"The goal is to turn data into information, and information into insight"

Carly Fiorina, HP CEO, 2000s

Hands-on H5
Dataflow Programming

CS205: Computing Foundations for Computational Science

Dr. David Sondak

Spring Term 2021

INSTITUTE FOR APPLIED COMPUTATIONAL SCIENCE AT HARVARD UNIVERSITY

HARVARD School of Engineering and Applied Sciences

Lectures adapted from Ignacio M. Llorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

B. Parallel Computing

C. Parallel Data Processing

    C1. Batch Data Processing

    C2. Dataflow Processing

    C3. Stream Data Processing

Wrap-Up: Advanced Topics

CS205: Contents

APPLICATION SOFTWARE

Application Parallelism | Program Design

Application Software

BIG COMPUTE

OpenACC | Optimization

OpenMP | MPI

BIG DATA

Spark

Map-Reduce

Programming Model

Slurm | Platform | Yarn

Architecture

Cloud Computing | Computing Cluster

Before We Start

Where We Are

Concepts → Platform → Programming

Week 9: Batch Data Processing => MapReduce

3/22	3/23		3/25	3/26
	Hands-on H4 MapReduce Programming	Lab I8 Hadoop	Lecture C2 Dataflow Processing (Quiz & Reading)	

Week 10: Dataflow Processing => Spark

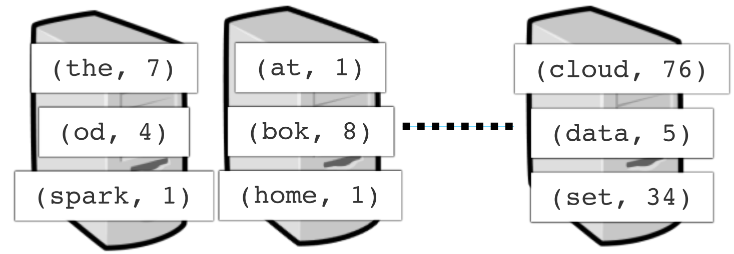
3/29	3/30		4/1	4/3
	Hands-on H5 Spark Programming	Lab I9 Spark Single Node	Lecture C3 Stream Data Processing (Quiz)	

Context

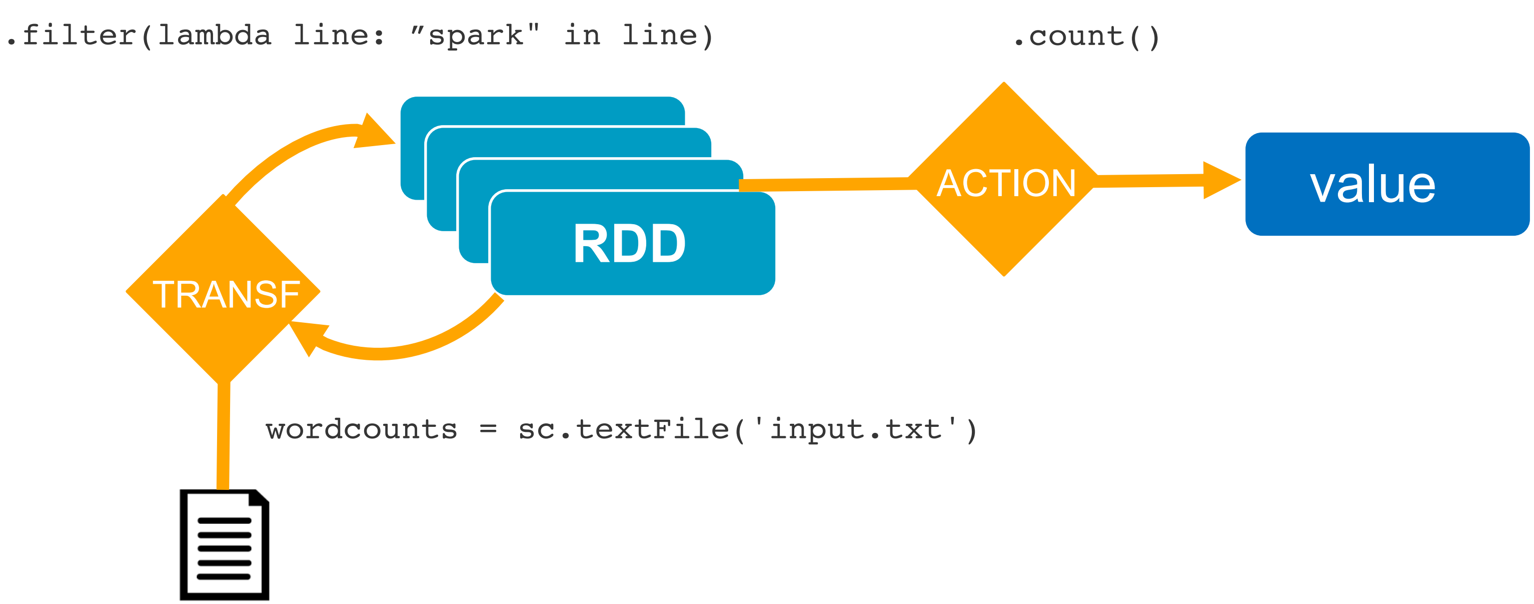
The Spark Programming Model

The Fundamental Data Structure - Resilient Distributed Dataset

- Resilient: Fault-tolerant
- Distributed: Multiple-node
- Dataset: Collection of partitioned data organized in records



Operations: Transformations and Actions



Hands-on Examples

Requirements

1. Unix-like shell (Linux, Mac OS or Windows/Cygwin)
2. Python installed
3. Installation of Spark (see guide “Install Spark in Local Mode”)

Roadmap

Dataflow Programming

PySpark

Resilient Distributed Datasets

Distributed Collections

Transformations

Actions

Caching and Persistence

Pipelining

PySpark

Interactive Shell for Python

Spark Interactive Shell for Python

- Easiest way to try Spark.
- Responsible for linking the python API to the spark core and initializing the Spark context
- Runs in local mode on 1 thread by default, but can control through `MASTER` environment variable

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.5
      /_/

Using Python version 3.7.4 (default, Aug 13 2019 15:17:50)
SparkSession available as 'spark'.
```

Resilient Distributed Datasets

RDD Creation

Different Ways to Create RDDs

- Text File

```
>>> logFile = '/var/log/syslog'
>>> textRDD = sc.textFile(logFile) // create RDD
>>> textRDD.count() // RDD => result
>>> linesWithRoot = textRDD.filter(lambda line: 'root' in line) // RDD => RDD
>>> linesWithRoot.take(9) // RDD => result
```

- HDFS : Data residing on a distributed file system

```
>>> sc.textFile("hdfs://namenode:9000/path/file")
```

- New Defined RDD

```
>>> data = [1, 2, 3, 4, 5]
>>> distData = sc.parallelize(data) // create distributed collection
```

- From Other RDD

```
>>> distDataS = distData.map(lambda x: x * x) // RDD => RDD
```

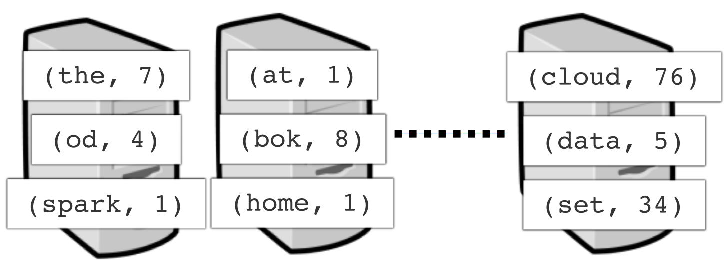
Distributed Collections

Parallel Processing

RDD Partitions

- A “parallelized” data set where the elements are copied across the nodes of a distributed system to form a distributed collection that can be computed in parallel

```
>>> data = [1, 2, 3, 4, 5]
>>> sc.defaultParallelism // default number of partitions
>>> distData = sc.parallelize(data) // create a distributed collection
>>> distDataP = sc.parallelize(data, 3) // slice the data set into 3 partitions, 3 way parallelism
>>> distDataP.count() // do some ’statistics’
>>> distDataP.getNumPartitions() // give number of partitions
>>> distDataP.reduce(lambda x, y : x + y) // more ’statistics’
```



Transformations

Create a New RDD from an Existing One

map()

- Reads one element at a time
- Takes one value, creates a new value

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

flatMap()

- Produce multiple elements for each input element

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

Transformations

Create a New RDD from an Existing One

filter()

- Reads one element at a time
- Evaluates each element
- Returns the elements that pass the filter()

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

Key-value Operations

```
>>> pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])
>>> pets.reduceByKey(lambda x, y: x + y).collect() // => [('cat', 3), ('dog', 1)]
>>> pets.groupByKey().mapValues(list).collect()// => [('cat’, [1, 2]), ('dog’, [1])]
>>> pets.sortByKey().collect() // => [('cat', 1), ('cat', 2), ('dog', 1)]
```

Transformations

Create a New RDD from an Existing One

union()

- Merges two RDDs together

```
>>> erin_brown = [('physics',85),('math',75),('chemistry',95)]
>>> paul_adams = [('physics',65),('math',45),('chemistry',85)]
>>> erin = sc.parallelize(erin_brown)
>>> paul = sc.parallelize(paul_adams)
>>> erin.union(paul).collect()
```

join()

- Joins two RDDs based on a common key

```
>>> Subject_wise_erin = erin.join(paul)
>>> Subject_wise_erin.collect()
```

Transformations

Create a New RDD from an Existing One

intersection()

- Gives you the common terms or objects from the two RDDS

```
>>> techs = ['sachin', 'abhay', 'michael', 'rahane', 'david', 'ross',
'raj', 'rahul', 'hussy', 'steven', 'sourav']
>>> managers = [‘alice', 'abhay', ‘erin', ‘sasha', 'steve ']
>>> techRDD = sc.parallelize(techs)
>>> managersRDD = sc.parallelize(managers)
>>> managertechs = techRDD.intersection(managersRDD)
>>> managertechs.collect()
```

Transformations

Create a New RDD from an Existing One

distinct()

- Gets rid of any ambiguities

```
>>> best_screenplay = ["movie10","movie4","movie6","movie7","movie3"]
>>> best_story = ["movie9","movie4","movie6","movie5","movie1"]
>>> best_direction = ["movie10","movie4","movie7","movie12","movie8"]
>>> story_rdd = sc.parallelize(best_story)
>>> direction_rdd = sc.parallelize(best_direction)
>>> screen_rdd = sc.parallelize(best_screenplay)
>>> total_nomination_rdd = story_rdd.union(direction_rdd).union(screen_rdd)
>>> total_nomination_rdd.distinct().collect()
```

Actions

Compute a Result Based on an Existing RDD

count()

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

collect()

- `collect()` retrieves the entire RDD
- Useful for inspecting small datasets locally and for unit tests

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

Actions

Compute a Result Based on an Existing RDD

take(), first(), top(), takeSample()

- `take(n)` returns n elements from an RDD
- `takeSample()` – Take a random sample of the dataset (should specify a random seed)
- Use `takeOrdered()`, `top(n)` for ordered return

takeOrdered()

```
>>> rdd = sc.parallelize([5, 1, 3, 2])
>>> rdd.takeOrdered(4)
[1, 2, 3, 5]
>>> rdd.takeOrdered(4, lambda n: -n)
[5, 3, 2, 1]
```

reduce()

- Takes two elements of the same type and returns one new element

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

Caching and Persistence

Efficiency

Transformations are lazy!

- A transformed RDD is only executed when actions run on it

```
>>> pyLines = lines.filter(lambda line: 'Python' in line)
>>> pyLines.first()
```

- No need for Spark to load all the lines containing "Python" into memory!

An Example

```
>>> textFile = sc.textFile("/user/emp.txt")
```

- It does nothing. It creates an RDD that says "we will need to load this file". The file is not loaded at this point.
- RDD operations that require observing the contents of the data cannot be lazy (these are called actions). An example is `RDD.count`
- So if you write `textFile.count`, at this point the file will be read, the lines will be counted, and the count will be returned.
- What if you call `textFile.count` again? The same thing: the file will be read and counted again. Nothing is stored. An RDD is not data.

Caching and Persistence

Efficiency

Caching

- Decreases the computation time **by almost 100X** when compared to other distributed computation frameworks like hadoop mapreduce

```
>>> textFile = sc.textFile("/user/emp.txt")
```

```
>>> textFile.cache
```

- It does nothing. `RDD.cache` is also a lazy operation. The file is still not read. But now the RDD says *"read this file and then cache the contents"*. If you then run `textFile.count` the first time, the file will be loaded, cached, and counted. If you call `textFile.count` a second time, the operation will use the cache. It will just take the data from the cache and count the lines.
- `cache` is like `persist(MEMORY_ONLY)`

Caching and Persistence

Efficiency

Persistence

- RDDs are recomputed for every action, can be expensive and can also cause data to be read from disk again!
- RDDs can be cached for reuse, `rdd.persist()`

```
>>> lines = sc.textFile("README.md")
>>> lines.count()
>>> pythonLines = lines.filter(lambda line : "Python" in line)
>>> pythonLines.count()
```

Causes Spark to reload lines from disk!!!

```
>>> lines = sc.textFile("README.md")
>>> lines.persist() # Spark keeps lines in RAM
>>> lines.count()
>>> pythonLines = lines.filter(lambda line : "Python" in line)
>>> pythonLines.count()
```

Spark will avoid reloading lines every time it is used

Pipelining

Defining a Workflow

Building a Pipeline of Operations

```
>>> lines = sc.textFile("README.md")
>>> lines.map(…).filter(…).count(…)
>>> (lines
     .map(…)
     .filter(…)
     .count(…))
```

Pipelining

The WordCount Example with Spark

A Pipeline of Transformations

```
wordcounts = sc.textFile('input.txt')
```

```
'The Project Gutenberg EBook of Moby Dick; or The Whale, by Herman'
'Melville. This eBook is for the use of anyone anywhere at no cost and'
```

```
.map(lambda x: x.replace(',',' ').replace('.',' '). lower())
```

```
'the project gutenberg ebook of moby dick or the whale by herman'
'melville this ebook is for the use of anyone anywhere at no cost and'
```

```
.flatMap(lambda x: x.split())
```

```
'the' 'project' 'gutenberg' 'ebook' 'of' 'moby' 'dick' 'or' 'the 'whale'
'by' 'herman' 'melville' 'this' 'ebook' 'is' 'for' 'the' 'use' 'of'
```

```
.map(lambda x: (x, 1))
```

```
'(the, 1)' '(project ,1)' '(gutenberg, 1)' '(ebook, 1)' '(of, 1)' '(moby
, 1)' '(dick, 1)' '(or, 1)' '(the, 1)' '(whale, 1)' '(by, 1)'
```

```
.reduceByKey(lambda x,y:x+y)
```

```
'(the, 11)' '(project ,10)' '(gutenberg, 9)' '(ebook, 37)' '(of, 15)'
'(moby , 5)' '(dick, 7)' '(or, 9)' '(the, 9)' '(whale, 123)' '(by, 98)'
```

DataFrames

Processing of Tabular Data

```
Year  Type Size
2018  A    120
2018  A    200
2019  B    300
2020  C    150
2021  D    400
2021  D    450
```

- Create RDD

```
>>> rdd = sc.parallelize([(2018,"A", 120),(2018,"A",
200),(2019,"B", 300),(2020,"C",150), (2021, “D”, 400), (2021,
“D”, 450)])
>>> rdd.collect()
```

How to extract some of the columns, for example Year and Type?

How to select a group of rows, for example those from Year 2019?

How to aggregate rows, for example count by Type?

DataFrames

Processing of Tabular Data

```
>>> rdd = sc.parallelize([(2018,"A", 120),(2018,"A", 200),(2019,"B", 300),(2020,"C",150), (2021, “D”, 400), (2021, “D”, 450)])

# Convert RDD into DataFrame (you can read directly from JSON, CSV and XML)
>>> df = rdd.toDF(["year","type","size"])

# Displays the content of the DataFrame to stdout
>>> df.show()

>>> df.printSchema()

>>> df.select("type").show()

>>> df.select(df['year'], df['size'] + 1).show()

>>> df.filter(df['year'] > 2019).show()

>>> df.groupBy("type").count().show()
```

Parallel Execution

Single Node

Calculate pi with Spark

```
from pyspark import SparkConf, SparkContext
import string

conf = SparkConf().setMaster('local[2]').setAppName('Pi')
sc = SparkContext(conf = conf)

N = 10000000
delta_x = 1.0 / N
print sc.parallelize( xrange (N),4 ).map( lambda i: (i +0.5) * delta_x ).map( lambda x: 4 / (1 + x **2) ).reduce ( lambda a, b: a+b) * delta_x
```

Execute with different number of partitions and threads, and compare number of tasks and execution time

Next Steps

- Get ready for next lecture**:**
  C3. Stream Data Processing

- Get ready for next lab session
  I9. Spark in Local Mode