Queen Mary University of London

IN-MEMORY AND STREAM PROCESSING

BIG DATA

Félix Cuadrado

felix.cuadrado@qmul.ac.uk

Workshop on Scientific Applications for IoT

ICTP, Trieste, Italy. 24th March 2015

Contents

- **In-memory Processing**
- Stream Processing

Hadoop is a batch processing framework

- Designed to process very large datasets
- Efficient at processing the Map stage
  - Data already distributed
- Inefficient in I/O - Communications
  - Data must be loaded and written from HDFS
  - Shuffle and Sort incur on large network traffic
- Job startup and finish takes seconds, regardless of size of the dataset

Map/Reduce is not a good fit for every case

- Rigid structure: Map, Shuffle Sort, Reduce
- No support for iterations
- Only one synchronization barrier
- See graph processing as an example…

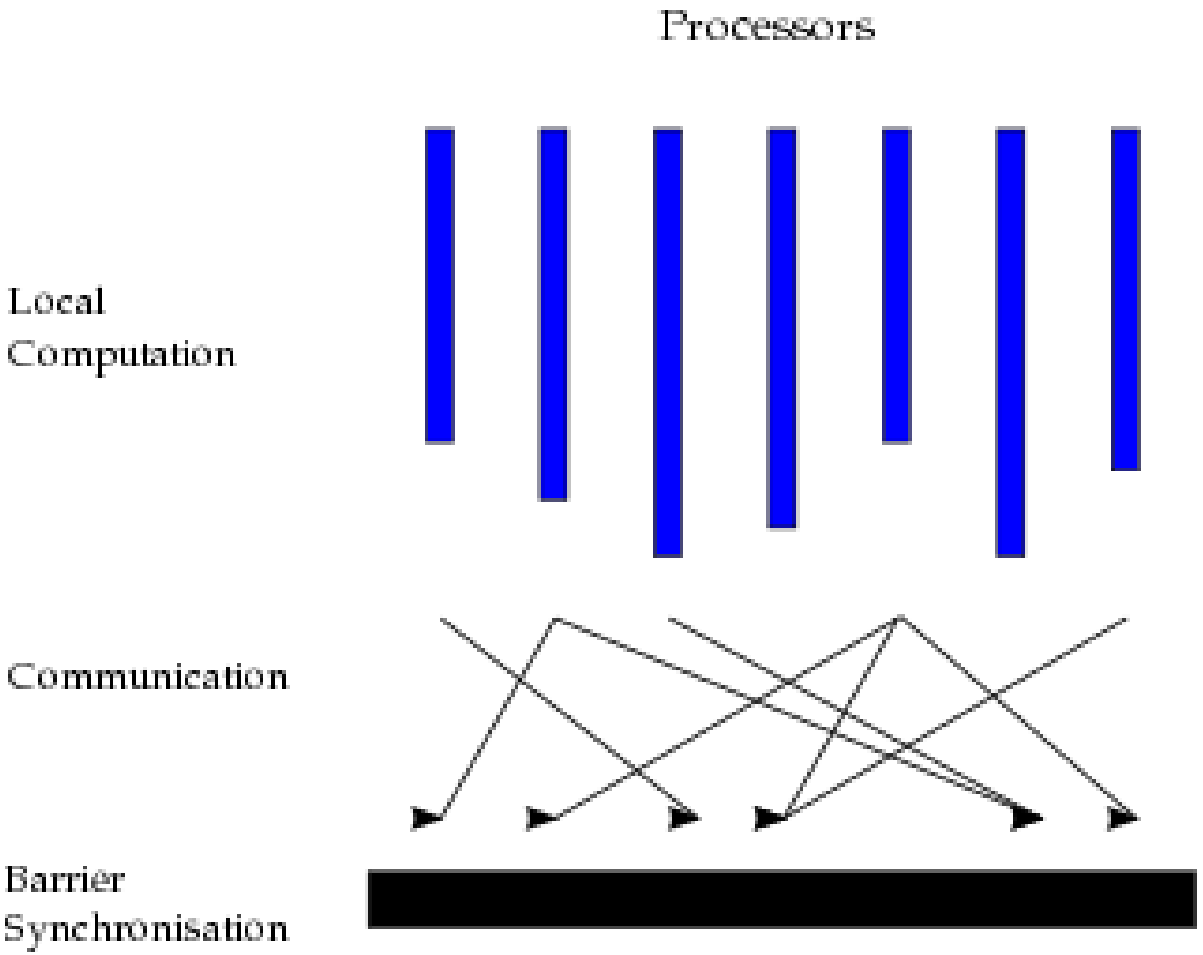
In-memory processing

- Data is already loaded in memory before starting computation
- More flexible computation processes
- Iterations can be efficiently supported
- Three big initiatives
  - Graph-centric: Pregel
  - General purpose: Spark
  - SQL focused (read-only) : Cloudera Impala (Google Dremel)

Bulk Synchronous Parallel (BSP)

- Iterative Parallel computing model proposed by Valiant in the 70s
- Computation happens in supersteps (iterations), with global synchronization points
- Every process works independently
- Processes can send messages to other processes
- A global synchronization barrier forces all processors to wait until everyone has finished

BSP synchronization model



Google Pregel

- Implement BSP for vertex-centric graph processing
- Different high-level abstraction: vertex, receiving and sending messages every iteration
- Open source implementation in Apache Giraph (built on top of Hadoop), other frameworks (Hama, Spark GraphX)
- Graph is automatically partitioned among the distributed machines
- No fault tolerance

Pregel example: SSSP in Apache Giraph

```
public void compute(Iterator<DoubleWritable> msgIterator) {
    if (getSuperstep() == 0) {
       setVertexValue(new DoubleWritable(Double.MAX_VALUE));
    }
    double minDist = isSource() ? 0d : Double.MAX_VALUE;
    while (msgIterator.hasNext()) {
          minDist = Math.min(minDist, msgIterator.next().get());       }
    if (minDist < getVertexValue().get()) {
          setVertexValue(new DoubleWritable(minDist));
          for (LongWritable targetVertexId : this) {
                FloatWritable edgeValue = getEdgeValue(targetVertexId);
                sendMsg(targetVertexId,  new DoubleWritable(minDist + edgeValue.get()));
          }
    }
    voteToHalt();
}
```

Spark project

- Originated at Berkeley uni, at AMPLab (creator Matei Zaharia)
  - Now spin off company, DataBricks, handles development
- Origin: Resillient Distributed Datasets Paper
  - NSDI' 12 – Best paper award
- Released as open source
- Became Apache top level project recently
  - Currently the most active Apache project!

Spark

- Goal: Provide distributed collections (across a cluster) that you can work with as if they were local
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes stragglers)
  - Data locality
  - Scalability
- **Approach:** augment data flow model with “resilient distributed datasets” (RDDs)

Resillient Distributed Datasets

- Resilient distributed datasets (RDDs)
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Can be *cached* across parallel operations
- Transformations (e.g. map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g. count, collect, save)
  - Return a result or write it to storage

Spark RDD operations

Transformations (define a new RDD from an existing one)	**Parallel operations** (take an RDD and return a result to driver)
map filter sample union groupByKey reduceByKey join cache …	reduce collect count save lookupKey …

Scala notes for Spark

- It is possible to write Spark programs in Java, or Python, but Scala is the native language
- Syntax is similar to Java (bytecode compatible), but has powerful type inference features, as well as functional programming possibilities.
  - We declare all variables as val (type is automatically inferred)
  - Tuples of elements (a,b,c) are first order elements.
    - Pairs (2-Tuples) will be very useful to model key-value pair elements
  - We will make extensive use of Scala functional capabilities for passing functions as parameters
    - x => x+2

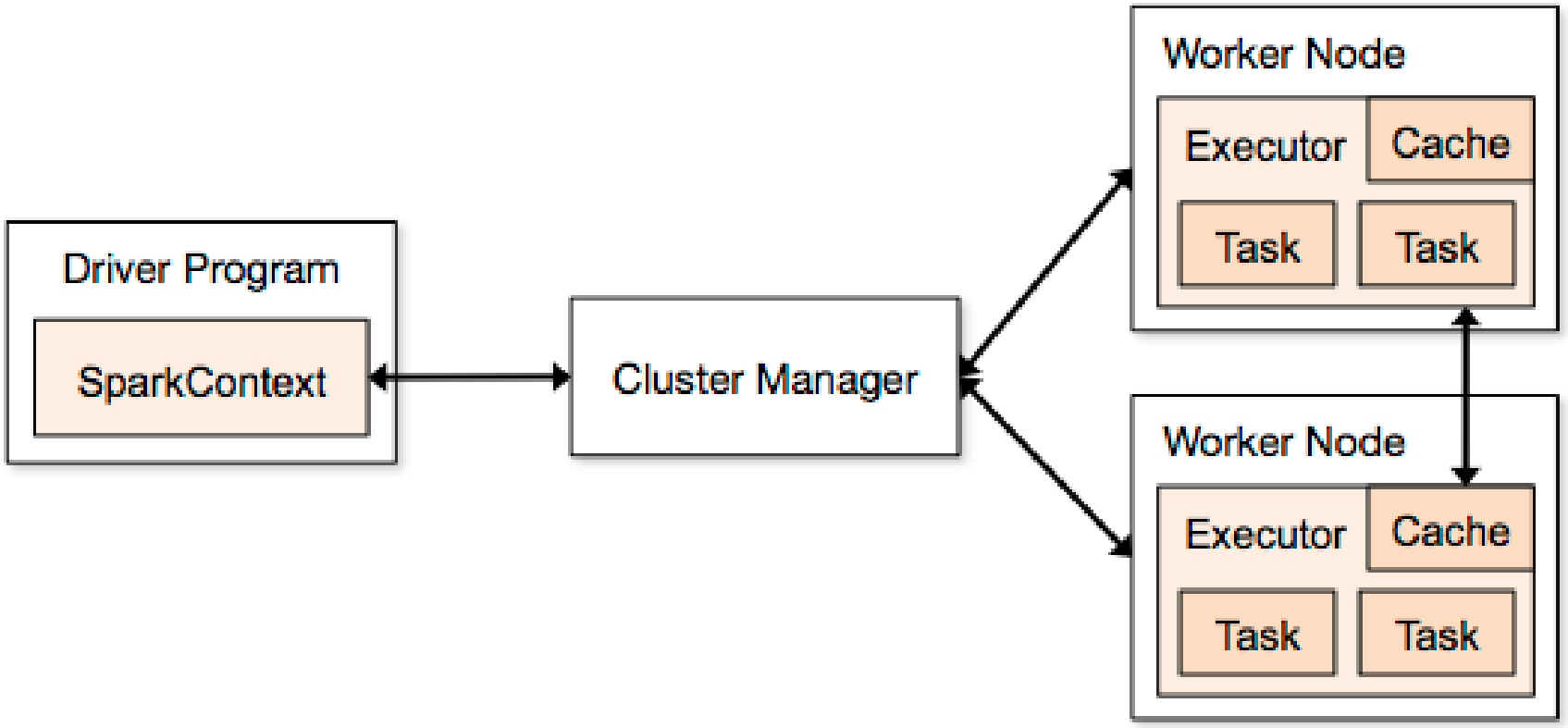
Word Count in Spark (Scala code)

```
val lines = spark.textFile(“hdfs://...”)

val words = lines.flatMap(lines =>
                          lines.split(“\\s”) )

val counts = words.map(word => (word, 1))
                  .reduceByKey((a,b)=>a+b)

counts.saveAsTextFile(“hdfs://...”)
```

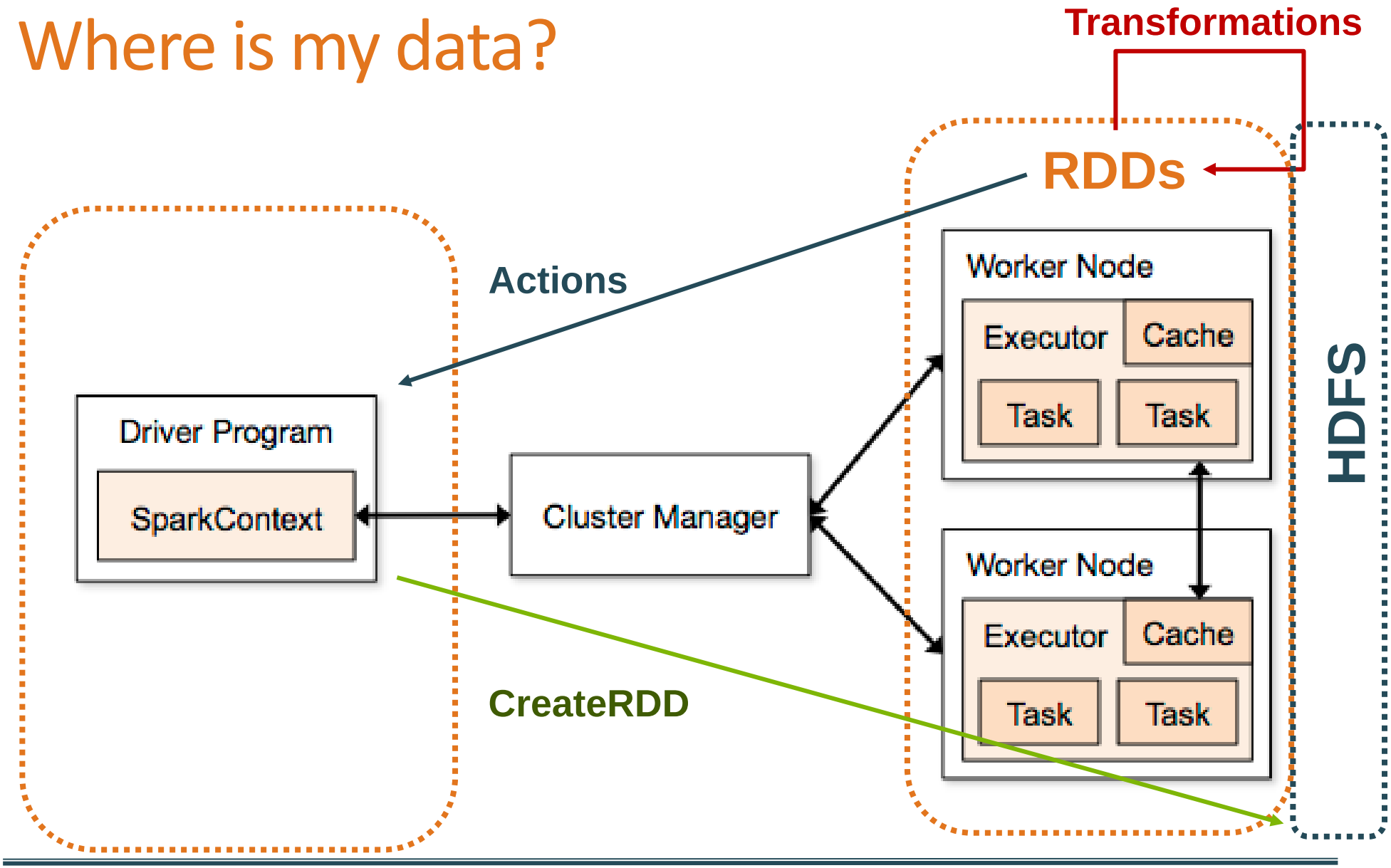
A closer look at Spark

- A Spark application consists of a **driver** program that executes various **parallel** operations on **RDDs** partitioned across the cluster.
- **RDDs** are **created** by starting with a HDFS or an existing Scala collection in the driver program, and **transforming** it.
  - Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations.
- **Actions** transfer **RDDs** are retrieved to either HDFS storage, or the memory of the driver program
- Spark also supports shared variables that can be used in parallel operations: broadcast variables, and accumulators

Spark Execution Architecture



Where is my data?



Creating RDDs

- Any existing collection can be converted to an RDD using parallelize
  - sc.parallelize(List(1, 2, 3))
- HDFS input can be read with sc methods
  - sc.textFile("hdfs://namenode:9000/path/file")
  - Returns a collection of lines
  - Other sc methods for reading SequenceFiles, or any Hadoop compatible InputFormat

‘Move computation to the code’ in Spark

- Computation is expressed as **functions** that are applied in RDD transformations, actions
- Anonymous functions (implemented inside the transformation)

```
timeSeries.map ((x: Int) => x + 2) // full version
timeSeries.map ( x => x + 2 )// type inferred
timeSeries.map (_ + 2 )// when each argument is used exactly once
timeSeries.map ((x => { // when body is a block of code
 val numberToAdd = 2
 x + numberToAdd     })
```

  - Named functions

```
def addTwo(x: Int): Int = x + 2
list.map(addTwo)
```

Sample Spark RDD Transformations

- **map**: creates a new RDD with the same number of elements, each one is the result of applying the transformation function to it
  - `val tweet = messages.map( x => x.split(“,”)(3) ) //we select the 3rd element`

- **filter**: creates a new RDD with at most the number of elements from the original one. The element is only transferred if the function returns true for the element
  - `val grave= logs.filter( x => x.startsWith(“GRAVE”) )`

Spark RDD reduce operations

- Analogous to functional programming. Returns **one single value** from a list of values
- Applies a binary function that returns one value from two equal types
  - list.reduce ((a,b) => (a+b))
  - [1,2,3,4,5] -> 15
- ReduceByKey is the transformation analogous to MapReduce's Reduce + Combine

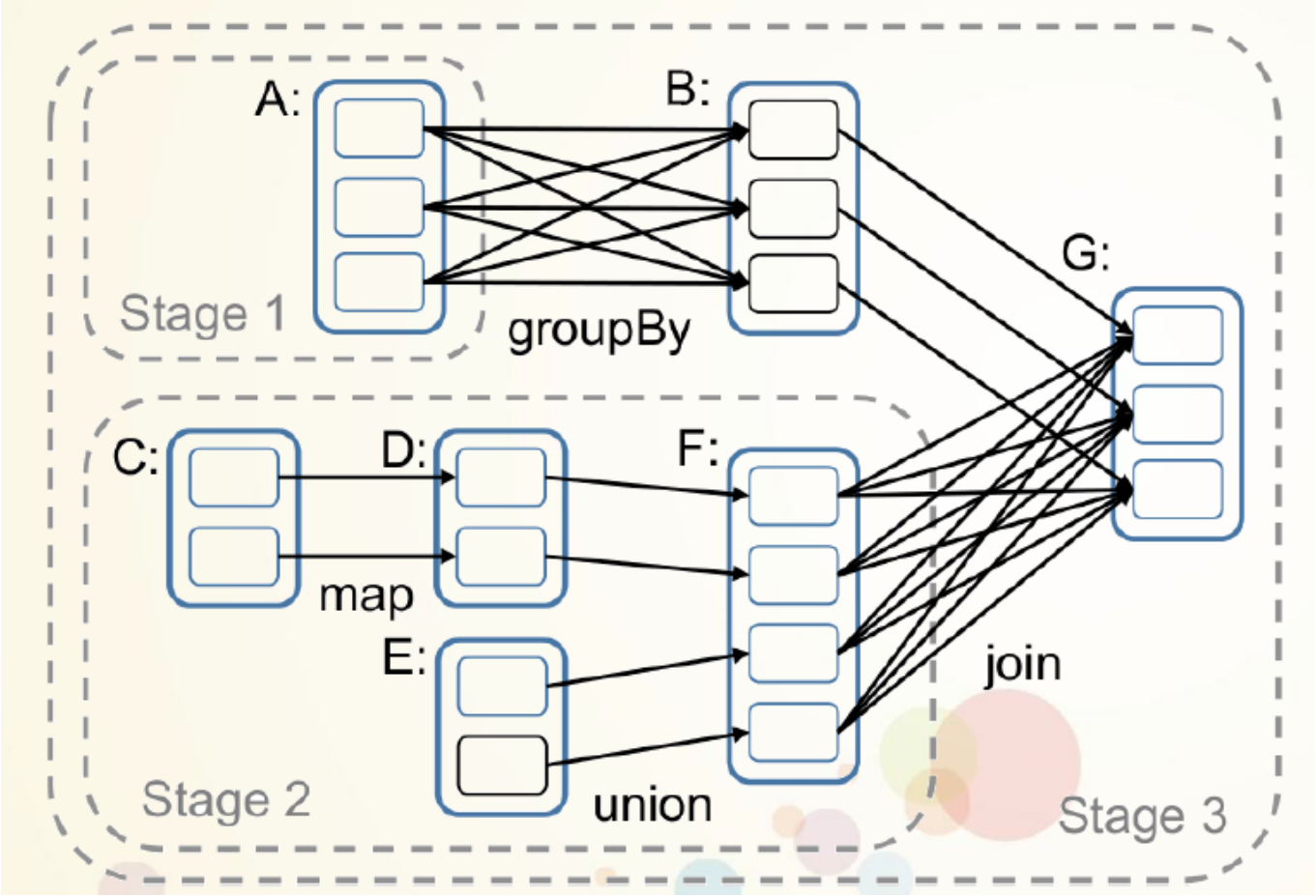
Map/Reduce pattern in Spark

- Spark has specific transformations that mirror the shuffling taking place between Map and Reduce jobs
  - They require the input RDD to be a collection of pairs of (key,value) elements
- **reduceByKey:** groups together all the values belonging to the same key, and compute a reduce function (returning a single value from them)
- **groupByKey:** returns a dataset of (K, Iterable<V>) pairs (more generic)
  - If followed by a Map it is equivalent to MapReduce's Reduce

Spark Parallelism

- RDD transformations are executed in parallel
  - An RDD is partitioned into n **slices**
  - Slices might be located in different machines
  - Slice: Unit of parallelism
  - How many? 2-4 slices are ok per CPU
  - Number of slices is automatically computed
    - Default: 1 per HDFS block size when reading from HDFS, can be higher

RDD Execution & message flows



Scala/Spark use of tuples

- Contrary to MapReduce, RDDs do not have to be key/value pairs
- Key/value pairs are usually represented by Scala tuples
- Easily created with map functions
  - x => (x,1)
- The ._1, ._2 operator allows to select key or value respectively

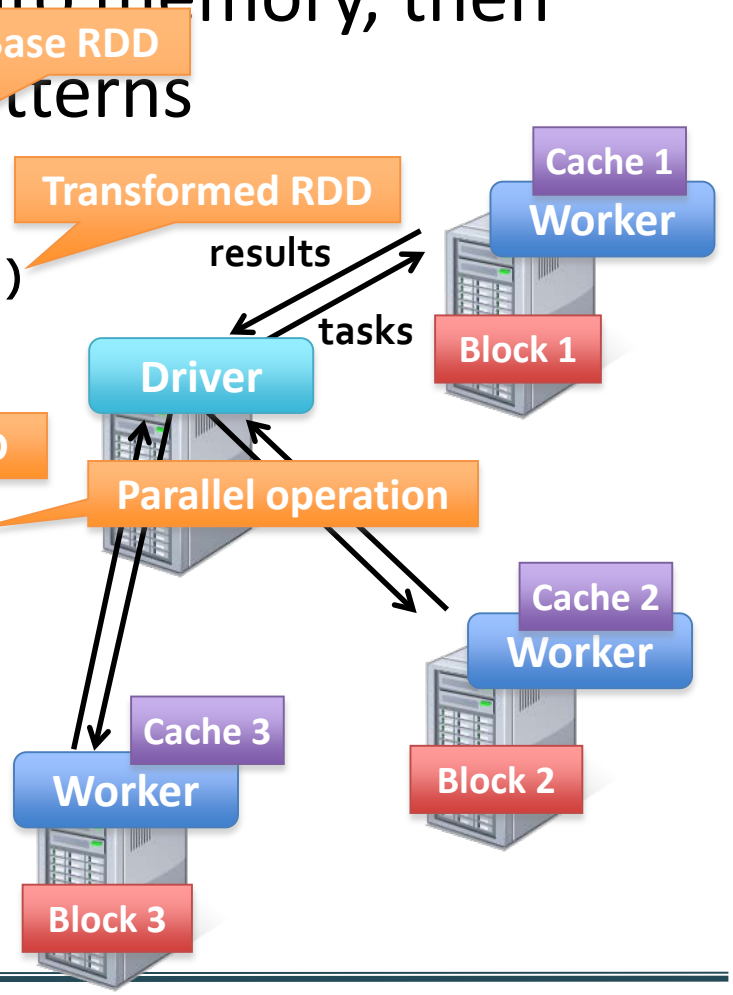
RDD Persistence

- Spark can **persist** (cache) a dataset in memory across operations: Each node stores in memory the partitions it computes for later reuse.
  - Much faster future actions to be much faster (>10x).
  - **Key tool for iterative** algorithms and fast **interactive** use.
- Explicit action: use persist() or cache() methods
  - The first time it is computed in an action, it will be kept in memory on the nodes. created it.
- Multiple persistence options (memory &| disk)
  - Can be difficult to use properly

Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(_.startsWith(“ERROR”))
messages = errors.map(_.split(‘\t’)(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains(“foo”)).count
cachedMsgs.filter(_.contains(“bar”)).count
. . .
```



Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

Spark execution platform

- Spark provides option on which execution platform to use
  - Mesos: solution developed also at UC Berkeley, default option. Also supports other frameworks
  - Apache Hadoop YARN: integration with the Hadoop resource manager (allows Spark and MapReduce to coexist)

Deferred execution

- Spark only executes RDD transformations the moment are needed
- When defining a set of transformations, only the invocation of an action (needing a final result) triggers the execution chain
- Allows several internal optimisations
  - Combining several operations to the same element without keeping internal state

Logistic Regression Code

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Spark performance issues

- "With great power comes great responsibility"

Ben Parker

- All the added expressivity of Spark makes the task of efficiently allocating the different RDDs much more challenging
- Errors appear more often, and they can be hard to debug
- Knowledge of basics (eg Map/Reduce greatly helps)

Spark Performance Tuning

- Memory tuning
  - Much more prone to OutOfMemory errors than MapReduce.
  - How much memory is taken for each RDD slice?
- How many partitions make sense for each RDD?
- What are the performance implications of each operation?
- Good advice can be found in
  - http://spark.apache.org/docs/1.2.1/tuning.html

Spark ecosystem

- GraphX
  - Node and edge-centric graph processing RDD
- **Spark Streaming**
  - Stream processing model with D-Stream RDDs
- MLib
  - Set of machine learning algorithms implemented in Spark
- Spark SQL

Contents

- In-memory Processing
- **Stream Processing**

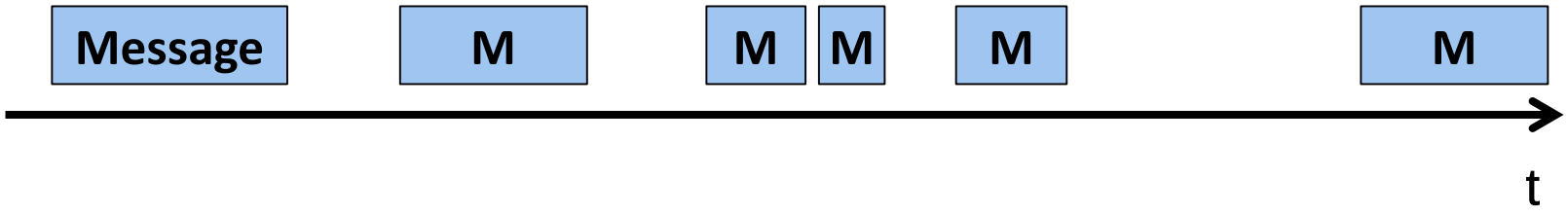
Information streams

- Data is continuously generated from multiple sources
  - Messages from a social platform (e.g. Twitter)
  - Network traffic going over a switch
  - Readings from distributed sensors
  - Interactions of users with a web application
- For faster analytics, we might need to process the information the moment it is generated
  - Process the information streams

Stream processing

- Continuous processing model
- Rather than processing a static dataset, we apply a function to each new element that comes from an information stream
- Rather than single results, we look for the evolution of computations, or to raise alerts when something is different than the norm
- Near real-time response times

Information Streams

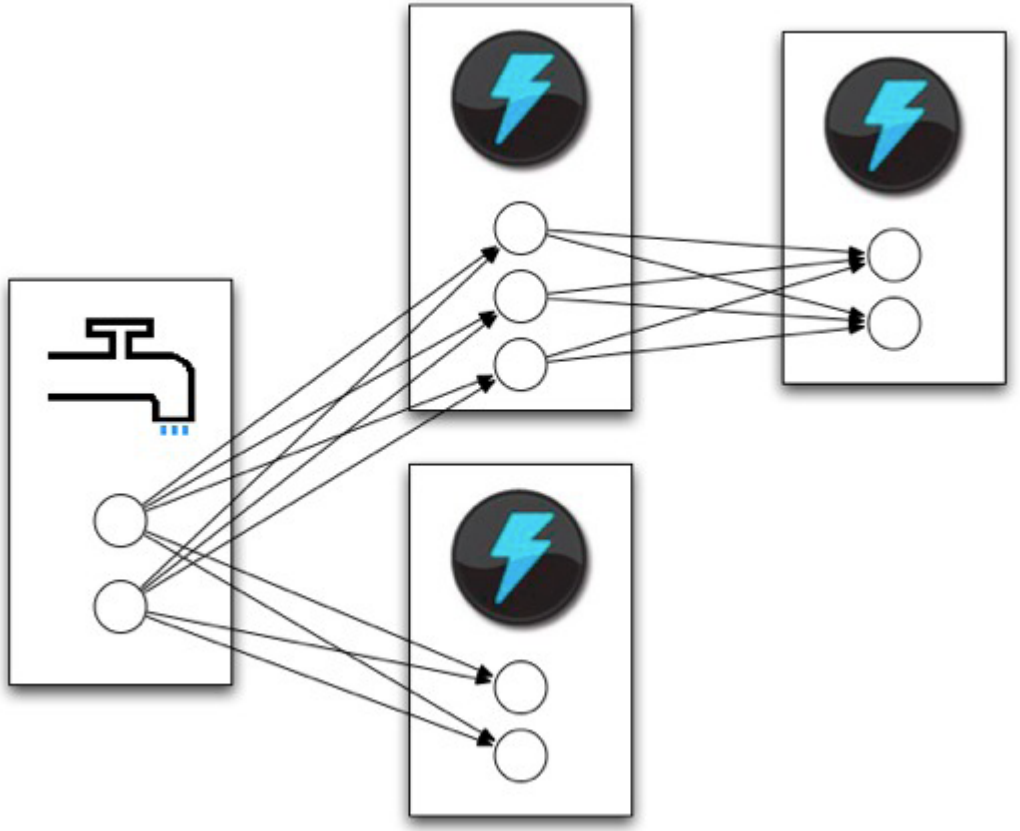


Unbounded sequence of messages
Arrival time is not fixed

Apache Storm

- Developed by BackType which was acquired by Twitter. Now donated to Apache foundation
- Storm provides realtime computation of data streams
  - Scalable (distribution of blocks, horizontal replication)
  - Guarantees no data loss
  - Extremely robust and fault-tolerant
  - Programming language agnostic

Storm Topology



Spouts and bolts execute as many tasks across the cluster
Horizontal scaling/parallelism

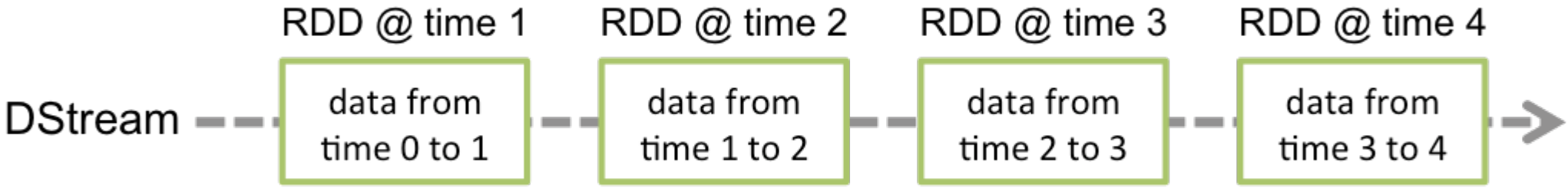
Spark Streaming: Discretized Streams

- Unlike pure stream processing, we process the incoming messages on micro batches

Discretized Streams

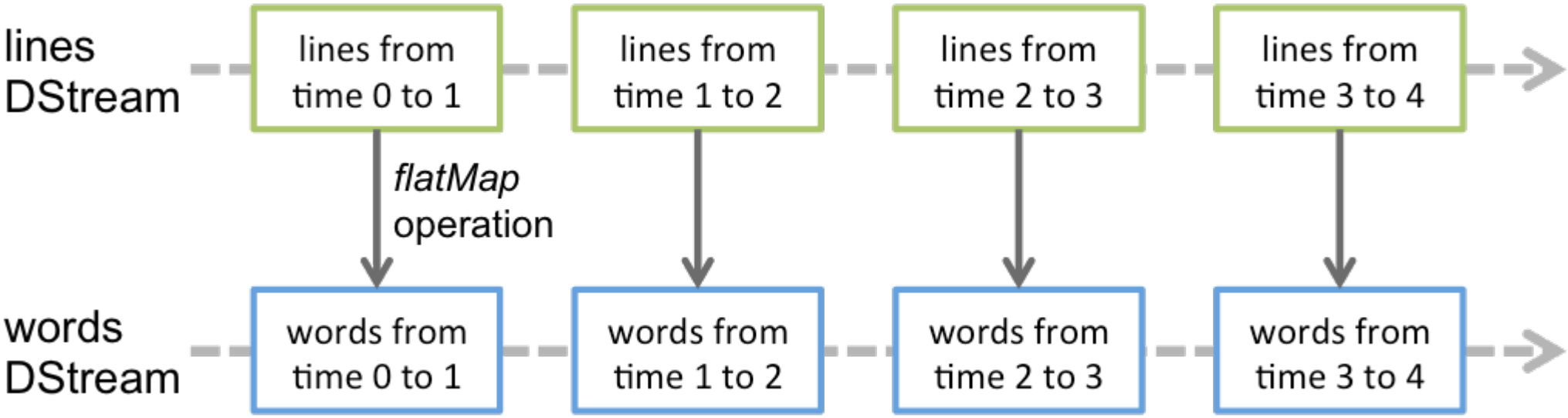
- Reuse Spark Programming model
  - Transformations on RDDs
- RDDs are created combining all the messages in a defined time interval
- A new RDD is processed at each slot
- Spark code for creating one:
  - `val streamFromMQTT = MQTTUtils.createStream(ssc, brokerUrl, topic, StorageLevel.MEMORY_ONLY_SER_2)`

Dstream RDDs



Discreteness of time matters!
- The shorter the time the faster response potentially
- … but also makes it slower to process

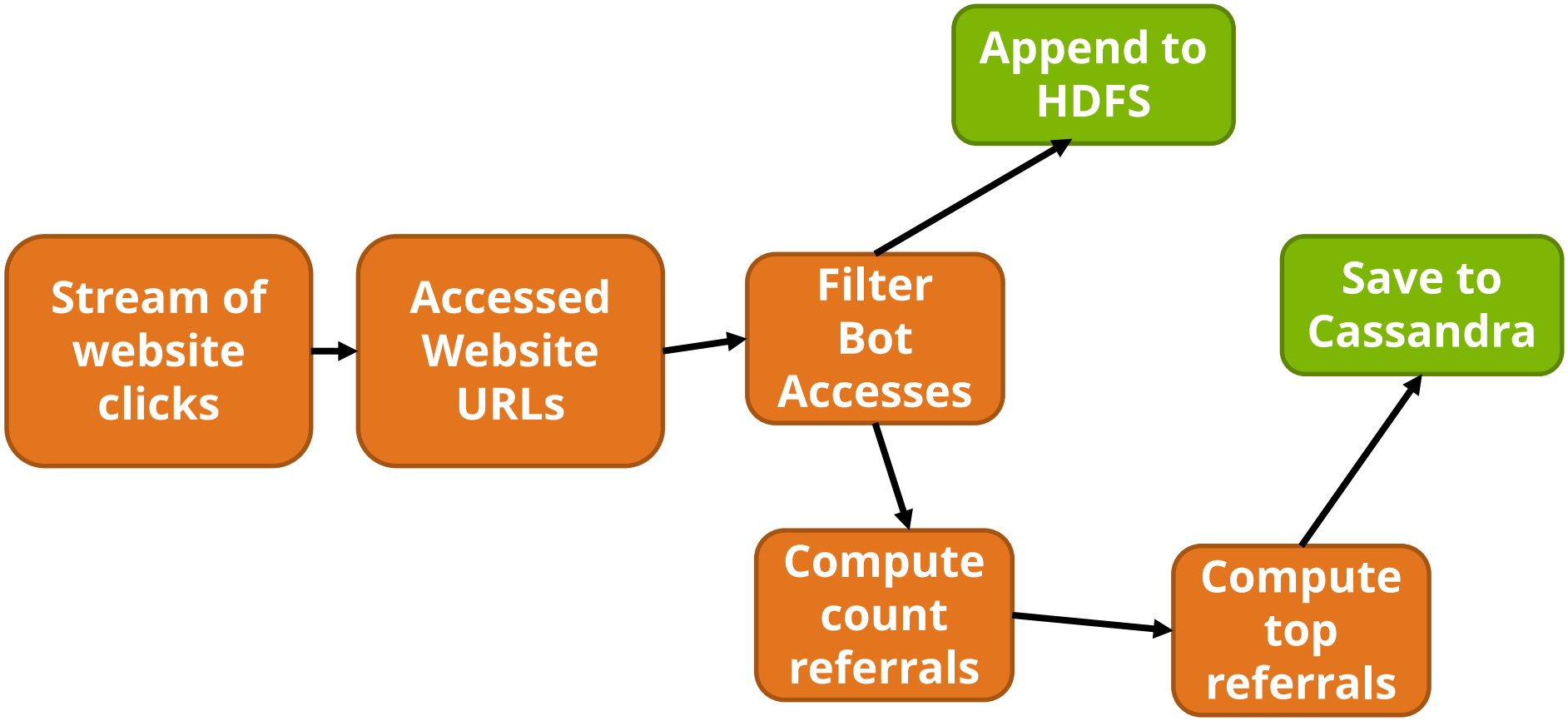
D Stream transformations



D-Stream Streaming context

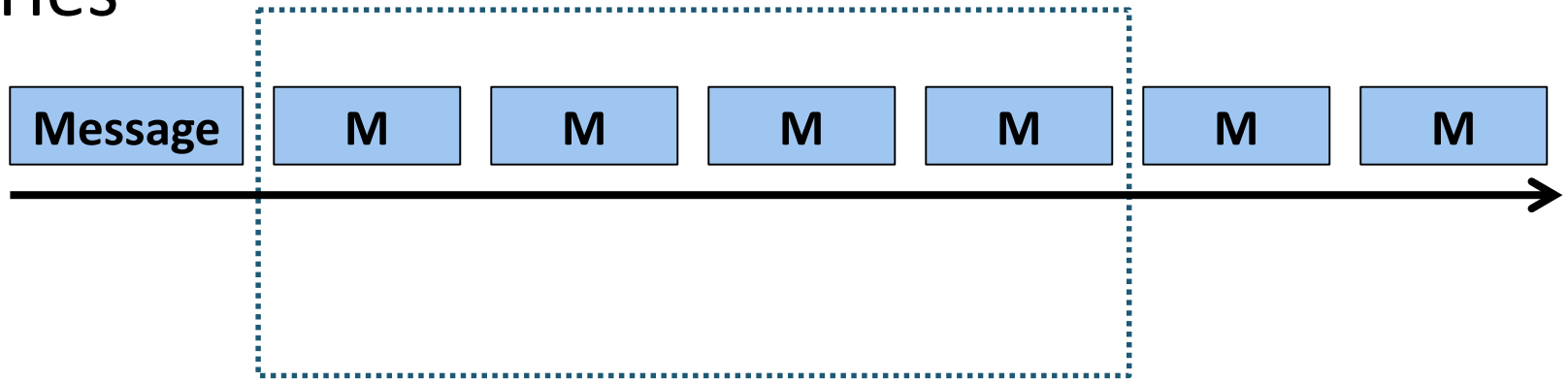
- Spark streaming flows are configured by creating a StreamingContext, configuring what transformations flow will be done, and the invoke the start method
  - **`val ssc = new StreamingContext(sparkUrl, "Tutorial", Seconds(1), sparkHome, Seq(jarFile))`**
- There must be some action collecting in some way the results of a temporal RDD

Sample topology: Website click analysis



Sliding windows

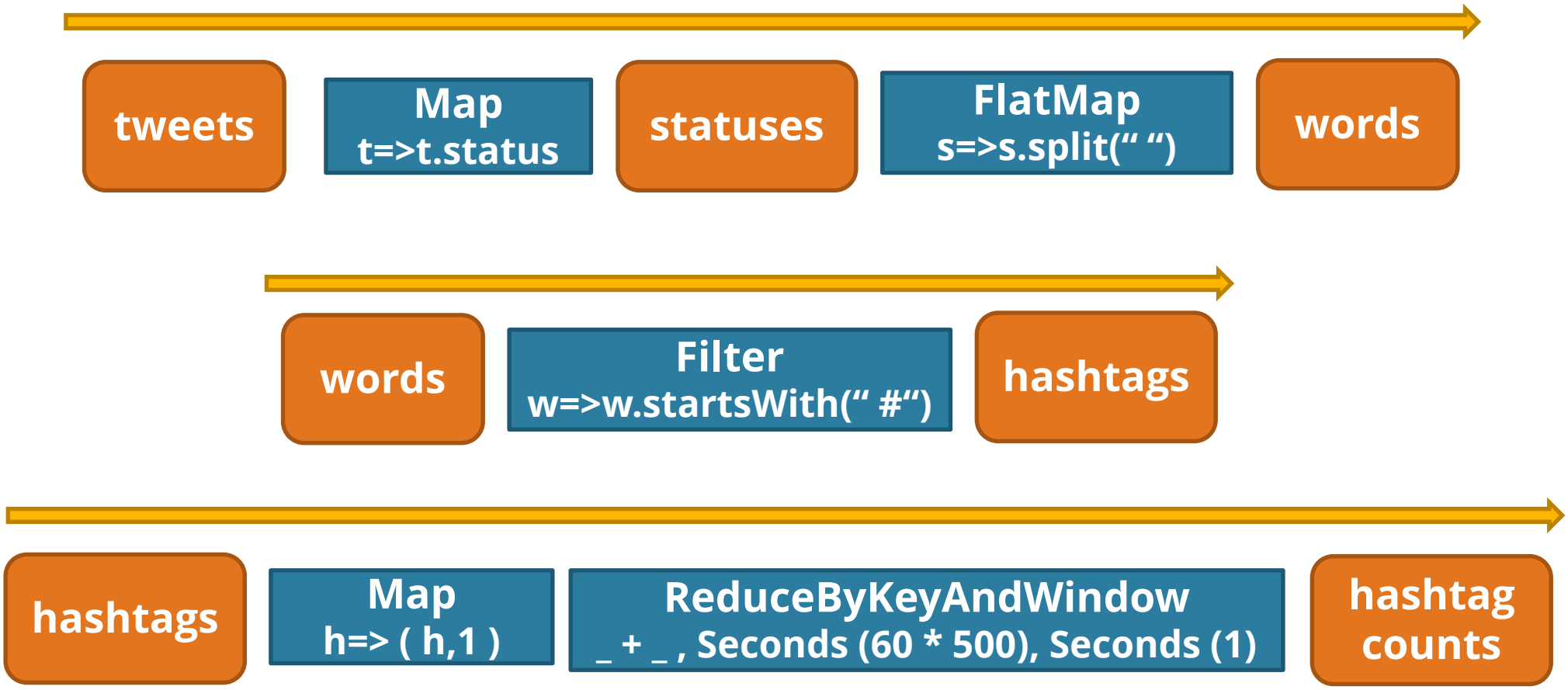
- Some computations need to look at a set of stream messages in order to perform its computation
- A sliding window stores a rolling list with the latest items from the stream
- Contents change over time, replaced by new entries



Sliding window operations in Spark D-Stream

- D-Stream provides direct API support for specifying streams
- Two parameters:
  - Size of the window (in seconds)
  - Frequency of computations (in seconds)
- E.g. process the maximum temperature over the last 60 seconds, every 5 seconds.
  - **`reduceByWindowAndKey((a,b)=>math.max(a,b), Seconds(60, Seconds(5) )`**

Sample Twitter processing stream



Sample Twitter Processing Stream

```
val ssc = new StreamingContext(sparkUrl,
"Tutorial", Seconds(1), sparkHome, Seq(jarFile))
val tweets = ssc.twitterStream()
val statuses = tweets.map(status => status.getText()
val words = statuses.flatMap(
                status => status.split(" "))
val hashtags = words.filter(
                word => word.startsWith("#"))
val hashtagCounts = hashtags.map(tag => (tag, 1)).
     reduceByKeyAndWindow(
           _ + _, Seconds(60 * 5), Seconds(1))
ssc.checkpoint(checkpointDir)
ssc.start();
```