



Big Stream Processing 2: Spark Streaming

Shiow-yang Wu (吳秀陽)
CSIE, NDHU, Taiwan, ROC

Motivation for Real-Time Stream Processing



- Data is being created at unprecedented rates
 - Exponential data growth from mobile, web, social
 - Connected devices: 9B in 2012 to 50B by 2020
 - Over 1 trillion sensors by 2020
 - Datacenter IP traffic growing at CAGR of 25%
- Many important applications must process **large streams of live data** and provide results in **near-real-time**
 - Social network trends
 - Website statistics
 - Ad impressions
 - ...



Motivation




- How can we harness the data in **real-time**?
 - Value can quickly degrade → capture value immediately
 - From reactive analysis to direct operational impact
 - Unlocks new competitive advantages
 - Requires a completely new approach...
- **Distributed stream processing framework** is required to
 - Scale to large clusters (100s of machines)
 - Achieve low latency (few seconds)

Why Streaming?




- “Without stream processing there’s no big data and no Internet of Things” – Dana Sandu, SQLstream
- **Operational Efficiency** - 1 extra mph for a locomotive on it’s daily route can lead to \$200M in saving (Norfolk Southern)
- **Tracking Behavior** - McDonalds (Netherlands) realized a 700% increase in offer redemptions using personalized advertising based on location, weather, previous purchase, and preference.
- **Predict machine failure** - GE monitors over 5500 assets from 70+ customer sites globally. Can predict failure and determine when something needs maintenance
- **Improving Traffic Safety and Efficiency** – According to EU Commission congestion in EU urban areas costs ~ €100 billion or 1 percent of EU GDP annually

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.



CSIE59830 Big Data Systems
Big Stream Processing 2 – Spark Streaming 5

From Volume and Variety to Velocity



Big Data has evolved

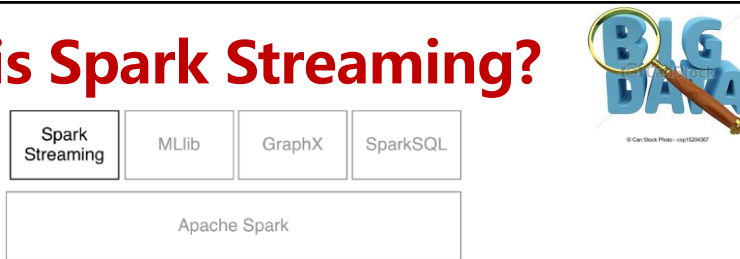
Past	Present
Big-Data = Volume + Variety	Big-Data = Volume + Variety + Velocity

Hadoop Ecosystem evolves as well...

Past	Present
Batch Processing Time to insight of Hours	Batch + Stream Processing Time to Insight of Seconds

CSIE59830 Big Data Systems
Big Stream Processing 2 – Spark Streaming 6


What is Spark Streaming?



- Provides efficient, fault-tolerant stateful stream processing
- Provides a simple API for implementing complex algorithms
- Integrates with Spark's batch and interactive processing
- Integrates with other Spark extensions

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 7

What is Spark Streaming?



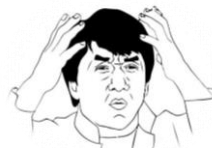
- **Extends Spark** for doing large scale **stream** processing
- **Scales** to 100s of nodes and achieves second scale latencies
- Efficient and **fault-tolerant** stateful stream processing
- Simple **batch-like API** for implementing complex algorithms
- **High throughput** on large data streams

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 8

Integration with Batch Processing



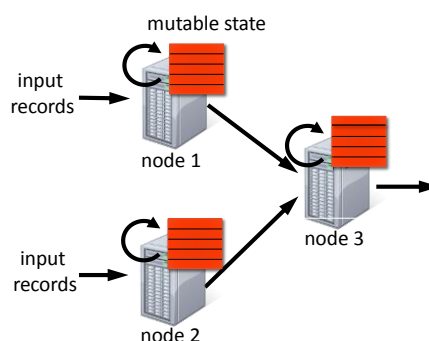
- Many environments require processing same data in **live streaming** as well as **batch** post processing
- Existing framework cannot do both
 - Either do stream processing of 100s of MB/s with low latency
 - Or do batch processing of TBs / PBs of data with high latency
- Extremely painful to maintain two different stacks
 - Different programming models
 - Double the implementation effort
 - Double the number of bugs



Stateful Stream Processing



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



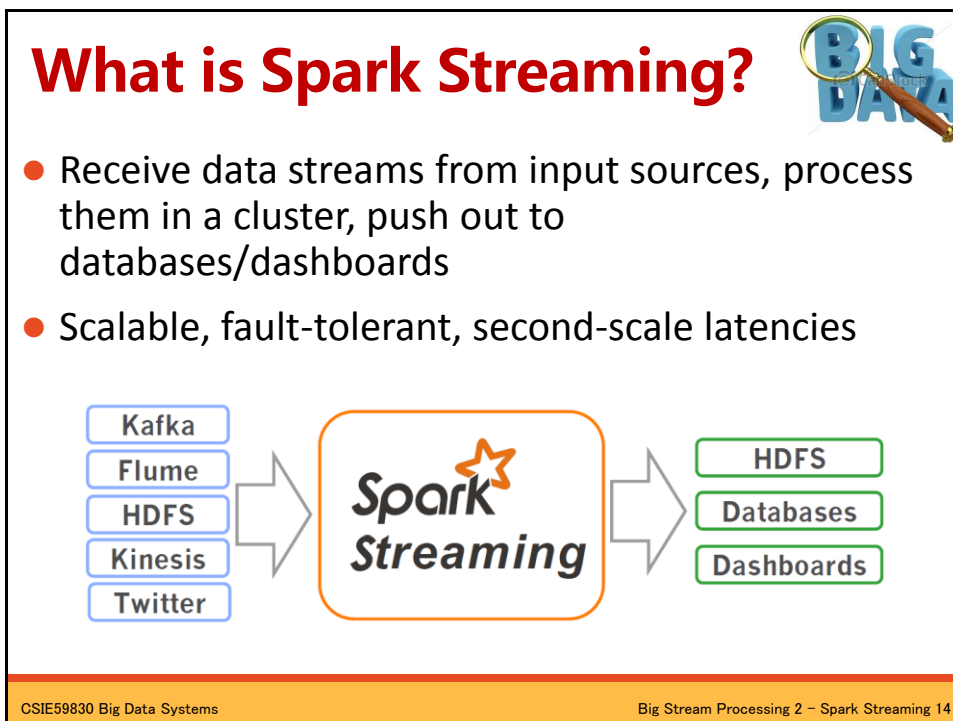
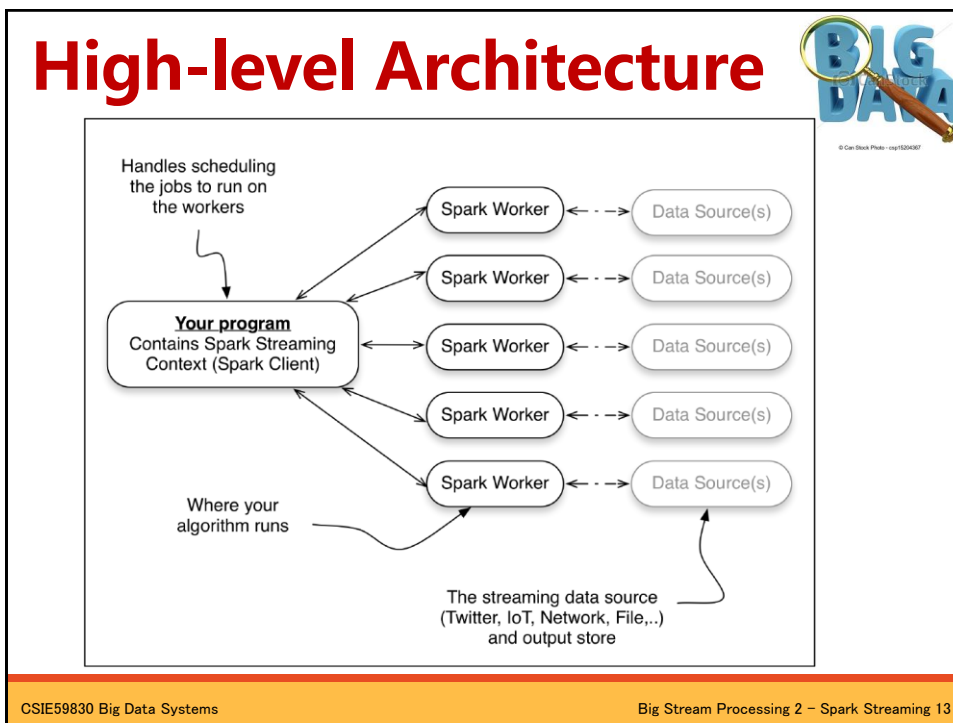
Existing Streaming Systems



- **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 to external database is slow

Spark Streaming

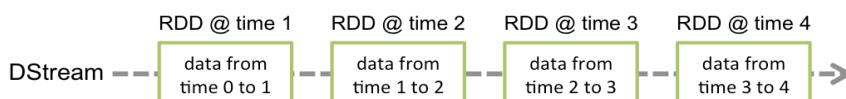




Spark Streaming



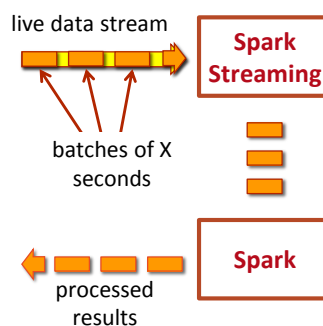
- Incoming data represented as **Discretized Streams (DStreams)**
- Stream is broken down into **micro-batches**
- Each micro-batch is an **RDD** – can share code between batch and streaming



Discretized Stream Processing



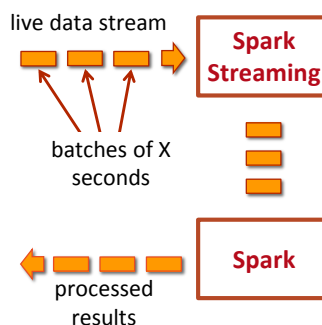
- Run a streaming computation as a **series of very small, deterministic batch jobs**
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing



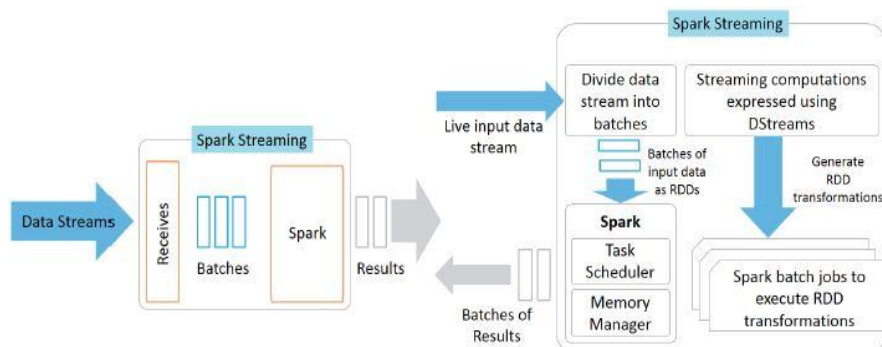
- Run a streaming computation as a **series of very small, deterministic batch jobs**
- Batch sizes as low as $\frac{1}{2}$ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system



Working of Spark Streaming



- It takes **live input data streams** and then divides them into **batches**. After this, the Spark engine processes those streams and generates the **final stream results in batches**.



Spark Streaming Programming Model



- *Discretized Stream (DStream)*
 - Represents a stream of data
 - Implemented as a sequence of RDDs
- DStreams API very similar to RDD API
 - Functional APIs in Scala, Java
 - Create input DStreams from different sources
 - Apply parallel operations

Example – Get hashtags from Twitter



```
val tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data

Twitter Streaming API

batch @ t

batch @ t+1

batch @ t+2




tweets DStream



stored in memory as an RDD
(immutable, distributed)

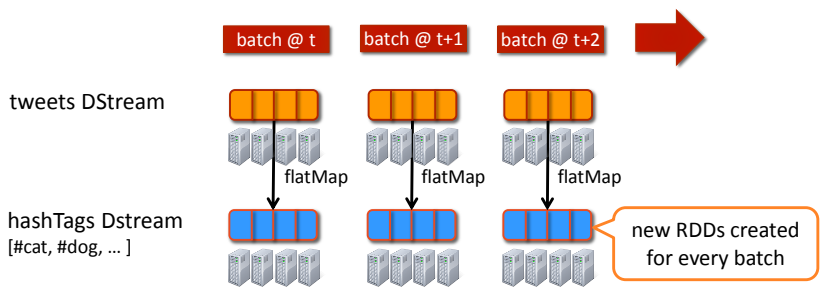
Example – Get hashtags from Twitter



```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

transformation: modify data in one DStream to create another DStream



tweets DStream

hashTags DStream
[#cat, #dog, ...]


batch @ t batch @ t+1 batch @ t+2

flatMap

new RDDs created for every batch

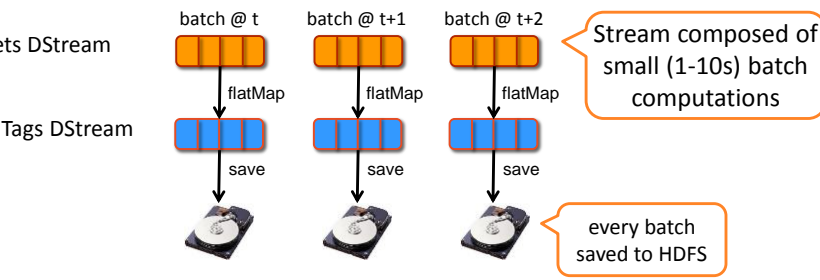
CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 21

“Micro-batch” Architecture



```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage



tweets DStream

hashTags DStream

batch @ t batch @ t+1 batch @ t+2

flatMap


save

Stream composed of small (1-10s) batch computations

every batch saved to HDFS

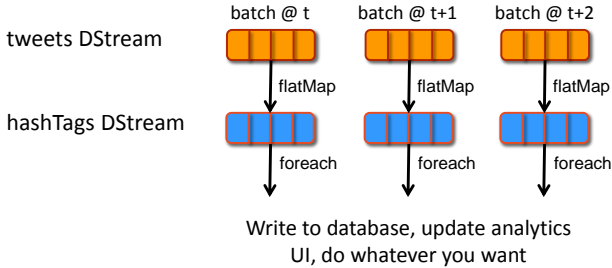
CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 22

Example – Get hashtags from Twitter



```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



tweets DStream

hashTags DStream

batch @ t batch @ t+1 batch @ t+2


flatMap flatMap flatMap

foreach foreach foreach

Write to database, update analytics
UI, do whatever you want

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 23

Languages



Scala

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java

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

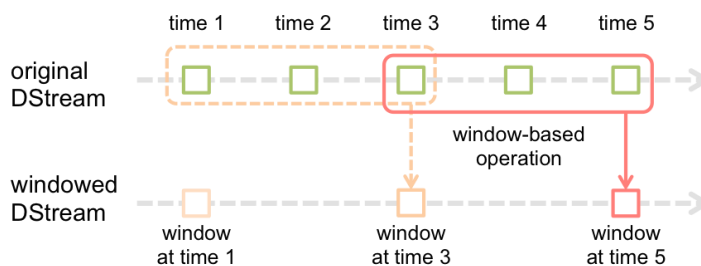
Function object

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 24

Window-based Operations



- To apply transformations over a sliding window of data



- Two parameters
 - Window length**: the duration of the window
 - Sliding interval**: the interval at which the window operation is performed

Window-based Transformations



```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

sliding window
operation

window length

sliding interval

Arbitrary Stateful Computations



- Specify function to generate new state based on previous state and new data
- Example: Maintain per-user mood as state, and update it with their tweets

```
updateMood(newTweets, lastMood) => newMood  
moods = tweets.updateStateByKey(updateMood _)
```

Arbitrary Combinations of Batch and Streaming Computations



Inter-mix RDD and DStream operations!

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

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

DStream Input Sources



- Out of the box:
 - Kafka, HDFS, Flume, Akka Actors, Raw TCP sockets
- Very easy to write a *receiver* for your own data source
 - Define what to when receiver is started and stopped
- Also, generate your own sequence of RDDs, etc. and push them in as a “stream”

Output Sinks



- HDFS, S3, etc (Hadoop API compatible filesystems)
- Cassandra (using Spark-Cassandra connector)
- HBase (existing Spark-Hbase connector can be used directly)
- Directly push the data anywhere

Spark Streaming



- Runs as a Spark job
- YARN or standalone for scheduling
 - YARN has KDC(Kerberos Key Distribution Center) integration
- Use the **same code** for **real-time** Spark Streaming and for **batch** Spark jobs.
- Integrates natively with messaging systems such as Flume, Kafka, Zero MQ....
- Easy to write “**Receivers**” for custom messaging systems.

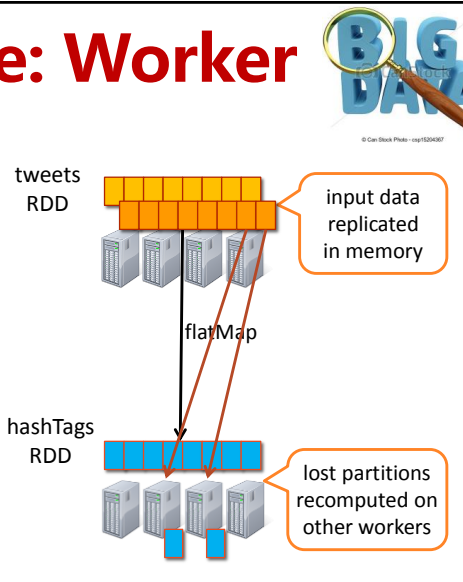
DStreams + RDDs = Power



- Combine live data streams with historical data
 - Generate historical data models with Spark, etc.
 - Use data models to process live data stream
- Combine streaming with MLlib, GraphX algos
 - Offline learning, online prediction
 - Online learning and prediction
- Interactively query streaming data using SQL
 - `select * from table_from_streaming_data`



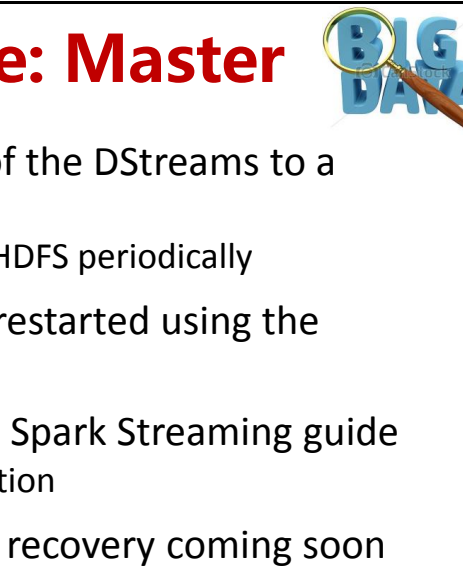
Fault-tolerance: Worker



- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure can be recomputed from replicated input data
- All transformed data is fault-tolerant, and exactly-once transformations

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 33

Fault-tolerance: Master



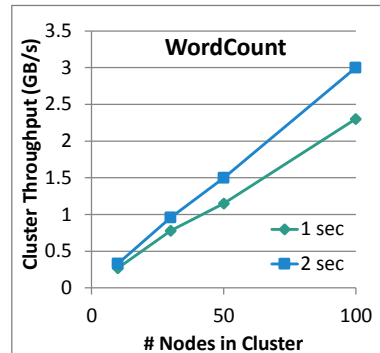
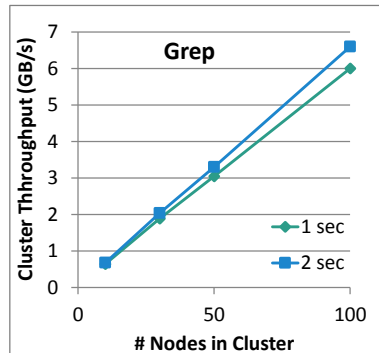
- Master saves the state of the DStreams to a **checkpoint** file
 - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- More information in the Spark Streaming guide
 - Link later in the presentation
- Automated master fault recovery coming soon

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 34

Performance



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



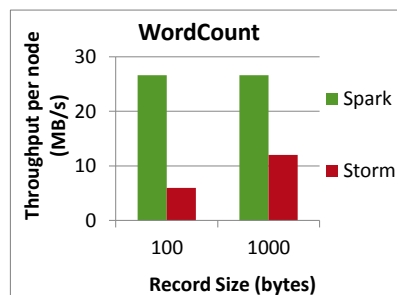
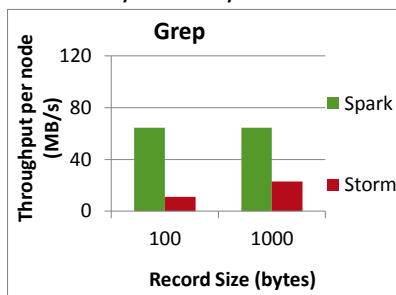
CSIE59830 Big Data Systems

Big Stream Processing 2 – Spark Streaming 35

Comparison with Storm and S4



- Higher throughput than Storm and S4
 - Spark Streaming: **670k** records/second/node
 - Storm: **115k** records/second/node
 - Apache S4(Simple Scalable Streaming System): 7.5k records/second/node



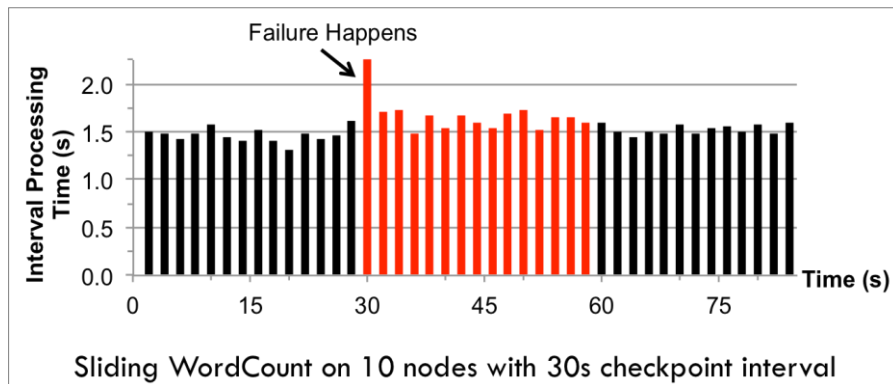
CSIE59830 Big Data Systems

Big Stream Processing 2 – Spark Streaming 36

Fast Fault Recovery



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



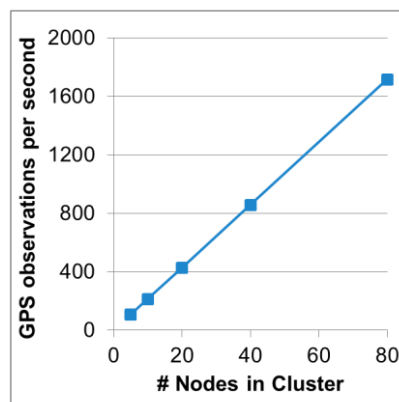
CSIE59830 Big Data Systems

Big Stream Processing 2 – Spark Streaming 37

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

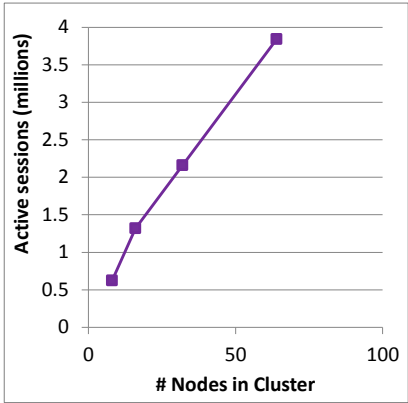


CSIE59830 Big Data Systems

Big Stream Processing 2 – Spark Streaming 38

Real Applications: Conviva

- Real-time monitoring and optimization of video metadata
- Aggregation of performance data from millions of active video sessions across thousands of metrics
- Multiple stages of aggregation
- Successfully ported to run on Spark Streaming
- Scales linearly with cluster size



# Nodes in Cluster	Active sessions (millions)
10	0.6
20	1.4
30	2.2
60	3.8


CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 39

Unifying Batch and Stream Processing Models

- Spark program on Twitter log file using RDDs
 - `val tweets = sc.hadoopFile("hdfs://...")`
 - `val hashTags = tweets.flatMap(status => getTags(status))`
 - `hashTags.saveAsHadoopFile("hdfs://...")`
- Spark Streaming program on Twitter stream using DStreams
 - `val tweets = ssc.twitterStream()`
 - `val hashTags = tweets.flatMap(status => getTags(status))`
 - `hashTags.saveAsHadoopFiles("hdfs://...")`

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 40

Vision - one stack to rule them all



- Explore data interactively using Spark Shell 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"))
...
scala> val mapped = filtered.map(...)
...


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

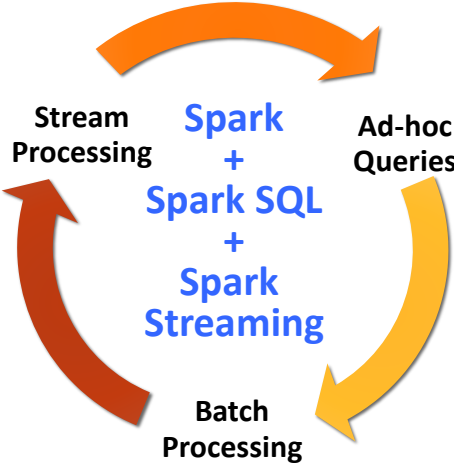
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}

```

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 41

Vision - one stack to rule them all





Stream Processing **Spark** **Ad-hoc Queries**
 +
Spark SQL
 +
Spark Streaming
 +
Batch Processing

CSIE59830 Big Data Systems Big Stream Processing 2 – Spark Streaming 42

Conclusions & References



- Integrated with Spark as an extension
 - Takes 5 minutes to spin up a Spark cluster to try it out
- Streaming programming guide –
<http://spark.incubator.apache.org/docs/latest/streaming-programming-guide.html>
- Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. ACM SOSP 2013.