CS 696 Intro to Big Data: Tools and Methods
Spring Semester, 2020
Doc 14 Spark Intro
Feb 27, 2020

Spark

Created at UC Berkeley's AMPLab

- 2009 Project started
- 2014 May Version 1.0
- 2016 July Version 2.0.2
- 2017 July Version 2.2.0

Programming interface for
- Java, Python, Scala, R

Interactive shell for
- Python, Scala, R (experimental)

Runs on
- Linux, Mac, Windows

Cluster manager
- Native Spark cluster
- Hadoop YARN
- Apache Mesos

File System
- HDFS
- MapR File System
- Cassandra
- OpenStack Swift
- S3

Pseudo-Distributed Mode
- Single machine
- Uses local file system

Time Line

1991 - Java project started

1995 - Java 1.0 released, Design Patterns book published

2000 - Java 3

2001 - Scala project started

2002 - Nutch started

2004 - Google MapReduce paper

Scala version 1 released

2005 - F# released

2006 - Hadoop split from Nutch

Scala version 2 released

2007 - Clojure released

2009 - Spark project started

2012 - Hadoop 1.0

2014 - Spark 1.0

Hadoop Word Count - Map

```
public class WordCount {

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, Context context) throws IOException,
                                                                    InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }
}
```

Hadoop Word Count - Reduce

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
                                                    throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

Hadoop Word Count - Main

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
}
```

Spark Word Count - Python

```
from __future__ import print_function
import sys
from pyspark.sql import SparkSession

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        sys.exit(-1)

spark = SparkSession.builder.appName("PythonWordCount").getOrCreate()

lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")

spark.stop()
```

Spark Word Count - Java

```
public final class JavaWordCount {
    private static final Pattern SPACE = Pattern.compile(" ");
    public static void main(String[] args) throws Exception {

    if (args.length < 1) {
      System.err.println("Usage: JavaWordCount <file>");
      System.exit(1);
    }

    SparkSession spark = SparkSession.builder().appName("JavaWordCount").getOrCreate();

    JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
    JavaRDD<String> words = lines.flatMap(s -> Arrays.asList(SPACE.split(s)).iterator());
    JavaPairRDD<String, Integer> ones = words.mapToPair(s -> new Tuple2<>(s, 1));
    JavaPairRDD<String, Integer> counts = ones.reduceByKey((i1, i2) -> i1 + i2);

    counts.saveAsTextFile("hdfs://...");
    spark.stop();
  }
}
```

Scala

```
object SparkWordCount {
  def main(args: Array[String]) {
    val spark = SparkSession.builder.appName("Spark Pi").getOrCreate()
    val textFile = sc.textFile("hdfs://...")
    val counts = textFile.flatMap(line => line.split(" "))
                         .map(word => (word, 1))
                        .reduceByKey(_ + _)
    counts.saveAsTextFile("hdfs://...")
    spark.stop()
  }
}
```

Python vs Scala on Spark

Scala is faster that Python

- But that is not so important here
- Most of the computation on Spark is done in Spark

Using Python with Spark

- Python data has to be
  - Converted between Python format and Scala/Java format
  - Sent between Python process and JVM

Installing PySpark using Anaconda

pip install pyspark

make sure using Anaconda pip

Sample Program

```
import pyspark
import random
sc = pyspark.SparkContext(appName="Pi")
num_samples = 100000
def inside(p):
  x, y = random.random(), random.random()
  return x*x + y*y < 1
count = sc.parallelize(range(0, num_samples)).filter(inside).count()
pi = 4 * count / num_samples
print(pi)
sc.stop()
```

Installing Spark - Java/Scala/Python

http://spark.apache.org/downloads.html

Download Apache Spark™

1. Choose a Spark release: 2.4.0 (Nov 02 2018)
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later
3. Download Spark: spark-2.4.0-bin-hadoop2.7.tgz
4. Verify this release using the 2.4.0 signatures, checksums and project release KEYS.

Read the Readme.md file

Helps to set your path

https://spark.apache.org/docs/latest/

Running PySpark from Command Line

```
 3->pyspark
Python 2.7.15 (default, Nov 27 2018, 21:40:55)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
2019-03-07 12:49:20 WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.0
      /_/

Using Python version 2.7.15 (default, Nov 27 2018 21:40:55)
SparkSession available as 'spark'.
>>>
```

Having PySpark Run in Jupyter Notebook

Don't need PySpark installed via pip, just regular install

In your shell

```
export  SPARK_HOME /Java/spark-2.4.0-bin-hadoop2.7
export PYSPARK_DRIVER_PYTHON jupyter
export PYSPARK_DRIVER_PYTHON_OPTS 'notebook'
```

In path

```
$SPARK_HOME/bin $SPARK_HOME/sbin
```

Having PySpark Run in Jupyter Notebook

pyspark

Then will start pyspark in jupyter notebook

In jupypter notebook "spark" will be a SparkSession

Standard Warning

Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

Major Parts of Spark

Spark Core
  Resilient Distributed Dataset (RDD)

Original (Old) Spark

Spark SQL
  SQL, csv, json
  Dataframe

Newer Version Spark

Spark Streaming
  Near real-time response

MLib Machine Learning Library
  Statistics, regression, clustering, dimension reduction, feature extraction
  Optimization

GraphX

Spark

Ecosystem of packages, libraries and systems on top of Spark Core

Unstructured API

Resilient Distributed Datasets (RDD)
Accumulators
Broadcast variables

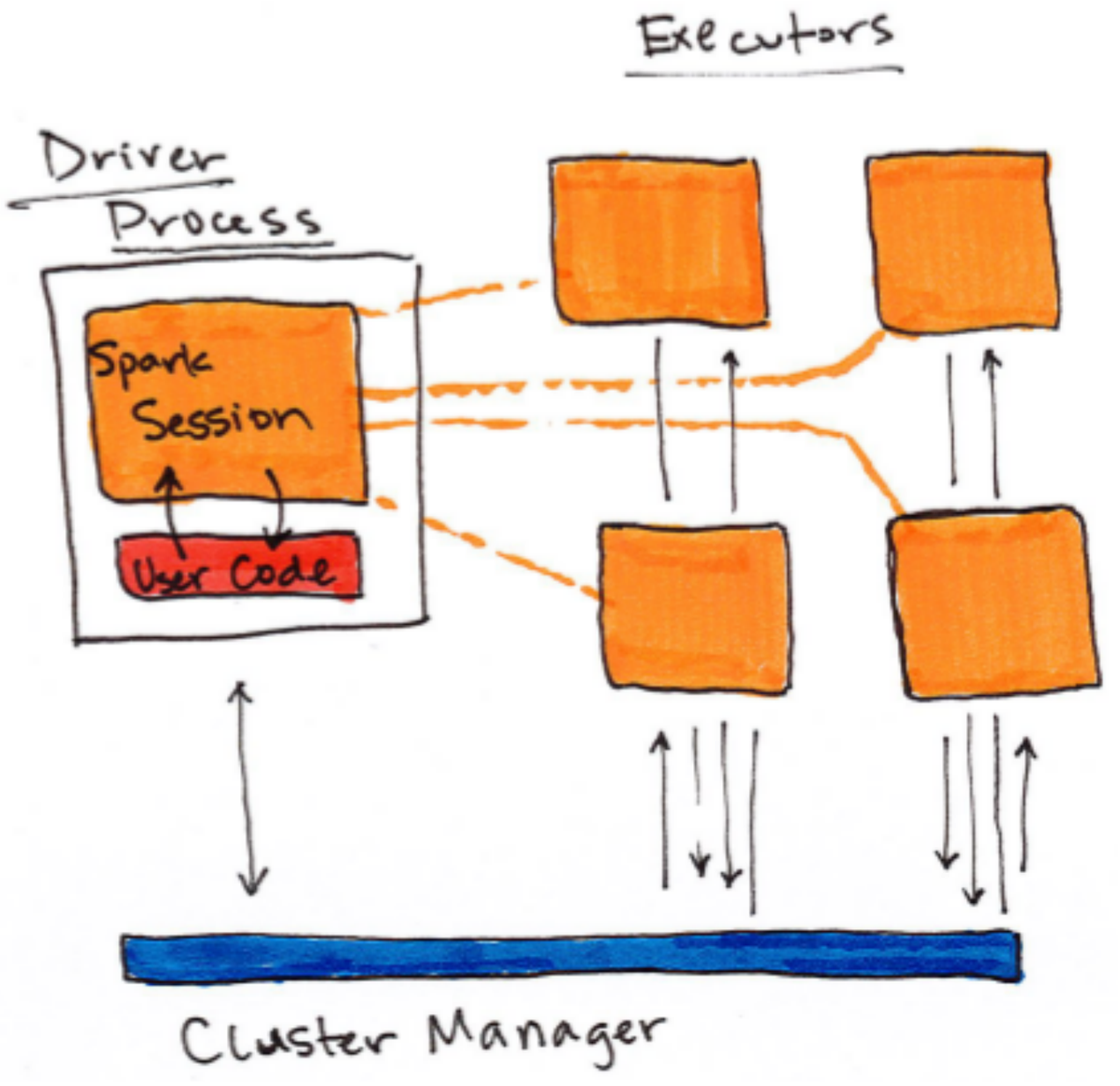
Old Spark

Structured API

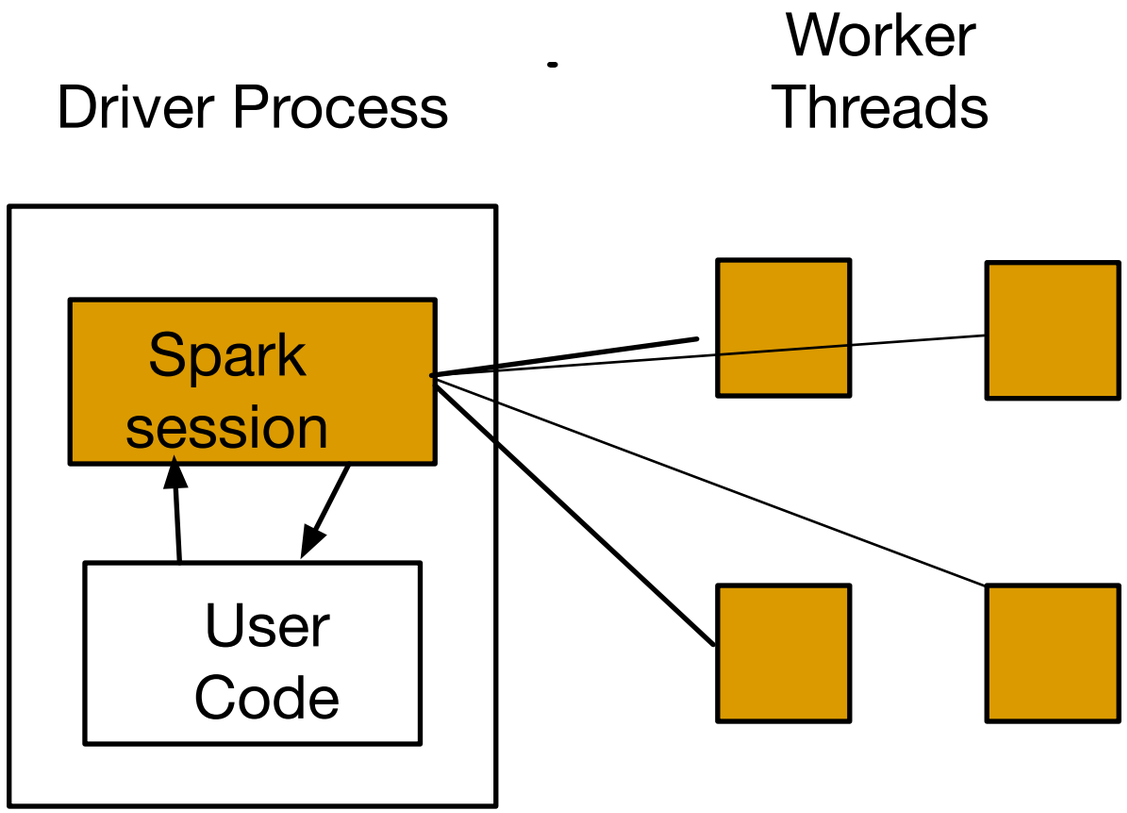
DataFrames
Datasets
Spark SQL

Newer, faster, higher level
Preferred over Unstructured

Basic Architecture



Local Mode



We will start using local mode

Use local mode to

Develop Spark code

SparkContext

Entry point for Unstructured API (Old Spark)

Connection to Spark cluster
Runs on master node
Used to create RDDs, accumulators, broadcast variables
Only one SparkContext per JVM
stop() the current SparkContext before starting another

SparkContext org.apache.spark.SparkContext
  Scala version

JavaSparkContext org.apache.spark.api.java.JavaSparkContext
  Java version

pyspark.SparkContext
  Python version

SparkSession

org.apache.spark.sql.SparkSession
pyspark.sql.SparkSession

Contains a SparkContext

Entry point to use Dataset & DataFrame

Connection to Spark cluster
Runs on master node

Major Data Structures

Resilient Distributed Datasets (RDDs)

Fault-tolerant collection of elements that can be operated on in parallel

Dataset & Dataframes

Fault-tolerant collection of elements that can be operated on in parallel

Rows & Columns

JSON, csv, SQL tables

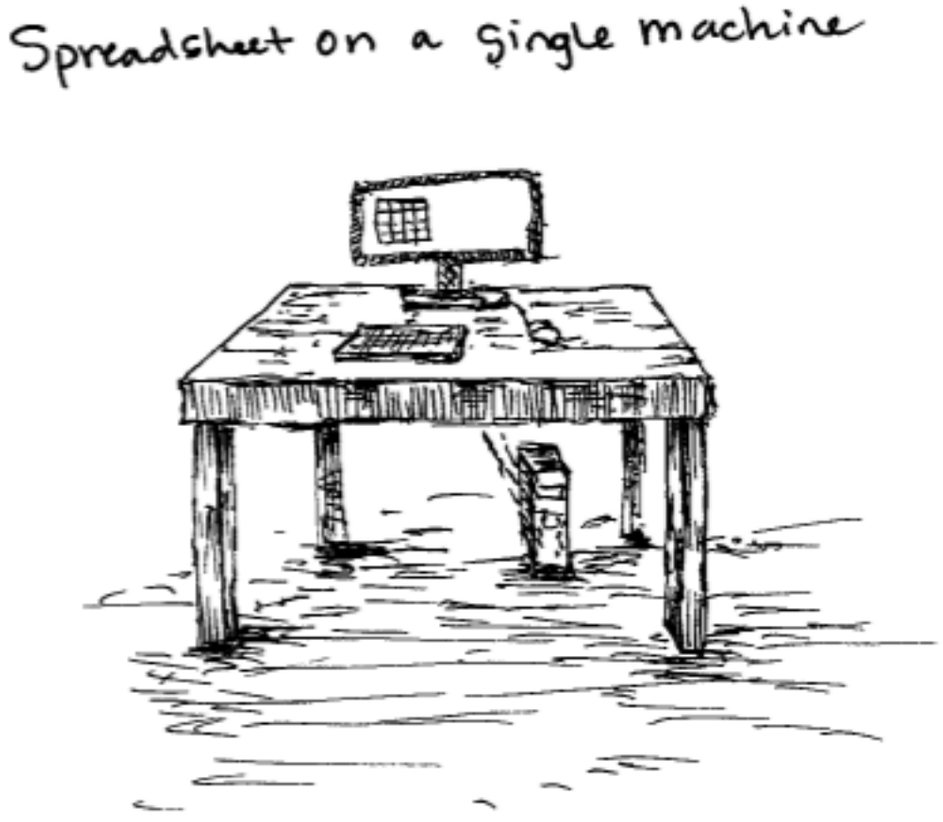
Part of SparkSQL

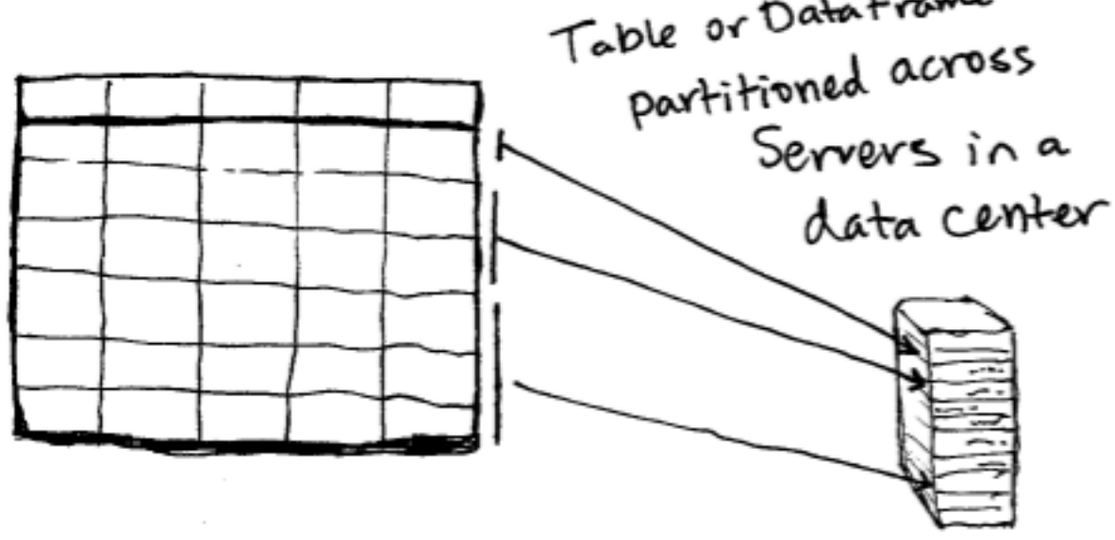
Use RDDs as underlying data structure

Partitions

RDD & Dataset

- Divided into partitions
- Each partition is on different machine





Resilient & Distributed

Distributed

- Partitions on different machines

Resilient

- Each partition can be replicated on multiple machines
- Data structure knows how to reproduce operations

Basic Operations

RDDs, Dataframes, Datasets

- Immutable

Transformations

- Create new dataset (RDD) from existing one
- Lazy
- Only done when needed by an action
- Examples
  - map, filter, sample, union, distinct, groupByKey, repartition

Actions

- Return results to driver program
- Examples
  - reduce, collect, count, first, take

Actions & Transformations on DataSet

View the Spark Scala API

org.apache.spark.sql.Dataset

Actions

▶ `def collect(): Array[T]`
Returns an array that contains all rows in this Dataset.

▶ `def collectAsList(): List[T]`
Returns a Java list that contains all rows in this Dataset.

▶ `def count(): Long`
Returns the number of rows in the Dataset.

▶ `def describe(cols: String*): DataFrame`
Computes statistics for numeric and string columns, including count, mean, stddev, min, and max.

▶ `def first(): T`
Returns the first row.

Typed transformations

▶ `def alias(alias: Symbol): Dataset[T]`
(Scala-specific) Returns a new Dataset with an alias set.

▶ `def alias(alias: String): Dataset[T]`
Returns a new Dataset with an alias set.

▶ `def as(alias: Symbol): Dataset[T]`
(Scala-specific) Returns a new Dataset with an alias set.

DataFrame, DataSet & RDD

What are they

What is the difference

When do use which one

Which languages can use them

DataFrame

Table with rows and Columns

Row

org.apache.spark.sql.Row

Schema

Column labels

Column types

```
+-------+----+
|   name| age|
+-------+----+
|   Andy|  30|
| Justin|  19|
|Michael|null|
+-------+----+
```

Partitioner

Distributes DataFrame among cluster

Plan

Series of transformations to perform on DataFrame

Langauges

Scala, Java, JVM languages, Python, R

Optimized

Spark Catalyst Optimizer

Python DataFrame & Spark DataFrame

They are different

Need Apache Arrow to convert between them

DataSet

Same as DataFrame except for Rows

Programmer defines Row class

- Scala Cas Class
- Java Bean

Difference from DataFrame

- Compiler knows column names and column types in DataSet
- Compile time error checking
- Better data layout

Languages

- Scala, Java, JVM languages

RDD

Table

No information about types

No compile time or runtime type checking

```
+-------+----+
|   Andy|  30|
| Justin|  19|
|Michael|null|
+-------+----+
```

Shares same basic operations as DataFrames & DataSets

Far fewer optimizations

No Catalyst Optimizer

No space optimization

Example - Same data

RDD	33.3 MB
DataFrame	7.3 MB

Languages

Java, Scala

Python, R - not recommended

Spark Types

Java Types are not space efficient

“abcd” - 48 bytes

Spark has its own types

Special memory representation of each type

Space efficient

Cache aware

Spark	Scala	Python	Python API
ByteType	Byte	int or long	ByteType()
ShortType	Short	int or long	ShortType()
IntegerType	Int	int or long	IntergerType()
LongType	Long	int or long	LongType()

Structured verses Unstructured

Structured = DataSet, DataFrame

Unstructured = RDD

Typed verses Untyped

Typed = DataSet

Untyped = DataFrame

Some Sample Data

JSON flight Data 2015

United States Bureau of Transportation statistics

The Definitive Guide, Zaharia & Chambers, O'Reilly Media, Inc, 2017-10-??

2015-summary.json

{"ORIGIN_COUNTRY_NAME":"Romania","DEST_COUNTRY_NAME":"United States","count":15}
{"ORIGIN_COUNTRY_NAME":"Croatia","DEST_COUNTRY_NAME":"United States","count":1}
{"ORIGIN_COUNTRY_NAME":"Ireland","DEST_COUNTRY_NAME":"United States","count":344}
{"ORIGIN_COUNTRY_NAME":"United States","DEST_COUNTRY_NAME":"Egypt","count":15}
{"ORIGIN_COUNTRY_NAME":"India","DEST_COUNTRY_NAME":"United States","count":62}
{"ORIGIN_COUNTRY_NAME":"Singapore","DEST_COUNTRY_NAME":"United States","count":1}
{"ORIGIN_COUNTRY_NAME":"Grenada","DEST_COUNTRY_NAME":"United States","count":62}
{"ORIGIN_COUNTRY_NAME":"United States","DEST_COUNTRY_NAME":"Costa Rica","count":588}
{"ORIGIN_COUNTRY_NAME":"United States","DEST_COUNTRY_NAME":"Senegal","count":40}
{"ORIGIN_COUNTRY_NAME":"United States","DEST_COUNTRY_NAME":"Moldova","count":1}

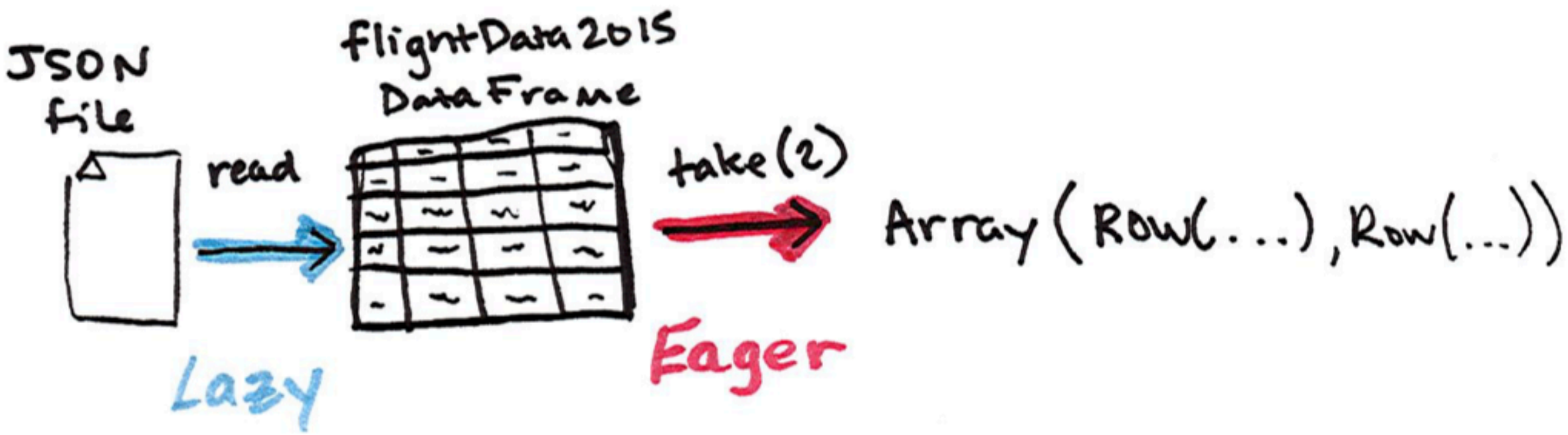
```
jsonFlightFile =
    "/Users/whitney/Courses/696/Fall17/SparkBookData/flight-data/json/2015-summary.json"
```

```
flightData2015 = spark.read.json(jsonFlightFile)
flightData2015
```

```
DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```

```
flightData2015.take(2)
```

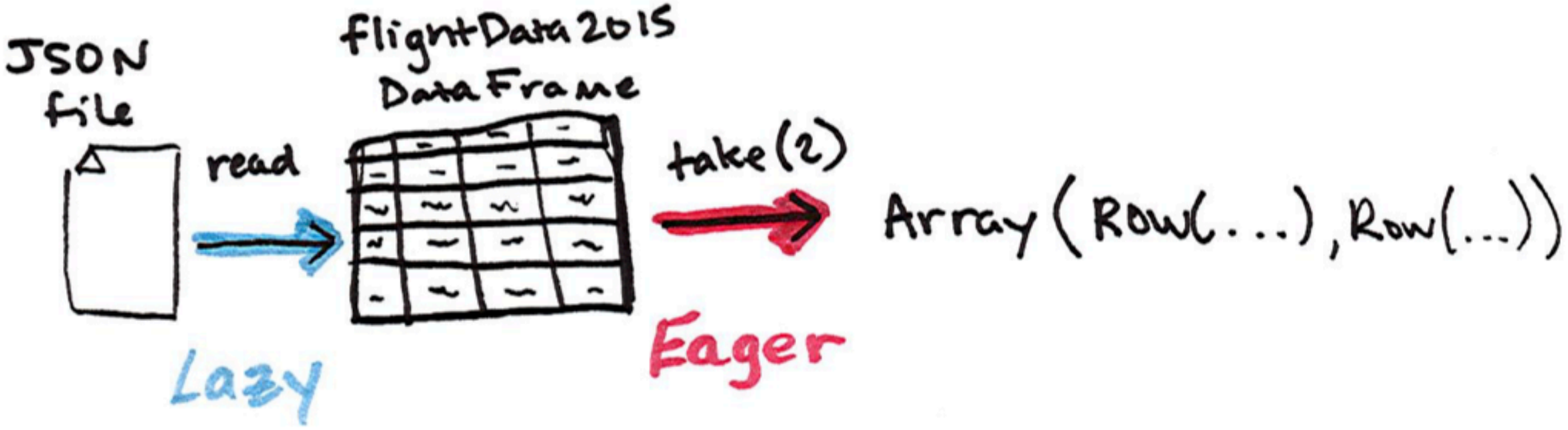
```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1)]
```



Explain - Spark Plan

flightData2015.explain()

```
== Physical Plan ==
*(1) FileScan json [DEST_COUNTRY_NAME#6,ORIGIN_COUNTRY_NAME#7,count#8L]
  Batched: false, Format: JSON,
  Location: InMemoryFileIndex[file:/Users/whitney/Courses/696/Fall17/SparkBookData/
flight-data/json/2015-summ...,
  PartitionFilters: [],
  PushedFilters: [],
  ReadSchema: struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:bigint>
```



sortedFlightData2015 = flightData2015.sort("count")

sortedFlightData2015: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string ... 1 more field]

sortedFlightData2015.take(2)

```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore', count=1),
 Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1)]
```

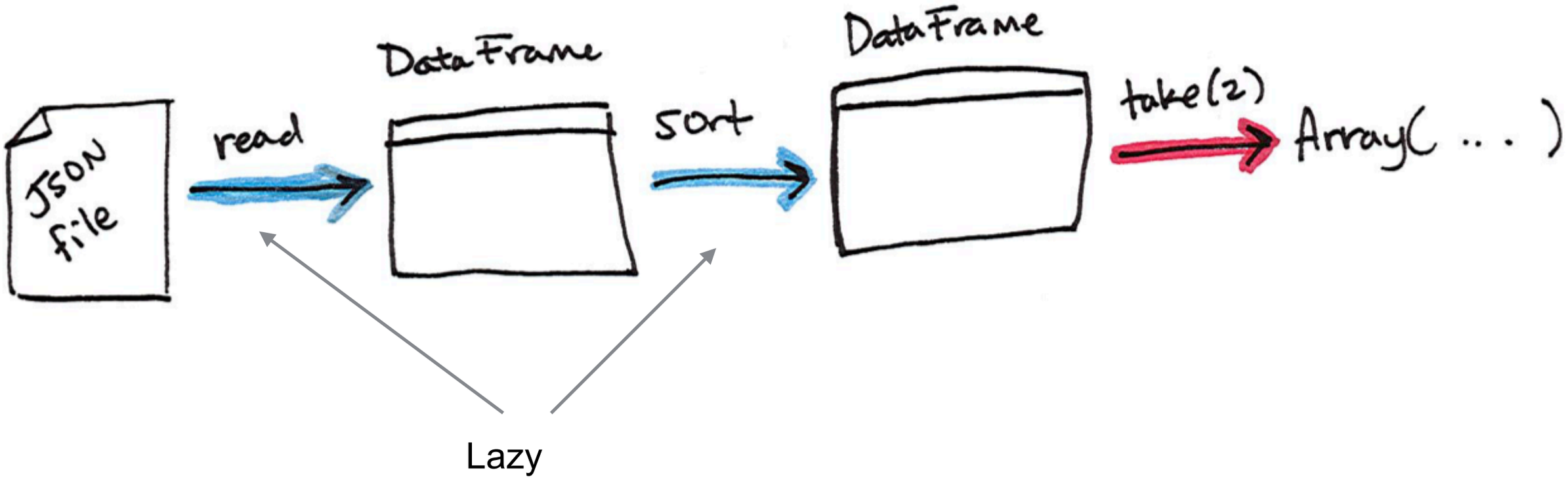
sortedFlightData2015.show(3)

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|          Moldova|      United States|    1|
|    United States|          Singapore|    1|
|    United States|            Croatia|    1|
+-----------------+-------------------+-----+
```

sortedFlightData2015.explain()

```
== Physical Plan ==
*(2) Sort [count#23L ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#23L ASC NULLS FIRST, 200)
   +- *(1) FileScan json [DEST_COUNTRY_NAME#21,ORIGIN_COUNTRY_NAME#22,count#23
            Batched: false, Format: JSON,
            Location: InMemoryFileIndex[file:/Users/whitney/Courses/696/Fall17/SparkBookData/fli
data/json/2015-summ...,
            PartitionFilters: [],
            PushedFilters: [],
            ReadSchema:
                struct<DEST_COUNTRY_NAME:string,ORIGIN_COUNTRY_NAME:string,count:b
```

Conceptual Plan



Spark stores the plan in case it needs to recompute the result

Schema

sortedFlightData2015.schema

```
StructType(
   List(
      StructField(DEST_COUNTRY_NAME,StringType,true),
      StructField(ORIGIN_COUNTRY_NAME,StringType,true),
      StructField(count,LongType,true)))
```

StructField

name The name of this field.

dataType The data type of this field.

nullable Indicates if values of this field can be null values.