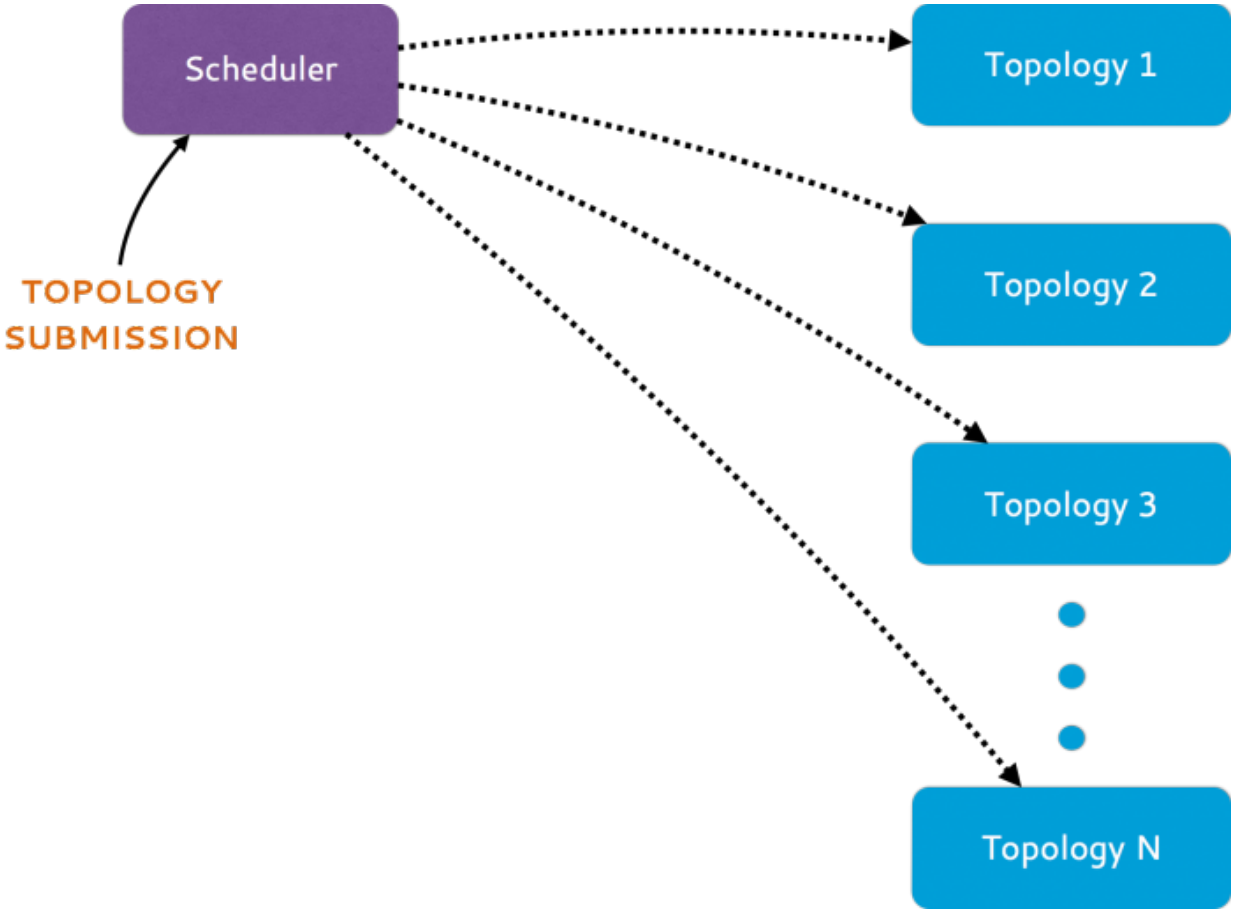
Shuffle grouping: randomly to different instances

Field grouping: based on a field in the tuple

Global grouping: to only a single instance

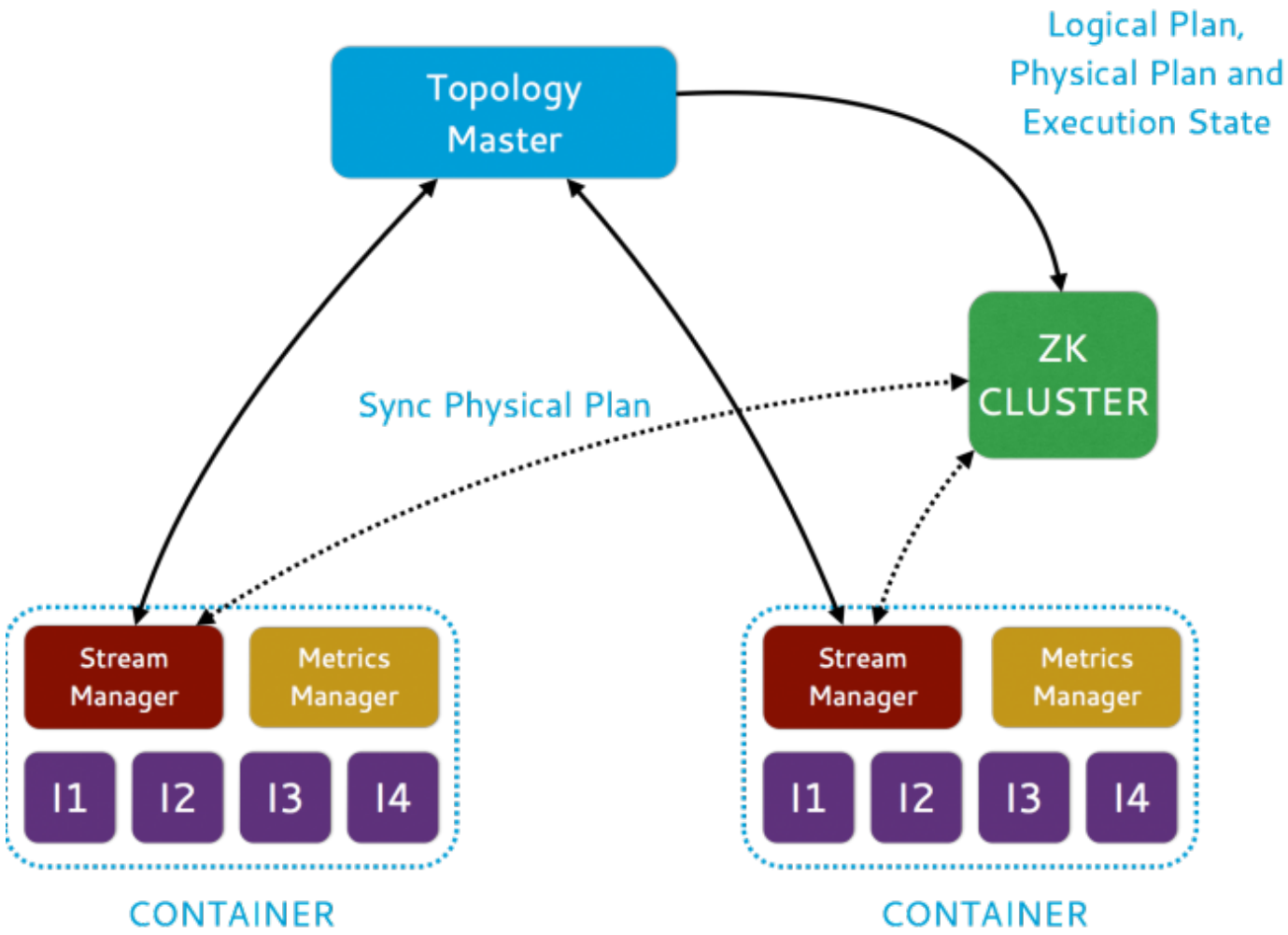
All grouping: to every instance

Heron Architecture



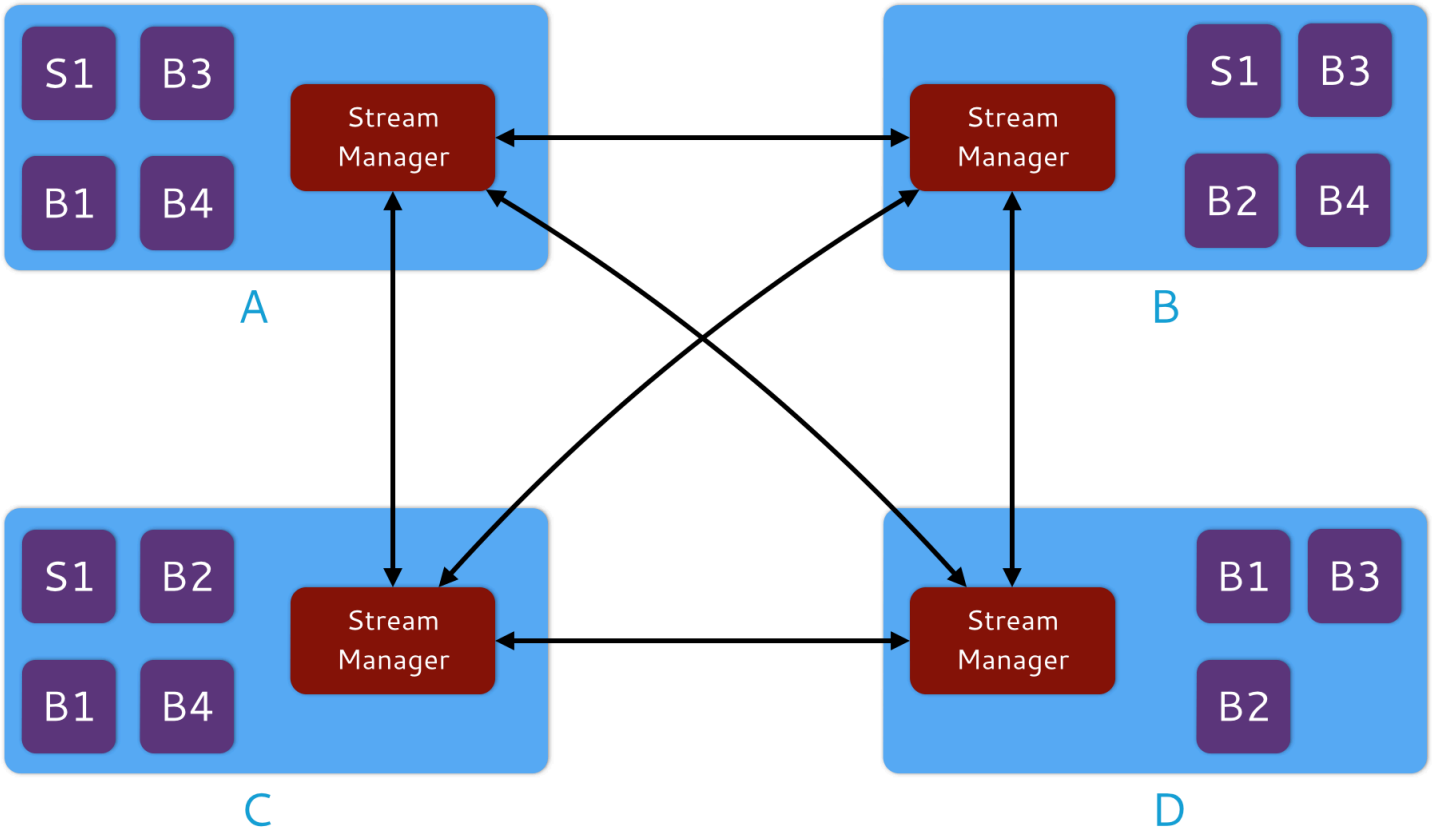
Source: https://blog.twitter.com/2015/flying-faster-with-twitter-heron

Heron Architecture



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

Heron Architecture



Stream Manager

Manages routing tuples between spouts and bolts

Responsible for applying backpressure

Some me some code!

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("word", new WordSpout(), parallelism);
builder.setBolt("consumer", new ConsumerBolt(), parallelism)
    .fieldsGrouping("word", new Fields("word"));

Config conf = new Config();
// Set config here
// ...

StormSubmitter.submitTopology("my topology”, conf,
    builder.createTopology());
```

Some me some code!

```
public static class WordSpout extends BaseRichSpout {
  @Override
  public void declareOutputFields(
      OutputFieldsDeclarer outputFieldsDeclarer) {
    outputFieldsDeclarer.declare(new Fields("word"));
  }

  @Override
  public void nextTuple() {
    // ...
    collector.emit(word);
  }
}
```

Some me some code!

```
public static class ConsumerBolt extends BaseRichBolt {
  private OutputCollector collector;
  private Map<String, Integer> countMap;

  public void prepare(Map map, TopologyContext
      topologyContext, OutputCollector outputCollector) {
    collector = outputCollector;
    countMap = new HashMap<String, Integer>();
  }

  @Override
  public void execute(Tuple tuple) {
    String key = tuple.getString(0);
    if (countMap.get(key) == null) {
      countMap.put(key, 1);
    } else {
      Integer val = countMap.get(key);
      countMap.put(key, ++val);
    }
  }
}
```

What's the issue?

Remember this?

Space: The Final Frontier

Data streams are potentially infinite

Unbounded space!

How do we get around this?

Throw old data away

Accept some approximation

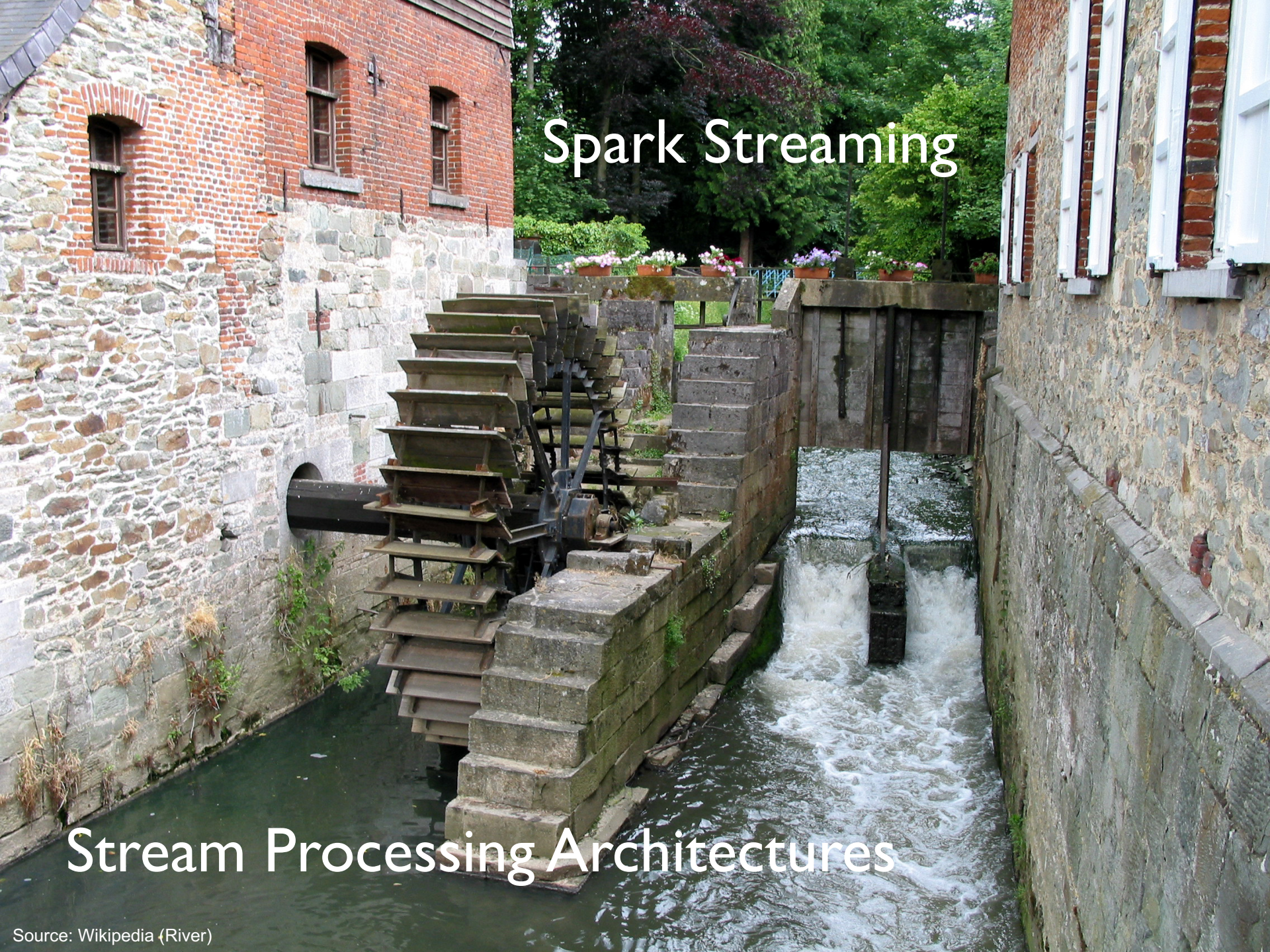
General techniques

Sampling

Hashing

So what does Storm/Heron actually provide?

Source: Wikipedia (Plumbing)



Source: Wikipedia (River)

Hmm, I gotta get in on this streaming thing…

But I got all this batch processing framework that I gotta lug around.

I know: we'll just chop the stream into little pieces, pretend each is an RDD, and we're on our merry way!

Want real-time stream processing? I got your back.

I've got the most intuitive implementation: a computation graph!

APACHE
STORM™
Distributed • Resilient • Real-time

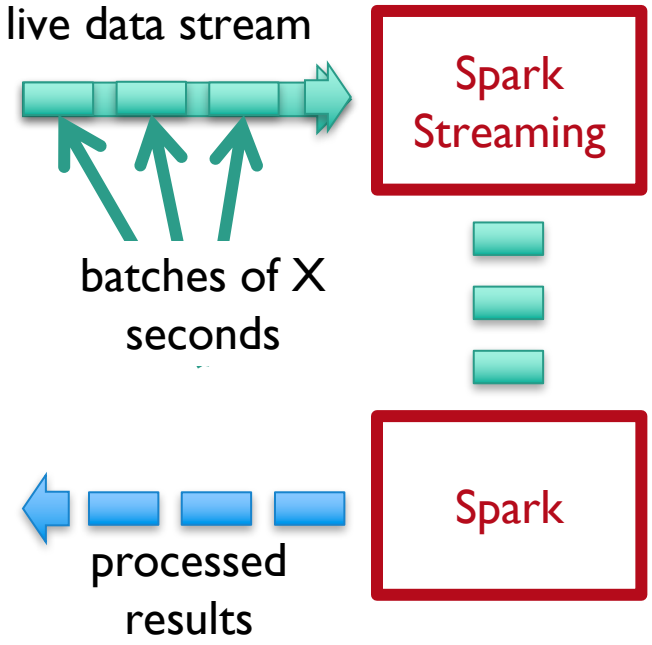
Spark Streaming: Discretized Streams

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

Chop up the stream into batches of *X* seconds

Process as RDDs!

Return results in batches



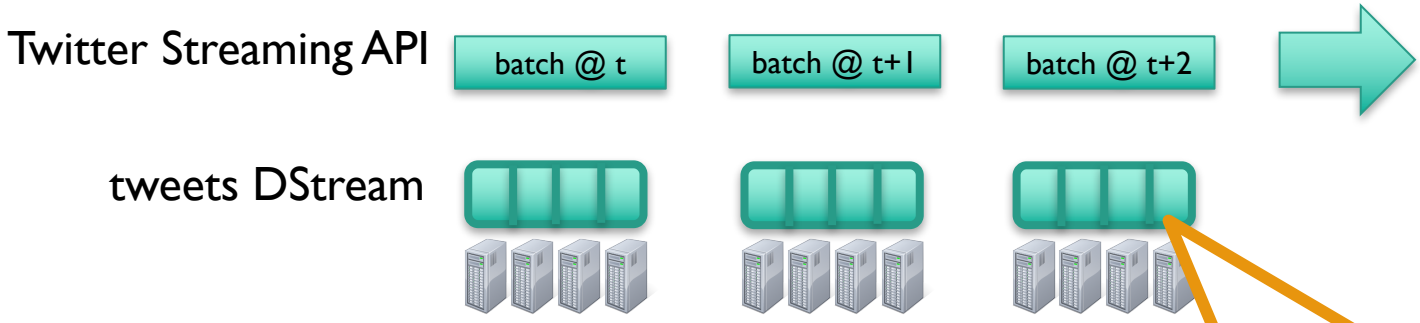
Typical batch window ~1s

Source: All following Spark Streaming slides by Tathagata Das

Example: Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data



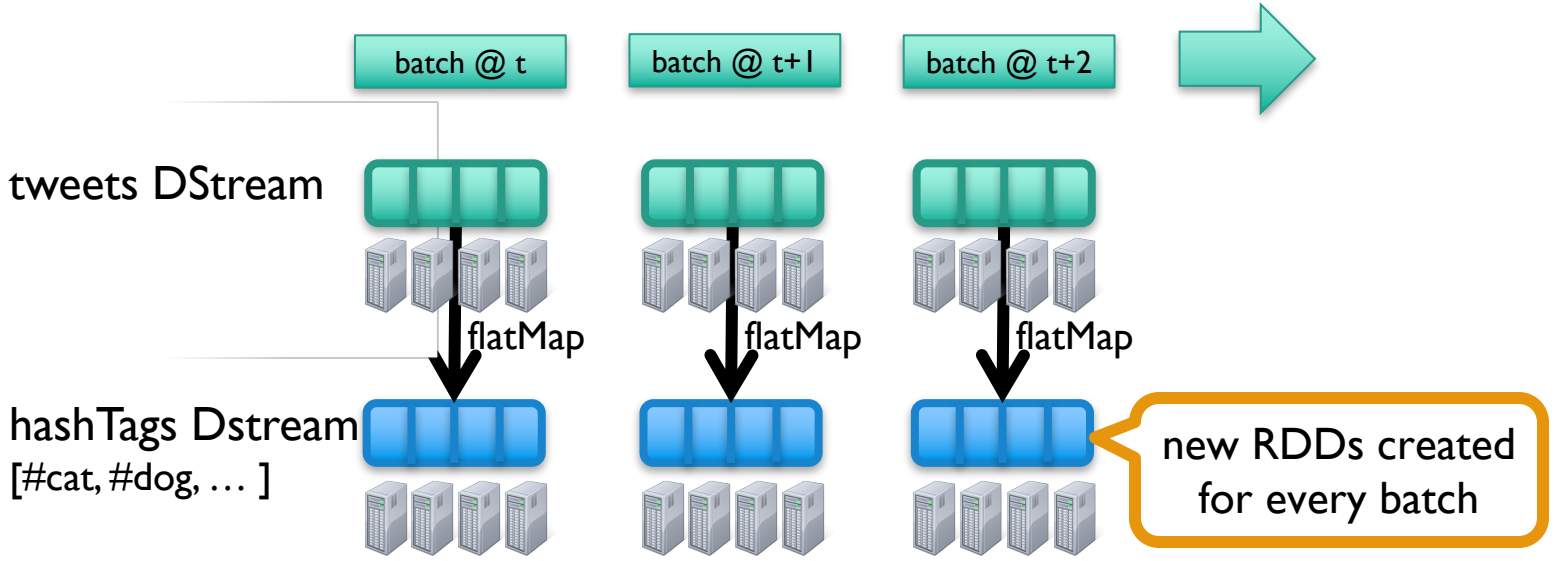
stored in memory as an RDD
(immutable, distributed)

Example: Get hashtags from Twitter

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

new DStream

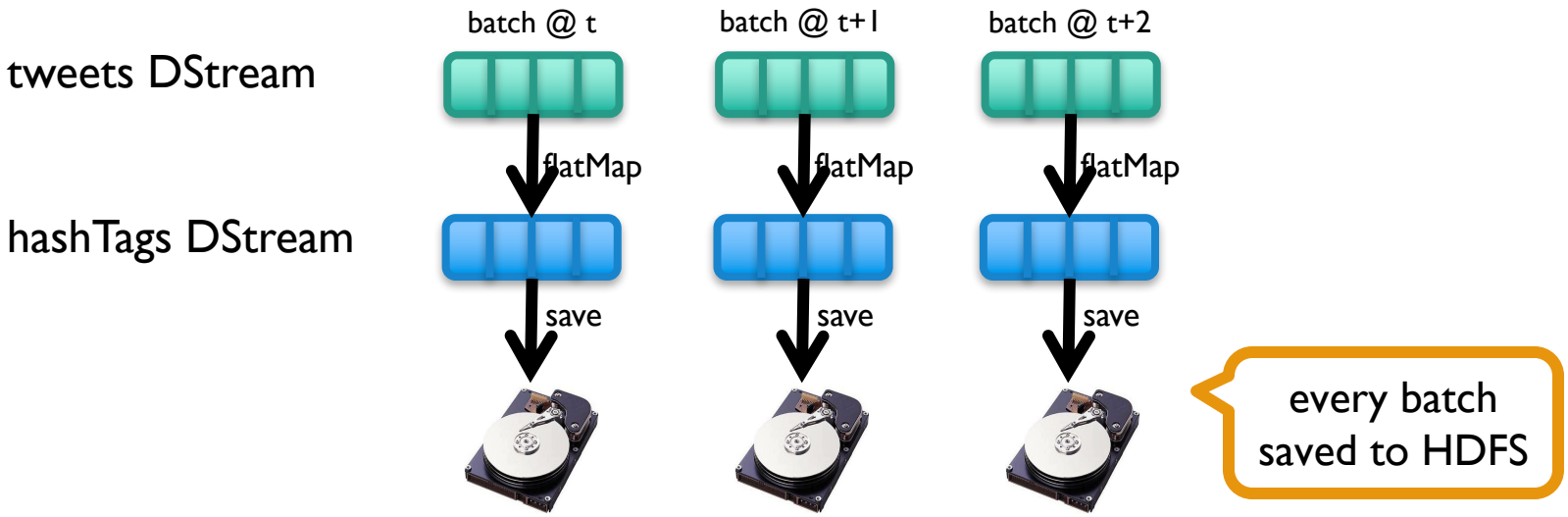
transformation: modify data in one Dstream to create another DStream



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

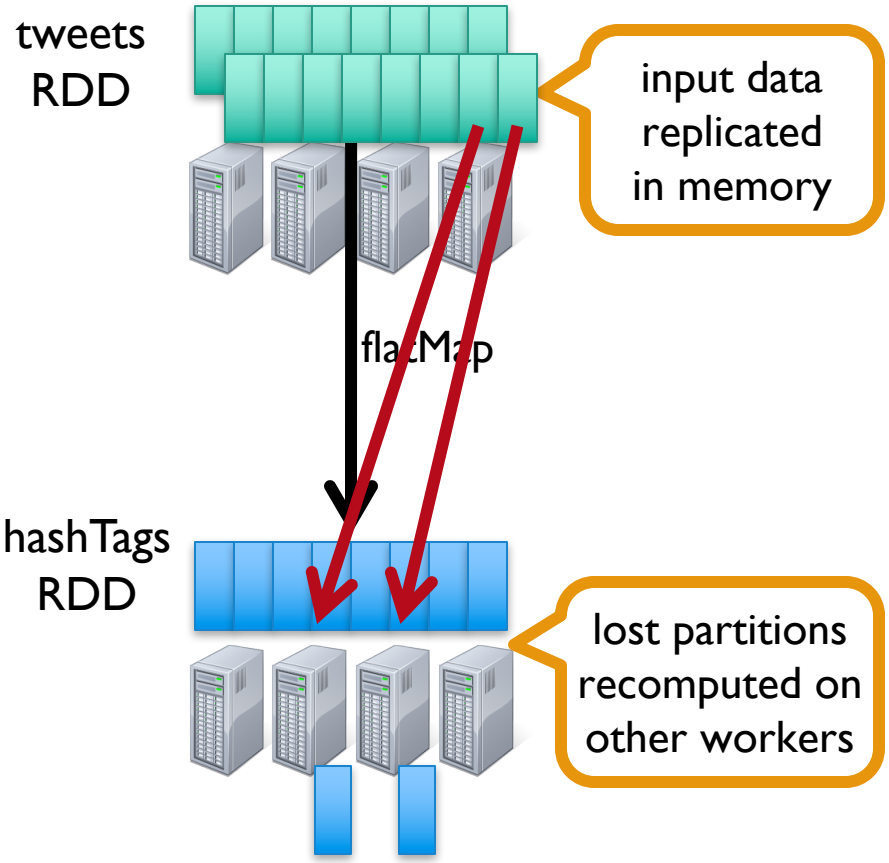


Fault Tolerance

Bottom line: they're just RDDs!

Fault Tolerance

Bottom line: they're just RDDs!



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, …

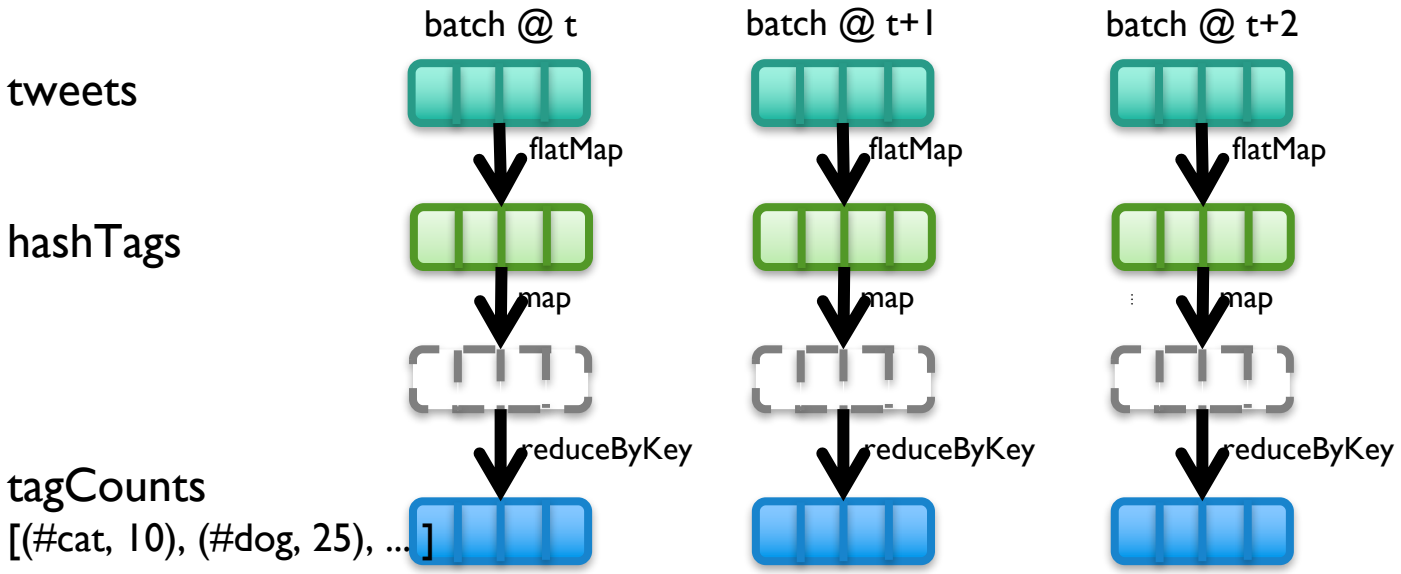
Output Operations – send data to external entity

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()
```

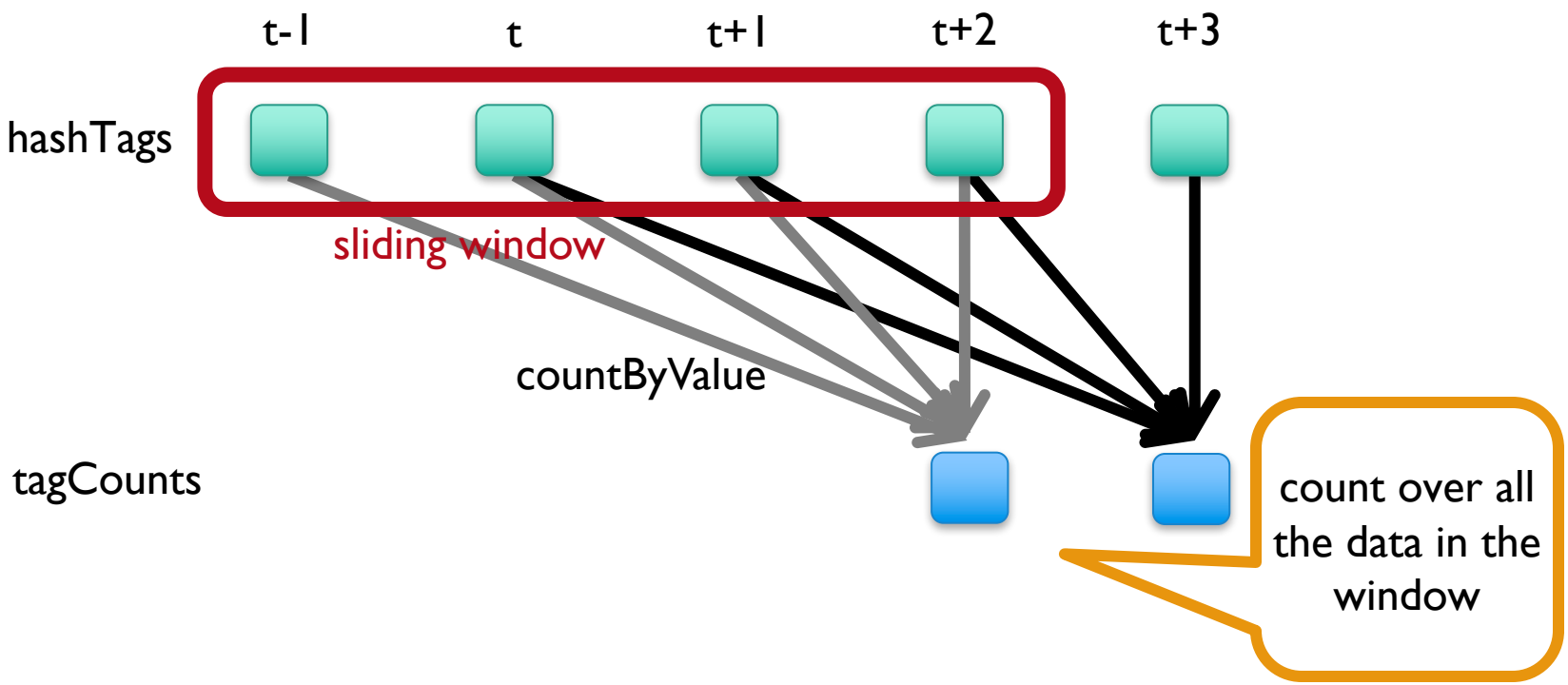


Example: Count the hashtags over last 10 mins

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

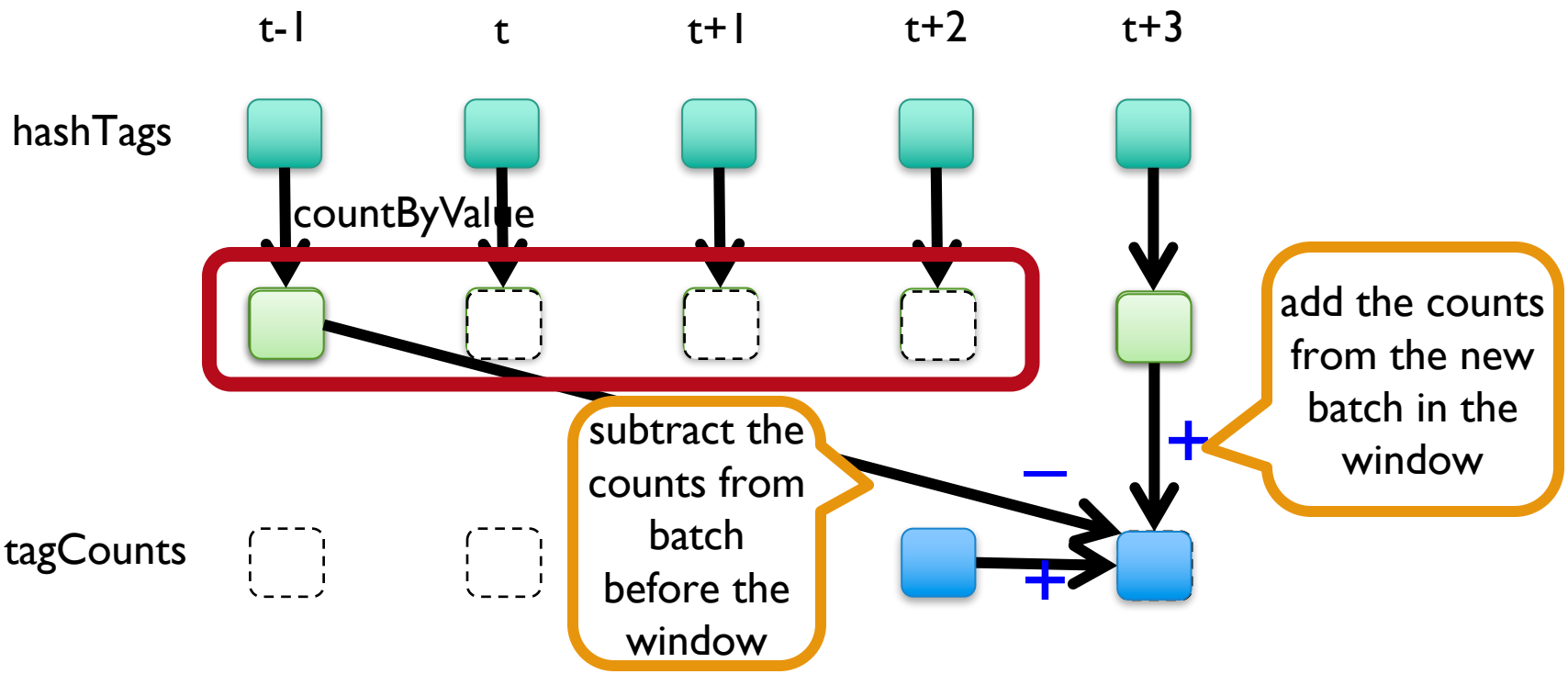
Example: Count the hashtags over last 10 mins

```
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

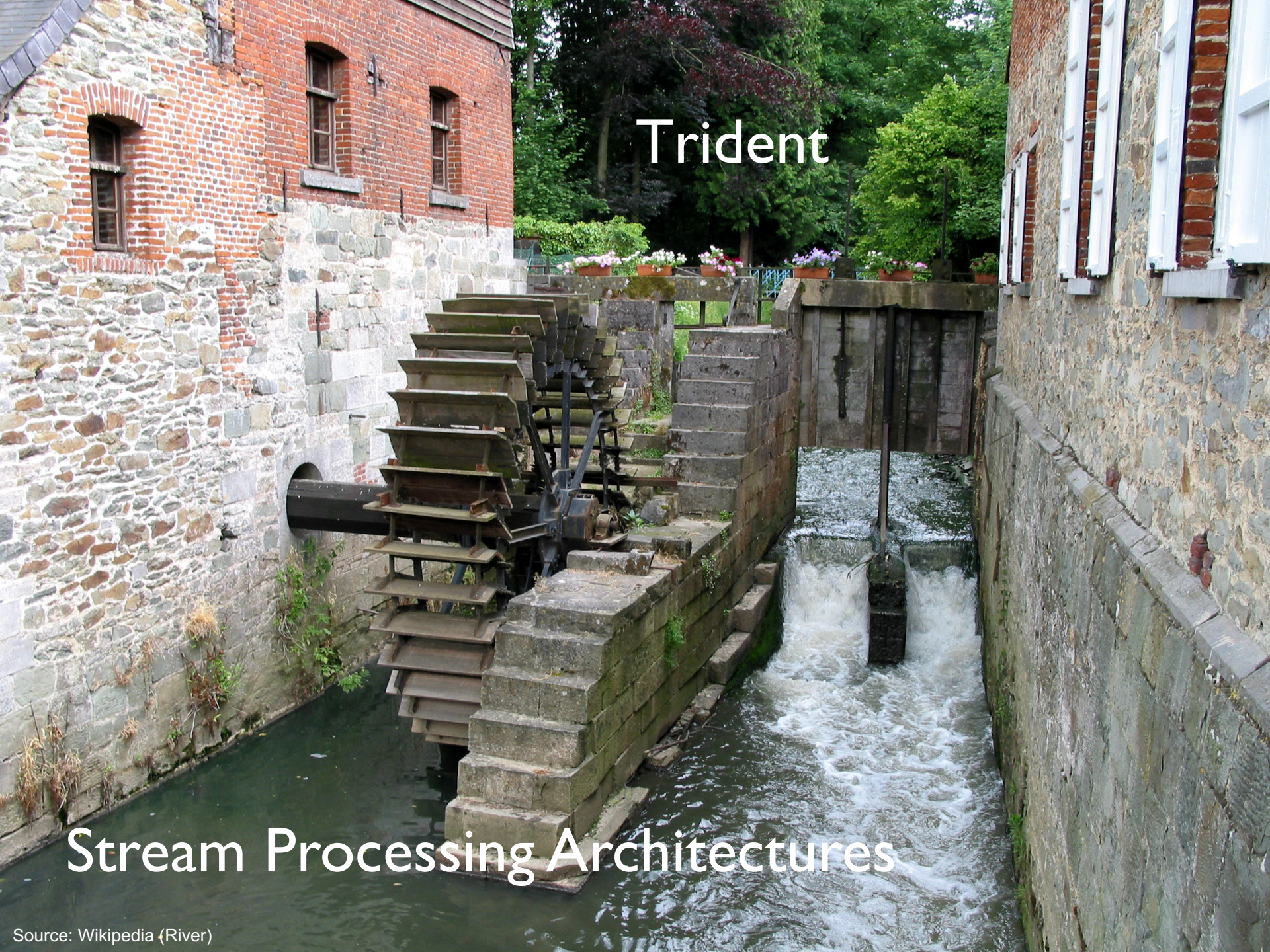
Incremental counting generalizes to many reduce operations

Need a function to "inverse reduce" ("subtract" for counting)

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

```
val tagCounts = hashtags
    .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(1))
```

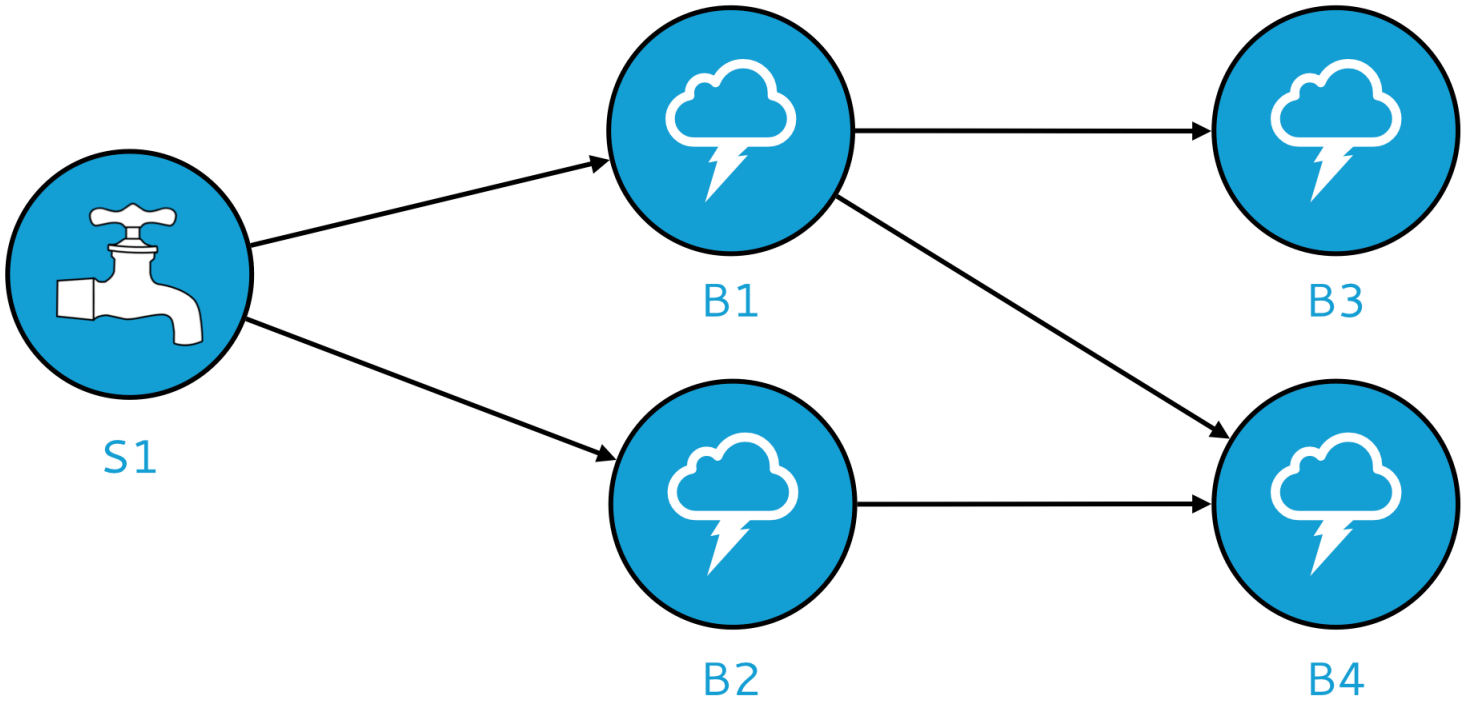
Final question: what's the processing semantics of Spark Streaming?



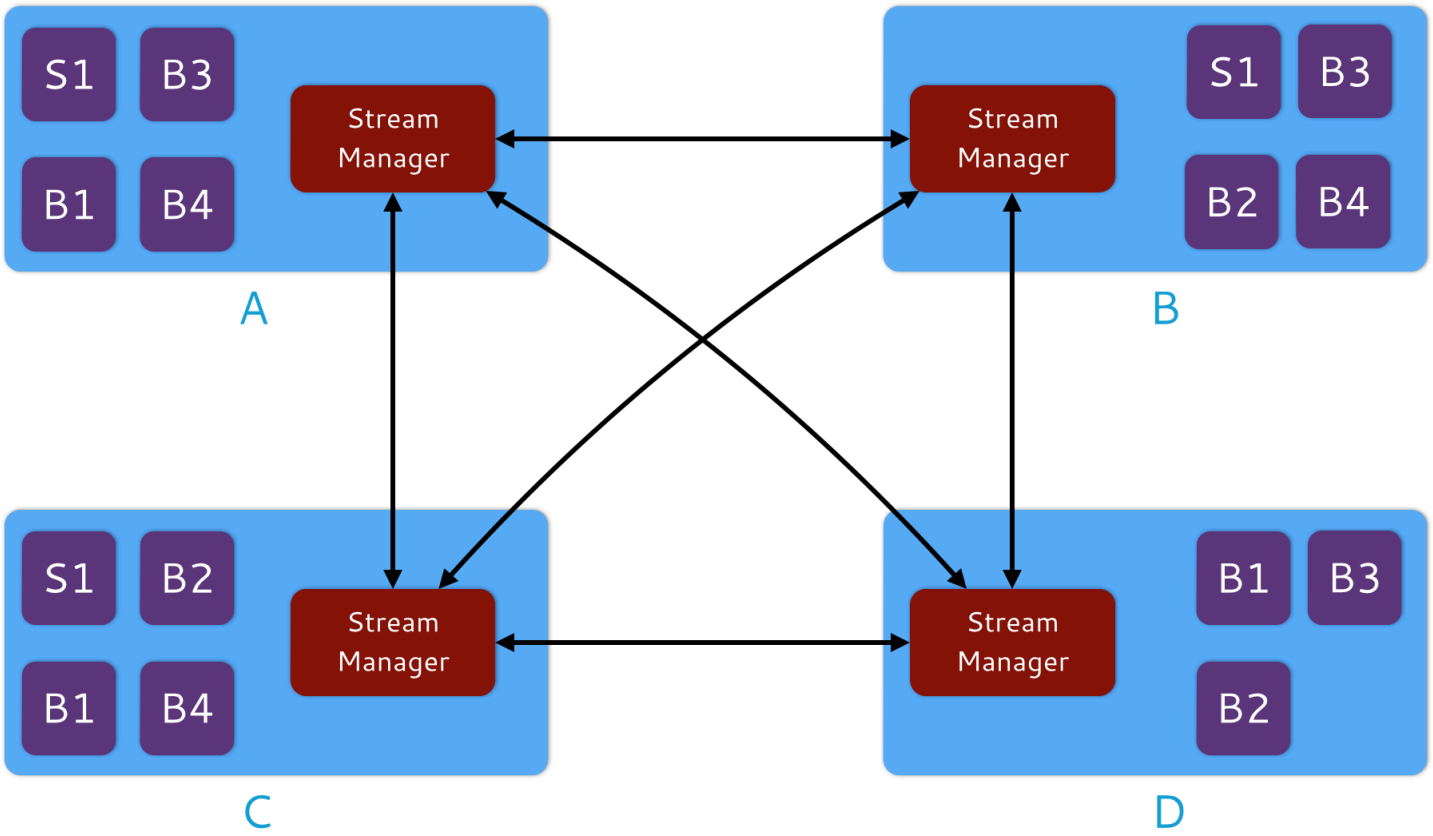
Stream Processing Architectures

Source: Wikipedia (River)

Spouts and Bolts: Logical Plan



Heron Architecture



So what does Storm/Heron actually provide?

Source: Wikipedia (Plumbing)

We need something like this!

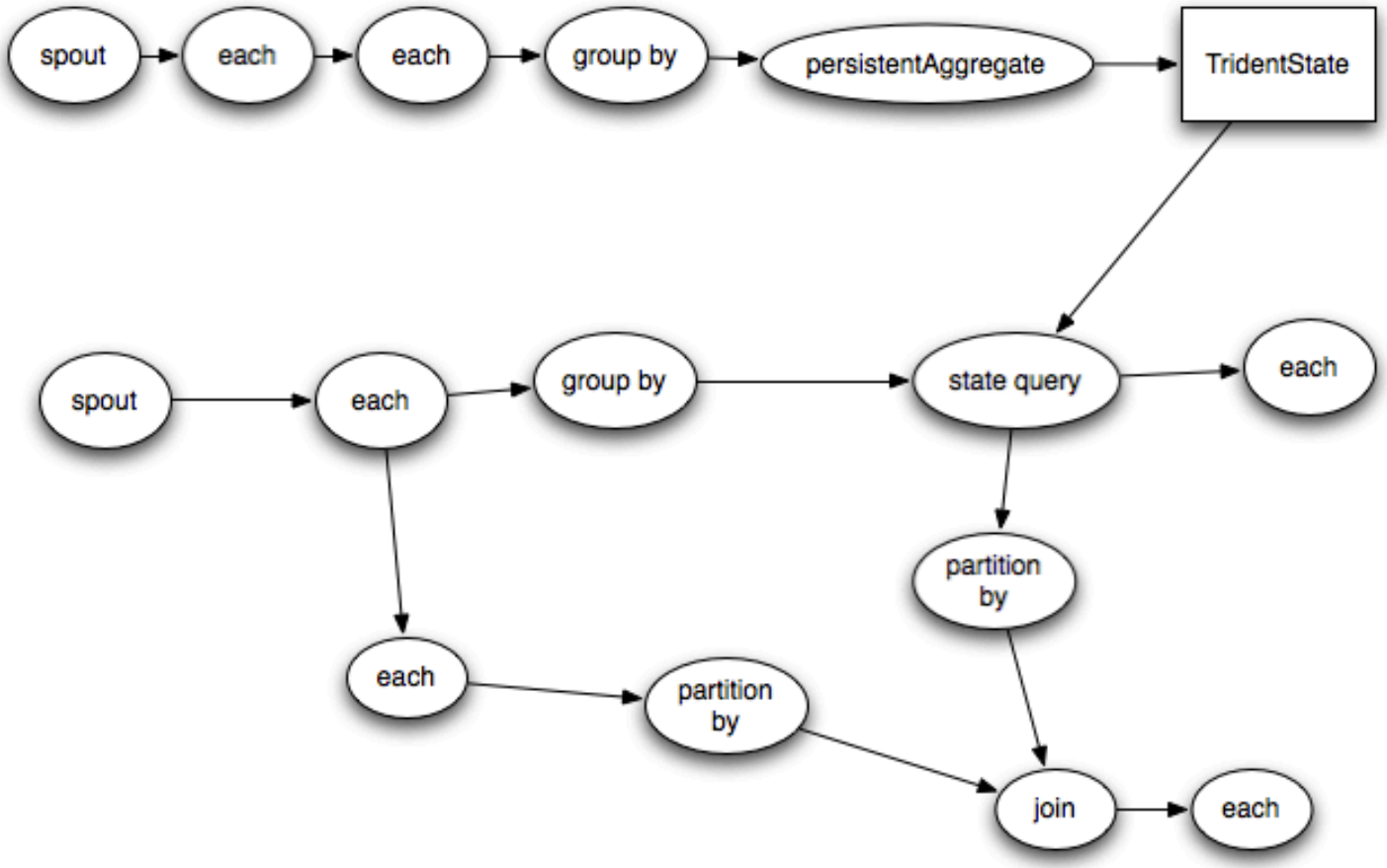
```
TridentTopology topology = new TridentTopology();

TridentState wordCounts =
    topology.newStream("spout1", spout)
      .each(new Fields("sentence"), new Split(),
          new Fields("word"))
      .groupBy(new Fields("word"))
      .persistentAggregate(new MemoryMapState.Factory(),
           new Count(), new Fields("count"))
```

Also, would be nice to have exactly once semantics…

The answer: Trident!

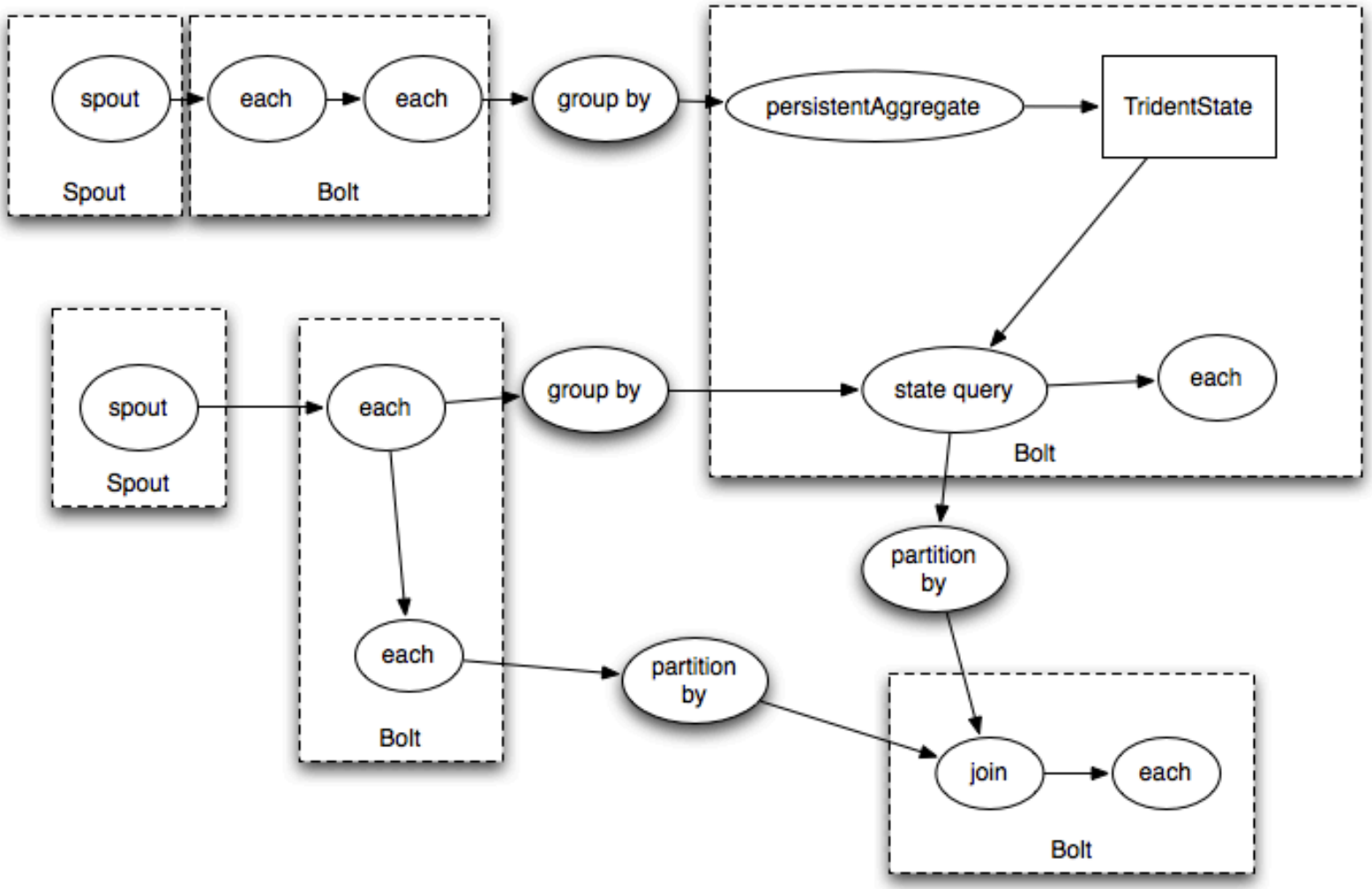
Trident

A high-level query graph…



Trident

Compiles into a Storm topology



Source: Wikipedia (Cake)

Necessary Ingredients

Some reliable method of delivering data in real time

Kafka has become the *de facto* solution

Some underlying execution engine

Storm/Heron

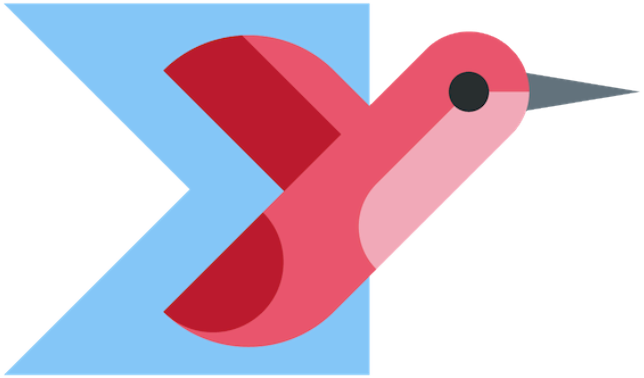
Spark's physical execution engine

Some high-level API or DSL

Trident for Storm/Heron

Spark Streaming for Spark

Source: Wikipedia (River)



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

Boykin, Ritchie, O'Connell, and Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. PVLDB 7(13):1441-1451, 2014.

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 , $\oplus$)

$\oplus : M \times M \rightarrow M$, 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

ε 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 ⊕ b ⊕ c ⊕ d ⊕ e ⊕ f)

(((((a ⊕ b) ⊕ c) ⊕ d) ⊕ e) ⊕ f)

((a ⊕ b ⊕ c) ⊕ (d ⊕ e ⊕ f))

Batch = Hadoop

Online = Storm

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

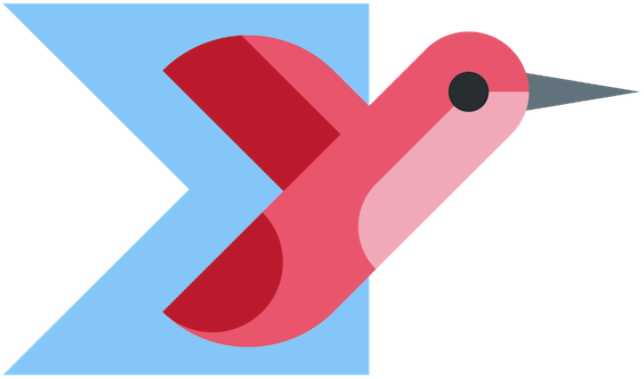
write to HDFS

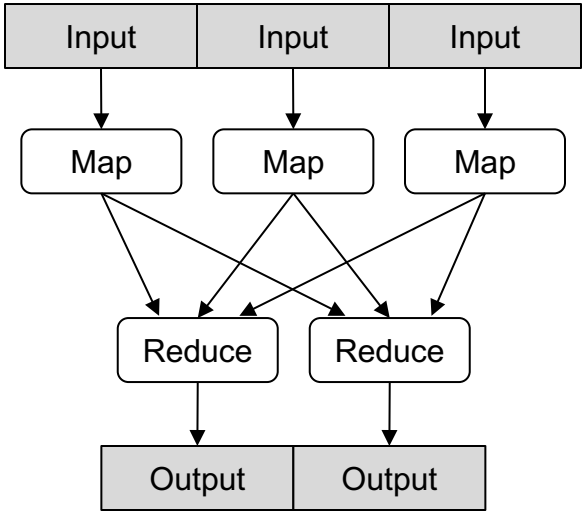
Run on Storm

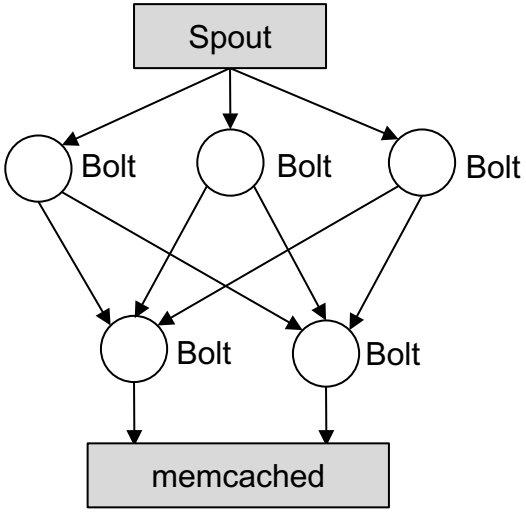
```
Storm.run {
  wordCount[Storm](
    new TweetSpout(),
    new MemcacheStore[String, Long]
  )
}
```

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?

“Interesting” monoids

Bloom filters (set membership)

HyperLogLog counters (cardinality estimation)

Count-min sketches (event counts)

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

Cheat Sheet

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

Example: Count queries by hour

Exact with hashmaps

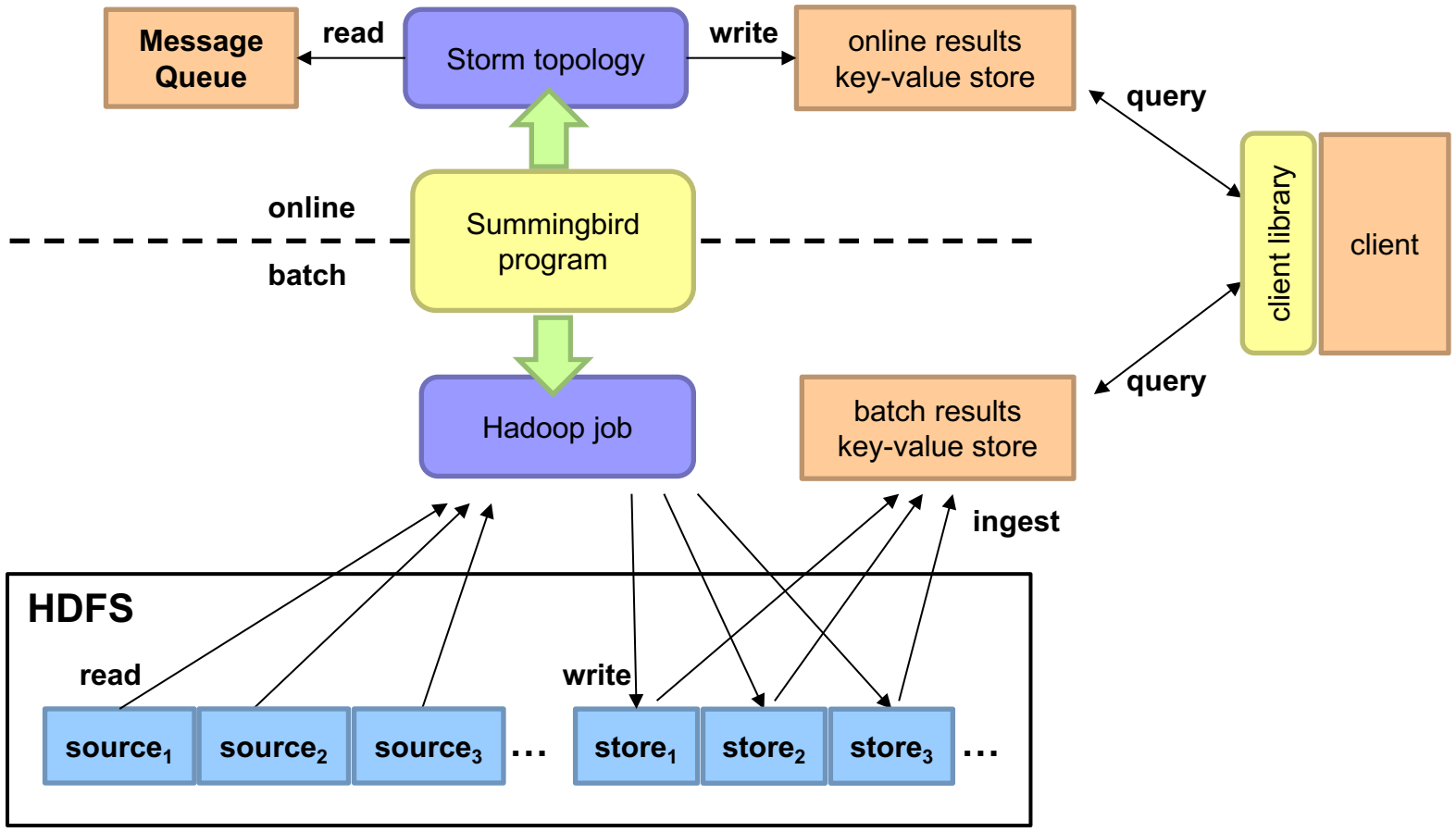
```
def wordCount[P <: Platform[P]]
  (source: Producer[P, Query],
   store: P#Store[Long, Map[String, Long]]) =
   source.flatMap { query =>
      (query.getHour, Map(query.getQuery -> 1L))
    }.sumByKey(store)
```

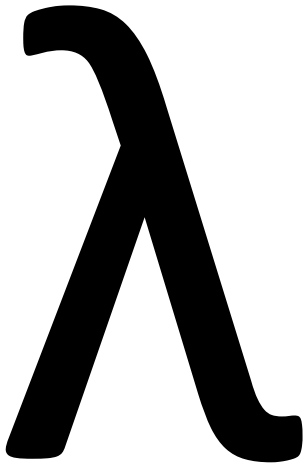
Approximate with CMS

```
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

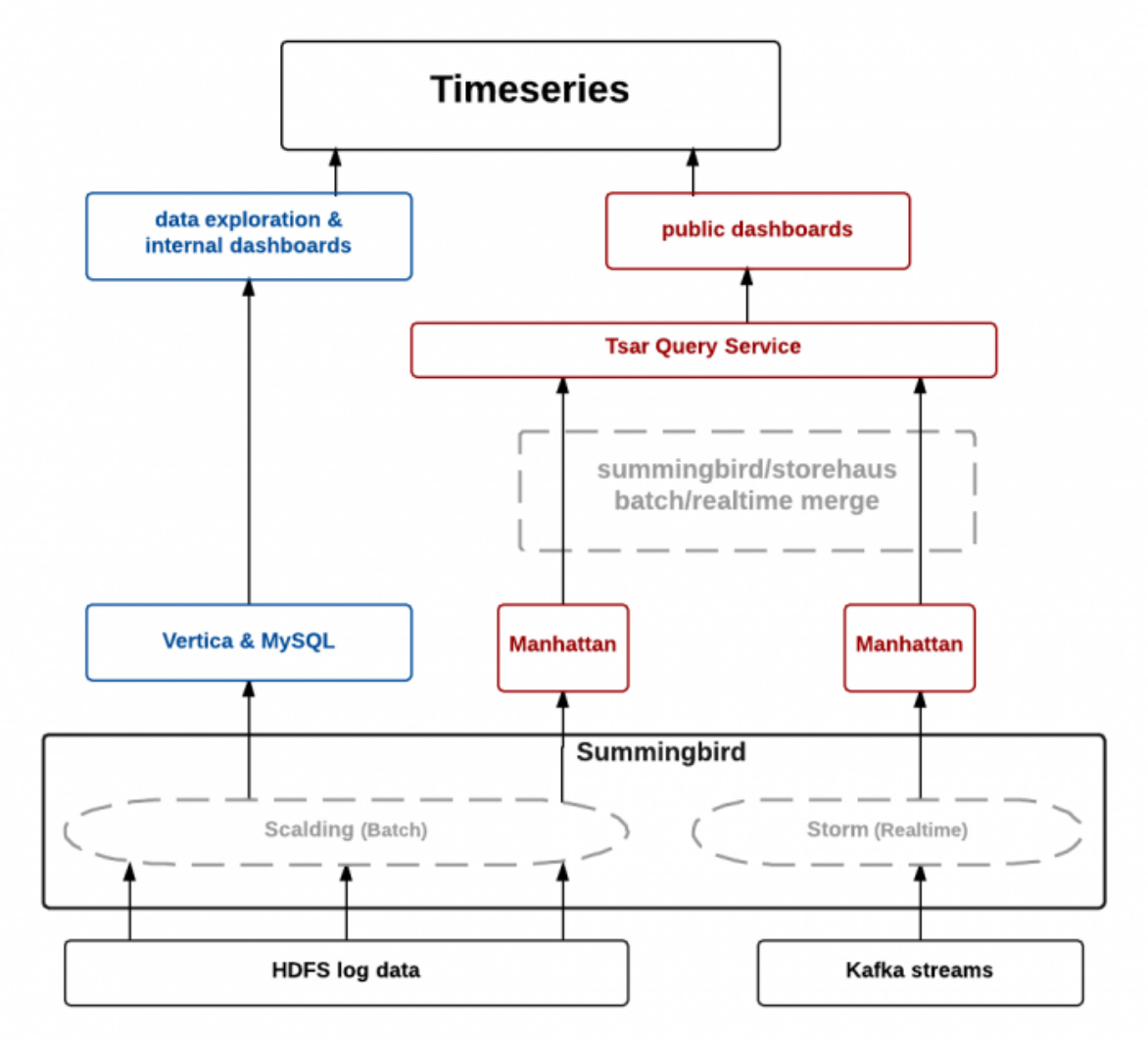
Example: count historical clicks and clicks in real time



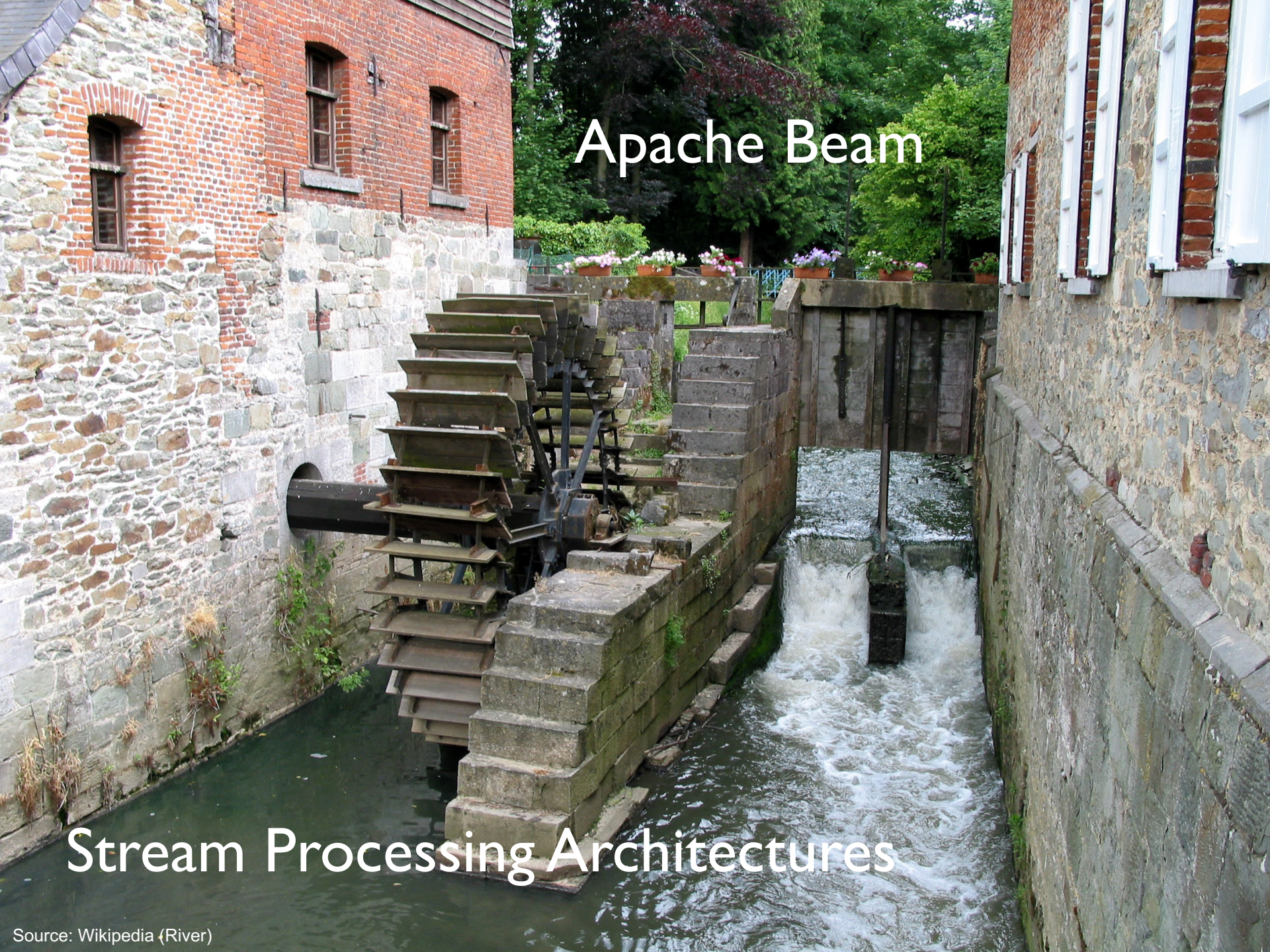


(I hate this.)

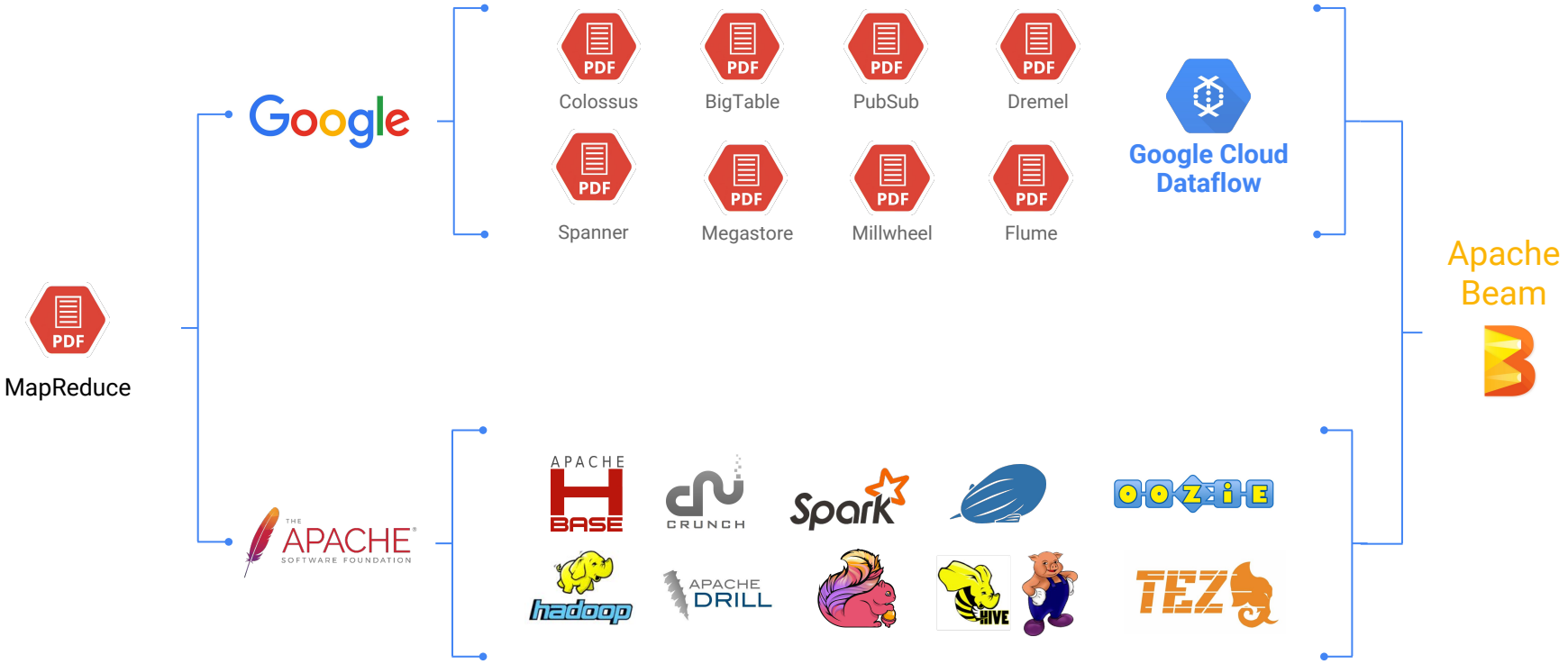
TSAR, a TimeSeries AggregatoR!



Source: https://blog.twitter.com/2014/tsar-a-timeseries-aggregator



Source: Wikipedia (River)

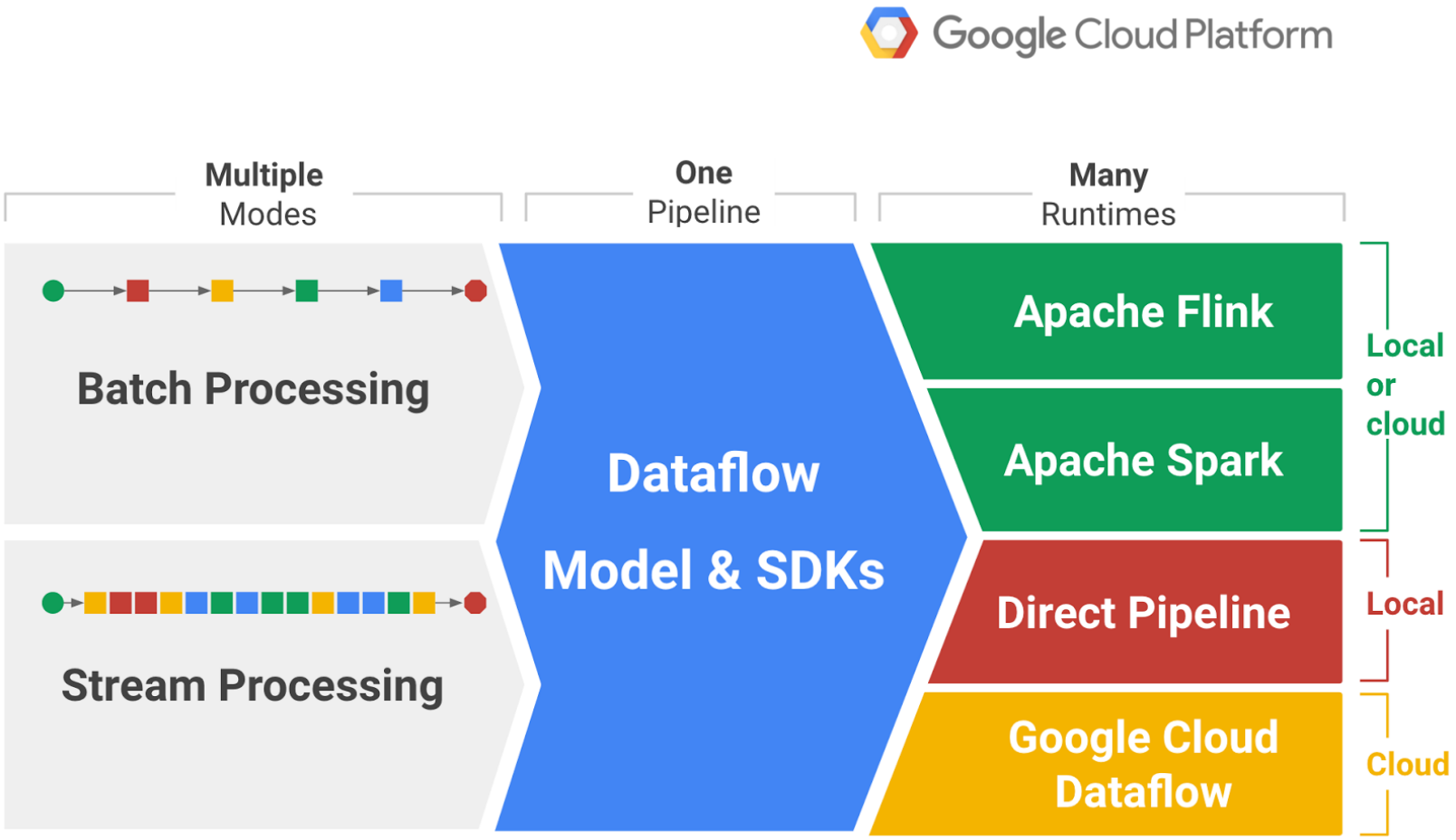

MapReduce
Google
Colossus
BigTable
PubSub
Dremel
Spanner
Megastore
Millwheel
Flume
Google Cloud Dataflow
APACHE SOFTWARE FOUNDATION
APACHE HBASE
CRUNCH
Spark
OOZIE
hadoop
APACHE DRILL
HIVE
TEZ
Apache Beam

Apache Beam

2015: Google releases Cloud Dataflow

2016: Google donates API and SDK to Apache
to become Apache Beam

The Vision



Source: https://cloudplatform.googleblog.com/2016/01/Dataflow-and-open-source-proposal-to-join-the-Apache-Incubator.html

The Beam Model

What results are computed?

Where in event time are the results computed?

When in processing time are the results materialized?

How do refinements of results relate?

Programming Model

Fundamental distinction: bounded vs. unbounded datasets

Unbounded datasets need windowing

Core Concepts

Pipeline: a data processing task

PCollection: a distributed dataset that a pipeline operates on

Transform: a data processing operation

Source: for reading data

Sink: for writing data

Processing semantics: exactly once!

Processing Bounded Datasets

```
Pipeline p = Pipeline.create(options);

p.apply(TextIO.Read.from("gs://your/input/"))
 .apply(FlatMapElements.via((String word) ->
     Arrays.asList(word.split("[^a-zA-Z']+"))))
 .apply(Filter.by((String word) -> !word.isEmpty()))
 .apply(Count.perElement())
 .apply(MapElements.via((KV<String, Long> wordCount) ->
    wordCount.getKey() + ": " + wordCount.getValue()))
 .apply(TextIO.Write.to("gs://your/output/"));
```

Event Time vs. Processing Time

What's the distinction?

Watermark: System's notion when all data in a window is expected to arrive

Where in event time are the results computed?

When in processing time are the results materialized?

How do refinements of results relate?

Trigger: a mechanism for declaring when output of a window should be materialized

Default trigger "fires" at watermark

Late and early firings: multiple "panes" per window

Event Time vs. Processing Time

What's the distinction?

Watermark: System's notion when all data in a window is expected to arrive

Where in event time are the results computed?

When in processing time are the results materialized?

How do refinements of results relate?

How do multiple "firings" of a window (i.e., multiple "panes") relate?

Options: Discarding, Accumulating, Accumulating & retracting

Processing Bounded Datasets

```
Pipeline p = Pipeline.create(options);

p.apply(TextIO.Read.from("gs://your/input/"))
 .apply(FlatMapElements.via((String word) ->
     Arrays.asList(word.split("[^a-zA-Z']+"))))
 .apply(Filter.by((String word) -> !word.isEmpty()))
 .apply(Count.perElement())
 .apply(MapElements.via((KV<String, Long> wordCount) ->
    wordCount.getKey() + ": " + wordCount.getValue()))
 .apply(TextIO.Write.to("gs://your/output/"));
```

Processing Unbounded Datasets

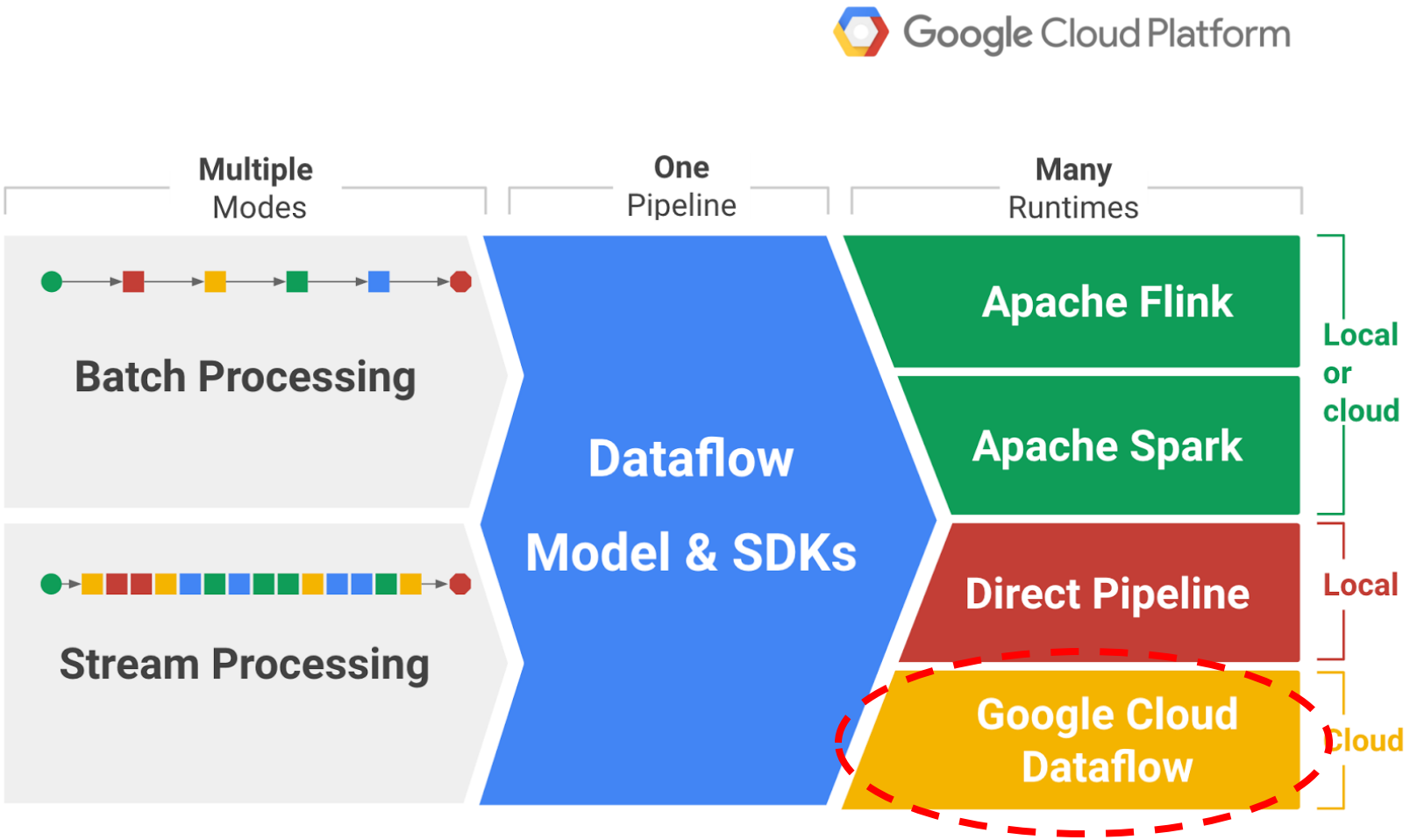
```
Pipeline p = Pipeline.create(options);

p.apply(KafkaIO.read("tweets")
    .withTimestampFn(new TweetTimestampFunction())
    .withWatermarkFn(kv ->
        Instant.now().minus(Duration.standardMinutes(2))))
 .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2)))
    .triggering(AtWatermark()
        .withEarlyFirings(AtPeriod(Duration.standardMinutes(1)))
        .withLateFirings(AtCount(1)))
    .accumulatingAndRetractingFiredPanes())
 .apply(FlatMapElements.via((String word) ->
     Arrays.asList(word.split("[^a-zA-Z']+"))))
 .apply(Filter.by((String word) -> !word.isEmpty()))
 .apply(Count.perElement())
 .apply(KafkaIO.write("counts"))
```

Where in event time?

When in processing time?

How do refines relate?

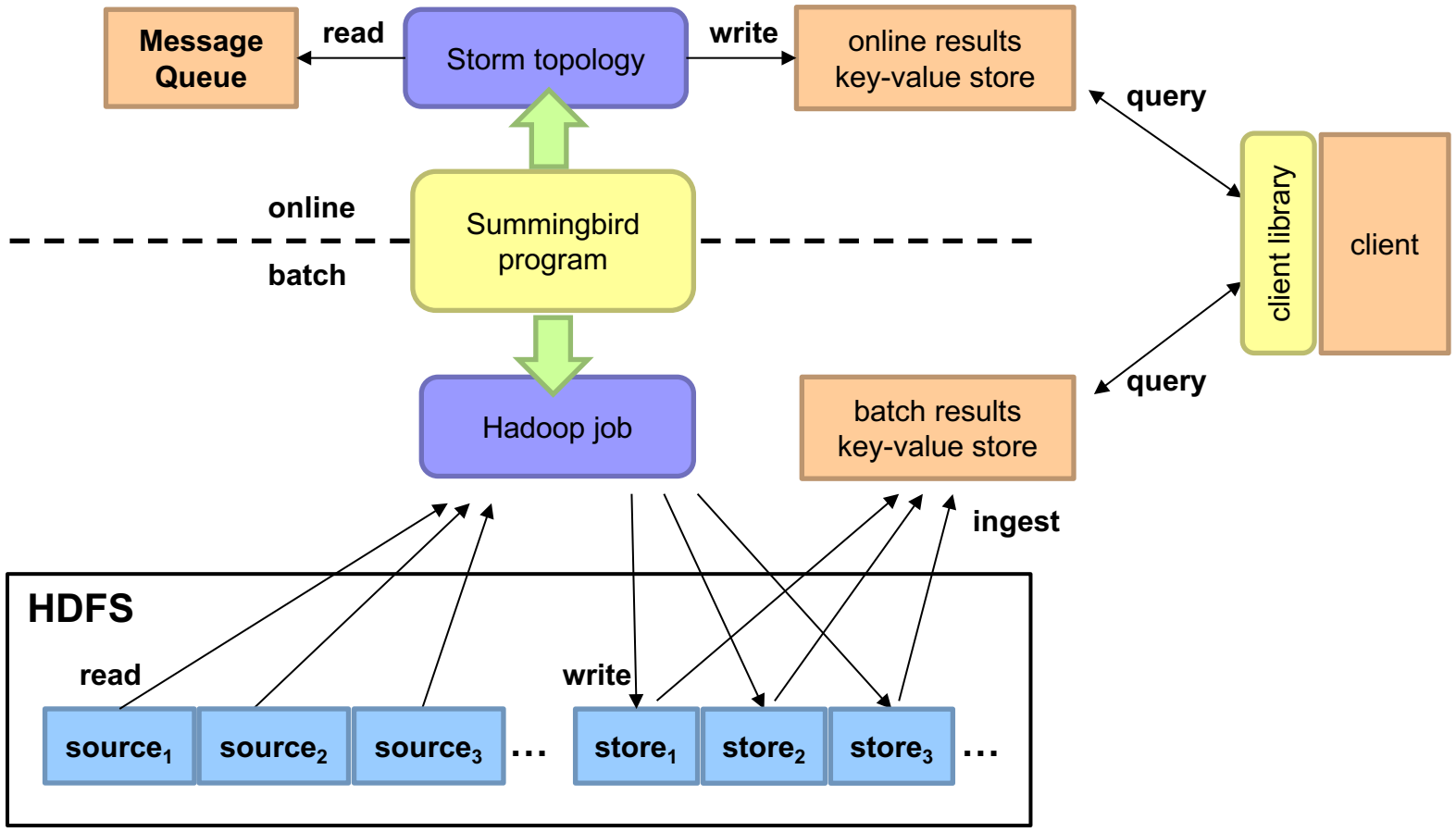
The Vision


Google Cloud Platform
Multiple
Modes
One
Pipeline
Many
Runtimes
Batch Processing
Stream Processing
Dataflow
Model & SDKs
Apache Flink
Apache Spark
Direct Pipeline
Google Cloud
Dataflow
Local
or
cloud
Local
Cloud

versus

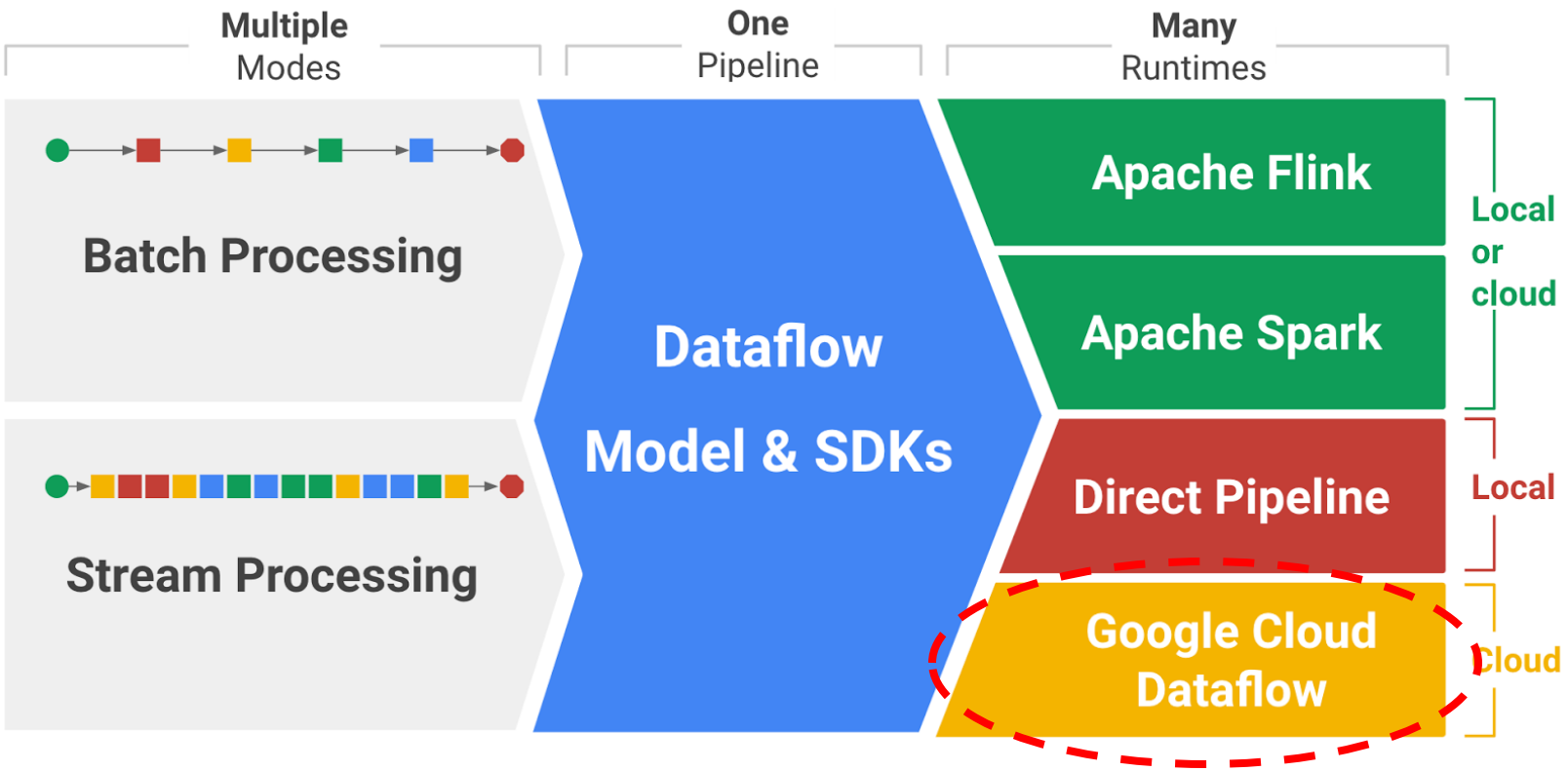
Hybrid Online/Batch Processing

Example: count historical clicks and clicks in real time



The Central Value Proposition

Our stream processing engine is so fast that to do "batch" processing, just stream over bounded datasets.



Source: https://cloudplatform.googleblog.com/2016/01/Dataflow-and-open-source-proposal-to-join-the-Apache-Incubator.html

K

(I hate this too.)

Everything is streaming!

Source: Wikipedia (Cake)

Source: flickr (https://www.flickr.com/photos/39414578@N03/16042029002)