Productionalizing Spark Streaming

Spark Summit 2013

Ryan Weald
@rweald

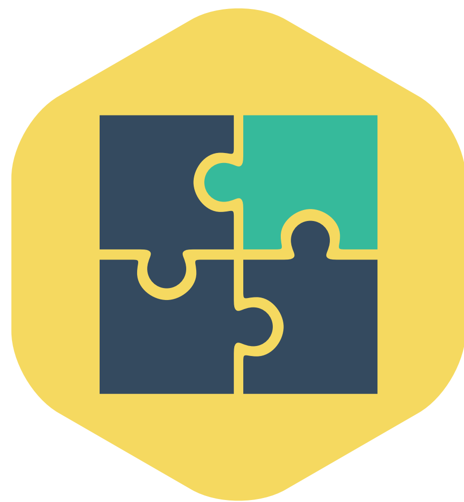
What We're Going to Cover

- **What** we do and **Why** we choose Spark
- Fault tolerance for long lived streaming jobs
- Common patterns and functional abstractions
- Testing before we "do it live"

Special focus on common patterns and their solutions

What is Sharethrough?

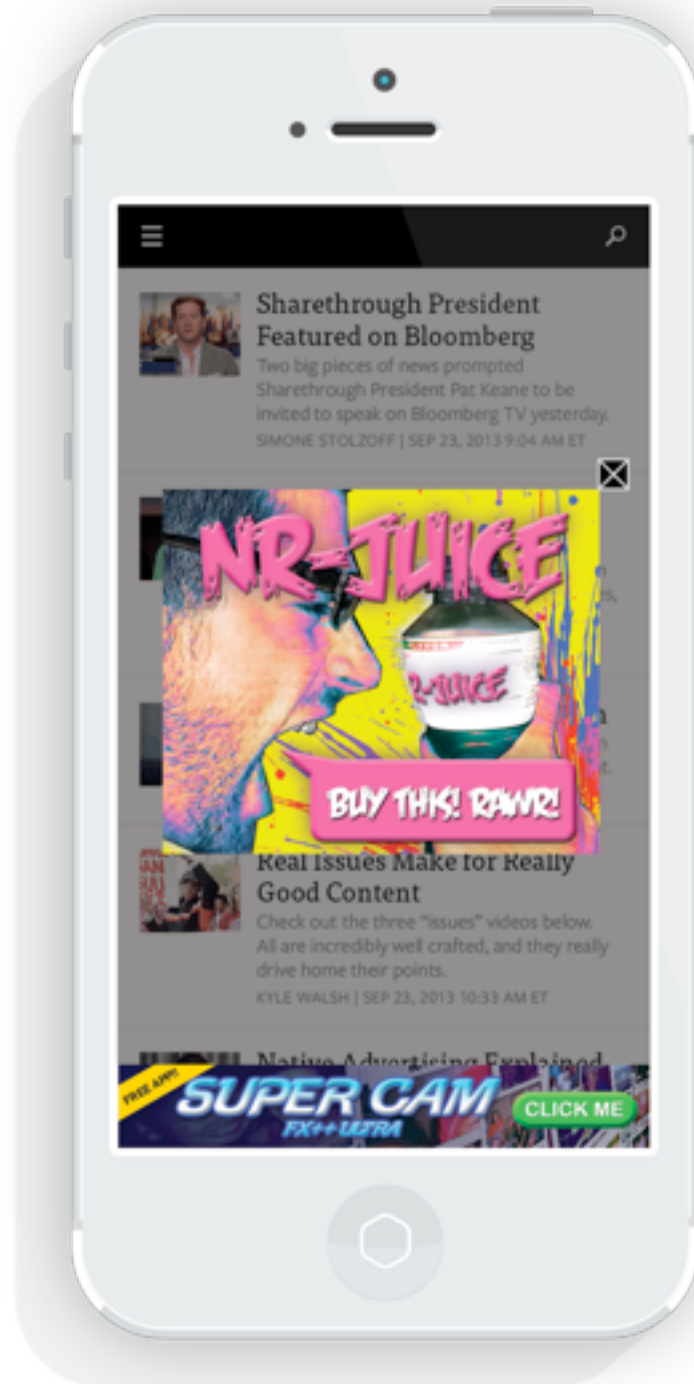
Advertising for the Modern Internet

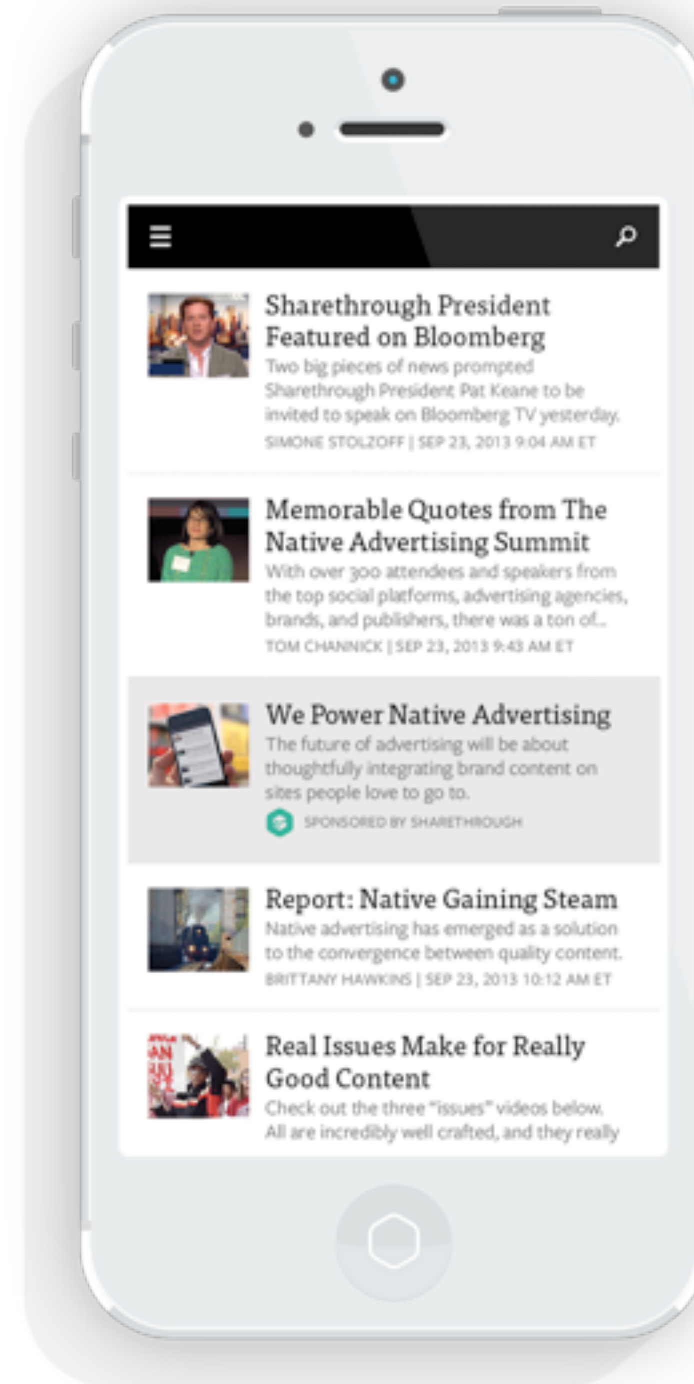


Form

Function

What is Sharethrough?





Why Spark Streaming?

Why Spark Streaming

- Liked theoretical foundation of mini-batch
- Scala codebase + functional API
- Young project with opportunities to contribute
- Batch model for iterative ML algorithms

Great…
Now productionalize it

Fault Tolerance

Keys to Fault Tolerance

1. Receiver fault tolerance

2. Monitoring job progress

Receiver Fault Tolerance

- Use Actors with supervisors
- Use self healing connection pools

Use Actors

```
class RabbitMQStreamReceiver (uri:String, exchangeName: String,
routingKey: String) extends Actor with Receiver with Logging {

  implicit val system = ActorSystem()
  override def preStart() = {
    //Your code to setup connections and actors

    //Include inner class to process messages
  }

  def receive: Receive = {
    case _      => logInfo("unknown message")
  }
}
```

Track All Outputs

- Low watermarks - Google MillWheel
- Database updated_at
- Expected output file size alerting

Common Patterns & Functional Programming

Common Job Pattern

Map -> Aggregate ->Store

Mapping Data

```
inputData.map { rawRequest =>
  val params = QueryParams.parse(rawRequest)
  (params.getOrElse("beaconType", "unknown"), 1L)
}
```

Aggregation

Basic Aggregation

```
//beacons is DStream[String, Long]
//example Seq(("click", 1L), ("click", 1L))
val sum: (Long, Long) => Long = _ + _
beacons.reduceByKey(sum)
```

What Happens when we want to sum multiple things?

Long Basic Aggregation

```
val inputData = Seq(
  ("user_1",(1L, 1L, 1L)),
  ("user_1",(2L, 2L, 2L))
)
def sum(l: (Long, Long, Long),
        r: (Long, Long, Long)) = {
  (l._1 + r._1, l._2 + r._2, l._3 + r._3)
}
inputData.reduceByKey(sum)
```

Now Sum 4 Ints instead

(ノಠ益ಠ)ノ ┻━━┻

Monoids to the Rescue

WTF is a Monoid?

```
trait Monoid[T] {
  def zero: T
  def plus(r: T, l: T): T
}
```

* Just need to make sure plus is associative.

(1+ 5) + 2 == (2 + 1) + 5

Monoid Based Aggregation

```
object LongMonoid extends Monoid[(Long, Long, Long)]
{
  def zero = (0, 0, 0)
  def plus(r: (Long, Long, Long),
           l: (Long, Long, Long)) = {
    (l._1 + r._1, l._2 + r._2, l._3 + r._3)
  }
}

inputData.reduceByKey(LongMonid.plus(_, _))
```

Twitter Algebird

http://github.com/twitter/algebird

Algebird Based Aggregation

```
import com.twitter.algebird._
val aggregator = implicitly[Monoid[(Long,Long, Long)]]

inputData.reduceByKey(aggregator.plus(_, _))
```

How many unique users per publisher?

Too big for memory based naive Map

HyperLogLog FTW

HLL Aggregation

```
import com.twitter.algebird._
val aggregator = new HyperLogLogMonoid(12)
inputData.reduceByKey(aggregator.plus(_, _))
```

Monoids == Reusable Aggregation

Common Job Pattern

Map -> Aggregate ->Store

Store

How do we store the results?

Storage API Requirements

- Incremental updates (preferably associative)
- Pluggable to support “big data” stores
- Allow for testing jobs

Storage API

```scala
trait MergeableStore[K, V] {
  def get(key: K): V
  def put(kv: (K,V)): V
  /*
   * Should follow same associative property
   * as our Monoid from earlier
   */
  def merge(kv: (K,V)): V
}
```

Twitter Storehaus

http://github.com/twitter/storehaus

Storing Spark Results

```
def saveResults(result: DStream[String, Long],
  store: RedisStore[String, Long]) = {
    result.foreach { rdd =>
      rdd.foreach { element =>
        val (keys, value) = element
        store.merge(keys, impressions)
      }
    }
}
```

Everyone can benefit

Potential API additions?

```
class PairDStreamFunctions[K, V] {
  def aggregateByKey(aggregator: Monoid[V])
  def store(store: MergeableStore[K, V])
}
```

Twitter Summingbird

http://github.com/twitter/summingbird

*https://github.com/twitter/summingbird/issues/387

Testing Your Jobs

Testing best Practices

- Try and avoid full integration tests
- Use in-memory stores for testing
- Keep logic outside of Spark
- Use Summingbird in memory platform???

Thank You

Ryan Weald
@rweald