Stream Processing In The Cloud

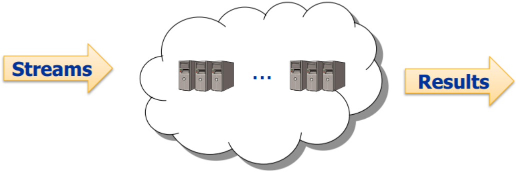
Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Stream Processing In The Cloud

Motivation

- Users of big data applications expect fresh results.
- New stream processing systems are designed to scale to large numbers of cloud-hosted machines.



Motivation

- Clouds provide virtually infinite pools of resources.
- Fast and cheap access to new machines (VMs) for operators.
- How do you decide on the optimal number of VMs?
  - Over-provisioning system is expense.
  - Too few nodes leads to poor performance.

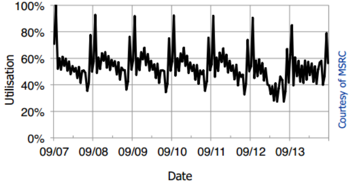
Challenges

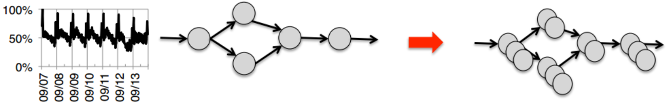
- Elastic data-parallel processing

- Fault-tolerant processing

Challenge: Elastic Data-Parallel Processing

- Typical stream processing workloads are bursty.
- High and bursty input rates $\rightarrow$ detect bottleneck + parallelize



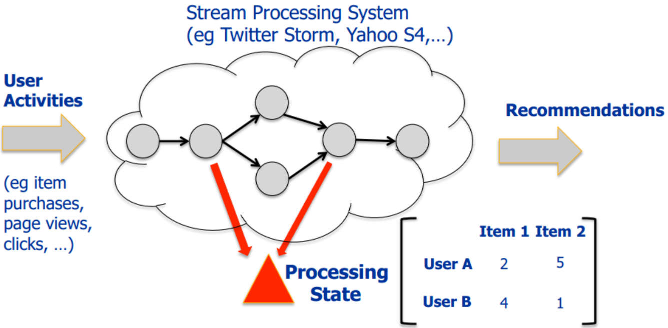


Challenge: Fault-Tolerant Processing

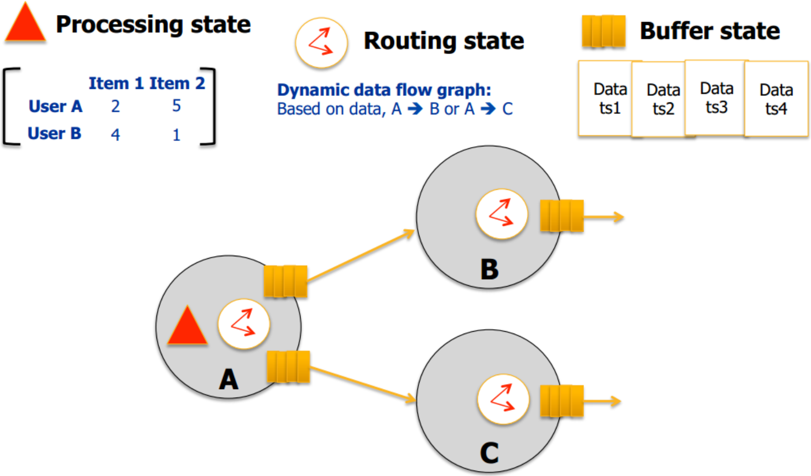
- Large scale deployment → handle node failures.

States in Stream Processing

- Many online applications, like machine learning algorithms, require state.

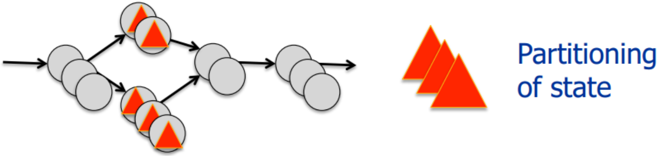


What is State?



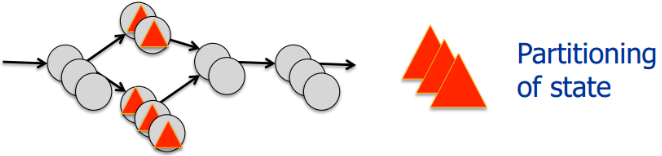
State Complicates Things

- Dynamic scale out impacts state.

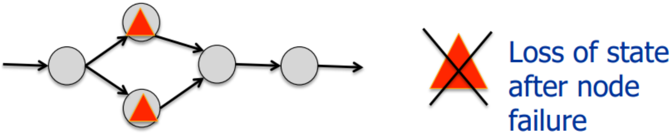


State Complicates Things

- Dynamic scale out impacts state.



- Recovery from failures.



Operators States

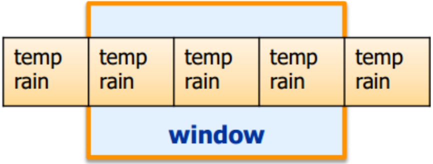
- Stateless operators, e.g., filter and map

Operators States

- Stateless operators, e.g., filter and map
- Stateful operators, e.g., join and aggregate

Operators States

- Stateless operators, e.g., filter and map
- Stateful operators, e.g., join and aggregate
- Window operators, use use the concept of a finite window of tuples.



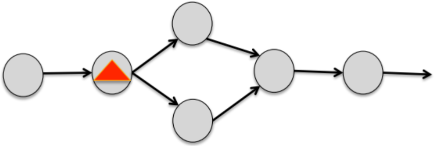
SEEP

Contribution

- Build a stream processing system that scale out while remaining fault tolerant when queries contain stateful operators.

Core Idea

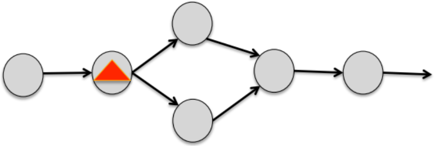
- Make operator state an external entity that can be managed by the stream processing system.



Core Idea

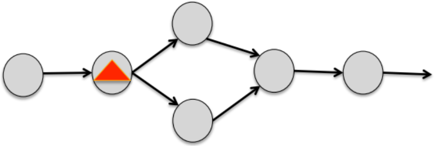
- Make operator state an external entity that can be managed by the stream processing system.

- Operators have direct access to states.



Core Idea

- Make operator state an external entity that can be managed by the stream processing system.
- Operators have direct access to states.
- The system manages states.



Operator State Management

- On scale out: partition operator state correctly, maintaining consistency

Operator State Management

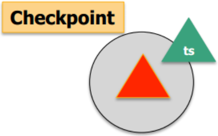
- On scale out: partition operator state correctly, maintaining consistency
- On failure recovery: restore state of failed operator

Operator State Management

- On scale out: partition operator state correctly, maintaining consistency
- On failure recovery: restore state of failed operator
- Define primitives for state management and build other mechanisms on top of them.

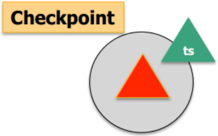
State Management Primitives

- Checkpoint
  - Makes state available to system.
  - Attaches last processed tuple timestamp.

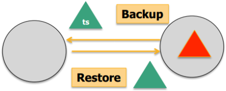


State Management Primitives

- Checkpoint
  - Makes state available to system.
  - Attaches last processed tuple timestamp.

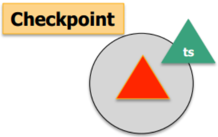


- Backup/Restore
  - Moves copy of state from one operator to another.

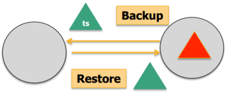


State Management Primitives

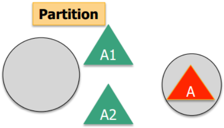
- Checkpoint
  - Makes state available to system.
  - Attaches last processed tuple timestamp.



- Backup/Restore
  - Moves copy of state from one operator to another.



- Partition
  - Splits state to scale out an operator.



State Primitives: Checkpoint

- Checkpoint state = the processing state + the buffer state

State Primitives: Checkpoint

- Checkpoint state = the processing state + the buffer state
- That routing state is not included in the state checkpoint.
  - It only changes in case of scale out or recovery.

State Primitives: Checkpoint

- Checkpoint state = the processing state + the buffer state
- That routing state is not included in the state checkpoint.
  - It only changes in case of scale out or recovery.
- The system executes checkpoint asynchronously and periodically.

State Primitives: Backup and Restore (1/2)

- The operator state (i.e., the checkpoint output) is backed up to an upstream operator.

State Primitives: Backup and Restore (1/2)

- The operator state (i.e., the checkpoint output) is backed up to an upstream operator.

- After the operator state was backed up, already processed tuples from output buffers in upstream operators can be discarded.
    - They are no longer required for failure recovery.

State Primitives: Backup and Restore (2/2)

- Backed up operator state is restored to another operator to recover a failed operator or to redistribute state across partitioned operators.

State Primitives: Backup and Restore (2/2)

- Backed up operator state is restored to another operator to recover a failed operator or to redistribute state across partitioned operators.

- After restoring the state, the system replays unprocessed tuples in the output buffer from an upstream operator to bring the operator's processing state up-to-date.

State Primitives: Partition

- Split the state of a stateful operator across the new partitioned operators when it scales out.

State Primitives: Partition

- Split the state of a stateful operator across the new partitioned operators when it scales out.
- Partitioning the key space of the tuples processed by the operator.

State Primitives: Partition

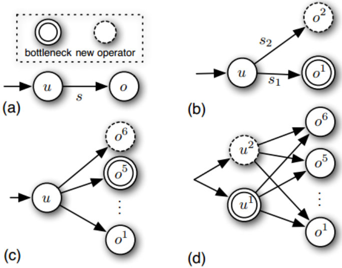
- Split the state of a stateful operator across the new partitioned operators when it scales out.
- Partitioning the key space of the tuples processed by the operator.
- The routing state of its upstream operators must also be updated to account for the new partitioned operators.

State Primitives: Partition

- Split the state of a stateful operator across the new partitioned operators when it scales out.
- Partitioning the key space of the tuples processed by the operator.
- The routing state of its upstream operators must also be updated to account for the new partitioned operators.
- The buffer state of the upstream operators is partitioned to ensure that unprocessed tuples are dispatched to the correct partition.

Scale Out

- To scale out queries at runtime, the system partitions operators on-demand in response to bottleneck operators.
- The load of the bottlenecked operator is shared among a set of new partitioned operators.



Fault-Tolerance

- Overload and failure are handled in the same fashion.
- Operator recovery becomes a special case of scale out, in which a failed operator is scaled out.

Fault-Tolerant Scale Out Algorithm

- Two versions of operator's state that can be partitioned for scale out:
  - The current state
  - The recent state checkpoint

Fault-Tolerant Scale Out Algorithm

- Two versions of operator's state that can be partitioned for scale out:
  - The current state
  - The recent state checkpoint

- In SEEP, the system partitions the most recent state checkpoint.

Fault-Tolerant Scale Out Algorithm

- Two versions of operator's state that can be partitioned for scale out:
  - The current state
  - The recent state checkpoint

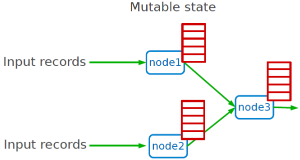
- In SEEP, the system partitions the most recent state checkpoint.

- Its benefits:
  - Avoids adding further load to the operator, which is already overloaded, by requesting it to checkpoint or partition its own state.
  - Makes the scale out process itself fault-tolerant.

Spark Stream

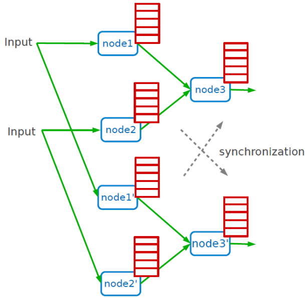
Existing Streaming Systems (1/2)

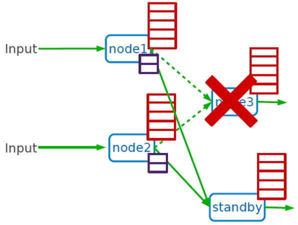
- Record-at-a-time processing model:
  - Each node has mutable state.
  - For each record, updates state and sends new records.
  - State is lost if node dies.



Existing Streaming Systems (2/2)

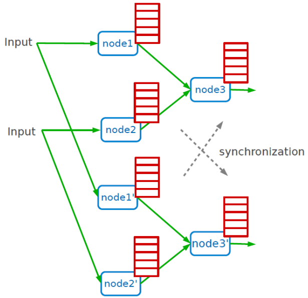
- Fault tolerance via replication or upstream backup.

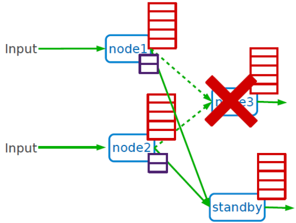




Existing Streaming Systems (2/2)

- Fault tolerance via replication or upstream backup.





Fast recovery, but 2x hardware cost

Only need one standby, but slow to recover

Observation

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

Core 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.

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.

Discretized Stream Processing (D-Stream)

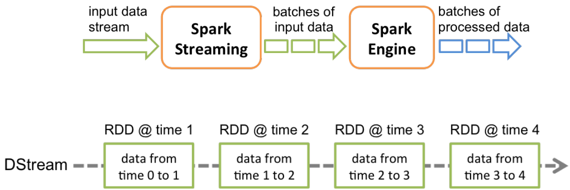
- Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.

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.

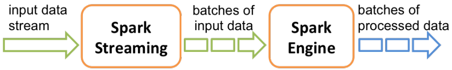
D-Stream API (1/4)

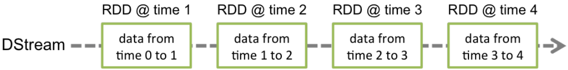
- DStream: sequence of RDDs representing a stream of data.
  - TCP sockets, Twitter, HDFS, Kafka, ...



D-Stream API (1/4)

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



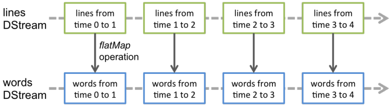


- Initializing Spark streaming

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

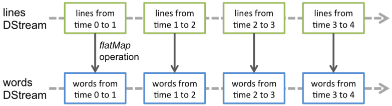
D-Stream API (2/4)

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

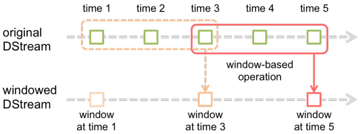


D-Stream API (2/4)

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



  - Window 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

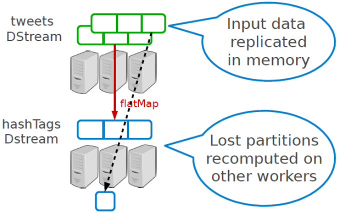
```
freqs.slice("21:00", "21:05").topK(10)
```

- 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.

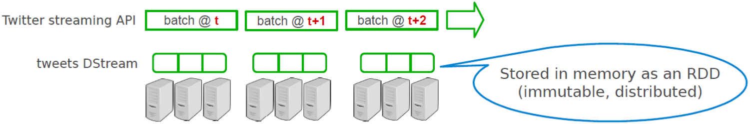


Example 1 (1/3)

- 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



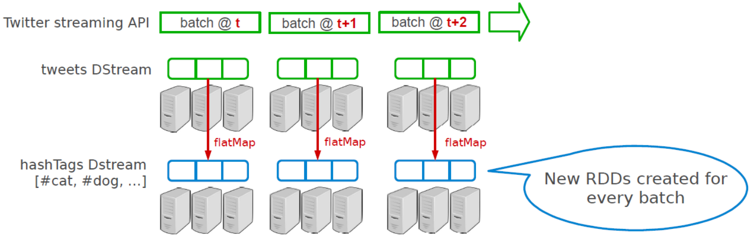
Example 1 (2/3)

- 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

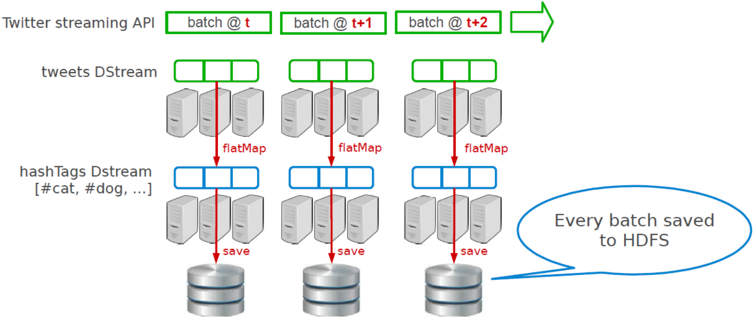
to create another DStream



Example 1 (3/3)

- Get hash-tags from Twitter.

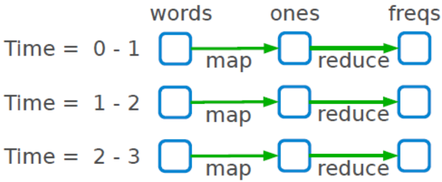
```
val ssc = new StreamingContext("local[2]", "test", Seconds(1))
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



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(_ + _)
```



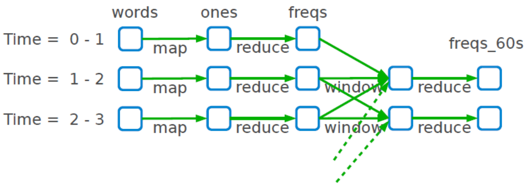
Example 3

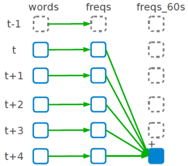
- Count frequency of words received in last minute.

```
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

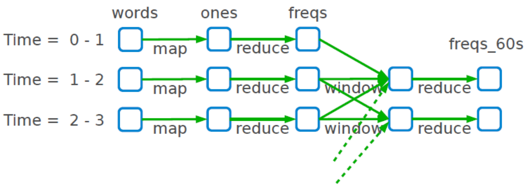


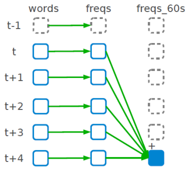


Example 3 - Simpler Model

- 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))
```



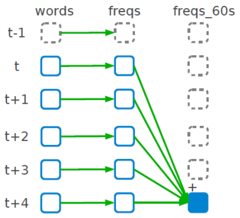


Example 3 - Incremental Window Operators

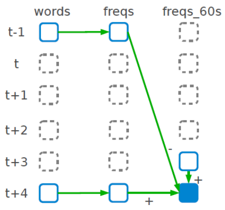
- Count frequency of words received in last minute.

```
// 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

- SEEP
  - Make operator state an external entity
  - Primitives for state management: checkpoint, backup/restore, partition

- Spark Stream
  - Run a streaming computation as a series of very small, deterministic batch jobs.
  - DStream: sequence of RDDs
  - Operators: Transformations (stateless, stateful, and window) and output operations

Questions?

Acknowledgements

Some slides and pictures were derived from Matei Zaharia (MIT University) and Peter Pietzuch (Imperial College) slides.