BIG DATA

Streaming Spark

K V Subramaniam

Computer Science and Engineering

BIG DATA

Examples of Streaming

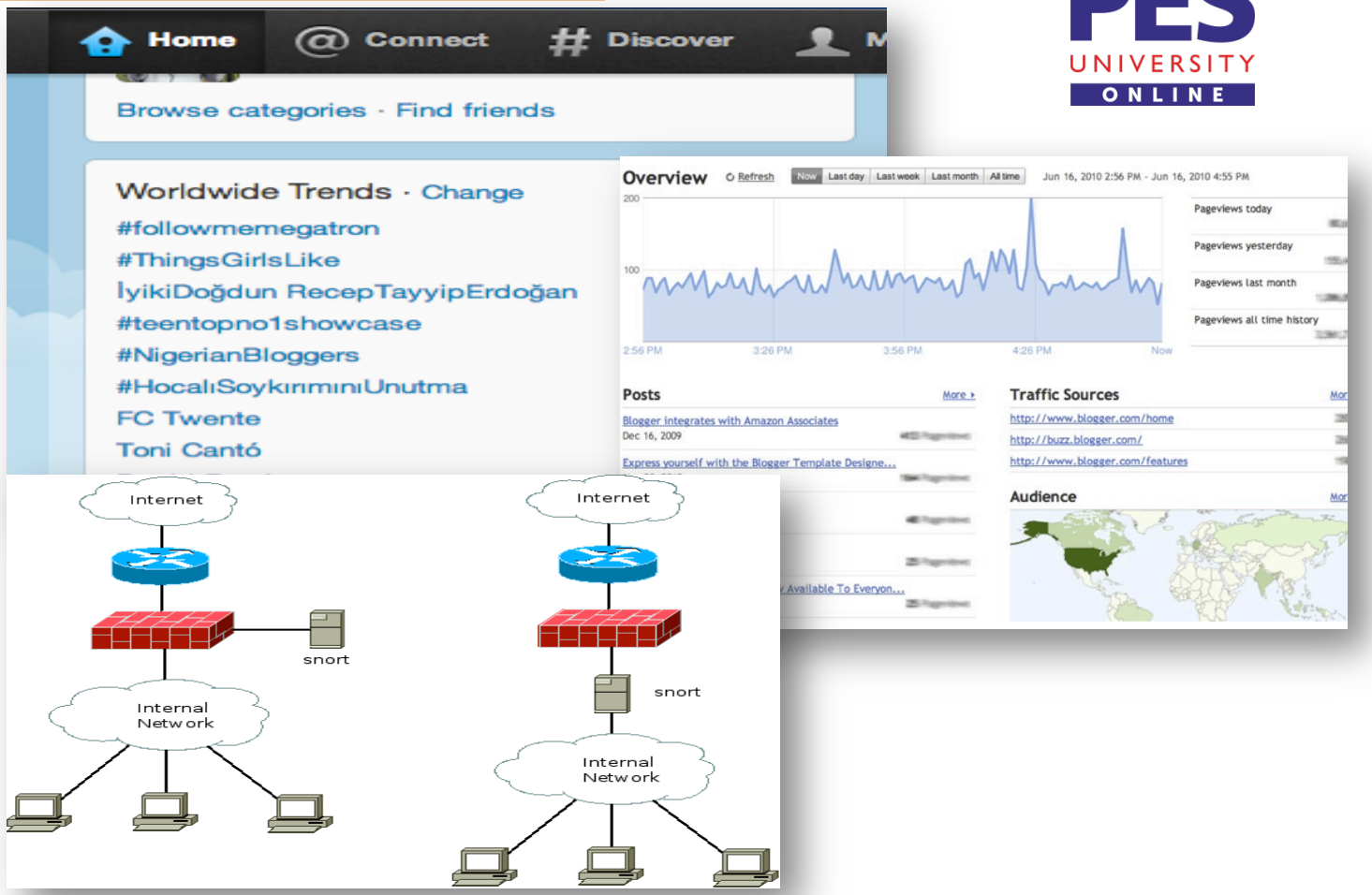
- Sensor data, e.g.,
  - Temperature sensor in the ocean
  - Reports GPS
  - 1 sensor every 150 square miles => 1,000,000 sensors
  - 10 readings/sec => 3.5 TB/day
- Images
  - London: 6 million video cameras
- Internet / Web traffic
  - Google: hundreds of millions of queries/ day
  - Yahoo: billions of clicks/day

BIG DATA

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrusion detection systems
  - Transportation system - Uber
  - etc.
- Require large clusters to handle workloads

- Require latencies of few seconds



BIG DATA

Stream Data Model

- Multiple streams
- Different rates, not synchronized
- Archival store
  - Offline analysis, not real-time
- Working store
  - Disk or memory
  - Summaries
  - Parts of streams
- Queries
  - Standing queries
  - Ad-hoc queries

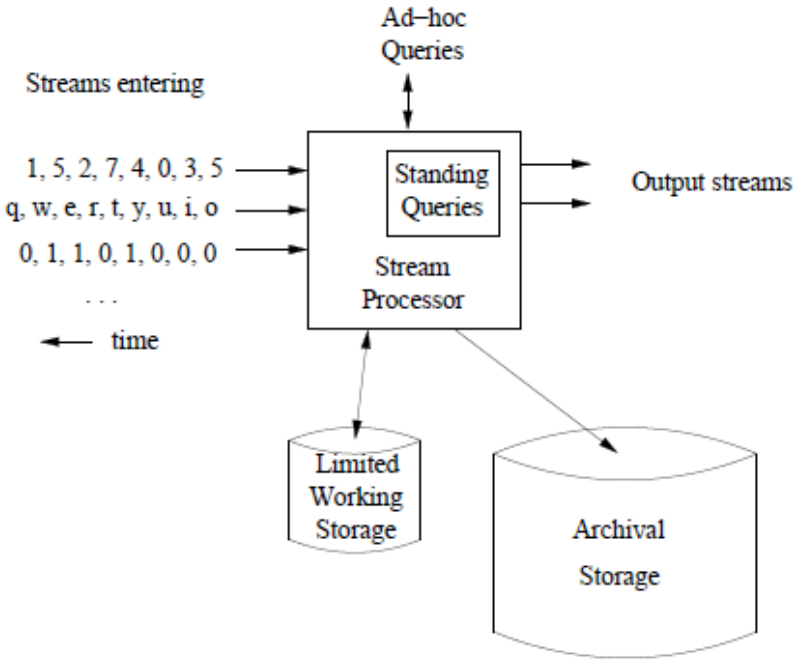


Figure 4.1: A data-stream-management system

Examples of Stream Queries

- **Standing queries**: produce outputs at appropriate time
  - Query is continuously running
  - Constantly reading new data
  - Query execution can be optimized
  - Example: maximum temperature ever recorded
- ***Ad hoc* query**: not predetermined, arbitrary query
  - Need to store stream
  - Approach: store sliding window in SQL DB
  - Do SQL query
  - Example: number of unique users over last 30 days
    - Store logins for last 30 days

BIG DATA

Exercise 1

Consider the queries on the right. Which among them are **STANDING QUERIES** and which are **AD HOC**?

- Alert when temperature > threshold
- Display average of last n temperature readings; n arbitrary
- List of countries from which visits have been received over last year
- Alert if website receives visit from a black-listed country

BIG DATA

Exercise 1 Solution

Consider the queries on the right. Which among them are **STANDING QUERIES** and which are **AD HOC**?

Solution

- Alert when temperature > threshold - standing
- Display average of last *n* temperature readings – *ad hoc*
- List of countries from which visits have been received over last year – *ad hoc*
- Alert if website receives visit from a black-listed country - standing

BIG DATA

Issues in Stream Processing

- Velocity
  - Streams can have high data rate
  - Need to process very fast
- Volume
  - Low data rate, but large number of streams
    - Ocean sensors, pollution sensors
- Need to store in memory
  - May not have huge memory
    - Approximate solutions

Need for a framework …

… for building such complex stream processing applications

But what are the requirements

from such a framework?

BIG DATA

Requirements

- **Scalable** to large clusters
- **Second**-scale latencies
- **Simple** programming model

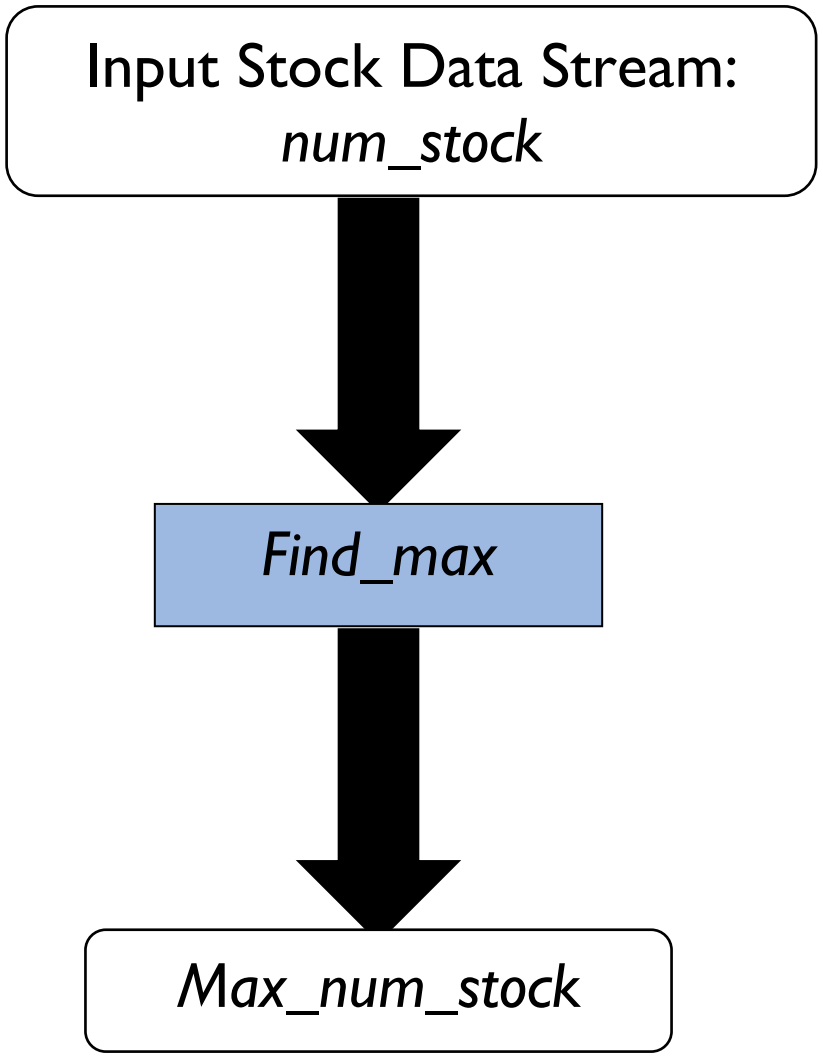
Need for integrated Batch and Stream processing

BIG DATA

Exercise 2

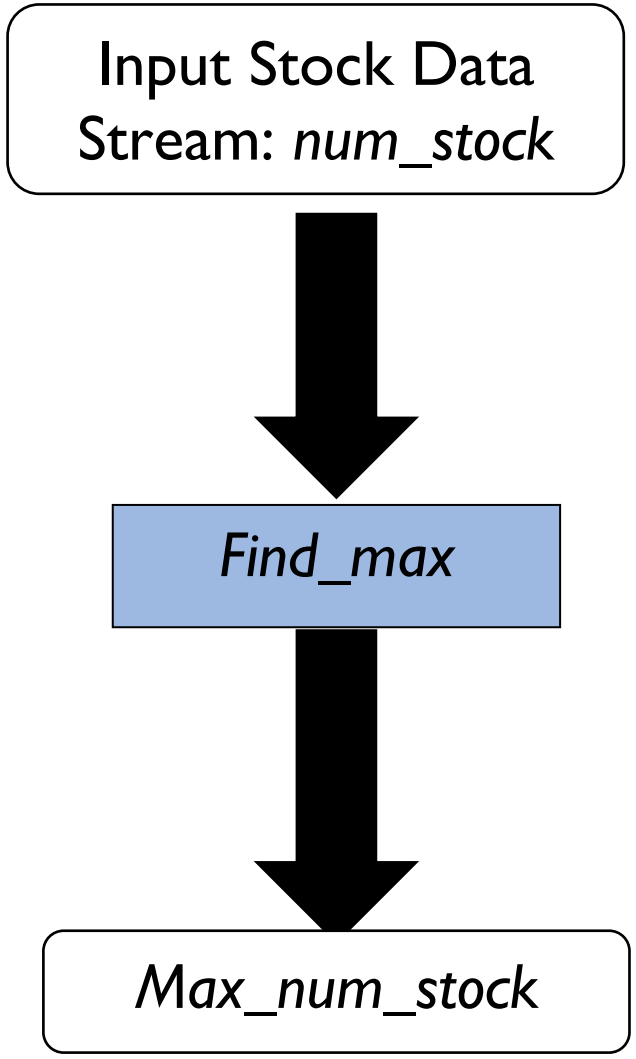
CAN WE USE HADOOP?

- Consider the simple program on the right.
- The input is a stream of records from the stock market.
  - Each time a stock is sold, a new record is created.
  - The record contains a field num_stock which is the number of stocks sold.
- Find_max is a program that updated a variable Max_num_stock which is the maximum of num_stock.



Exercise 2 - Solution CAN WE USE HADOOP?

- Write the pseudo-code for *Find_max*
  - *If num_stock > Max_num_stock*
    Max_num_stock = num_stock
- Can this be implemented in Hadoop?
  - We need to process one record at a time. Hadoop processes a full file in the Map
  - *Max_num_stock* is a global variable
- Does your solution assume that *Find_max* runs on a single node? Is this a reasonable assumption?
  - No, the number of transactions could be more than what a single node can handle



Similar problems can arise in Spark

Case study: Conviva, Inc.

- Real-time monitoring of online video metadata
  HBO, ESPN, ABC, SyFy, …

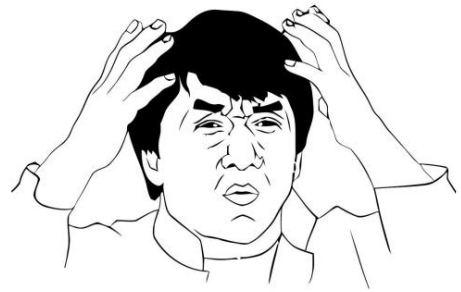
Since we can't use Hadoop

- Two processing stacks
  - Custom-built distributed stream processing system
    - 1000s complex metrics on millions of video sessions
    - Requires many dozens of nodes for processing
  - Hadoop backend for offline analysis
    - Generating daily and monthly reports
    - **Similar computation as the streaming system**

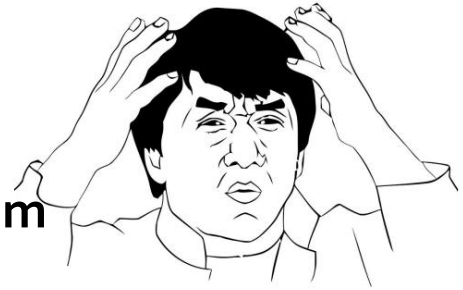
BIG DATA

Case study: XYZ, Inc.

- ***Any company who wants to process live streaming data has this problem***
- **Twice** the effort to implement any new function
- **Twice** the number of bugs to solve
- **Twice** the headache



- Two processing stacks
  - Custom-built distributed stream processing system
    - 1000s complex metrics on millions of video sessions
    - Requires many dozens of nodes for processing
  - Hadoop backend for offline analysis
    - Generating daily and monthly reports
    - **Similar computation as the streaming system**



BIG DATA

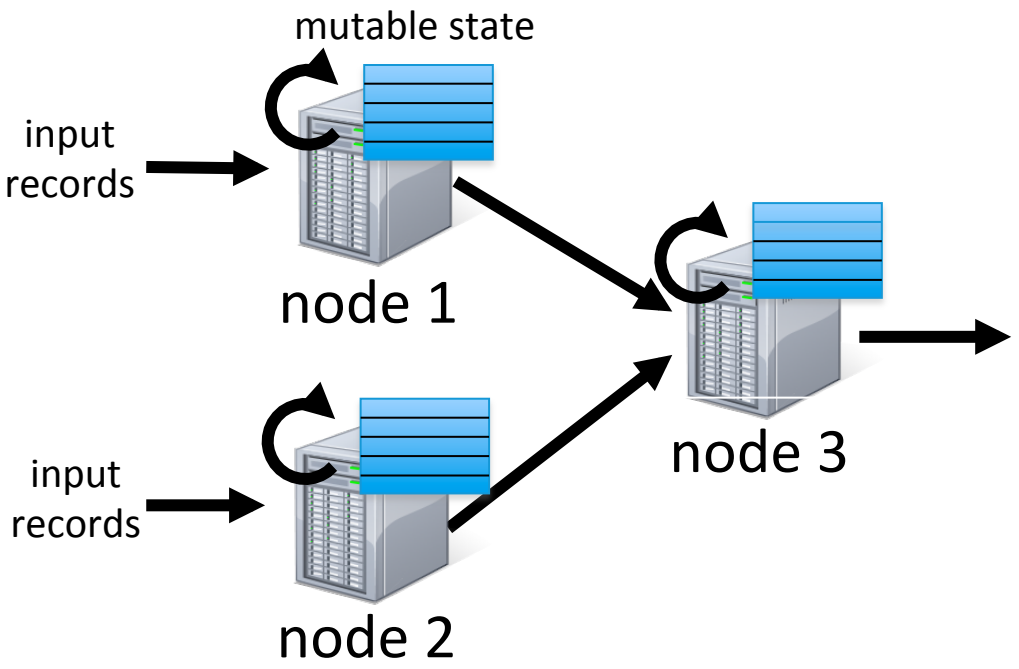
Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing

State in Stream processing

Stateful Stream Processing

- Traditional streaming systems have a event-driven **record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging

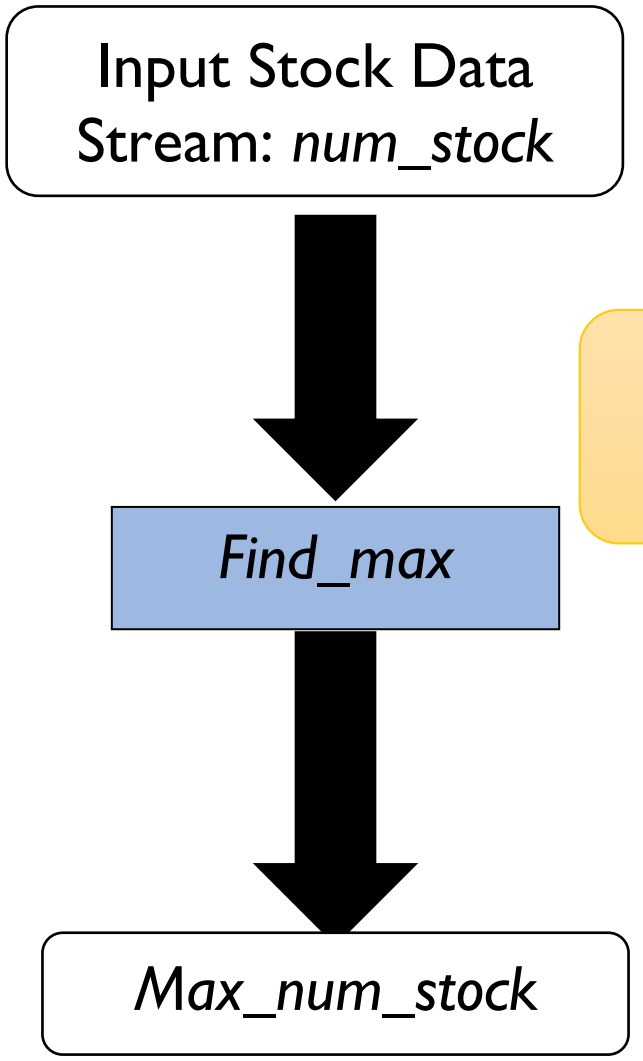


BIG DATA

Exercise 2 - Solution

CAN WE USE HADOOP?

- Write the pseudo-code for *Find_max*
  - *If num_stock > Max_num_stock*
    Max_num_stock = num_stock
- Can this be implemented in Hadoop?
  - We need to process one record at a time. Hadoop processes a full file in the Map
  - *Max_num_stock* is a global variable
- Does your solution assume that *Find_max* runs on a single node? Is this a reasonable assumption?
  - No, the number of transactions could be more than what a single node can handle



Similar problems can arise in Spark

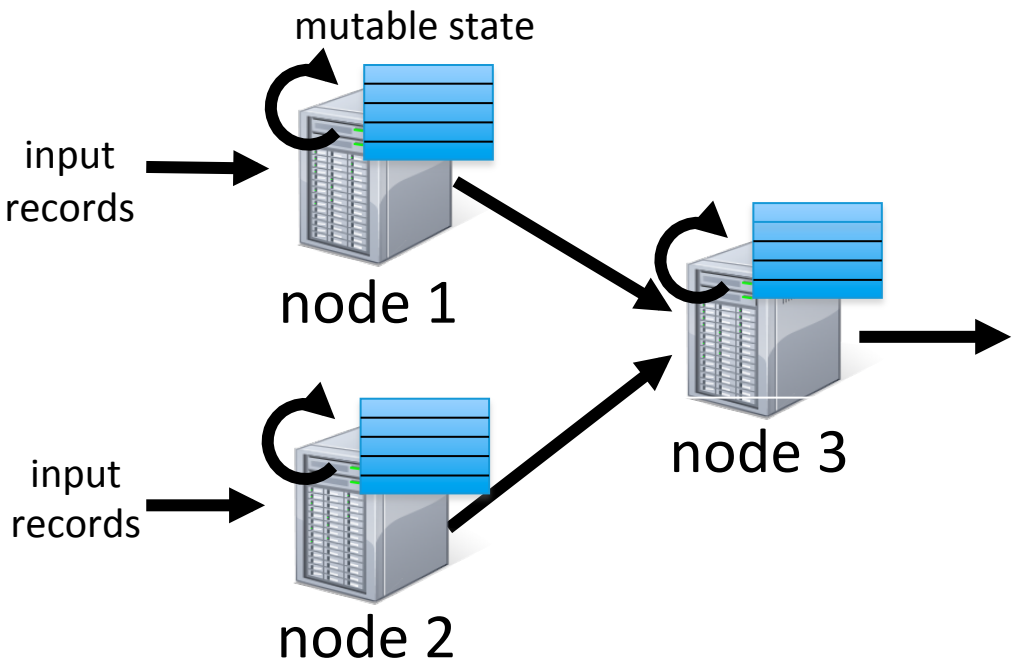
Exercise 3: Stateful Stream Processing

- Traditional streaming systems have a event-driven **record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records

In the stock market example, where is the state?

- State is lost if node dies!

- Making stateful stream processing be fault-tolerant is challenging

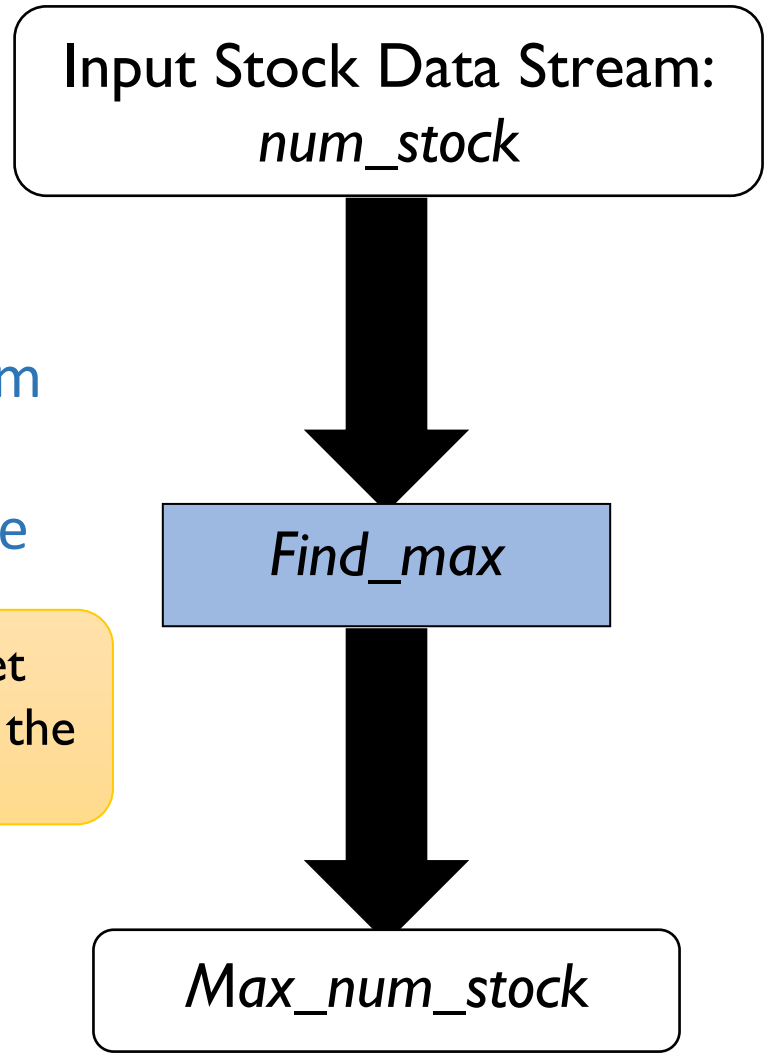


BIG DATA

Exercise 3: (solution): Stateful Stream Processing

- Write the pseudo-code for Find_max
  - If num_stock > Max_num_stock
    - Max_num_stock = num_stock
- Max_num_stock is a global state
  - Its value depends upon the entire stream sequence
  - The first num_stock could have been the largest

In the stock market example, where is the state?



Existing Streaming Systems

- Storm
  - Replays record if not processed by a node
  - Processes each record at least once
  - May update mutable state twice!
  - Mutable state can be lost due to failure!

- Trident – Use transactions to update state
  - Processes each record exactly once
  - Per state transaction updates slow

BIG DATA

Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
- **Efficient fault-tolerance** in stateful computations

Spark Streaming

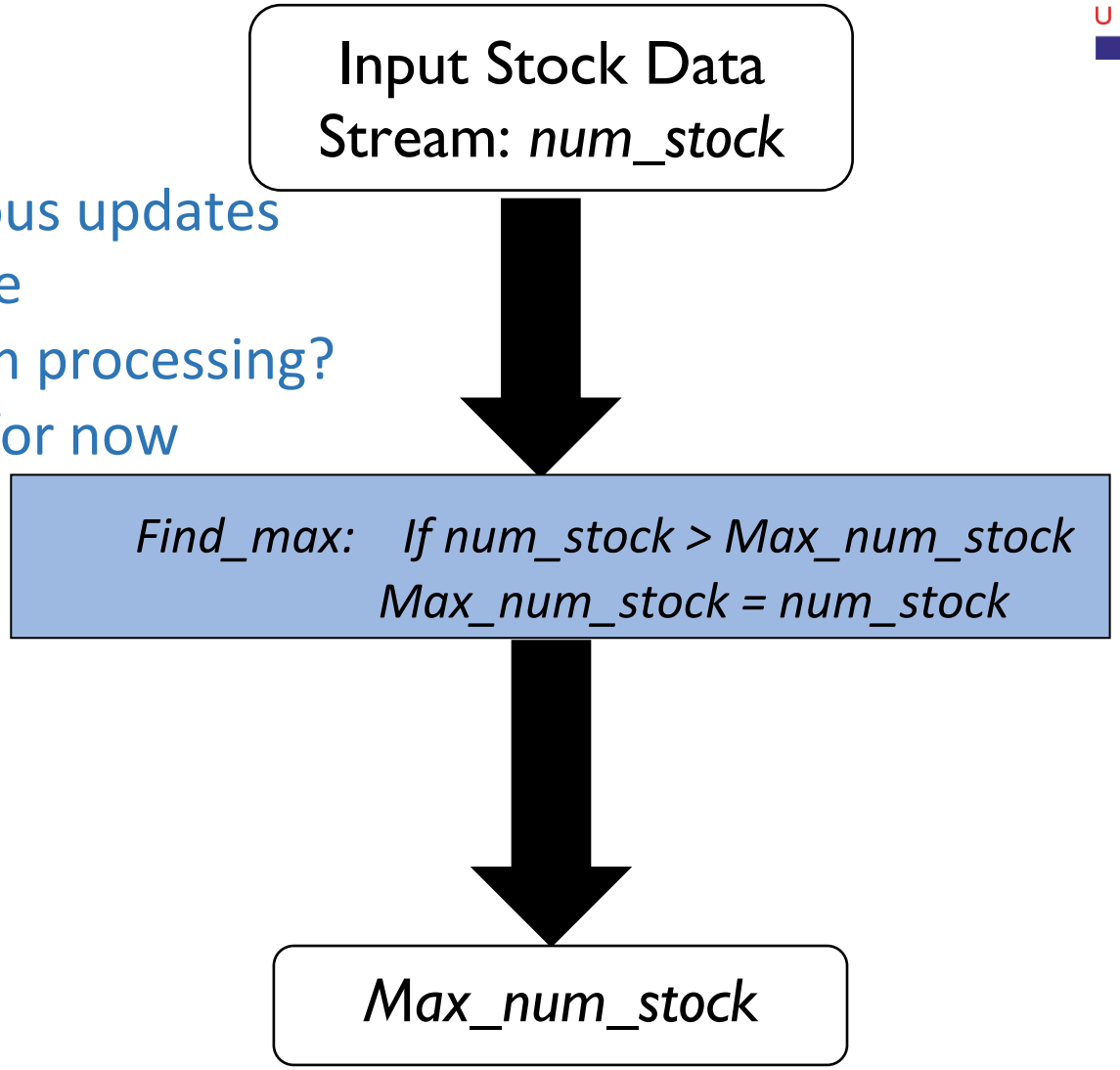
What is Spark Streaming?

- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Integrates with Spark's batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.

Exercise 4:

Can we modify Hadoop?

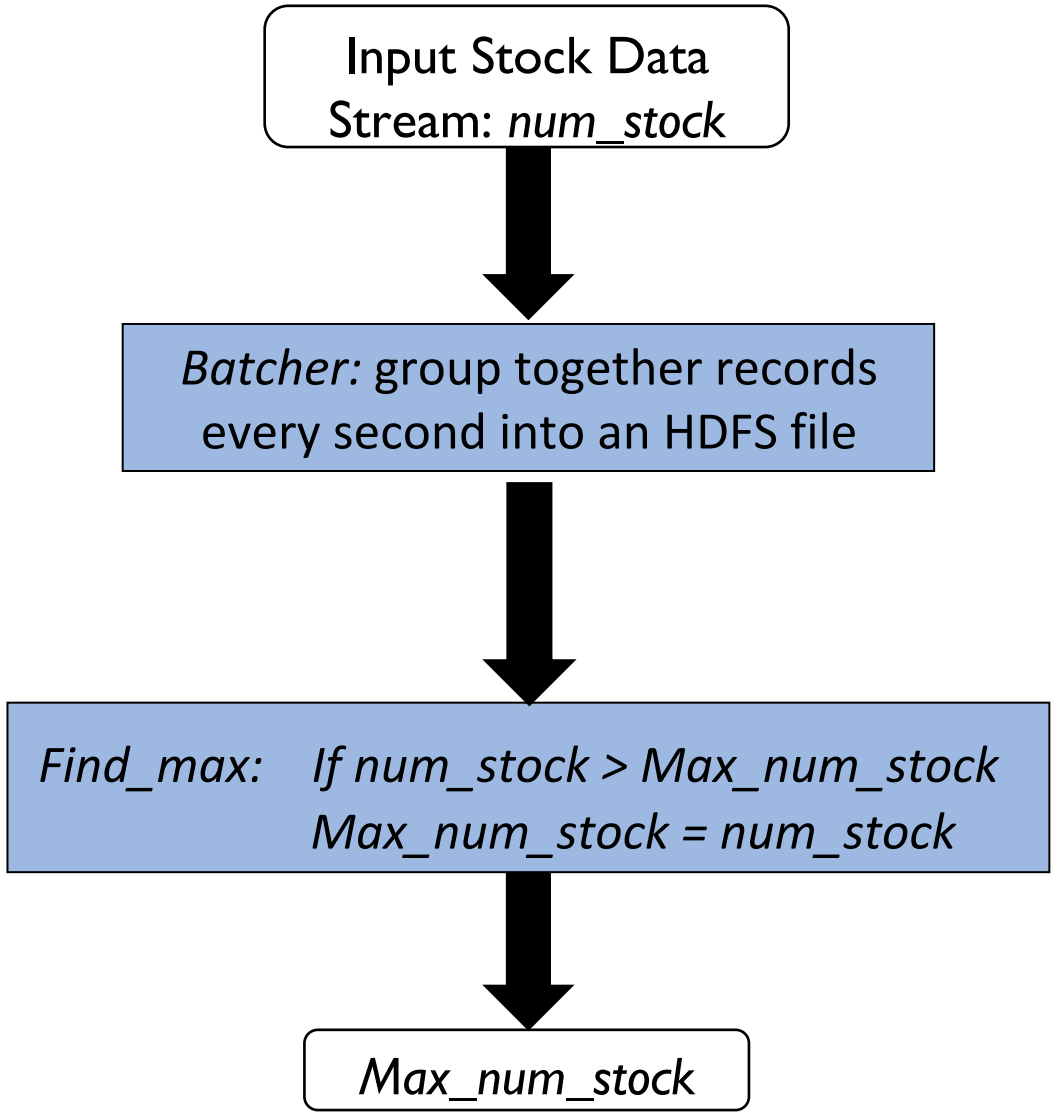
- Suppose we don't want instantaneous updates
- Say 1 second updates are acceptable
- Can we modify Hadoop to do stream processing?
- Ignore the global variable problem for now



BIG DATA

Exercise 4: Solution

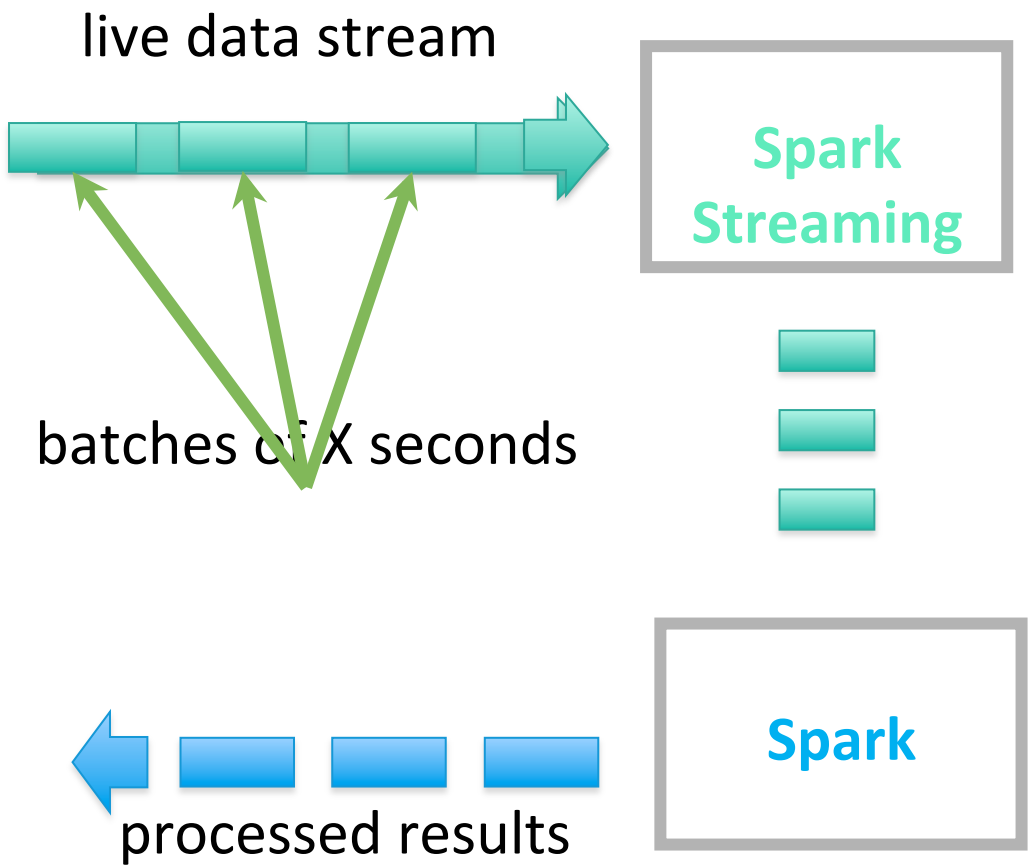
- We can *batch* input records into an HDFS file
- We can batch together the inputrecords every second
- This file can be processed using MapReduce
- We can produce an update every second
- We have ignored the global variable problem for now



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

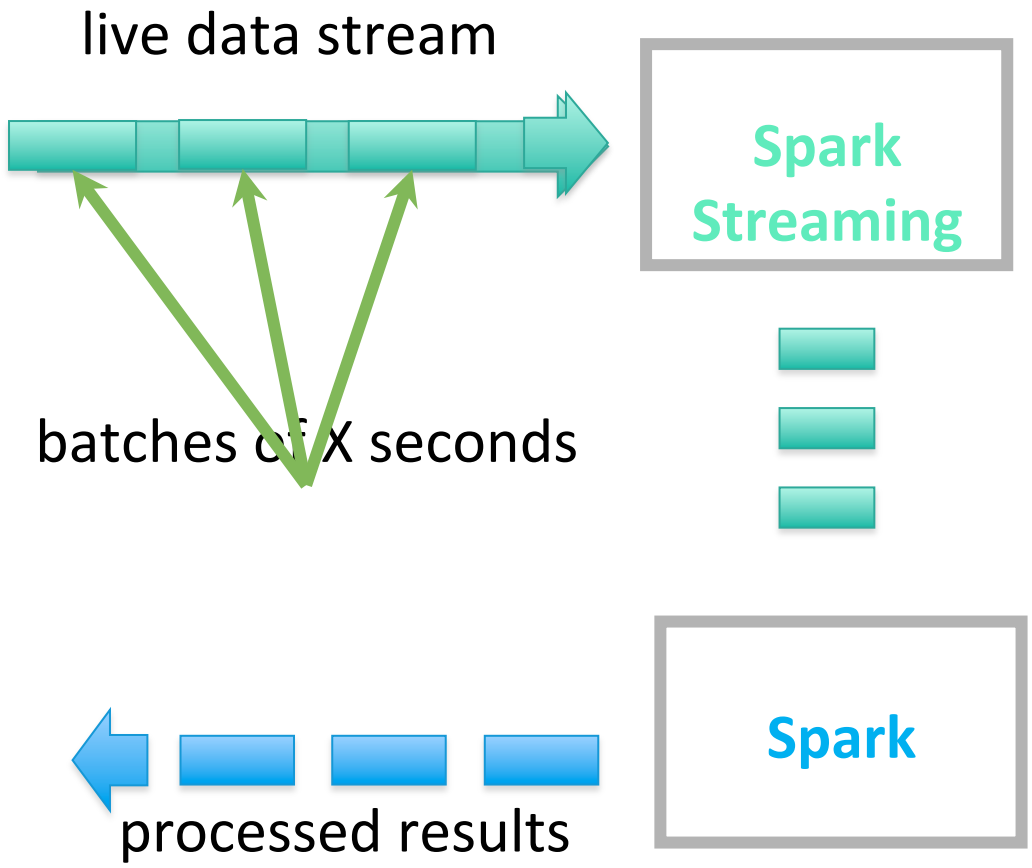


BIG DATA

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



Streaming Spark - DStreams

Remember!!!!

- **In Spark (not Streaming Spark)**
  - Every variable is an **RDD**
  - There are two kinds of **RDDs**
  - ***Pair RDDs*** are RDDs that consist of key-value pairs
  - **Special operations**, such as *reduceByKey* are defined on *pair RDDs*

BIG DATA

Example 1 – Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

twitterstream returns a variable of type *DSTREAM*

Twitter Streaming API

tweets DStream

stored in memory as an RDD (immutable, distributed)

BIG DATA

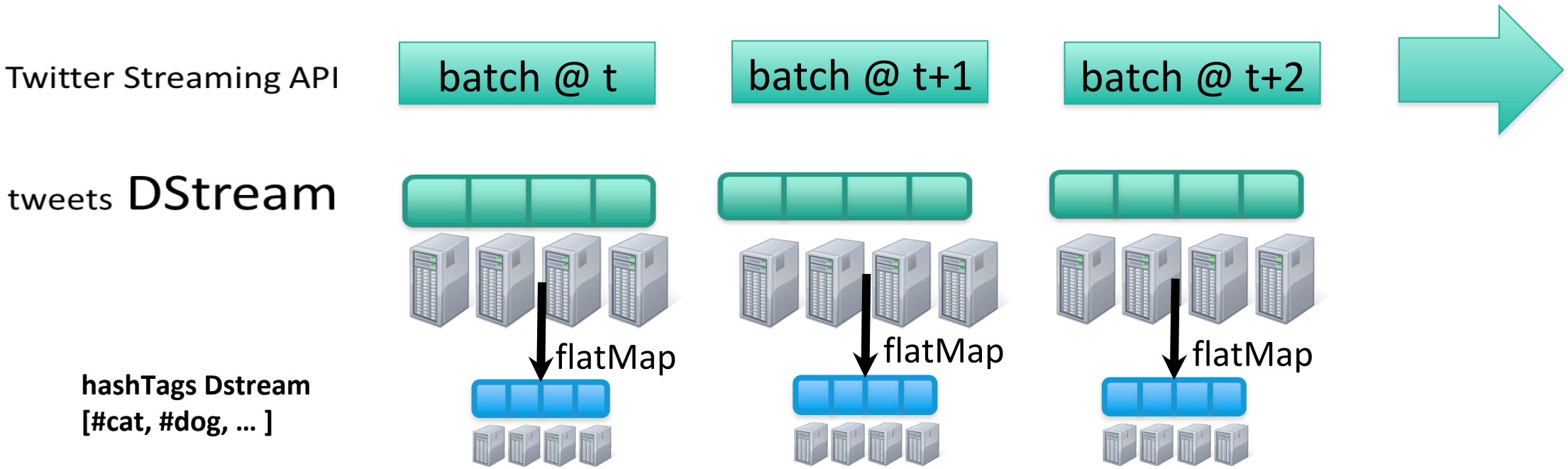
Example 1 – Get hashtags from Twitter

```
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

getTags is a function. A *DStream* is a sequence of RDDs. The function is applied to each RDD. The result is another *DStream*



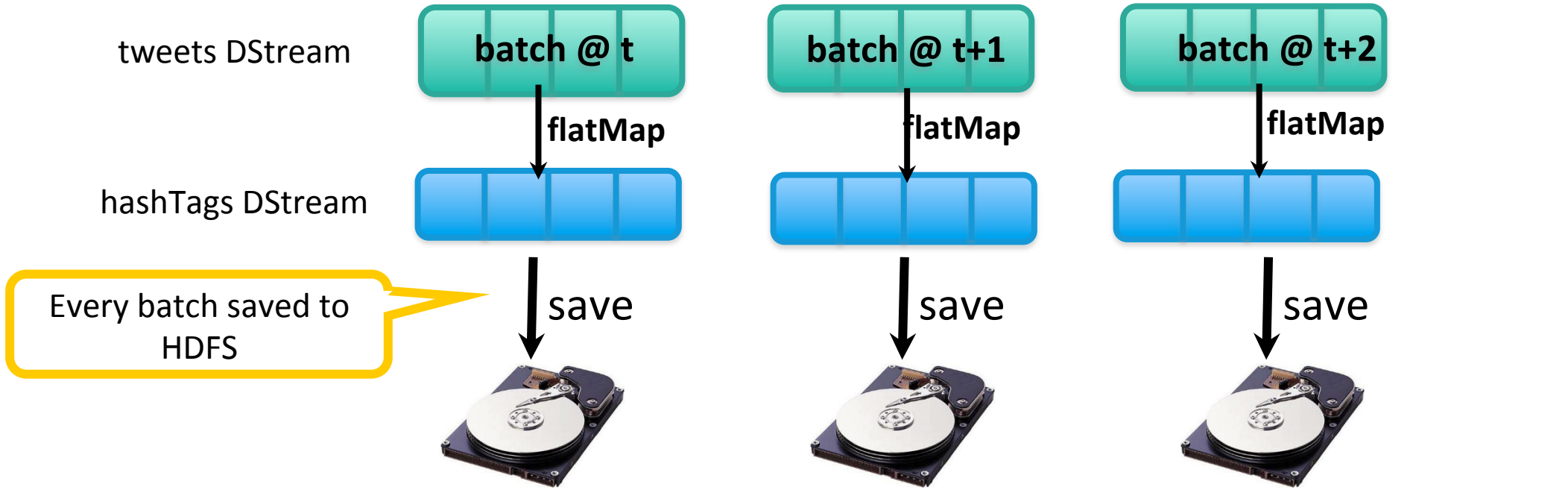
BIG DATA

Example 1 – 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



Java Example

Scala

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

Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>,
<Twitter password>)
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })
hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object to define the transformation

Spark Streaming – Execution of jobs

BIG DATA

Streaming Spark Flow

Dstreams and Receivers

- Twitter, HDFS, Kafka, Flume

Transformations

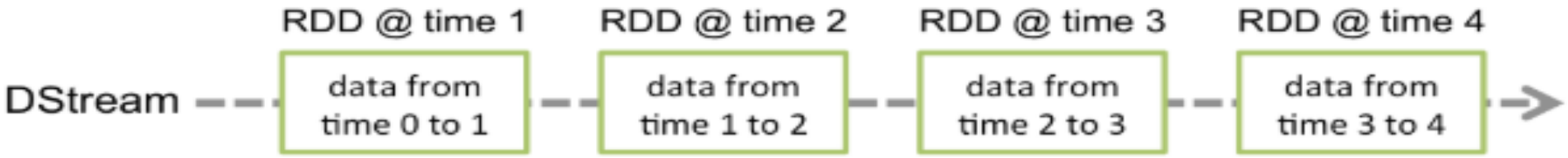
- Standard RDD operations – map, countByValue, reduce, join, …
- Stateful operations – window, countByValueAndWindow, …

Output Operations on Dstreams

- saveAsHadoopFiles – saves to HDFS
- foreach – do anything with each batch of results

Dstreams and Receivers

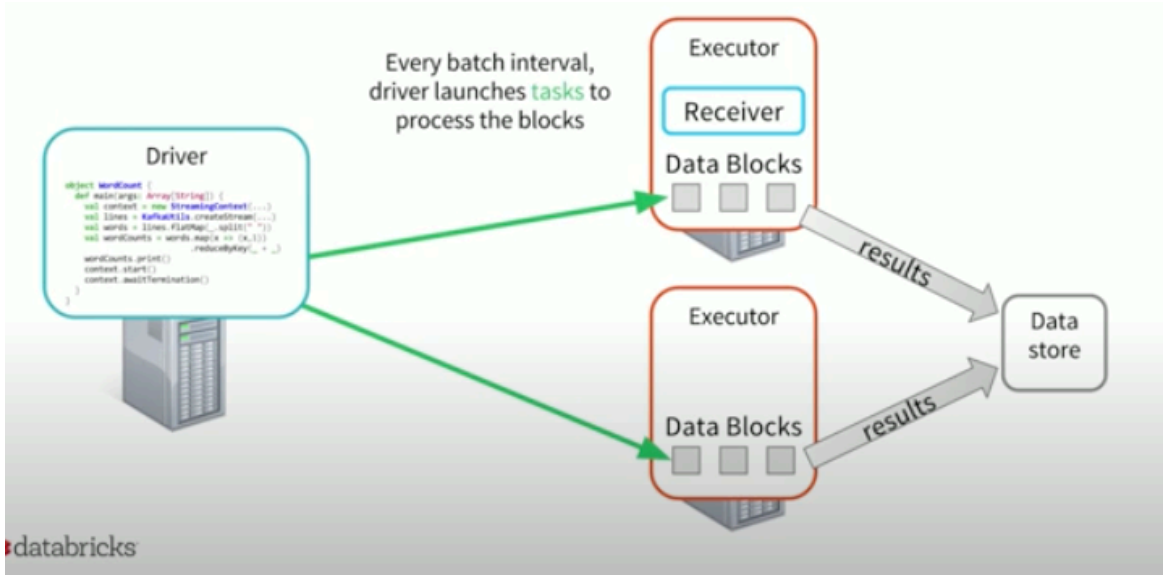
- **Streaming Spark processes data in batches**
  - Together termed the Dstream
- **Every Dstream is associated with a Receiver**
  - Receivers read data from a source and store into Spark Memory for processing
  - Types of sources
    - Basic – file systems and sockets
    - Advanced – Kafka, flume
- **Relationship between Dstream and RDD**



BIG DATA

Dstreams and Receivers

- Streaming Spark processes job
- Starts a Receiver on an executor as a long running job
- Driver starts tasks to process blocks on every interval



Source: https://www.youtube.com/watch?v=NHfggxItokg

Spark Streaming – stateless and stateful processing

BIG DATA

Transformations in Spark

- Stateless transformations
- Stateful Transformations

BIG DATA

Stateless transformations in Spark

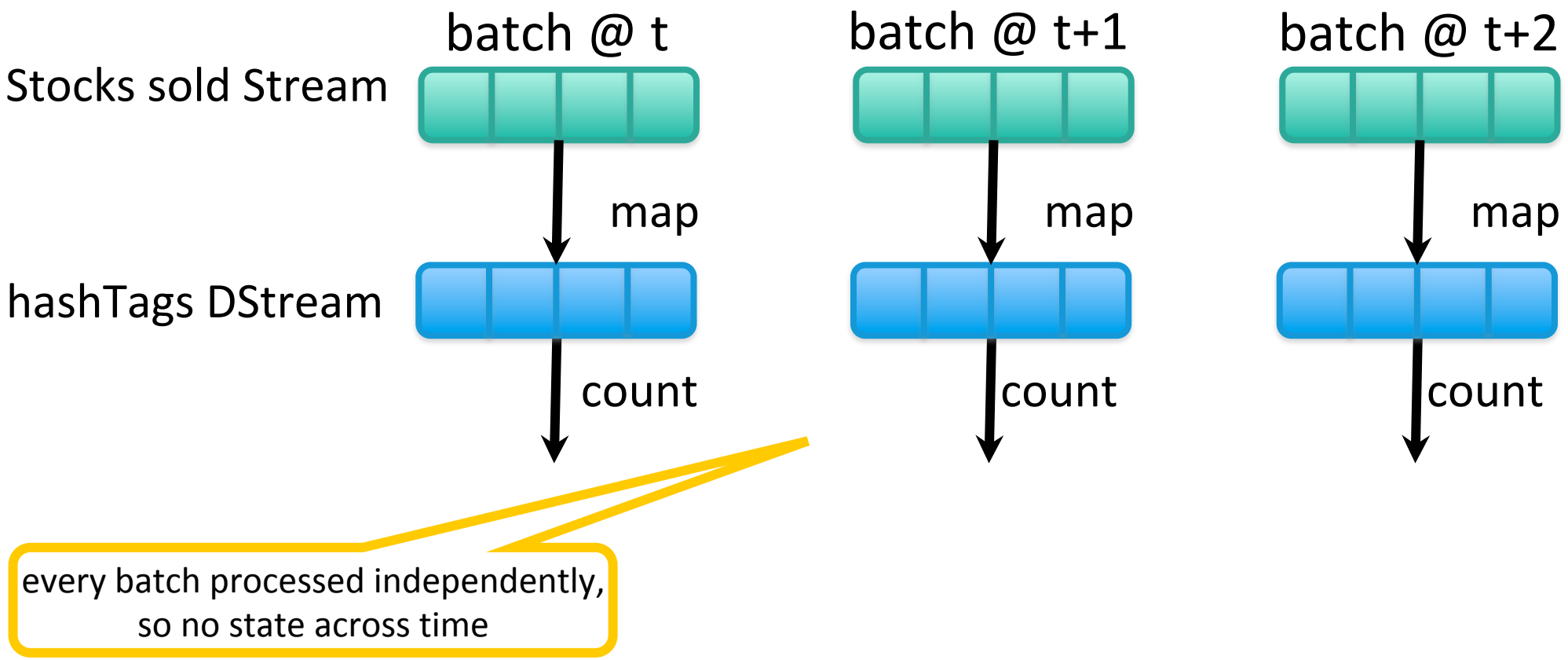
- Transformation is applied to every batch of data independently.
- No information is carried forward between one batch and the next batch
- **Examples**
  - Map()
  - FlatMap()
  - Filter()
  - Repartition()
  - reduceByKey()
  - groupByKey()

Class Exercise : Stateless stream processing (10 mins)

- Consider a Dstream on stock quotes generated similar to earlier that contains
  - A sequence of tuples that contain <company name, stock sold>
  - Need to find total shares sold per company in the last 1 minute
- Show Streaming spark design for the same.

BIG DATA

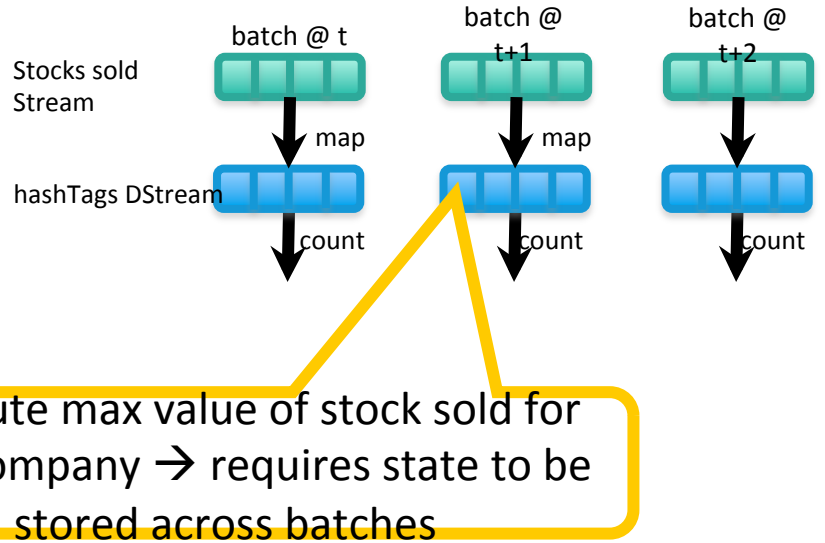
Solution – count stock in every window



BIG DATA

Stateful transformations

- Sometimes we need to keep some state across different batches of data
- For example, what's the max amount of stock sold across the whole day for a company?
  - In this case we need to store max value for each company
- How do we store state across batches?
- First we ensure that data is in pairRDDs
  - Key, value format
  - Helps to ensure that we have a state per key



Stateful processing

BIG DATA

Stateful transformations

- Spark provides two options
  - Window operator – when we want a state to be maintained across short periods of time

  - Session based – where state is maintained for the entire session

BIG DATA

Example 3 – 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

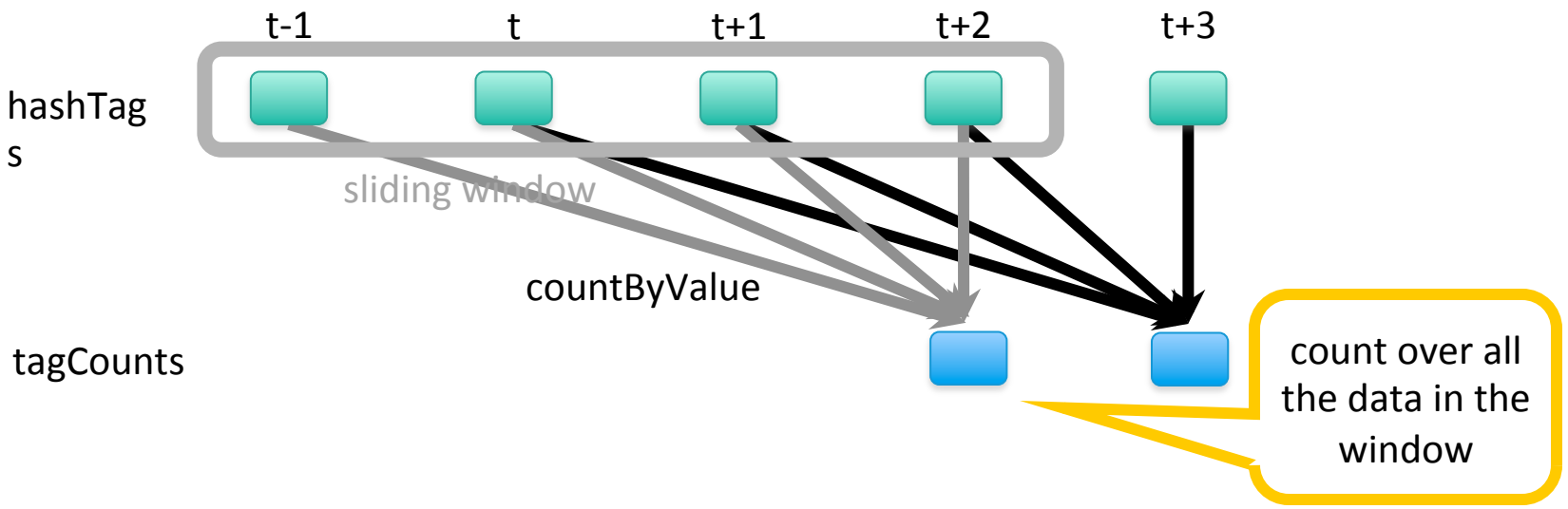
window length

sliding interval

BIG DATA

Example 3 – Counting the hashtags over last 10 mins

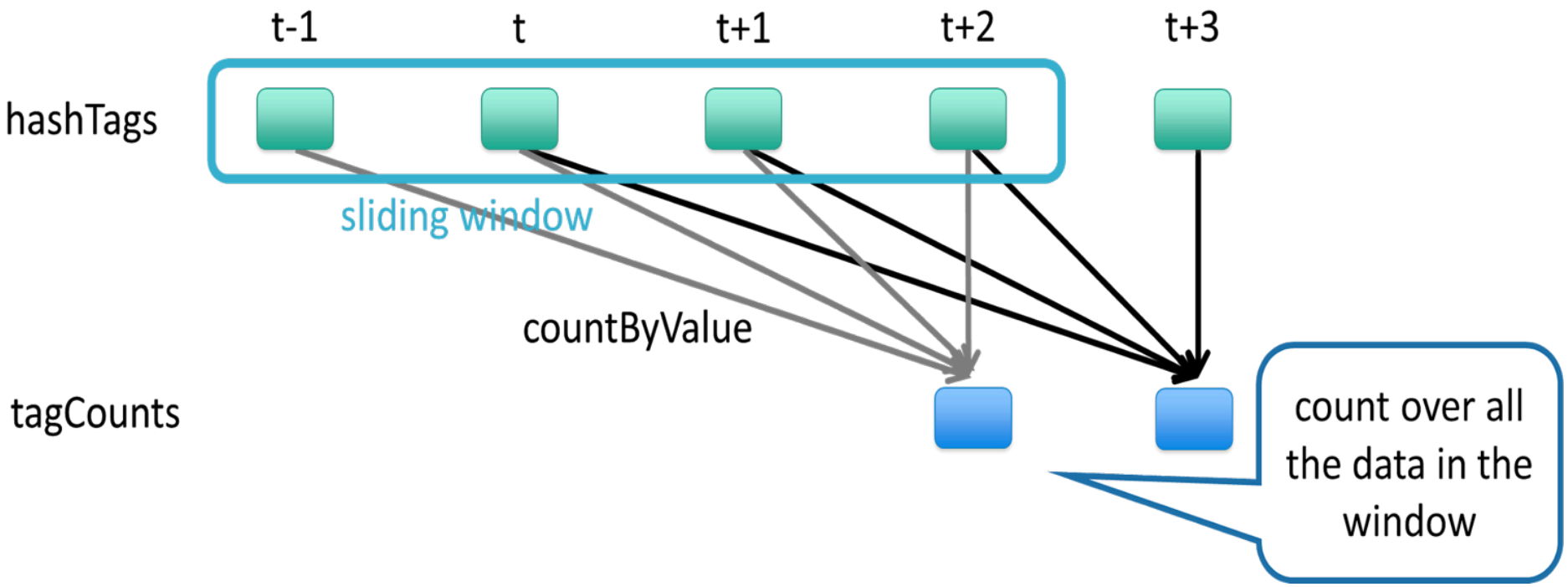
```
val tagCounts = hashTags.window(Minutes(10),Seconds(1)).countByValue()
```



BIG DATA

Class Exercise: (5 mins)

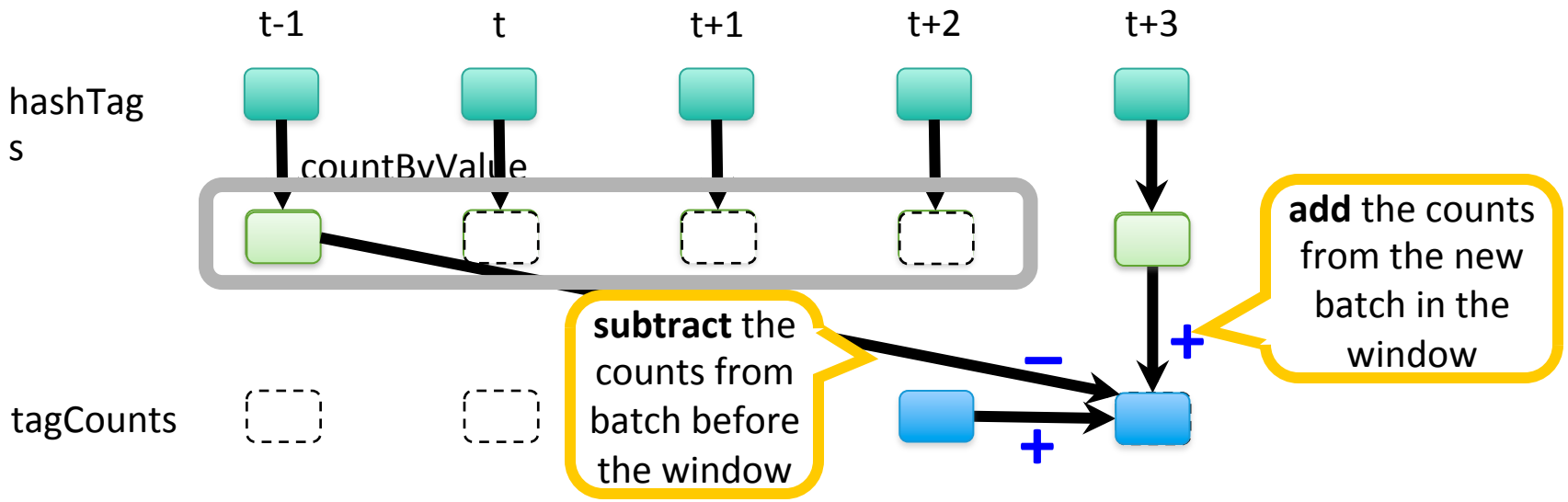
```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



How can we make this count operation more efficient?

Smart window-based countByValue

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



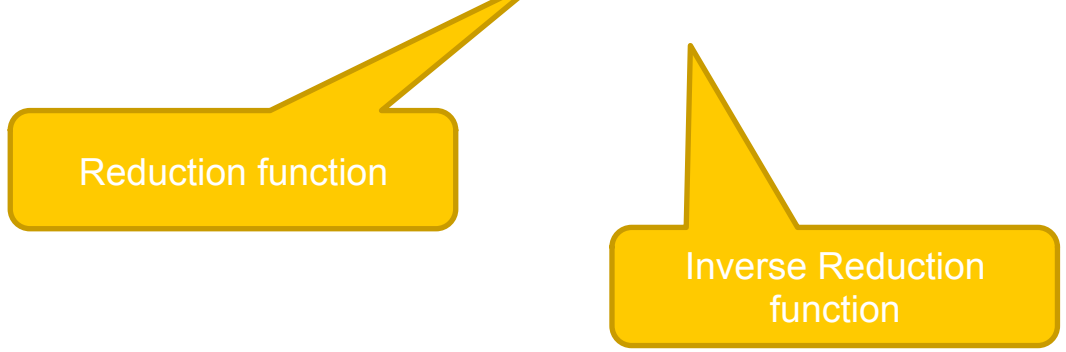
BIG DATA

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), …)
```



BIG DATA

Session based state

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet => updateMood(tweet))
```

- `updateStateByKey` uses the current mood and the mood in the tweet to update the user's mood

BIG DATA

Exercise 4 – Maintaining State (10 minutes)

- Consider the code to the right
  - What has to be the structure of the RDD *tweets*?
    - Hint – note that *updateStateByKey* needs a key
  - What does the function *updateMood* do?
    - Hint – note that it should update per-user mood

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet =>
updateMood(tweet))
```

BIG DATA

Exercise 4 – Maintaining State: solution

- What has to be the structure of the RDD *tweets*?
  - Must consist of key value pairs with user as key and mood as value
    - Dinkar Happy
    - KVS VeryHappy
    - …
- What does the function *updateMood* do?
  - Compute the new mood based upon the old mood and tweet
- Suppose user Jack (key) tweets “Eating icecream” (value)

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet =>
updateMood(tweet))
```

Exercise 4 – Maintaining State: solution

- Consider the code to the right
  - What has to be the structure of the RDD *tweets*?
    - Must consist of key value pairs with user as key and mood as value
      - Dinkar Happy
      - KVS VeryHappy
      - …
  - Suppose user Dinkar (key) tweets "Eating icecream" (value)
    - *updateStateByKey* finds the current mood – Happy
    - current mood (Happy) and tweet (Eating icecream) is passed to *updateMood*
    - *updateMood* calculates new mood as VeryHappy
    - *updateStateByKey* stores the new mood for Dinkar as VeryHappy

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

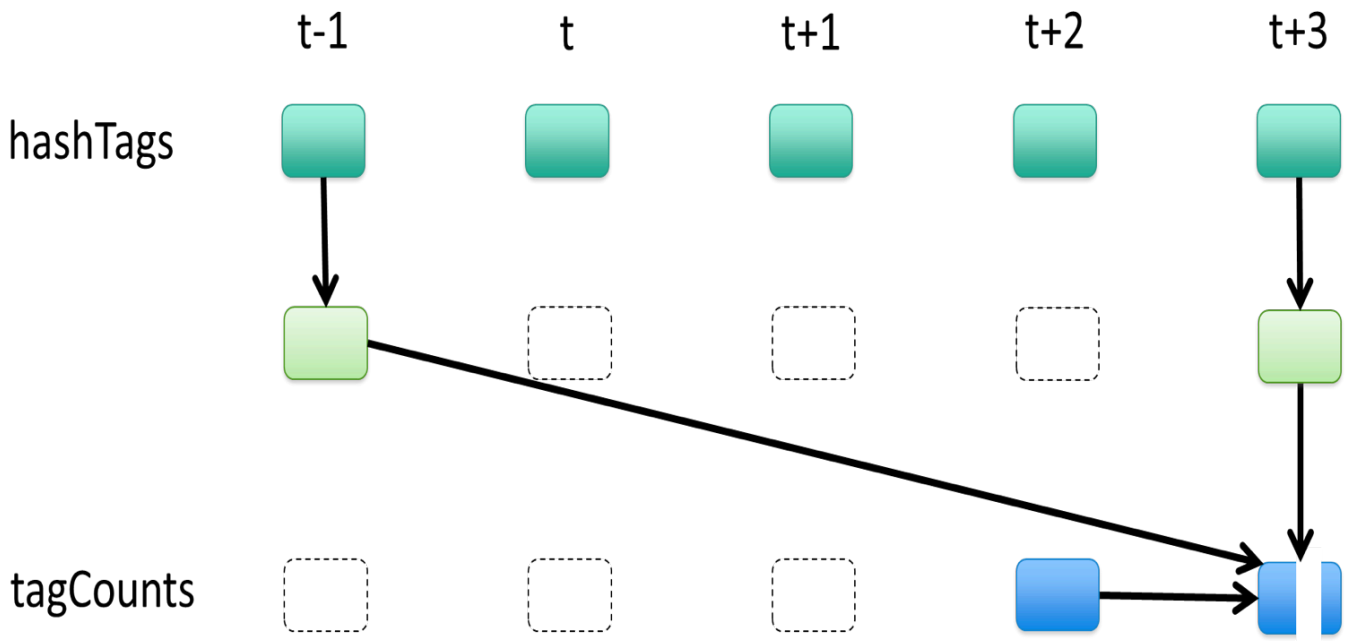
```
tweets.updateStateByKey(tweet =>
updateMood(tweet))
```

Fault tolerant Stateful processing

BIG DATA

Fault-tolerant Stateful Processing

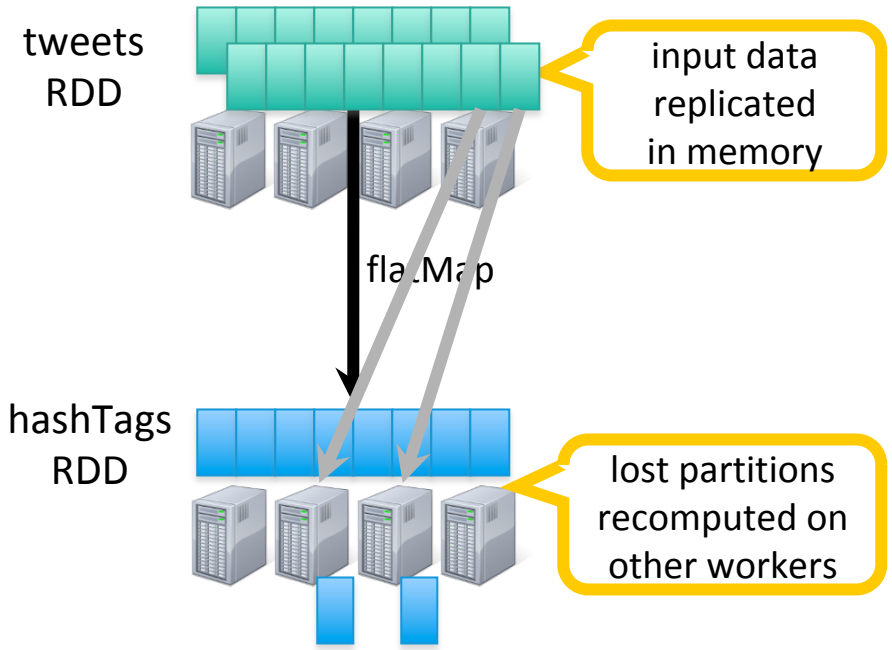
- All intermediate data are RDDs, hence can be recomputed if lost.



BIG DATA

Fault-tolerance

- RDDs 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



BIG DATA

Fault Tolerance

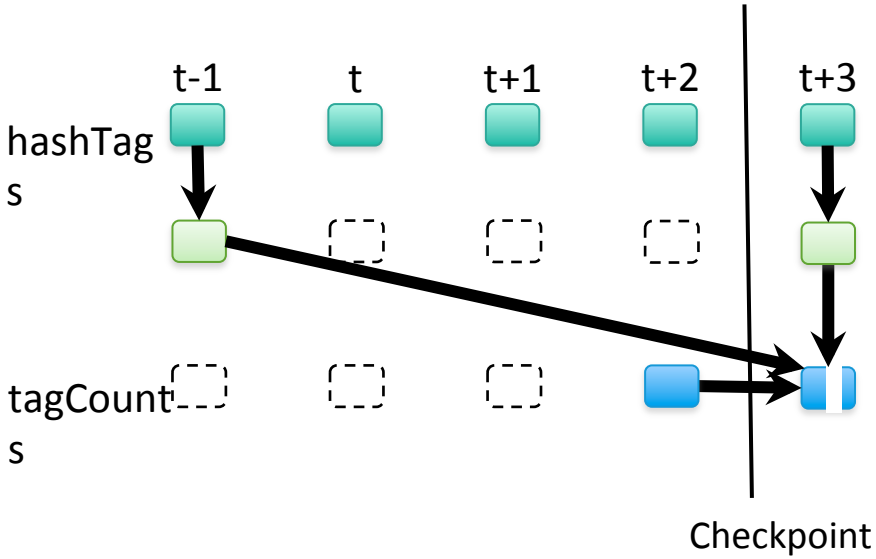
- What happens if there is a failure with
  - Stateless processing
  - Stateful processing

- Stateless
  - Previous history is not required.
  - Processing can just be recomputed
- Stateful
  - State from previous batches required for computation.
  - How much state to retain?

BIG DATA

Checkpointing

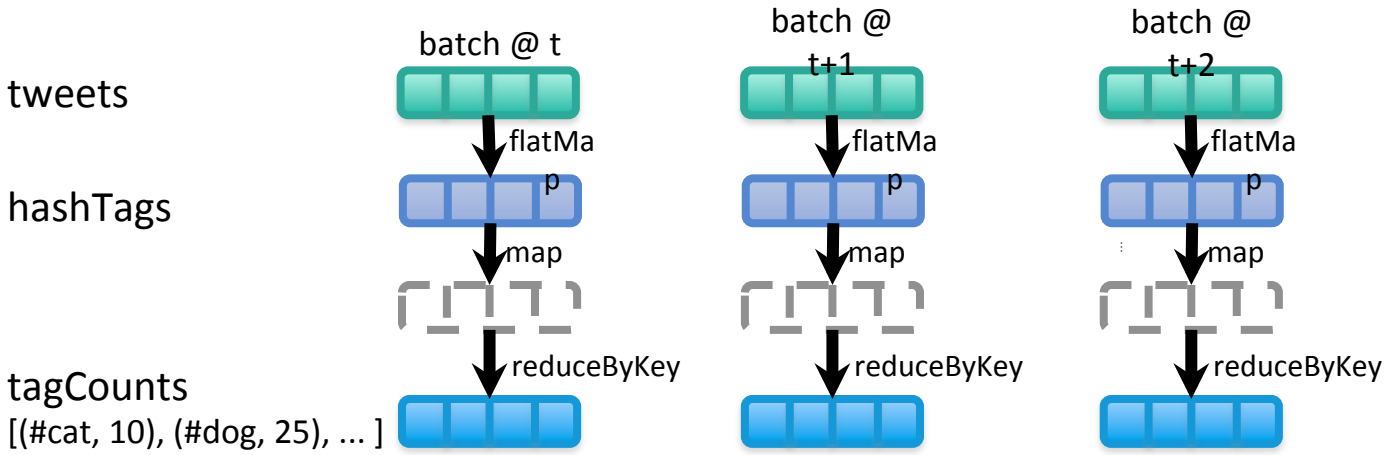
- Sometimes there may be too much data to be stored
  - For a streaming algorithm, may have to store all the streams
- Checkpointing
  - Stores an RDD
  - Forgets the lineage
  - A checkpoint at *t+2* will
    - store the hashTags and tagCounts at t+2
    - Forget the rest of the lineage



BIG DATA

Example – Count the hashtags

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



BIG DATA

Other Interesting Operations

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet => updateMood(tweet))
```

- Do arbitrary Spark RDD computation within DStream
  - Join incoming tweets with a spam file to filter out bad tweets

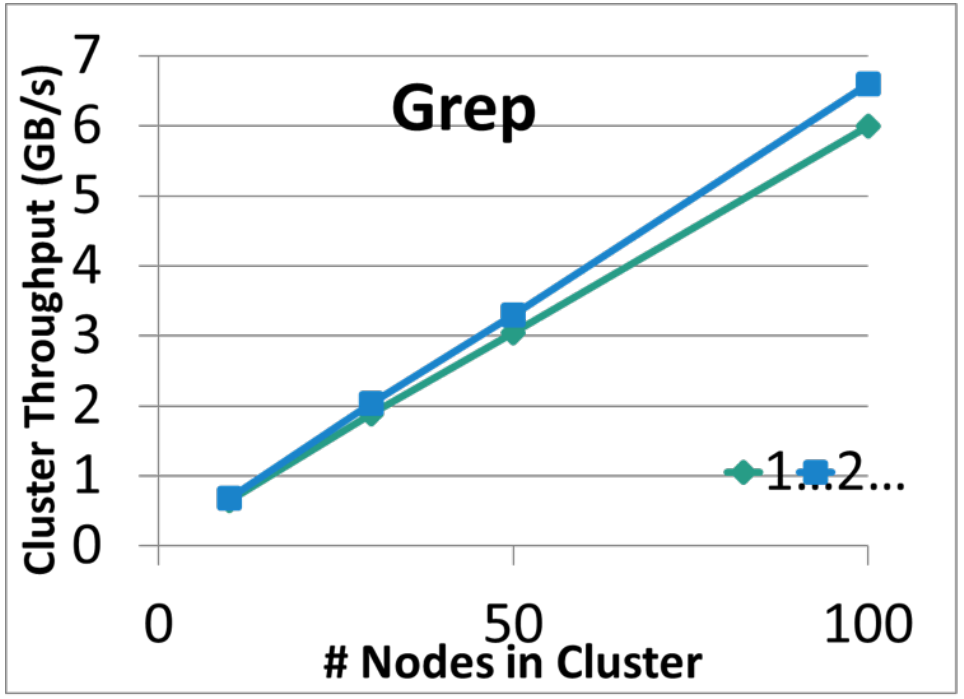
```
tweets.transform(tweetsRDD => {
        tweetsRDD.join(spamHDFSFile).filter(...)
})
```

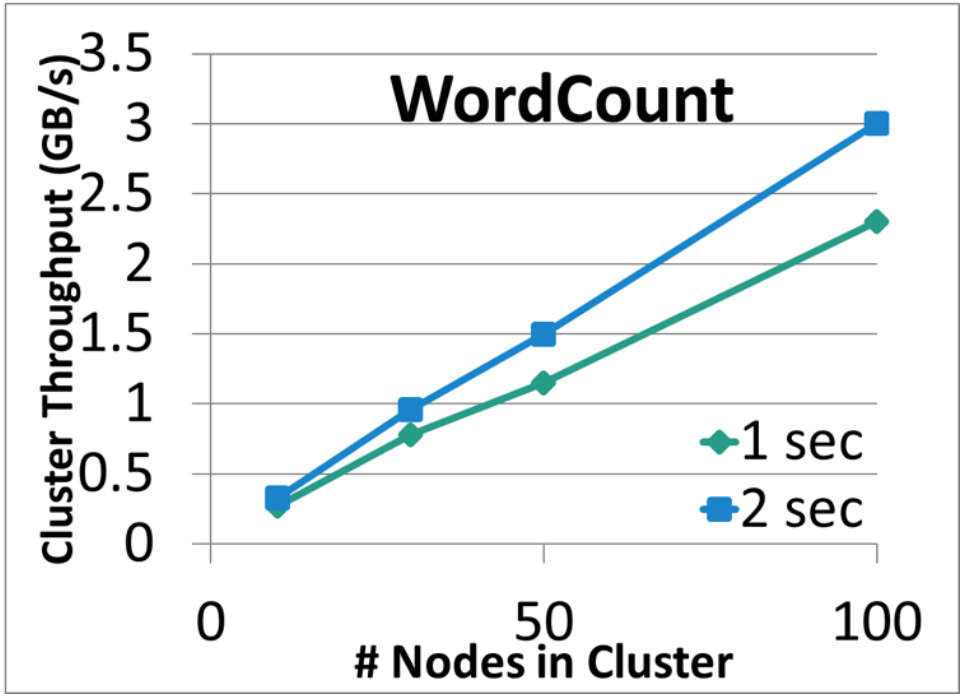
Performance

BIG DATA

Performance

- Can process **6 GB/sec (60M records/sec)** of data on 100 nodes at **sub-second** latency
  - Tested with 100 streams of data on 100 EC2 instances with 4 cores each

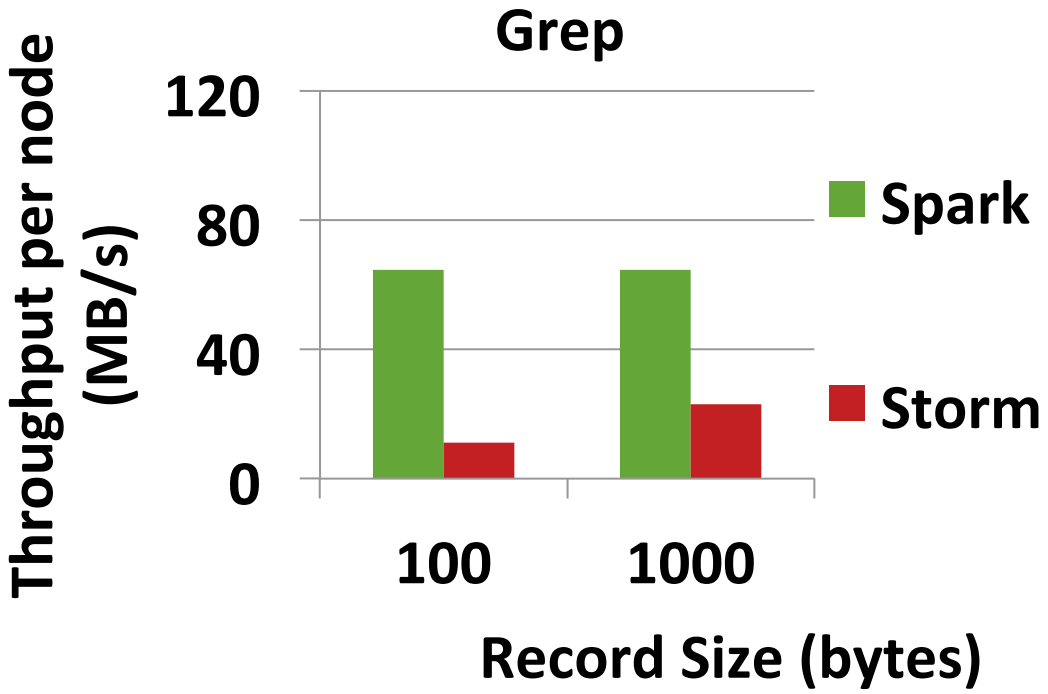


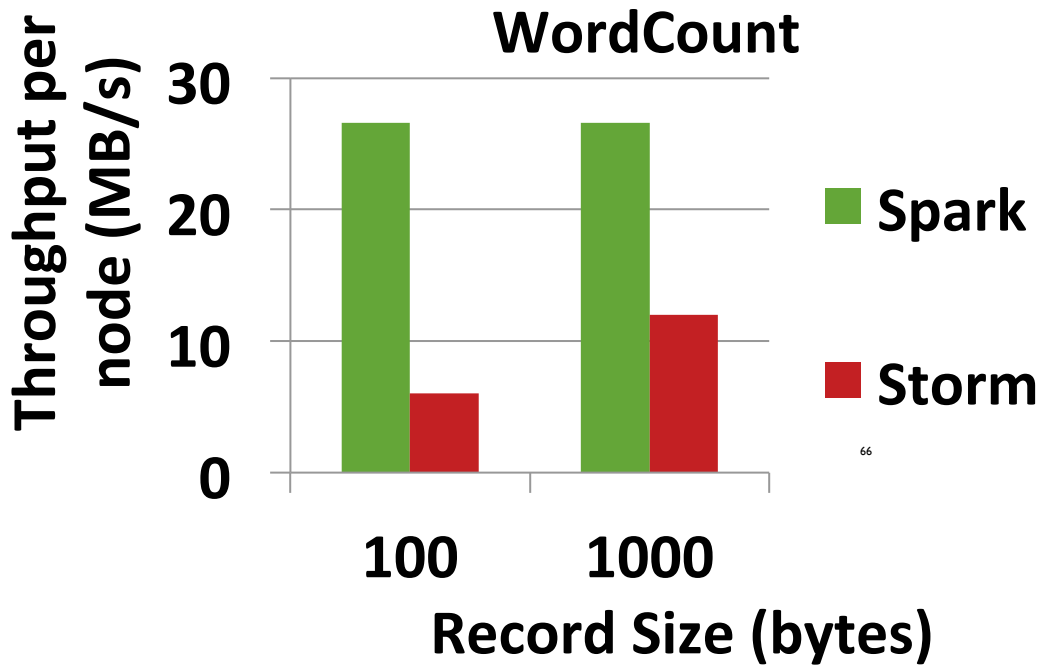


Comparison with Storm and S4

Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node

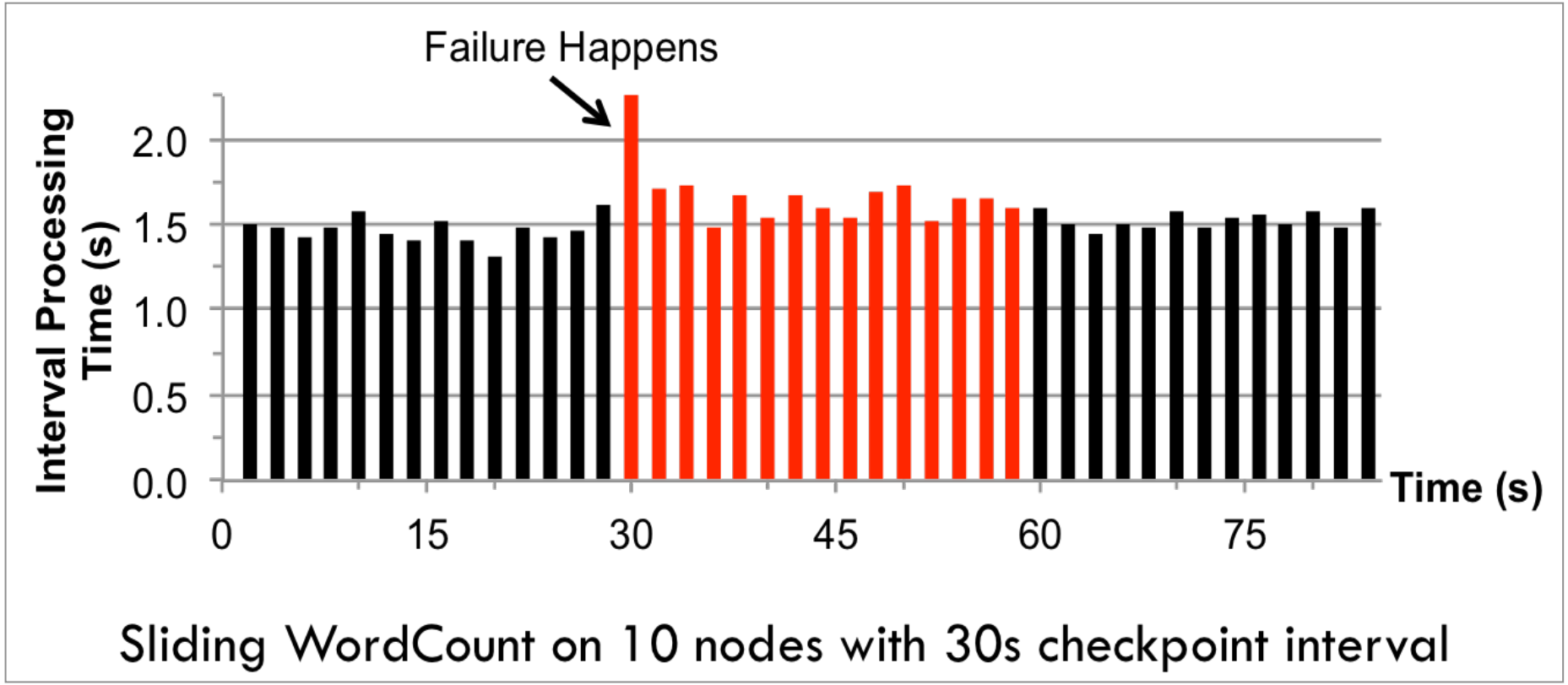




BIG DATA

Fast Fault Recovery

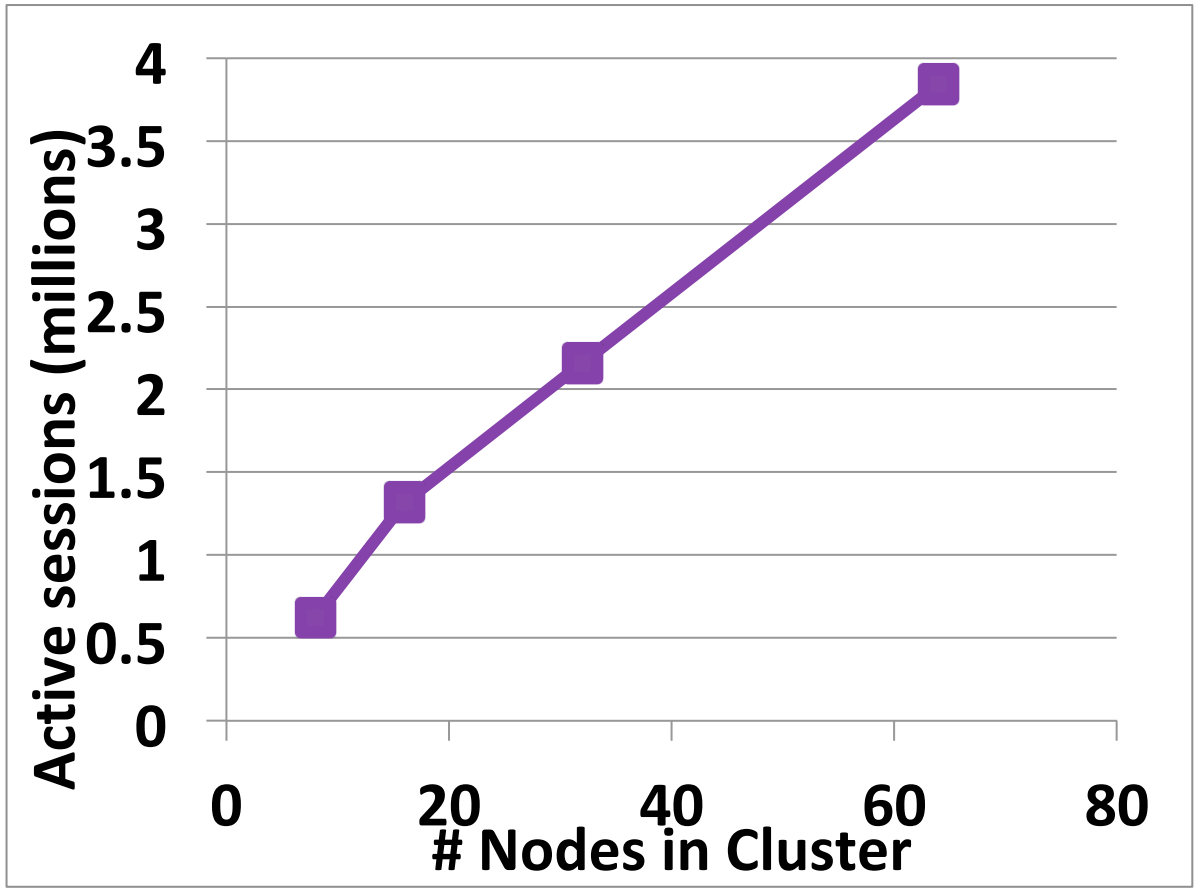
- Recovers from faults/stragglers within **1 sec**



BIG DATA

Real Applications: Conviva

- Real-time monitoring of video metadata

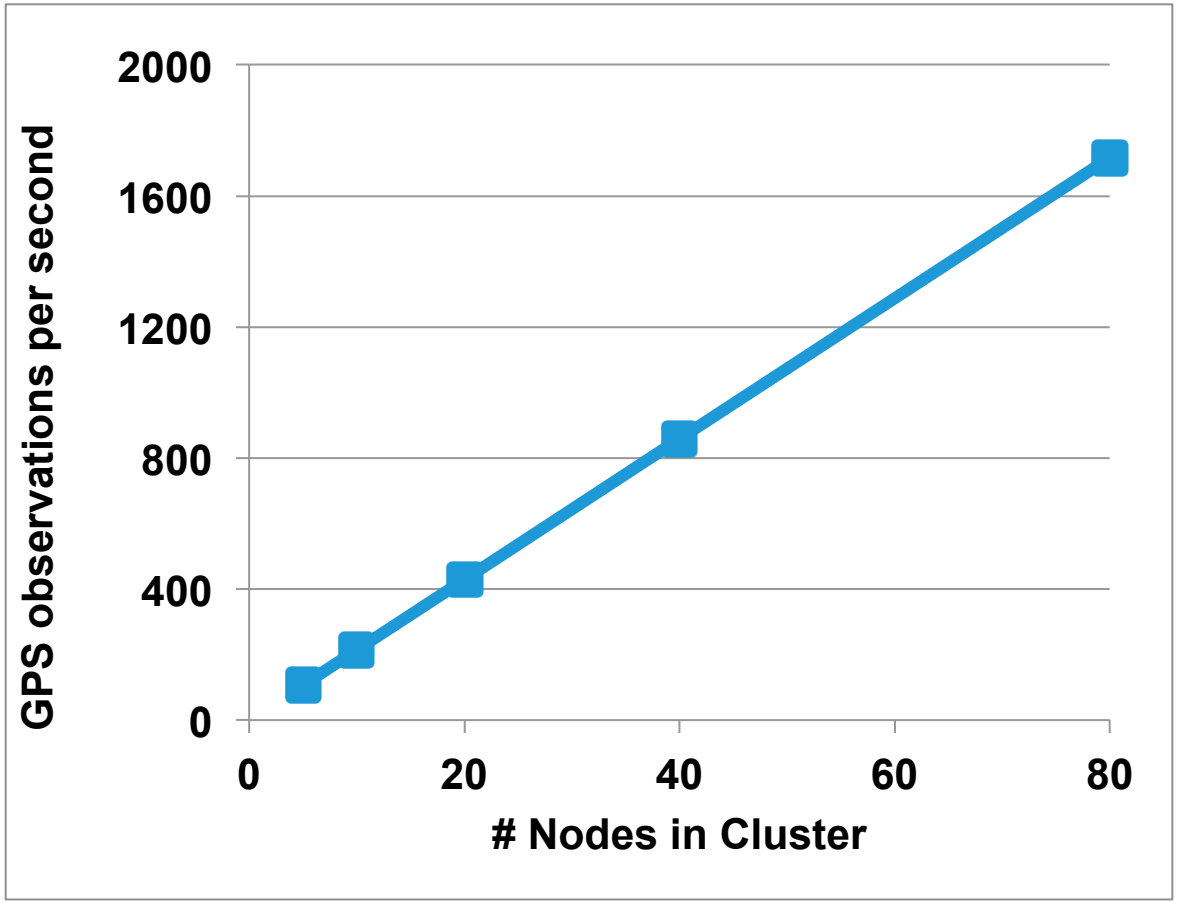


- Achieved 1-2 second latency
- Millions of video sessions processed
- Scales linearly with cluster size

BIG DATA

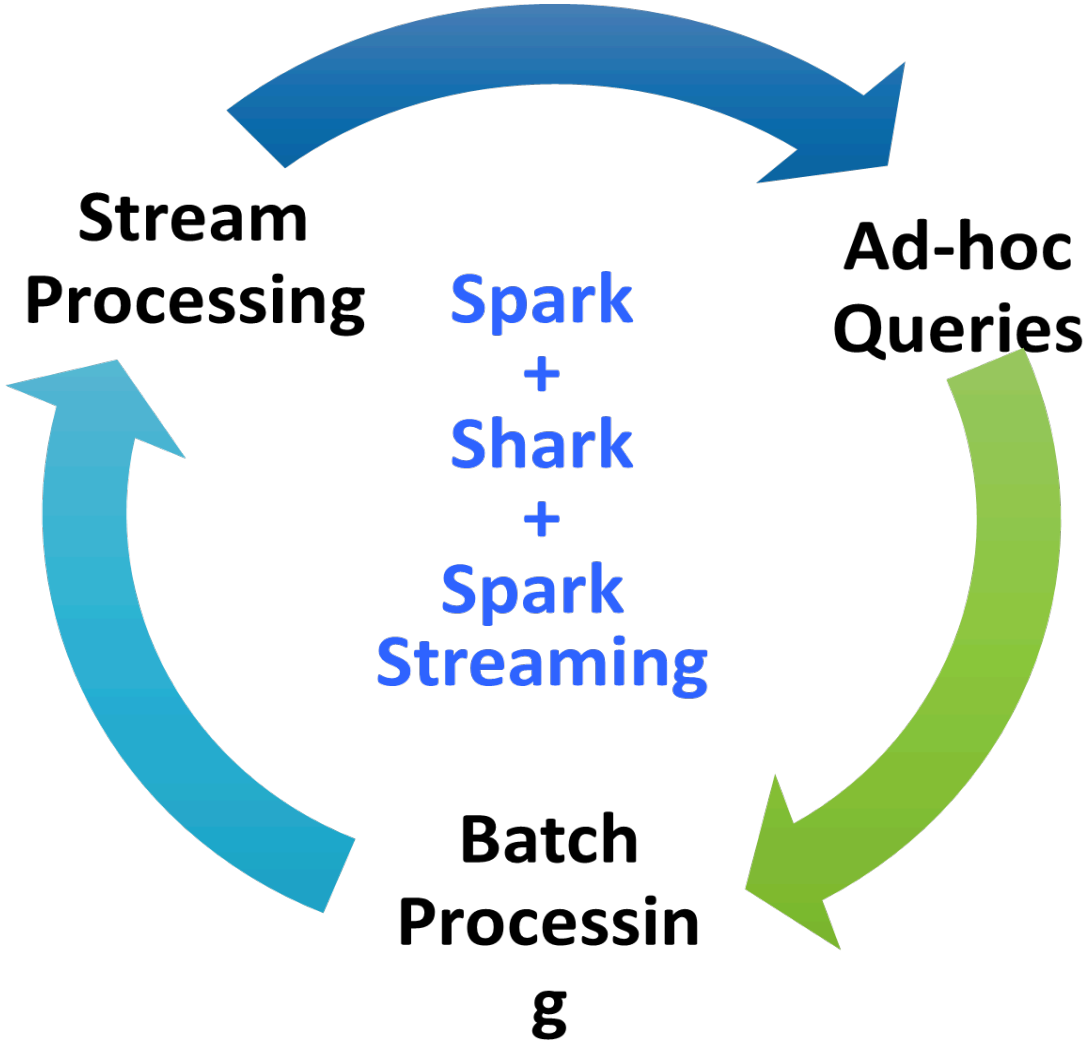
Real Applications: Mobile Millennium Project

- Traffic transit time estimation using online machine learning on GPS observations



- Markov chain Monte Carlo simulations on GPS observations
- Very CPU intensive, requires dozens of machines for useful computation
- Scales linearly with cluster size

Vision - one stack to rule them all



BIG DATA

Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

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

Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

BIG DATA

Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
```

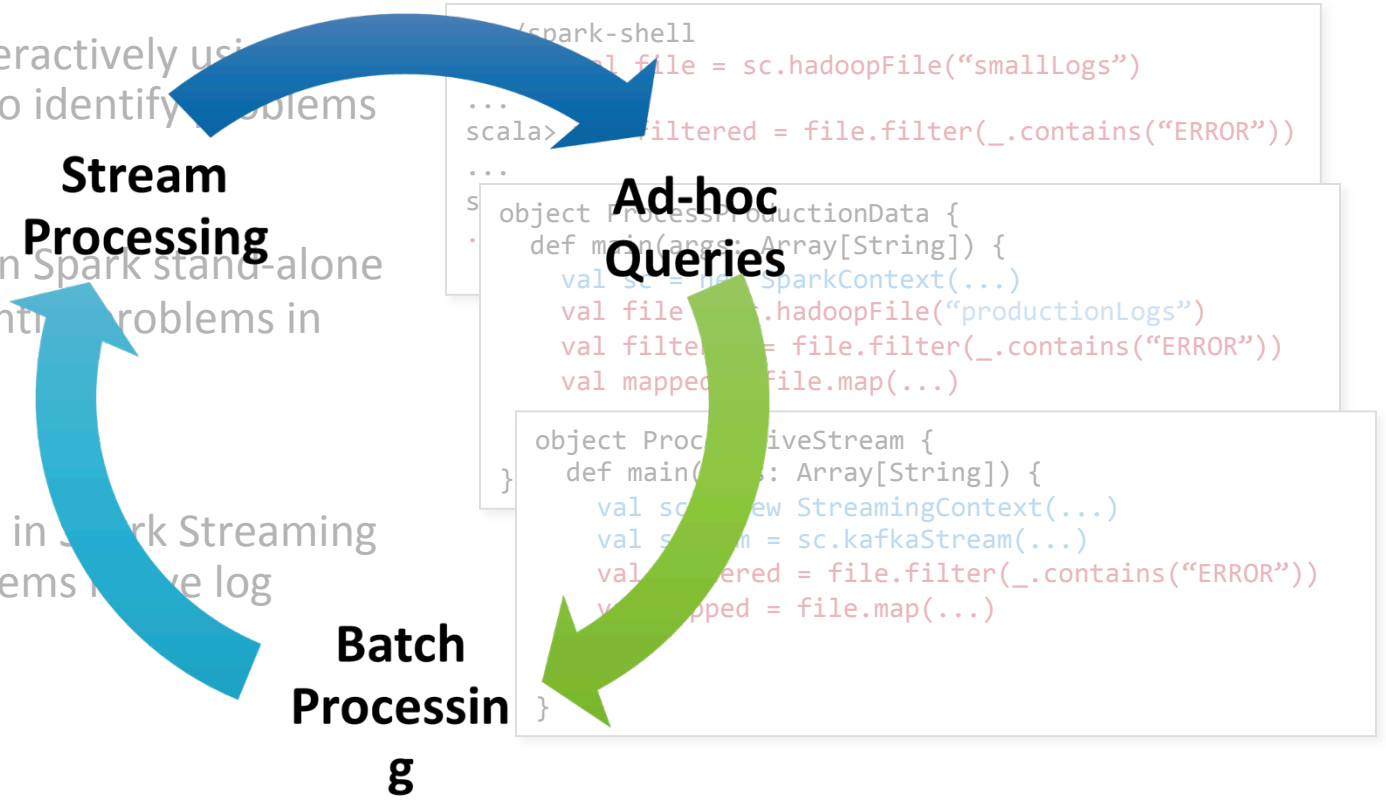
```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
```

```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
  }
}
```

BIG DATA

Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems live log streams



BIG DATA

Alpha Release with Spark 0.7

- Integrated with Spark 0.7
  - Import **spark.streaming** to get all the functionality
- Both Java and Scala API
- Give it a spin!
  - Run locally or in a cluster
- Try it out in the hands-on tutorial later today

BIG DATA

Limitations

- Streaming Spark processes data in batches
  - [?] Near Real Time

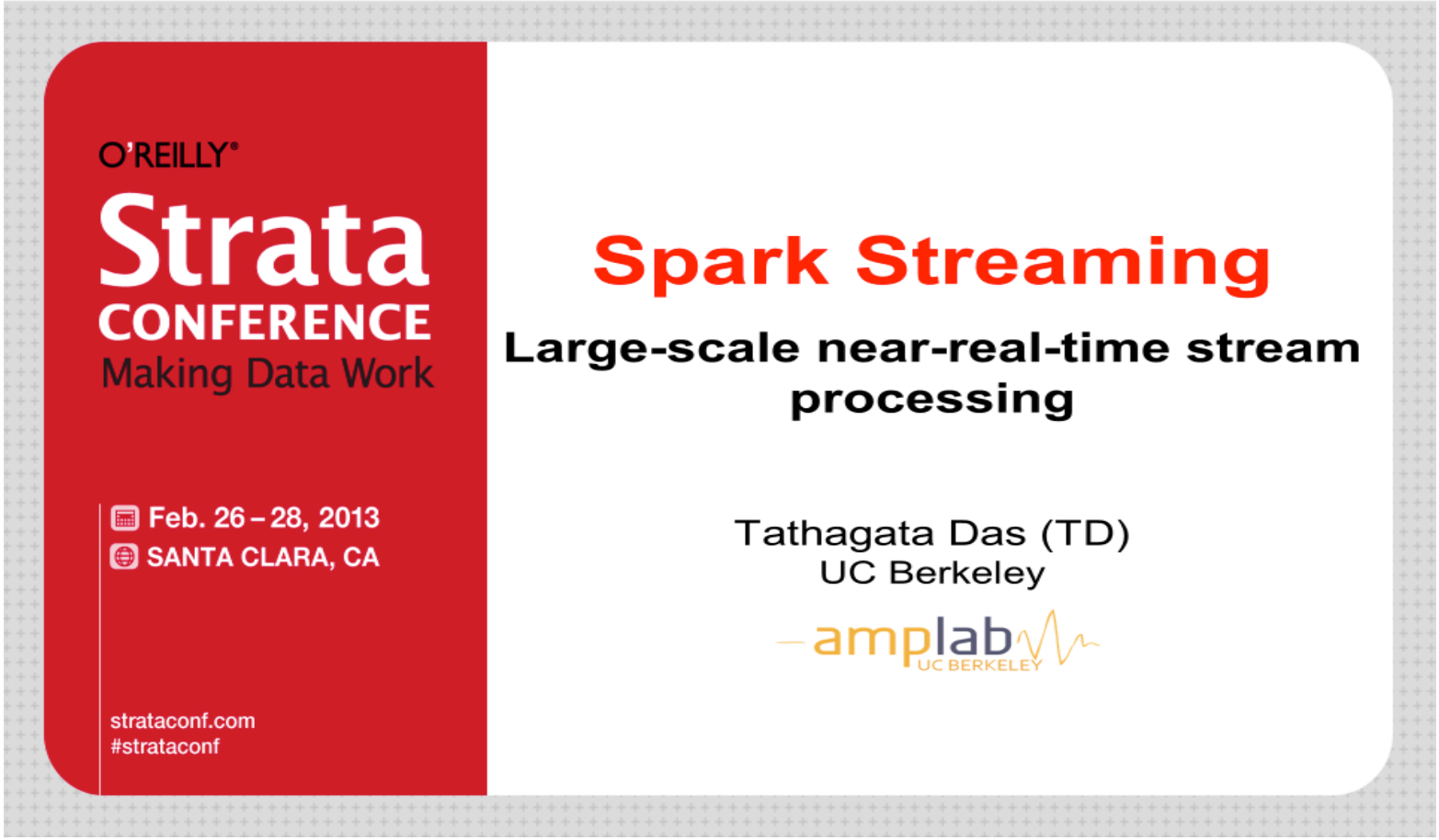
  - Not necessarily acceptable for certain scenarios

Summary

- Stream processing framework that is ...
  - Scalable to large clusters
  - Achieves second-scale latencies
  - Has simple programming model
  - Integrates with batch & interactive workloads
  - Ensures efficient fault-tolerance in stateful computations

- For more information, checkout the paper: http://tinyurl.com/dstreams

Source:



Video : Use Cases and Design Patterns for SPARK STREAMING,
https://www.youtube.com/watch?v=NHfggxItokg

BIG DATA

Kafka Architecture and Usage

K V Subramaniam
Computer Science and Engineering

BIG DATA

Kafka Overview

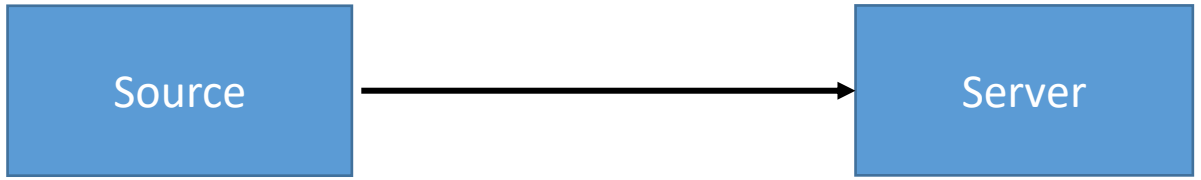
- Need for data pipelines
- Kafka architecture
- Kafka components – messaging model
- Communication and Routing
- Messages
- Scaling
- Fault Tolerance

Kafka : Need for data pipelines

BIG DATA

The need for processing events

- Stream processing requires requires processing of events
- Think of an event as something that happens at a time
- Events are processed on the server
- For example
  - “Student with ID 23489 entered building”

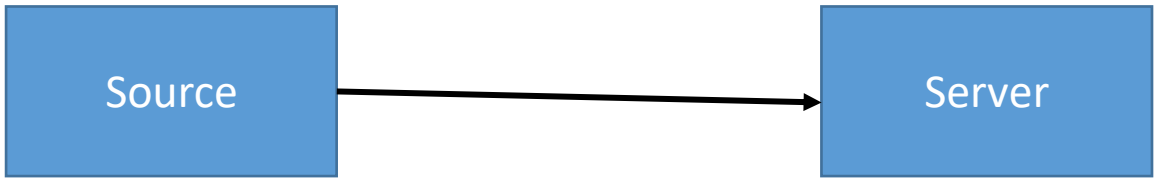


Data pipelines starts like this.

BIG DATA

What are Data Pipelines

- Data sources are varied and
  - Stream data at varied rates
  - Are very lightweight and not capable of compute
- Analysis is done on servers
- Data has to be transferred from sources to the compute servers
- Point to point connection is a simple way to connect

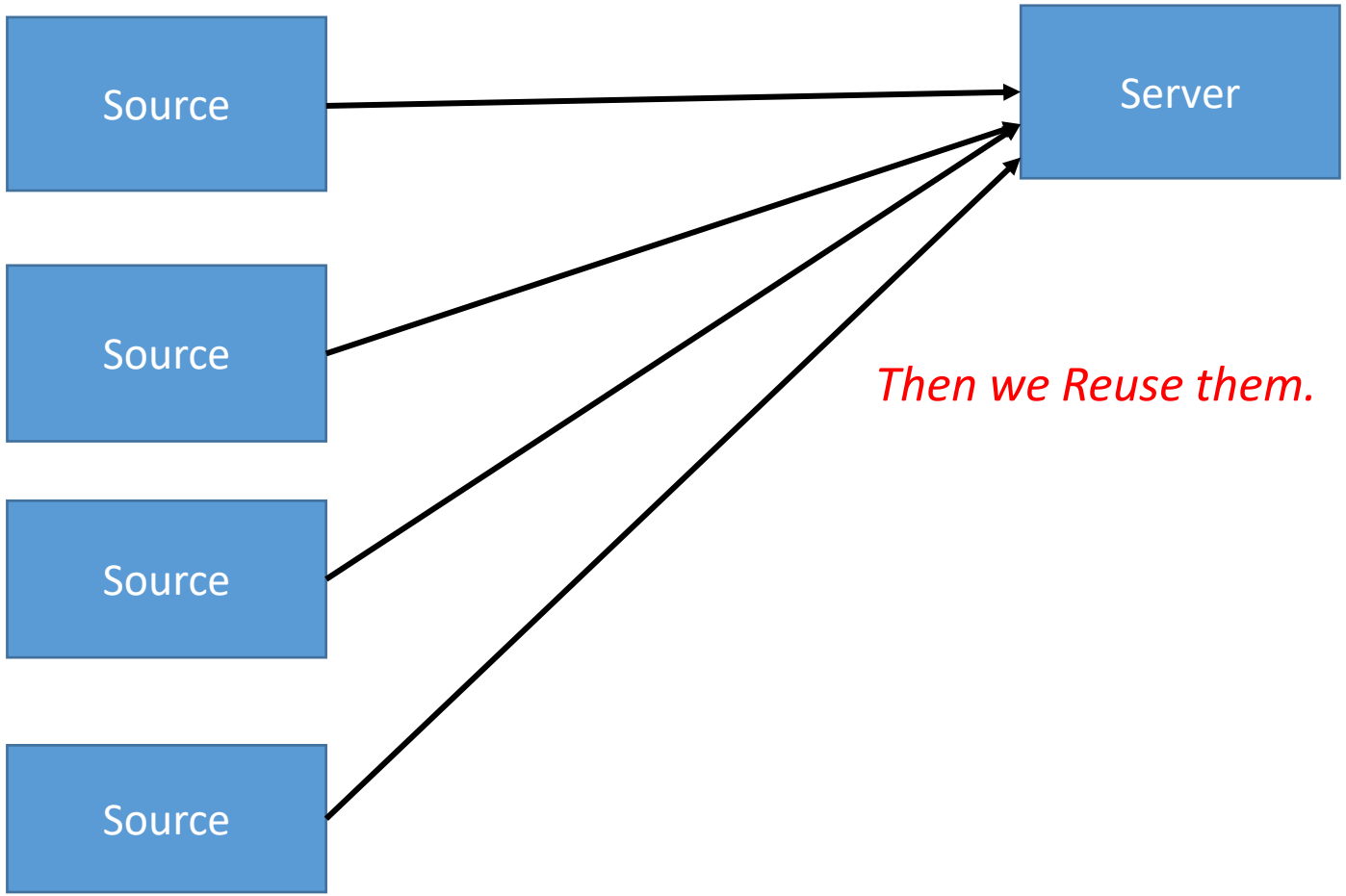


Data pipelines starts like this.

BIG DATA

Handling multiple sources

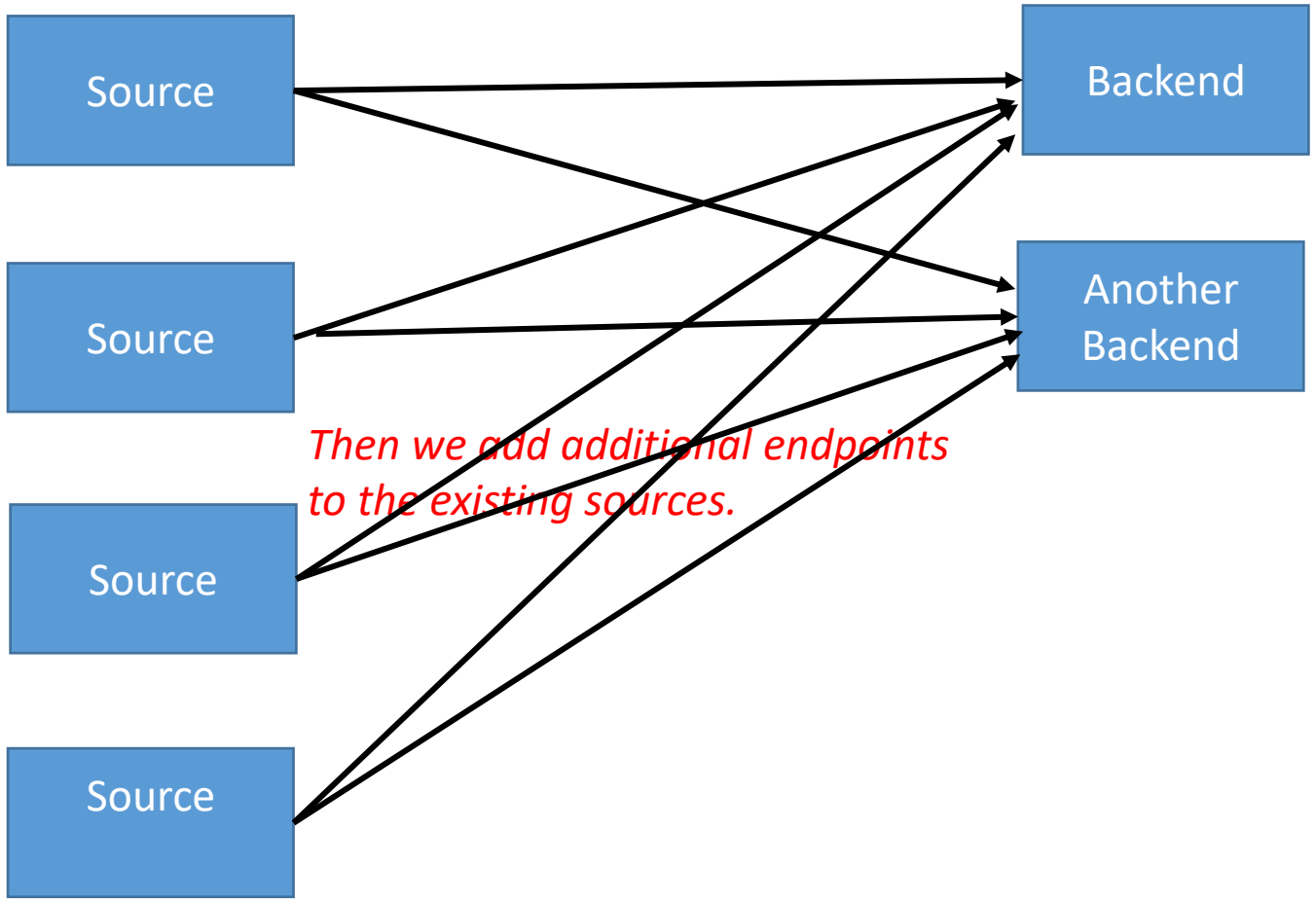
- But sources are many
- We often have to read from multiple data sources
  - E.g: Multiple cameras connected to a central processing system
- We can connect multiple clients each over a pool of connections



BIG DATA

Need for multiple backends

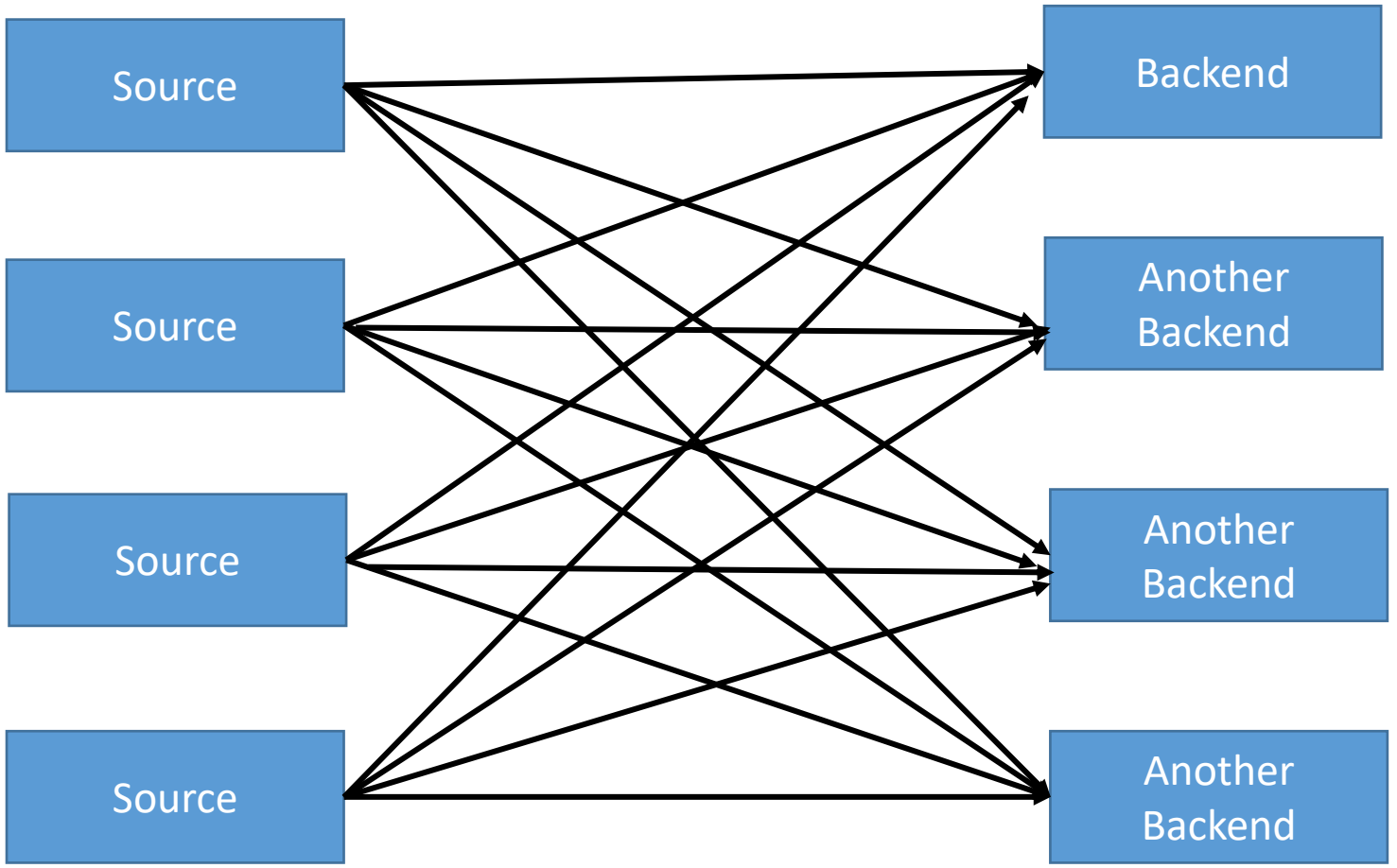
- But we might have multiple servers
- On which to process the same data
- Use case: stock trading – may have different analysts wanting to analyze the behavior



BIG DATA

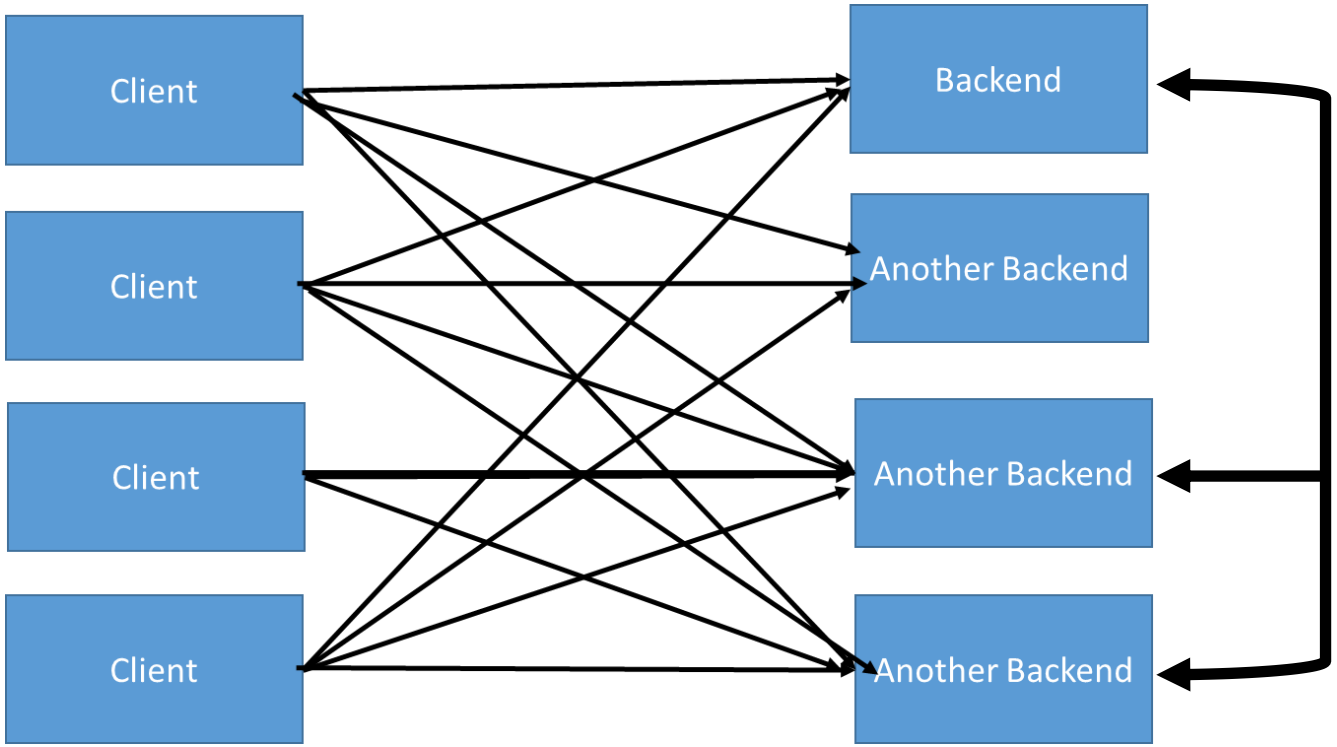
The maze of connections

- Then it starts to look like this.



BIG DATA

The maze of connections



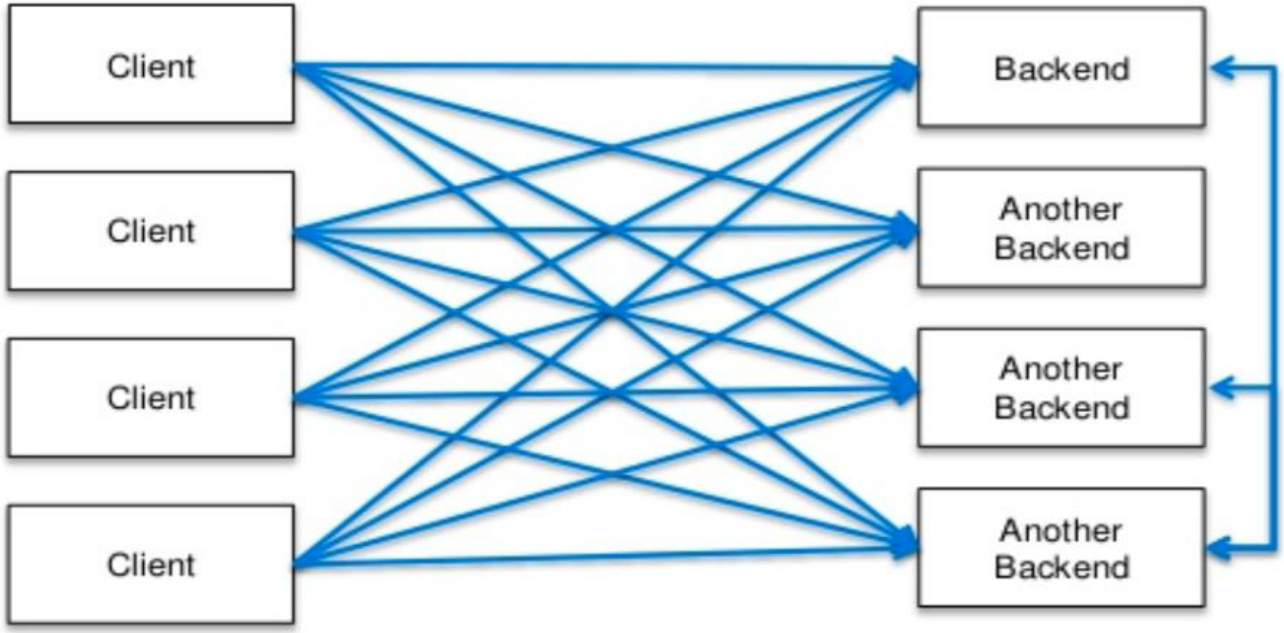
With may be some of these.

- As Distributed systems and services increasingly become part of modern architecture, this makes for a fragile system.

BIG DATA

Exercise 1 (5 minutes)

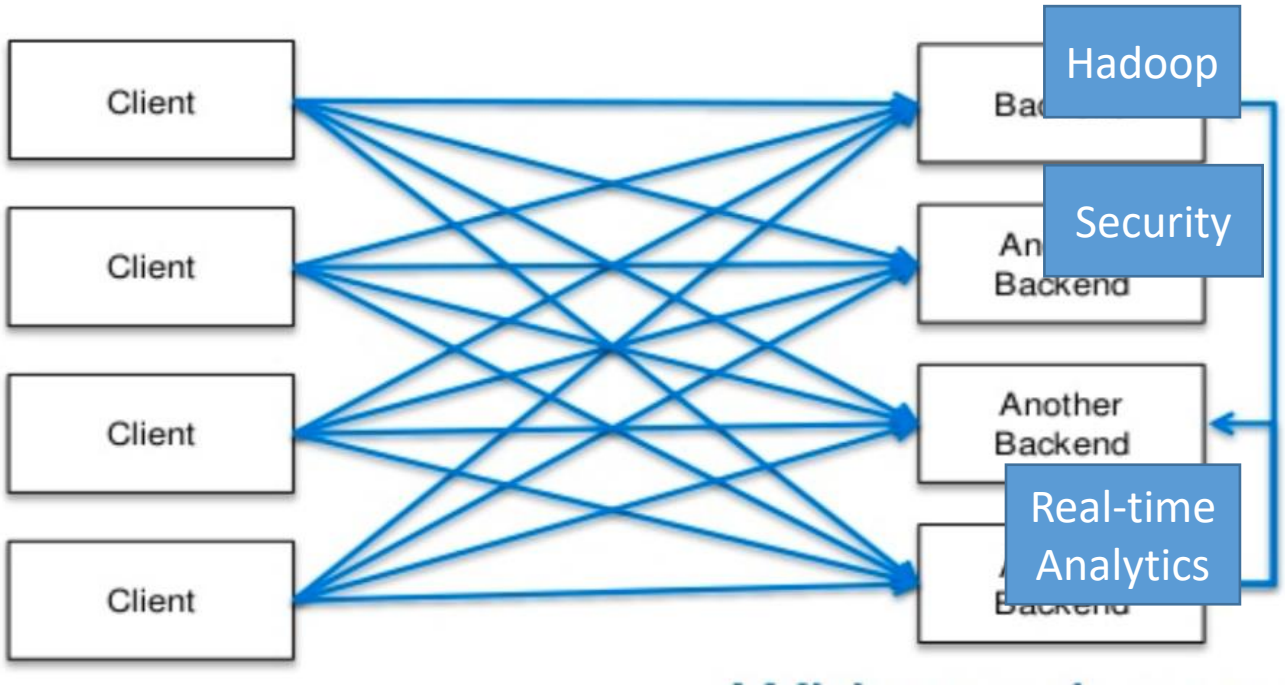
- Give an example of how datapipelines could be used
  - What are some examples of backends?



BIG DATA

Exercise 1 (5 minutes)

- Give an example of how datapipelines could be used
  - What are some examples of backends?

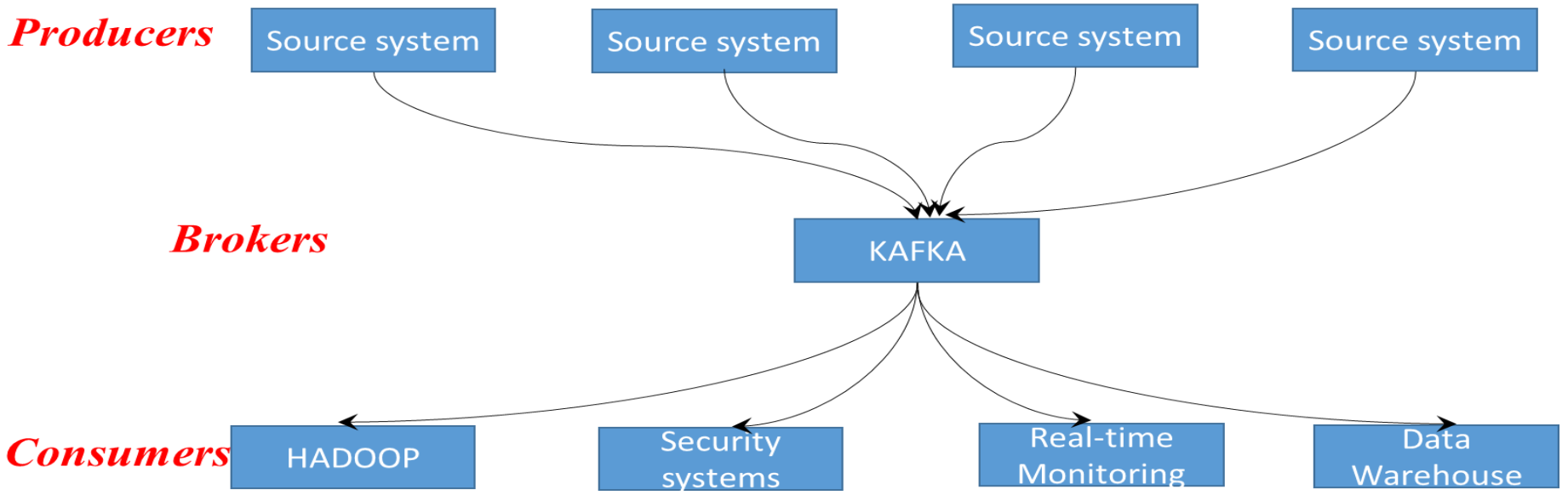


Kafka architecture

BIG DATA

Why Kafka?

- Can we have an intermediary that connects
  - Different sources
  - Multiple backends
- Decouple the pipeline so that producers and consumers do not need to know about each other



Kafka Decouples Data Pipelines

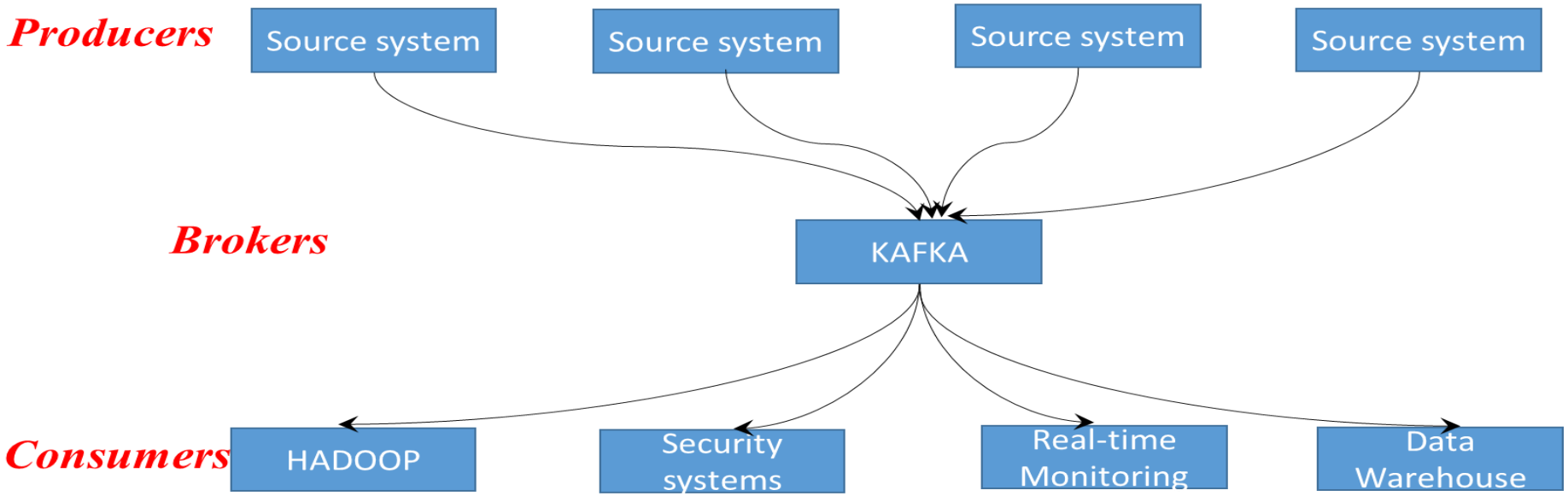
Kafka Components – Publish Subscribe Model

BIG DATA

KAFKA components

The Three essential Elements /Components:

1. Publisher
2. Subscriber
3. Communication Infrastruture



Kafka Decouples Data Pipelines

PES UNIVERSITY ONLINE

Publishers and Subscribers

So,

Q) What does a Publisher do ..?

A. It publishes messages to the Communication Infrastructure.

Okay . So

Q) What does a Subscriber do ..?

A. It subscribes to a category of messages.

PES UNIVERSITY ONLINE

The Routing Mystery

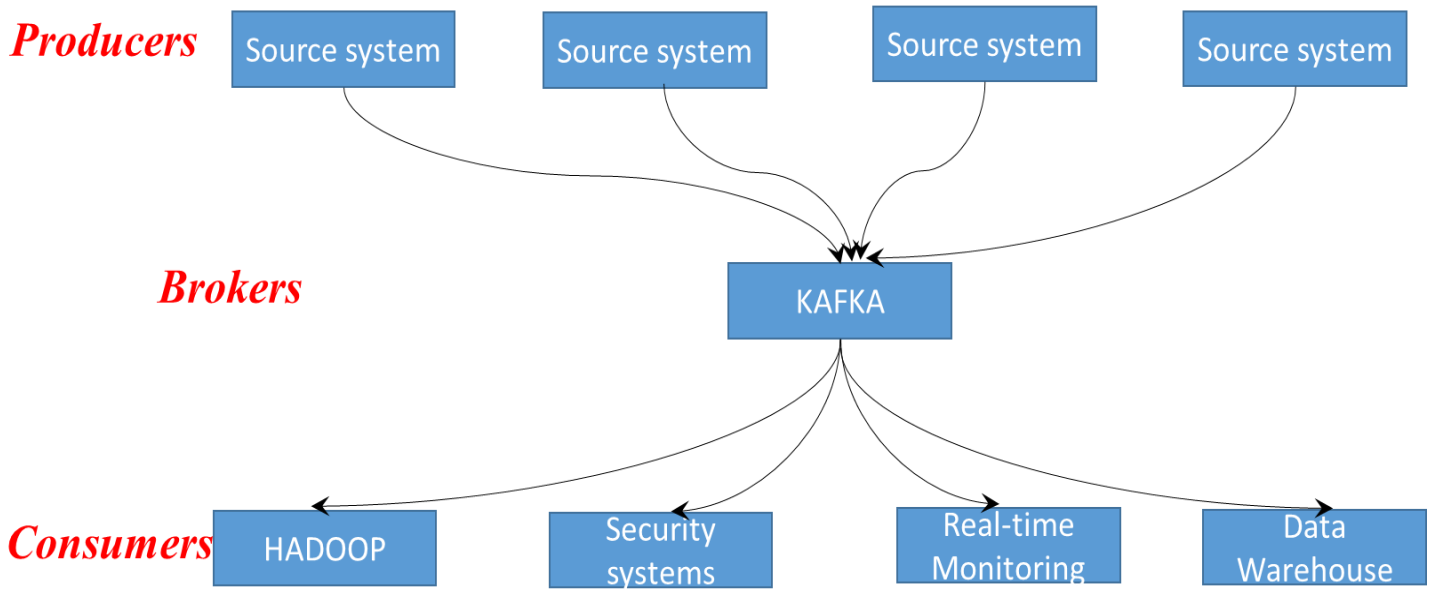
What If I tell you
I am the Publisher
and
I have no idea about
the Subscriber.

What If I tell you
I am the Subscriber
and
I have no idea about
the Publisher.

BIG DATA

Role of a producer

- Defines what data it wants to send
- Packages that go into a message and..
- It publishes messages on the communication infrastructure
- Simplest part → called the *publisher*

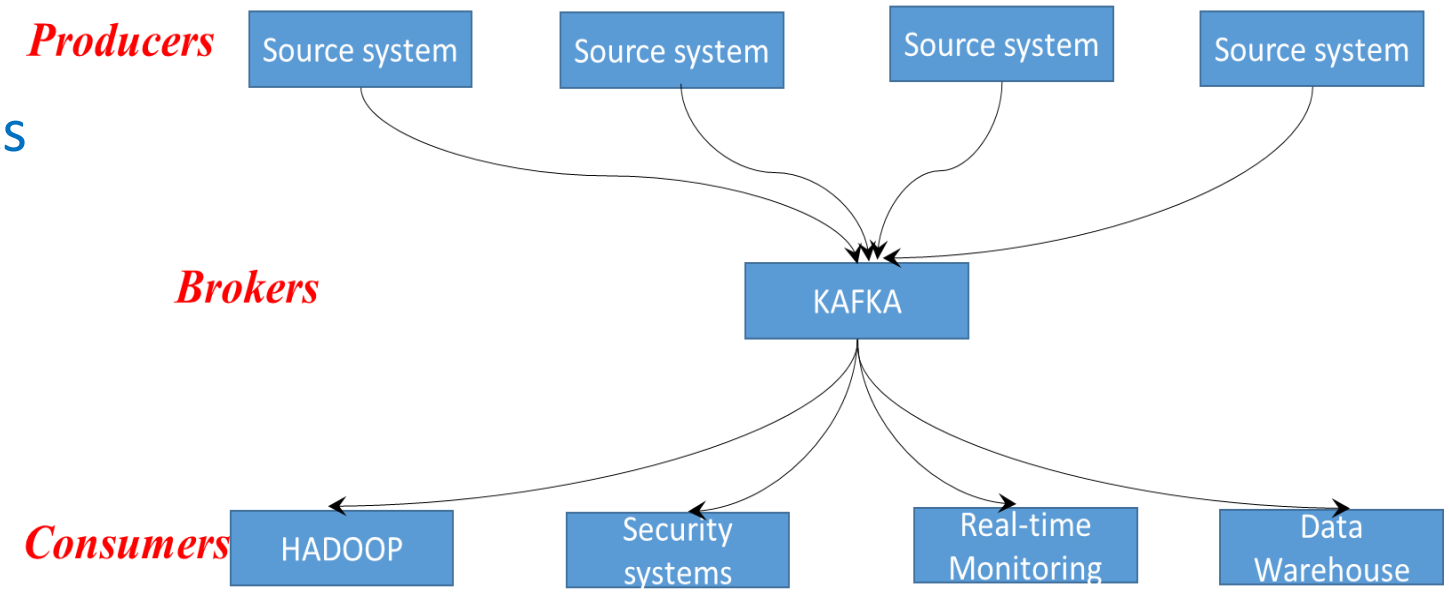


Kafka Decouples Data Pipelines

BIG DATA

Role of a Consumer

- Tells the communication infrastructure
  - What type of messages it wants to receive
- Does not tell "who" to receive message from.
- Messages are delivered to the consumer by the communication infrastructure.
- Called the *subscriber*



Kafka Decouples Data Pipelines

Kafka Communication and Routing

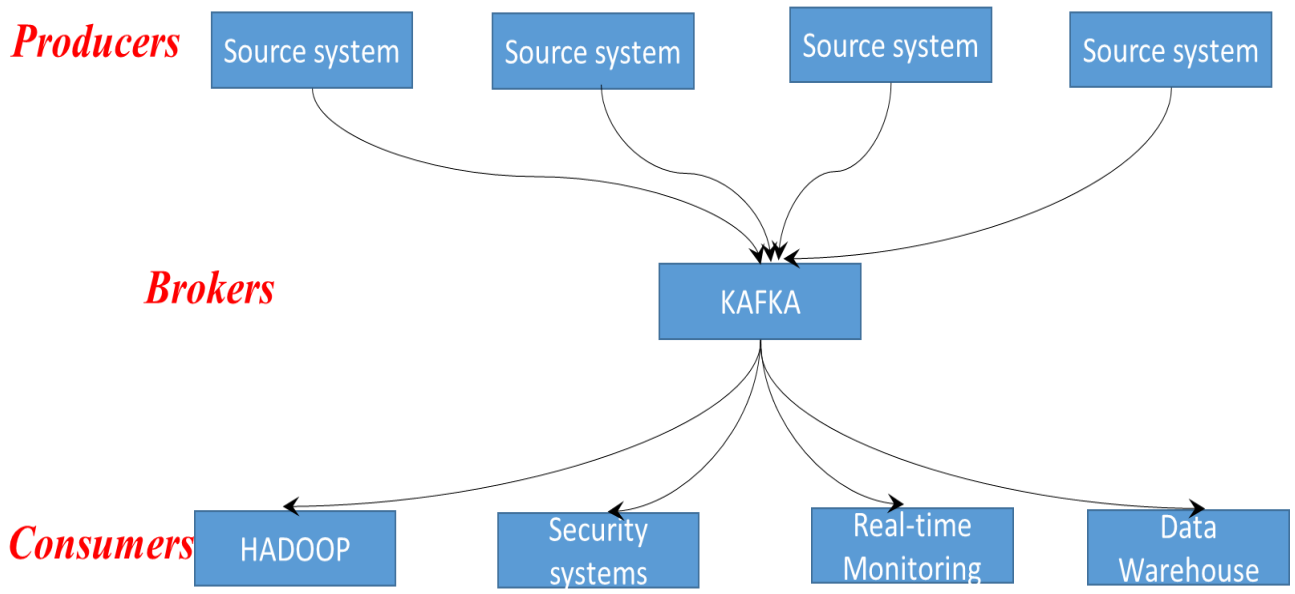
Which Messages get Delivered to Each Subscriber?

- Topic based system
- Content-based system

BIG DATA

The Communication infrastructure

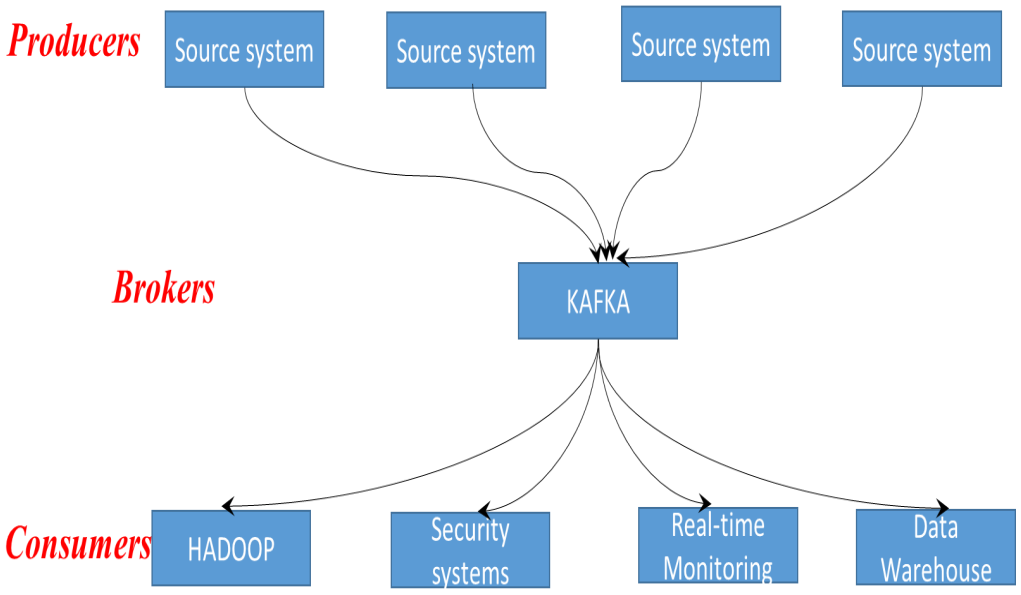
- The most complex part
- Which messages are delivered to each subscriber
- Two models of Routing
  - Topic based systems
  - Content based systems



Kafka Decouples Data Pipelines

Topic based routing

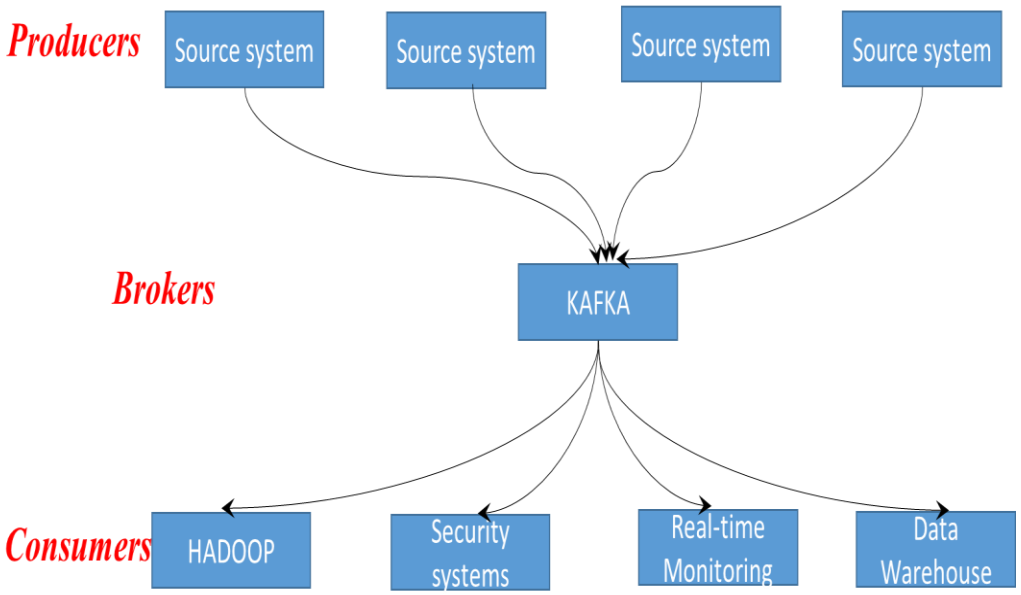
- Publishers send messages with topic labels
- Subscribers subscribe to topics
  - And will receive all messages on that topic
- Example
  - Subscribe to all fire sensors in b block



Kafka Decouples Data Pipelines

Content based routing

- Subscribers define matching criteria
- And will receive all messages that match the criteria
- Example
  - Subscribe to advertisements that feature ViratKohli
  - This is not supported by Kafka
- Pattern based
  - Supports a simpler version called *pattern* based where we can give a wildcard expression for a topic
  - Get all topics that have ipl*



Kafka Decouples Data Pipelines

BIG DATA

Communication advantages/disadvantages

Pros

- No hard-wired connections between publishers and subscribers
- Flexible: Easy to add and remove publishers or subscribers

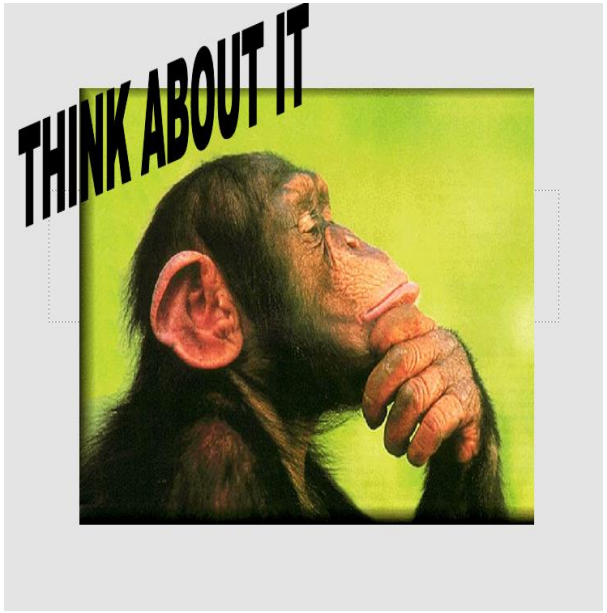
Cons

- Design and maintenance of topics
- Performance overhead due to communication infrastructure

BIG DATA

Exercise 2 (5 minutes)

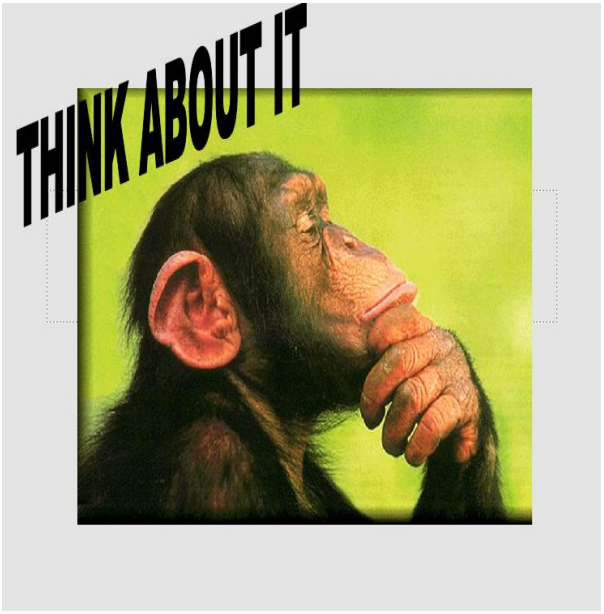
- Consider a bookstore portal with various activities such as
  - Login
  - List books
  - Get book details
  - Buy book
  - Check status of order
  - Return book
  - Logout
- Assume we have 3 backend modules
  - Security
  - Order processing
  - Book information
- **Would you use a topic-based or content-based system? What would be the topics / content..?**



BIG DATA

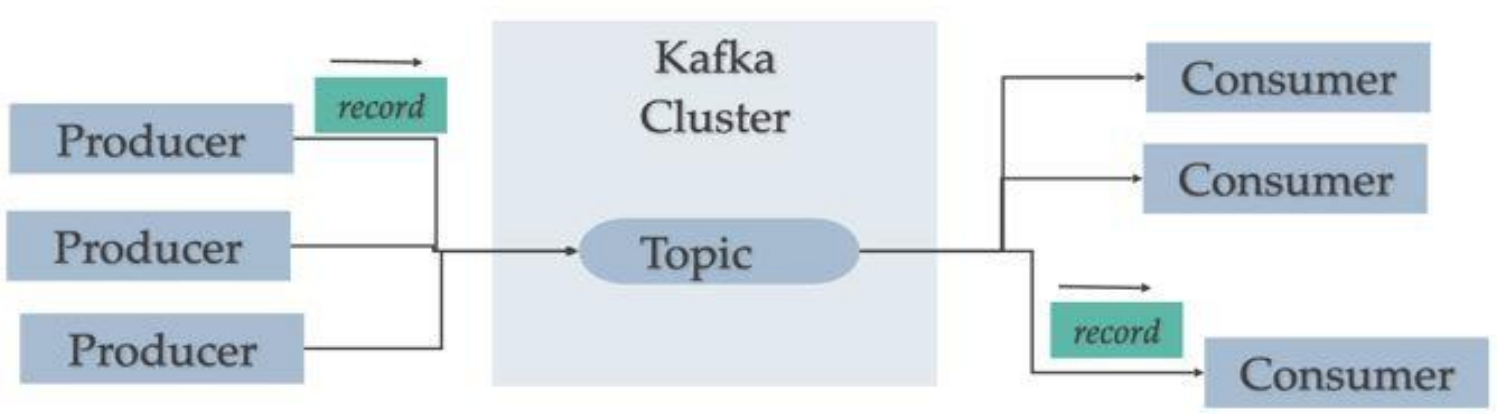
Exercise 2 (Solution)

- Would you use a topic-based or content-based system? What would be the topics / content..?
  - **Probably topic-based, since each message type can be a topic**



Kafka Messages

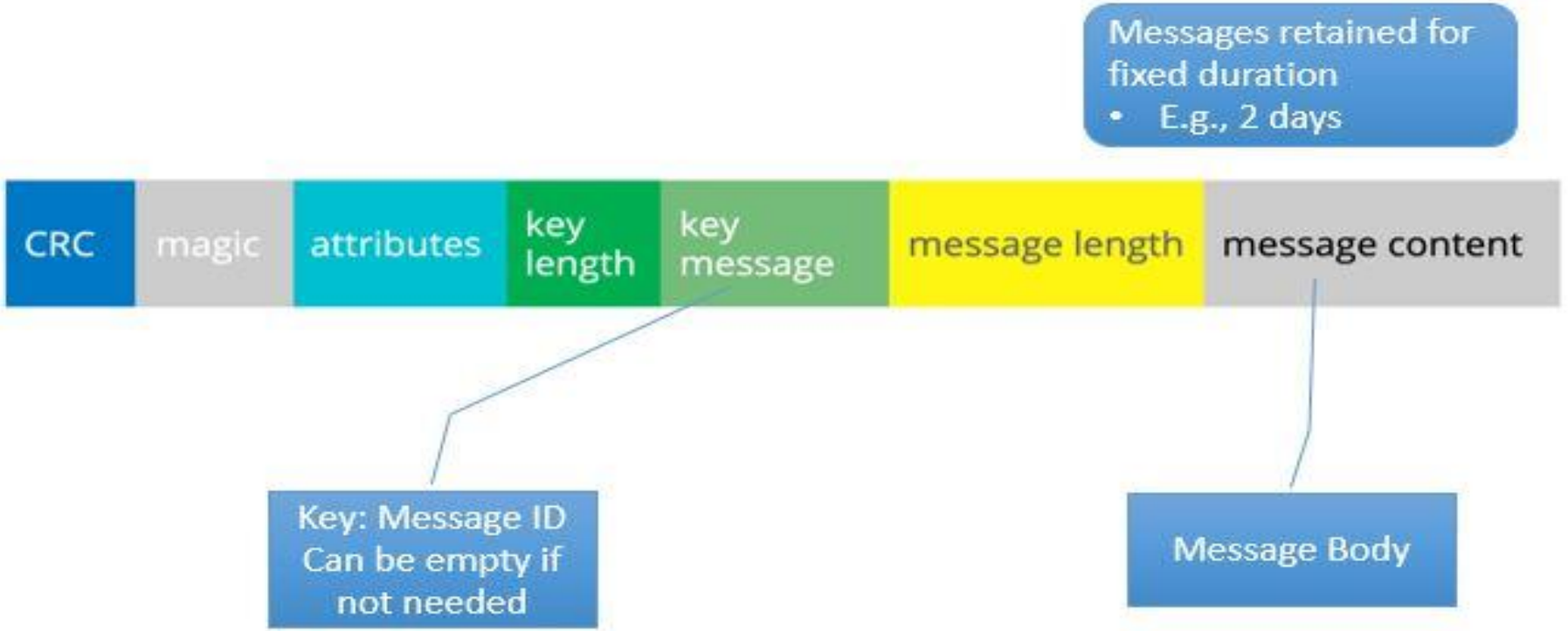
Topics, Producers and Consumers.



Topic name: “/shopping-cart-done“, “/user-signups”

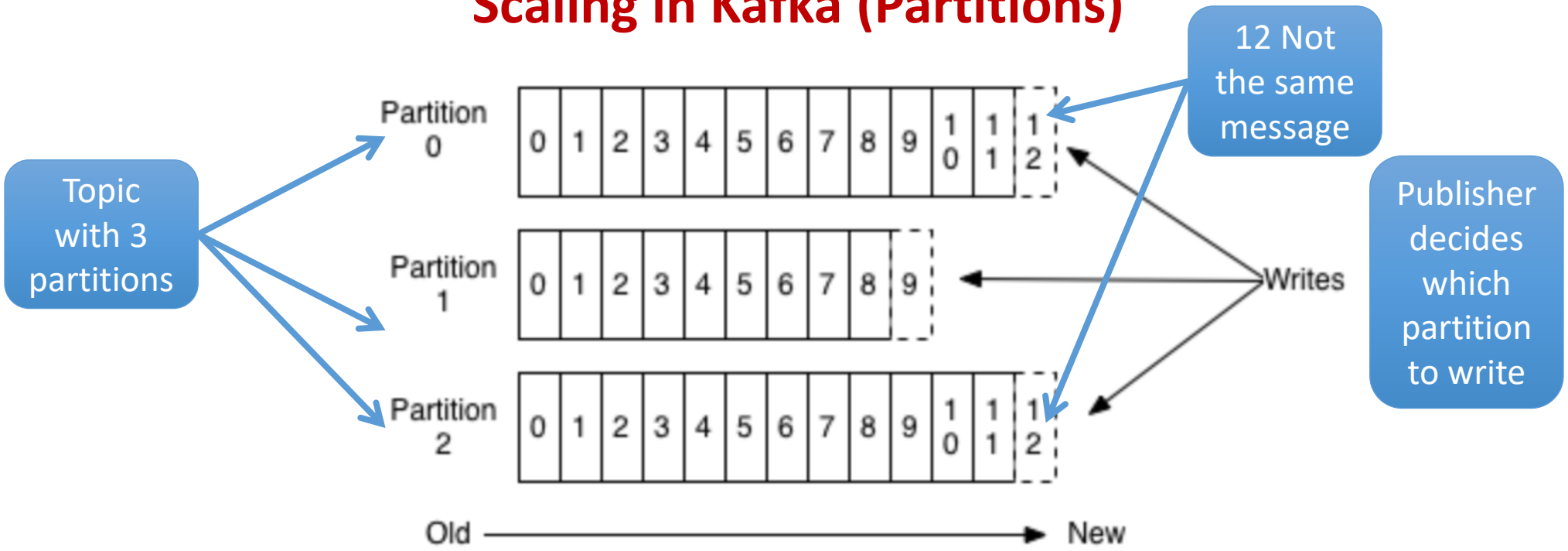
Topic: log data structure on disk

A Kafka Message



Kafka Scaling

Scaling in Kafka (Partitions)

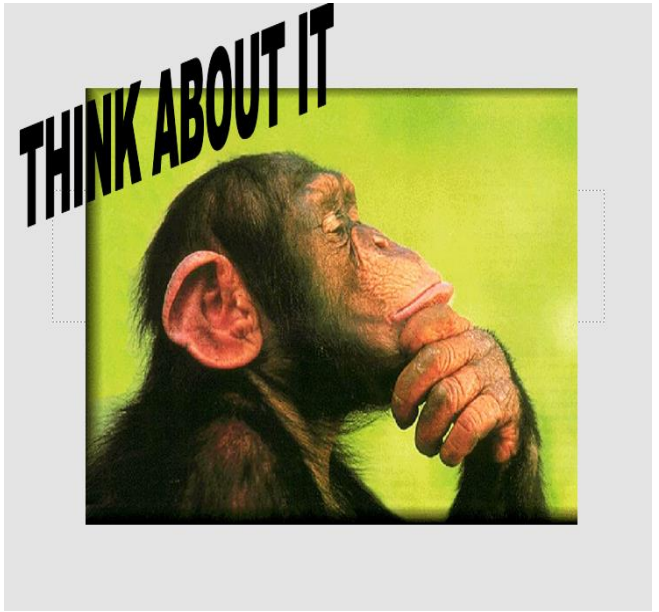


- Partitions allow
  - Log greater than disk size
  - Throughput > single server
- Distributed over servers
- Publisher can load balance
  - Round-robin
  - Based on key
- Messages have an offset

BIG DATA

Exercise 3

- How can reliability be guaranteed in Kafka?
  - Hint: How does HDFS guarantee reliability?



Fault Tolerance in Kafka

BIG DATA

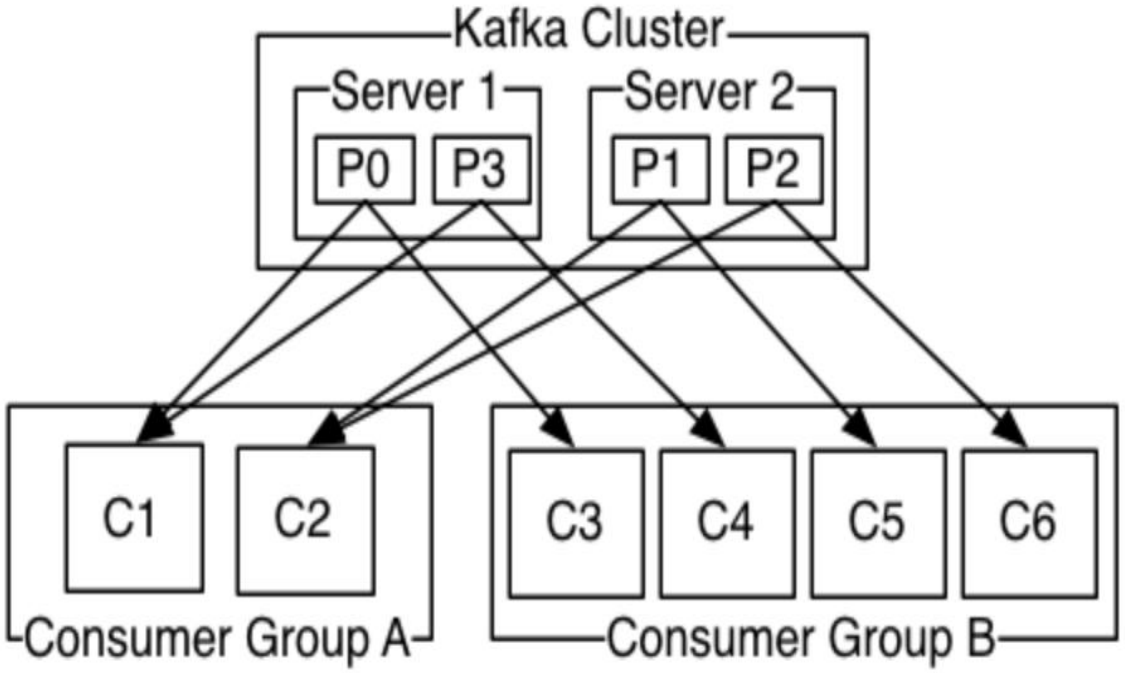
Fault-tolerance in Kafka (Replication)

- Partitions can be replicated
  - Leader: all reads, writes
  - Follower replicates
  - New leader if leader fails
- Durability levels
  - Sync: after quorum writes
  - Async
    - 0 = leader only
    - -1 = no write
- Recommended
  - Replicas = 3
  - Quorum = 2

BIG DATA

Message delivery to consumers

Topic with 4 partitions

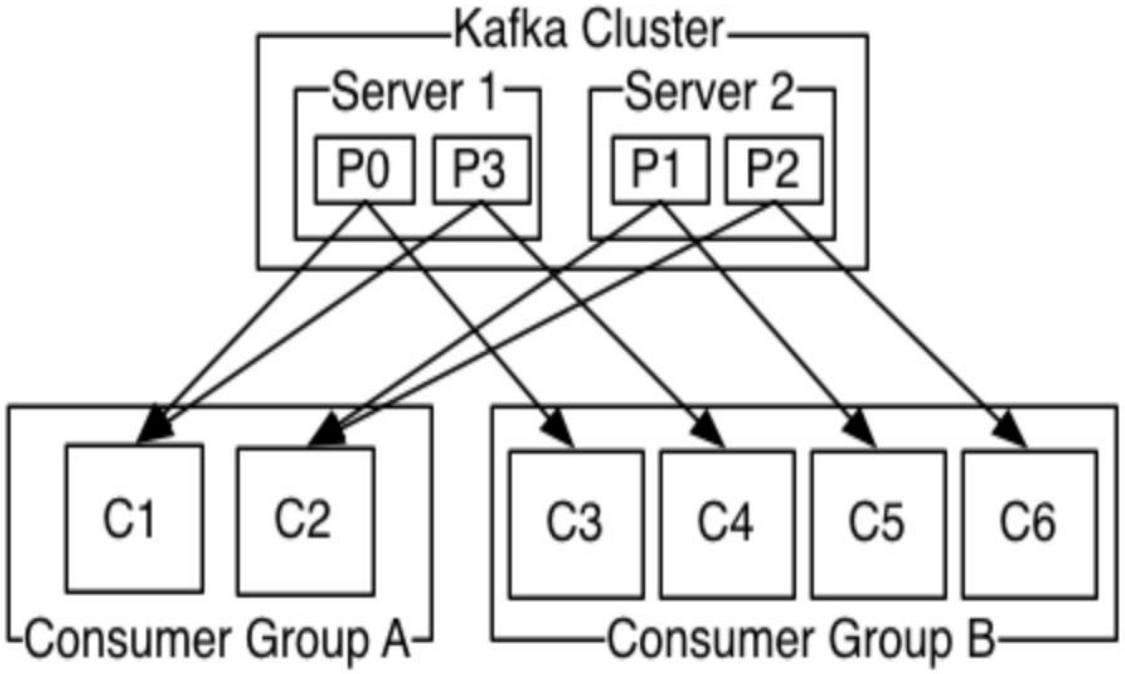


- Consumer group
  - Typically multiple instances of an application
- Partition delivers message to ONE of the group members
  - Load balancing

BIG DATA

Exercise 4

Topic with 4 partitions

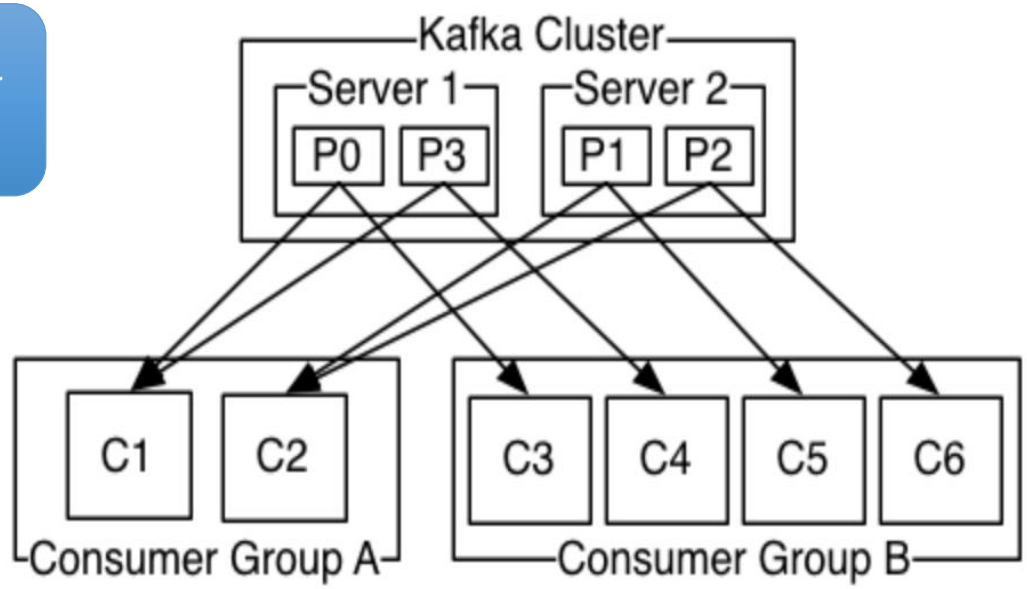


- In the above configuration, how is the load balanced over all the instances?

BIG DATA

Exercise 4 (Solution)

Topic with 4 partitions



- Group A
  - C1 is assigned P0, P3
  - C2 is assigned P1, P2

- Group B
  - C3 is assigned P0
  - C4 is assigned P3
  - …

- Each instance has the same number of partitions

BIG DATA

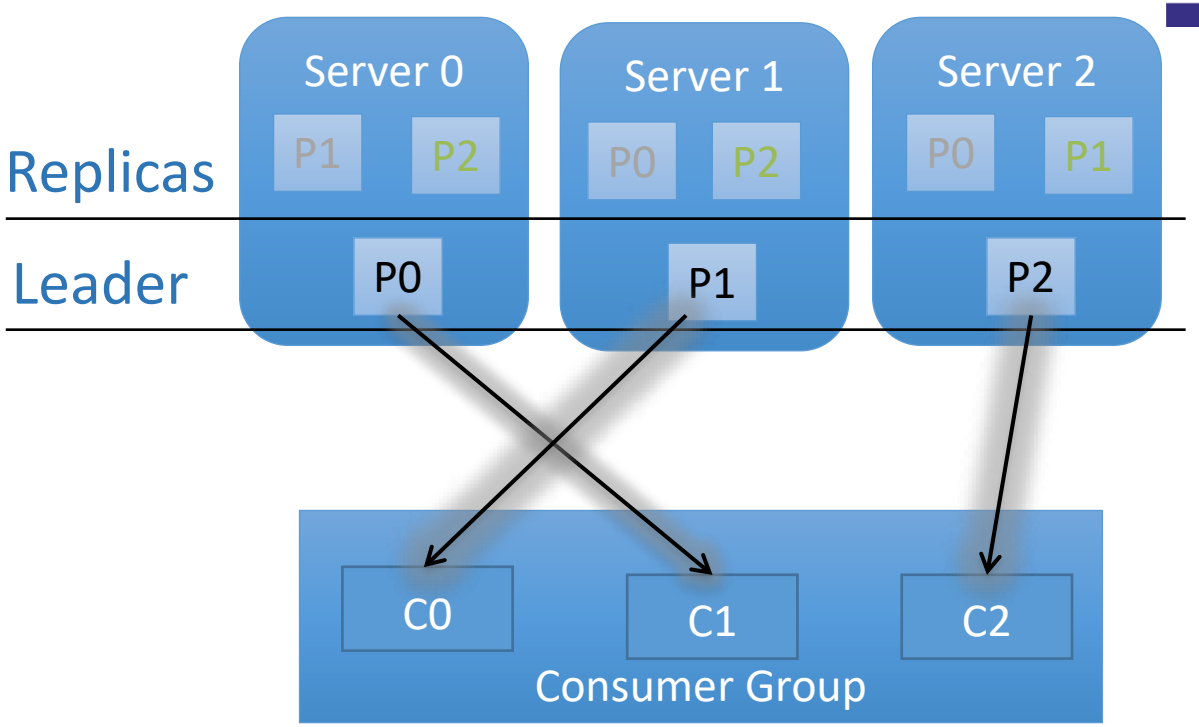
Exercise 5

- Suppose we have a Kafka system
  - 1 topic
  - 3 servers
  - 3 partitions
  - 3 replicas per partition
  - Consumer group with 3 instances
- Draw a diagram showing
  - Servers
  - Partitions
  - Consumer instances
  - Partition assignments

BIG DATA

Exercise 5 (Solution)

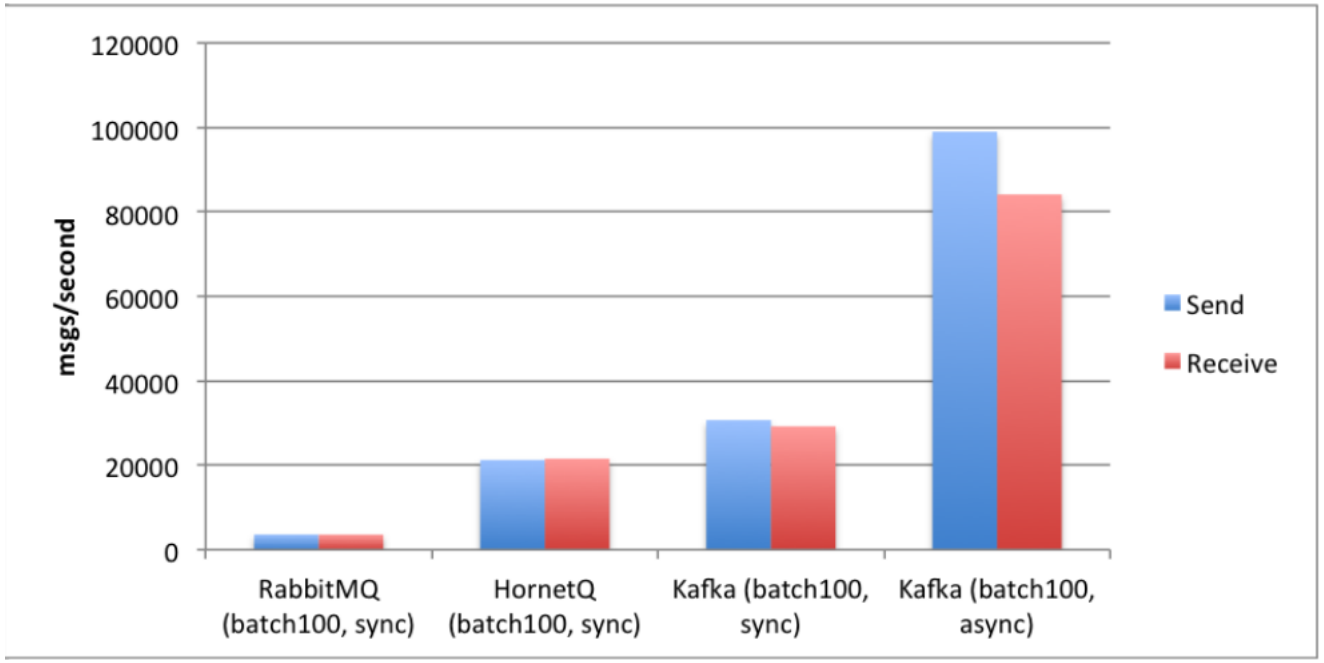
- Suppose we have a Kafka system
  - 1 topic
  - 3 servers
  - 3 partitions
  - 3 replicas per partition
  - Consumer group with 3 instances
- Draw a diagram showing
  - Servers
  - Partitions
  - Consumer instances
  - Partition assignments



Kafka Performance

BIG DATA

Kafka Performance



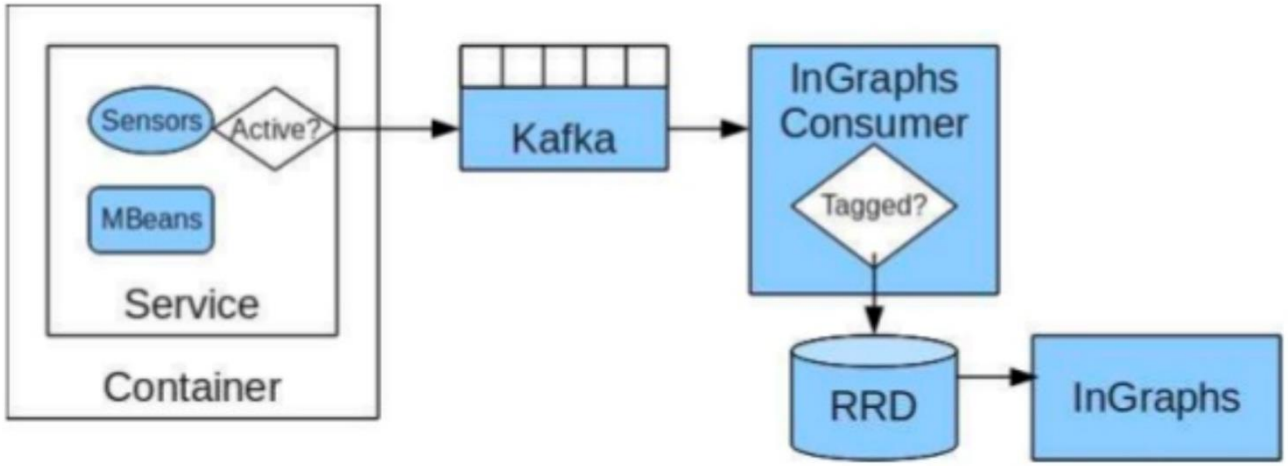
- I/O
  - Sequential reads by consumers
  - Sequential writes by producers
- Zero Copy I/O
  - DMA
  - No copy from kernel to user
- Write batching

PES UNIVERSITY ONLINE

Who uses Kafka ?

- ***LinkedIn***: Activity data and Operational metrics.
- ***Twitter***: Uses it as part of Storm – stream Processing infrastructure.
- ***Square***: Kafka as bus to move all system events to various Square data centers(logs, custom events, metrics, and so on). Outputs to Splunk, Gtaphite,Esper-like alerting systems.
- ***Spotify, Uber, Tumbler, Goldman Sachs, PayPal, Box, Cisco, Cloud Fatr, DataDog, LucidWorks, MailChimp, Netflix, etc.***

Kafka at LinkedIn



- Multiple data centers
- System monitoring
  - 320,000,000 metrics/minute
  - 530 TB of disk space
  - > 210,000 metrics / service
- Analysis – Hadoop, …

BIG DATA

Additional Reading

- Introduction to Kafka
  - https://kafka.apache.org/intro
  - Watch the video on the introduction.
- Kafka in a nutshell
  - https://sookocheff.com/post/kafka/kafka-in-a-nutshell

THANK YOU

K V Subramaniam, Usha Devi

Dept. of Computer Science and Engineering

subramaniamkv@pes.edu

ushadevibg@pes.edu

BIG DATA

Streaming Algorithms

K V Subramaniam
Computer Science and Engineering

BIG DATA

Overview: Streaming Algorithms

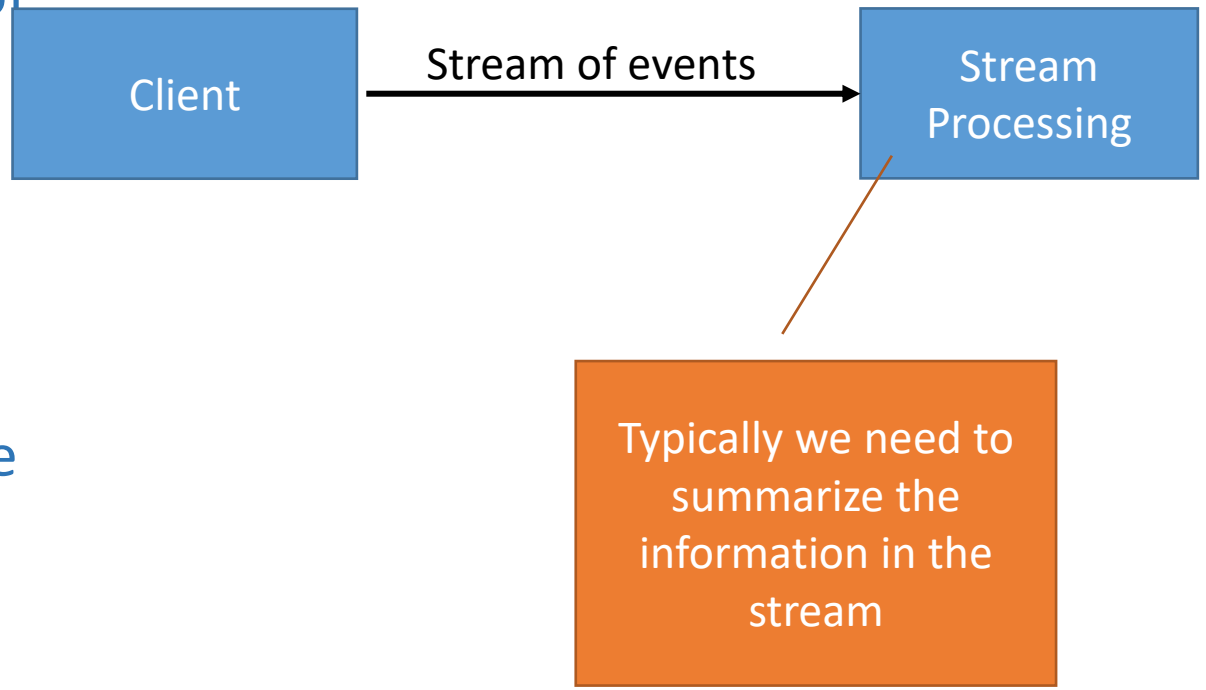
- Why study streaming algorithms?
- Sampling Algorithms

Streaming Algorithms overview

BIG DATA

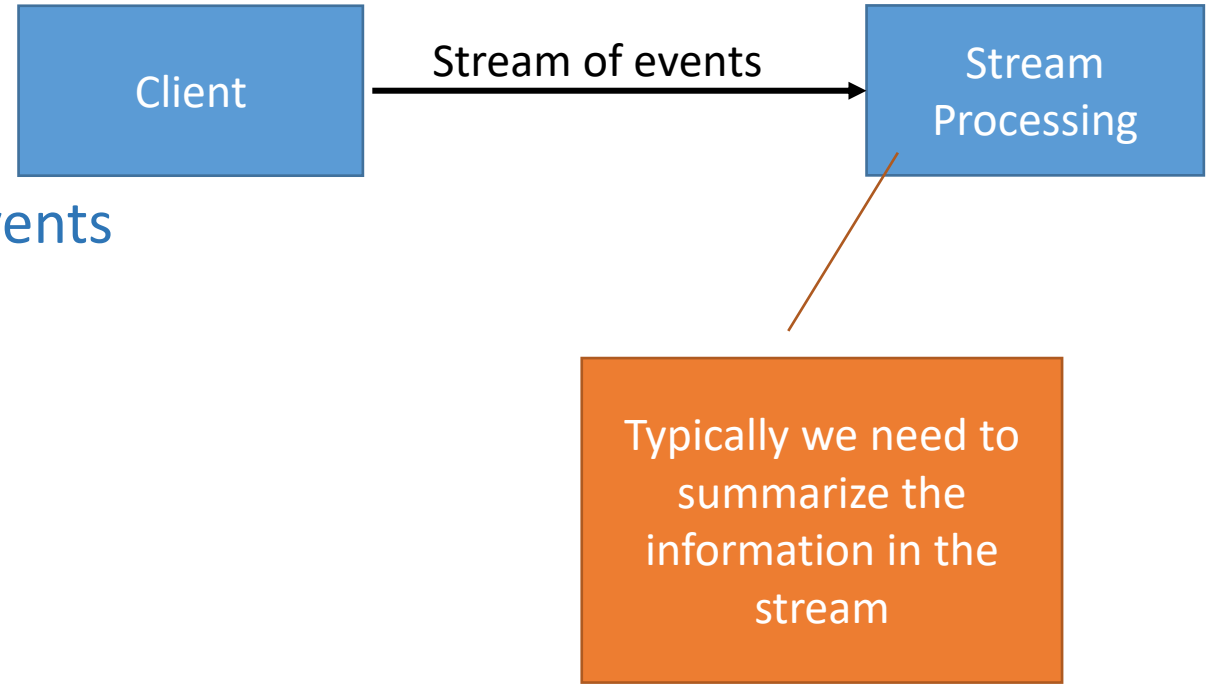
The need for processing events

- Stream processing requires processing of events in a never ending stream
- Need to look at a summaries
- For example
  - How many unique elements have we seen in the stream?
- In a relational data this is equivalent to counting the keys, but how to do it on a never ending stream



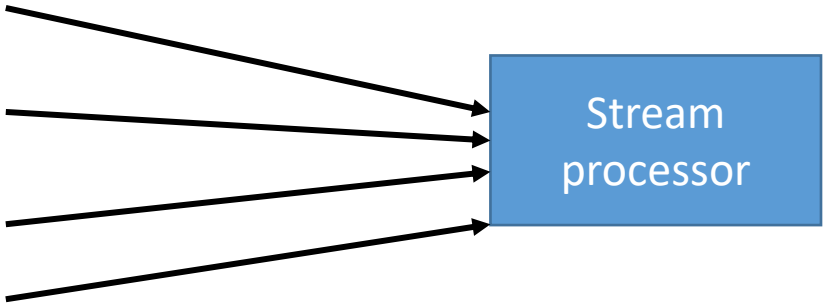
The need for processing events

- One approach
  - Breakup stream into a window of events
  - Window size *n*
  - Allows us to perform relational operations
  - Similar to Apache Spark model



BIG DATA

Issues in stream processing

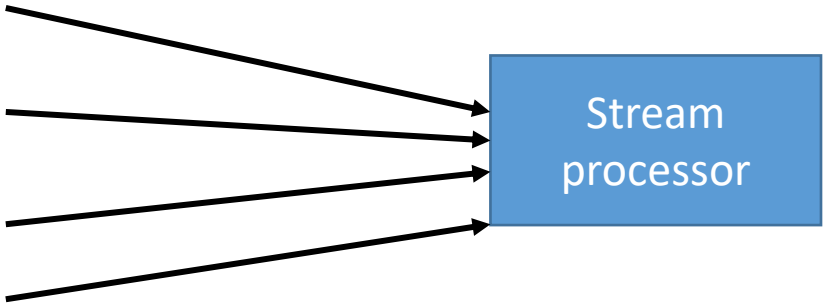


- Velocity of the stream
  - The rate at which data is being sent
  - Sometimes requires instantaneous decision.
  - Different streams may have different rates
  - For ex: mission critical systems

- #streams
  - Processing multiple streams each requiring a small amount of memory may stress memory system

BIG DATA

Stream processing algorithms



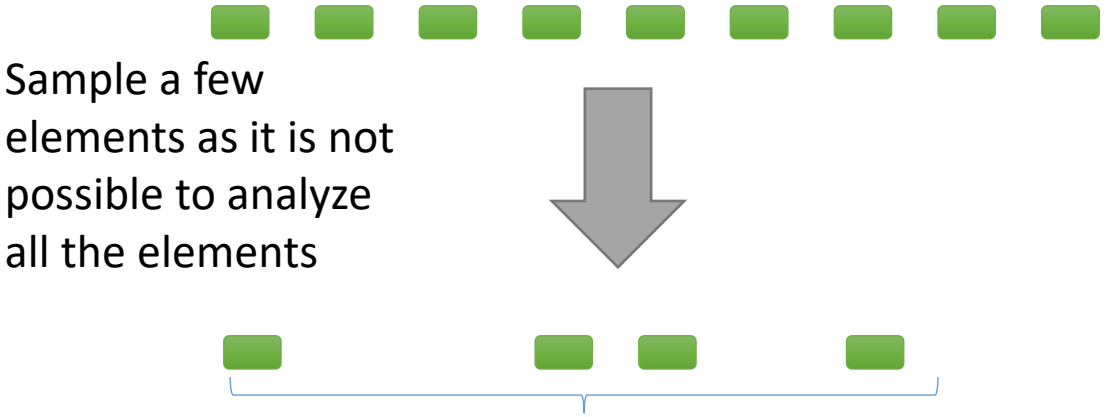
- Must process data in memory
  - Not off disk (too slow)
- More efficient to get *approximate* solution rather than an exact one.
- Often use *hashing* techniques to introduce randomness

Sampling Algorithms

BIG DATA

Extracting reliable samples

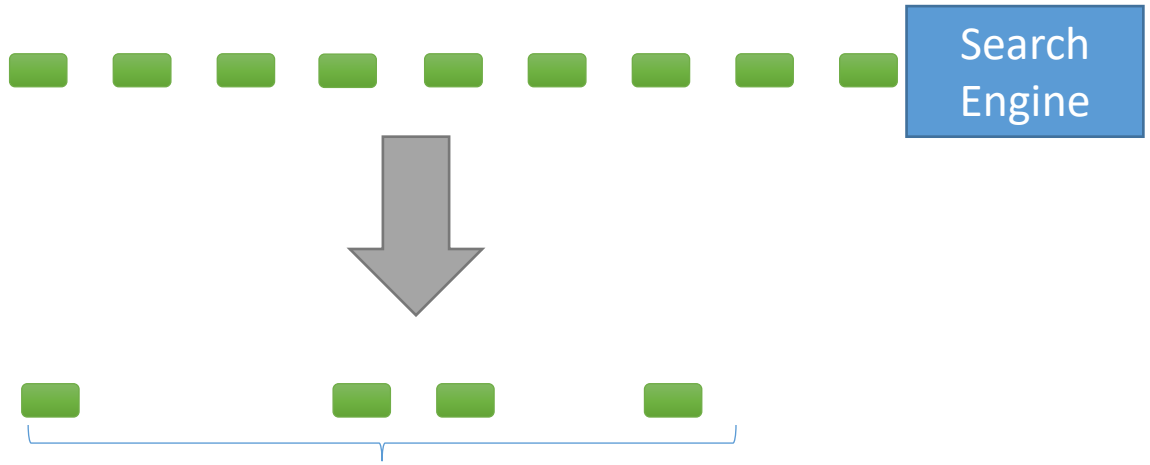
- Given a long stream of data
- How to create a representative sample?



BIG DATA

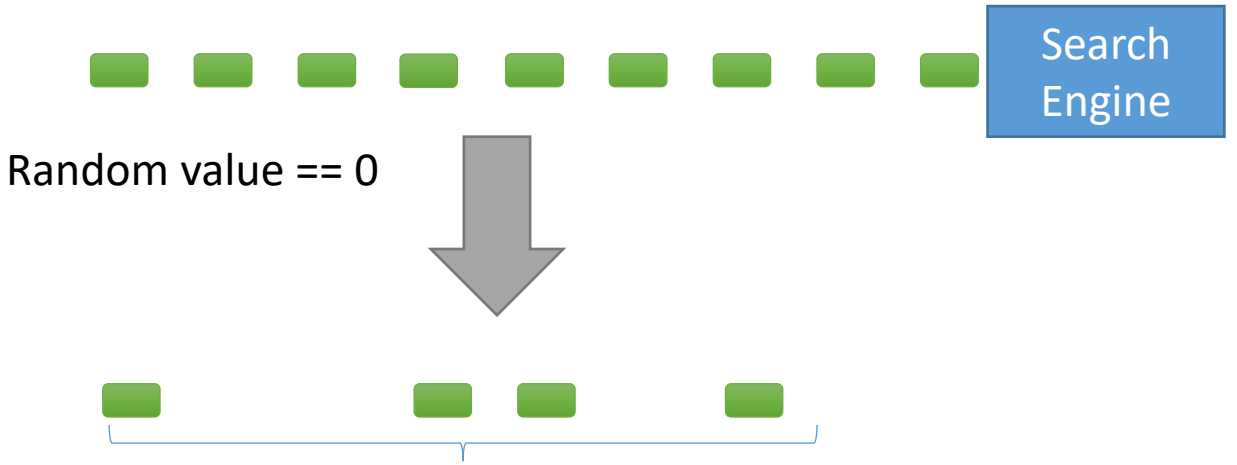
Motivating Example

- Each element of the query represents a query
- How to create a representative sample?
- We want to answer the question?
  - "What fraction of the typical user's queries were repeated over the past month?"
- What's obvious algorithm?



Obvious Algorithm

- For every stream tuple seen..
- Generate a random integer between 0..9
- If value is 0
  - Then store(use) the tuple
- Otherwise – discard.



Flaw in obvious algorithm

Flaw in obvious algorithm

- Suppose user has issued a particular search query s twice
- There's a probability 1/100 (not 1/10) that both queries will show up in our sample
- There's only a probability 1/100 that we will know query s was repeated

So our sampling will be wrong

Query number *m* is *s*

p(m sampled) = 1/10

Query number *n* is also *s*

p(n sampled) = 1/10

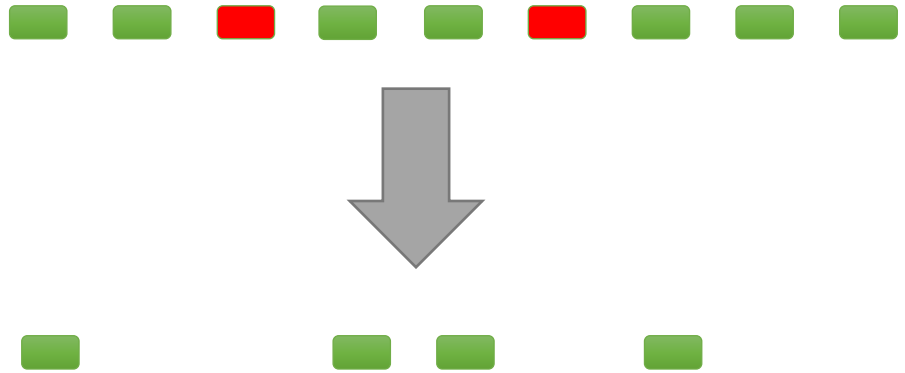
p(m and n sampled) = 1/100

Flaw in obvious algorithm

Flaw in obvious algorithm

- Suppose user has issued a particular search query s twice
- There's a probability 1/100 (not 1/10) that both queries will show up in our sample
- There's only a probability 1/100 that we will know query s was repeated

So our sampling will be wrong



Sampling Algorithms - refinement

Refined algorithm

- Sample 1/10th of the users
  - Not the transactions
- Details
  - When a query arrives
  - Look up user to see if they are in sample.
  - If so, add query to sample
  - If first time we have seen the user, generate a random number 0..9
    - Add user to sample if number is 0

Optimization to algorithm

- Checking if we have seen the user
- Requires a search through a data structure
- Not really required
- Just hash(username) → 0..9
  - If hash is 0, select the userid

Generalization of the algorithm

- We used “user” here to select
- How to extend this algorithm generically?
- Identify the *key components* of the query.
  - In previous example tuple is *<user, query, time>*
  - But *key component* is only user.
- Hash key components in the range *(0..b)*
- To get sample size (a/b)
  - Select query if *hash(key components) < a*

Exercise

- Suppose we want a sample dataset to debug a program that profiles transactions by user and country
- How would I generate a 1/20 sample?

Exercise - Solution

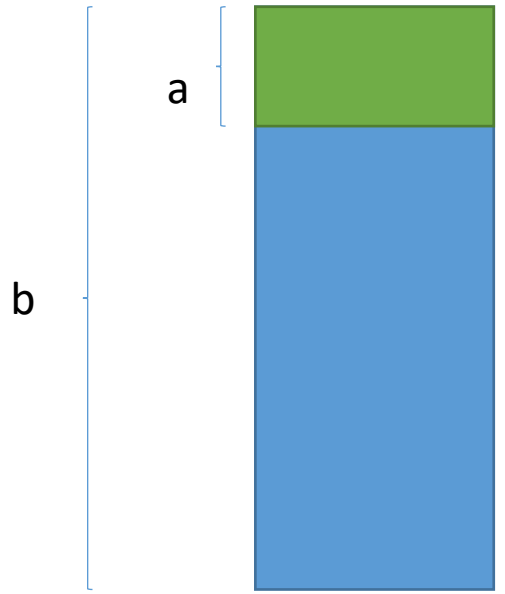
- There isn't a unique solution
- Suppose I want a sample dataset to debug a program that profiles transactions by user and country
- How would I generate a 1/20 sample? Hash userid and country in the range 0..19; select if the hash is 0
- The above method will give me 1/20 of all user-country pairs

Varying Sample size

BIG DATA

Varying the sample size

- We are selecting *a/b* samples from data arriving from outside.
- As more data is added to the system, the number of keys in the system also increases
  - Additional storage space required
- If there is a limit on #keys that can be stored, then how to handle it, so representatives is not lost
- Reduce *a* to *a-1.*



THANK YOU

K V Subramaniam, Usha Devi

Dept. of Computer Science and Engineering

subramaniamkv@pes.edu

ushadevibg@pes.edu

BIG DATA

Streaming Algorithms - 2

K V Subramaniam
Computer Science and Engineering

BIG DATA

Overview: Streaming Algorithms

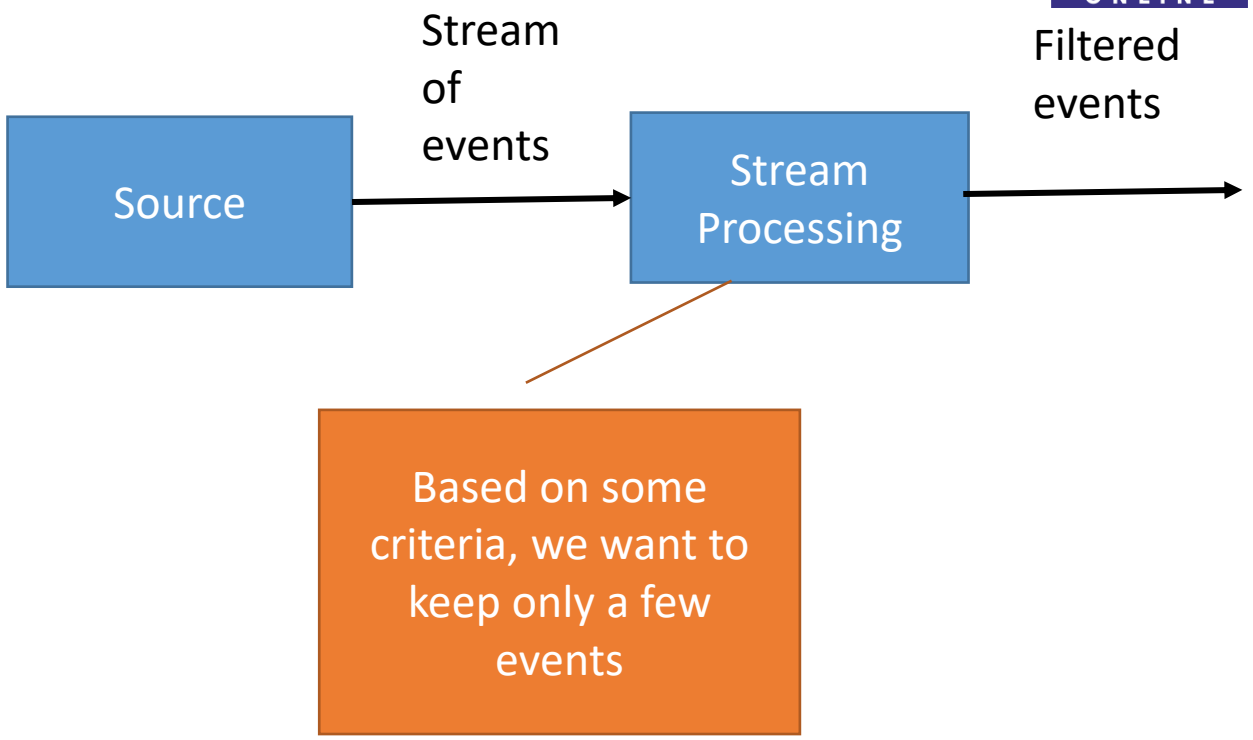
- Filtering algorithms – Bloom Filter
  - Motivation
  - General bloom filters
  - Extensions
- Counting unique elements
  - Motivation
  - Flajolet Martin Algorithm and working
  - Practical considerations

Bloom Filters

BIG DATA

Filtering Data

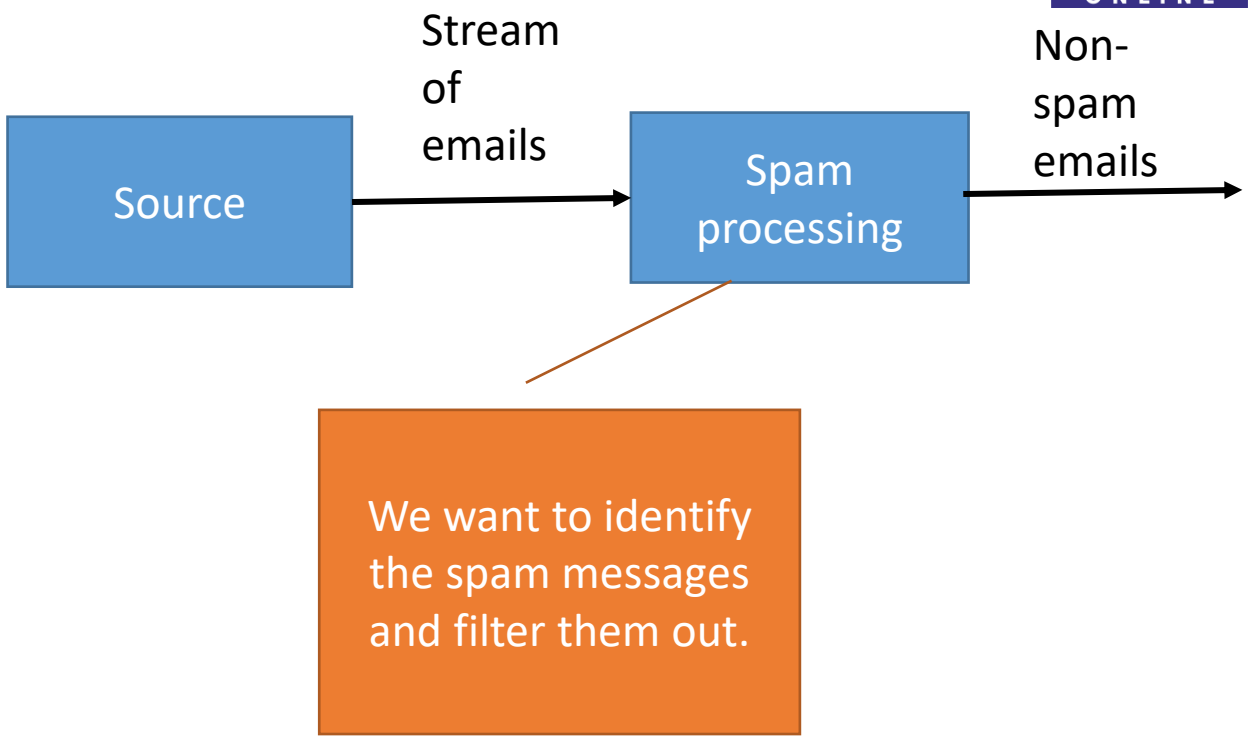
- Sometimes we need to take a decision
  - To filter out certain events
  - The decision has to be taken instanteously
  - Large number of events need to be processed.



BIG DATA

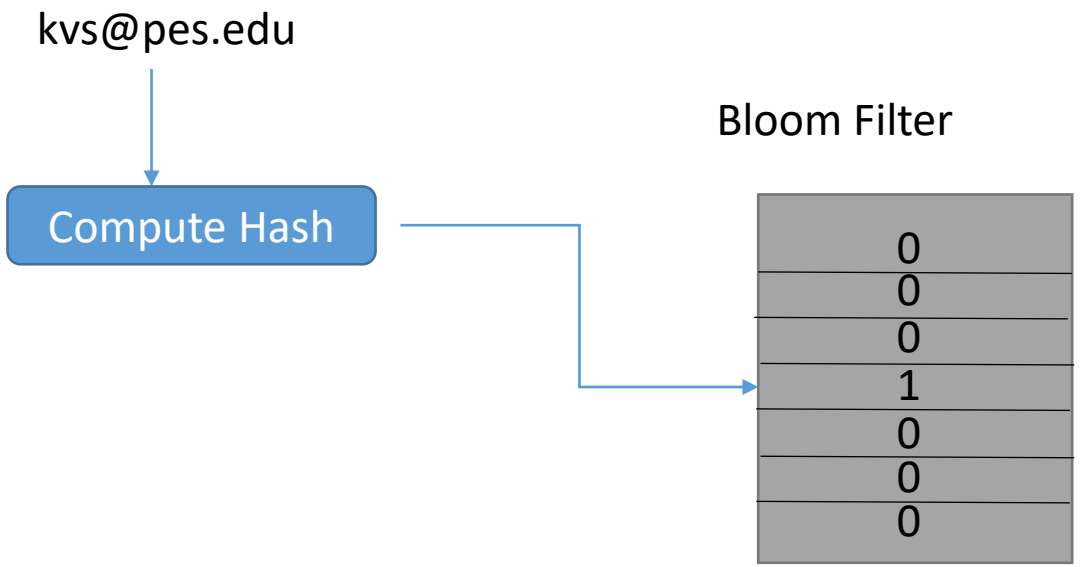
Filtering Data – Motivational example

- Incoming email
  - Remove all spam emails
- Constraints
  - 1GB of main memory
  - 1 billion non-spam email ids (well known)
  - 20 bytes/email address
- Can’t store all email ids in memory
  - Disk is a slow store



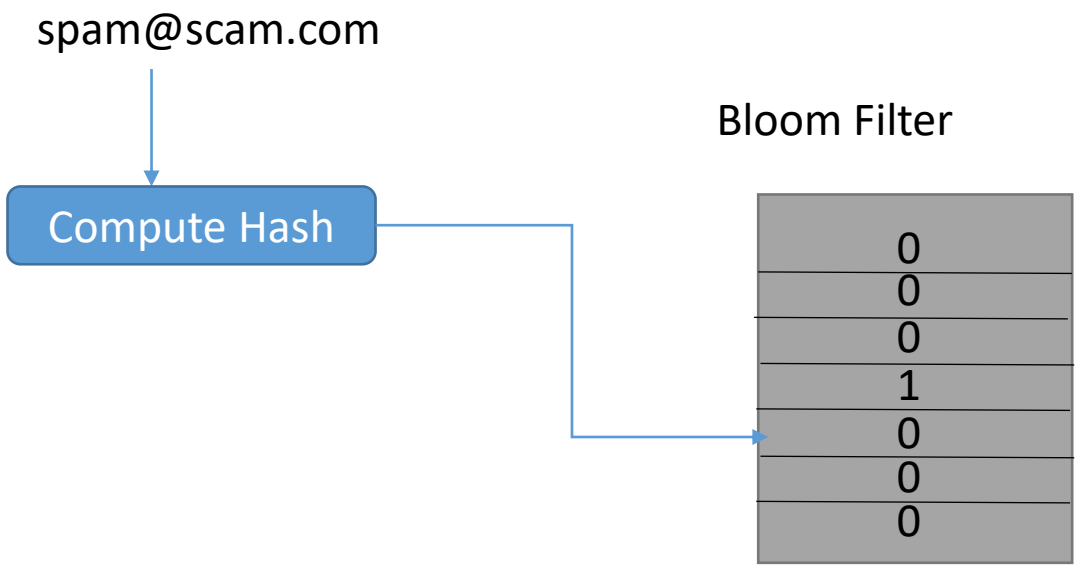
Bloom filter basic: initialization

- 1 GB memory => 8 billion bit string
- Bloom filter initialization
  - Hash non-spam email ids to 0..8 billion-1
  - Set corresponding bit to 1

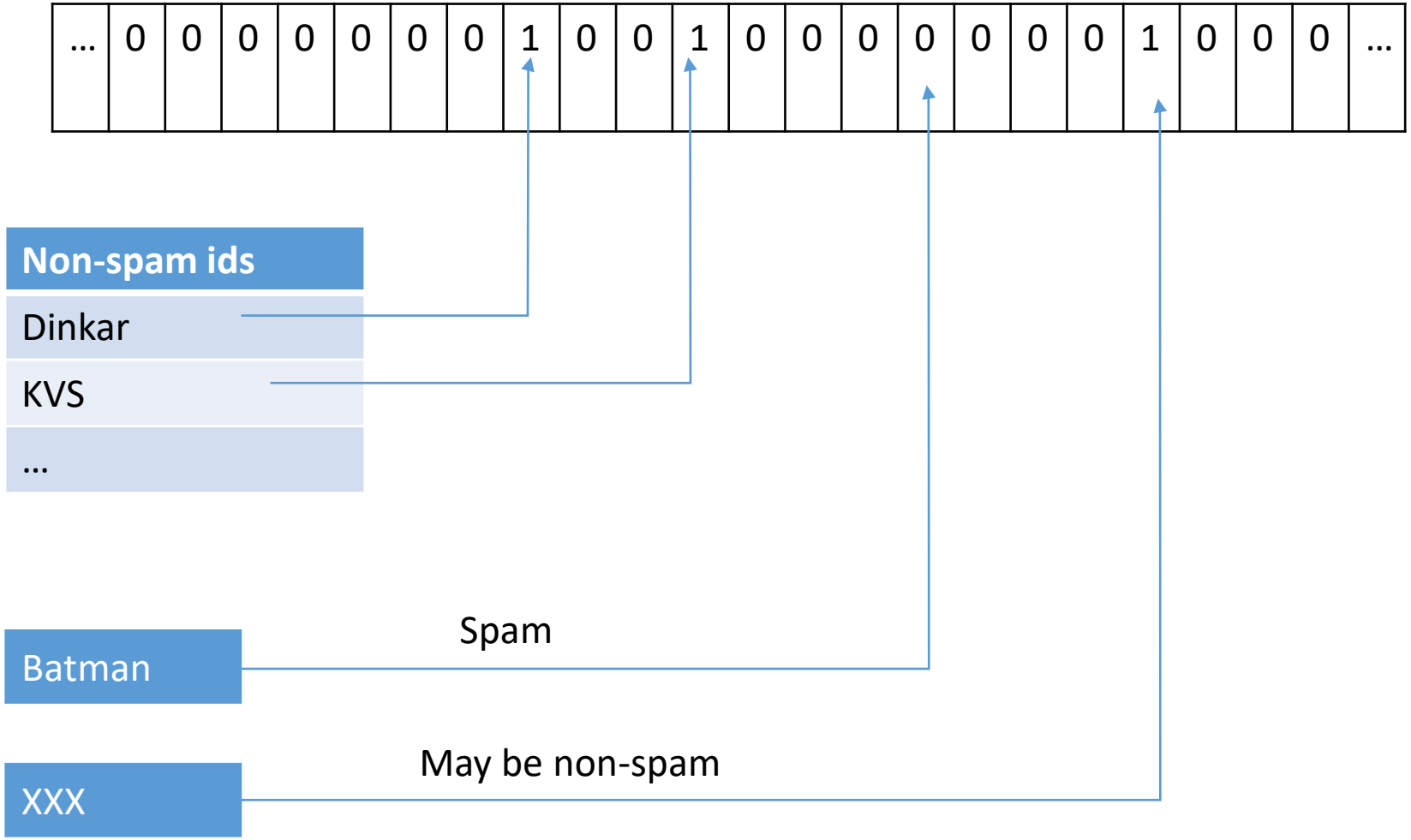


Bloom filter basic: initialization

- Usage
  - Hash incoming email id
  - Check bloom filter entry
    - If (0)
      - Definitely has not been seen before → it is a spam
    - If (1)
      - Not sure if it has been seen before

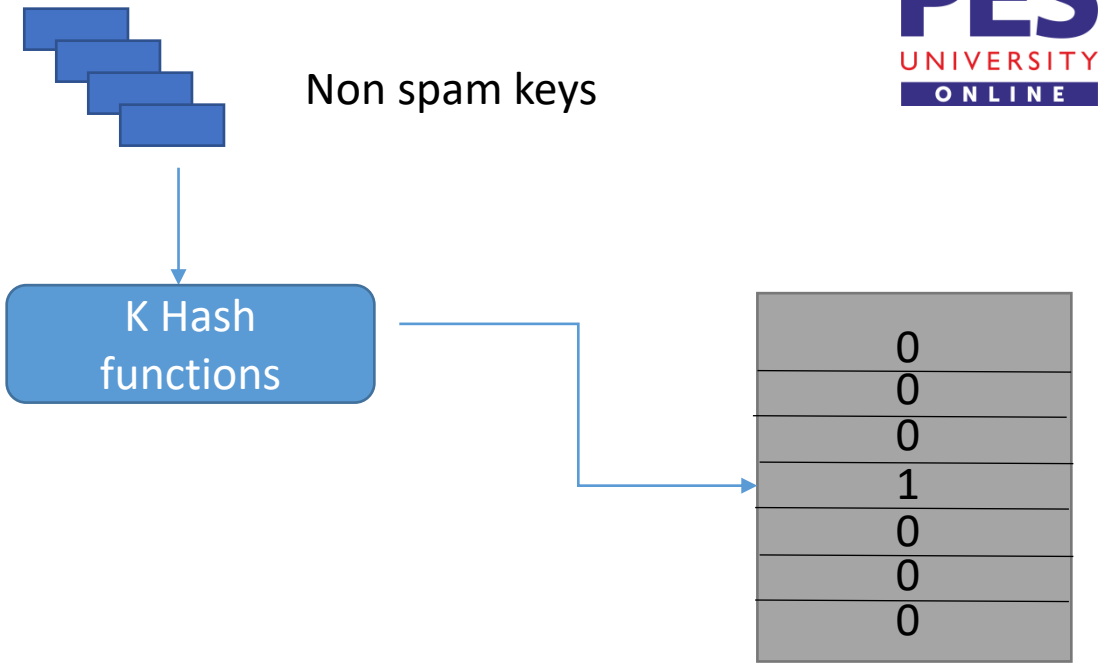


Bloom filter basic: illustration



General Bloom Filters

General Bloom Filters



Bloom filter consists of

- Array of n bits (size of memory in example)
- A collection of k hash functions h1, ..., hk
- A set S of keys with m elements (non-spam email ids in example)
- Purpose: given a key a, determine if it is in S (in example, given email id, determine if non-spam)

- Initialization: for all keys in S,
  - Compute the k hash functions
  - Set corresponding bits to 1

BIG DATA

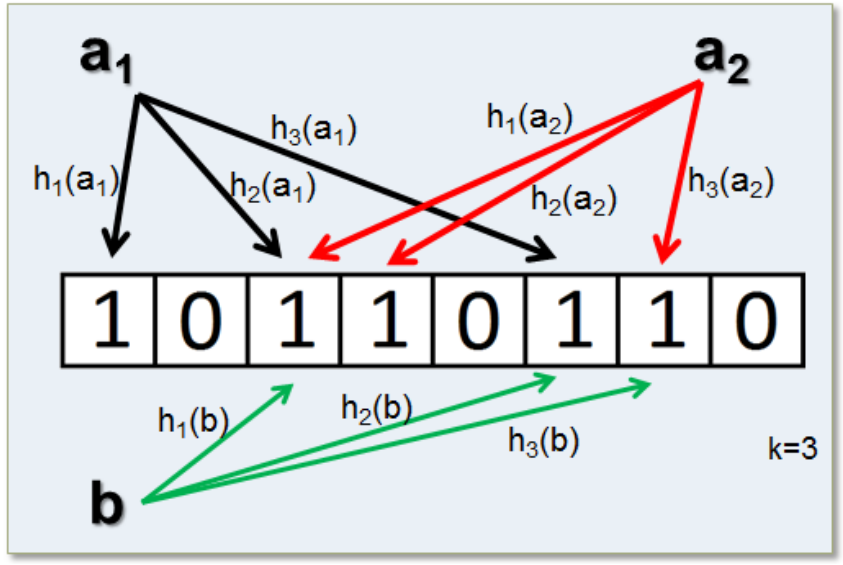
General Bloom Filters

- Usage
  - Hash a using the k hash functions
  - If all the corresponding k bits are 1, a€S

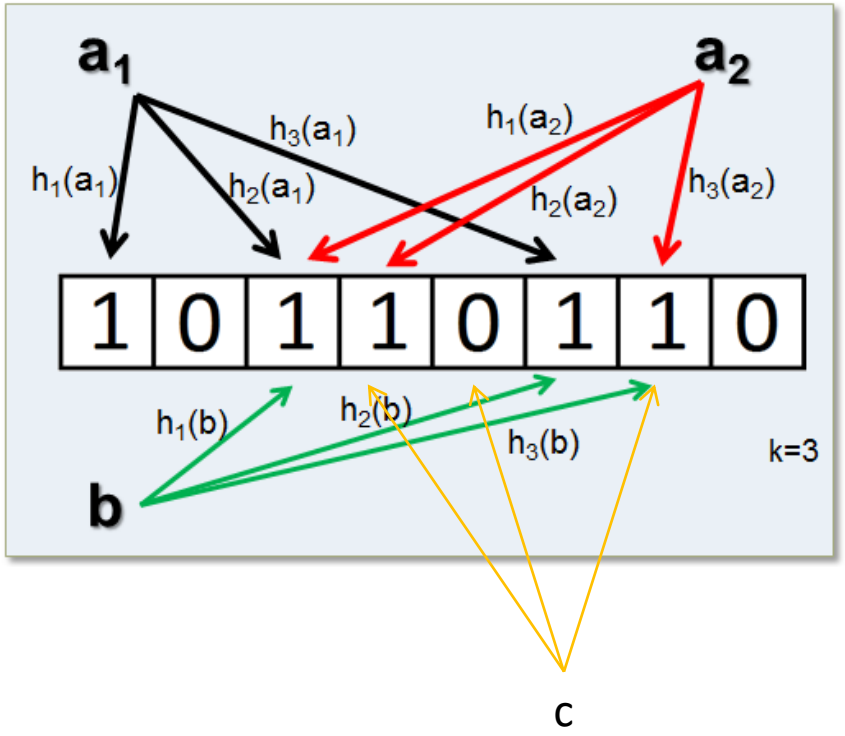
- Probability of false positive $(1 - e^{-km/n})^k$.
  - Read derivation in book

General Bloom Filters

- Top
  - Shows the insertion
- Bottom
  - Shows the check for set membership
  - B is checked to see if it is part of the bloom filter.

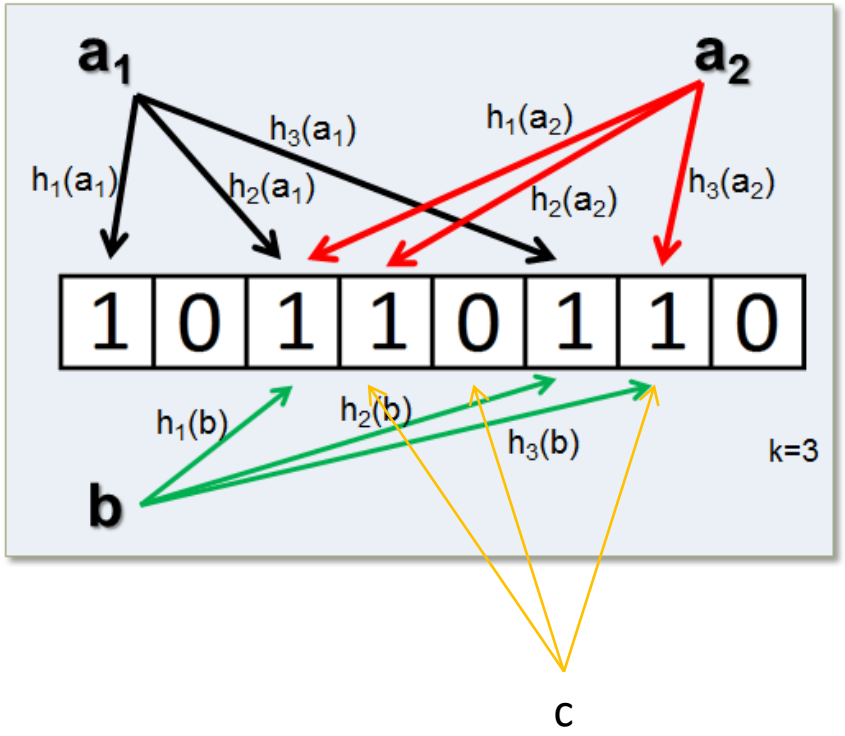


Exercise



- Suppose c hashes as shown
- Is c spam, not spam, or possibly spam?

Exercise - solution

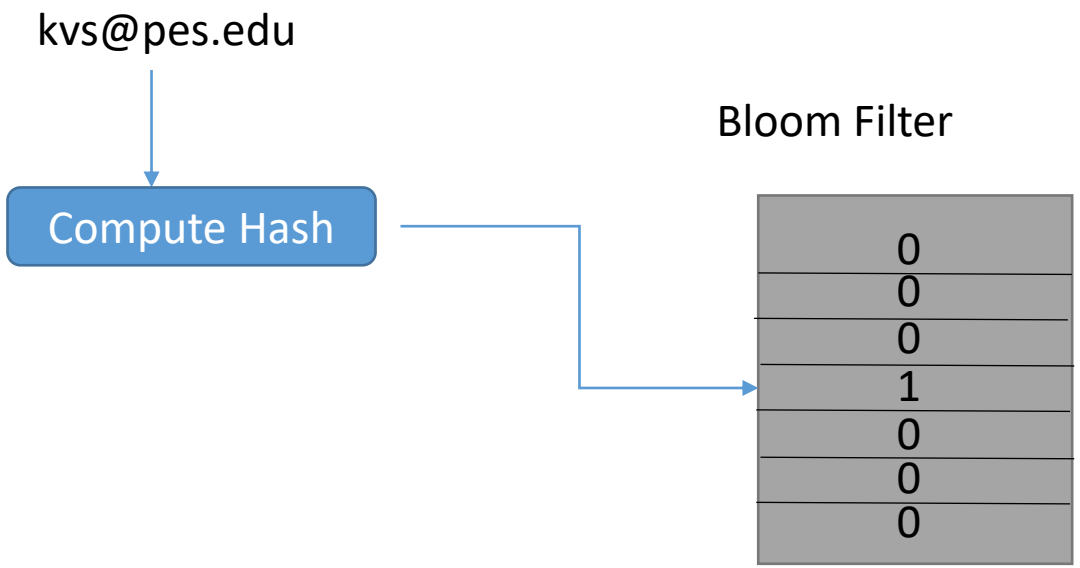


- Suppose c hashes as shown
- Is c spam, not spam, or possibly spam?
- C – is a spam email as it hashes to one bucket that contains a 0

Bloom filter extensions

Bloom filter basic: extensions

- Use secondary storage
  - 7/8 of the time, we can filter from memory
    - 7/8 of the time, a spam id will hash to 0
    - Because, there are 8B bits, 1B are 1, 7B are 0
  - 1/8 of the time, verify non-spam by disk lookup



BIG DATA

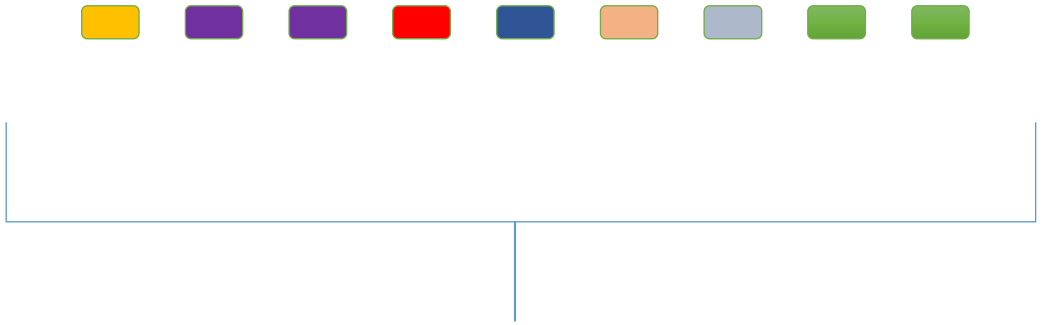
Cascaded Bloom filters

- We can have 2 Bloom filters in series with different hashes
- If bit is 1, use second Bloom filter
- We will reject much more of the spam

Counting Distinct Elements

Motivation

- Website that wants to know how many distinct users visit
  - Amazon: have userid
  - Google: have to use IP address
  - 4 billion IP addresses
- A more complex problem
  - How many different users visit each web page?
  - Users x Web pages combinations



How many distinct users are accessing the website?

Flajolet Martin Algorithm

- Pick hash function that is bigger than set to be hashed

- To count IP addresses: hash > 4 billion
- To count URLs: use 64 bits

Flajolet Martin Algorithm: basic property

- Tail length for hash function: number of 0's at the end of the hash for a given hash function
- Hash each element in stream
- Let R be the maximum tail length
- 2^R is approximately the number of distinct elements seen

11110100 has tail length 2

Exercise

Tail length for hash function: number of 0's at the end of the hash for a given hash function

11110100 has tail length 2

Hash each element in stream

Let R be the maximum tail length

2^R is approximately the number of distinct elements seen

- Suppose we want to count the number of userids that visit a Web page
  - Suppose userid is 0..15
- Mid square hash
  - Cube userid, make 12 bits, take middle 6 bits
  - Hint: powers of 2: 0 1 2 4 8 16 32 64 128 256 512 1024 2048 4096
- Suppose the userid sequence is 10 10 7 10 6 14 14 12 6 5 7

Exercise

Tail length for hash function: number of 0's at the end of the hash for a given hash function

11110100 has tail length 2

Let r be the tail length

What is the probability that r is the tail length?

Flajolet Martin - working

Why does Flajolet Martin Algorithm work

- p ($h(a)$ ends in at least r 0's) = 2^{-r}
  - Suppose the hash is $h_1h_2...h_n$
  - Probability any bit is 0 is ½
  - Probability h_n is o is ½ = 2^{-1}
  - Probability last two bits $h_{n-1}h_n$ are both 0 is (1/2) x (1/2) = 2^{-2}
  - Similarly, probability last r bits are all 0 is 2^{-r}

Exercise

p (*tail length is* r) = 2^{-r}

If there are m elements in the stream, what is the probability that none of them have tail length r?

Generalization of the algorithm

Suppose there are m distinct elements in the stream

p(no element has tail length r) = $(1 - 2^{-r})^m$

Suppose the elements (e.g., userids) are $u_1, u_2, \ldots, u_m$

p (u_1 has tail length r) = 2^{-r}

p (u_1 doesn't have tail length r) = $1-2^{-r}$

Similarly, p (u_2 doesn't have tail length r) = $1-2^{-r}$

p (u_1 and u_2 don't have tail length r) = $(1-2^{-r})(1-2^{-r})$

p ($u_1 \ldots u_m$ all don't have tail length r) = $(1-2^{-r})^m$

Recall

p ($h(a)$ ends in at least r 0's) = 2^{-r}

- Suppose the hash is $h_1 h_2 \ldots h_n$
- Probability any bit is 0 is ½
- Probability h_n is o is ½ = 2^{-1}
- Probability last two bits $h_{n-1} h_n$ are both 0 is (1/2) x (1/2) = 2^{-2}
- Similarly, probability last r bits are all 0 is 2^{-r}

Exercise

p ($h(a)$ ends in at least r 0's) = 2^{-r}

Probability any bit is 0 is 1/2

Suppose there are m distinct elements in the stream

p(no element has tail length r) = $(1 - 2^{-r})^m$

p(no element has tail length r) = $(1 - 2^{-r})^m \sim e^{-mx}$
where $x=2^{-r}$ (See textbook)

p(at least one element has tail length r) = $1 - e^{-mx}$

Exercise

p(at least one element has tail length r) = $1 - e^{-mx}$

$mx = m2^{-r} = m/2^r$

What will p be if

- $m >> 2^r$
- $m \sim 2^r$
- $m << 2^r$

Flajolet Martin – why the algorithm works

p(at least one element has tail length r) = $1 - e^{-mx}$

$mx = m2^{-r} = m/2^r$

There are 3 cases: $m>>2^r$, $m<<2^r$, $m\sim2^r$

$m>>2^r$

$mx = m2^{-r} = m/2^r$ will be very large

Therefore, $e^{-mx} \sim 0$

Therefore, p(at least one element has tail length r) = $1 - e^{-mx} \sim 1$

So we are likely to find tail lengths r where $m>>2^r$

$m\sim2^r$

$mx = m2^{-r} = m/2^r$ will a small number

Therefore, $e^{-mx} \sim$ will be some fraction

Therefore, p(at least one element has tail length r) = $1 - e^{-mx} \sim$ some fraction

So there is some probabilty to find tail lengths r where $m\sim2^r$

Flajolet Martin – why the algorithm works

p(at least one element has tail length r) = $1 - e^{-mx}$

$mx = m2^{-r} = m/2^r$

There are 3 cases: $m>>2^r$, $m\sim2^r$, $m<<2^r$

Suppose, $m>>2^r$

So we are likely to find tail lengths r where $m>>2^r$

Suppose, $m\sim2^r$

So there is some probabilty to find tail lengths r where $m\sim2^r$

Suppose, $m<<2^r$

$mx = m2^{-r} = m/2^r$ will be very small

Therefore, $e^{-mx} \sim 1$ since $e^0 = 1$

Therefore, p(at least one element has tail length r) = $1 - e^{-mx} \sim 0$

So we are not likely to find tail lengths r where $m<<2^r$

BIG DATA

Flajolet Martin – summary

p(at least one element has tail length r) = $1 - e^{-mx}$

$mx = m2^{-r} = m/2^r$

There are 3 cases: $m>>2^r$, $m\sim2^r$, $m<<2^r$

Suppose, $m>>2^r$

We are likely to find tail lengths r where $m>>2^r$

Suppose, $m\sim2^r$

There is some probabilty to find tail lengths r where $m\sim2^r$

Suppose, $m<<2^r$

We are not likely to find tail lengths r where $m<<2^r$

- We are likely to find tail lengths r where $m>>2^r$ or $m\sim2^r$
- If we take the largest r, we must have $m\sim2^r$

Flajolet Martin – practical considerations

Flajolet Martin – in practise

Simple approach

- If we have only one hash function, *m* will always be a power of 2
- We can pick *k* hash functions, estimate $m=2^R$ for each, take average or median
- Will also give better estimate

Problems with simple approach

- Average will be pulled towards max (maybe outlier)
- Median: estimate will always be power of 2

Flajolet Martin – in practise

Combined approach

- Divide *k* hashes into groups
  - E.g., if we have 6 hash functions $(h_1 \ldots h_6)$, we might have 3 groups where $g_1=(h_1\ h_2)$ and so on
- Compute average of each group
  - E.g.,estimate m_1 as average calculated from $g_1=(h_1\ h_2)$ and so on
- Then median of averages

THANK YOU

K V Subramaniam, Usha Devi

Dept. of Computer Science and Engineering

subramaniamkv@pes.edu

ushadevibg@pes.edu

BIG DATA

Hands On Session - 4

STREAMING SPARK

K V Subramaniam

Usha Devi B G

Dept of Computer Science and Engineering

BIG DATA

STREAMING SPARK Hands-on

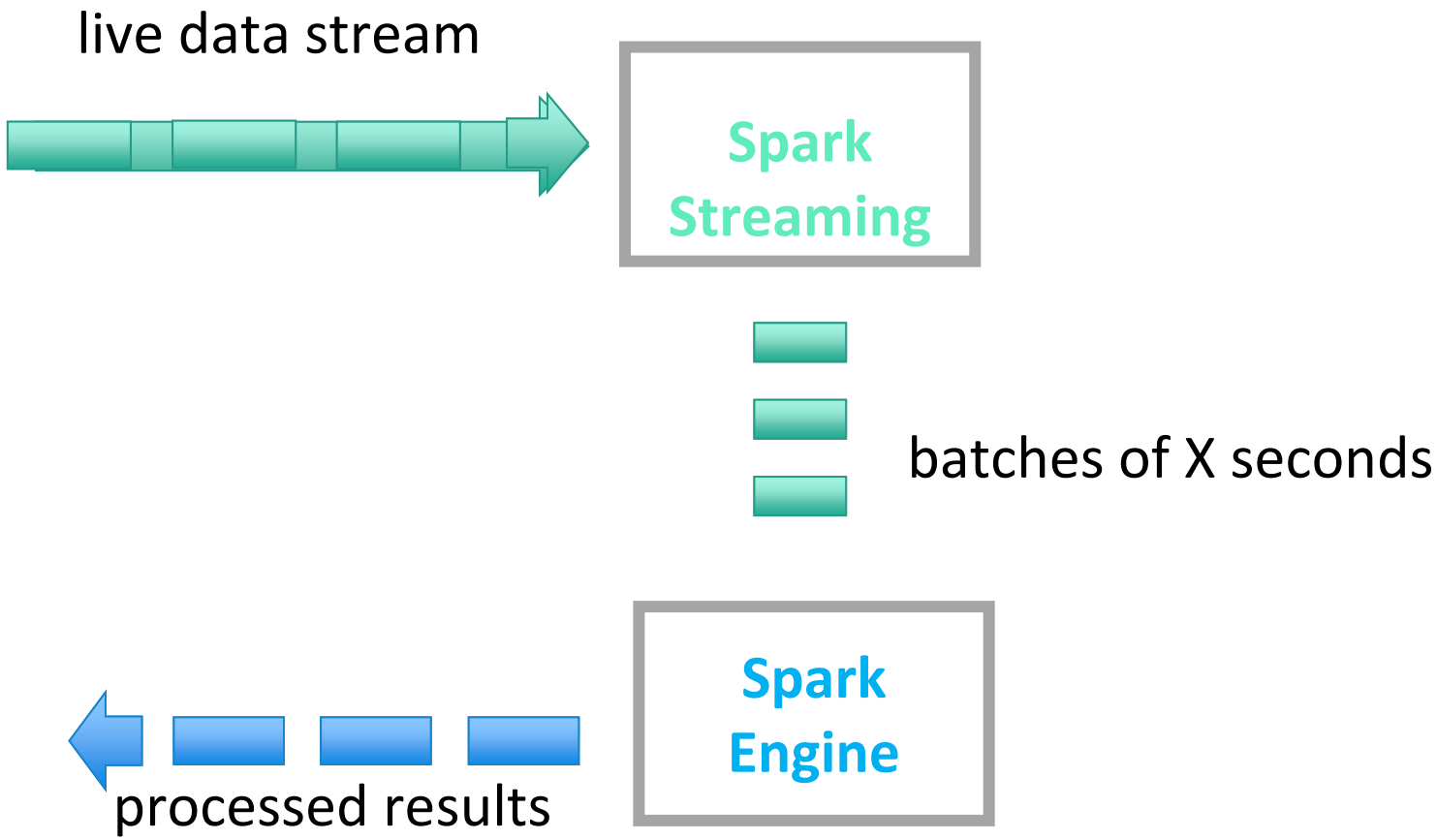
SPECIFICATIONS

1. Hadoop: 3.2
2. Java: 1.8
3. Apache Spark 3.0
4. Dataset: Please download the dataset from the forum.

Configurations

- **Download Apache Spark using the below link**: (choose the version compatible for your Hadoop version)
  https://spark.apache.org/downloads.html
- **Untar and move the file to /usr/local/**
  $ tar –zxvf spark-3.0.0-bin-hadoop2.7.tgz
  $ mv spark-3.0.0-bin-hadoop2.7.tgz /usr/local/spark
- **Edit the .bashrc file to add the spark variables**
  export SPARK_HOME=/usr/local/spark
- **Run the below command to check whether spark has been installed successfully**.
  $ spark-submit

Discretized Stream Processing



BIG DATA

Problem Statement

- Calculate the total number of tweets in the given data set using streaming spark.

- Calculate the number of tweets tweeted using Twitter for Android.

"ID","language","Date","source","len","likes","RTs","Hashtags","Usernames","Userid","name","Place","followers","friends"

01E+18;en;02-07-18 1:35;Twitter for Android;140;0;477;WorldCup POR ENG;Squawka Football;Squawka;Cayleb;Accra;861;828

01E+18;en;02-07-18 1:35;Twitter for Android;139;0;1031;WorldCup;FC Barcelona Ivan Rakitic HNS | CFF;FCBarcelona ivanrakitic HNS_CFF;Febri Aditya;Bogor;667;686

01E+18;en;02-07-18 1:35;Twitter for Android;140;0;477;WorldCup POR ENG;Squawka Football;Squawka;tar;;137;216

01E+18;en;02-07-18 1:35;Twitter Web Client;138;0;1;PowerByEXO WorldCupFIFAStadiumDJ XiuminLeague;Frida Carrillo;FridaCarrillo05;STAN LEGENDS STAN EXO;Lima Peru;7;9

Steps to run spark streaming program

- Create a directory “checkpoint_BIGDATA” with write permission to store the intermediate values while computing the streaming data.

  $mkdir checkpoint_BIGDATA
  $ chmod 777 checkpoint_BIGDATA

- Use the “app.py” to stream data to spark using sockets

  $python3 app.py

- Run “spark.py” to accept streaming data and process it using spark engine

  $spark-submit spark.py

- Stderr and Stdout outputs are jumbled, redirect Stdout to a file to see the actual output

  $spark-submit spark.py > output.txt

THANK YOU

K V Subramaniam
Prafullata Auradkar
Usha Devi B G
Department of Computer Science and Engineering