CARLSON SCHOOL OF MANAGEMENT
UNIVERSITY OF MINNESOTA

# Working With RDDs in Spark

MSBA 6330 Prof Liu

*Mostly based on Dr. Anthony Joseph (UC Berkeley)'s course on Apache Spark





## Outline

- How do spark programs work?
- How RDDs are created from files or data in memory
- Commonly used RDD transformations
- Commonly used RDD actions
- Understand a typical lifecycle of a Spark program.





Working with RDDs in Spark

## SPARK PROGRAMMING FRAMEWORK

Carlson School of Management

## Spark Deployment Modes

- Spark has several deployment modes
  - **Local mode**: run everything in a single JVM (java virtual machine), useful for testing or demonstration.
  - **Cluster mode:**
    - Standalone:
      - Spark manages its own cluster; simple and easy to setup
    - YARN
      - Using YARN as cluster manager
    - Mesos
      - Using Apache Mesos as cluster manager

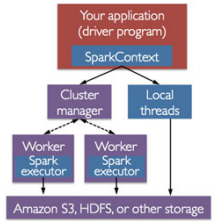
https://techvidvan.com/tutorials/spark-modes-of-deployment/
(for additional reading)
http://stackoverflow.com/questions/28664834/which-cluster-type-should-i-choose-for-spark (choosing between different cluster modes)

4

Carlson School of Management

## Anatomy of a Spark Cluster

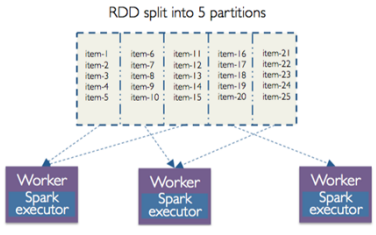
- Each Spark application has a **driver program**, which uses **SparkContext** to communicate with **cluster manager** or **local threads**
  - The Spark program we write is the driver program.
- An **executor** is created on each worker node per application. It runs Spark tasks and interact with external storage (HDFS, S3, etc)
- RDDs are distributed among worker nodes.




5

Carlson School of Management

## Partition RDDs




More partitions = More parallelism

Each worker can work on one partition at a time
Typically you want 2-4 partitions for each CPU in your cluster (this can to be automatically set by Spark)

6

Carlson School of Management

## So, how many executors and partitions?*

- 4-6 cores per executor to maximize HDFS I/O throughput
  - Number of executors (--num-executors)
  - Cores for each executor ( --executor-cores):
  - Memory for each executor ( --executor-memory):
- How many partitions to have
  - Rule of thumb: ~128MB per partition
  - Don't have too big partitions
    - your job may fail due to 2GB shuffle block limit
  - Don't have too few partitions
    - Your job will be slow, to making use of parallelism.

*Mark Grover, Ted Malaska (2016) Top 5 Mistakes when writing spark applications, Spark SUMIT EAST.

7

Carlson School of Management

## Spark Context

- A Spark program first creates a SparkContext object that
  - Establishes a connection to the Spark's execution environment
  - tells Spark how and where to access a cluster
  - is required for every Spark application
  - is automatically created in a Spark Shell as variable `sc`.
  - is manually created in a standalone program
- Use SparkContext to create RDDs, access Spark services and run jobs.

https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-SparkContext.html

8

Carlson School of Management

## Spark Context

**Every Spark application requires a Spark Context**
- The main entry point to the Spark API

**Spark Shell provides a preconfigured Spark Context called `sc`**

Python

```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

Scala

```
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

Before Spark 2, separate contexts need to be created to use Spark SQL & Streaming. After Spark 2.X, a **spark session** ("spark") is introduced to provide a single point of entry that include all of the functionality of SparkContext, plus APIs for SQL, Hive, and Streaming.

Carlson School of Management

Working with RDDs in Spark

# RESILIENT DISTRIBUTED DATASETS (RDDS)



Carlson School of Management

## Resilient Distributed Datasets

- RDDs (Resilient Distributed Dataset) are part of core Spark
  - **Resilient:** If data in memory is lost, it can be recreated
  - **Distributed**: Processed across the cluster
- Characteristics of RDDs
  - **Immutable** once constructed
  - Track lineage information to efficiently recomputed lost data
  - Unstructured
    - No schema defining columns and rows
    - Not table-like; cannot be queried using SQL-like transformations such as where and select
    - Often used to convert unstructured or semi-structured data into structured form.



Carlson School of Management

## Content of RDDs

- RDDs can hold any serializable type of element
  - Primitive types: integers, characters, booleans etc.
  - Collections such as: strings, lists, arrays, tuples, dicts, nested collection types
  - Scala/Java objects (if serializable)
  - Mixed types
- Some types of RDDs have additional functionality
  - Pair RDDs
    - RDDs consisting of Key-Value pairs
  - Double RDDs
    - RDDs consisting of numeric data

Working with RDDs in Spark

# CREATING AN RDD





## RDD Data Sources

- RDDs can be constructed from
  - files in HDFS or any other storage system
  - transforming an existing RDDs
  - parallelizing existing Python collections (lists)





## Creating RDDs From Collections

- You can create RDDs from collections instead of files
  `sc.parallelize(collection)`
- Useful when
  - Testing
  - Generating data programmatically
  - Integrating

myData is a python collection on your local host

```
> myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2)
['Alice', 'Carlos']
```

myRdd is a distributed dataset in the Spark execution environment configured by sc.

Carlson School of Management

## Create RDDs from Python collections

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]
>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

This argument specifies the number of partitions.

16

Carlson School of Management

## Creating RDDs from Files (1)

- For file-based RDDs, use `sc.textFile`
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files, e.g.:
    - `sc.textFile("myfile.txt")`
    - `sc.textFile("mydata/*.log")`
    - `sc.textFile("myfile1.txt,myfile2.txt")`
  - Each line in the file(s) is a separate record in the RDD
- Use `sc.hadoopFile` or `sc.newAPIHadoopFile` to read other formats

17

Carlson School of Management

## Creating RDDs from Files (cont.)

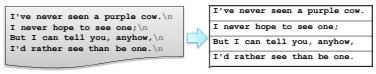
- Files are referenced by absolute or relative URI
  - Absolute URI:
    - `file:/home/training/myfile.txt` -- a file on a local host under /home/training
    - `file:///c:/Users/John/documents/myfile.txt` -- a file on C:/Users… (in windows systems)
    - `hdfs://localhost:8020/loudacre/myfile.txt` – a file on the HDFS cluster at host "localhost" at port 8020 under directory /loudacre
    - `s3n://bucket/directory/filename.ext` – reading a file on S3
  - Relative URI (assume the cluster mode):
    - `myfile.txt` – files on user's home directory on HDFS cluster (i.e. under `/user/training/`)
    - `/loudacre/weblogs/*.log` – files on spark cluster under absolute path "/loudacre/weblogs/"

18

Carlson School of Management

## Creating RDDs from Files (2)

- `textFile` maps each line in a file to a separate RDD element




- textFile only works with line-delimited text files



Carlson School of Management

## Create RDDs from Files

- From HDFS, text files, Hyper table, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop InputFormat, and directory or glob wildcard: /data/201404*

```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
MappedRDD[2] at textFile at
   NativeMethodAccessorImpl.java:-2
```

  - RDD distributed in 4 partitions
  - Elements are lines of input
  - **Lazy evaluations** – no execution happens now

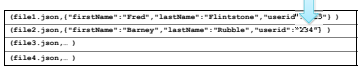


Carlson School of Management

## Whole File--Based RDDs (1)

- `sc.textFile` maps each line in a file to a separate RDD element
  - What about files with a multi-line input format, e.g. XML or JSON?
- sc.`wholeTextFiles`(directory)
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)





Carlson School of Management

## Whole File-Based RDDs (2)

```
> import json
> myrdd1 = sc.wholeTextFiles(mydir)
> myrdd2 = myrdd1
  .map(lambda (fname,s): json.loads(s))
> for record in myrdd2.take(2):
>     print record["firstName"]
```

Output:

```
Fred
Barney
```

```
> import scala.util.parsing.json.JSON
> val myrdd1 = sc.wholeTextFiles(mydir)
> val myrdd2 = myrdd1
  .map(pair => JSON.parseFull(pair._2).get.
            asInstanceOf[Map[String,String]])
> for (record <- myrdd2.take(2))
    println(record.getOrElse("firstName",null))
```

22

Carlson School of Management

## Anonymous functions

- RDD operations often involves anonymous functions
- Python: `lambda` functions
  - Restricted to a single expression, and is not re-used
  - E.g. `lambda a, b: a + b`
    `a, b` are arguments for the function
    the function returns `a + b`
- Scala: anonymous function syntax with "=>"

```
val xboxRDD = auctionRDD.filter(line=>line.contains("xbox"))
```

  - The function accepts one argument: `line`
  - It returns a boolean value, `line.contains("xbox")`

23

Carlson School of Management

Working with RDDs in Spark

# SPARK TRANSFORMATIONS AND ACTIONS

24

Carlson School of Management

## Operations on RDDs

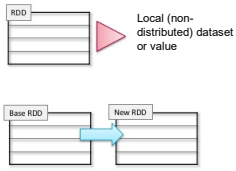
- Two types of operations: **transformations** and **actions**
- **Transformations** are lazy (not computed immediately)
- Transformed RDD is executed when action runs on it

25

Carlson School of Management

## RDD actions and transformations

- Two types of operations on RDD
  - **Actions** – returns local dataset or value (non-RDD), e.g.,
    - count
    - take(n)
    - collect()
  - **Transformations** – generate new RDDs based on the current one, e.g.
    - filter
    - map
    - reduceByKey




**Transformations** are lazy (not computed immediately) -- just like Pig is lazy

26

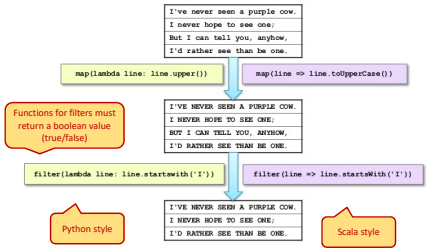
Carlson School of Management

## Commonly Used Transformations

| | |
|---|---|
| **map** | Returns new RDD by applying func to each element of source |
| **filter** | Returns new RDD consisting of elements from source on which function is true |
| **groupByKey** | Returns dataset (K, Iterable<V>) pairs on dataset of (K,V) |
| **reduceByKey** | Returns dataset (K, V) pairs where value for each key aggregated using the given reduce function |
| **flatMap** | Similar to map, but function should return a sequence rather than a single item |
| **distinct** | Returns new dataset containing distinct elements of source |

27

## Example: map and filter transformations




28

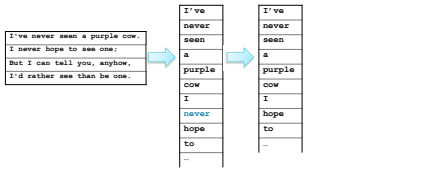


## Example: `flatMap` and `distinct`

Python
```
> sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .distinct()
```

Scala
```
> sc.textFile(file).
  flatMap(line => line.split(' ')).
  distinct()
```

Each function returns a list; the collection of lists are then "flattened"






## RDD Transformations

- `map`, `filter`, and `distinct` transformations

```
>>>  rdd  =  sc.parallelize([1,2, 3, 4])
>>>  rdd.map(lambda x: x  *  2)
RDD:  [1, 2, 3, 4]   -> [2, 4, 6, 8]
>>>  rdd.filter(lambda x: x  %  2  == 0)
RDD:  [1,2, 3, 4] -> [2, 4]
>>>  rdd2  =  sc.parallelize([1, 4, 2, 2, 3])
>>>  rdd2.distinct()
RDD:  [1, 4, 2, 2, 3]   -> [1, 4, 2, 3]
```

Function literals (green) are *closures* automatically passed to Spark workers

30

## RDD Transformations

- `map` and `flatMap`

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.map(lambda x: [x, x+5])
RDD: [1, 2, 3] -> [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] -> [1, 6, 2, 7, 3, 8]
```

`flatMap` merges the results into a single sequence





## Commonly Used Actions

| | |
|---|---|
| **count()** | Returns the number of elements in the dataset |
| **reduce(func)** | Aggregate elements of dataset using function func |
| **collect()** | Returns all elements of dataset as an array to driver program |
| **take(n)** | Returns an array with first n elements |
| **first()** | Returns the first element of the dataset |
| **takeOrdered(n, [ordering])** | Return first n elements of RDD using natural order or custom operator |

Note reduceByKey is a transformation but reduce is an action

Same as `take(1)`

- An action on an RDD returns values to the driver program, after running the computation on the dataset.
- As mentioned earlier, transformations are lazy. They are only computed when an action requires a result to be returned to the Driver program.





## reduce, take, and collect

- Getting data out of RDDs

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # the first two as list

>>> rdd.collect()
Value: [1,2,3] # the entire collection as list
```

The reducer is applied recursively to produce a single result. The function must be commutative and associative.

Carlson School of Management

## takeOrdered and count:

Indicating a descending order of values

```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list

lines = sc.textFile("...", 4)
print lines.count()
```

lines.count() causes Spark to:
- read data
- sum within partitions
- combine sums in driver

34

Carlson School of Management

## Other General RDD Operations

- **Other RDD actions**
  - **`first`** – return the first element of the RDD
  - **`foreach`** – apply a function to each element in an RDD
  - **`top(n)`** – return the largest *n* elements using natural ordering
- **Sampling**
  - **`sample`** – create a new RDD with a sampling of elements [transformation]
  - **`takeSample`** – return an array of sampled elements [action]
- **Double RDD operations – operate on double RDDs only**
  - Statistical functions, e.g., **`mean`**, **`sum`**, **`variance`**, **`stdev`**

Documentation and more examples: https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Carlson School of Management

Working with RDDs in Spark

# LAZY EXECUTION & CACHING

36

Carlson School of Management

## Lazy Execution

- Consider a the following

```
RDD = sc.textFile(…)
newRDD = RDD.filter(….)
newRDD.count()
```

- What will happen after each line?

37

Carlson School of Management

## After this line:

```
RDD = sc.textFile(…)
newRDD = RDD.filter(….)
newRDD.count()
```

No computation has happened yet

38

Carlson School of Management

## After this line:

```
RDD = sc.textFile(…)
newRDD = RDD.filter(….)
newRDD.count()
```

No computation has happened yet

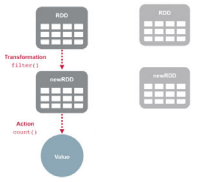
39

Carlson School of Management

## After this line:

```
RDD = sc.textFile(…)
newRDD = RDD.filter(….)
newRDD.count()
```

Data will be read into memory. RDD and newRDD will be created. After getting the result - count, the RDDs are no longer in memory.




40

Carlson School of Management

## Caching

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

- Spark will read the source data twice
  - 1st time – `lines.count()`
    - read data, sum within partitions, combine sums in driver
  - 2nd time – `comments.count()`
    - Read data (again), filter & sum within partition, combine sums in driver

41

Carlson School of Management

## Caching

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(),comments.count()
```

- Reading is a common step in two processes. Use `cache()` to avoid re-computing.

42