Data-Intensive Distributed Computing

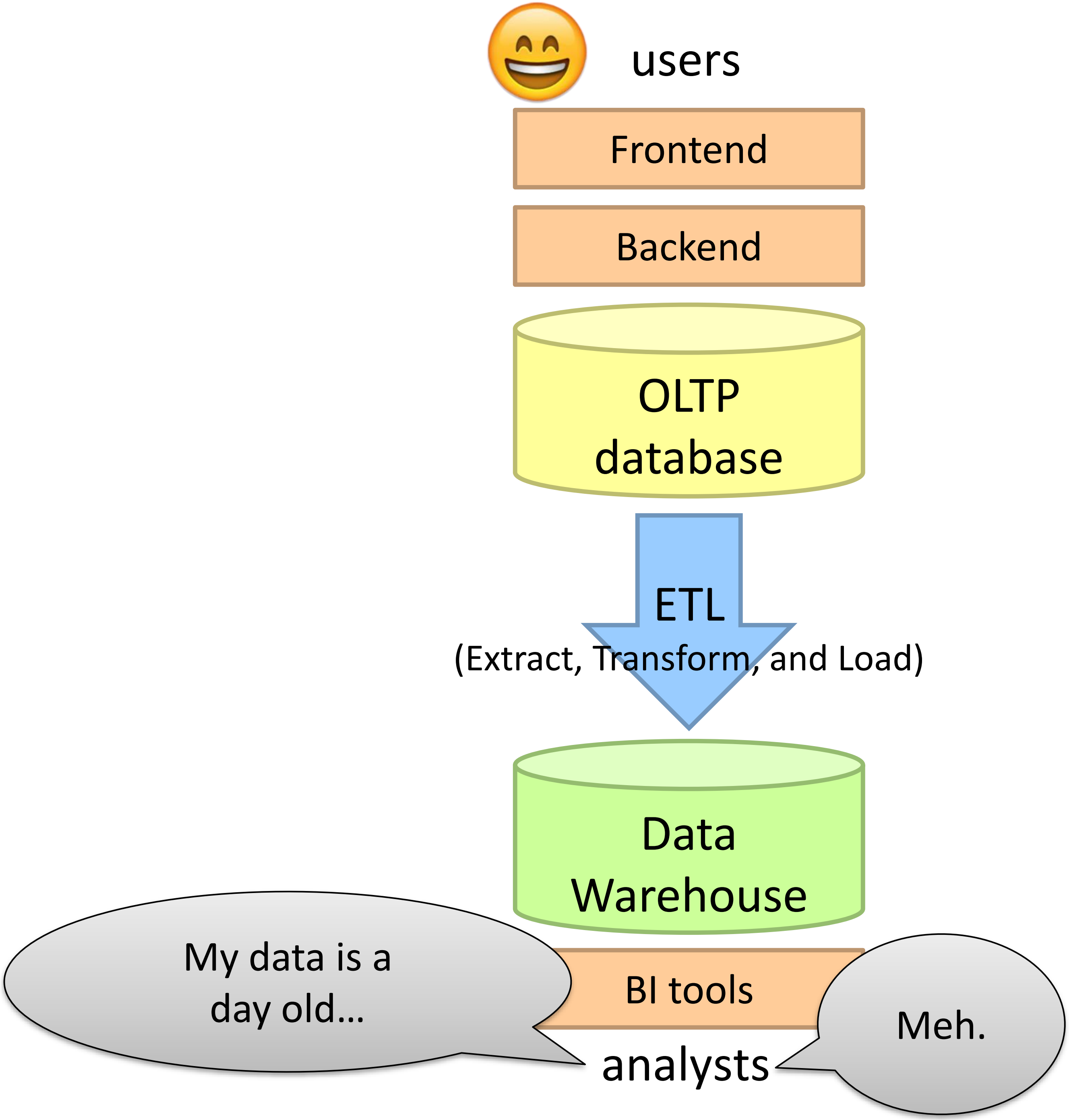
CS 431/631 451/651 (Fall 2019)

Part 9: Real-Time Data Analytics (1/2)

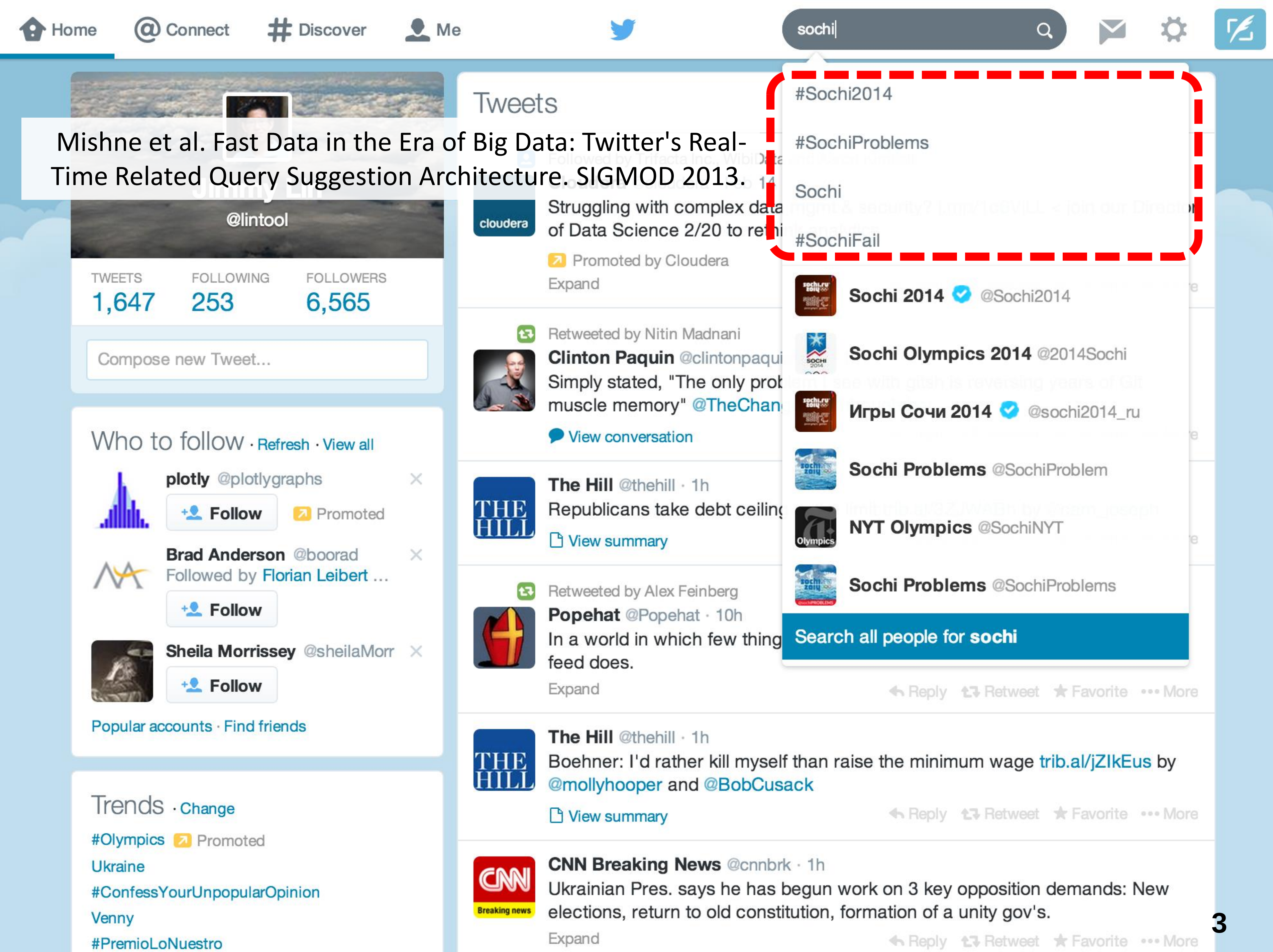
November 26, 2019

Ali Abedi

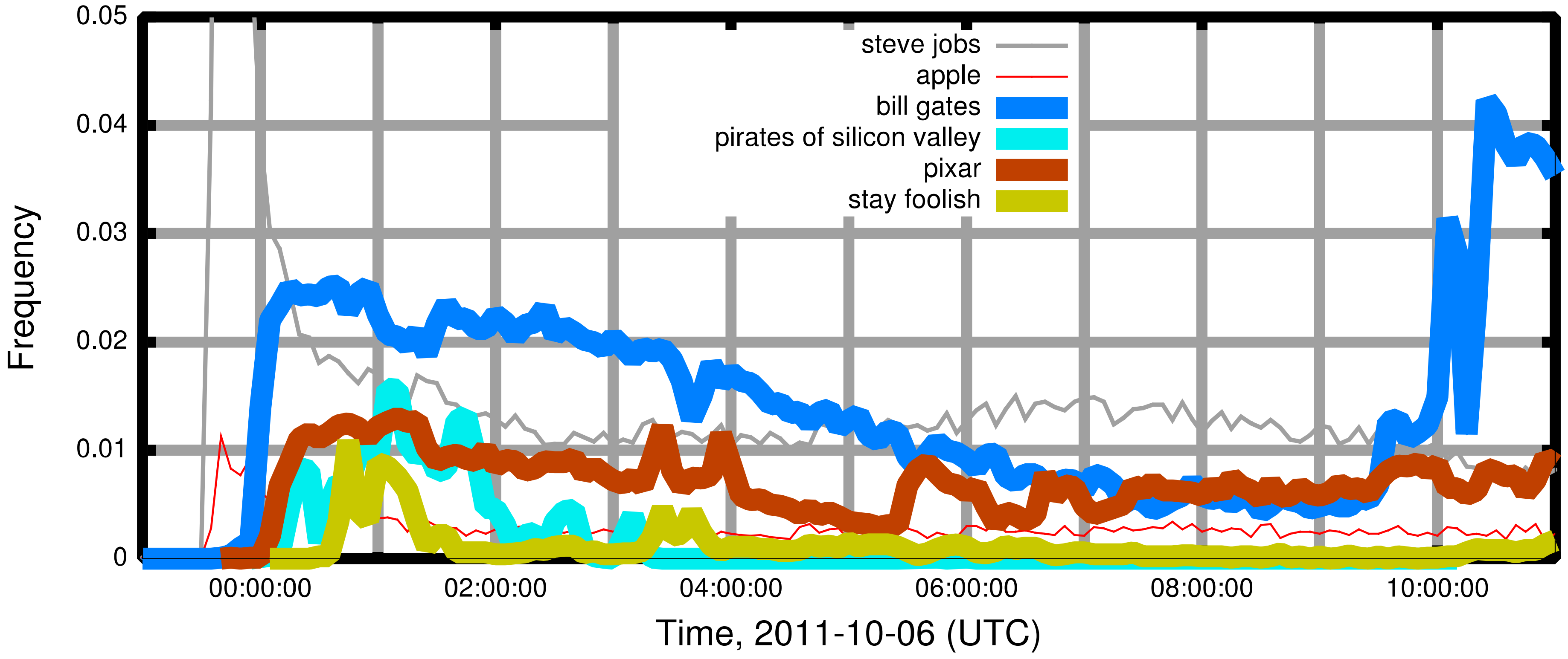
These slides are available at https://www.student.cs.uwaterloo.ca/~cs451


users
Frontend
Backend
OLTP
database
ETL
(Extract, Transform, and Load)
Data
Warehouse
My data is a
day old…
BI tools
Meh.
analysts

Mishne et al. Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. SIGMOD 2013.



Case Study: Steve Jobs passes away



Initial Implementation

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

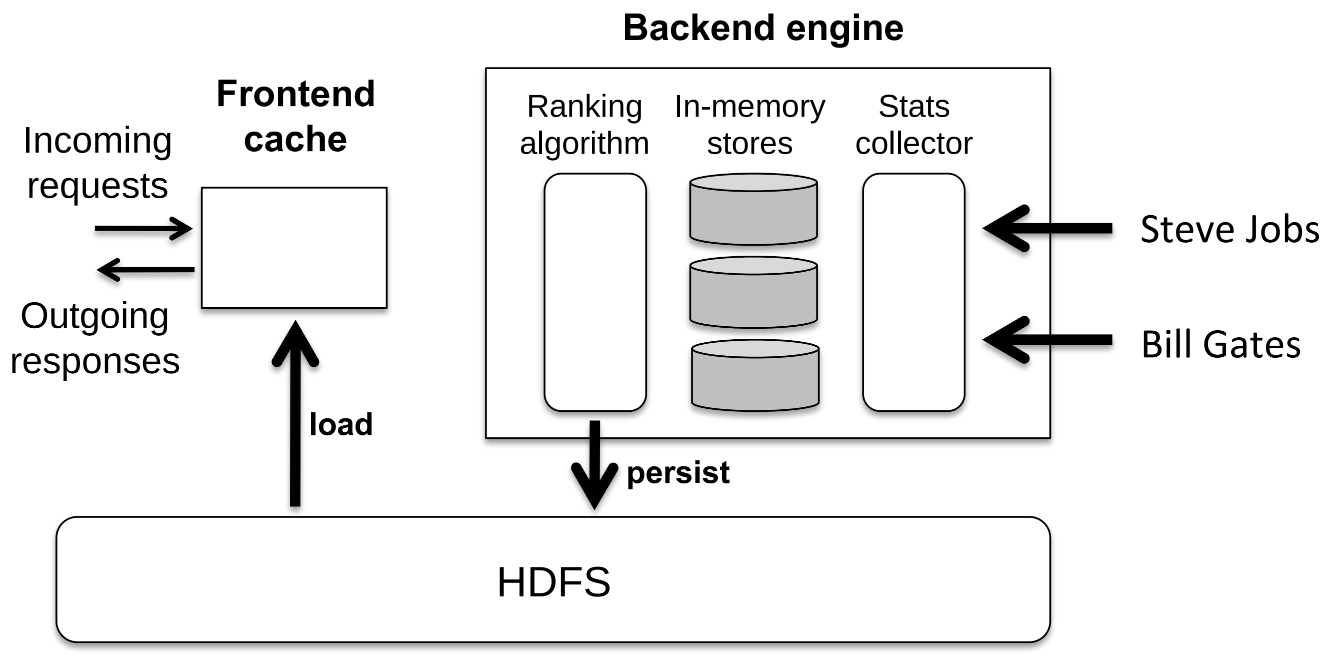
Problem: Query suggestions were several hours old!

Why?

Log collection lag
Hadoop scheduling lag
Hadoop job latencies

We need real-time processing!

Solution?



Can we do better than one-off custom systems?

Stream Processing Frameworks

Source: Wikipedia (River)

Background Review -- Stream Processing

Batch processing	Stream processing
All the data	Continuously incoming data
Not real time	Latency critical (near real time)

Use Cases Across Industries

Credit
Identify fraudulent transactions as soon as they occur.

Transportation
Dynamic Re-routing Of traffic or Vehicle Fleet.

Retail
- Dynamic Inventory Management
- Real-time In-store Offers and recommendations

Consumer Internet & Mobile
Optimize user engagement based on user's current behavior.

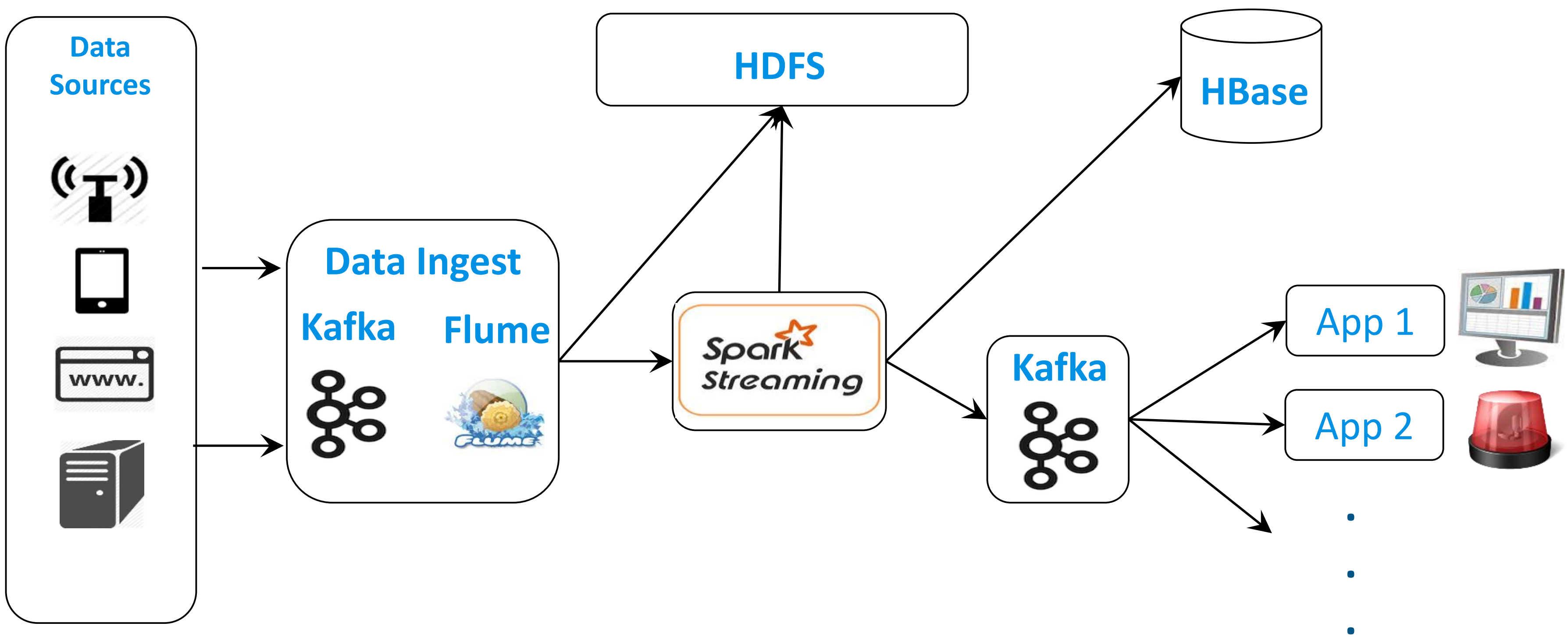
Healthcare
Continuously monitor patient vital stats and proactively identify at-risk patients.

Manufacturing
- Identify equipment failures and react instantly
- Perform Proactive maintenance.

Surveillance
Identify threats and intrusions In real-time

Digital Advertising & Marketing
Optimize and personalize content based on real-time information.

Canonical Stream Processing Architecture



What is a data stream?

Sequence of items:

Structured (e.g., tuples)

Ordered (implicitly or timestamped)

Arriving continuously at high volumes

Sometimes not possible to store entirely

Sometimes not possible to even examine all items

What exactly do you do?

"Standard" relational operations:

Select

Project

Transform (i.e., apply custom UDF)

Group by

Join

Aggregations

What else do you need to make this "work"?

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?

Windows

Windows restrict processing scope:

Windows based on ordering attributes (e.g., time)

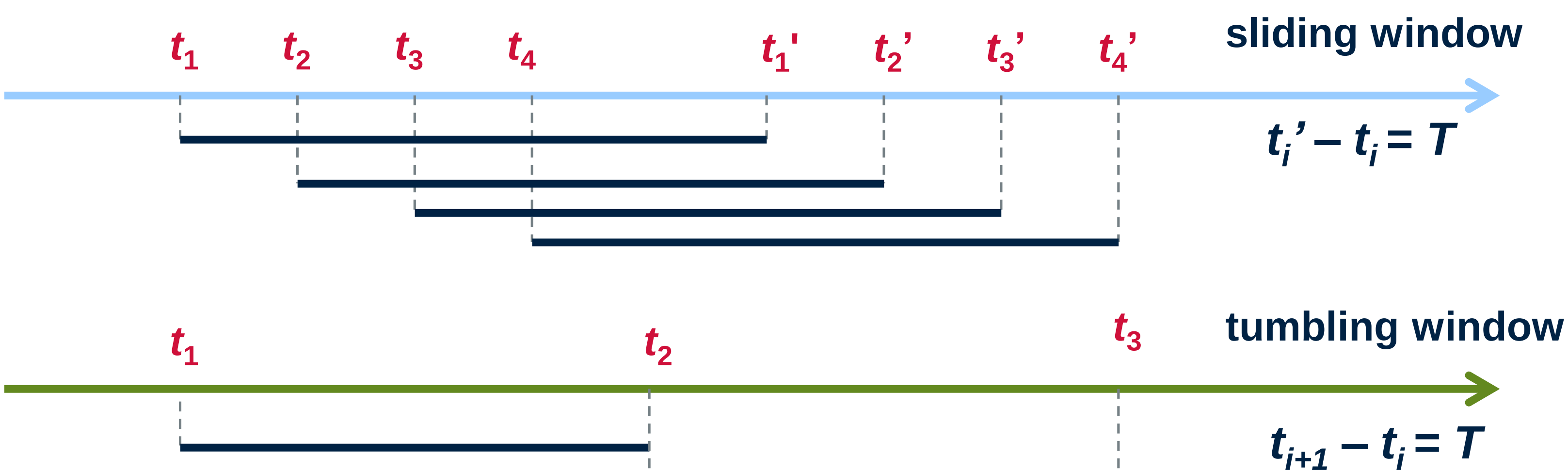
Windows based on item (record) counts

Windows based on explicit markers (e.g., punctuations)

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



Windows on Counts

Window of size N elements (sliding, tumbling) over the stream

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)

Streams Processing Challenges

Inherent challenges

Latency requirements

Space bounds

System challenges

Bursty behavior and load balancing

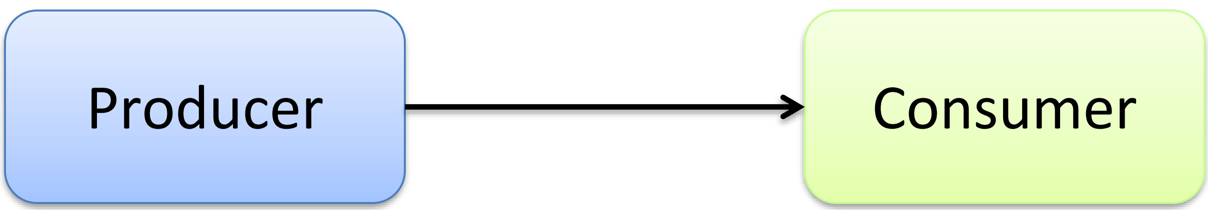
Out-of-order message delivery and non-determinism

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

Producer/Consumers

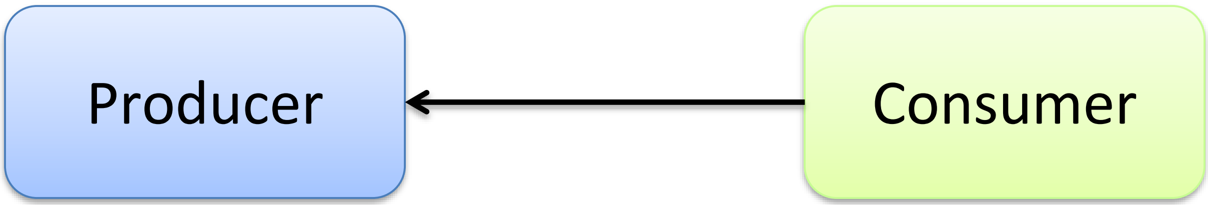
How do consumers get data from producers?

Producer/Consumers



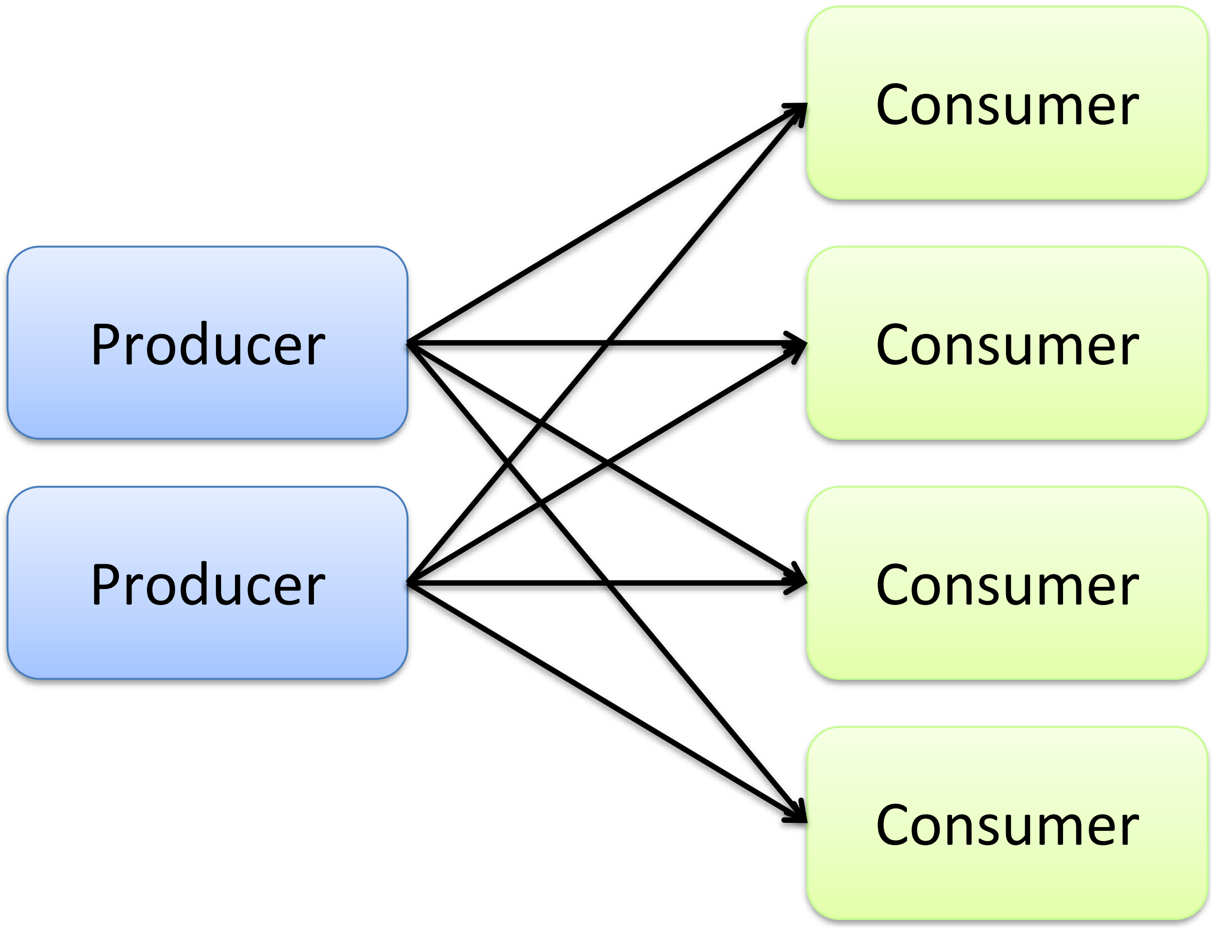
Producer pushes
e.g., callback

Producer/Consumers

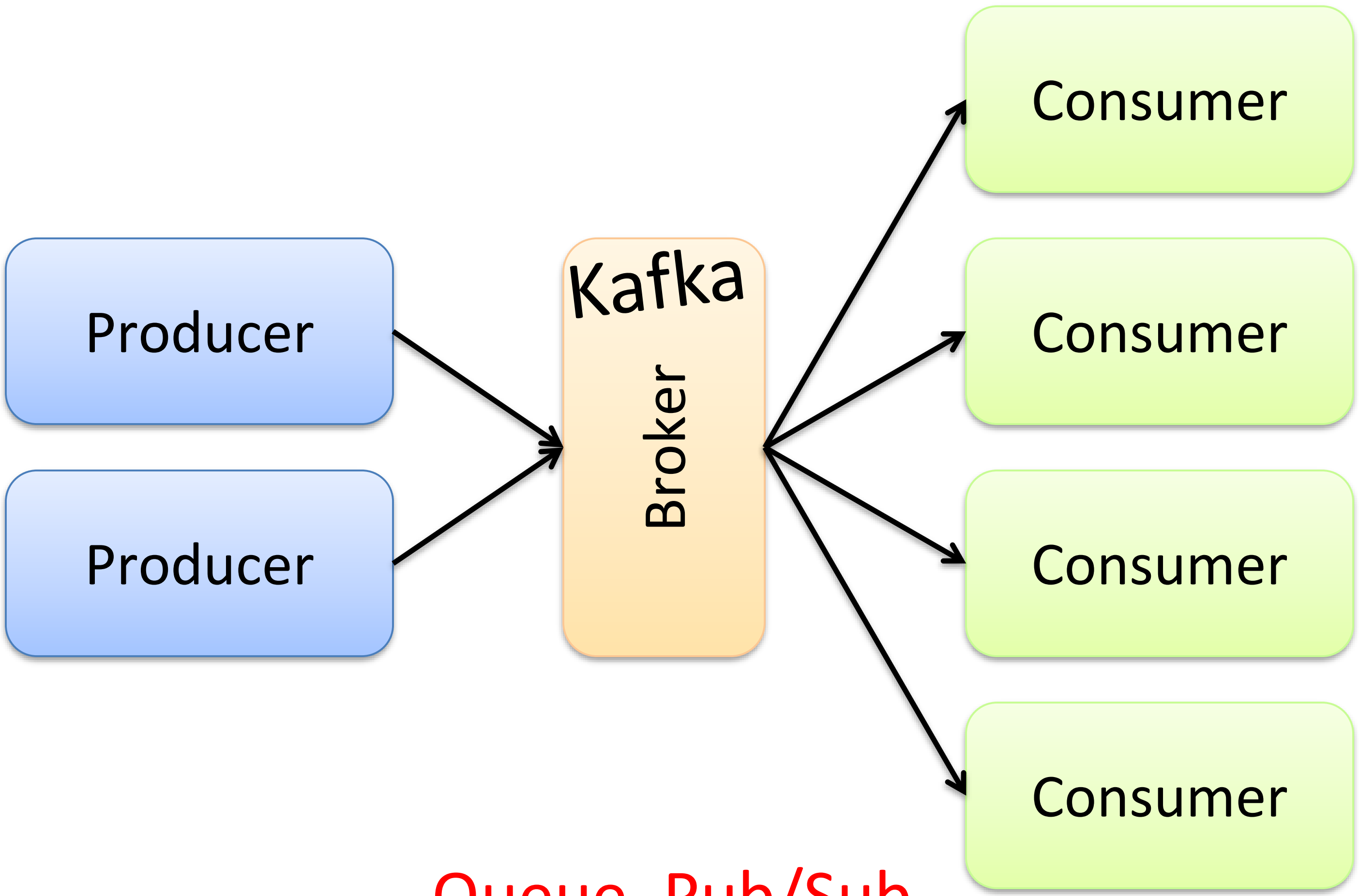


Consumer pulls
e.g., poll, tail

Producer/Consumers

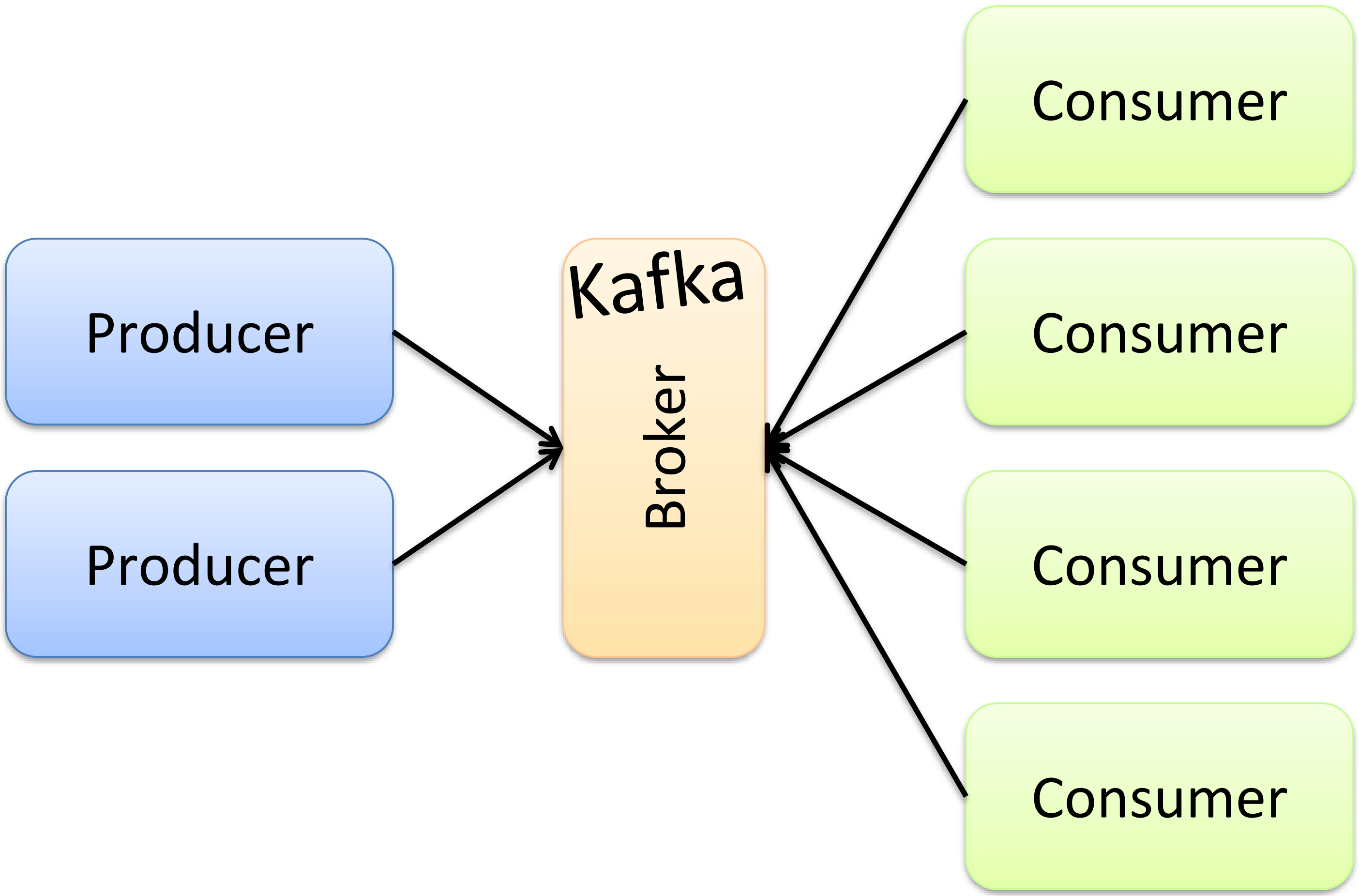


Producer/Consumers



Queue, Pub/Sub

Producer/Consumers



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

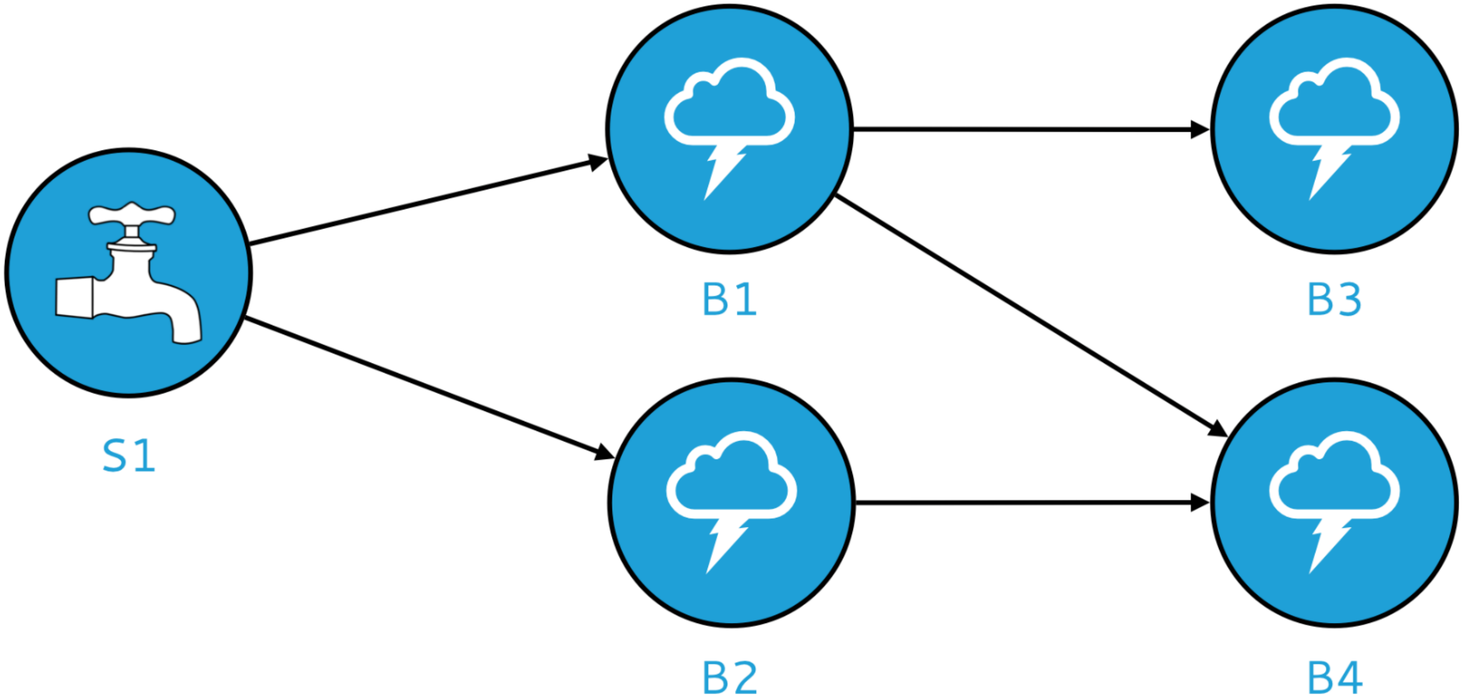
Directed edges indicate communication between vertices

Processing semantics

At most once: without acknowledgments

At least once: with acknowledgements

Spouts and Bolts: Logical Plan



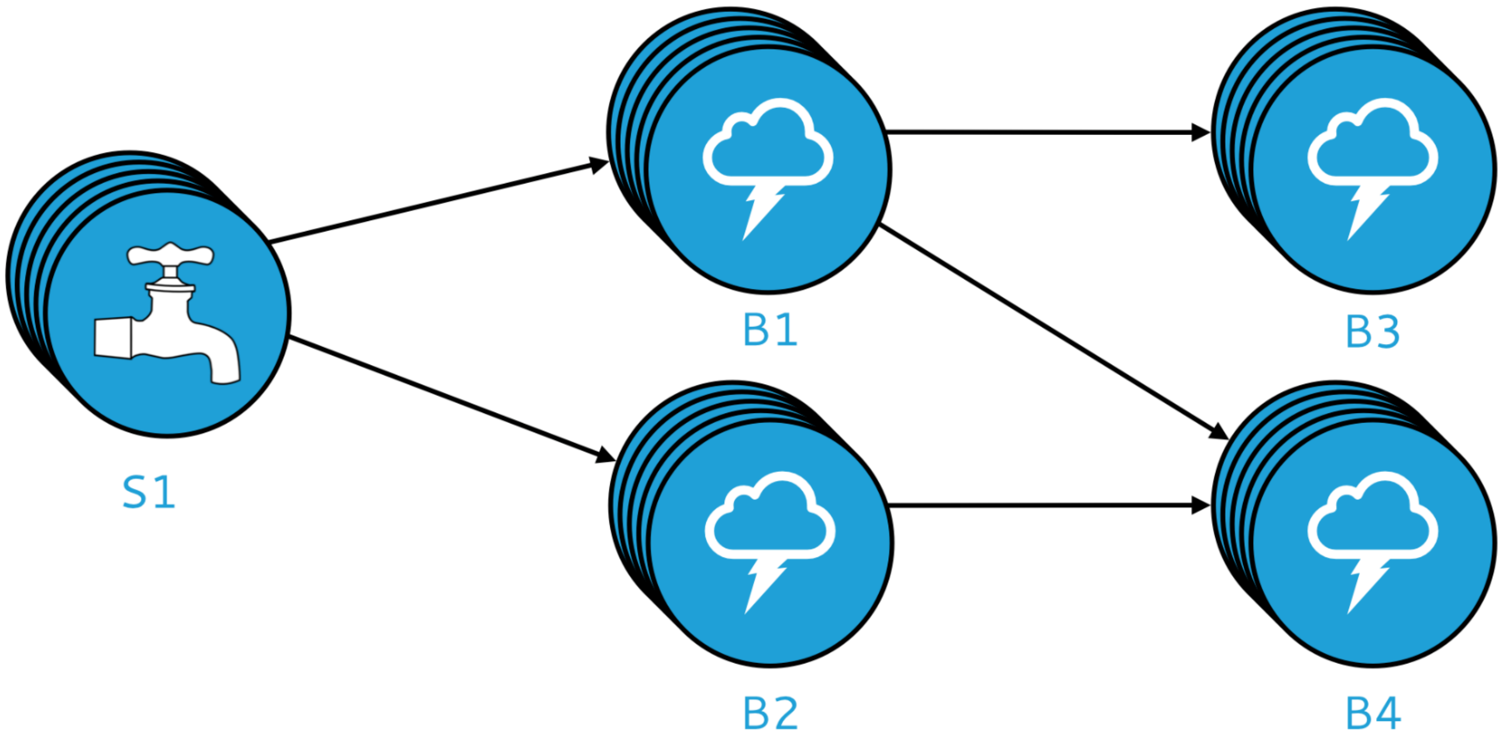
Components

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

Spark Streaming

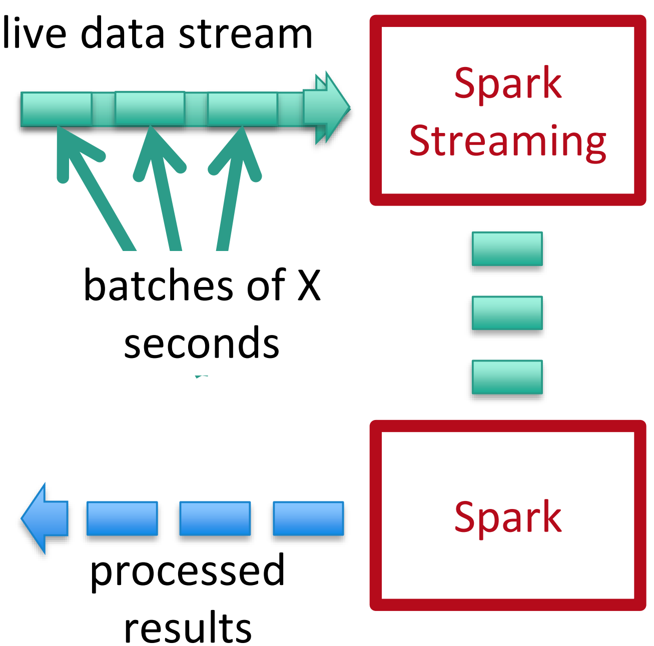
Spark Streaming: Discretized Streams

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

Chop up the stream into batches of *X* seconds

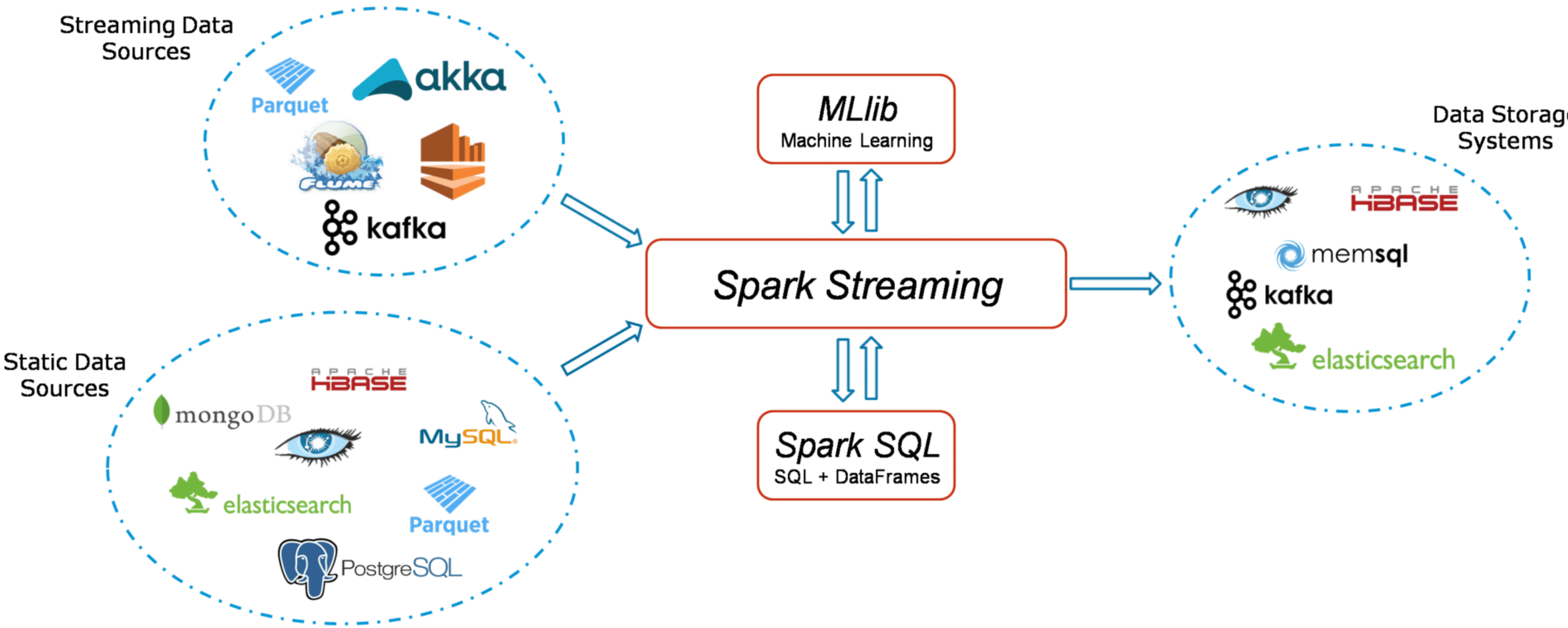
Process as RDDs!

Return results in batches



Typical batch window ~1s

Source: All following Spark Streaming slides by Tathagata Das


Streaming Data Sources
Parquet
akka
kafka
Static Data Sources
HBASE
mongoDB
MySQL
elasticsearch
Parquet
PostgreSQL
MLlib
Machine Learning
Spark Streaming
Spark SQL
SQL + DataFrames
Data Storage Systems
HBASE
memsql
kafka
elasticsearch

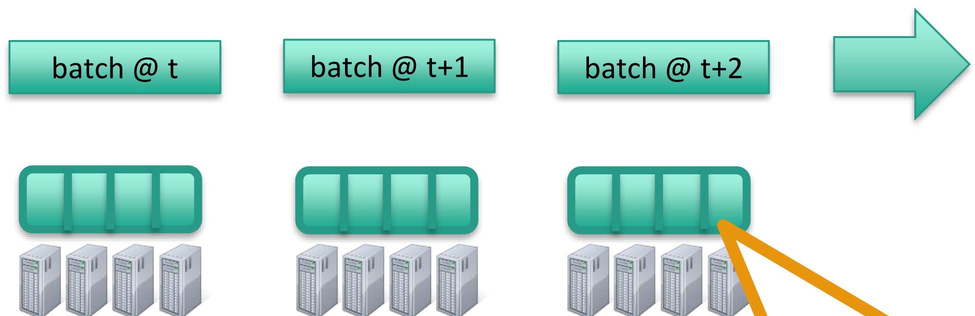
Example: Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

Twitter Streaming API

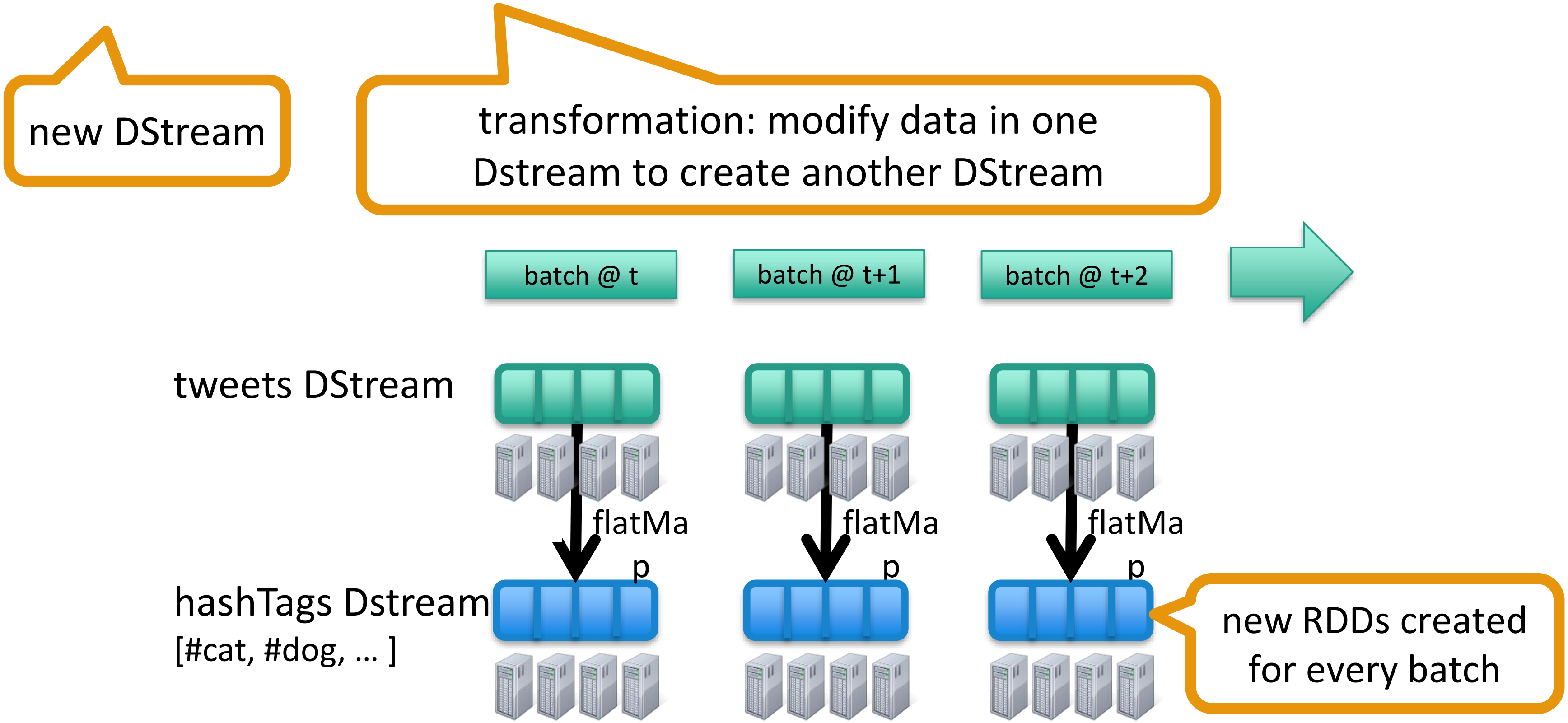
tweets DStream



stored in memory as an RDD (immutable, distributed)

Example: Get hashtags from Twitter

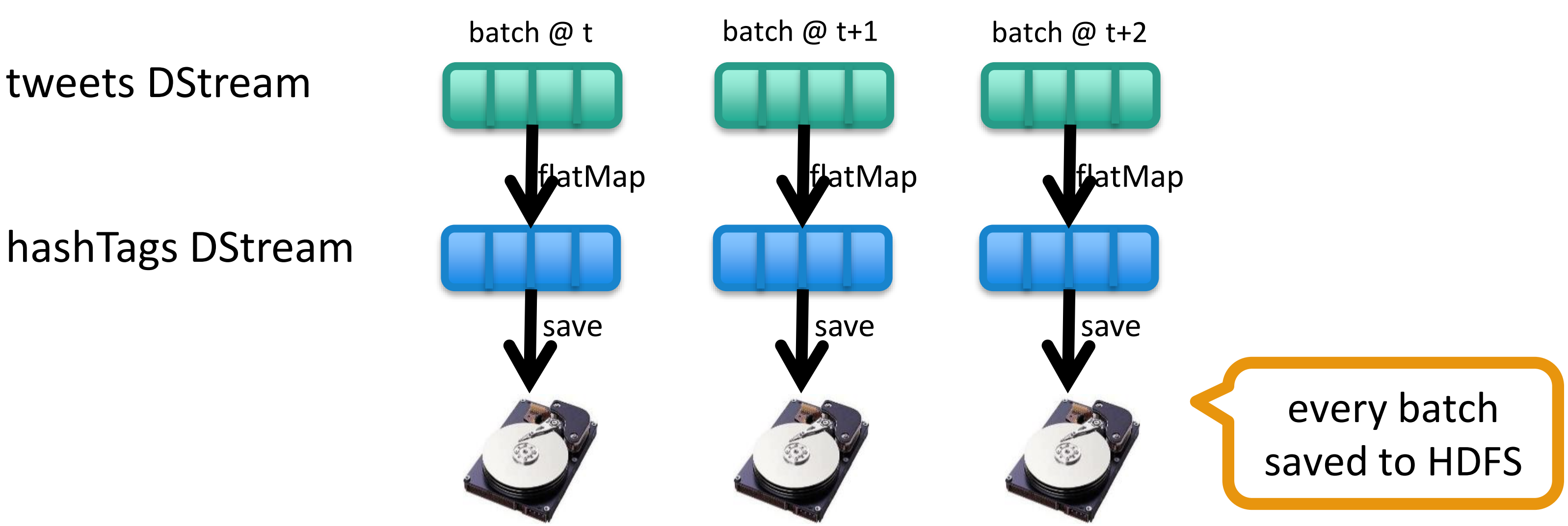
```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```



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

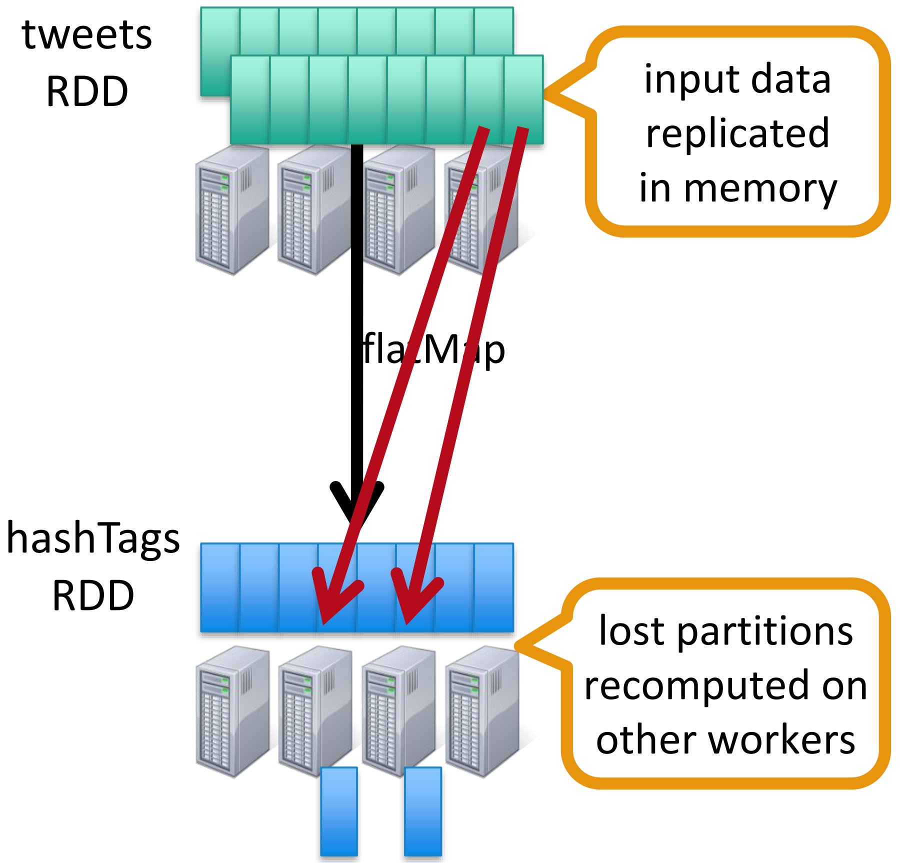


Fault Tolerance

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, TCP sockets

Transformations – modify data from on DStream to another

Standard RDD operations – map, countByValue, reduce, join, …

Stateful operations – window, countByValueAndWindow, …

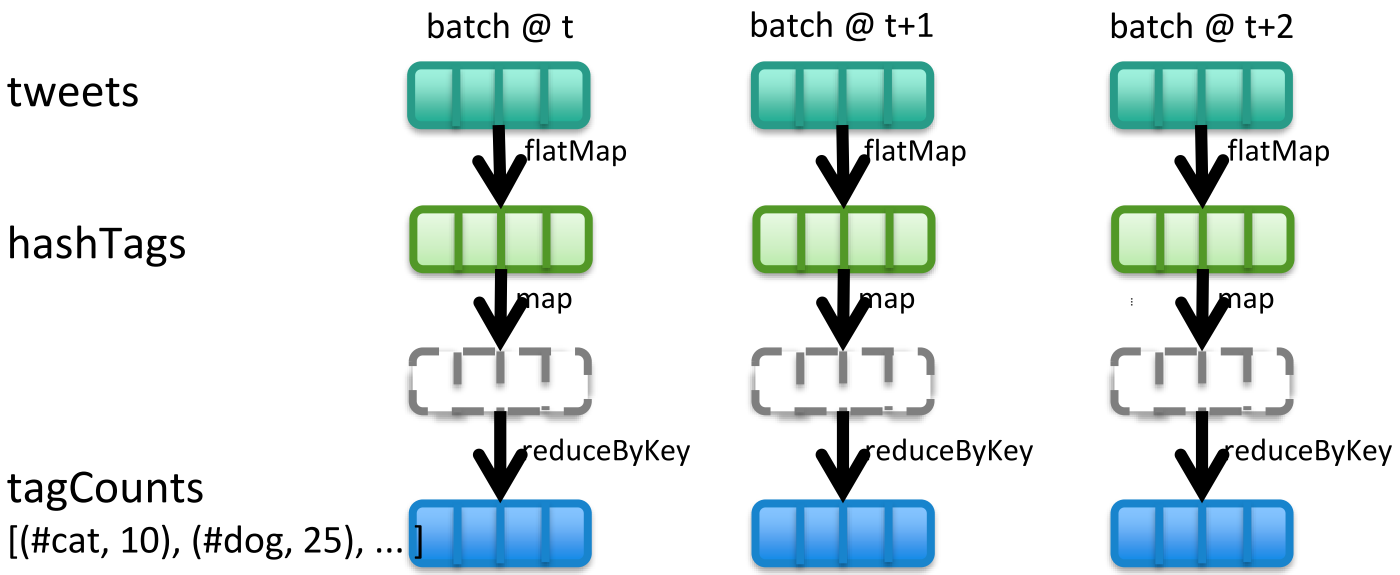
Output Operations – send data to external entity

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()
```

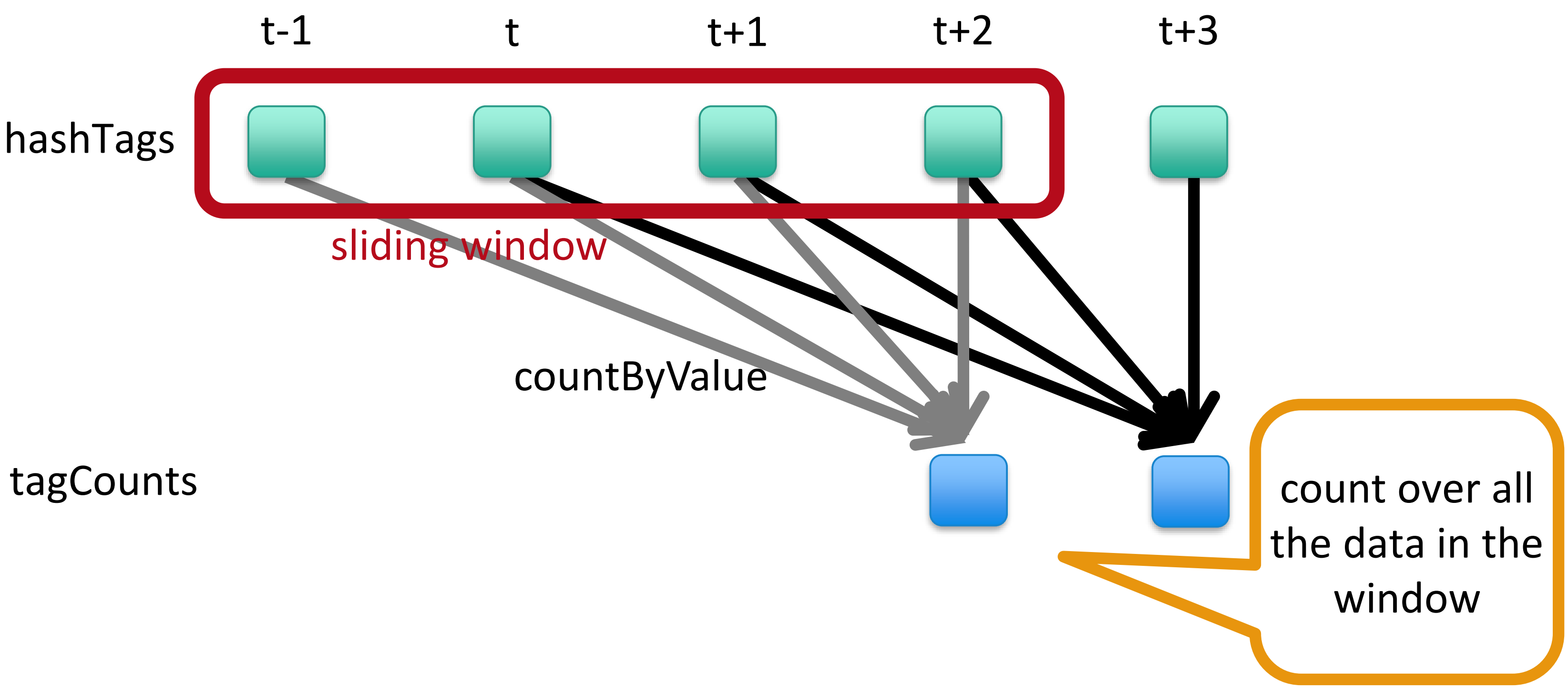
sliding window operation

window length

sliding interval

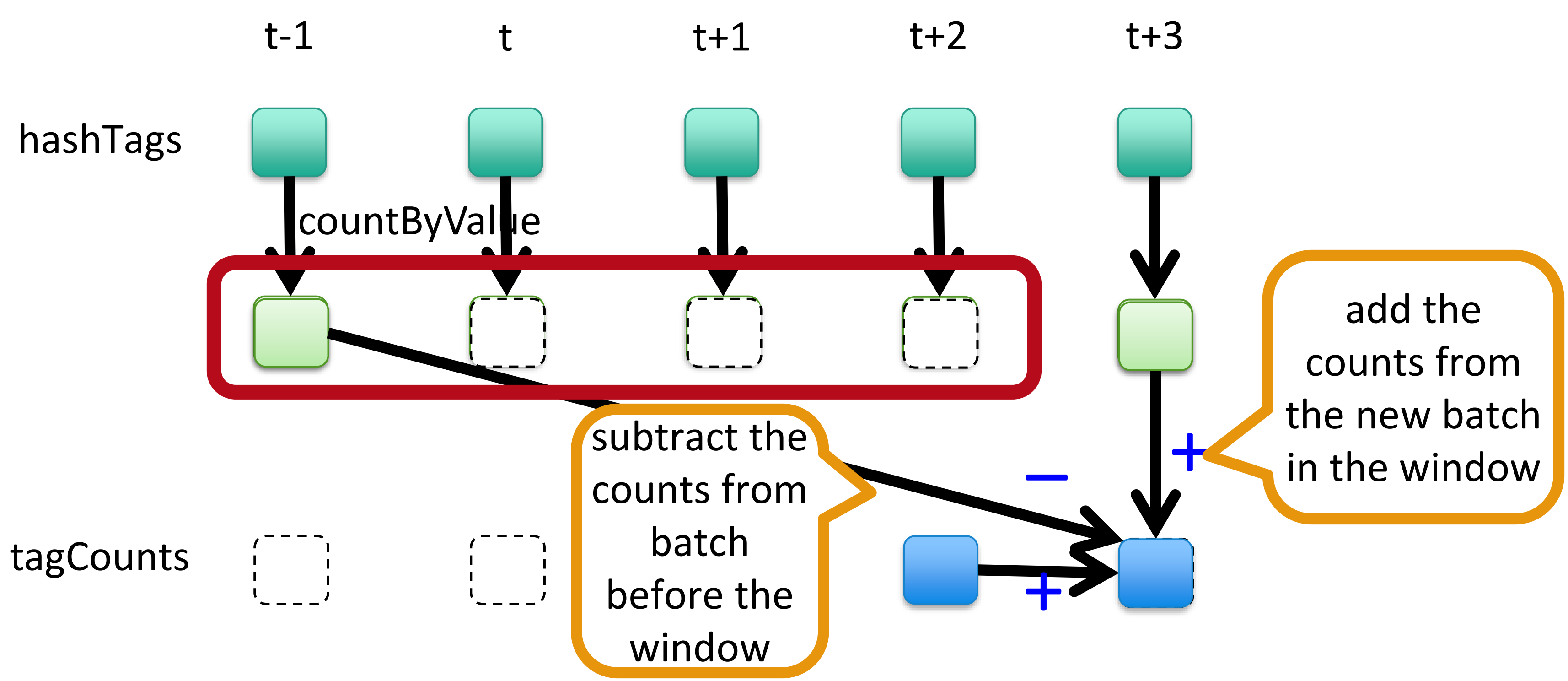
Example: Count the hashtags over last 10 mins

```
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

Incremental counting generalizes to many reduce operations

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

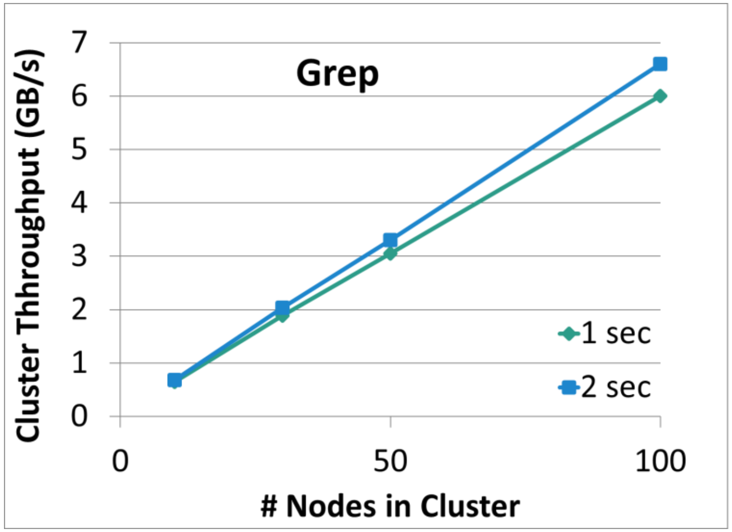
```
val tagCounts = hashtags
    .countByValueAndWindow(Minutes(10), Seconds(1))
```

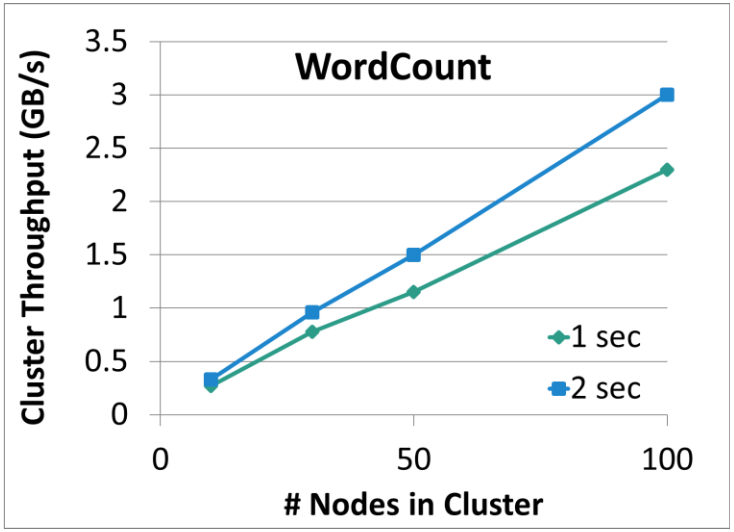
```
val tagCounts = hashtags
    .reduceByKeyAndWindow(_ + _, _ - _, Minutes(10), Seconds(1))
```

Performance

Can process **6 GB/sec (60M records/sec**) of data on 100 nodes at **sub-second** latencyTested

- with 100 streams of data on 100 EC2 instances with 4 cores each





Comparison with Storm

Higher throughput than Storm

- Spark Streaming: 670k records/second/node
- Storm: 115k records/second/node