IN-MEMORY PROCESSING WITH SPARK (PYSPARK EDITION)

BIG DATA PROCESSING

Félix Cuadrado, Ben Steer

felix.cuadrado@qmul.ac.uk, b.a.steer@qmul.ac.uk

Queen Mary University of London

School of Electronic Engineering and Computer Science

Contents

- **In-memory Processing**
- Apache Spark
- Spark programming
- Spark parallelism considerations

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 native support for iterations
- Only one synchronization barrier

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, Flink
  - SQL focused (read-only) : Cloudera Impala (Google Dremel)

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!

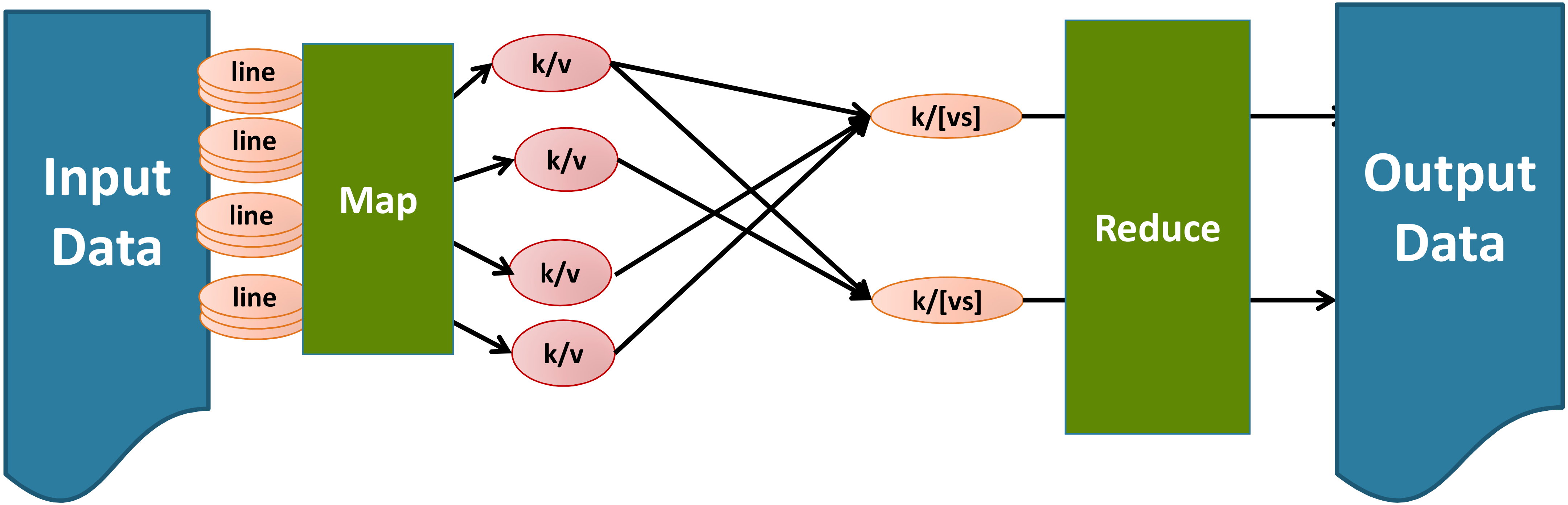
Spark

- Data flow programming model, operating on distributed collections
- Collections are kept in-memory
  - Good support for iterations/interactive queries
- Retain the attractive properties of MapReduce:
  - No references to parallelism in programming logic
  - Fault tolerance (for crashes / stragglers)
  - Data locality
  - Scalability

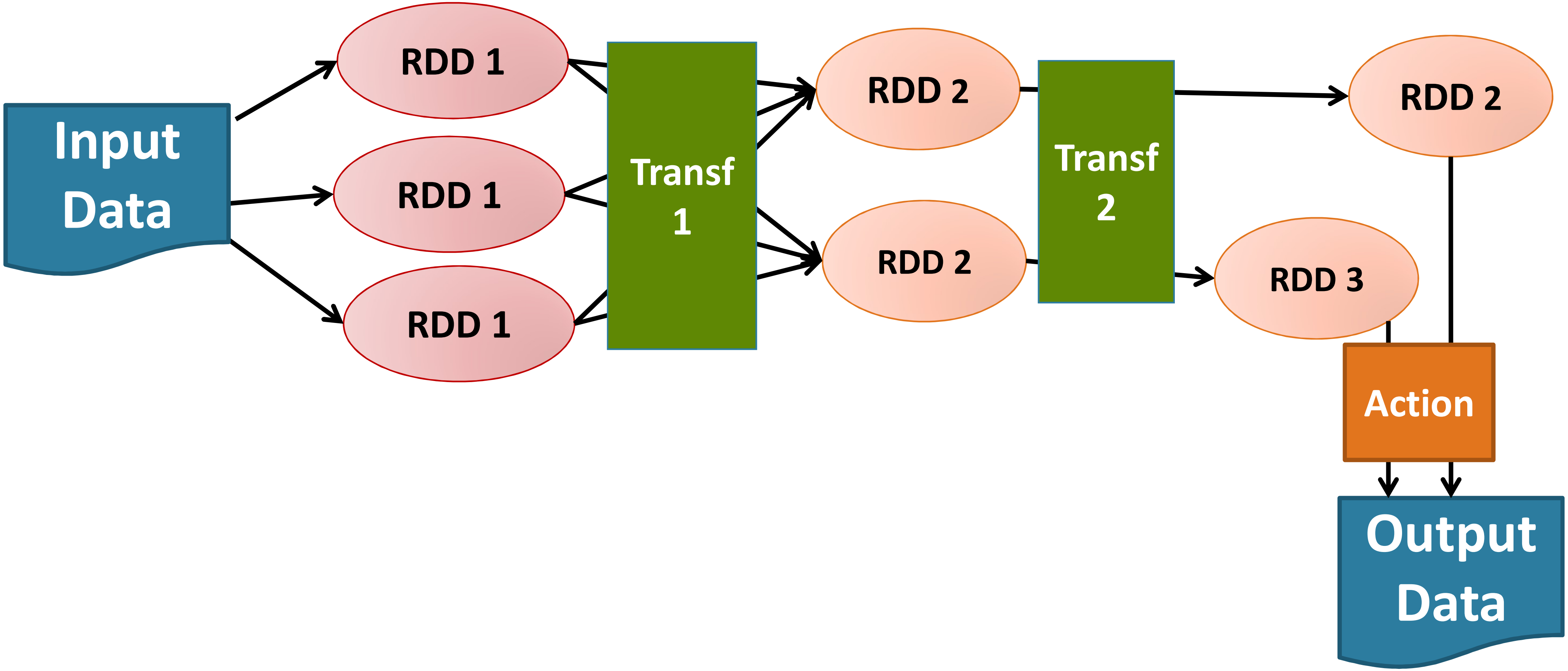
Resillient Distributed Datasets (RDDs)

- Immutable collections **distributed** across the cluster
  - Can be **rebuilt** if a partition is lost
  - Can be ***cached*** across parallel operations
- Immutable: can be transformed into new RDDs as part of the data flow, but no edits
- Created generally by reading data from HDFS
- Can be saved back to HDFS / other programs with actions

Map Reduce Processing Flow



RDD data processing flows



RDD Operations

- 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

Deferred execution

- Spark only executes RDD transformations the moment are needed
- 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 RDD operations

Transformations (define a new RDD from an existing one)	Actions (take an RDD and return a result to driver//HDFS)
map	reduce
filter	collect
sample	count
union	saveAsTextFile
groupByKey	lookupKey
reduceByKey	forEach
join	…
persist	
…	

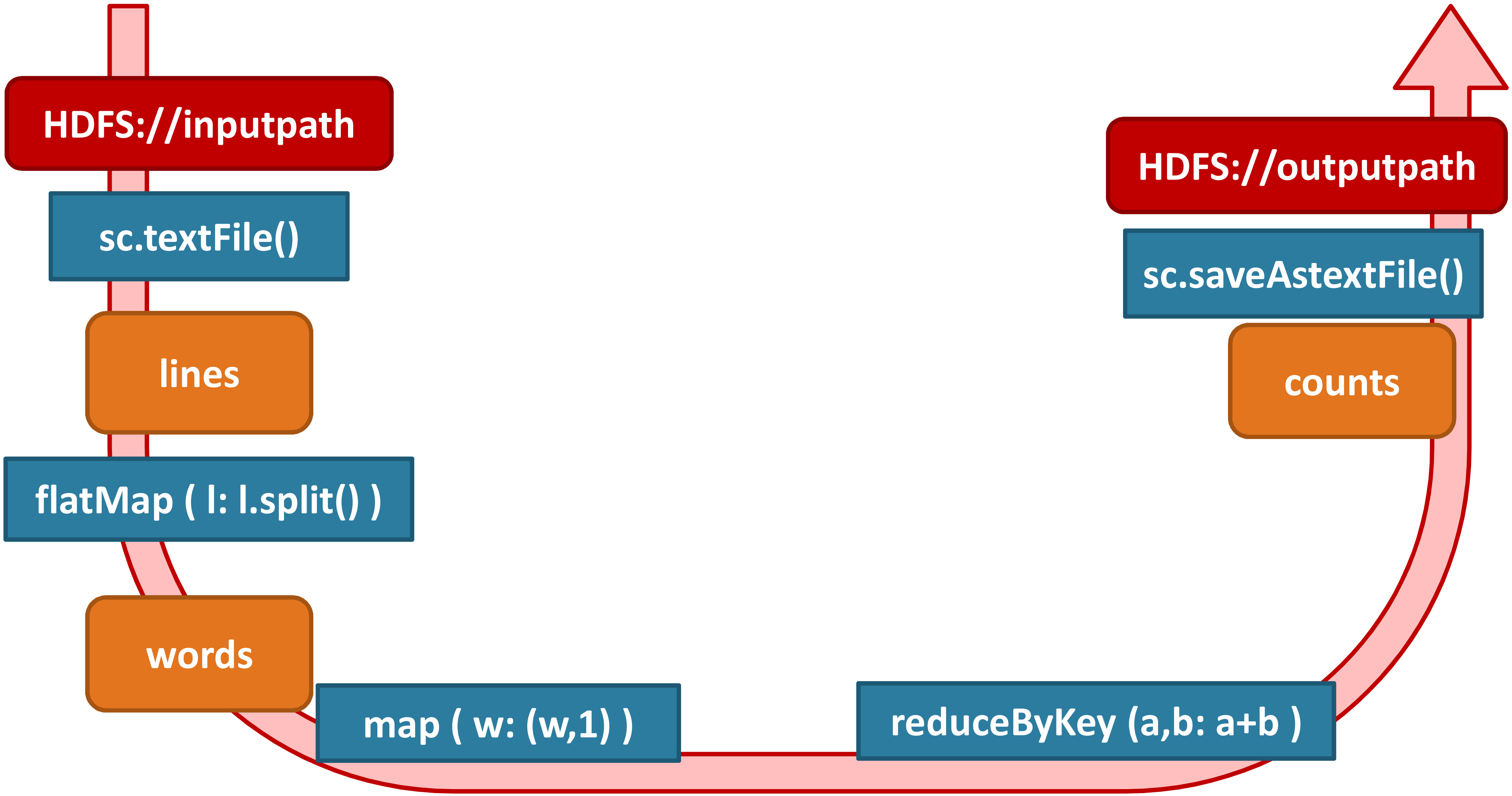
Python notes for Spark

- It is possible to write Spark programs in Java, or Python, but Scala is the native language
- Dynamically typed language: we do not specify types in variable creation
- Tuples of elements (a,b,c) are first order elements.
  - Pairs (2-Tuples) will be very useful to model key-value pair elements
- Python lambda expressions allow us to declare functions
  - `lambda x:  x + 2  // adds 2 to each value`
  - `lambda s:  (s, 1) //creates a tuple with 1 as value`

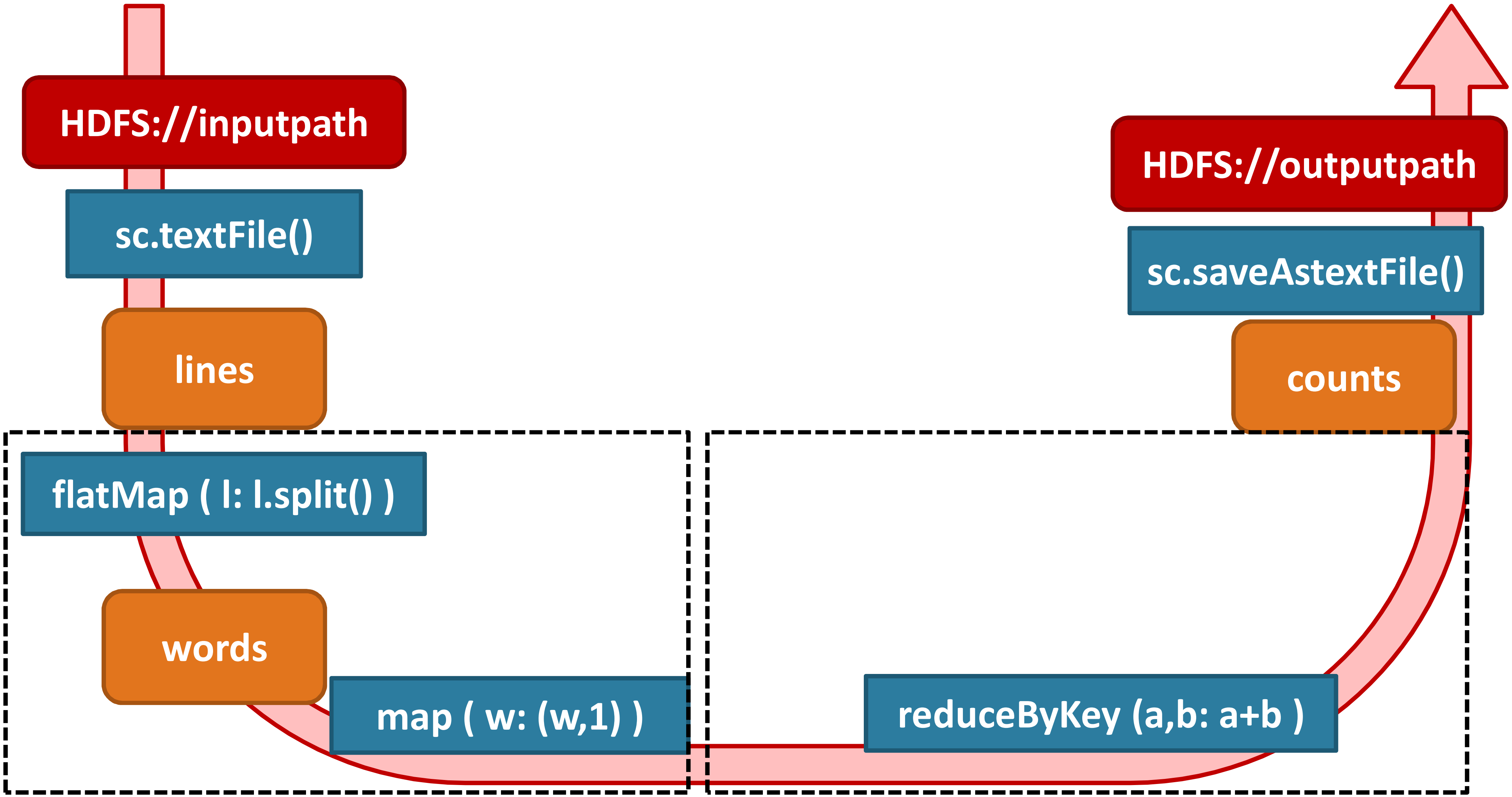
Word Count in Spark (Python code)

```
lines = sc.textFile("/input/path")

words = lines.flatMap(
              lambda lines: lines.split(“ “) )

counts = words.map(lambda word : (word, 1))
              .reduceByKey(lambda a,b : a + b)

counts.saveAsTextFile("/output/path")
```

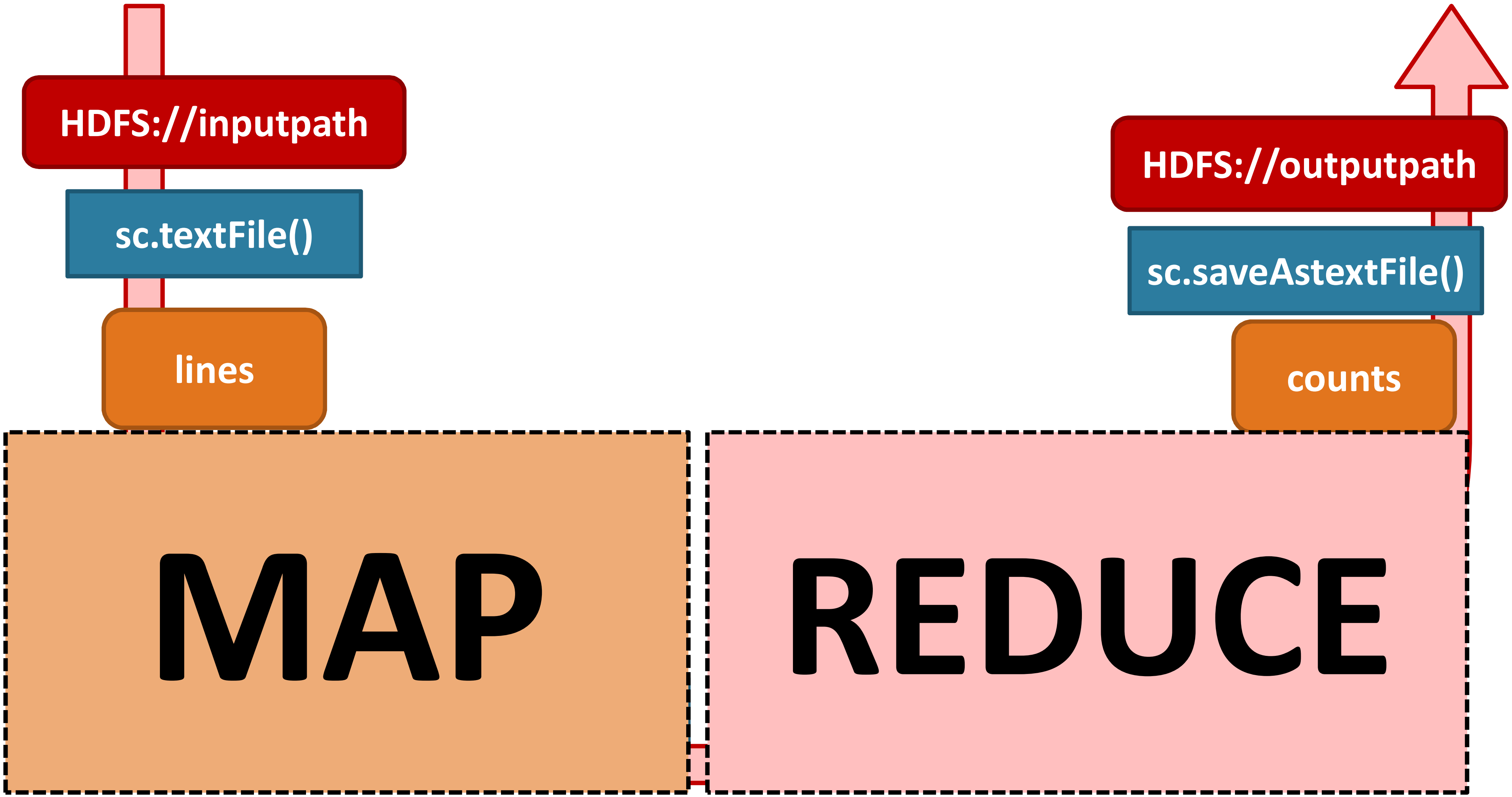
Word Count in Spark (RDD flow)



Word Count in Spark (RDD flow)



Word Count in Spark (RDD flow)



Spark parallelism

- RDDs are split into n **partitions**
  - Partitions might be located in different machines
- Transformations/actions are executed in parallel on each partition
- How many partitions?
  - Default: 1 per HDFS block size when reading from HDFS, can be higher
  - CPU cores process one partition at a time.
  - Number of partitions is automatically computed

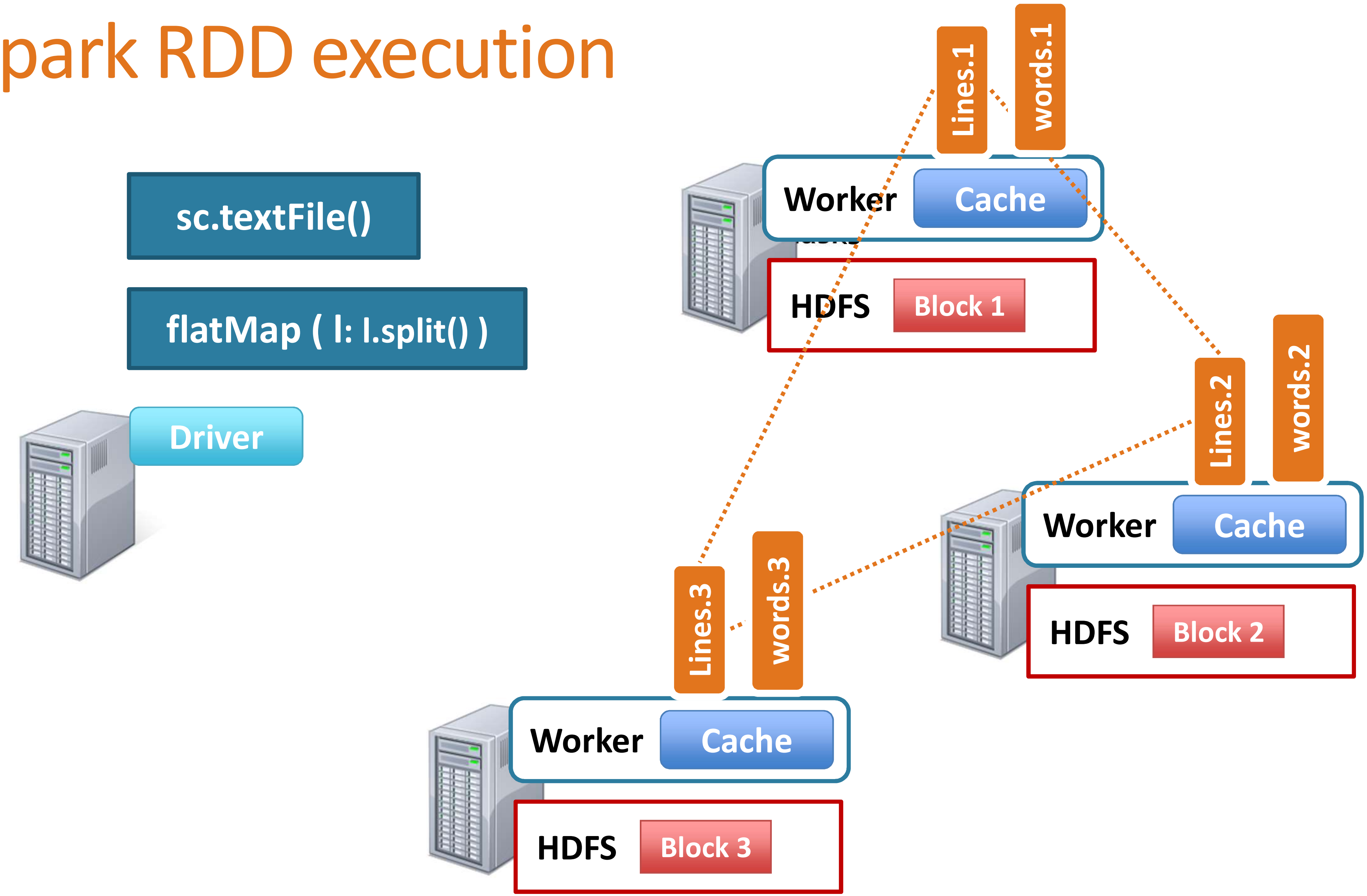
Spark applications

- A Spark application consists of a **driver** program that executes various **parallel** operations on **RDDs** partitioned across the cluster.
- The application is a 'standard' program written in any programming language
- The driver is in a **different machine** of the machines where the RDDs are created
  - Actions are required to retrieve values from the RDDs (e.g. count)

Spark RDD execution

sc.textFile()

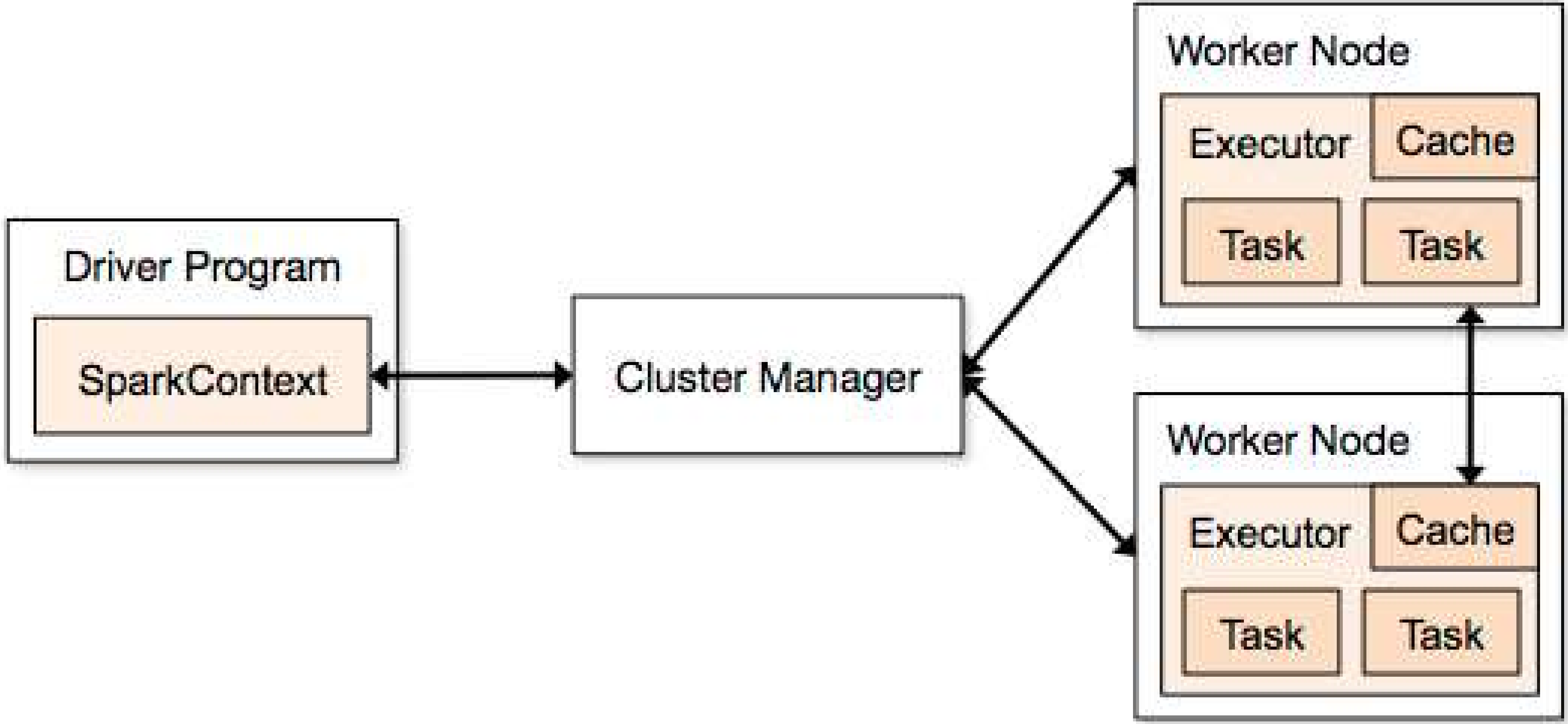
flatMap (l: l.split())



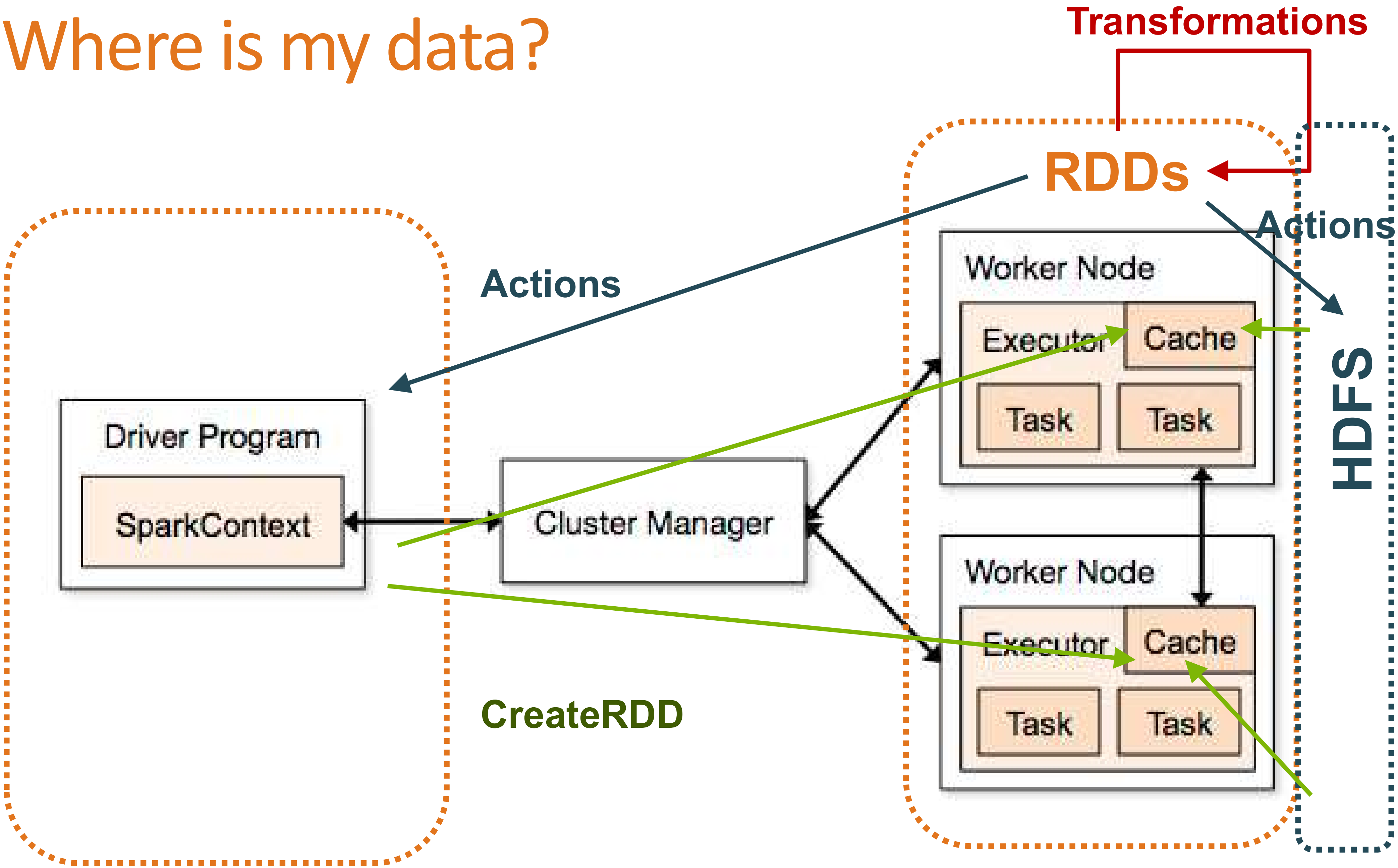
Spark RDD dataflows

- The dataflow consists of **transformations** from one RDD to another
- The initial RDD is **created** from an HDFS input folder, or an existing Scala collection in the driver program.
  - Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations.
- **Actions** allow to retrieve **RDDs** to either HDFS storage, or the memory of the driver program

Spark Execution Architecture



Where is my data?



Creating RDDs

- Any existing collection can be converted to an RDD using parallelize

```
sc.parallelize([1, 2, 3])
```

- RDDs can be created from HDFS input data with sc methods

```
sc.textFile("/hdfspath/to/file")
```

  - The created RDD is a collection of lines
  - Other sc methods for reading SequenceFiles, or any Hadoop compatible InputFormat
  - Analogous RDD actions save to HDFS (e.g. saveasTextFile)

Types of RDD operations

- Two main types of RDD operations
- **Element-wise** operations are applied to each list element independently
  - E.g. `map`, `flatMap`, `filter`
- **Shuffle** operations require to aggregate/collect elements from all the other partitions
  - Equivalent to running one Shuffle in MapReduce
  - E.g. `groupByKey`, `join`, `reduceByKey`
  - Significantly more costly in performance.

Spark Transformations – MapReduce equivalence

map	Map
flatMap	Map
filter	Map
sample	Map
union	Map (2 input)
groupByKey	Shuffle
reduceByKey	ShuffleReduce
join	ShuffleReduce
persist	-----
…	

Spark Element-wise Transformations (I)

- `map`: creates a new RDD with the **same number** of elements, each one is the result of applying the transformation function to it

```
tweet = messages.map( lambda 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

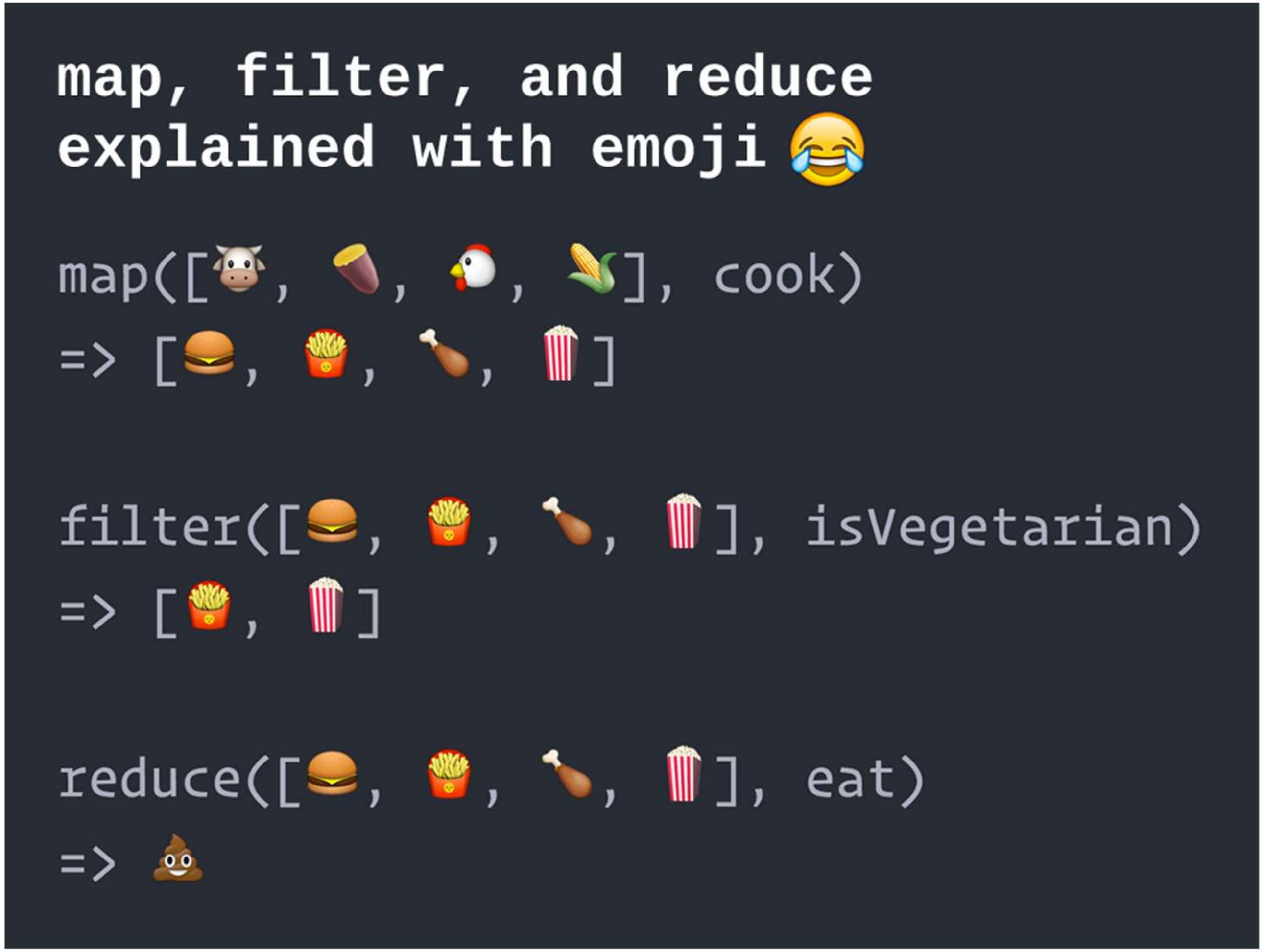
```
grave = logs.filter(lambda x: x.startswith("GRAVE"))
```

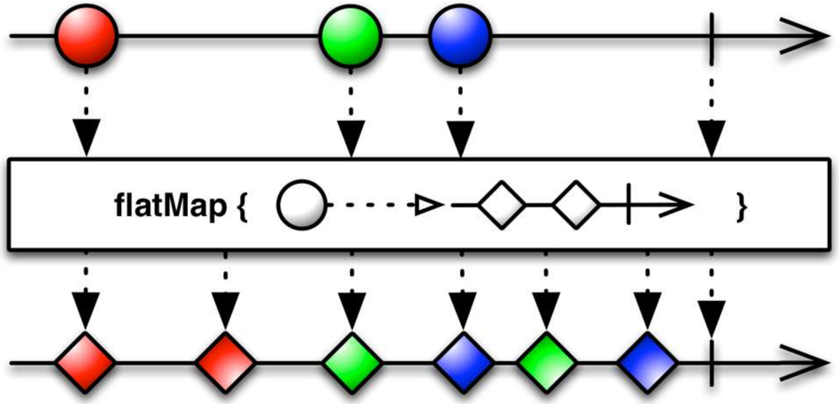
- Both map and filter results have the same partitions as source RDD

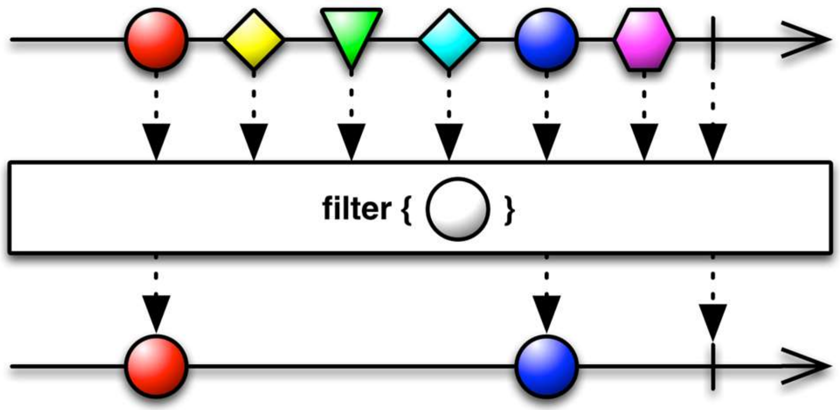
Spark Element-wise Transformations (II)

- `flatMap`: creates a new RDD with a new collection. Each original element generates a variable number of elements when applying the transformation.
  - All elements belong to the same collection (no hierarchy)
  - Same partitions as source RDD
  - Frequently used for item segmentation/ splitting
  - `words = lines.flatMap(lambda x: x.split(" "))`

Visualise map, flatMap, filter







http://www.globalnerdy.com/2016/06/23/map-filter-and-reduce-explained-using-emoji/

Spark RDD Set transformations

- `union`: returns elements contained in either RDD
- `intersection, substraction`: returns elements contained in both RDDs // appearing in the first RDD and not the second
  - Requires shuffle: costly to compute
- `distinct:` returns a set with the unique elements
  - Requires shuffle: costly to compute
- `cartesian`: returns all possible pairs from both sets
  - Requires shuffle: costly to compute
  - Base for performing joins

Spark RDD reduce operations

- `reduce` is an **action**: returns to the driver one **single value** from the RDD
  - Analogous to functional programming.
  - Iteratively applies a binary function

```
list.reduce (lambda a, b: a + b )
[1,2,3,4,5] -> ( (1 + 2 ) + (3 + 4 ) ) + 5  = 15
```

- `reduceByKey` is a **transformation** analogous to MapReduce's Reduce + Combine
  - Reduces values for each key, into a new RDD

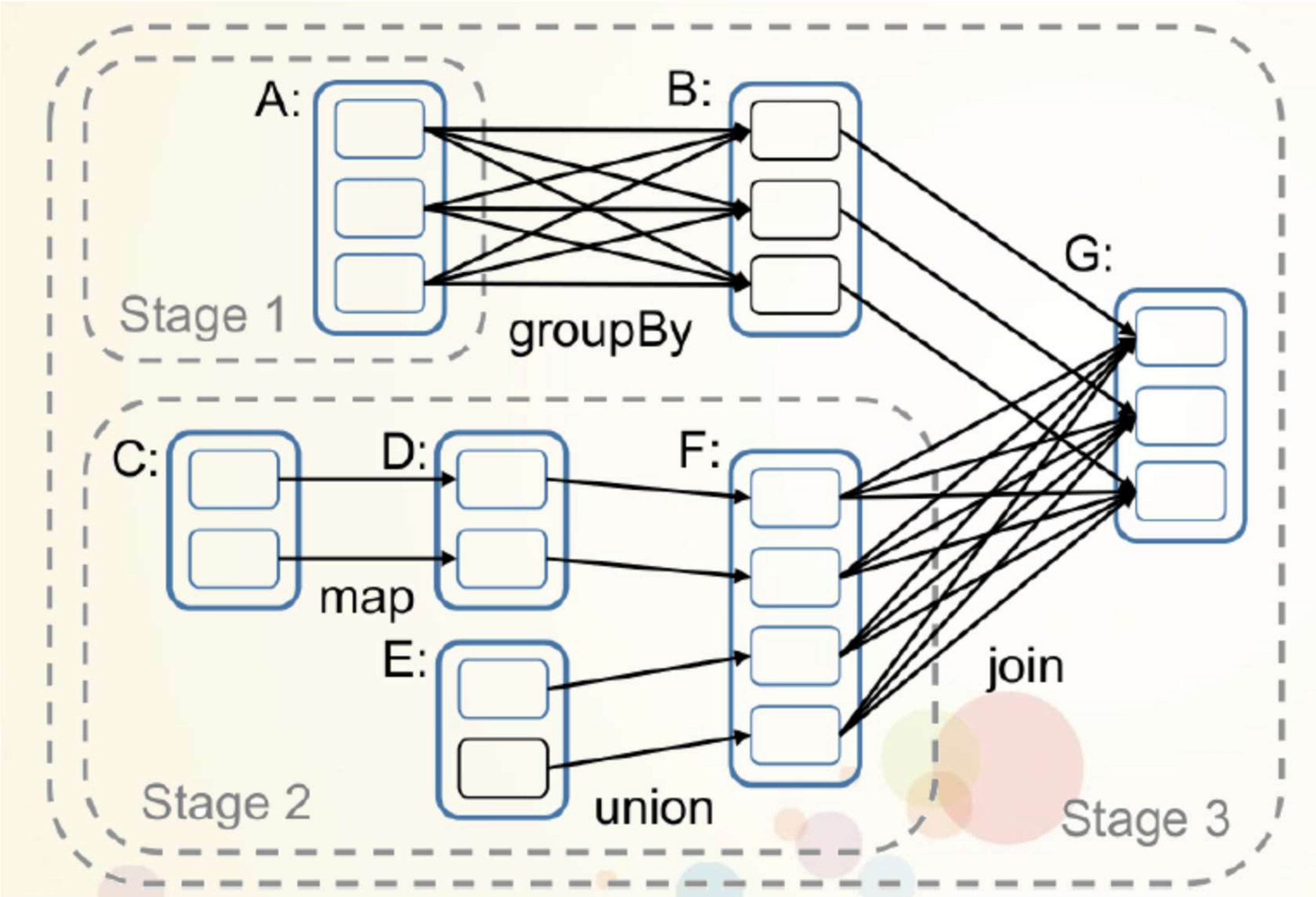
Retrieving information to the driver

- RDDs exist in the cluster, they cannot be read directly
- Actions allow the driver program to retrieve values from the RDDs
  - Useful for algorithms, and interactive applications
- Multiple actions defined for that purpose:
- `count`: returns number of elements
- `takeSample`: returns a sample of elements
- `reduce`: reduces collection to a single value
- `collect`: returns whole RDD to driver
  - **Potential Out Of Memory Errors**! Almost never used

Controlling RDD parallelism

- The number of partitions of an RDD can be explicitly set when creating the RDD for the first time, or through an RDD transformation
- Also possible to specify strategy (Hash, Range,...)
- Transformations that redistribute partitions
  - `coalesce`: collapses partitions into a smaller number. Useful after filter
  - `repartition`: random shuffle into n partitions

RDD Execution & message flows



Grouping RDDs in Spark

- Some RDDs will be lists of key/value pairs
  - Represented by Scala Tuple2 s
  - Easily created with `(k,v)` notation

    `rdd.map(lambda x: (x,1) )`
  - Tuple keys/values are accessed with the `[0]`/`[1]` operator
- Additional transformations/actions are available for RDD tuples
  - Eg `reduceByKey`

Group by transformations

- Group by transformations mirror the shuffling taking place between Map and Reduce jobs
  - RDD must be a collection of pairs of (key,value) elements
- `reduceByKey`: groups together all the values belonging to the same key, computing a reduce function on each
  - Combiner is automatically invoked
  - Almost equivalent to shuffle + Reduce
- `groupByKey`: returns a dataset of (K, Iterable<V>) pairs
  - Equivalent to MapReduce's shuffle
  - If followed by a Map it is equivalent to MapReduce's Reduce

Joins in Spark

- Joins in Spark are implemented for Tuple RDDs
  - Shuffle operation, same as MapReduce repartition joins
- Joins are performed by the tuple keys
- Often requires previous map to set up join keys
- `join`: Performs an Inner Join with another RDD.
  - Other join types also implemented: `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`
- Joins are computed much faster if both RDDs have same partitions and strategy

RDD memory management

- RDDs are not materialised until an action is needed
  - Some might never be, e.g. chaining `map`
- Once the result is obtained, they can be discarded, but might be temporarily held in node cache
- RDDs can always be recreated from a chain of transformations
- For iterative algorithms/ interactive queries, the driver program can explicitly request an RDD to be kept in memory

RDD Persistence

- The `persist` and `cache` methods of RDDs tell Spark to maintain an RDD in memory after its first computation.
  - Future transformations on the same RDD will run much faster.
  - Key tool for iterative algorithms and fast interactive use.
- Multiple persistence options (memory &| disk)
  - By default RDDs persisted only in memory
  - 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

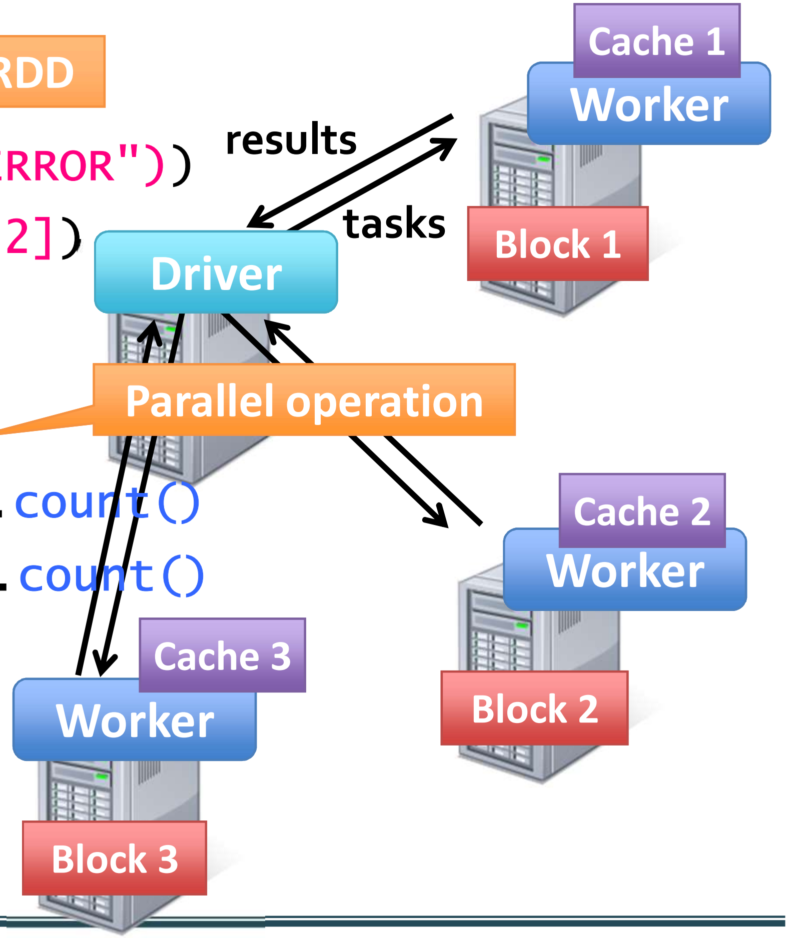
Transformed RDD

```
lines = sc.textFile("/data...")
errors = lines.filter(lambda l: l.startswith("ERROR"))
messages = errors.map(lambda l: l. split("\t")[2])
cachedMsgs = messages.cache()
```

Cached RDD

Parallel operation

```
cachedMsgs.filter(lambda l: l.contains("foo")).count()
cachedMsgs.filter(lambda l: l.contains("bar")).count()
. . .
```



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

Spark execution platform

- Spark runs on multiple cluster execution platforms
  - Apache YARN: integration with the Hadoop resource manager
    - allows Spark and MapReduce to coexist. One resource manager watches for resources for both systems
    - SparkApplicationMaster and MapReduceApplicationMaster control specific jobs
  - Mesos: solution developed also at UC Berkeley, default option. Also supports other frameworks

Logistic Regression Code

```
data = sc.textFile(...).map(readPoint).cache()

w = np.random.rand(D)

for i in range (1,ITERATIONS):
  gradient = data.map(lambda p:
    (1 / (1 + math.exp(-p[1]*(np.dot(w,p[0]))))-
1)*p[1]*p[0]
  ).reduce(lambda a, b: a + b)
  w -= gradient
}

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

Numeric RDD operations

- When the type of the elements of an RDD is numeric (eg Integer or Double), Spark also provides aggregated summarisation methods on the rdd
- `mean`, `sum`, `max`, `min`, `variance`, `stddev`

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/latest/tuning.html

Spark explicit data partitioning

- One possible approach to improve performance of Spark is to exert explicit control in how data is partitioned
  - partitionBy transformation, selecting partitioning strategy and number of operations
- Can substantially speedup groupBy operations, by having the datasets already distributed in the same destination nodes where the grouping takes place.

Spark Dataframes

- R-like interface for operating with large datasets
- More limited API than RDDs
- Better performance, as it is possible to optimize planning thanks to more predictability

Spark ecosystem

- Spark Dataframes
- 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