BEYOND MAP REDUCE

CLOUD COMPUTING

Dr. Atm Shafiul Alam

a.alam@qmul.ac.uk

Queen Mary University of London

School of Electronic Engineering and Computer Science

Thanks to Dr. Gareth Tyson and Dr. Félix Cuadrado who created the original version of these slides

Contents

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

Hadoop is a batch processing framework (1/2)

- Hadoop supports only batch processing
  - Does not process streamed data, and hence, overall performance is slower
- Designed to process very large datasets
  - HDFS works with a small number of large files for storing datasets rather than larger number of small files
  - Not suited for small files (default HDFS block size of 64MB (Hadoop 1.x) & 128MB (Hadoop 2.x))
    - HDFS lacks the ability to efficiently support the random reading of small files.

Hadoop is a batch processing framework (2/2)

- 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 (e.g. loops)
- Only one synchronization barrier
- See graph processing as an example
  - Think about the journey planning example…

In-memory processing

- Data is already loaded in memory before starting computation
- Advantages:
  - More flexible computation processes
  - Iterations can be efficiently supported
  - No slow I/O necessary → fast response times
- Disadvantages:
  - Data must fit in the memory of the distributed storage/database
  - Additional persistency (with asynchronous flushing) usually required
  - Fault-tolerant is mandatory

In-memory processing

- Three big initiatives
  - General purpose: Spark
  - Graph-centric: Pregel
  - SQL focused (read-only) : Cloudera Impala (Google Dremel)

Spark project

- A fast and general engine for large-scale data processing
- Originated at Berkeley Uni, at AMPLab (creator Matei Zaharia)
- Released as open source
- Became Apache top level project recently
  - Currently the most active Apache project!

https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Overview of Spark

- Apache Spark is a cluster-computing platform that provides an API for distributed programming
- In-memory processing (and storage) engine
  - Load data from HDFS, Cassandra, HBase
  - Resource management via YARN, Mesos, Spark, Amazon EC2

  → It can use Hadoop but also works standalone
- Task scheduling and monitoring
- Rich APIs:
  - APIs for Java, Scala, Python, R
  - High-level domain-specific tools/languages
- Interactive shells with tight integration
- Execution in either local (single node) or cluster mode

Spark

- **Goal**: Provide distributed datasets (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” (**RDD**s).

Resilient Distributed Datasets (RDDs)

- Resilient Distributed Dataset (RDD) is the basic level of abstraction in Spark.
- Distributed memory model: RDDs
  - Immutable collections of data **distributed** across the nodes of the cluster
    - Can be ***rebuilt*** if a partition is lost
    - Can be ***cached*** across parallel operations
  - Programmers create new RDDs by
    - loading data from an input source (e.g., from HDFS), or by
    - transforming an existing collection to generate a new one
  - RDDs can be saved back to HDFS/other programs with actions

RDD Operations

- RDDs are operated upon with functional programming constructs
  - i.e., RDDs are created and modified via programmatic operations
- Two types of operations can be applied to RDDs:
  - **Transformations** (e.g. map, filter, groupBy, join)
    - Lazy operations to build RDDs from other RDDs
    - Executed in parallel (similar to Map, Shuffle from Map/Reduce)
  - **Actions** (e.g. count, collect, save)
    - Return a result or write it to storage
- The Spark API is therefore essentially a collection of operations that create, transform, and export RDDs.

Spark RDD operations

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

A full list of supported transformations and actions can be found: http://spark.apache.org/docs/latest/rdd-programming-guide.html

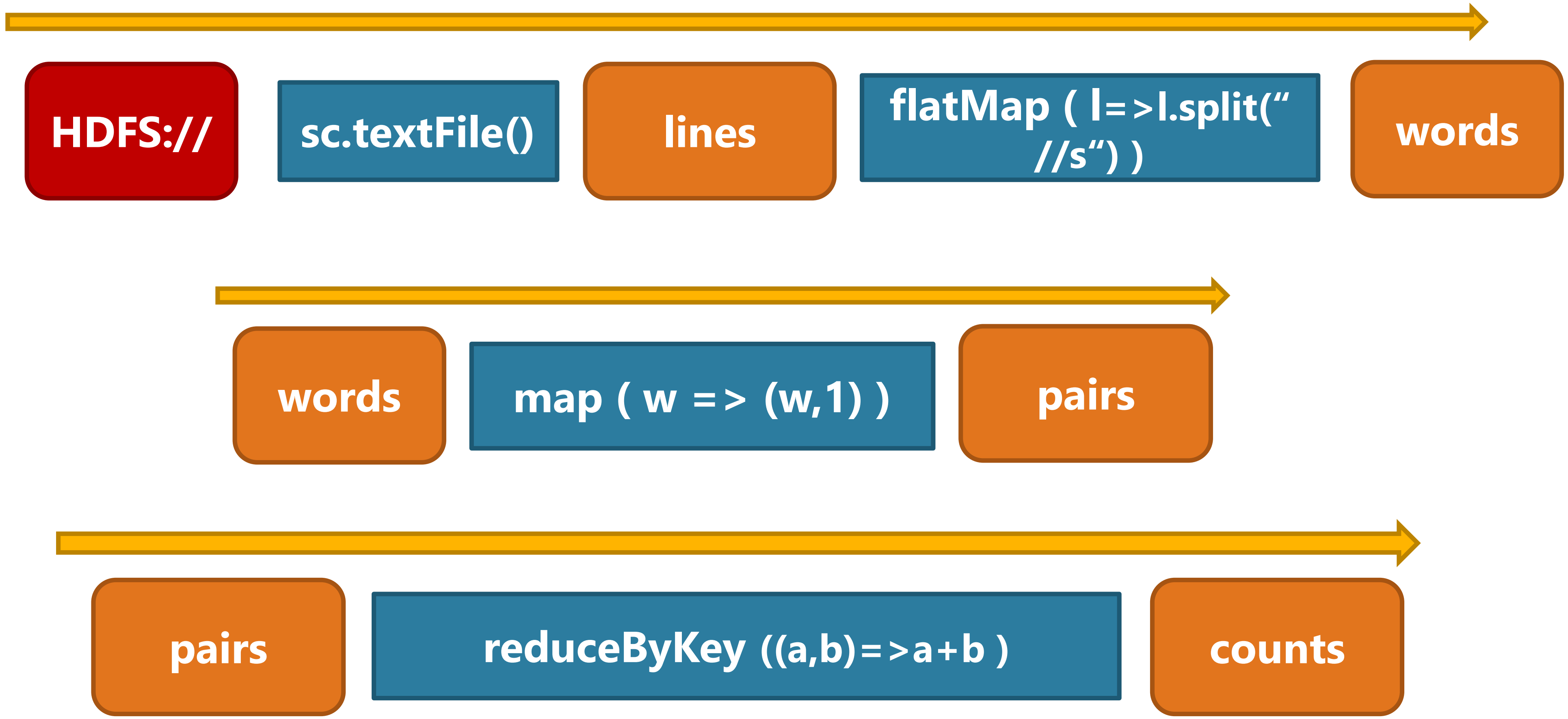
Scala notes for Spark

- 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

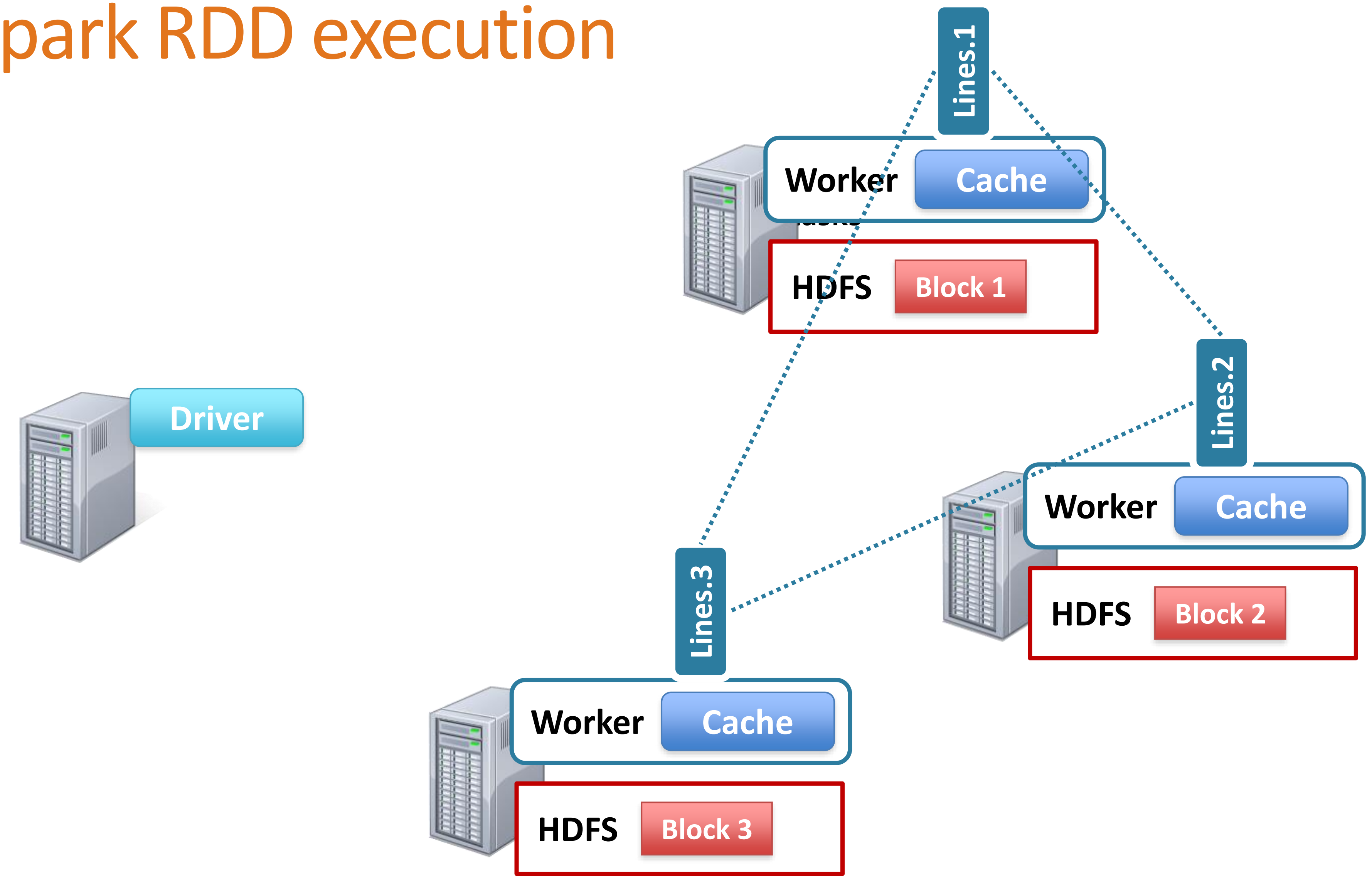
Word Count in Spark (Scala code)

```
val lines = sc.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://...")
```

Word Count in Spark (RDD flow)



Spark RDD execution



A closer look at Spark

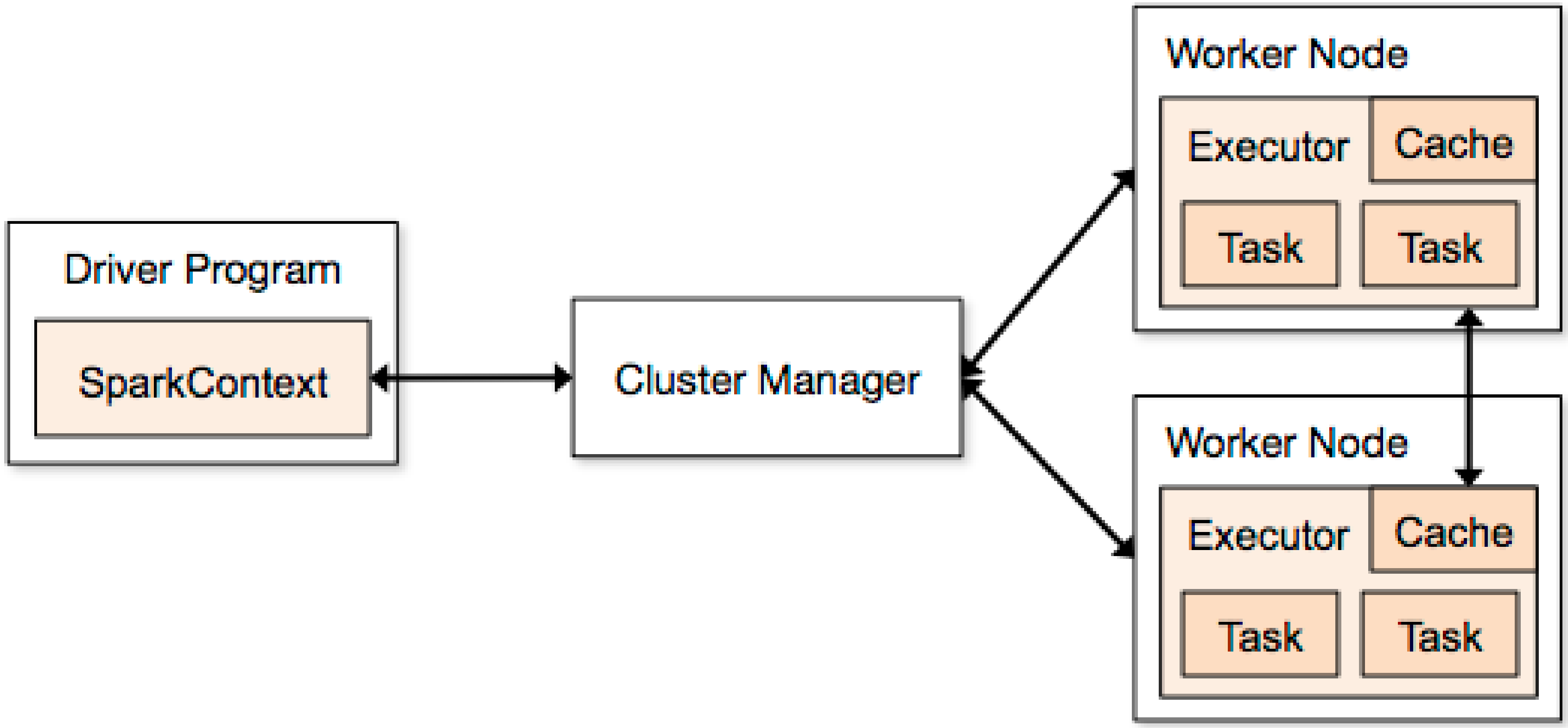
- **RDDs** are **created** by starting with a HDFS or an existing Scala collection in the driver program, and **transforming** it.
  - Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations.
- **Actions** transfer **RDDs** that are retrieved 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 (read-only access) – distributed to all workers.
  - Accumulators (reduce variables) – Counter variables that workers can "add" to using associative operations.

Spark applications

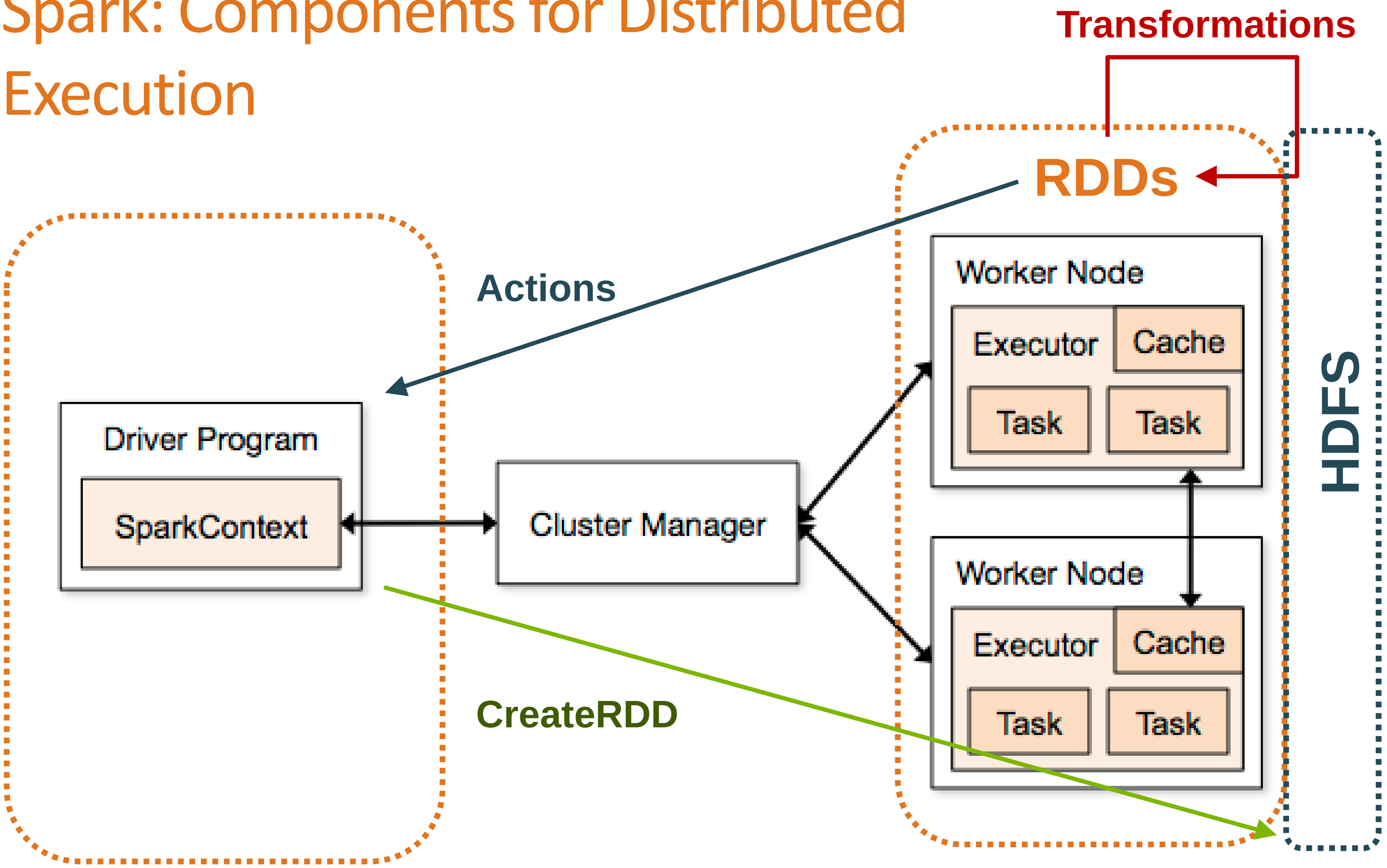
- A Spark application consists of a **driver** program that executes various **parallel** operations on **RDDs** partitioned across the cluster.
- The driver is not the same machine where the RDDs are created
  - Actions are required to retrieve values from the RDDs (e.g. count)

Queen Mary University of London

Spark: Components for Distributed Execution



Spark: Components for Distributed Execution



Spark: Components for Distributed Execution

1. A driver program (in charge of the high-level control flow of work) runs the user's main function and executes various parallel operations on a cluster.

   - Spark applications are run as independent sets of processes, coordinated by a **SparkContext** in a driver program.

   - The context will connect to some cluster manager (e.g., YARN), which allocates system resources.

2. Each worker in the cluster is managed by an executor, which is in turn managed by the SparkContext.
3. The executor manages computation, storage and caching on each machine.
4. The executor processes are responsible for executing this work, in the form of *tasks*, and for storing any data that the user chooses to cache.
5. A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime.
6. Deploying these processes on the cluster is up to the cluster manager in use (e.g., YARN, or Spark Standalone), but the driver and executor themselves exist in every Spark application.

Creating RDDs

- Any existing collection can be converted to an RDD using `parallelize`
  - `rdd = sc.parallelize(List(1, 2, 3))`
- HDFS input can be read with sc methods
  - `rrd2 = 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, and 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

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 any partitions it computes in memory for later reuse.
  - Allows 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 that 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

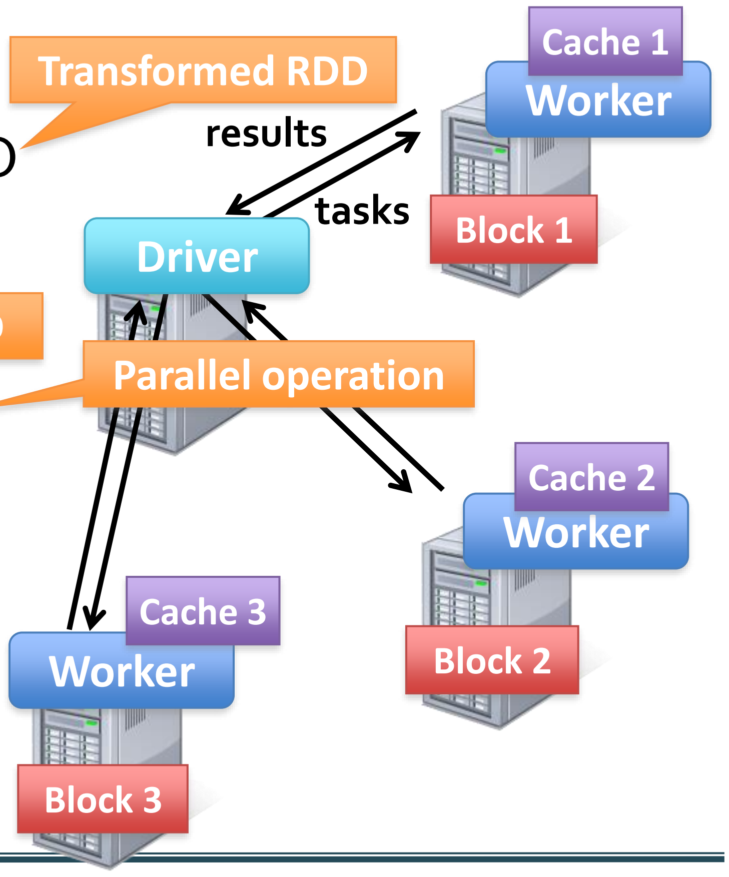
Base RDD

Transformed RDD

```
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
. . .
```

Cached RDD

Parallel operation



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

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 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

Spark performance issues

- 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 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
- MLib
  - Set of machine learning algorithms implemented in Spark
- Spark SQL
- **Spark Streaming**
  - Stream processing model with D-Stream RDDs
- SparkR (R on Spark)
- BlindDB (Approximate SQL)

Contents

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

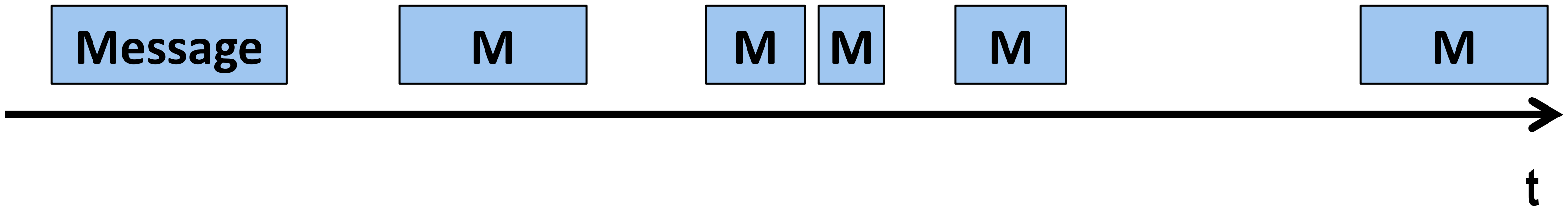
Information streams

- 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

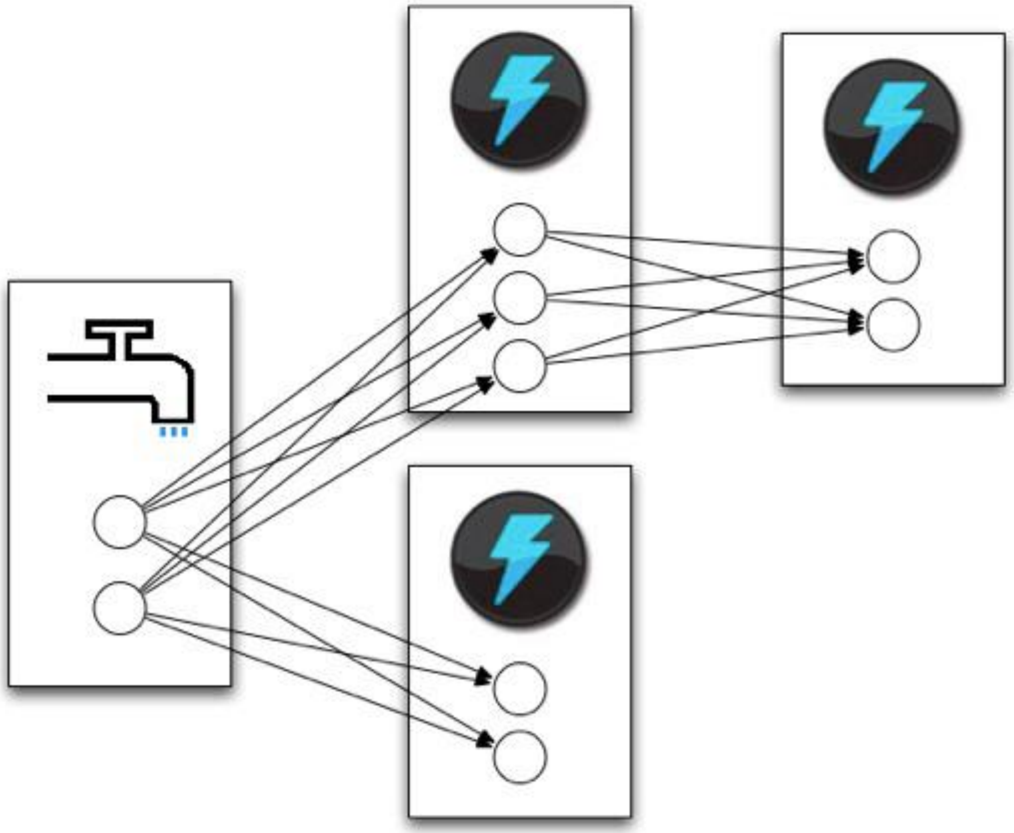


Unbounded sequence of messages
Arrival time is not fixed

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

Distributed Stream processing

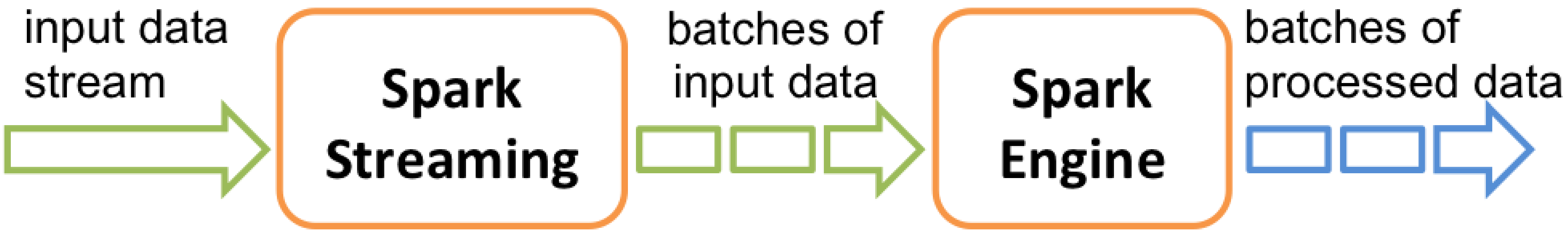


Tasks are distributed across the cluster

Horizontal scaling/parallelism

Spark Streaming: Discretized Streams

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



Chop up the live stream into batches of X seconds

Spark treats each batch of data as RDDs and processes them using RDD operations

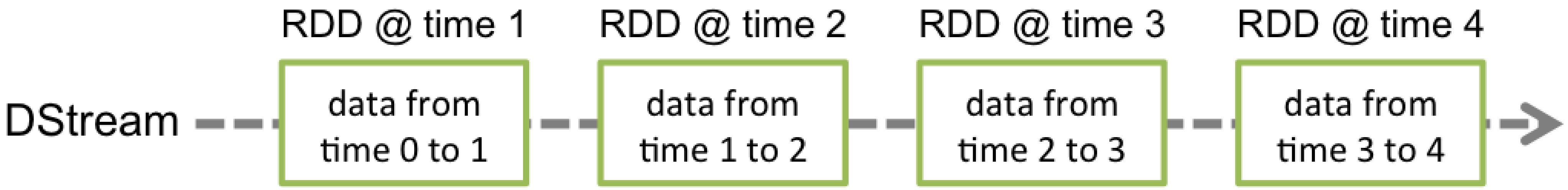
Finally, the processed results of the RDD operations are returned in batches

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)
    ```

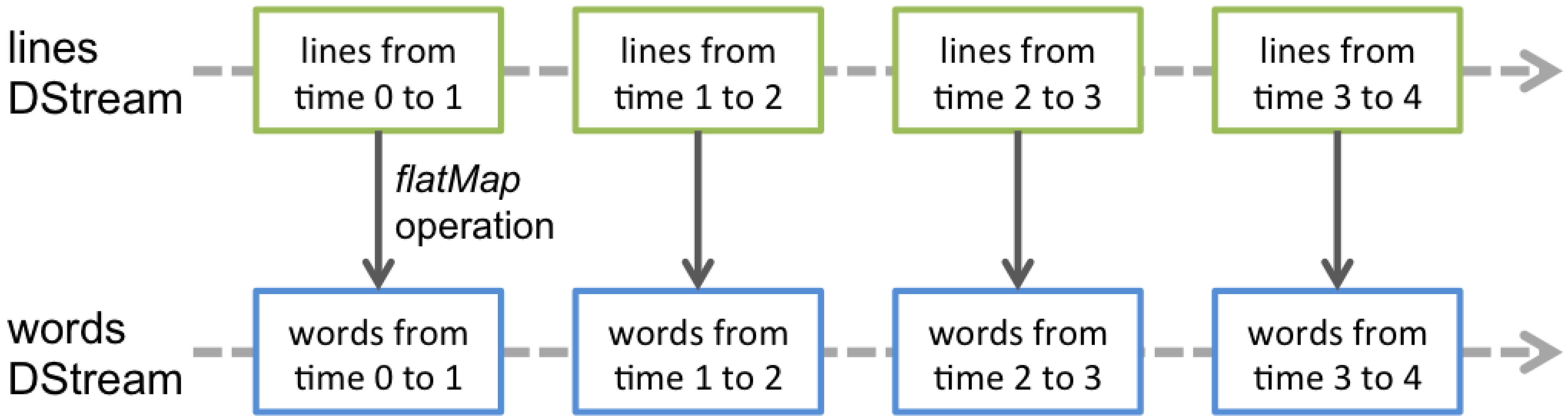
DStream RDDs



Discreteness of time matters!

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

DStream transformations

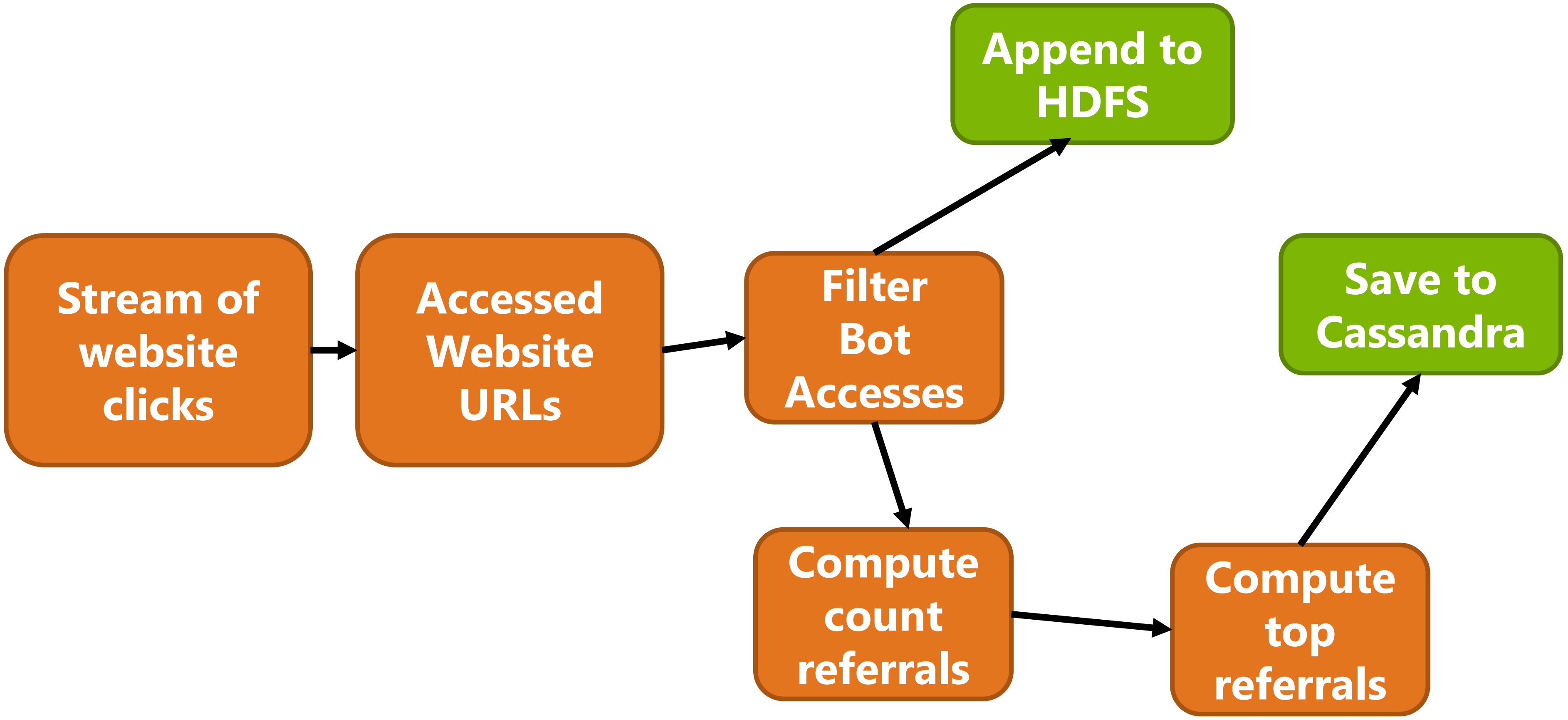


DStream Streaming context

- Spark streaming flows are configured by creating a `StreamingContext`, configuring what transformations flow will be done, and the invoke the start method

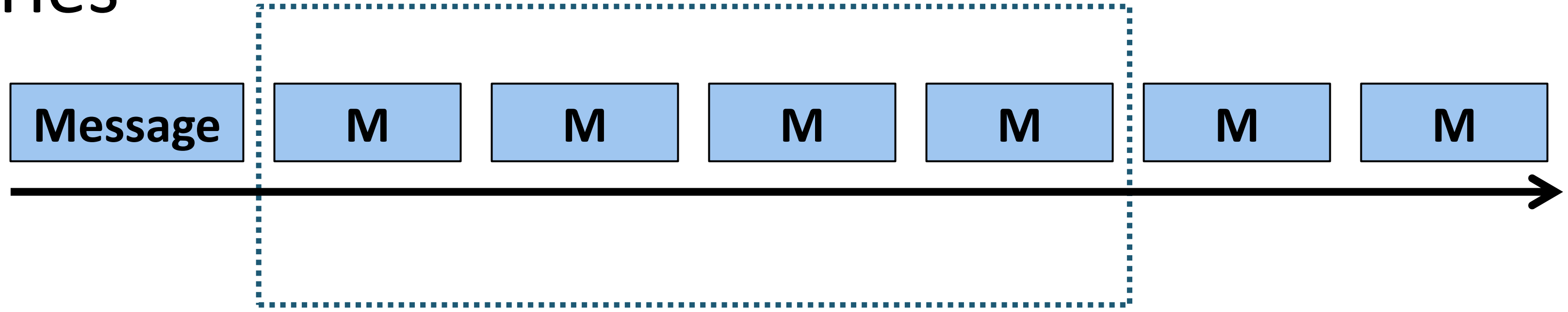
  -
    ```
    val ssc = new StreamingContext(sparkUrl,
    "Tutorial", Seconds(1), sparkHome,
    Seq(jarFile))
    ```
- There must be some action collecting in some way the results of a temporal RDD

Sample topology: Website click analysis



Sliding windows

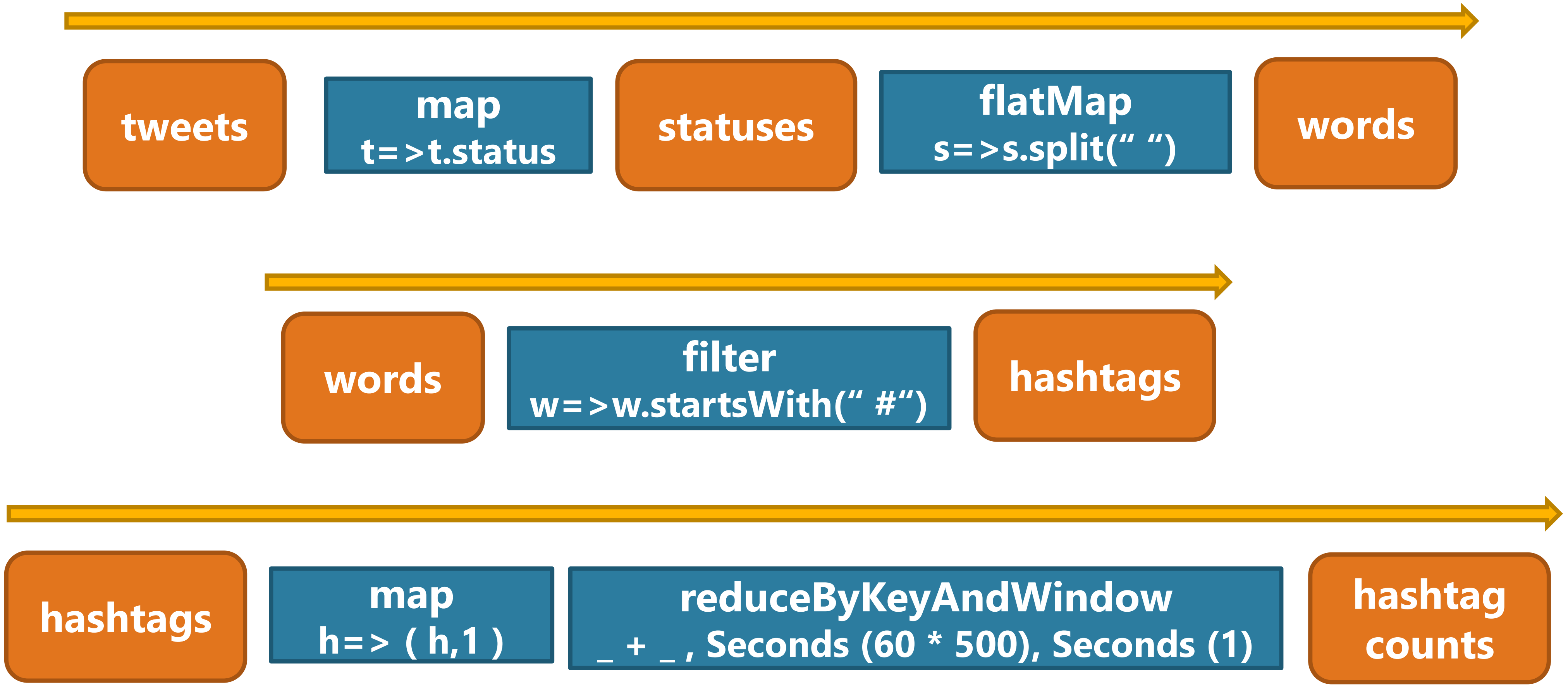
- Some computations need to look at a set of stream messages in order to perform its computation
- A sliding window stores a rolling list with the latest items from the stream
- Contents change over time, replaced by new entries



Sliding window operations in Spark DStream

- DStream 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();
```

Contents

- In-memory Processing
- Stream Processing