Going Deep with Spark Streaming

Andrew Psaltis (@itmdata)
Berlin Buzzwords, June 2, 2015

shutterstock

Outline

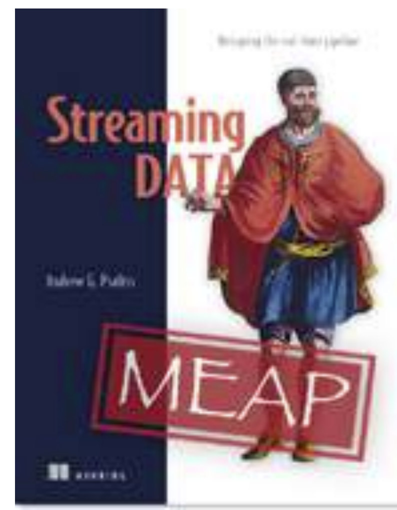
- Introduction
- DStreams
- Thinking about time
- Recovery and Fault tolerance
- Conclusion

shutterstock

About Me

Andrew Psaltis

Data Engineer @ Shutterstock



Fun outside of Shutterstock:

- Sometimes ramble here: @itmdata
- Author of Streaming Data
- Dreaming about streaming since 2008
- Conference Speaker
- Content provider for SkillSoft
- Lacrosse crazed

shutterstock

Introduction

Why Streaming?

"Without stream processing there's no big data and no Internet of Things" – Dana Sandu, SQLstream

Why Streaming?

- **Operational Efficiency** - 1 extra mph for a locomotive on it's daily route can lead to $200M in saving (Norfolk Southern)

- **Improving Traffic Safety and Efficiency** – According to EU Commission congestion in EU urban areas costs ~ €100 billion or 1 percent of EU GDP annually

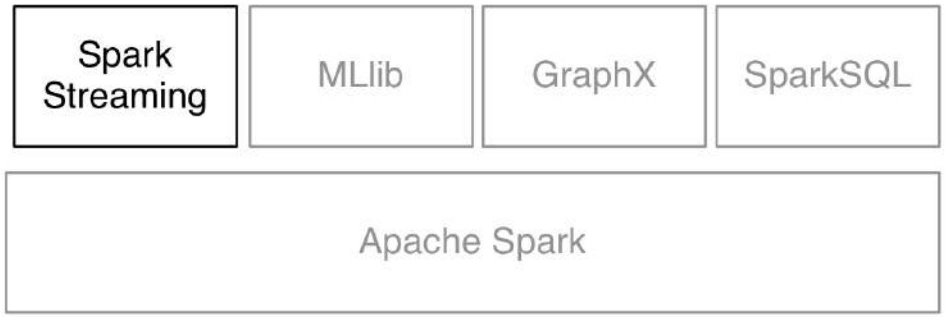
shutterstock

Our shared problem

Today if a byte of data was 1 gallon of water we could fill an average house in 10 seconds, by 2020 it will take only 2.

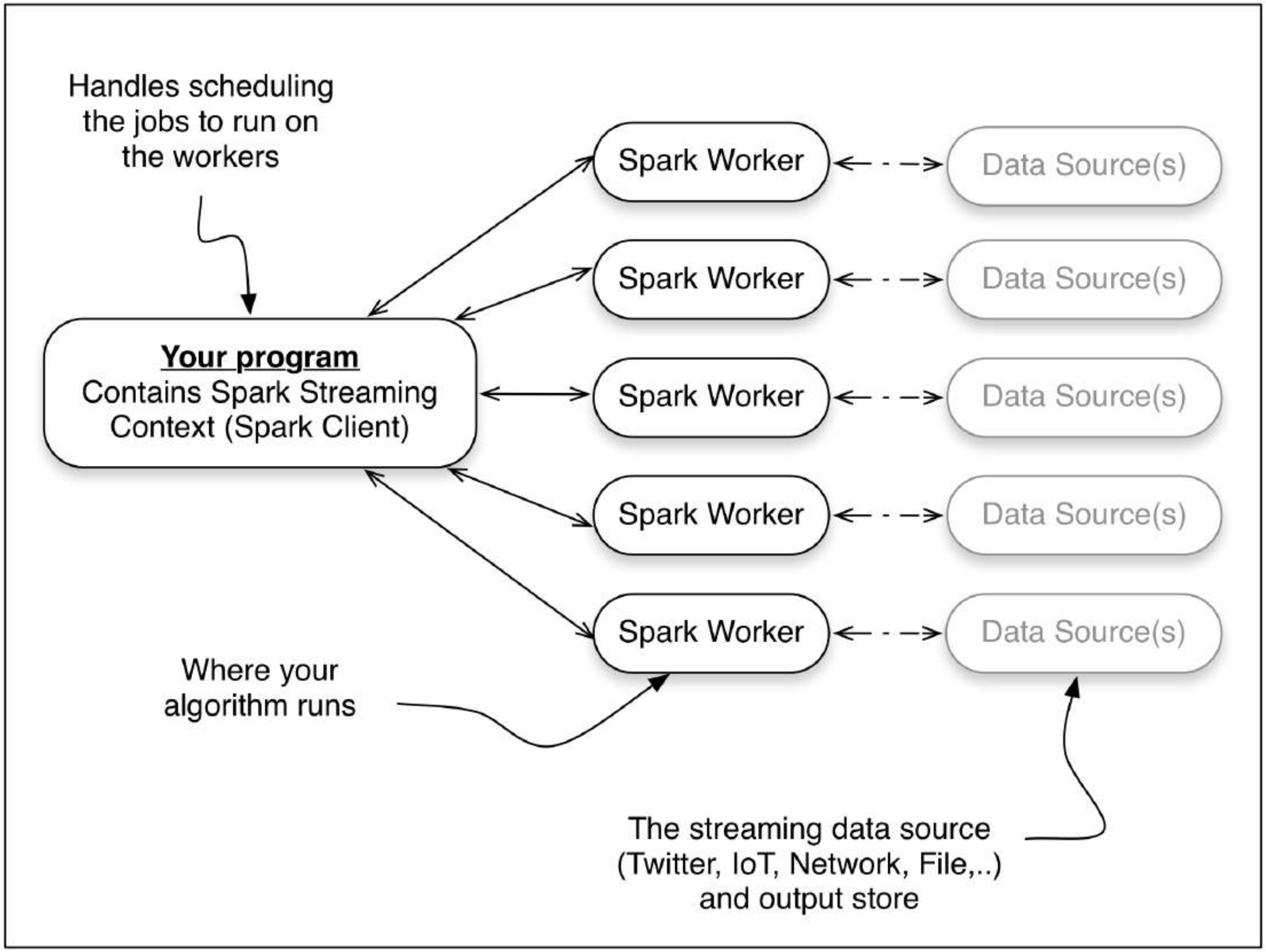
shutterstock

What is Spark Streaming?

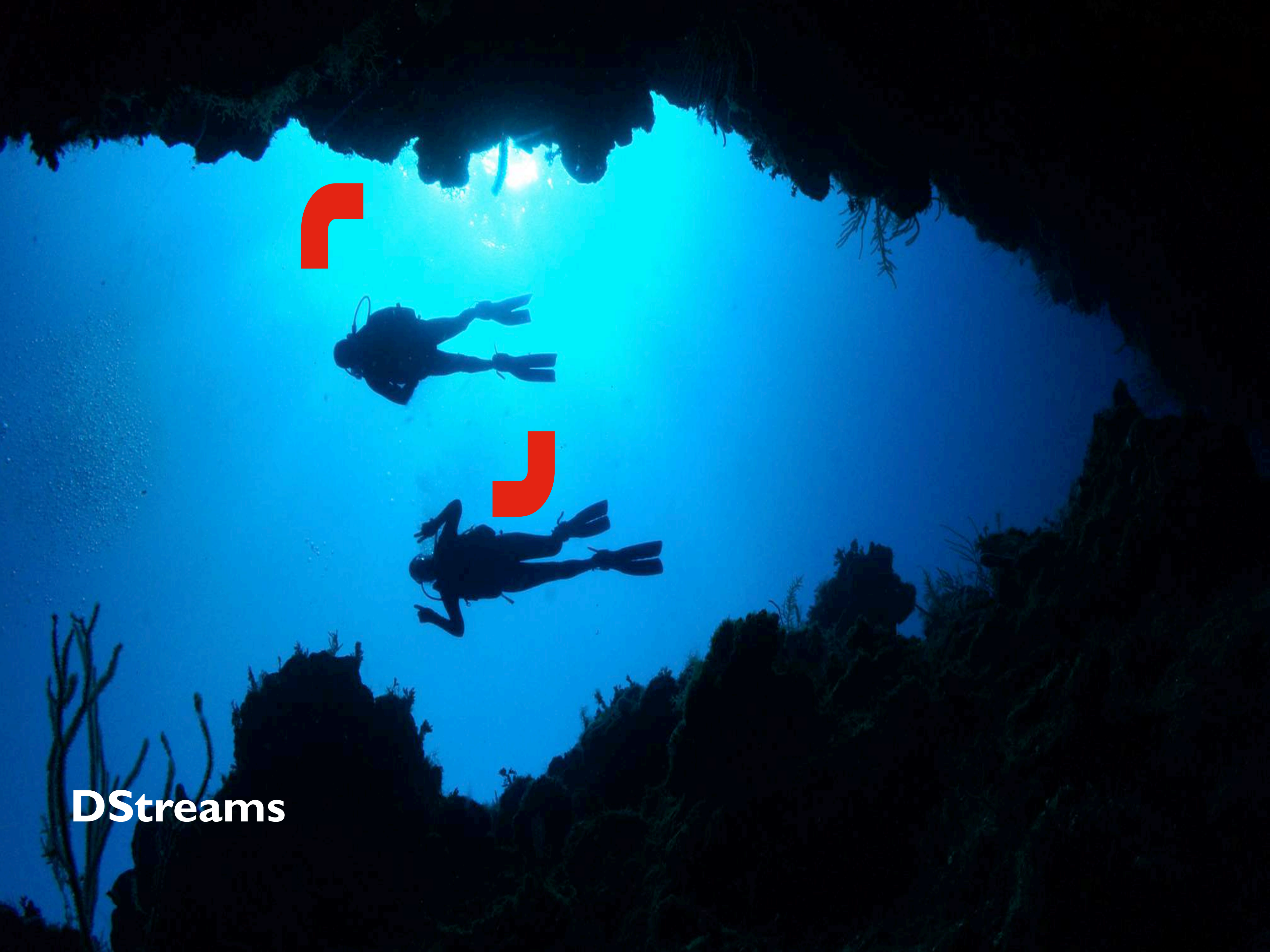


- Provides efficient, fault-tolerant stateful stream processing
- Provides a simple API for implementing complex algorithms
- Integrates with Spark's batch and interactive processing
- Integrates with other Spark extensions

High-level Architecture

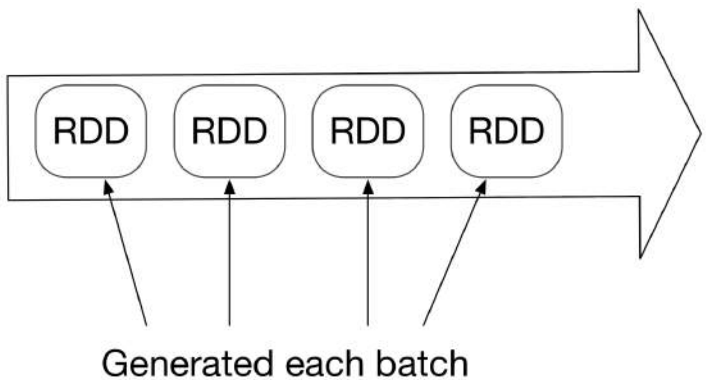


shutterstock


DStreams

Discretized Streams (DStreams)

- The basic abstraction provided by Spark Streaming
- Continuous series of RDDs



DStreams

- 3 Things we want to do
  - Ingest
  - Transform
  - Output

Input DStreams (Ingestion)

There are 3 ways to get data in:

- Basic sources
- Advanced sources
- Custom Sources

shutterstock

Basic Input DStreams

- Basic sources
  - Built-in (file system, socket, Akka actors)
  - Non-built in (Avro, CSV, …)
  - Not reliable

Advanced Input DStreams

- Advanced sources
  - Twitter, Kafka, Flume, Kinesis, MQTT, ….
  - Require external library
  - Maybe reliable or unreliable

Custom Input DStreams

- Implement two classes
  - InputDStream
  - Receiver

shutterstock

Custom Input DStream

Returns the receiver
that is sent to workers

```
class CustomInputDStream(
    @transient ssc_ : StreamingContext,
    storageLevel: StorageLevel
  ) extends ReceiverInputDStream[String](ssc_) {

  def getReceiver(): Receiver[String] = {
    new CustomReceiver(storageLevel)
  }
}
```

Custom Receiver

Start threads, open sockets, etc..
MUST BE non-blocking

```
class CustomReceiver(storageLevel: StorageLevel)
  extends Receiver[String](storageLevel){

  def onStart() {
  }

  def onStop() {
  }

  //Defined in Receiver class
  def store(….) {
  }
}
```

Cleanup everything started in onStart.
Stops receiving data

Call *store* (item, buffer, iterator)

Receiver Reliability

Two types of receivers

- Unreliable Receiver
- Reliable Receiver

shutterstock

Receiver Reliability

Unreliable Receiver

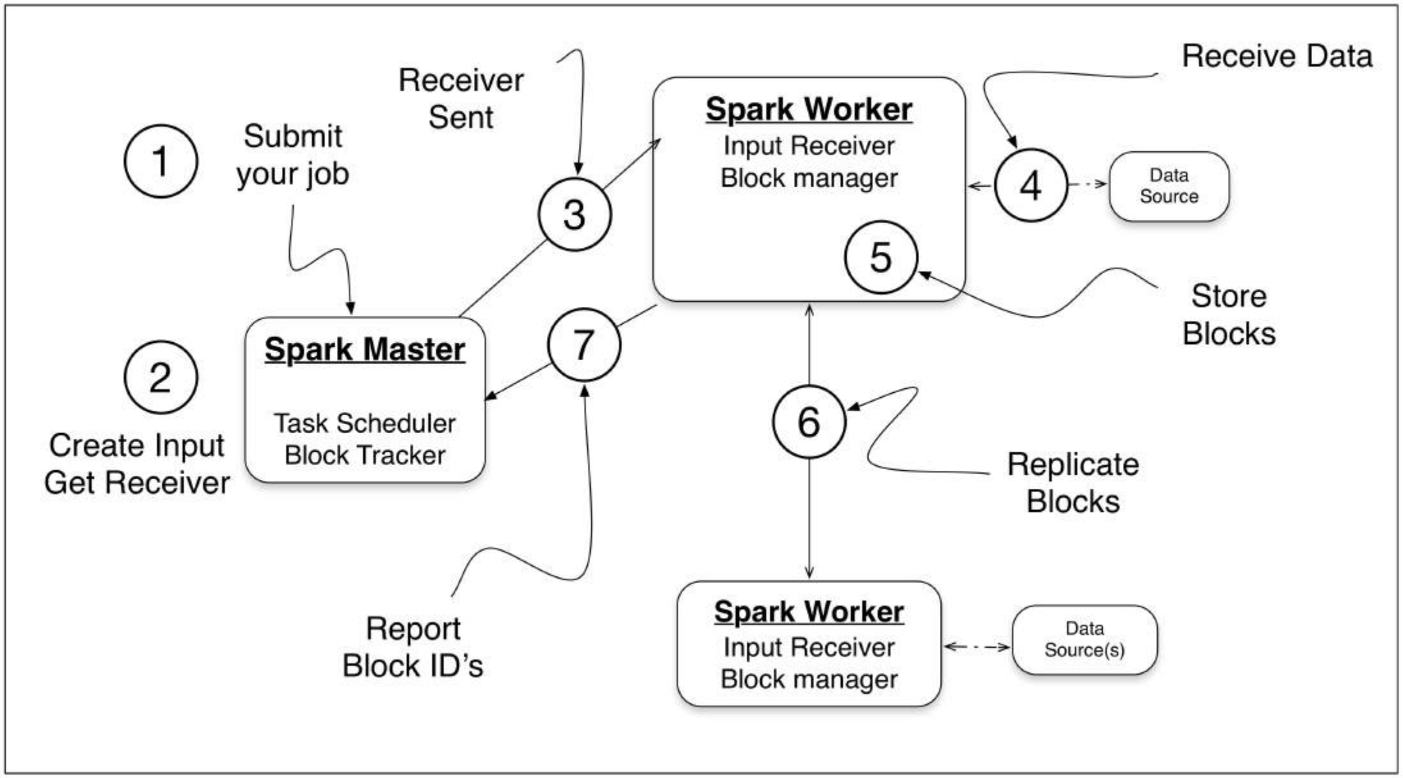
- Simple to implement
- No fault-tolerance
- Data loss when receiver fails

Receiver Reliability

Reliable Receiver

- Complexity depends on the source
- Strong fault-tolerance guarantees (zero data loss)
- Data source must support acknowledgement

Input DStream and Receiver



shutterstock

Creating DStreams

2 Ways to create a DStream

- Input – a streaming source
- Transforming a DStream

shutterstock

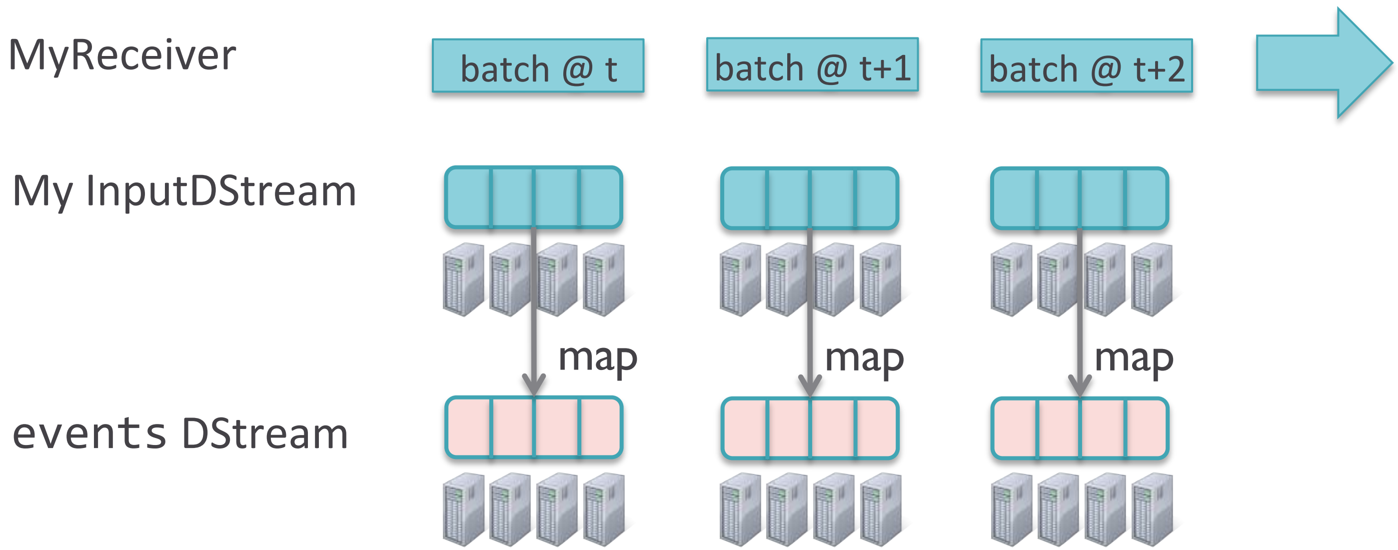
Creating a DStream via Transformation

- Transformations modify data from one DStream to another

- Two general classifications:
    - Standard RDD operations – map, countByValue, reduceByKey, join,…
    - Stateful operations – window, updateStateByKey, transform, countByValueAndWindow, …

Transforming the input - Standard Operation

```
val myStream = createCustomStream(streamingContext)
val events = myStream.map(….)
```



shutterstock

Stateful Operation - UpdateStateByKey

Provides a way for you to maintain arbitrary state while continuously updating it.

- For example – In-Session Advertising, Tracking twitter sentiment

Stateful Operation - UpdateStateByKey

Need to do two things to leverage it:

- Define the state – this can be any arbitrary data
- Define the update function – this needs to know how to update the state using the previous state and new values

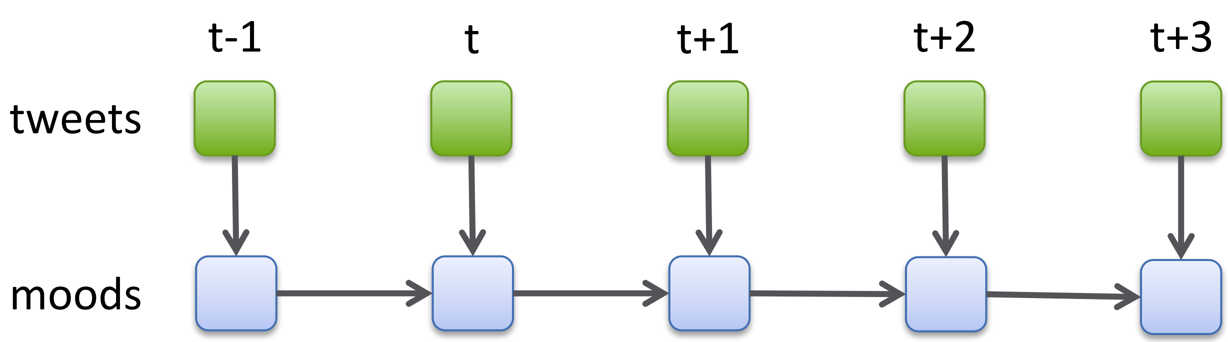
Requires Checkpoint to be configured

shutterstock

Using updateStateByKey

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

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



Transform

Allows arbitrary RDD-to-RDD functions to be applied on a DStream

```
transform (transformFunc: RDD[T] => RDD[U]): DStream[U]
```

Example: We want to eliminate "noise" words from crawled documents:

```
val noiseWordRDD = ssc.sparkContext.newAPIHadoopRDD(...)
val cleanedDStream = crawledCorpus.transform(rdd => {
  rdd.join(noiseWordRDD).filter(...)})
```

shutterstock

Joining streams

Allows you to combine two DStreams that share a key and produce a new DStream

```
join(other: DStream(K,V)): DStream[K,(V,W)]
```

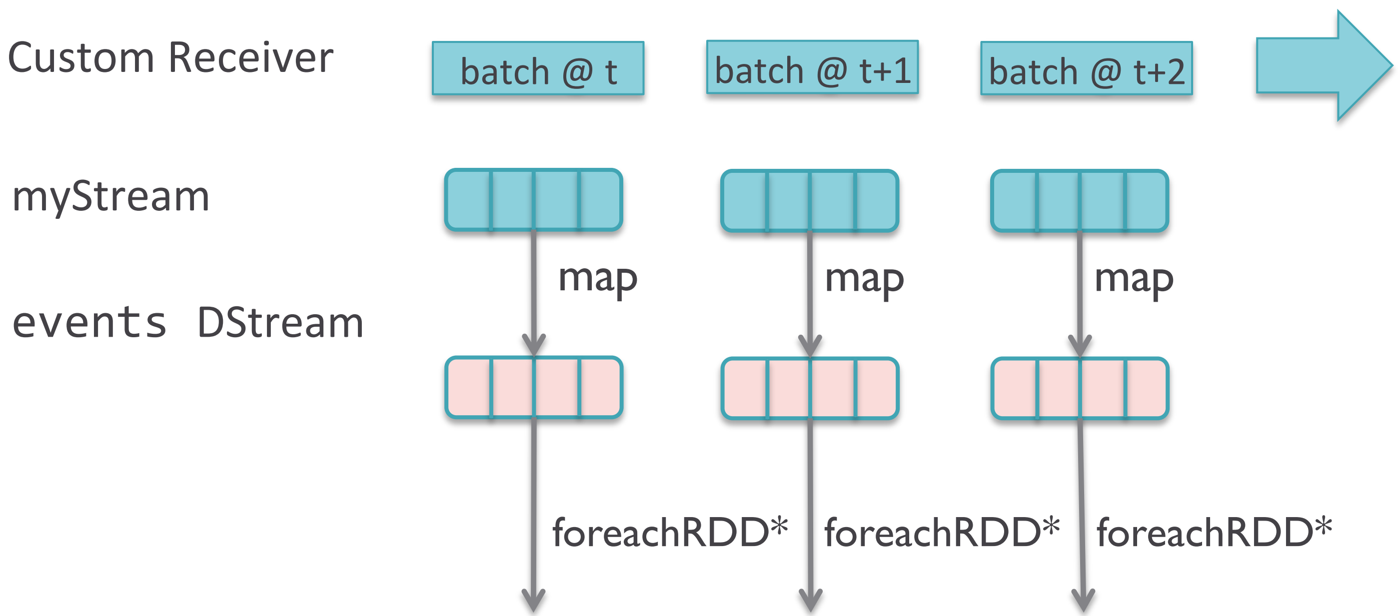
Example: We want to group Fitbit and MapMyRun streams

```
val musicBits = fitBitStream.join(mapMyRunStream)
```

shutterstock

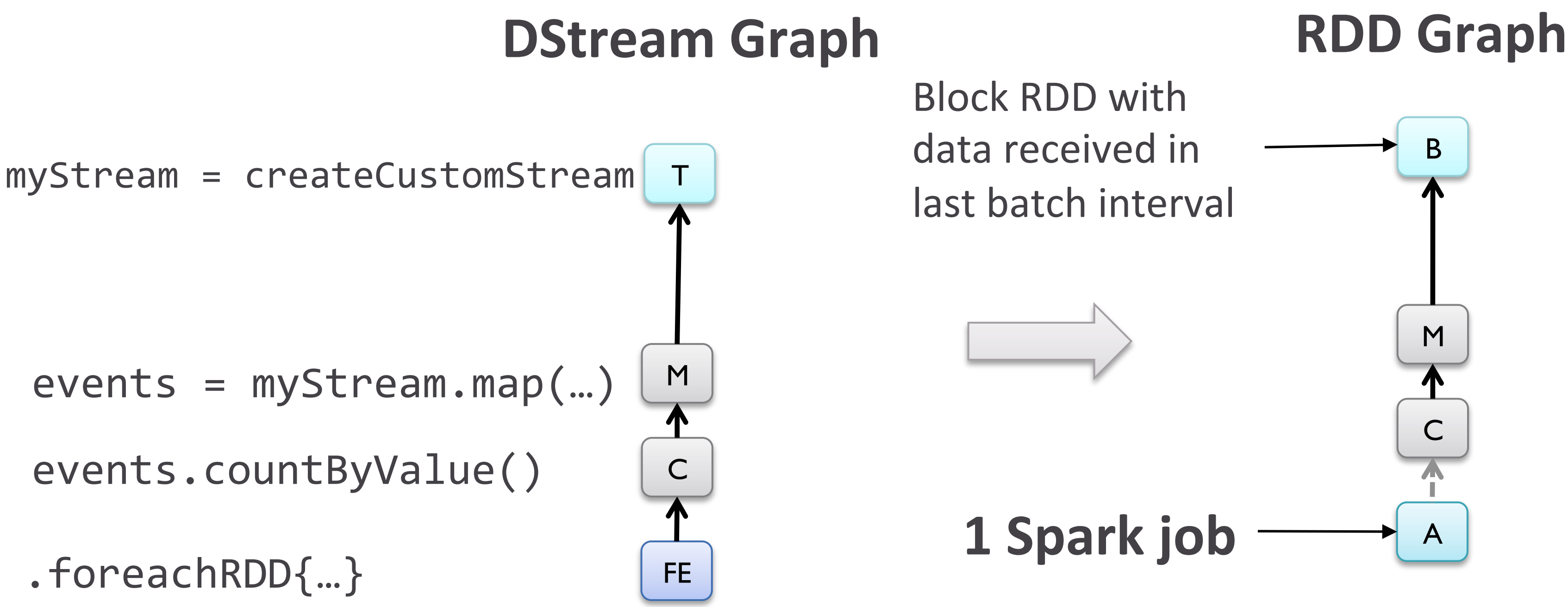
Outputting data

```
val myStream = createCustomStream(streamingContext)
val events = myStream.map(….)
events.countByValue().foreachRDD{…}
```



shutterstock

From Streaming Program to Spark jobs




Thinking about time

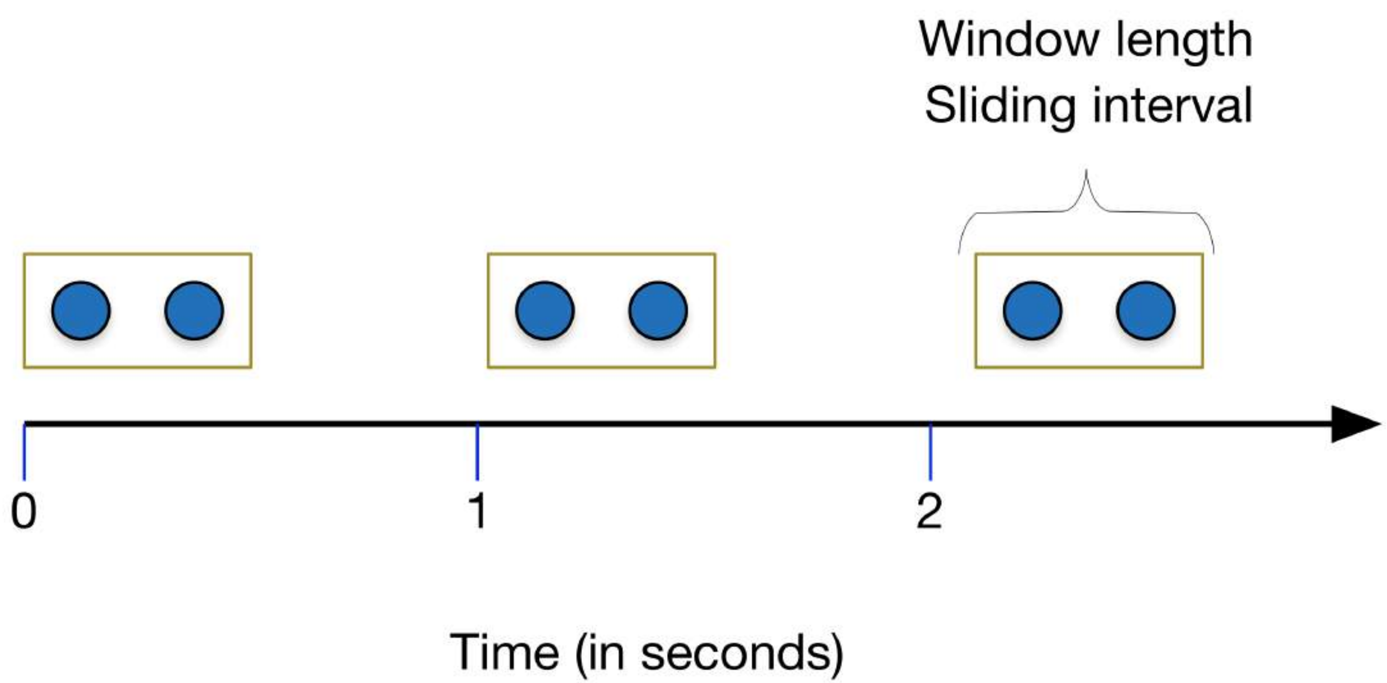
Thinking about time

- Windowing – Tumbling, Sliding
- Stream time vs. Event time
- Out of order data

Windowing

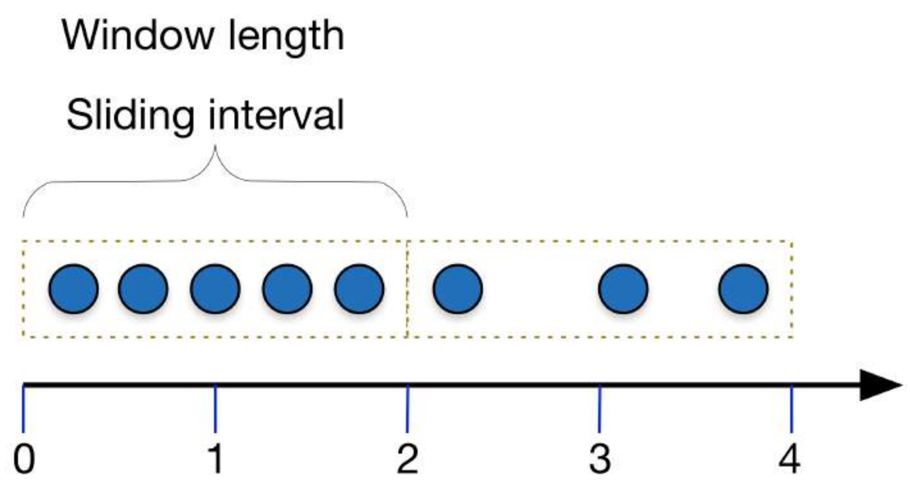
- Common Types
  - Tumbling
  - Sliding

Tumbling (Count) Windowing



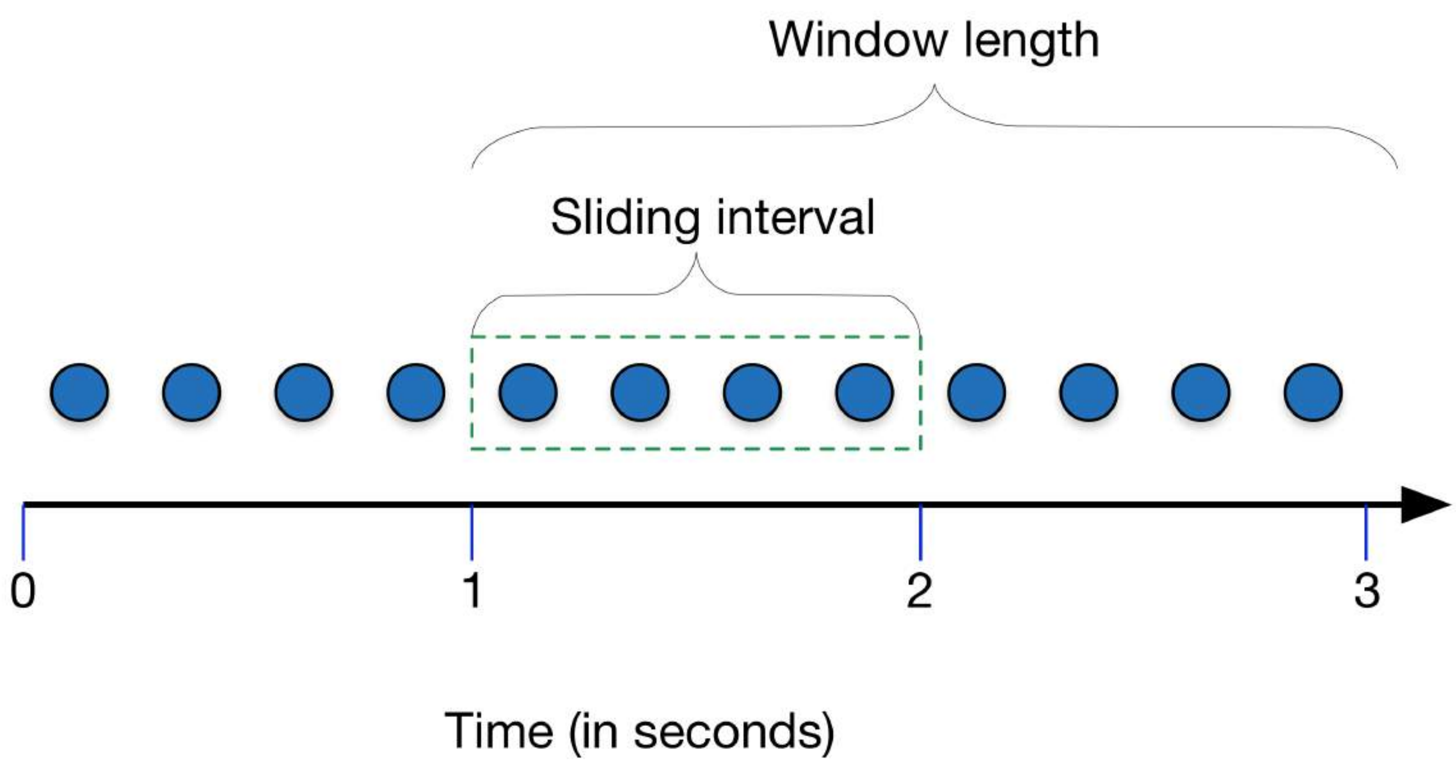
Tumbling count-window

Tumbling (temporal) Windowing



Tumbling temporal window

Sliding Window

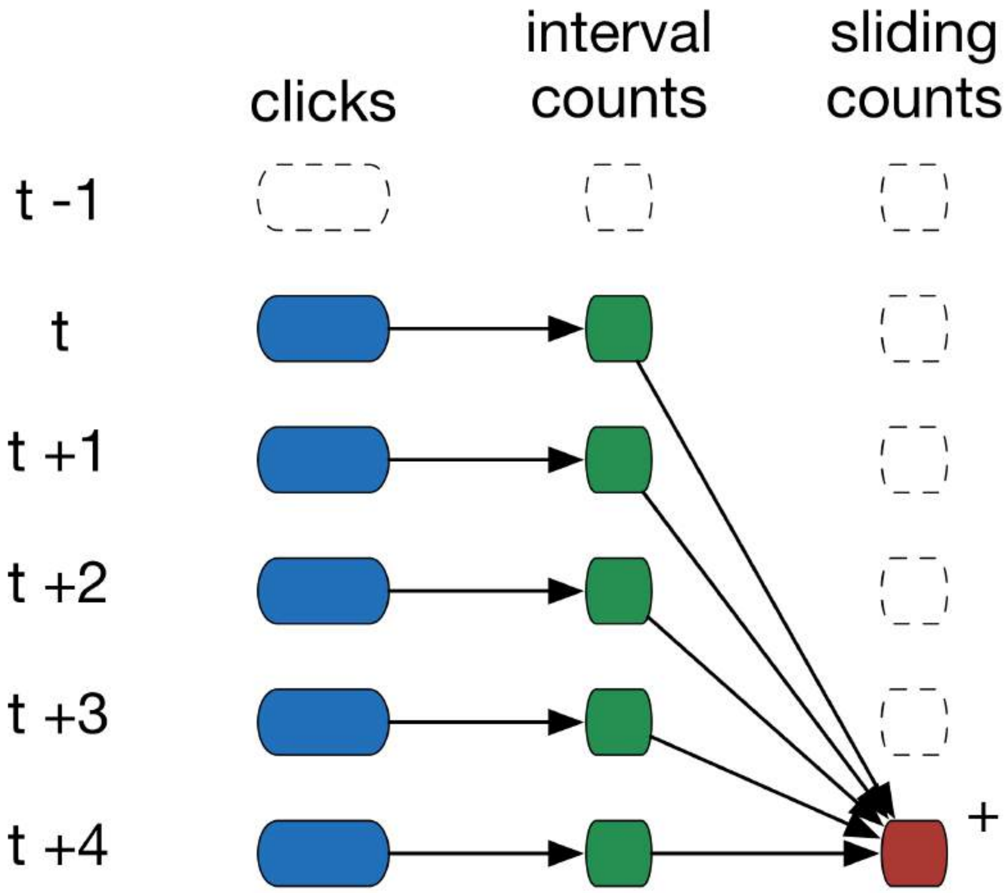


Spark Streaming -- Sliding Windowing

- Two types supported:
  - Non-Incremental
  - Incremental

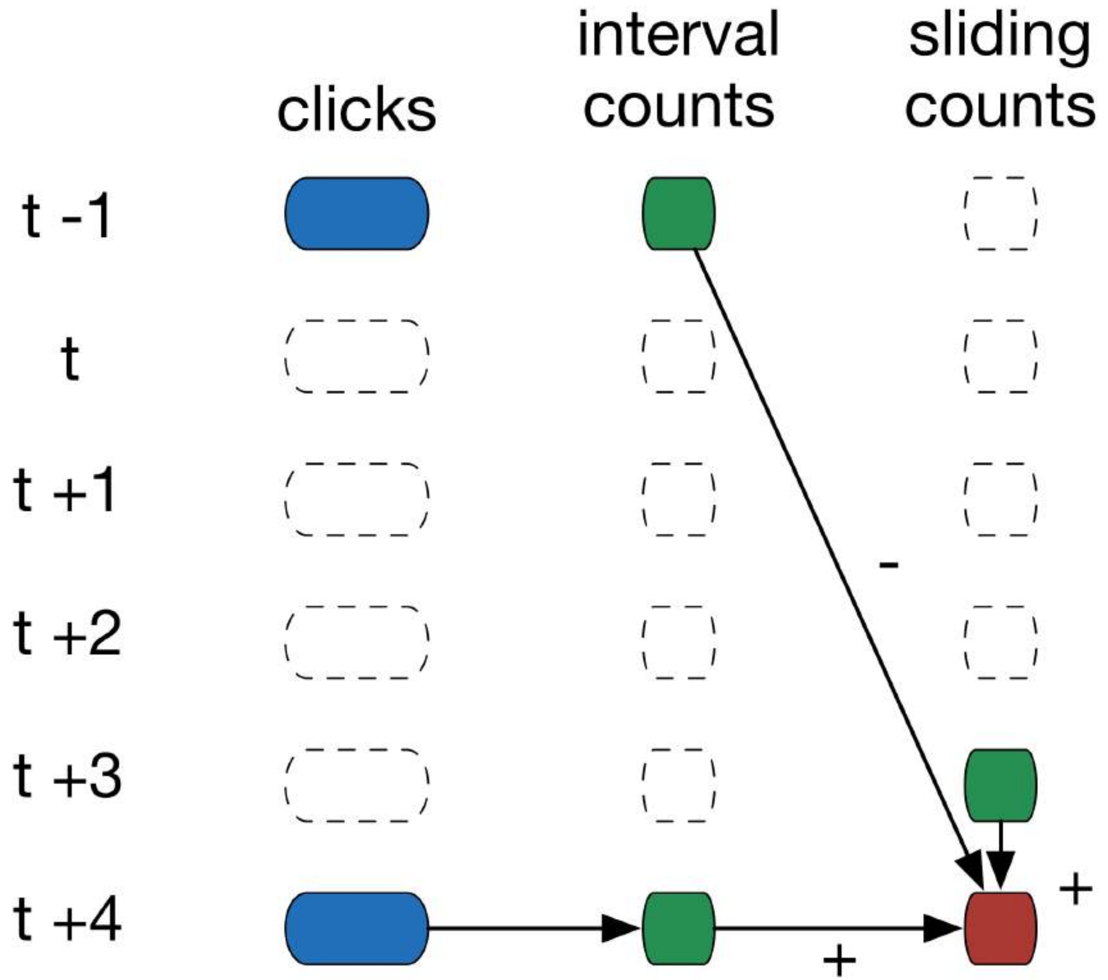
Non-Incremental Sliding Windowing

```
reduceByKeyAndWindow((a,b)=>(a + b),Seconds(5), Seconds(1))
```



Incremental Sliding Windowing

```
reduceByKeyAndWindow((a,b) => (a + b), (a,b) => (a-b),
Seconds(5), Seconds(1))
```



More thinking about time

Stream time vs. Event time

- **Stream time --** the time when the record arrives into the streaming system.
- **Event time** – the time that the event was generated, **not** when it entered the system.
- Spark Streaming uses stream time

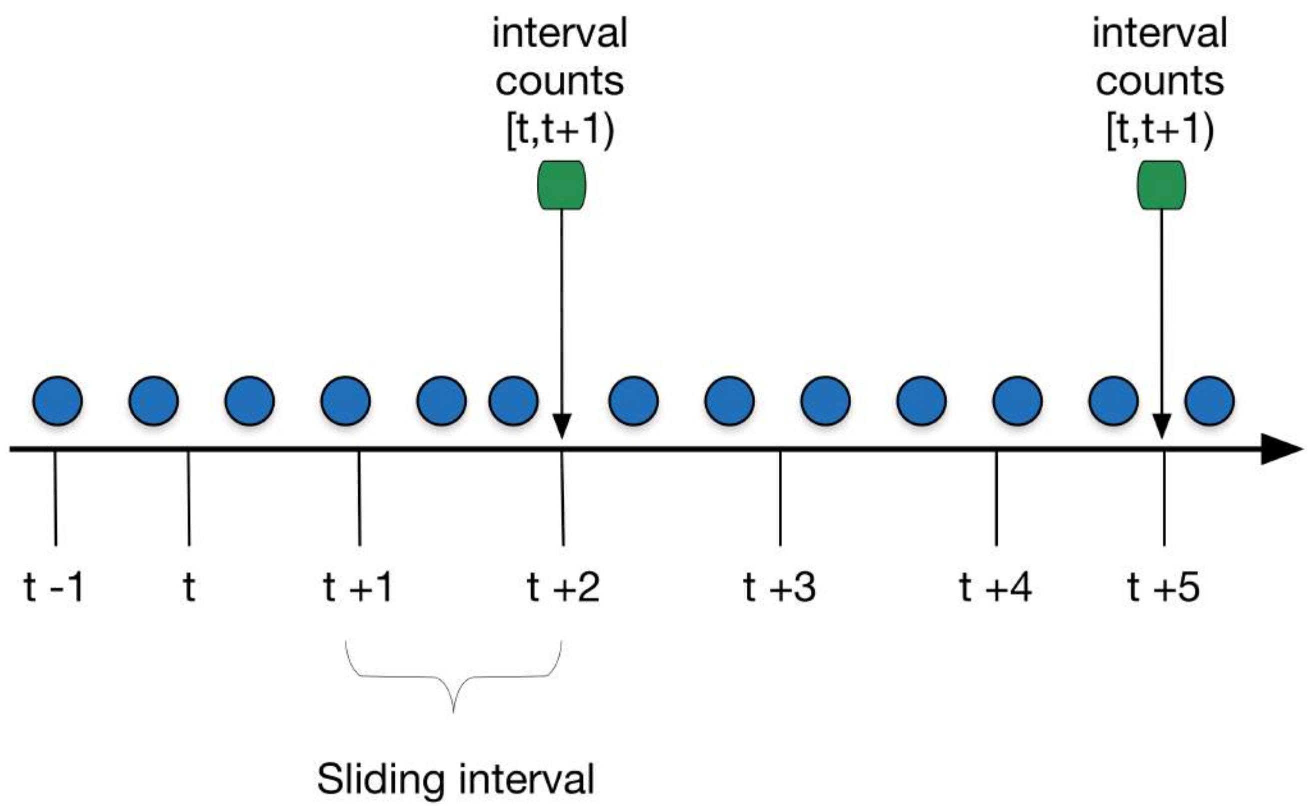
Out of order data

- Does it matter to your application?
- How do you deal with it?

shutterstock

Handling Out of Order Data

Imagine we want to track ad impressions between time *t* and *t + l*

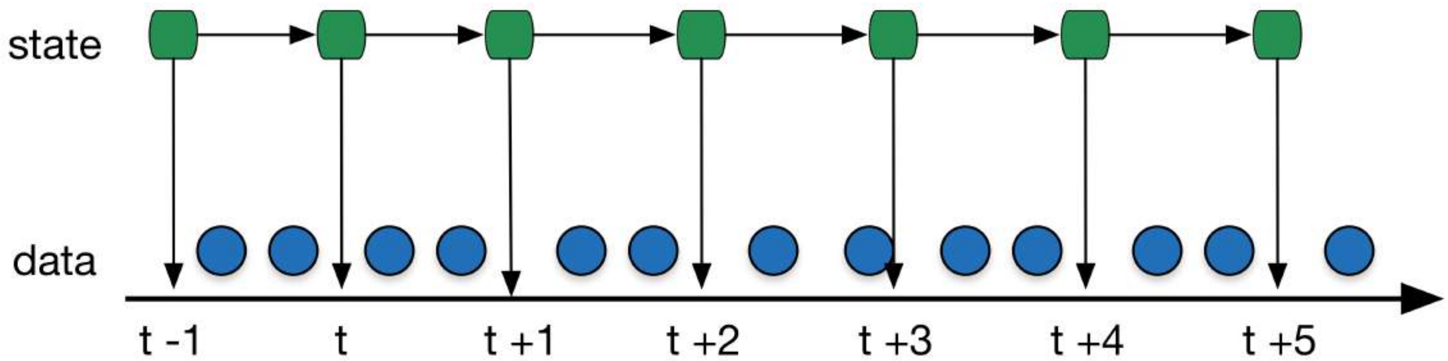


Continuous Analytics Over Discontinuous Streams
http://www.eecs.berkeley.edu/~franklin/Papers/sigmod10krishnamurthy.pdf

Recovery and Fault Tolerance

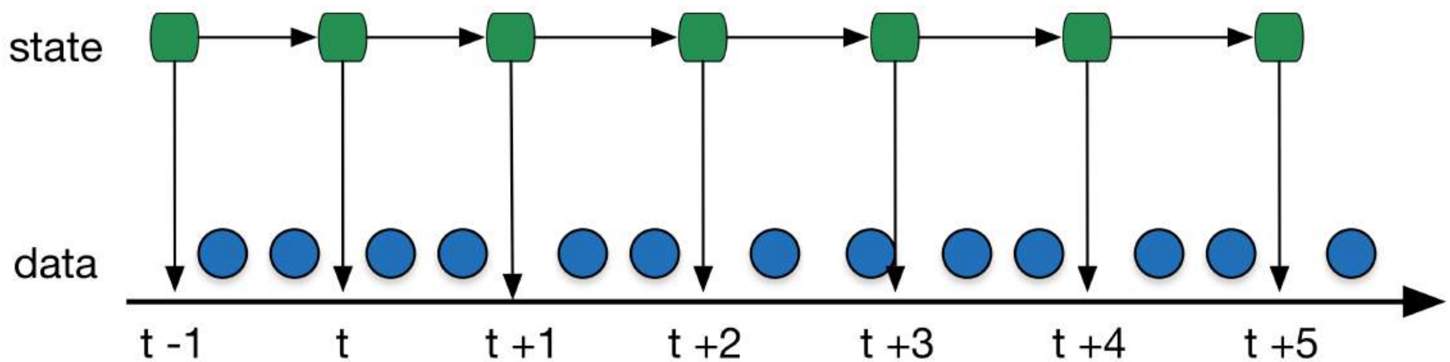
Recovery

- Checkpointing
  - Metadata checkpointing
  - Data checkpointing

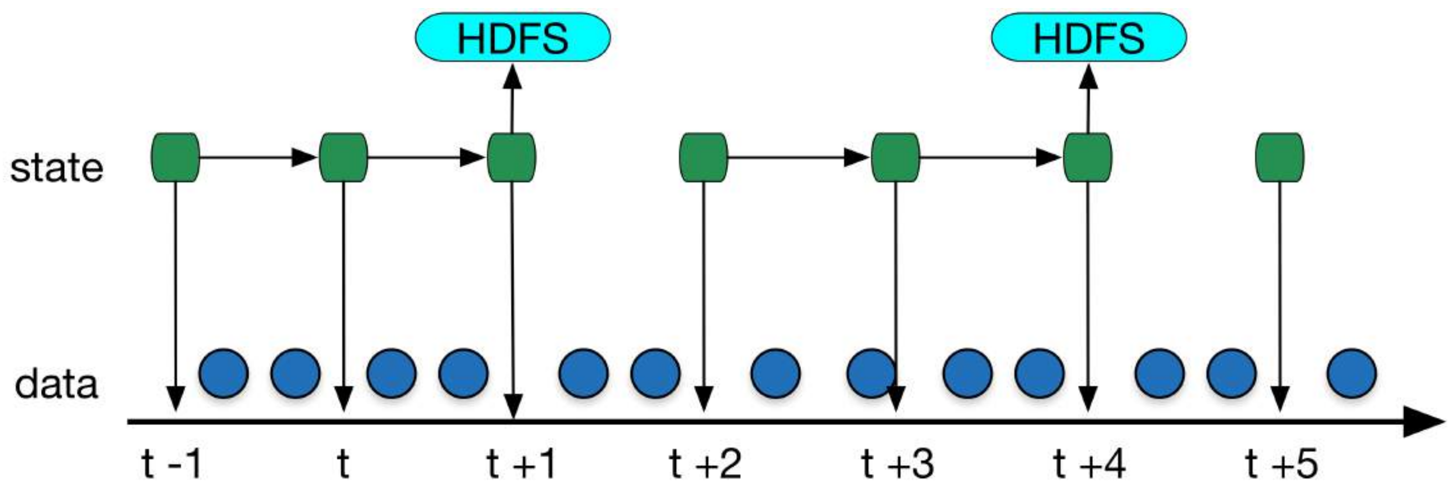


shutterstock

Recovery



Without



With

Recovery

- Too frequent: HDFS writing will slow things down
- Too infrequent: Lineage and task sizes grow
- Default setting: Multiple of batch interval at least 10 seconds
- **Recommendation:** checkpoint interval of 5 - 10 times of sliding interval

shutterstock

Fault Tolerance

- All properties of RDDs still apply
- We are trying to protect two things
  - Failure of a Worker
  - Failure of the Driver Node
- Semantics
  - At most once
  - At least once
  - Exactly once
- Where we need to think about it
  - Receivers
  - Transformations
  - Output

Conclusion

- Introduction
- High-level Architecture
- DStreams
- Thinking about time
- Recovery and Fault tolerance

Thank yu

Andrew Psaltis
@itmdata
psaltis.andrew@gmail.com
https://www.linkedin.com/in/andrewpsaltis