DADS 6002 / CI 7301
Big Data Analytics

Spark

- The batch processing model of MapReduce was not well suited to some common workflows such as iterative, interactive on-demand computations upon a single dataset.
- In order to achieve coordination and fault tolerance, the MapReduce model uses a pull execution model that requires intermediate results to be written back to HDFS and then read for next step processing.
- The reads/writes to HDFS take a large amount of time, hence MapReduce can be too slow for interactive on-demand computations.

Spark

- Spark is the first fast, general purpose distributed computing paradigm which does not require to decompose a data flow work into a series of MapReduce jobs.
- Spark achieves high speed via a new data model called resilient distributed datasets (RDDs).
- The RDDs are stored in memory while being computed upon thus eliminating expensive intermediate disk writes. Data sharing in memory is 10 to 100 times faster than network and disk.

Spark

- Spark provides an API for distributed programming similar to the MapReduce model, but is designed to be fast for interactive queries and iterative algorithms which is essential for optimization and machine learning.
- It primarily achieves this by caching data required for computation in the memory of nodes in the cluster.

Spark

- Spark not only supports map and reduce operations, it provides many APIs for distributed data processing including sample, filter, join and collect etc.
- Spark is implemented in Scala, but also provides programming APIs in Scala, Java, R and Python.
- Spark keeps the dataset in memory as much as possible throughout the course of application, preventing the reloading of data between iterations.

Spark

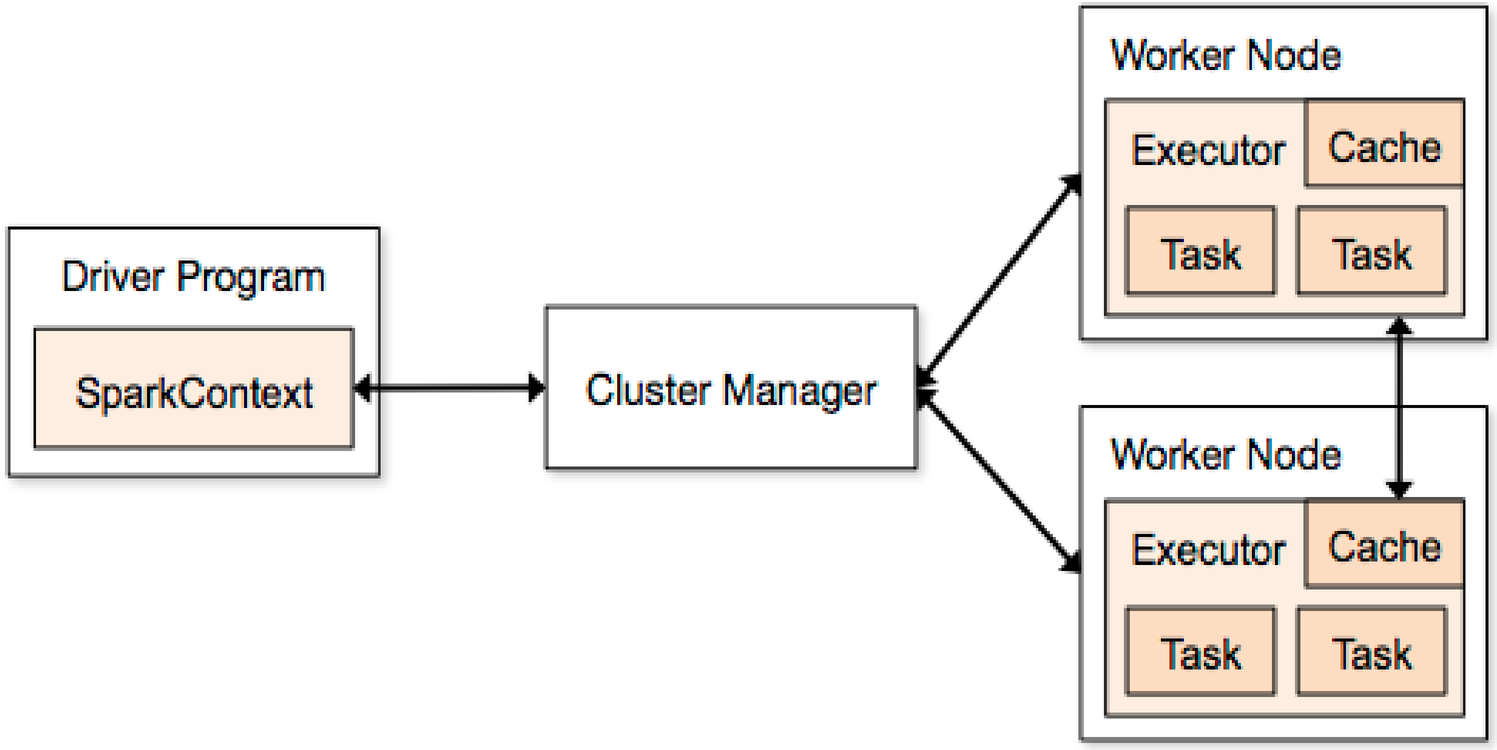
- A Spark program can be described as a data flow graph using a Directed Acyclic Graph (DAG).
- Spark’s execution engine knows ahead of time from the DAG and so distributes the computation across the cluster and manage the details of the computation.
- By increasing the number of nodes in the cluster and therefore the amount of available memory to hold an entire, very large dataset, Spark can process data very fast and so it can be used to run a program interactively.

Spark

- Spark focuses purely on computation rather than data storage.
- It can be run in a cluster that implements data warehousing and cluster management tools, e.g.

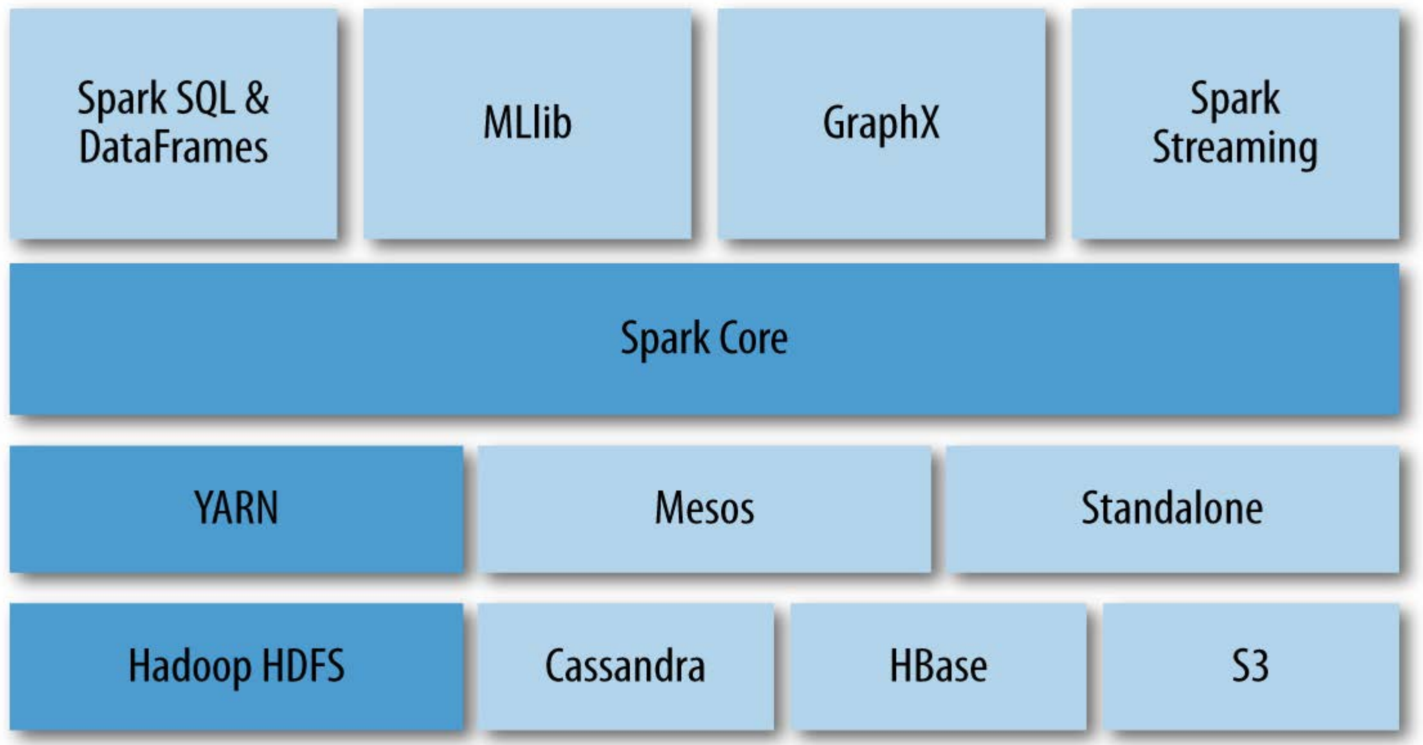
1. Hadoop with HDFS and Yarn,
2. Mesos (Apache Cluster Management Software, Mesos kernel runs on every nodes, up to 10,000 nodes or more, and provides applications, e.g., Hadoop, Spark, Kafka, Elasticsearch, with API's for resource management and scheduling across entire datacenter and cloud environments),
3. Standalone (Simple cluster with one master node distributing works to all one or more worker nodes)

Spark



Source : https://www.safaribooksonline.com/library/view/mastering-apache-spark/9781783987146/ch01s03.html

Spark



Spark Stack

Source https://www.safaribooksonline.com/library/view/data-analytics-with/9781491913734/ch04.html

Spark

- When Spark is built with Hadoop, it utilizes YARN to allocate and manage cluster resources like processors and memory via Resource Manager.
- Spark can then access any Hadoop data source e.g. HDFS, HBASE, Hive or Cassandra (NoSQL database)
- Spark contains a core module called Spark Core that provides basic and general functionality via API.

Spark

- Spark then builds upon this core, implementing special purpose libraries for a variety of data science tasks that interact with Hadoop.
- The component libraries are not integrated into the Spark Core, allowing flexibility to develop new component libraries on top of Spark Core.
- The primary component libraries include
- Spark SQL provides APIs for interacting with Spark via Spark SQL and results can be returned as Dataset (a strongly typed collection of domain-specific objects that can be transformed in parallel) or Data Frames (distributed collections of data organized into columns)

Spark

- Spark Streaming enables the processing and manipulation of unbounded streams of data in real time.
- MLlib is a library of common machine learning algorithms implemented as Spark operations on RDDs e.g. classifications, regressions, etc.
- GraphX is a collection of algorithms an tools for manipulating graphs and performing parallel graph operations and computation.

Spark

- These components combined with Spark programming model provide a rich methodology of interacting with cluster resources.

Resilient Distributed Datasets (RDDs)

- RDDs are programming abstractions that represent read-only (immutable) collection of objects that are partitioned across a set of machines.

Spark

- RDDs can be built via parallel operations which loads a dataset from data storage e.g. HDFS or S3 then partitions it into collections of data that can be cached in memory of worker nodes for immediate reuse (using cache command). By default, RDDs are stored in memory, but can be swapped out to disk to make room in the memory for new RDDs.
- Spark APIs is a collection of operations that create, transform and export RDDs in the form of functional programming constructs (always return new data objects).

Spark

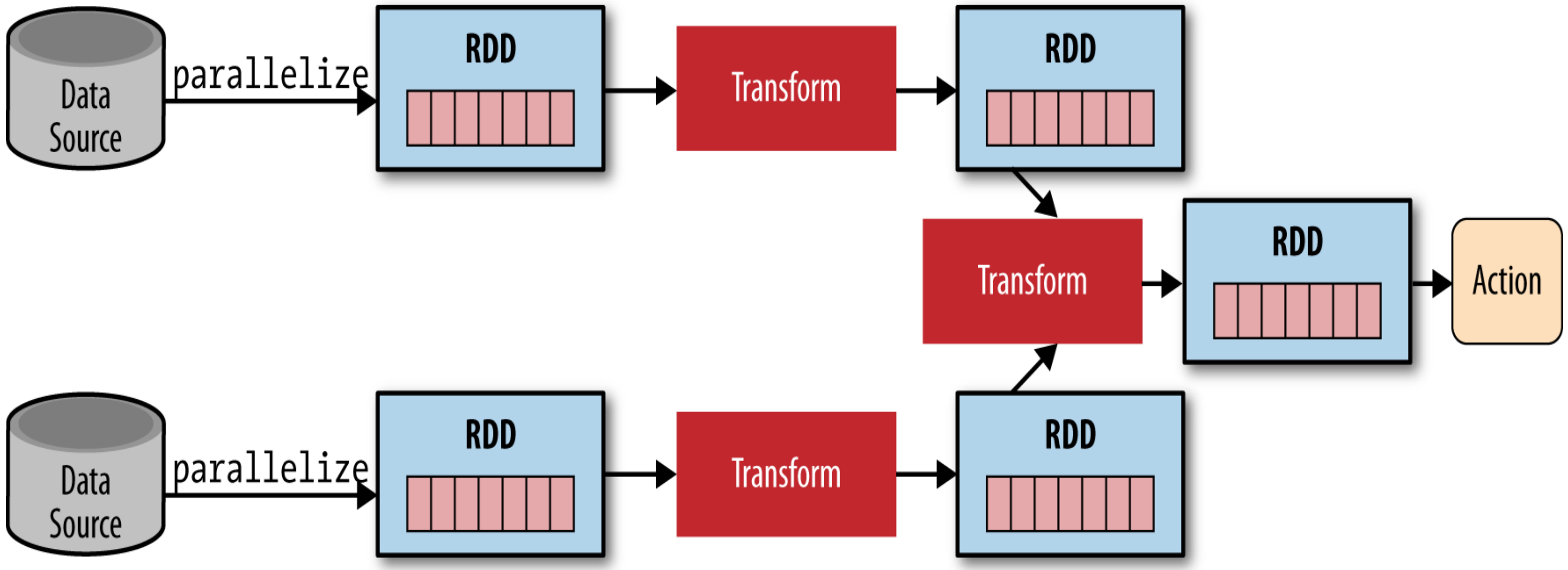
- Two types of operations can be applied to RDDs :

1. Transformations are the operations that are applied to an existing RDD to create a new RDD.
2. Actions are the operations that returns a result back to the Spark driver program – resulting in a coordination or aggregation of all partitions in an RDD.

Spark

- For example, map is a transformation, because a map function is passed and applied to every object stored in the RDD and the output of the function maps to a new RDD.
- Aggregation like reduce is an action, because reduce requires the RDD to be repartitioned (according to a key) and some aggregate value like sum or mean computed and returned.
- Most actions in Spark are designed solely for the purpose of output – return a single value or a small list of values or to write data back to distributed storage.

Spark



Source: https://www.safaribooksonline.com/library/view/data-analytics-with/9781491913734/ch04.html

Spark

- Spark applies transformations lazily – inspecting a complete sequence of transformations and an action before executing them by submitting a job to the cluster.
- This lazy-execution provides significant storage and computation optimizations by examining the complete transformation chain in order to compute upon only the data needed for a result e.g. if the action is first() on an RDD, Spark will avoid reading the entire dataset and return just the first item of the RDD.

Spark

- Code in a driver program is evaluated lazily on the driver-local machine when submitted, and upon an action, the driver code is distributed across the cluster to be executed by worker nodes on their partitions of the RDD.
- Results are then sent back to the driver for aggregation or compilation.

Spark

- A Typical data flow sequence for programming Spark is as follows:

1. Define one or more RDDs, either through accessing data stored on disk (e.g. HDFS, Cassandra, HBase or S3), parallelizing some collection (the act of partitioning a dataset and sending each part of the data to the worker nodes that will perform computation upon it), transforming an existing RDD or by caching in the memory.

Spark

2. Invoke operations on the RDD by applying the closure (a function that does not rely on external variables or data) to each element of the RDD. Spark offers many high-level operators beyond map and reduce.
3. Execute aggregating actions (e.g. count, collect, save etc.) on the result RDD. Action kick off the computation of transformations (step 2) and actions (step 3) on the cluster (lazy-execution)

Spark

- Closures can access their local variables. If global variables are needed to share data among closures running in different worker nodes, two types of shared variables are supported in Spark. The shared variables can be accessed by all worker nodes in a restricted fashion. The two types of variables are :

1. Broadcast variables are distributed to all workers but are read-only and often used as lookup tables or shared lists.

Spark

2. Accumulators are variables that workers can "add" using associative operations e.g. x += 1. They are typically used as counters.

Spark

- The basic fault-tolerant support of Spark are:

1. Since Apache Spark RDD is an immutable dataset, each Spark RDD remembers the lineage of the deterministic operation that was used on fault-tolerant input dataset to create it. The flow of lineages of the deterministic operations forms a logical execution plan called a lineage graph.
2. If a worker node fails and any partition of an RDD is lost, then that partition can be re-computed (replayed) from the original fault-tolerant dataset using the lineage graph of operations.

Spark

3. To achieve fault tolerance for all the generated RDDs, they can be made persistent in disk using persistent() command and so they can be replicated via HDFS among multiple Spark executors in worker nodes in the cluster.

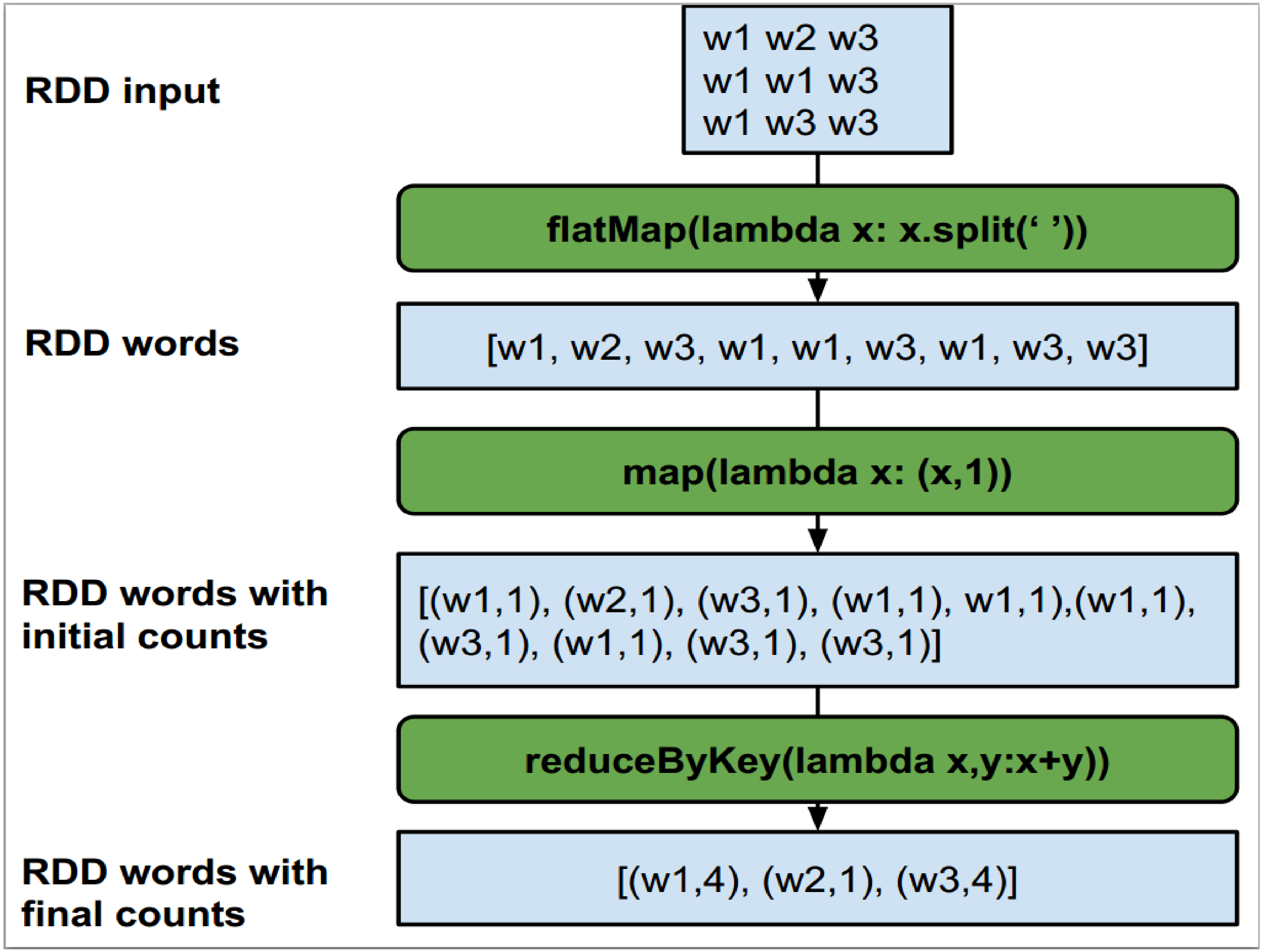
- For datasets that fit into the memory of a cluster, Spark is fast enough for users to interact and explore big data from interactive shell that implements a Python REPL (Read-Evaluate-Print Loop) called PySpark.

Spark

- Once PySpark is executed, it automatically creates a SparkContext to work with. Users can refer to the SparkContext via sc variable.

```
% pyspark
>>> text = sc.textFile(“shakespeare.txt”)
>>> text.collect()
>>> exit()        ( or ctrl-d )
```

- The textFile method of sc can be used to create an RDD from a text file.


RDD input
w1 w2 w3
w1 w1 w3
w1 w3 w3
flatMap(lambda x: x.split(‘ ’))
RDD words
[w1, w2, w3, w1, w1, w3, w1, w3, w3]
map(lambda x: (x,1))
RDD words with initial counts
[(w1,1), (w2,1), (w3,1), (w1,1), w1,1),(w1,1), (w3,1), (w1,1), (w3,1), (w3,1)]
reduceByKey(lambda x,y:x+y))
RDD words with final counts
[(w1,4), (w2,1), (w3,4)]

https://datamize.wordpress.com/2015/02/08/visualizing-basic-rdd-operations-through-wordcount-in-pyspark/

Spark

- Use PySpark to solve Word Count problem

```
% pyspark
    >>> text = sc.textFile("shakespeare.txt")
    >>> from operator import add
    >>> def tokenize(text):
    …           return text.split()
    …
    >>> words = text.flatMap(tokenize)
```

Spark

- A new RDD called words is created by transforming text RDD through application of flatMap operator with the defined closure tokenize. The flatMap is similar to map but can return zero or more values for each RDD element while map returns only one value for each RDD element.

  ```
  >>> wc = words.map(lambda x : (x,1) )
  ```

- Transform words RDD to a new wc RDD using map operator with anonymous function (with lambda keyword in Python)

Spark

```
>>> counts = wc.reduceByKey(add)

>>> counts.saveAsTextFile(“wc”)
```

- The reduceByKey action is performed to get word counts via add function into counts RDD.
- Counts RDD is then saved as a text file under wc directory in the current working directory.
- Check the text file containing word counts.

```
% ls wc/

% head wc/part-00000
```

Spark

- Writing a Spark application program in Python is similar to working with Spark in the interactive mode.
- A driver program in Python may contain some data declaration (for shared variables), define closures for transforming RDDs and step-by-step execution plan of RDD transformation and aggregation.

Spark

```
# Example program word_count.py

from pyspark import SparkContext
def main():
    sc = SparkContext(appName='SparkWordCount')
    input_file = sc.textFile('/user/cloudera/wc/input.txt')
    counts = input_file.flatMap(lambda line: line.split()) \
                 .map(lambda word: (word, 1)) \
                 .reduceByKey(lambda a, b: a + b )
```

Spark

```
    counts.saveAsTextFile(‘/user/cloudera/wc/output’)
    sc.stop()
if __name__ == ‘__main__’:
    main()
```

Spark

- To execute the spark program

```
% spark-submit --master local[*]  word_count.py
```

master options

local[*] - run the Spark program on local host with many processes as needed.

yarn - run the Spark program with YARN as the resource manager

spark://host:port – run the Spark program on stand alone mode with host as the master node (default port = 7077)

Mesos://host:port - run the Spark program with Mesos cluster (default port = 5050)

Spark

- Transformation Operations
- map(func) – returns a new RDD formed by passing each element of the source RDD through a function func.

```
>>> rdd = sc.parallelize( [1, 2, 3, 4 ] )
>>> out = rdd.map(lambda x: x * 2 )
>>> out.collect()
    [2, 4, 6, 8]
```

Spark

- filter(func) – returns a new dataset of an RDD formed by selecting those elements of the source on which func returns true.

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> out = rdd.filter( lambda x: x % 2 == 0 )
>>> out.collect()
    [2, 4]
```

Spark

- distinct() – returns a new dataset of an RDD that contains the distinct elements of the source dataset

```
>>> rdd1 = sc.parallelize([1,2,3,2,4,3])
>>> out = rdd1.distinct()
>>> out.collect()
    [1,2,3,4]
```

Spark

- flatMap(func) – similar to map, but each input item can be mapped to 0 or more output items (so func should return a sequence of items)

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> out = rdd.map(lambda x : [x, x+5] )
>>> out.collect()
     [[1,6],[2,7],[3,8],[4,9]]
>>> rdd = sc.parallelize([1,2,3,4])
>>> out = rdd.flatMap(lambda x : [x, x+5] )
>>> out.collect()
     [1,6,2,7,3,8,4,9]
```

Spark

- Action Operations
- reduce(func) – aggregate dataset's elements in an RDD using function func. Func takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel (in a cluster).

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> rdd.reduce(lambda a,b : a * b )
    24
```

Spark

- take(n) – returns an array with the first n elements of the RDD.

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> rdd.take(2)
    [1,2]
>>> rdd.take(5)
    [1,2,3,4]
```

Spark

- collect() – returns all of the elements of the RDD as an array.

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> rdd.collect()
     [1,2,3,4]
```

Spark

- takeOrdered(n, key=func) – returns an array containing first n elements of the RDD in their natural order of key specified by the func.

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> rdd.takeOrdered(3, lambda x : -x )
    [4, 3, 2]
```

Spark

- reduceByKey(func) – returns a new distributed dataset of (K,V) pairs in an RDD, where the values for each key are aggregated using the given reduce function func ((V,V) -> V)

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3, 6), (1,3),
(3,8)])
>>> out = rdd.reduceByKey(lambda a,b : a + b)
>>> out.collect()
    [(1,5), (3,18)]
```

Spark

- sortByKey() – returns a new RDD of (K,V) pairs sorted by key K in ascending order.

```
>>> rdd = sc.parallelize([(1,’c’), (3,’d’), (3, ‘a’),
(1,’b’), (3,’e’)])
>>> out = rdd.sortByKey()
>>> out.collect()
    [(1,’c’), (1,’b’), (3,’d’), (3,’a’), (3, ‘e’)]
```

Spark

- groupByKey() – returns a new RDD of (K, iterable <V>) pairs.

```
>>> rdd = sc.parallelize([(1,’c’), (3,’d’), (3, ‘a’), (1,’b’), (3,’e’)])
>>> out = rdd.groupByKey()
>>> out.map(lambda x : (x[0], list(x[1]))).collect()
      [(1,[’c’,’b’]), (3,[’d’,’a’,’e’])]
```

Spark

- count() – returns the number of data items in the RDD.

```
>>> rdd = sc.parallelize([(1,’c’), (3,’d’), (3, ‘a’),
(1,’b’), (3,’e’)])
>>> rdd.count()
     5
```

Spark

- foreach(func) – apply func to each item of the given RDD.

```
>>> def f(x):
...         print x
...
>>> rdd = sc.parallelize([1,2,3,4])
>>> rdd.foreach(f)
    1
    2
    3
    4
```

Spark

- Broadcast variables
- In driver program

```
>>> b = sc.broadcast([1,2,3,4])
```

- In closure

```
>>> def f(x):
…              return(b.value[x])
…
>>> rdd = sc.parallelize([0,3])
>>> out = rdd.map(f)
>>> out.collect()
     [1, 4]
```

Spark

- Accumulator variables

In a driver program

```
>>> rdd = sc.parallelize([1,2,3,4])
>>> c = sc.accumulator(0)
```

Create a accumulator variable c with initial value of zero.

In a closure

```
>>> def f(x):
…       global c
…       c += x
…
>>> rdd.foreach(f)
>>> c.value
        10
```

Spark

- Set Log Level Information Display

  sc.setLogLevel(level)

  level can be “ALL”, “TRACE”, “DEBUG”, “INFO”, “WARN”, “ERROR”, ”FATAL”, and ”OFF”

```
>>> sc.setLogLevel(“ALL”)
>>> sc.setLogLevel(“ERROR”)
```