Real-time Map Reduce: Exploring Clickstream Analytics with: Kafka, Spark Streaming and WebSockets

Andrew Psaltis

About Me

- Recently started working at Ensighten on Agile Maketing Platform
- Prior 4.5 years worked Webtrends on Streaming and Realtime Visitor Analytics – where I first fell in love with Spark

Where are we going?

- Why Spark and Spark Streaming or How I fell for Spark.
- Give a brief overview of architecture in mind
- Give birds-eye view of Kafka
- Discuss Spark Streaming
- Walk through some Click Stream examples
- Discuss getting data out of Spark Streaming

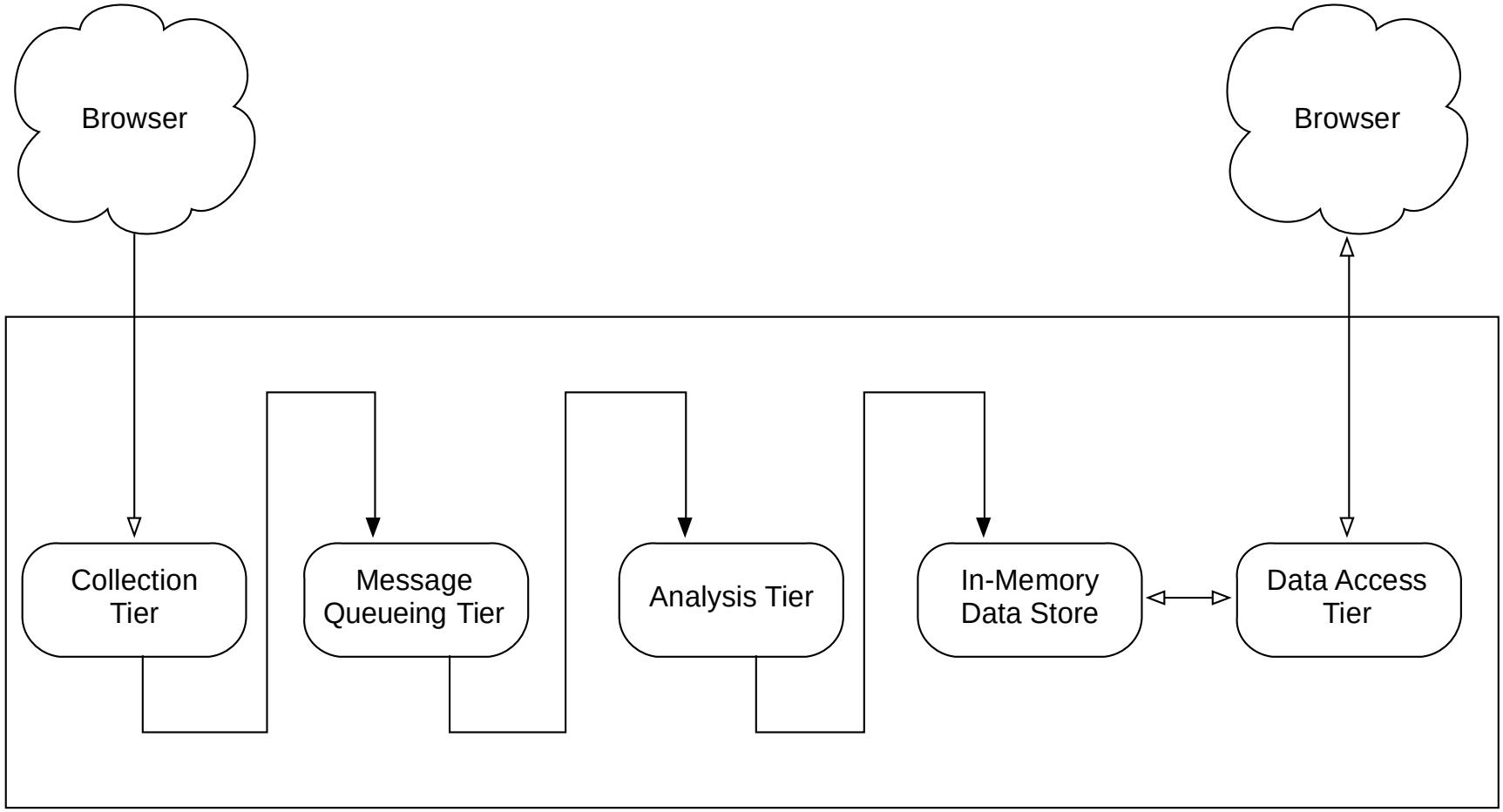
How I fell for Spark

- On Oct 14th/15th 2012 three worlds collided:
  - Felix Baumgartner jumped from Space (Oct 14, 2012)
  - Viewing of jump resulted in the single largest hour in 15 year history -- Analytics engines crashed analyzing data.
  - Spark Standalone Announced – no longer requiring Mesos.
- Quickly tested Spark and it was love at first sight.

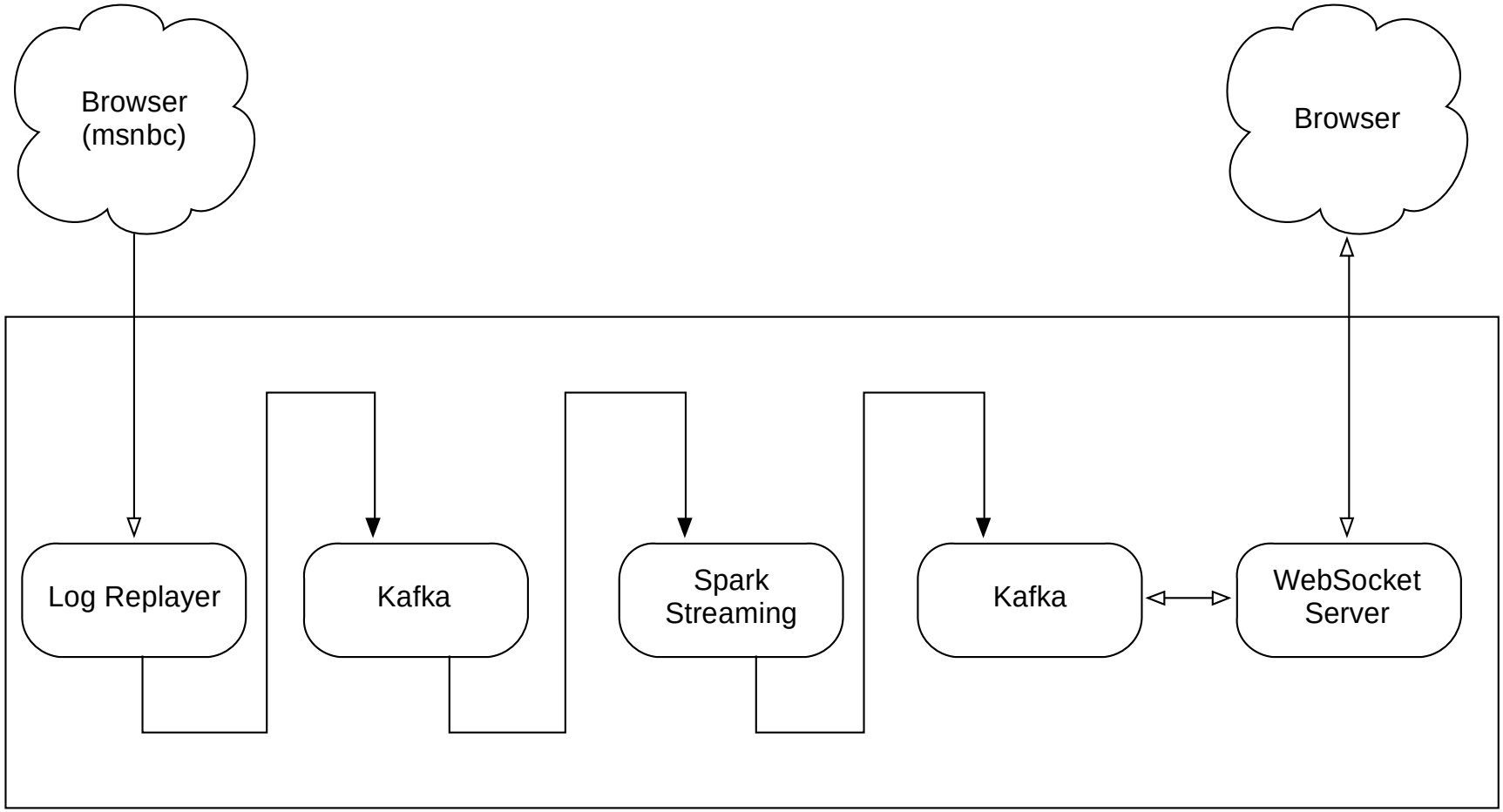
Why Spark Streaming?

- Already had Storm running in production providing an event analytics stream
- Wanted to deliver an aggregate analytics stream
- Wanted exactly-once semantics
- OK with second-scale latency
- Wanted state management for computations
- Wanted to combine with Spark RDD’s

Generic Streaming Data Pipeline



Demo Streaming Data Pipeline

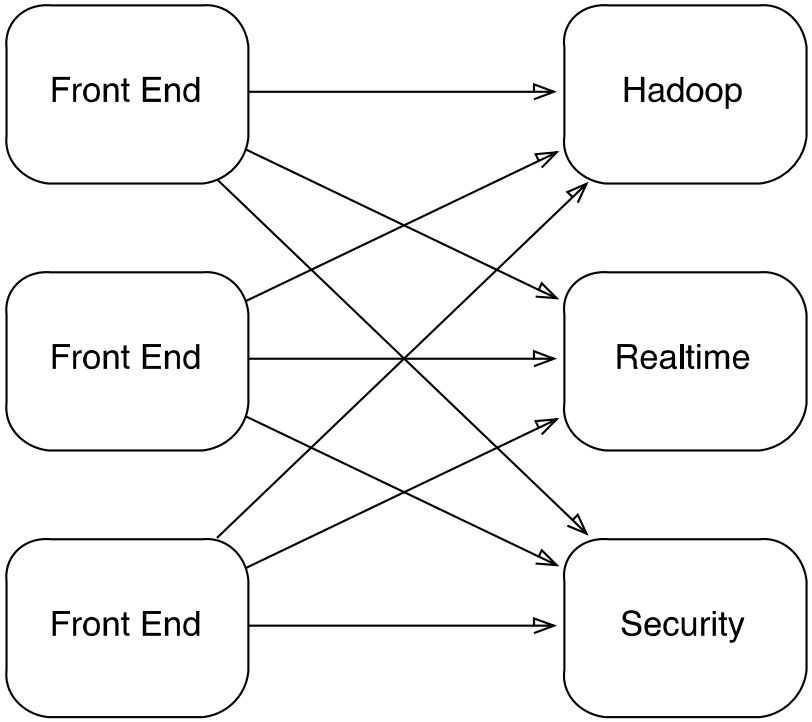


Apache Kafka

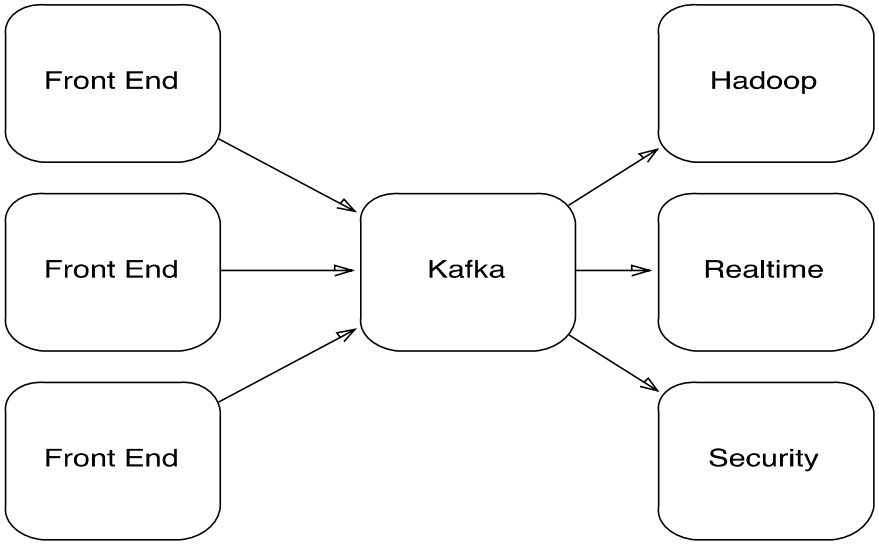
- Overview
  - An Apache project initially developed at LinkedIn
  - Distributed publish-subscribe messaging system
  - Specifically designed for real time activity streams
  - Does not follow JMS Standards nor uses JMS APIs

- Key Features
  - Persistent messaging
  - High throughput, low overhead
  - Uses ZooKeeper for forming a cluster of nodes
  - Supports both queue and topic semantics

Kafka decouples data-pipelines



Kafka decouples data-pipelines



What is Spark Streaming?

- Extends Spark for doing large scale stream processing
- Efficient and fault-tolerant stateful stream processing
- Integrates with Spark’s batch and interactive processing
- Provides a simple batch-like API for implementing complex algorithms

Programming Model

- A Discretized Stream or **DStream** is a series of RDDs representing a stream of data
  - API *very similar* to RDDs
- Input - DStreams can be created...
  - Either from live streaming data
  - Or by transforming other Dstreams
- Operations
  - Transformations
  - Output Operations

Input -DStream Data Sources

- Many sources out of the box
  - HDFS
  - Kafka
  - Flume
  - Twitter
  - TCP sockets
  - Akka actor
  - ZeroMQ
- Easy to add your own

Operations - Transformations

Allows you to build new streams from existing streams

- RDD-like operations
- **map**, flatMap, filter, **countByValue,** reduce,
- groupByKey, **reduceByKey**, **sortByKey**, join
- etc.
- Window and stateful operations
- window, countByWindow, reduceByWindow
- **countByValueAndWindow**, reduceByKeyAndWindow
- **updateStateByKey**
- etc.

Operations - Output Operations

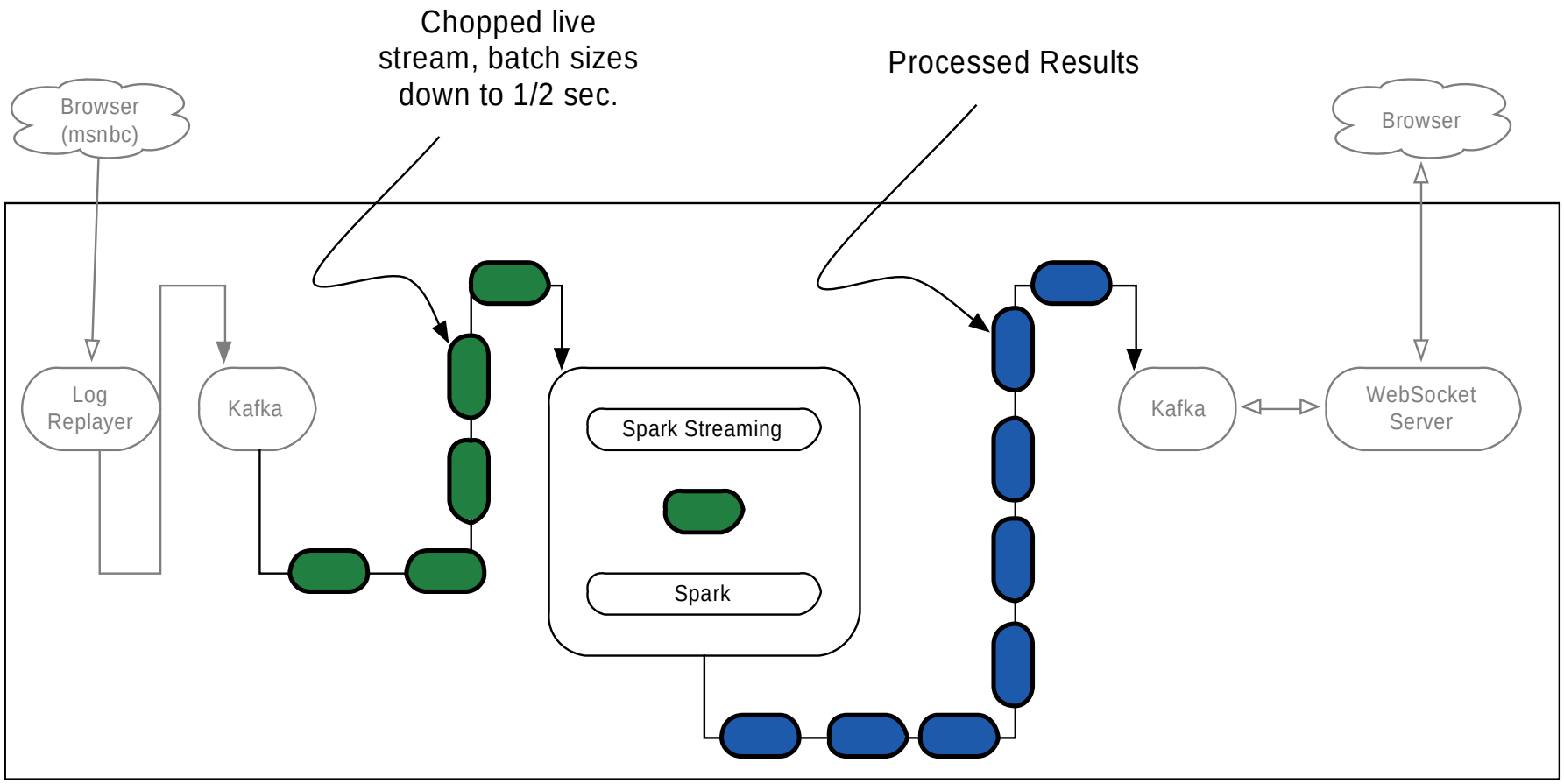
Your way to send data to the outside world.

Out of the box support for:

- print - prints on the driver's screen
- **foreachRDD** - arbitrary operation on every RDD
- saveAsObjectFiles
- saveAsTextFiles
- saveAsHadoopFiles

Discretized Stream Processing

n a streaming computation as a series of very small, deterministic batch jobs



Clickstream Examples

- PageViews per batch
- PageViews by Url over time
- Top N PageViews over time
- Keeping a current session up to date
- Joining the current session with historical

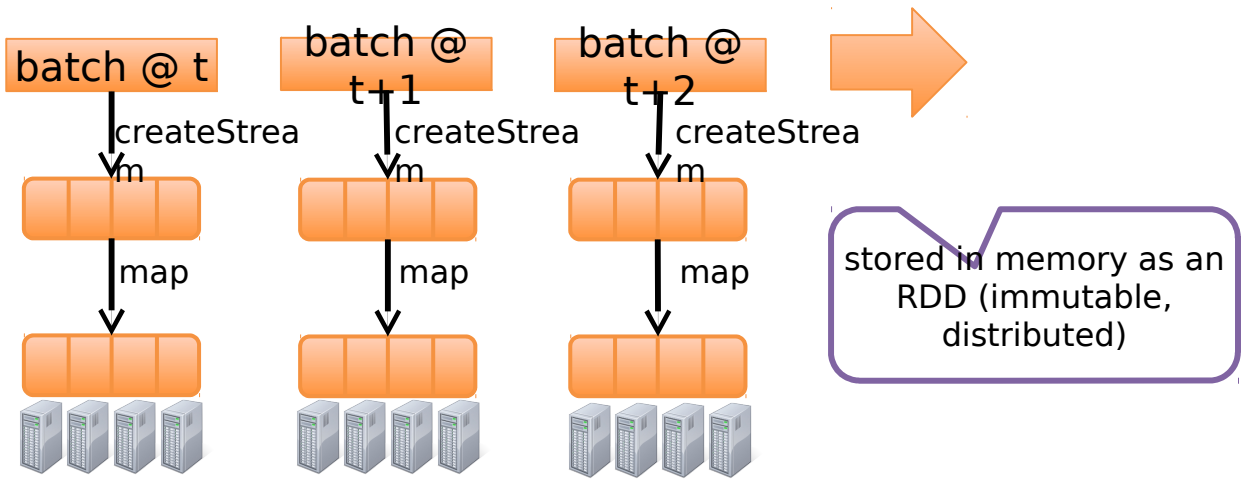
Example – Create Stream from Kafka

```
JavaPairDStream<String,String> messages = KafkaUtils.createStream(…);

JavaDStream<Tuple2<String,String>> events = messages.map(new
Function<Tuple2<String,String>,Tuple2<String,String>>(){

        @Override

        public Tuple2<String,String> call(Tuple2<String,String>
tuple2){

            String parts[]= tuple2._2().split("\t");

            return new Tuple2<>(parts[0],parts[1]);

                }});
```

Kafka Consumer

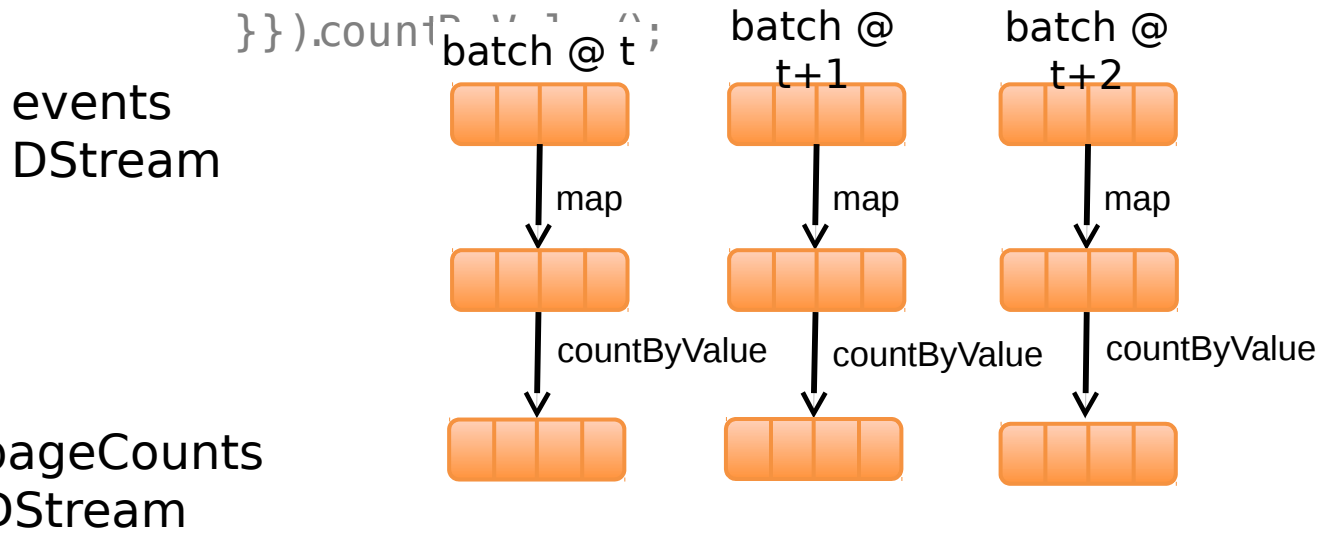
messages DStream

events DStream



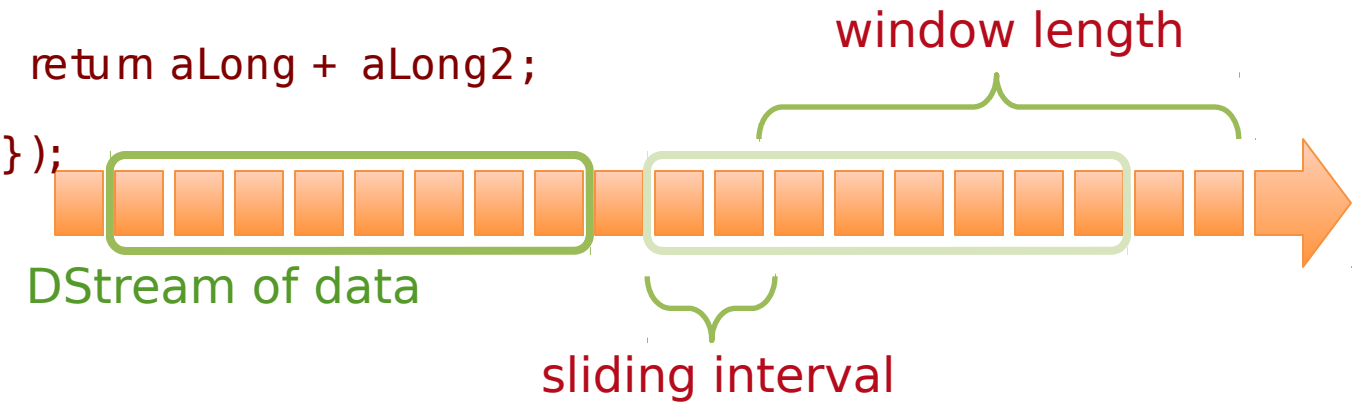
Example – PageViews per batch

```
JavaPairDStream<String,Long> pageCounts = events.map(new
Function<Tuple2<String,String>,String>(){

    @Override

    public String call(Tuple2<String,String> pageView){

      return pageView._2();

    }}).countByValue();
```



Example – PageViews per URL over time Window-based Transformations

```
JavaPairDStream<String,Long> slidingPageCounts = events.map(new
Function<Tuple2<String,String>,String>(){

        @Override public String call(Tuple2<String,String>
pageView){

            return pageView._2();

        }}).countByValueAndWindow(new Duration(30000),new
Duration(5000)).reduceByKey(new Function2<Long,Long,Long>(){

        @Override public Long call(Long aLong,Long aLong2){

           return aLong + aLong2;

        }});
```



Example – Top N PageViews

```
JavaPairDStream<Long,String> swappedCounts = slidingPageCounts.map(
  new PairFunction<Tuple2<String,Long>,Long,String>(){
   public Tuple2<Long,String> call(Tuple2<String,Long> in){
       return in.swap();
}});


JavaPairDStream<Long,String> sortedCounts =   swappedCounts.transform(new
Function<JavaPairRDD<Long,String>,JavaPairRDD<Long,String>>(){
 public JavaPairRDD<Long,String> call(JavaPairRDD<Long,String> in){
              return in.sortByKey(false);
}});
```

Example – Updating Current Session

- Specify function to generate new state based on previous state and new data

```
Function2<List<PageView>,Optional<PageView>,Optional<Session>> updateFunction = new
Function2<List<PageView>,Optional<PageView>,     Optional<Session>>(){

   @Override public Optional<Session> call(List<PageView> values,        Optional<Session>
state){

          Session updatedSession = ... //update the session

                return Optional.of(updatedSession)

    }

 }
```

```
JavaPairDStream<String,Session> currentSessions =
pageView.updateStateByKey(updateFunction);
```

Example – Join current session with history

JavaPairDStream<String, Session> currentSessions = ...

JavaPairDStream<String, Session> historicalSessions = <RDD from Spark>

currentSessions looks like ----

Tuple2<String, Sesssion>("visitorId-1", "{Current-Session}")

Tuple2<String, Sesssion>("visitorId-2", "{Current-Session}"))

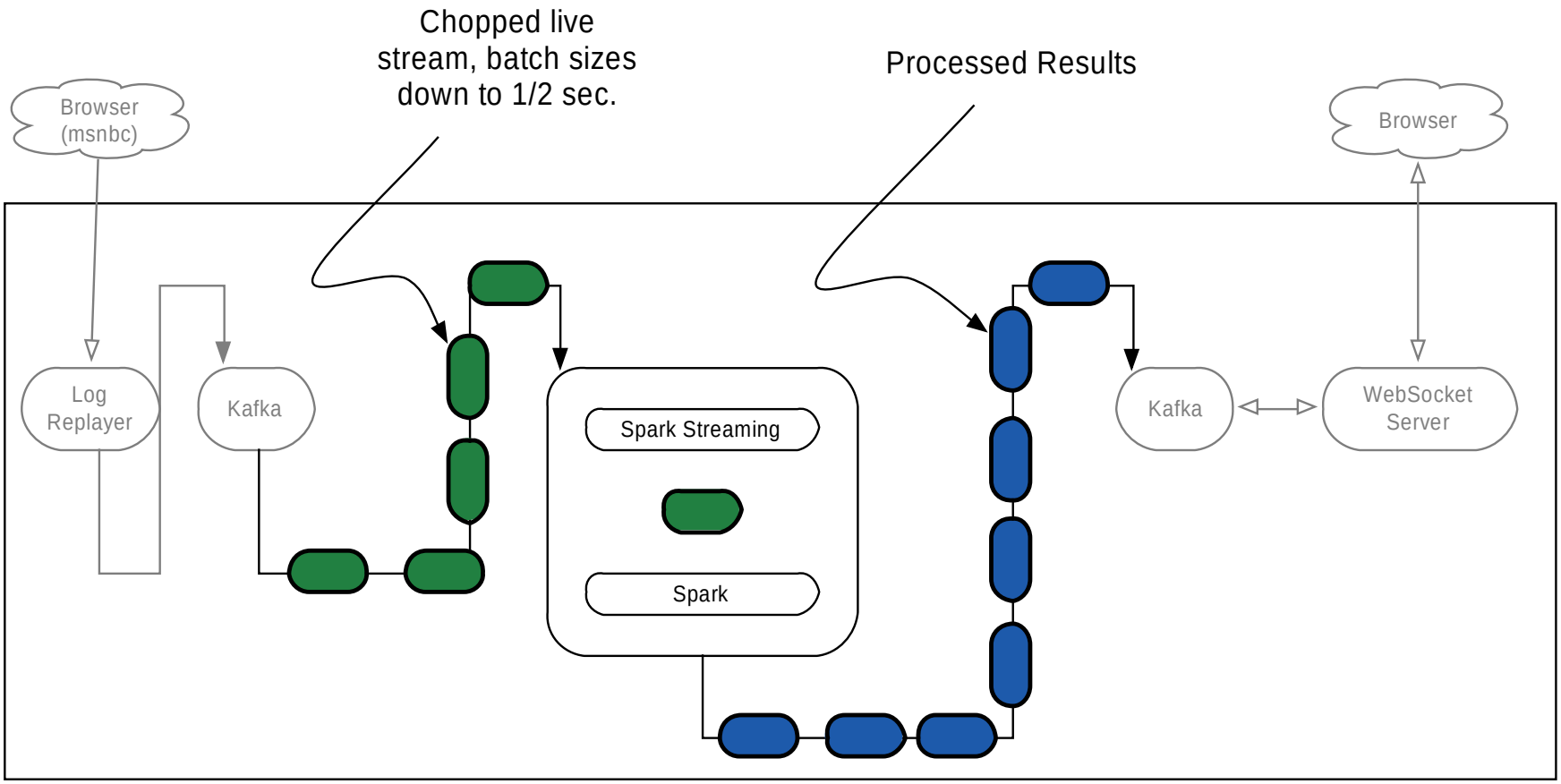
historicalSessions looks like ----

Tuple2<String, Sesssion>("visitorId-1", "{Historical-Session}")

Tuple2<String, Sesssion>("visitorId-2", "{Historical-Session}"))

JavaPairDStream<String, Tuple2<Sesssion, Sesssion>> joined = currentSessions.join(historicalSessions);

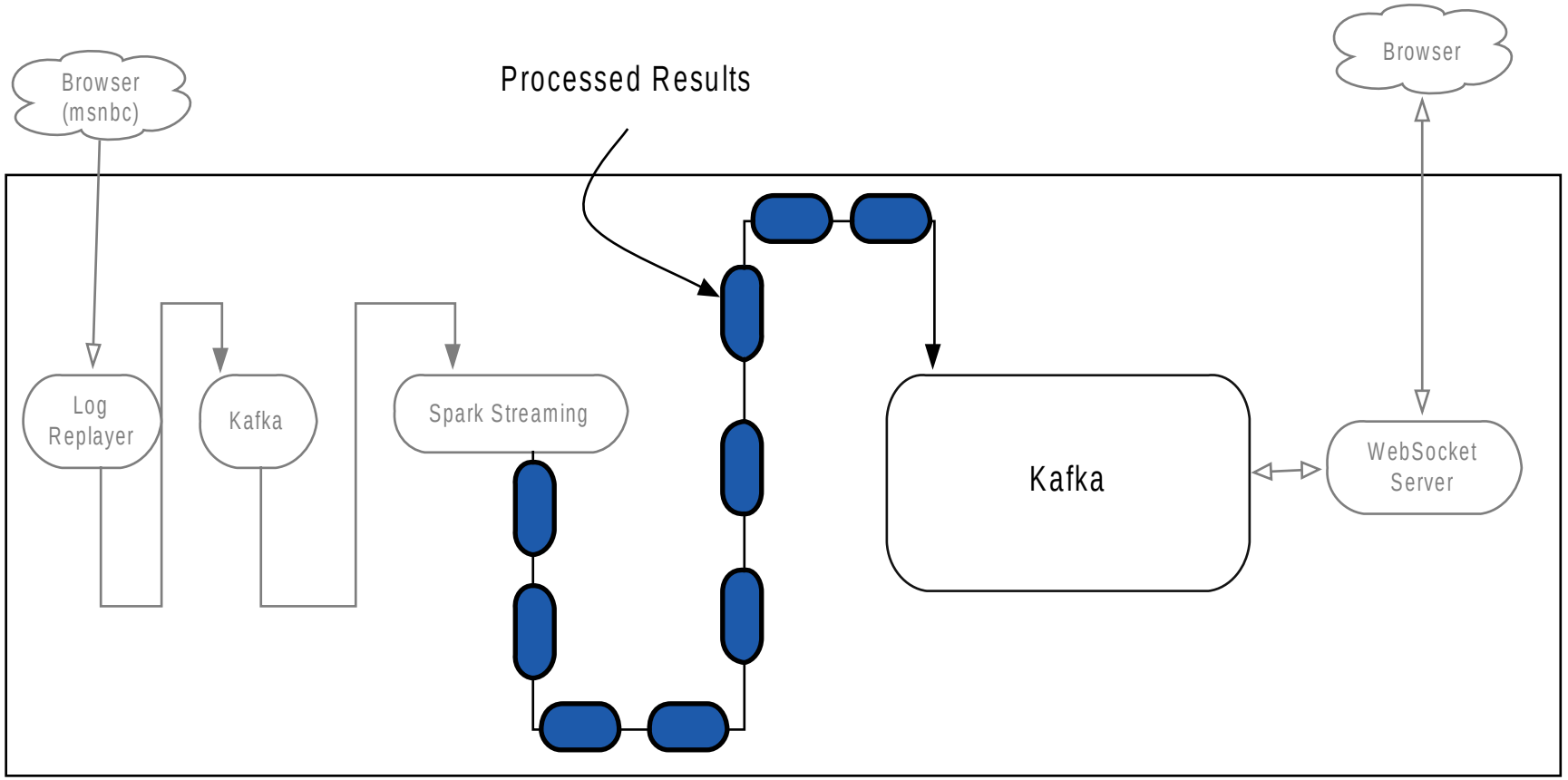
Where are we?


Chopped live stream, batch sizes down to 1/2 sec.
Processed Results
Browser (msnbc)
Browser
Log Replayer
Kafka
Spark Streaming
Spark
Kafka
WebSocket Server

Getting the data out

Spark Streaming currently only supports:

print, foreachRDD, saveAsObjectFiles, saveAsTextFiles, saveAsHadoopFiles



Example - foreachRDD

```
sortedCounts.foreachRDD(
 new Function<JavaPairRDD<Long,String>,Void>(){
  public Void call(JavaPairRDD<Long,String> rdd){
    Map<String,Long> top10 = new HashMap<>();
      for(Tuple2<Long,String> t:rdd.take(10)){
   top10List.put(t._2(),t._1());
      }
                        kafkaProducer.sendTopN(top10List);
   return null;
   }
 }
);
```

WebSockets

- Provides a standard way to get data out
- When the client connects –
  - Read from Kafka and start streaming
- When they disconnect
  - Close Kafka Consumer

Summary

- Spark Streaming works well for ClickStream Analytics
- But
  - Still no good out of the box output operations for a stream.
  - Multi tenancy – needs to be thought through.
  - How do you stop a Job?