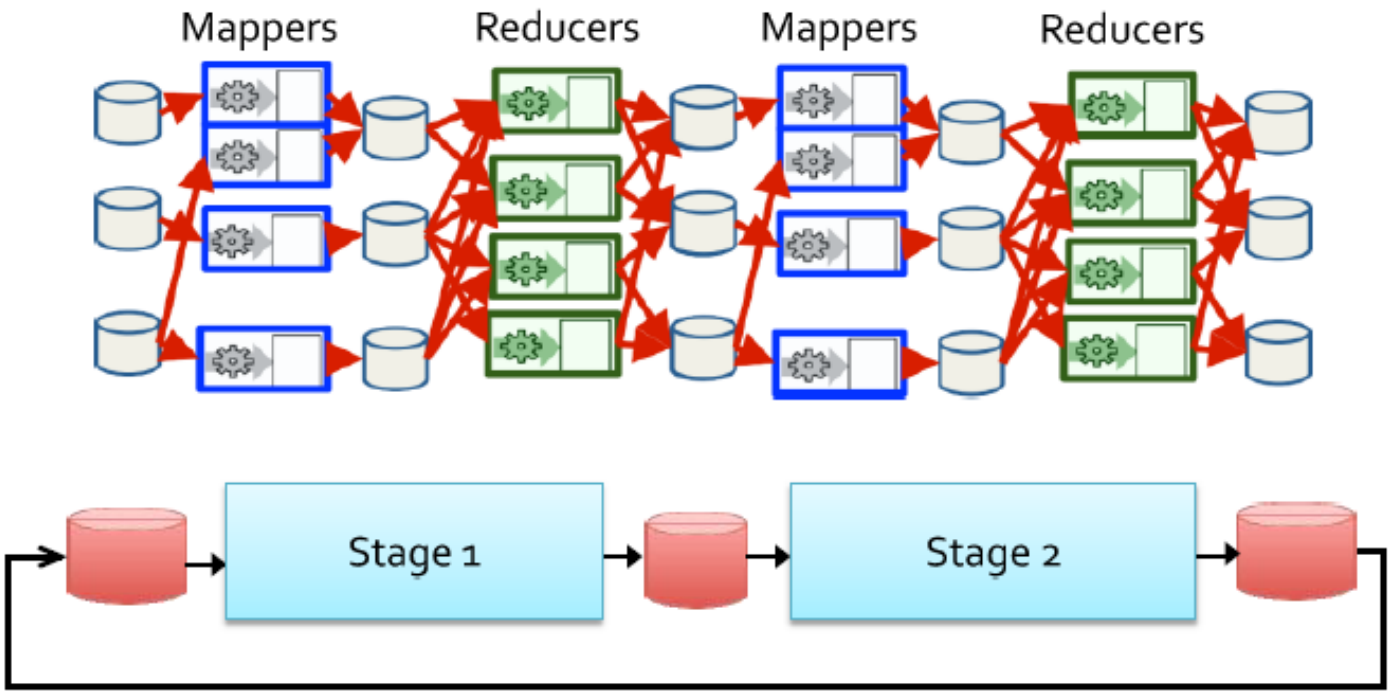
Large Scale Data Processing Frameworks: Spark

Exploiting distributed memory for iterative and interactive tasks

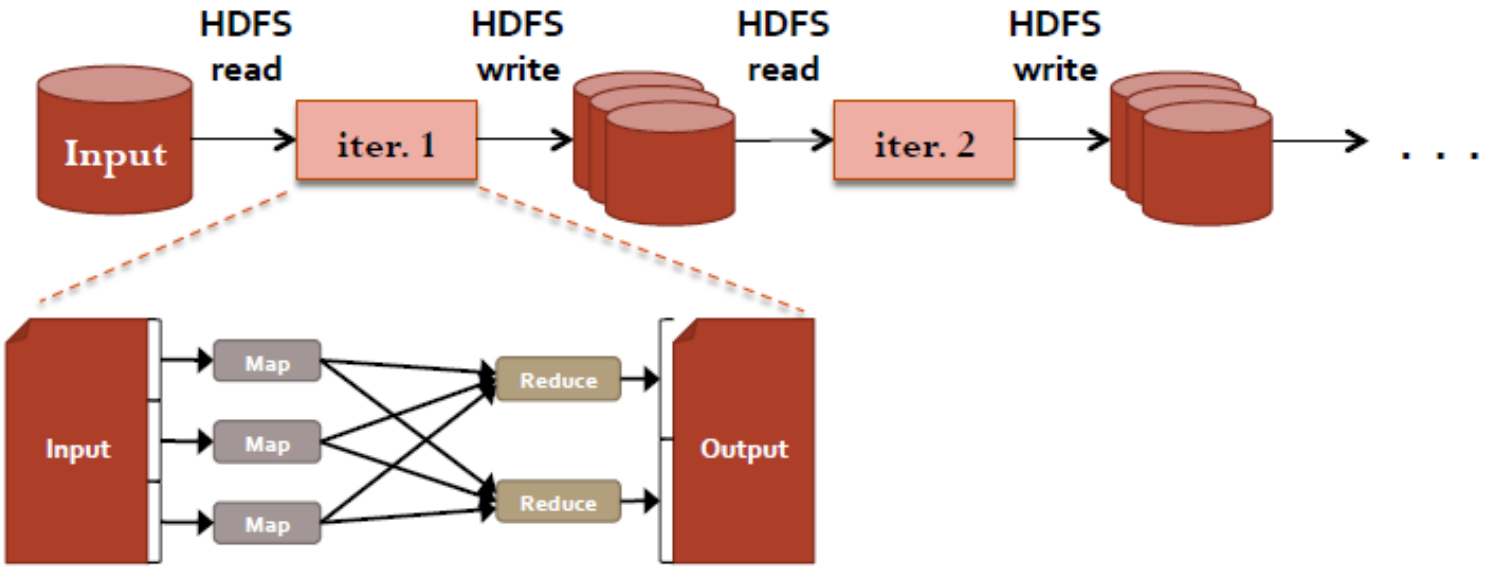
Introduction & Motivation

MapReduce limitations

- Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage



MapReduce limitations



- Slow due to disk I/O, high communication, and serialization
- Inefficient for:
  - **Iterative** algorithms (Machine Learning, Graphs & Network Analysis)
  - **Interactive** Data Mining (R, Excel-like computations, Ad hoc Reporting,Searching)

MapReduce limitations

- **Programming model**
  - Hard to implement everything as a MR program
  - Multiple MR steps can be needed also for simple operations
  - Lack of control structures and data types
- **Efficiency**
  - High communication cost
  - Frequent writing of output to disk
  - Limited exploitation of main memory
- **Real-time processing**
  - A MR job requires to scan the entire input
  - Stream processing and random access impossible

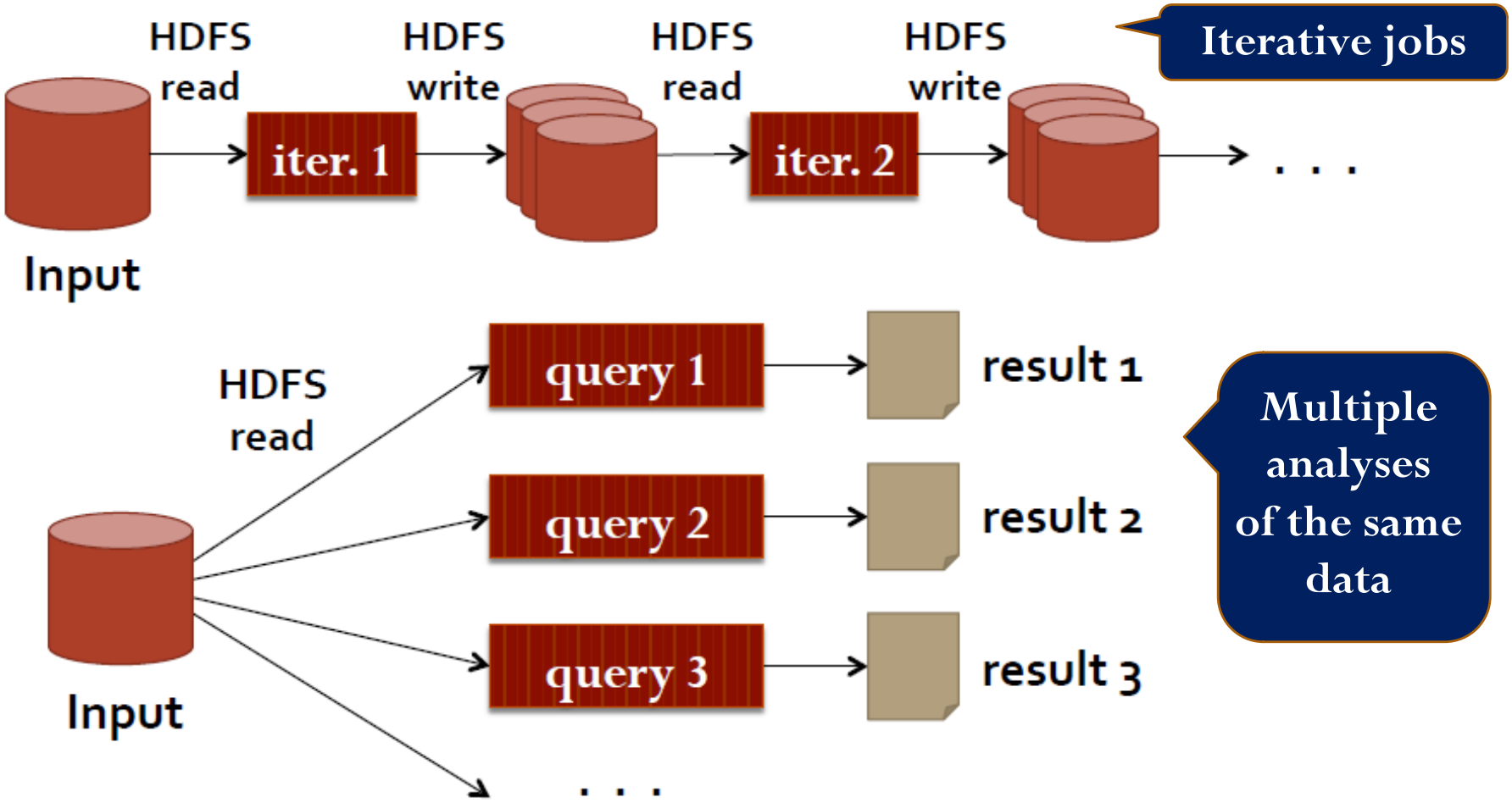
Need for a new programming model?

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage applications (e.g. iterative graph algorithms and machine learning)
  - More efficiency
  - More interactive ad-hoc queries
  - Both multi-stage and interactive applications require faster data sharing across parallel jobs

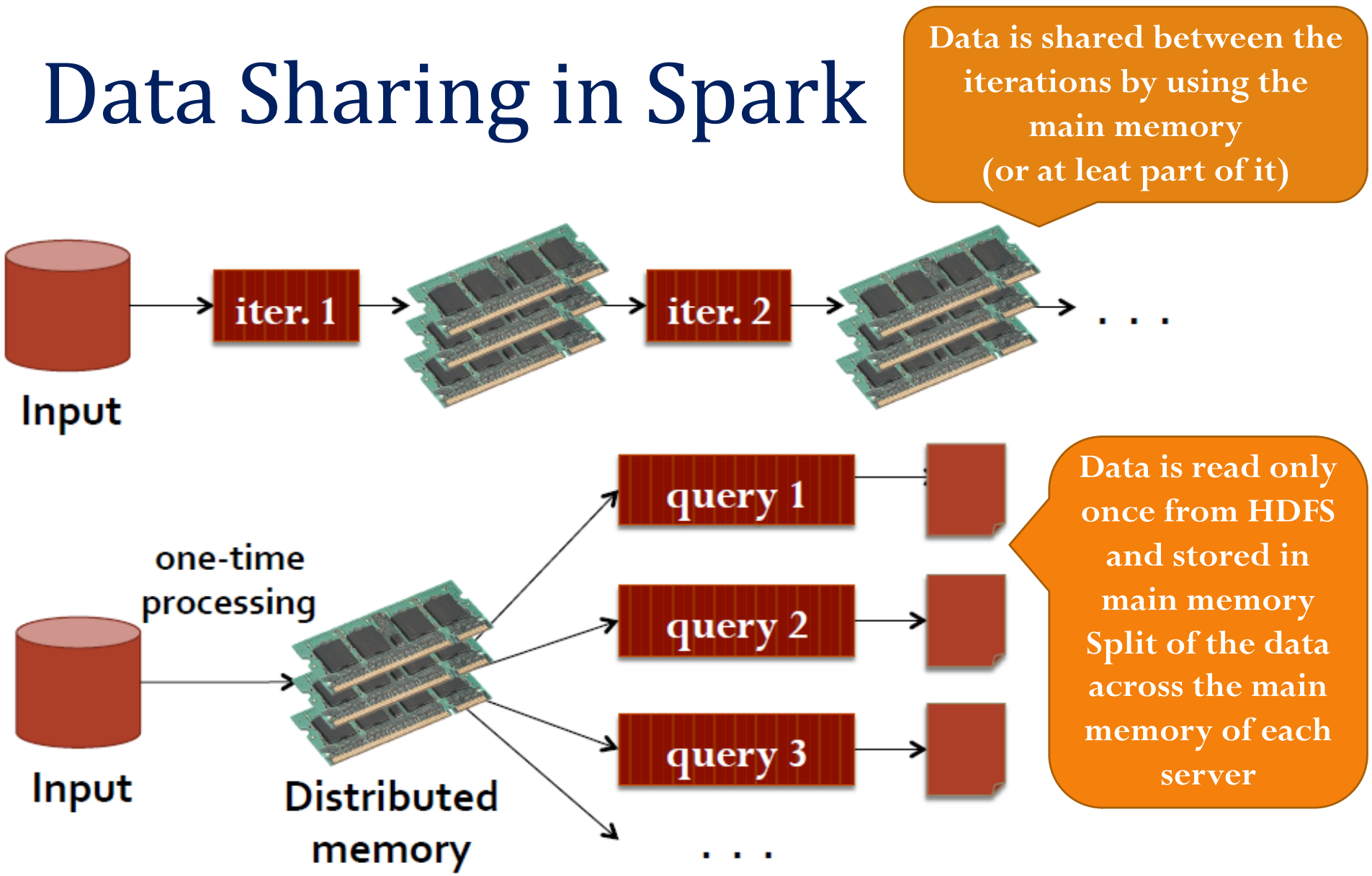
Motivation and Opportunity

- **Motivation**
  - Using MapReduce for **complex iterative jobs** or **multiple jobs on the same data** involves lots pf disk I/O
- **Opportunity**
  - The **cost of main memory decreased**
  - Hence, large main memories are available in each server
- **Solution**
  - **Keep more data in main memory**
  - Enabling data sharing in main memory as a resource of the cluster

Data Sharing in Map Reduce

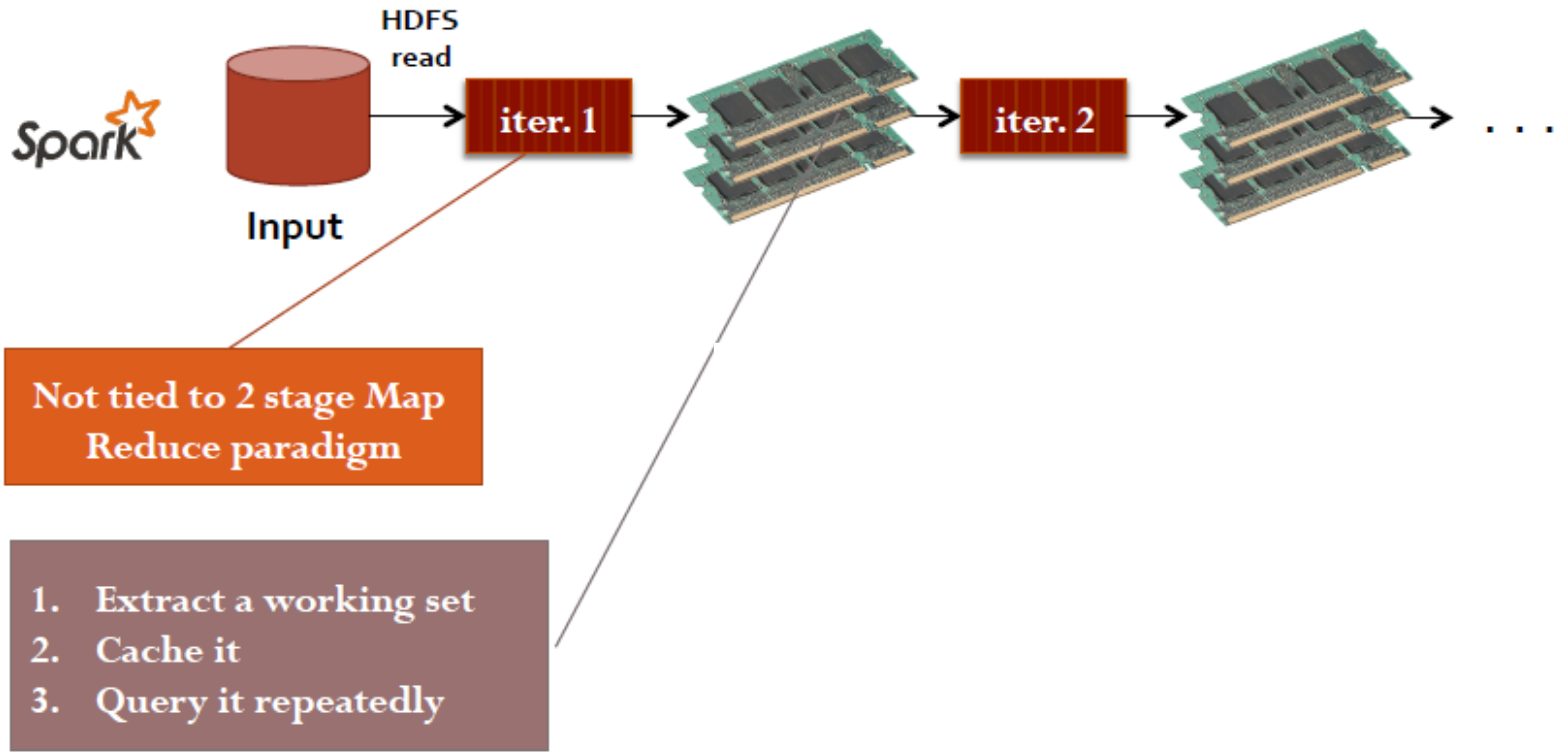


Data Sharing in Spark



10-100x faster than network and disk

Spark Data Flow



Spark Data Flow

- Everything you can do in Hadoop, you can also do in Spark
  - But it relies on Hadoop (HDFS) for storage
- Spark's computation paradigm is not "just" a single MapReduce job, but a **multi-stage, in-memory dataflow graph** based on **Resilient Distributed Datasets (RDDs)**

- Data are represented as RDDs
  - Partitioned/distributed collections of objects spread across the nodes of a cluster
  - Stored in main memory (when it is possible) or on local disk
- Spark programs are written in terms of operations on RDDs

Spark Computing Framework

- Provides a programming abstraction (based on RDDs) and transparent mechanisms to execute code in parallel on RDDs
  - Manages job scheduling and synchronization
  - Manages the split of RDDs in partition and allocates RDDs' partitions in the nodes of the cluster
  - Hides complexities of fault-tolerance and slow machines
    - RDDs are automatically rebuilt in case of machine failure

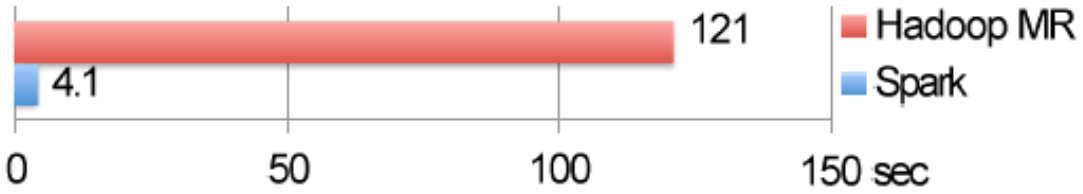
Map Reduce vs Spark

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc…
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, R, and Python

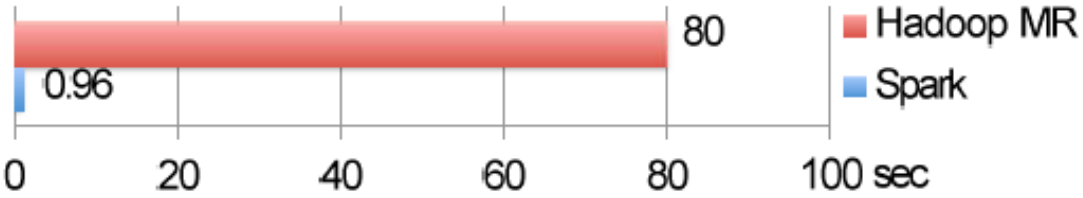
- Lower overhead for starting jobs
- Less expensive shuffles

In Memory RDDs can make a big difference

- Two iterative Machine Learning algorithms:
  - K-means Clustering



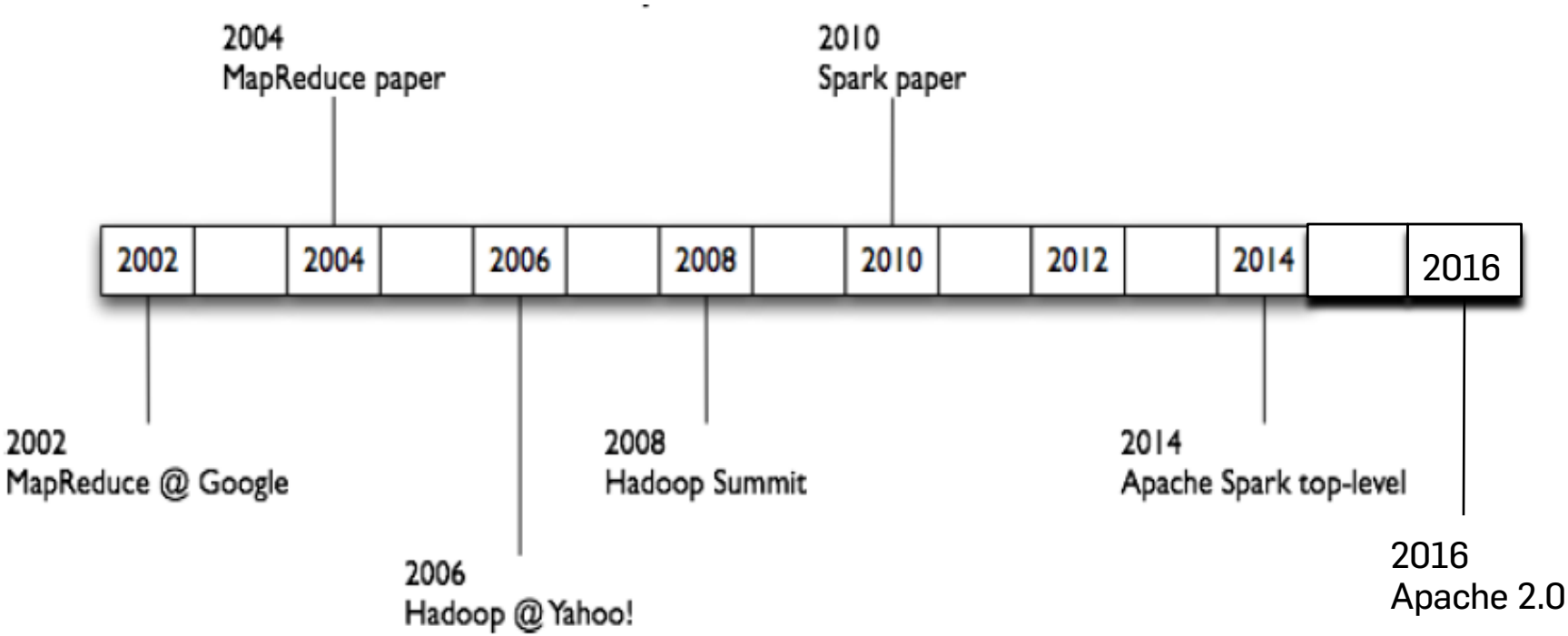
  - Logistic Regression



Spark Main Components

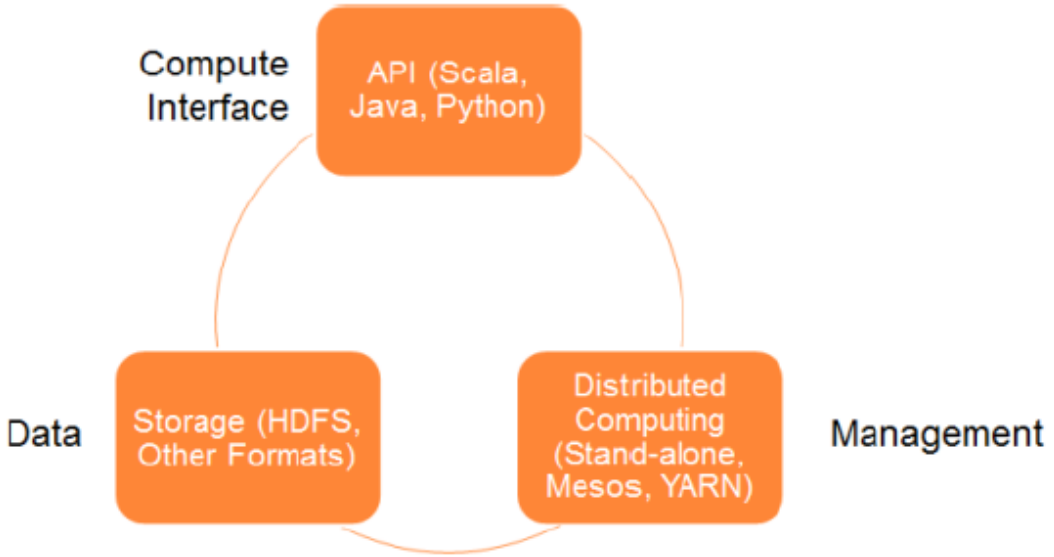
Spark History

- Originally developed at UC Berkley AMP Lab
- Project started in 2009, open sourced in 2010, donated to Apache Software Foundation in 2013, top-level project since 2014
- Spark 2.0.0 July 2016, Current: 2.4.4 – November 2019 Preview release of Spark 3.0



Spark Components

An API with core features and extended modules for data manipulation in different languages: Java, Scala, Python, R, SQL

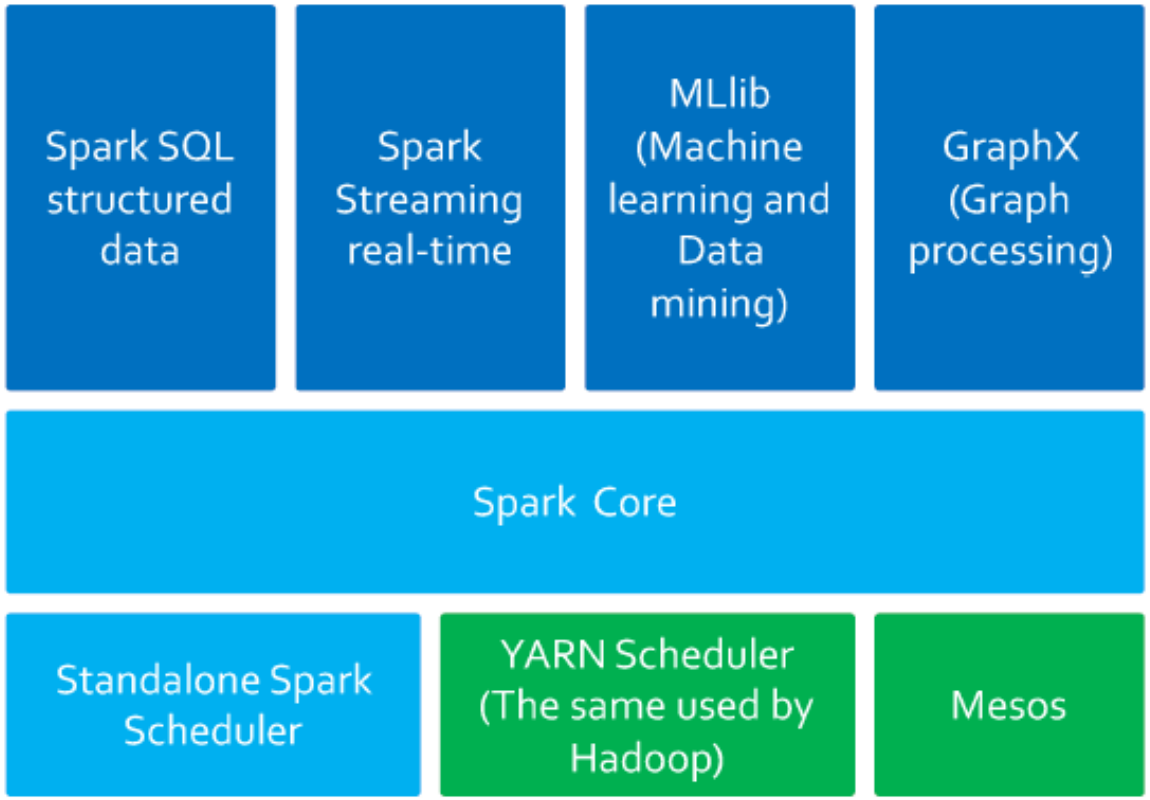


Spark supports any Hadoop-ready storage source:

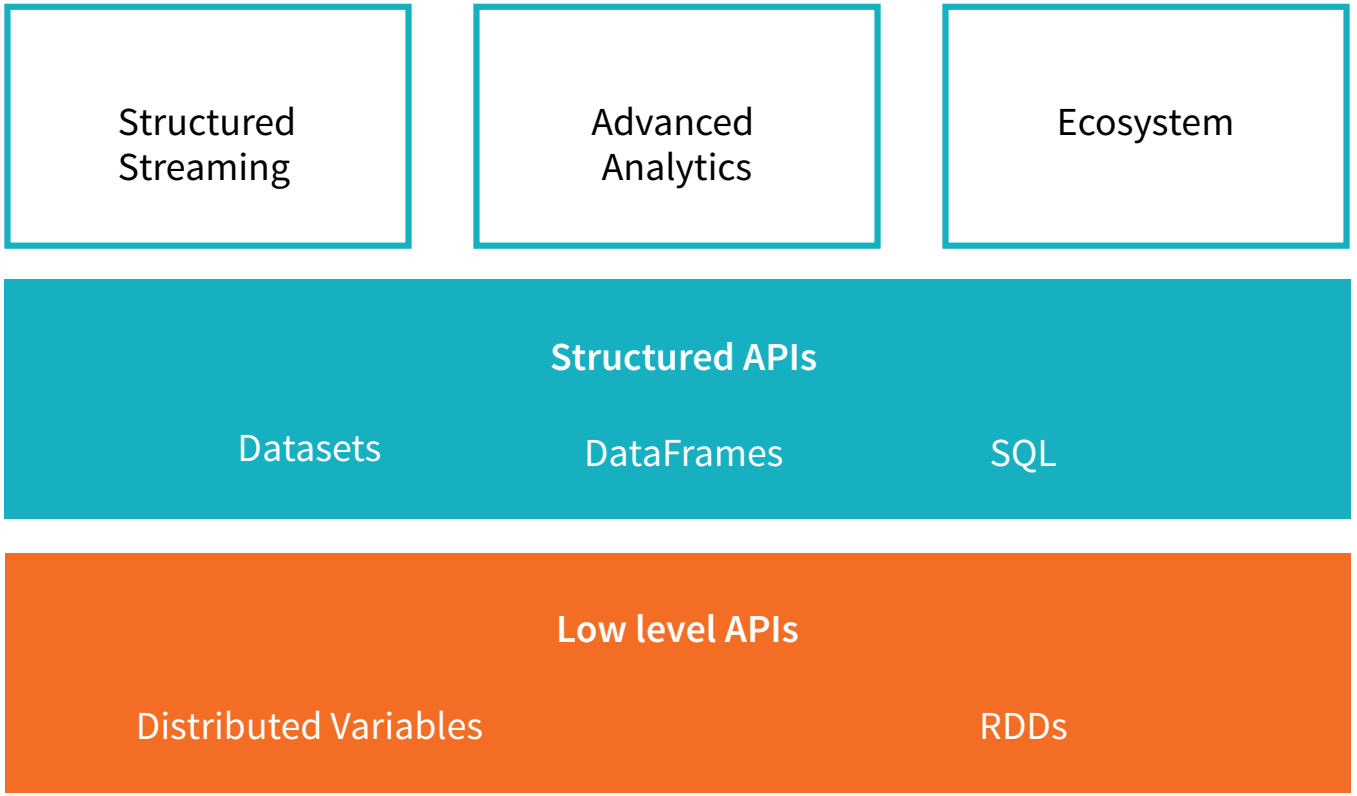
- local file system,
- HDFS, HBase, Amazon S3, Cassandra, …

Spark provides a stand-alone cluster manager, but both Apache YARN and Apache Mesos are also supported

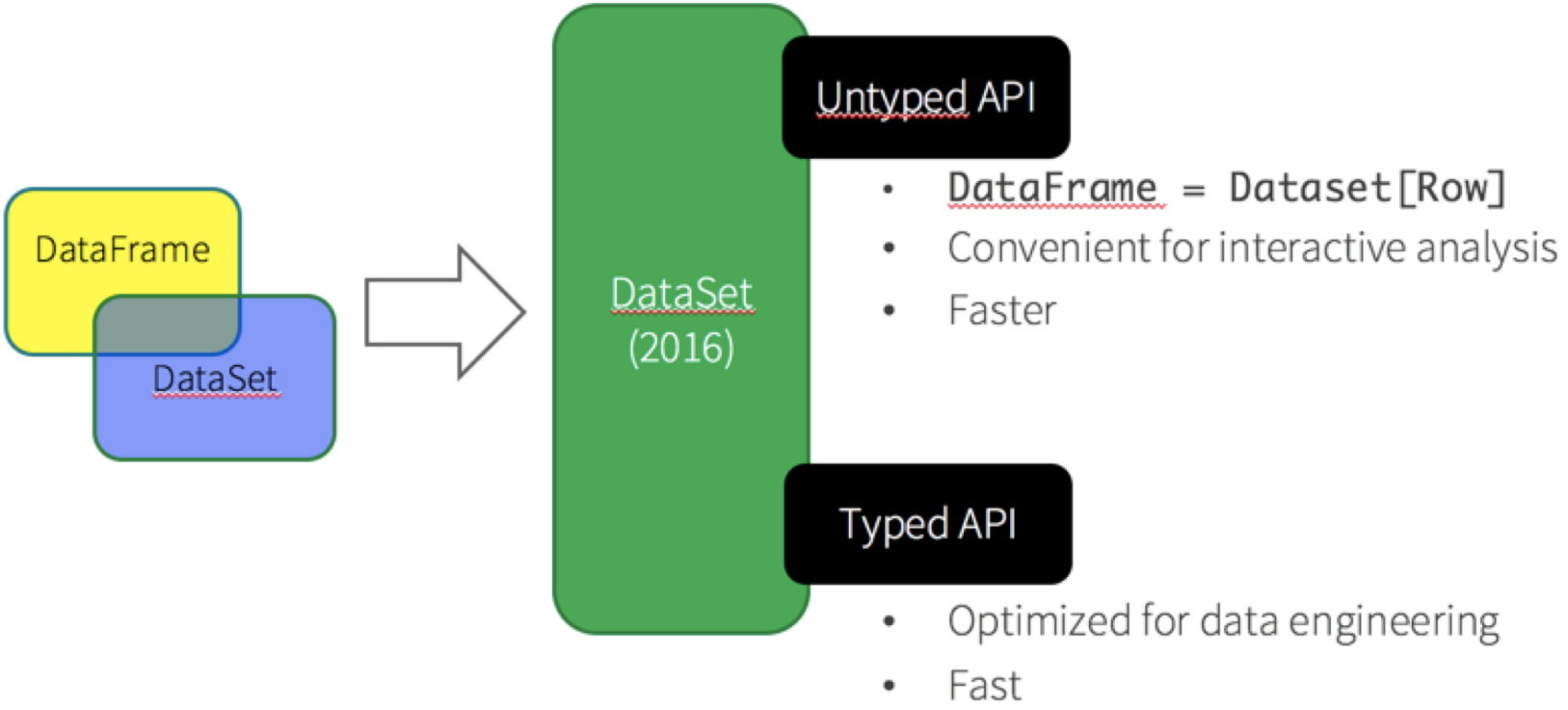
Spark: Components



Spark Components: Revisited



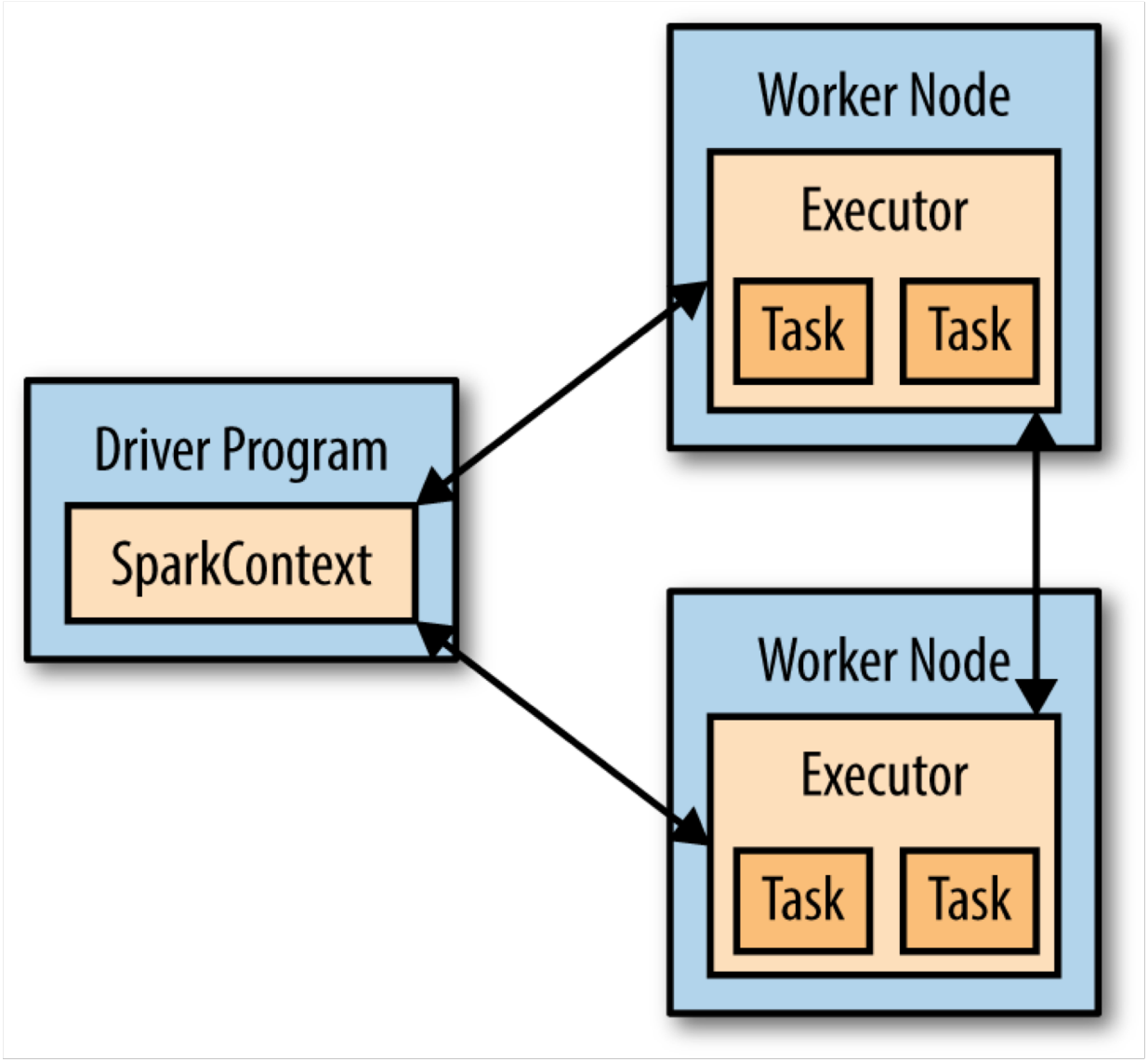
Spark 2.0 API (2016)



Spark Components

- Spark Core
  - Contains the basic functionalities of Spark exploited by all components
    - Task scheduling
    - Memory management
    - Fault recovery
    - …
  - Provides the API that are used to create RDDs and apply transformations and actions on them

Executing Spark



RDDs

RDDs

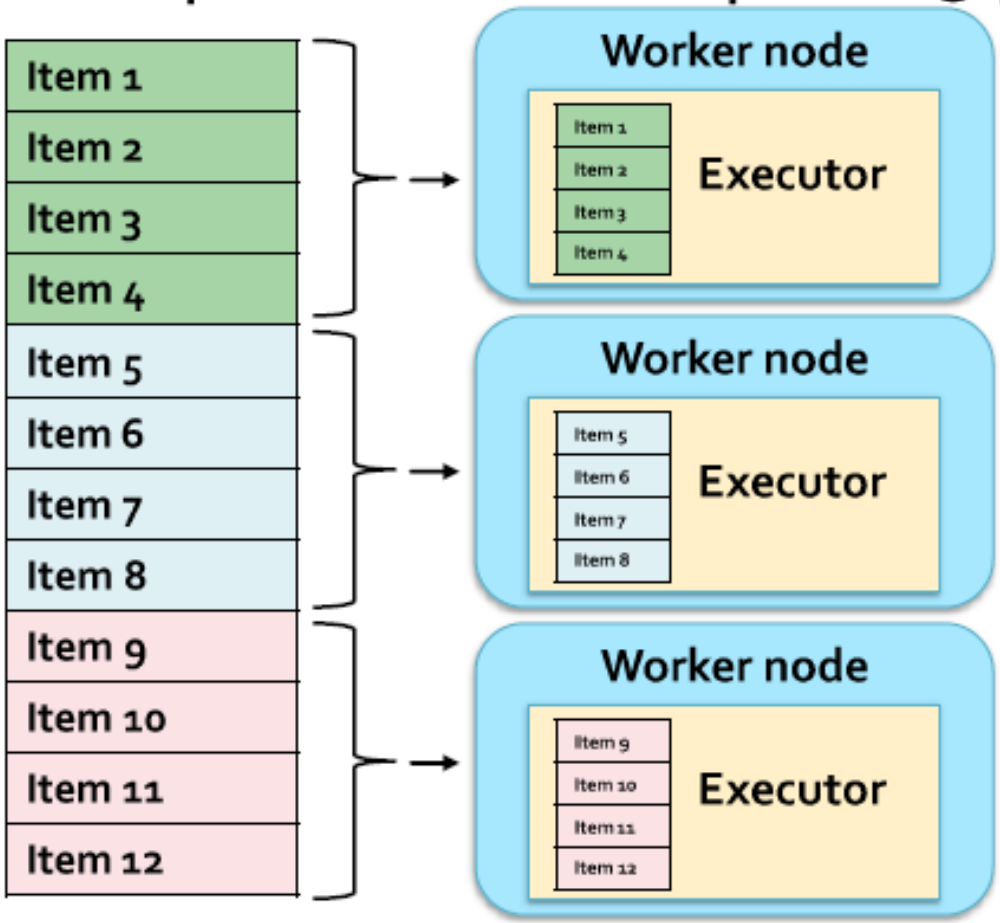
- RDDs are the primary abstraction in Spark
- RDDs are distributed collections of objects spread across the nodes of a cluster
  - They are split in partitions
  - Each node of the cluster that is used to run an application contains at least one partition of the RDD(s) that is (are) defined in the application

RDDs

- RDDs are stored in the main memory of the executor running in the node of the clusters (when it is possible) or on the local disk of the nodes if there is not enough main memory
- RDDs allow executing in parallel the code invoked on them
  - Each executor of a worker node runs the specified code on its partition of the RDD

RDDs

- Example of an RDD split in 3 partitions

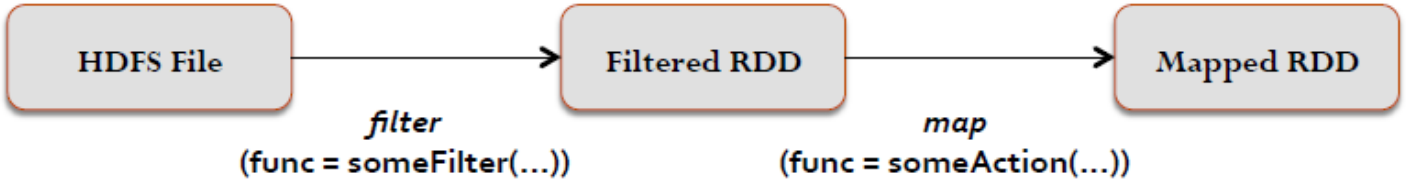


more partitions
=
more parallelism

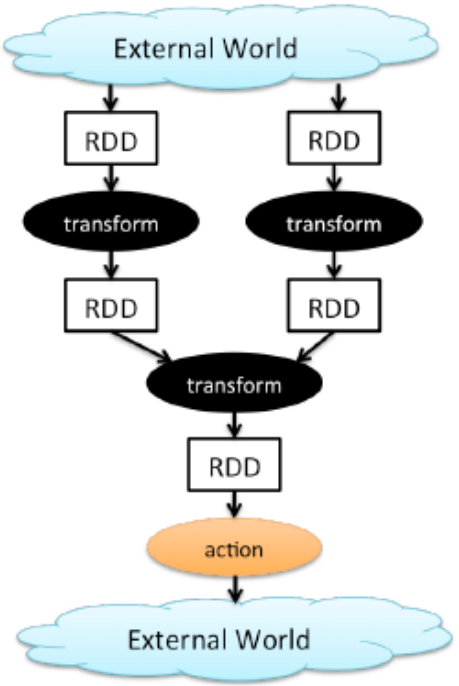
RDDs & Data Flow Graph

- RDDs are immutable once constructed
  - i.e., the content of an RDD cannot be modified
- Track lineage information to efficiently recompute lost data
  - i.e., for each RDD, Spark knows how it has been constructed and it can rebuild it if a failure occurs
  - this information is represented by means of a DAG connecting input data and RDDs

Operators over RDDs



General DAG of operators (e.g. map-reduce-reduce or even more complex combinations)



Operators over RDDs

RDDs - Creation

- RDDs can be created
  - **By parallelizing existing collections** of the hosting programming language (e.g., collections and lists of Scala, Java, Python, or R)
    - the number of partitions is specified by the user
  - **From (large) files stored in HDFS** or any other file system
    - there is one partition per HDFS block
  - **By transforming an existing RDD**
    - the number of partitions depends on the type of transformation

Operators over RDDs

- Programmer can perform 3 kinds of operations

Transformations

- Create a new dataset from and existing one.
- Lazy in nature. They are executed only when some action is performed.
- Example :
  - Map(func)
  - Filter(func)
  - Distinct()

Actions

- Returns to the driver program a value or exports data to a storage system after performing a computation.
- Example:
  - Count()
  - Reduce(funct)
  - Collect
  - Take()

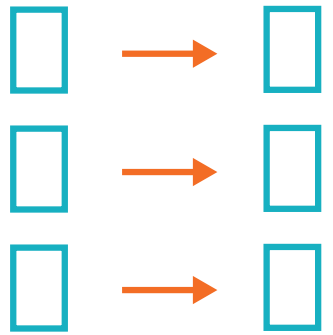
Persistence

- For caching datasets in-memory for future operations.
- Option to store on disk or RAM or mixed (Storage Level).
- Example:
  - Persist()
  - Cache()

Transformations

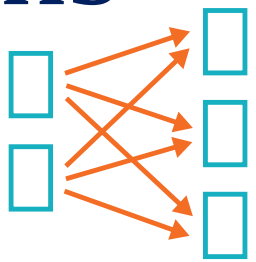
- Transformations are lazy operations on a RDD that return RDD objects or collections of RDD
  - e.g. map, filter, join, cogroup, etc.
- Transformations are lazy and are not executed immediately, but only after an action has been executed

Two kinds of transformations



- **Narrow transformations**
  - They are the result of map, filter and such that **the result is from the data from a single partition only**, i.e., it is self-sustained
  - An output RDD has partitions with records that originate from a single partition in the parent RDD
  - Spark groups narrow transformations as a **stage**

Two kinds of transformations



- **Wide transformations**
  - They are the result of groupByKey and reduceByKey
  - **The data required to compute the records in a single partition may reside in many partitions of the parent RDD**
  - All of the tuples with the same key must end up in the same partition, processed by the same task
  - Spark must execute RDD **shuffle**, which transfers data across cluster and results in a new stage with a new set of partitions

Main transformations

- filter
- map, flatMap
- sample
- union
- intersection
- distinct
- groupByKey
- reduceByKey
- sortByKey
- join
- cogroup
- cartesian

Actions

- Actions are synchronous operations that **return values**
- Any RDD operation that returns a value of any type but an RDD is an action
  - e.g., SaveAs, Collect, Take, Reduce…
- Actions **trigger execution of RDD transformations** to return values
- **Until no action is fired, the data to be processed is not even accessed**
- Only actions can materialize the entire process with real data
  - Actions cause data retrieval and execution of all transformations on RDDs
  - Actions cause data to be returned to driver or saved to output

Main actions

- reduce
- collect
- count
- first
- take
- takeSample
- takeOrdered
- saveAsTextFile
- saveAsSequenceFile
- foreach

Persistence

- By default, each transformed RDD is recomputed each time you run an action on it, unless you specify the RDD to be cached in memory
- RDD can be persisted on disks as well
- Caching is the key tool for iterative algorithms
- Using persist(), one can specify the Storage Level for persisting an RDD

Persistence

- caching allows us to avoid iterative recomputation of data:

```
rdd.persist(StorageLevel.MEMORY_ONLY);
// tells Spark to store the next computation in memory
rdd.count();
rdd.count(); // takes over result from last operation
```

- RDD persistence is especially useful for iterative algorithms and fast interactive use
- Fault-tolerant: if any partition of an RDD is lost, it is recomputed by applying the same transformations that created it
- `unpersist()` method allows to manually remove objects from cache, being handled by the Java garbage collector

RDD storage levels

Storage Level	Meaning
`MEMORY_ONLY`	Store RDD as *deserialized* Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as *deserialized* Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
`MEMORY_ONLY_SER`	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
`MEMORY_AND_DISK_SER`	Similar to `MEMORY_ONLY_SER`, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
`DISK_ONLY`	Store the RDD partitions only on disk.

…

(See http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence)

Spark Programs

Common Spark Use Cases

- **Spark Context & RDD Creation**
  - Connecting and loading data into the Spark context
- **Basic RDDs**
  - Unary RDD Transformations & Actions
  - Binary RDD Transformations & Actions
- **Key-Value Pairs: PairRDDs**
  - Unary PairRDD Transformations & Actions
  - Binary PairRDD Transformations & Actions

See: [1] "**Learning Spark**" book, Chapters 3 & 4

For a synopsis transformations & actions: see Martin Theobald's slide excerpt

Supported languages

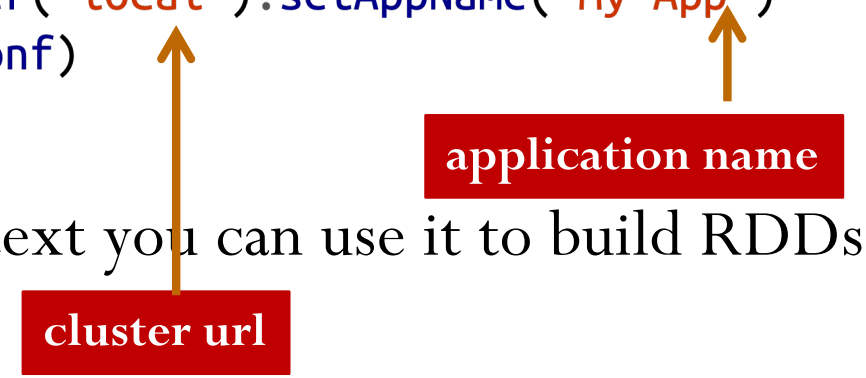
- Spark supports
  - Scala (the same language that is used to develop the Spark framework and all its components)
  - Java
  - Python 
  - R

Spark Context

- SparkContext is the basic entry point to the Spark runtime environment and underlying file system (such as HDFS)
- SparkContext represents a connection to a computing cluster

```
>>> sc
<pyspark.context.SparkContext object at 0x1025b8f90>
```

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```



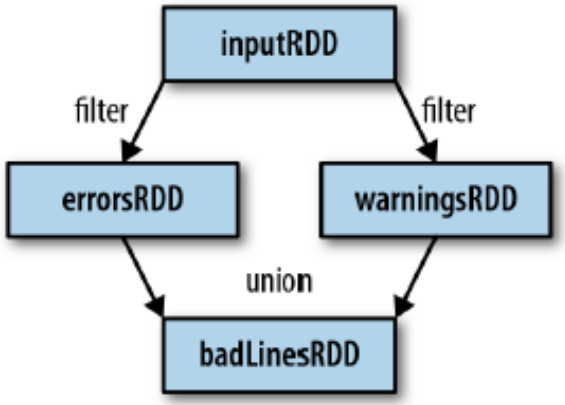
- Once you have a SparkContext you can use it to build RDDs

RDD Transformations

- Transformations take one RDD as input and return another RDD as output:

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- They define the logical structure of the dataflow, but they do not actually trigger any computation



Passing functions and lambda's

- Java (either anonymous inner classes or named classes)

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
  new Function<String, Boolean>() {
    public Boolean call(String x) { return x.contains("error"); }
  }
});
```

Table 3-1. Standard Java function interfaces

Function name	Method to implement	Usage
`Function<T, R>`	`R call(T)`	Take in one input and return one output, for use with operations like `map()` and `filter()`.

- Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

- Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- Java 8 (lambda expressions)

```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

RDD Actions

- Actions take an RDD object as input and perform a final operation to obtain a non-distributed, either mutable (e.g., a list) or immutable (e.g., a file) object as output

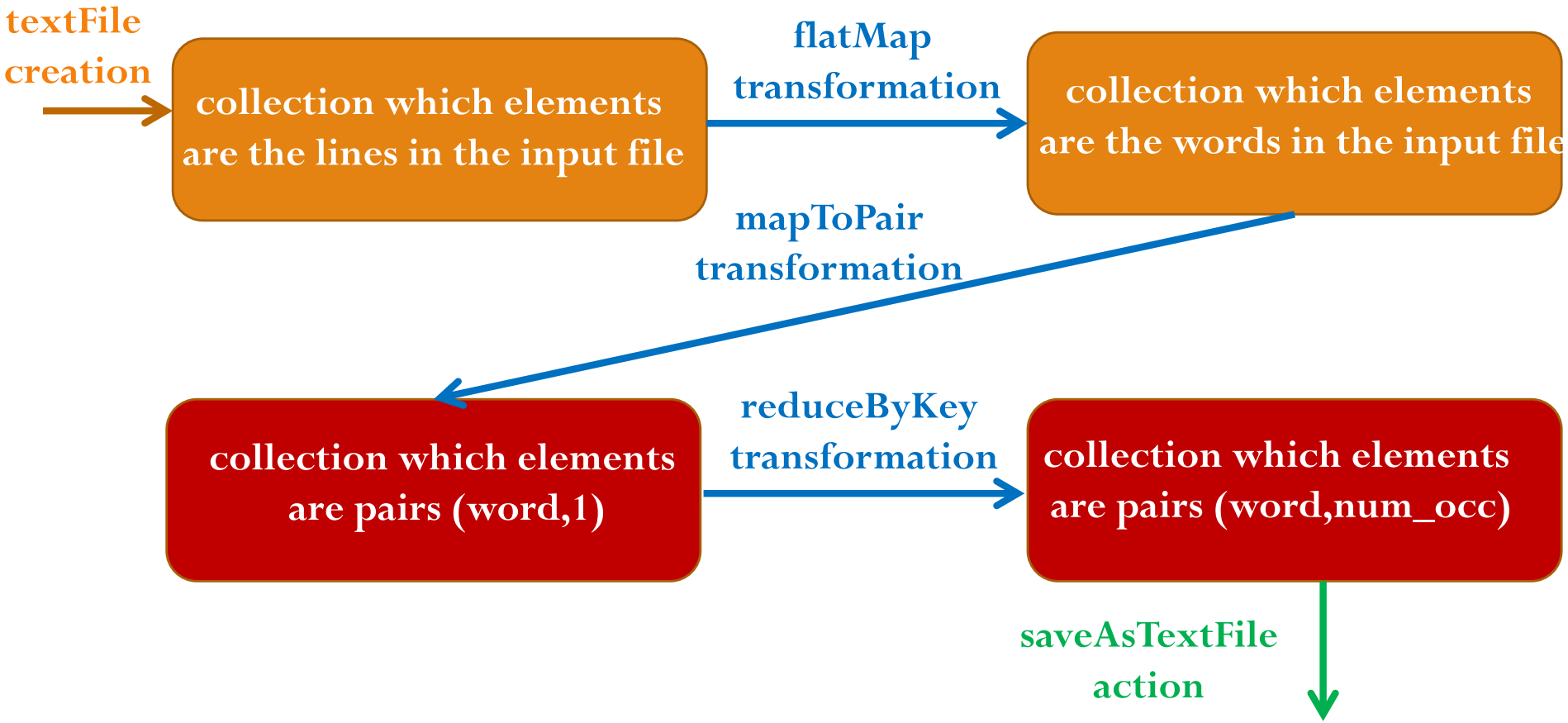
```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

- Actions trigger the actual computation of a dataflow

Persistency and Caching of RDDs

- The previous dataflow program access the **`badlinesRDD`** twice to perform two actions: a **`count()`** and a **`take()`**
- To avoid a repeated computation of the **`badlinesRDD`** from the input RDD, we can explicitly persist the former within the cluster's main memory
- Just call **`persist()`** on the RDD before the actions are invoked
- if main memory is full, Spark uses an LRU policy to free cached RDDs

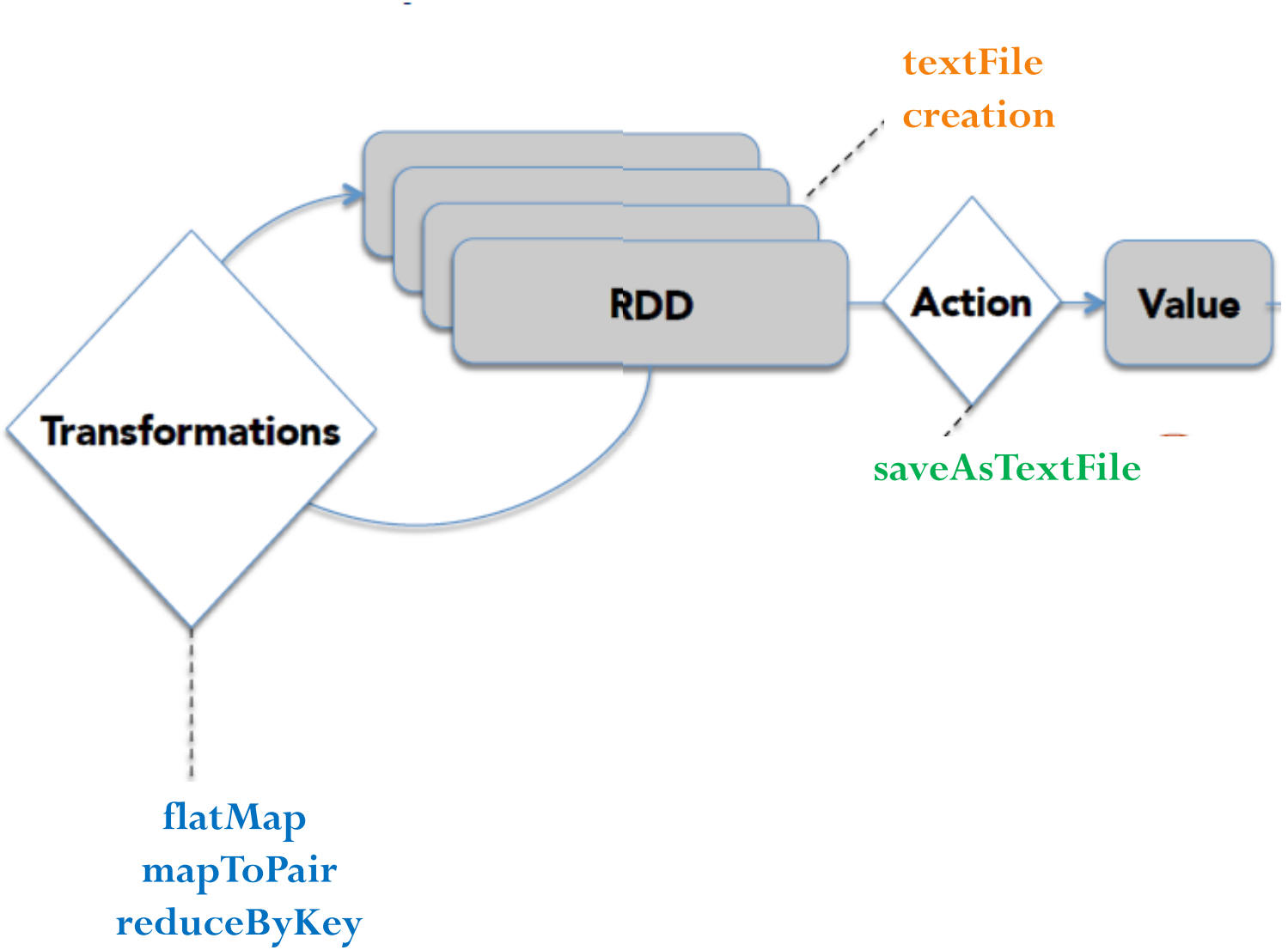
Spark program: Word Count



Spark program: Word Count

```
import sys from pyspark
import SparkContext, SparkConf

# create Spark context with necessary configuration
sc = SparkContext("local","PySpark Word Count Exmaple")

# read data from text file and split each line into words
words = sc.textFile("input.txt").flatMap(lambda line: line.split(" "))

# count the occurrence of each word
wordCounts = words.map(lambda word: (word, 1)).reduceByKey(lambda
a,b:a +b)

# save the counts to output wordCounts.saveAsTextFile("output/")
```

Spark program: Word Count Conceptual representation



Spark Transformations & Actions (Low-Level API)

Trasformations (1)

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

Transformations (2)

Transformation	Meaning
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Actions

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.