Information Flow Processing

Amir H. Payberah
Swedish Institute of Computer Science

amir@sics.se
May 8, 2014

Motivation

- Many applications must process large streams of live data and provide results in real-time.

Motivation

- Many applications must process large streams of live data and provide results in real-time.
  - Wireless sensor networks
  - Traffic management applications
  - Stock marketing
  - Environmental monitoring applications
  - Fraud detection tools
  - ...

Motivation

- Processing information as it flows, without storing them persistently.

Motivation

- Processing information as it flows, without storing them persistently.

- Traditional DBMSs:
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.

Motivation

- Processing information as it flows, without storing them persistently.

- Traditional DBMSs:
  - Store and index data before processing it.
  - Process data only when explicitly asked by the users.
  - Both aspects contrast with our requirements.

One Name, Different Technologies

- Several research communities are contributing in this area:
  - Each brings its own expertise
  - Point of view
  - Vocabulary: event, data, stream, ...

One Name, Different Technologies

- Several research communities are contributing in this area:
  - Each brings its own expertise
  - Point of view
  - Vocabulary: event, data, stream, ...

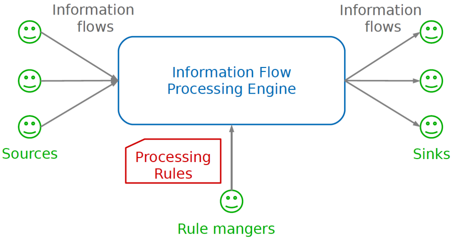
Tower of Babel Syndrome!

Come on! Let's go down and confuse them by making them speak different languages, then they won't be able to understand each other.

Genesis 11:7

Information Flow Processing (IFP)

- Source: produces the incoming information flows
- Sink: consumes the results of processing
- IFP engine: processes incoming flows
- Processing rules: how to process the incoming flows
- Rule manager: adds/removes processing rules



IFP Competing Models

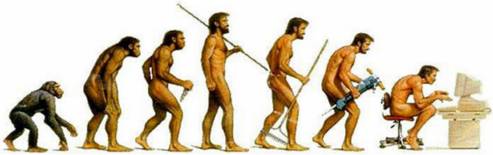
- Data Stream Management Systems (DSMS)
- Complex Event Processing (CEP)

IFP Competing Models

- Data Stream Management Systems (DSMS)
- Complex Event Processing (CEP)

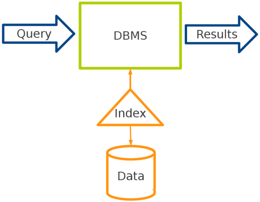
Data Stream Management Systems (DSMS)

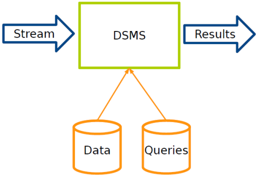
- An evolution of traditional data processing, as supported by DBMSs.



DBMS vs. DSMS (1/3)

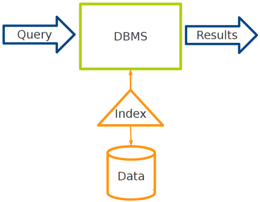
- DBMS: persistent data where updates are relatively infrequent.
- DSMS: transient data that is continuously updated.

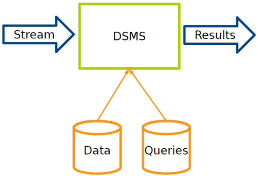




DBMS vs. DSMS (2/3)

- DBMS: runs queries just once to return a complete answer.
- DSMS: executes standing queries, which run continuously and provide updated answers as new data arrives.





DBMS vs. DSMS (3/3)

- Despite these differences, DSMSs resemble DBMSs: both process incoming data through a sequence of transformations based on SQL operators, e.g., selections, aggregates, joins.

Out of Scope of DSMS

- DSMSs focus on producing query answers.
- Detection and notification of complex patterns of elements are usually out of the scope of DSMSs:

Out of Scope of DSMS

- DSMSs focus on producing query answers.
- Detection and notification of complex patterns of elements are usually out of the scope of DSMSs: Complex Event Processing

IFP Competing Models

- Data Stream Management Systems (DSMS)
- Complex Event Processing (CEP)

Complex Event Processing (CEP)

- DSMSs limitation: detecting complex patterns of incoming items, involving sequencing and ordering relationships.

- CEP models flowing information items as notifications of events happening in the external world.
  - They have to be filtered and combined to understand what is happening in terms of higher-level events.

CEP vs. Publish/Subscribe Systems

- CEP systems can be seen as an extension to traditional publish/subscribe systems.

CEP vs. Publish/Subscribe Systems

- CEP systems can be seen as an extension to traditional publish/subscribe systems.
- Traditional publish/subscribe systems consider each event separately from the others, and filter them based on their topic or content.

CEP vs. Publish/Subscribe Systems

- CEP systems can be seen as an extension to traditional publish/subscribe systems.
- Traditional publish/subscribe systems consider each event separately from the others, and filter them based on their topic or content.
- CEPs extend this functionality by increasing the expressive power of the subscription language to consider complex event patterns that involve the occurrence of multiple related events.

Outline

- Stream processing engine
- Fault tolerance
- Related work
- Spark Stream (DStream)

Outline

- Stream processing engine
- Fault tolerance
- Related work
- Spark Stream (DStream)

Stream processing engine

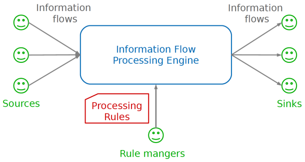
- Stream: a sequence of unbounded tuples generated continuously in time: $\cdots(a_1, a_2, \cdots, a_n, t-1)(a_1, a_2, \cdots, a_n, t)(a_1, a_2, \cdots, a_n, t+1)\cdots$, where a_i denotes an attribute.

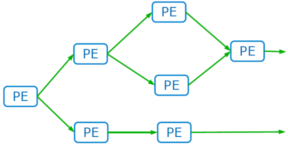
Stream processing engine

- Stream: a sequence of unbounded tuples generated continuously in time: $\cdots(a_1, a_2, \cdots, a_n, t-1)(a_1, a_2, \cdots, a_n, t)(a_1, a_2, \cdots, a_n, t+1)\cdots$, where a_i denotes an attribute.

- Stream processing engine: creates a logical network of PEs connected in a DAG.

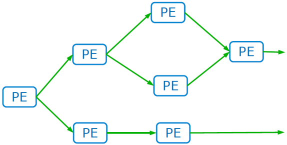
- Processing Element (PE): a processing unit in a stream processing engine.





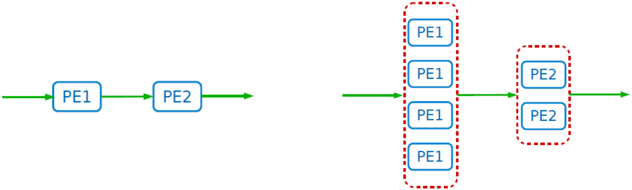
Processing Element (PE)

- Execute independently and in parallel
- Not synchronized
- Communicate through messaging: push-based vs. pull-based
- Upstream node vs. downstream node
- PE output: not emit a tuple, emit a tuple, or emit a tuple in a periodic manner



PE Physical Deployment

- A single PE can be running in parallel on different nodes.



Outline

- Stream processing engine
- Fault tolerance
- Related work
- Spark Stream (DStream)

Fault Tolerance

- The recovery methods of streaming frameworks must take:
  - Correctness, e.g., data loss and duplicates
  - Performance, e.g., low latency

Basic Idea

- Each processing node has an associated backup node.
- The backup node's stream processing engine is identical to the primary one.
- But the state of the backup node is not necessarily the same as that of the primary.
- If a primary node fails, its backup node takes over the operation of the failed node.

Recovery Methods

- GAP recovery
- Rollback recovery
- Precise recovery

GAP Recovery

- The weakest recovery guarantee
- A new task takes over the operations of the failed task.
- The new task starts from an empty state.
- Tuples can be lost during the recovery phase.

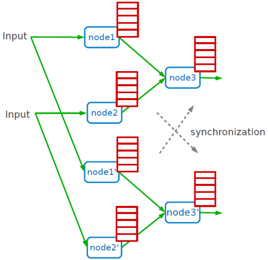
Rollback Recovery

- The information loss is avoided, but the output may contain duplicate tuples.

- Three types of rollback recovery:
  - Active backup
  - Passive backup
  - Upstream backup

Rollback Recovery - Active Backup

- Both primary and backup nodes are given the same input.
- The output tuples of the backup node are logged at the output queues and they are not sent downstream.
- If the primary fails, the backup takes over by sending the logged tuples to all downstream neighbors and then continuing its processing.

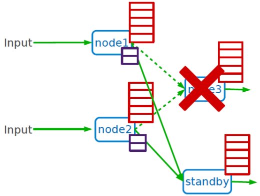


Rollback Recovery - Passive Backup

- Periodically check-points processing state to a shared storage.
- The backup node takes over from the latest checkpoint when the primary fails.
- The backup node is always equal or behind the primary.

Rollback Recovery - Upstream Backup

- Upstream nodes store the tuples until the downstream nodes acknowledge them.
- If a node fails, an empty node rebuilds the latest state of the failed primary from the logs kept at the upstream server.
- There is no backup node in this model.



Precise Recovery

- Post-failure output is exactly the same as the output without failure.
- Can be achieved by modifying the algorithms for rollback recovery.
  - For example, in passive backup, after a failure occurs the backup node can ask the downstream nodes for the latest tuples they received and trim the output queues accordingly to prevent the duplicates.

Outline

- Stream processing engine
- Fault tolerance
- Related work
- Spark Stream (DStream)

Related Work

- Aurora
- Borealis
- Storm
- S4

Aurora

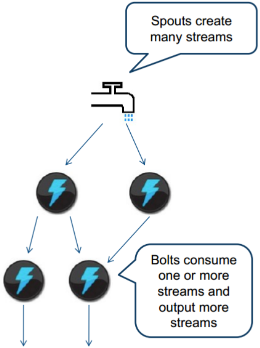
- A single site stream-processing engine (centralized).
- DAG based processing model for streams.
- Push-based strategy.
- The first Aurora did not support fault tolerance.
- Stream Query Algebra (SQuAl), i.e., SQL with additional features, e.g., windowed queries.

Borealis

- Distributed version of Aurora.
- Advanced functionalities on top of Aurora:
  - Dynamic revision of query results: correct errors in previously reported data.
  - Dynamic query modifications: change certain attributes of the query at runtime.
- Pull-based strategy.
- Rollback recovery with active backup.

Storm (1/2)

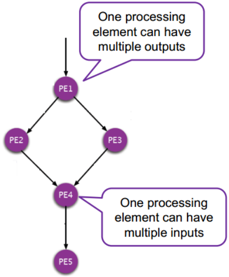
- Stream processing is guaranteed: a message cannot be lost due to node failures.
- DAG based processing:
  - the DAG is called Topology
  - the PEs are called Bolts
  - the stream sources are called Spouts
- It does not have an explicit programming paradigm.



Storm (2/2)

- Pull-based strategy.
- Rollback recovery with upstream backup.
- Three sets of nodes:
  - Nimbus: distributes the code among the worker nodes, and keeps track of the progress of the worker nodes
  - Supervisor: the set of worker nodes
  - Zookeeper: coordination between supervisor nodes and the Nimbus
- Built by twitter

S4 (1/2)



- S4: Simple Scalable Streaming System.
- Constructing a DAG structure of PEs at runtime.
  - A PE is instantiated for each value of the key attribute.
- The processing model is inspired by MapReduce.
- Events are dispatched to nodes according to their key.

S4 (2/2)

- Push-based strategy
- GAP recovery
- Communication layer: coordination between the processing nodes and the messaging between nodes.
  - Uses Zookeeper
- Built by yahoo

Outline

- Stream processing engine
- Fault tolerance
- Related work
- Spark Stream (DStream)

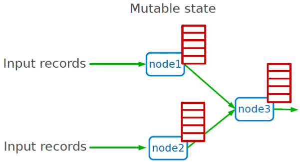
Motivation

- To run stream processing at large scale, system has to be both:
  - Fault-tolerant: recover quickly from failures and stragglers.
  - Cost-efficient: do not require significant hardware beyond that needed for basic processing.

- Existing streaming systems do not have both properties.

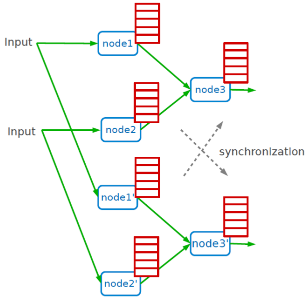
Existing Streaming Systems (1/2)

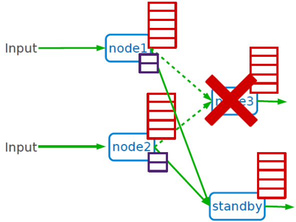
- Record-at-a-time processing model:
  - Each node has mutable state.
  - For each record, updates state and sends new records.
  - State is lost if node dies.



Existing Streaming Systems (2/2)

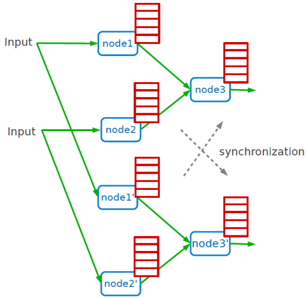
- Fault tolerance via replication or upstream backup.

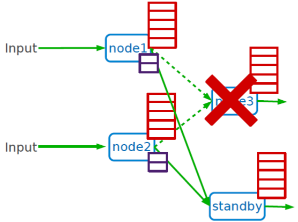




Existing Streaming Systems (2/2)

- Fault tolerance via replication or upstream backup.



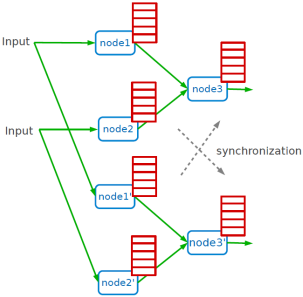


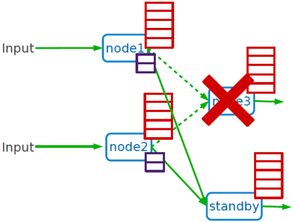
Fast recovery, but 2x hardware cost

Only need one standby, but slow to recover

Existing Streaming Systems (2/2)

- Fault tolerance via replication or upstream backup.





Fast recovery, but 2x hardware cost

Only need one standby, but slow to recover

Neither approach tolerates stragglers.

Observation

- Batch processing models for clusters provide fault tolerance efficiently.
- Divide job into deterministic tasks.
- Rerun failed/slow tasks in parallel on other nodes.

Idea

Idea

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

Challenges

- Latency (interval granularity)
  - Traditional batch systems replicate state on-disk storage: slow
- Recovering quickly from faults and stragglers

Proposed Solution

- Latency (interval granularity)
  - Resilient Distributed Dataset (RDD)
  - Keep data in memory
  - No replication

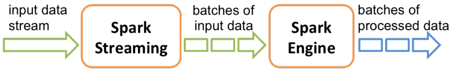
- Recovering quickly from faults and stragglers
  - Storing the lineage graph
  - Using the determinism of D-Streams
  - Parallel recovery of a lost node's state

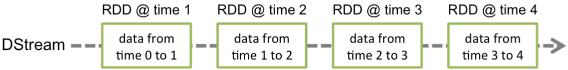
Discretized Stream Processing (D-Stream)

- 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.

D-Stream API (1/4)

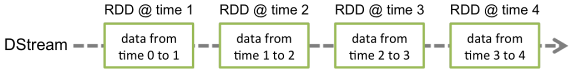
- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...





D-Stream API (1/4)

- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...

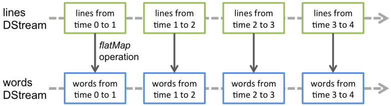


- Initializing Spark streaming

```
val scc = new StreamingContext(master, appName, batchDuration,
[sparkHome], [jars])
```

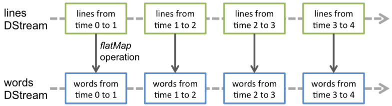
D-Stream API (2/4)

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless operations): map, join, ...

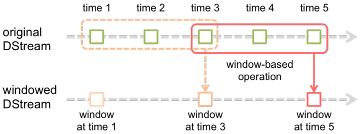


D-Stream API (2/4)

- Transformations: modify data from on DStream to a new DStream.
  - Standard RDD operations (stateless operations): map, join, ...



  - Stateful operations: group all the records from a sliding window of the past time intervals into one RDD: window, reduceByAndWindow, ...



  Window length: the duration of the window.
  Slide interval: the interval at which the operation is performed.

D-Stream API (3/4)

- Output operations: send data to external entity
  - saveAsHadoopFiles, foreach, print, ...

D-Stream API (3/4)

- Output operations: send data to external entity
  - saveAsHadoopFiles, foreach, print, ...

- Attaching input sources

```
ssc.textFileStream(directory)
ssc.socketStream(hostname, port)
```

D-Stream API (4/4)

- Stream + Batch: It can be used to apply any RDD operation that is not exposed in the DStream API.

```
val spamInfoRDD = sparkContext.hadoopFile(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = inputDStream.transform(_.join(spamInfoRDD).filter(...))
```

D-Stream API (4/4)

- Stream + Batch: It can be used to apply any RDD operation that is not exposed in the DStream API.

```
val spamInfoRDD = sparkContext.hadoopFile(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = inputDStream.transform(_.join(spamInfoRDD).filter(...))
```

- Stream + Interactive: Interactive queries on stream state from the Spark interpreter

```
freqs.slice("21:00", "21:05").topK(10)
```

D-Stream API (4/4)

- Stream + Batch: It can be used to apply any RDD operation that is not exposed in the DStream API.

```
val spamInfoRDD = sparkContext.hadoopFile(...)
// join data stream with spam information to do data cleaning
val cleanedDStream = inputDStream.transform(_.join(spamInfoRDD).filter(...))
```

- Stream + Interactive: Interactive queries on stream state from the Spark interpreter

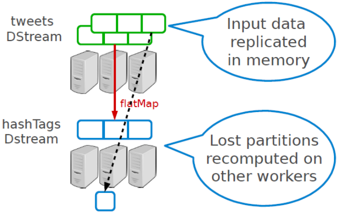
```
freqs.slice("21:00", "21:05").topK(10)
```

- Starting/stopping the streaming computation

```
ssc.start()
ssc.stop()
ssc.awaitTermination()
```

Fault Tolerance

- Spark remembers the sequence of operations that creates each RDD from the original fault-tolerant input data (lineage graph).
- Batches of input data are replicated in memory of multiple worker nodes.
- Data lost due to worker failure, can be recomputed from input data.

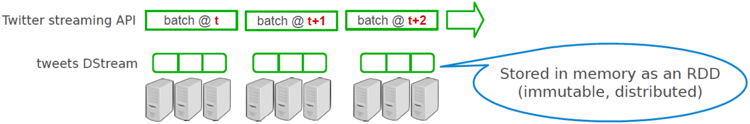


Example 1 (1/3)

- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
```

DStream: a sequence of RDD representing a stream of data

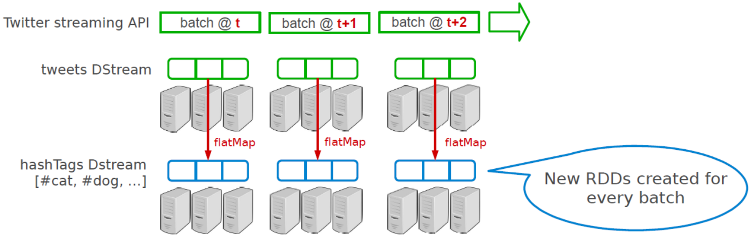


Example 1 (2/3)

- Get hash-tags from Twitter.

```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap(status => getTags(status))
```

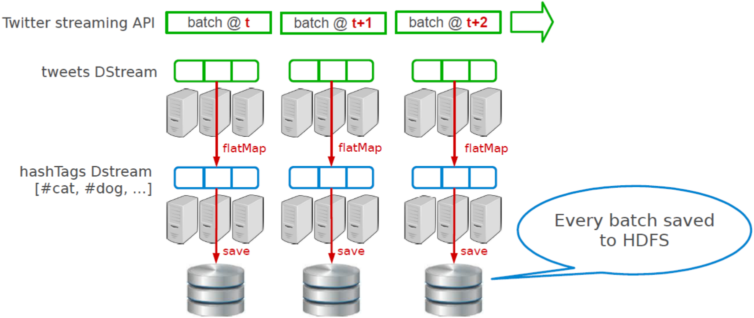
transformation: modify data in one DStream
to create another DStream



Example 1 (3/3)

- Get hash-tags from Twitter.

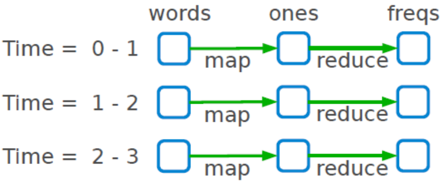
```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Example 2

- Count frequency of words received every second.

```
val ssc = new StreamingContext(args(0), "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(args(1), args(2).toInt)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs = ones.reduceByKey(_ + _)
```



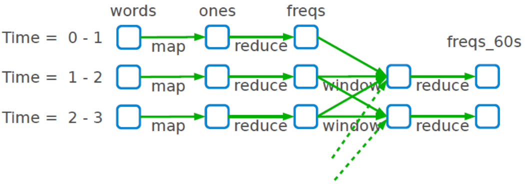
Example 3

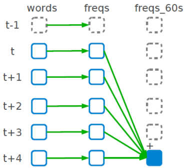
- Count frequency of words received in last minute.

```
val ssc = new StreamingContext(args(0), "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(args(1), args(2).toInt)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs = ones.reduceByKey(_ + _)
val freqs_60s = freqs.window(Seconds(60), Second(1)).reduceByKey(_ + _)
```

window length

window movement

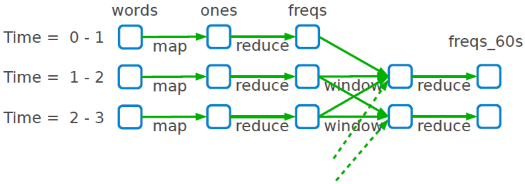


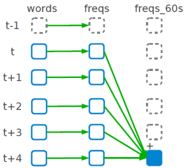


Example 3 - Simpler Model

- Count frequency of words received in last minute.

```
val ssc = new StreamingContext(args(0), "NetworkWordCount", Seconds(1))
val lines = ssc.socketTextStream(args(1), args(2).toInt)
val words = lines.flatMap(_.split(" "))
val ones = words.map(x => (x, 1))
val freqs_60s = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))
```



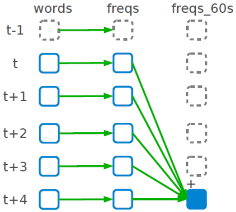


Example 3 - Incremental Window Operators

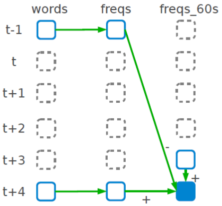
- Count frequency of words received in last minute.

```
// Associative only
freqs_60s = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))

// Associative and invertible
freqs_60s = ones.reduceByKeyAndWindow(_ + _, _ - _, Seconds(60), Seconds(1))
```



Associative only



Associative and invertible

Example 4 - Standalone Application (1/2)

```scala
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.storage.StorageLevel

object NetworkWordCount {
  def main(args: Array[String]) {
    ...
    val ssc = new StreamingContext(args(0), "NetworkWordCount", Seconds(1))
    val lines = ssc.socketTextStream(args(1), args(2).toInt)

    val words = lines.flatMap(_.split(" "))
    val ones = words.map(x => (x, 1))
    freqs = ones.reduceByKey(_ + _)
    freqs.print()

    ssc.start()
    ssc.awaitTermination()
  }
}
```

Example 4 - Standalone Application (2/2)

- sics.sbt:

```
name := "Stream Word Count"

version := "1.0"

scalaVersion := "2.10.3"

libraryDependencies ++= Seq(
    "org.apache.spark" %% "spark-core" % "0.9.0-incubating",
    "org.apache.spark" %% "spark-streaming" % "0.9.0-incubating"
)

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Summary

- IFP: DSMS and CEP
- Recovering models: GAP, Rollback, and Precise
- Spark Stream
  - Run a streaming computation as a series of very small, deterministic batch jobs.
  - DStream: sequence of RDDs
  - Operators: Transformations (stateless and stateful) and output operations

Questions?

Acknowledgements

Some slides and pictures were derived from Matei Zaharia slides and the Spark web site (`http://spark.apache.org/`).