CSE 4510/5310
BIG DATA

Instructor: Fitzroy Nembhard, Ph.D.

Week 7-10

Apache Spark

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

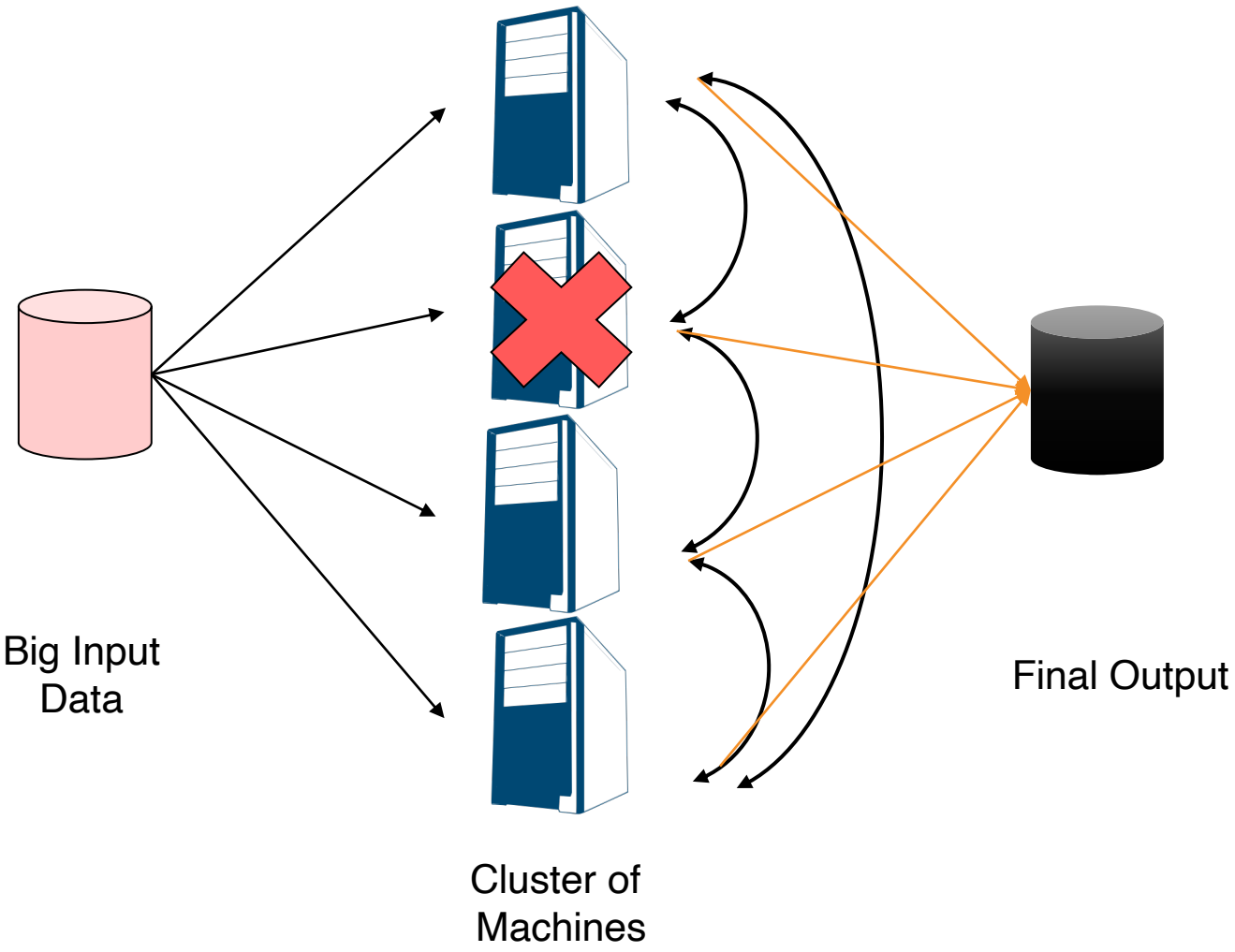
- All slides included in this class are for the exclusive use of students and instructors associated with Management and Processing of Big Data(CSE 4510/5310) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

Goals

- To discuss Distributed Systems
- To discuss MapReduce
- To discuss and apply Apache Spark to various problems
  - Datasets
  - DataFrames
  - SQL Tables
  - Resilient Distributed Datasets (RDDs)

7.1 DISTRIBUTED SYSTEMS

Distributed Data Processing



- Data partitioning
- Load balancing
- Fault tolerance
- Synchronization

MapReduce

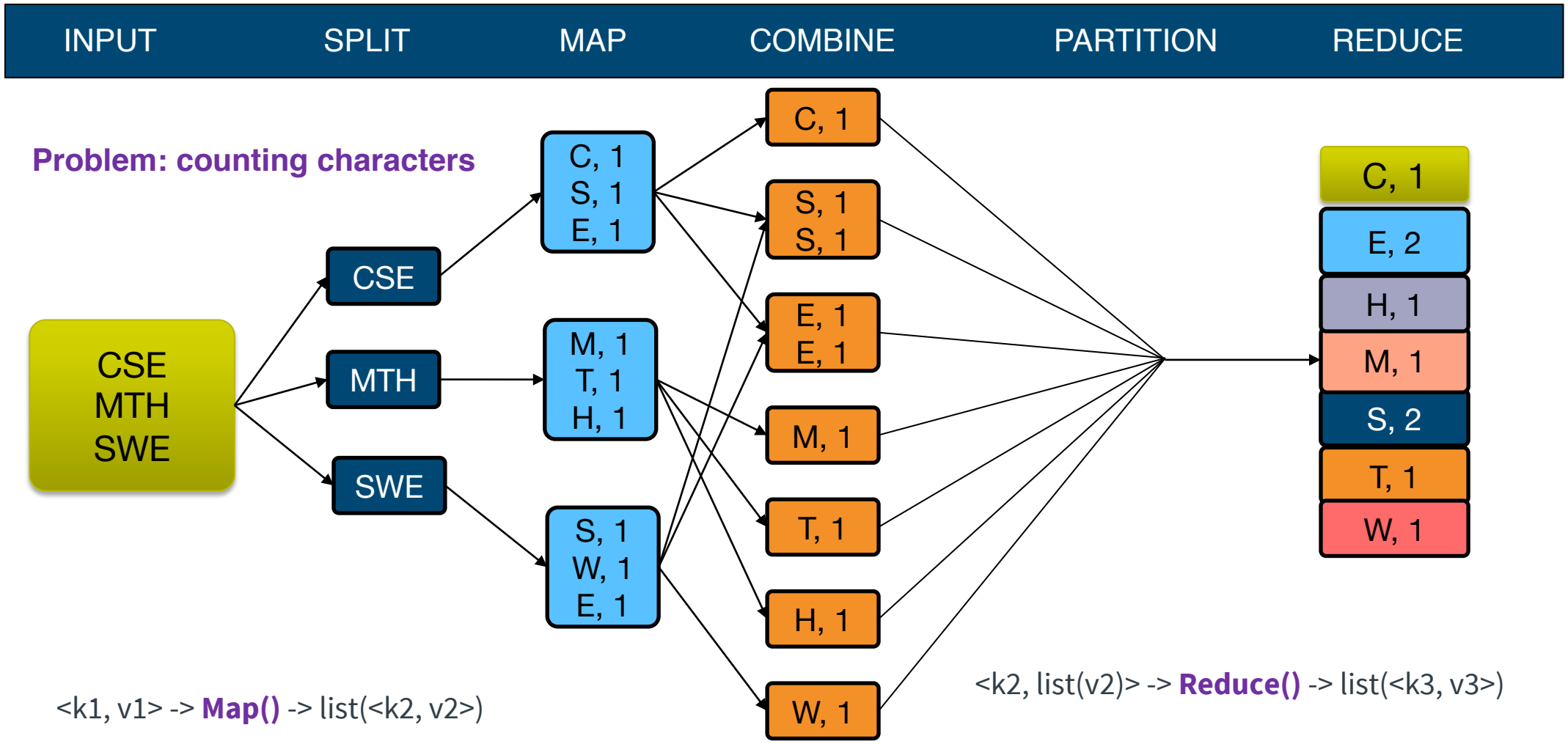
- A programing paradigm for expressing distributed algorithms
- Introduced by Google in 2004
  - Google File System (GFS) for distributed storage
  - Google MapReduce for distributed processing
- Hadoop is the open-source counterpart released in 2007 and contributed mainly by Yahoo!
  - HDFS
  - Hadoop MapReduce

MapReduce

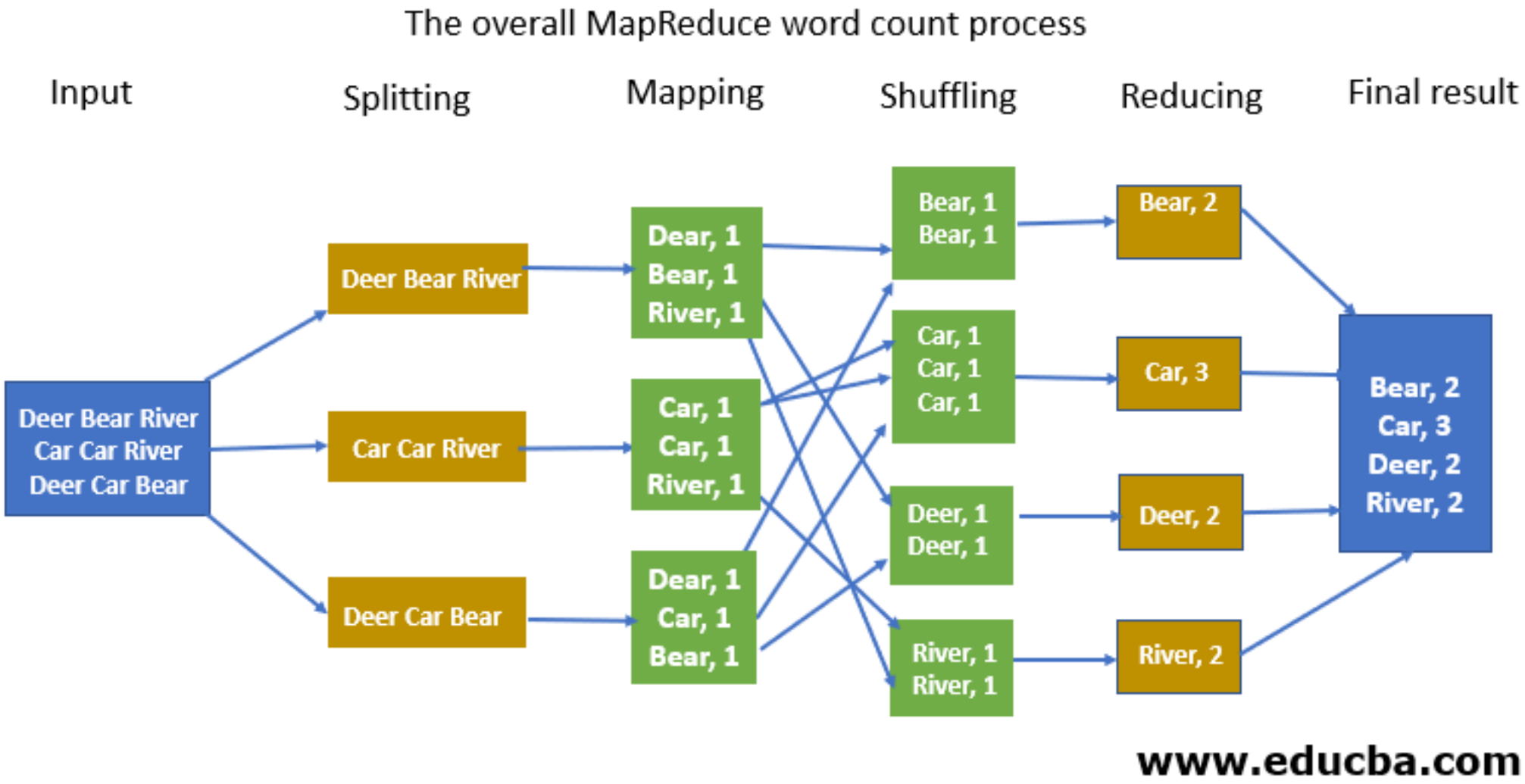
- The term **MapReduce** refers to two separate and distinct tasks with foundations in functional programming: ***map*** and ***fold***.
- **Map**
  - In functional programming, given a list, ***map*** takes as an argument a function ***f*** (that takes a single argument) and applies it to all element in a list.
- **Fold**
  - Given a list, **fold** takes as arguments a function ***g*** (that takes two arguments) and an initial value
  - ***g*** is first applied to the initial value and the first item in the list
  - The result is stored in an intermediate variable, which is used as an
  - input together with the next item to a second application of ***g***
  - *The process is repeated until all items in the list have been consumed*

- **Key-value pairs are the basic data structure in MapReduce**
  - Keys and values can be integers, float, strings, raw bytes
  - They can also be arbitrary data structures

MapReduce



MapReduce



Example MapReduce

- The programmer defines a mapper and a reducer as follows:
  - map: ($k1$, $v1$) → [($k2$, $v2$)]
  - reduce: ($k2$, [$v2$]) → [($k3$, $v3$)]

```
public static void main(String[] args) throws Exception {
  JobConf conf = new JobConf(ExceptionCount.class);
  conf.setJobName("wordcount");

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

  conf.setMapperClass(Map.class);
  conf.setReducerClass(Reduce.class);
  conf.setCombinerClass(Reduce.class);

  conf.setInputFormat(TextInputFormat.class);
  conf.setOutputFormat(TextOutputFormat.class);

  FileInputFormat.setInputPaths(conf, new Path(args[0]));
  FileOutputFormat.setOutputPath(conf, new Path(args[1]));

  JobClient.runJob(conf);
}
```

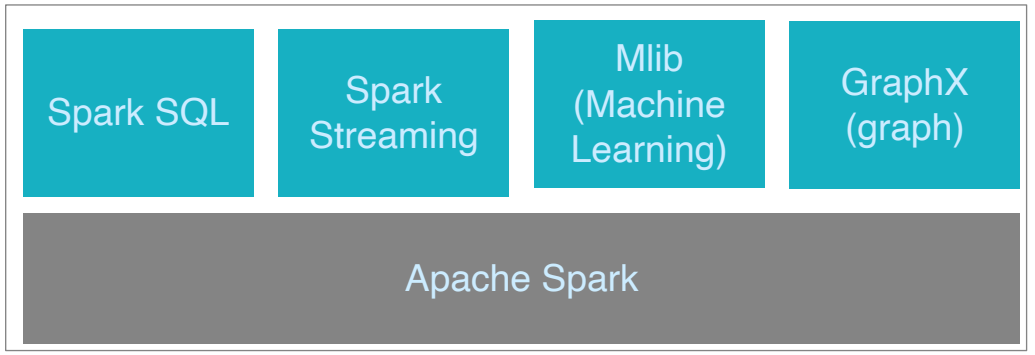
7.2 APACHE SPARK

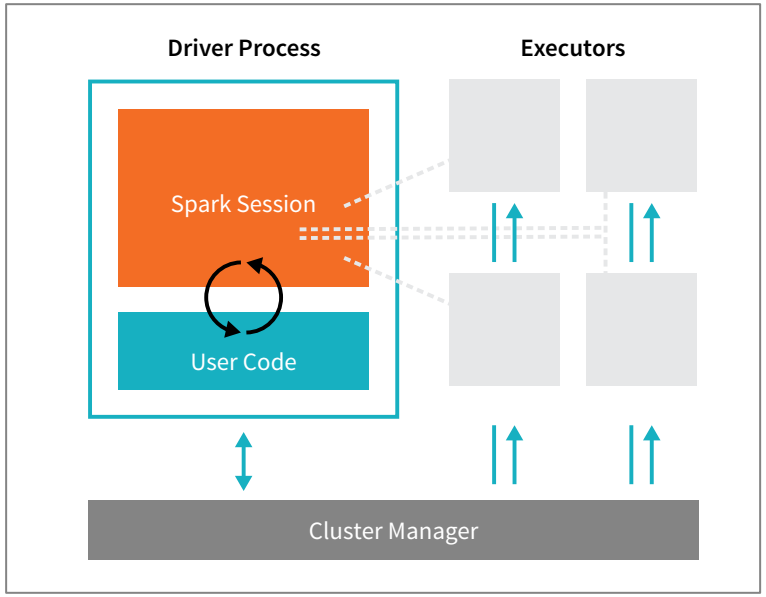
Spark

- Hadoop and MapReduce were a perfect research vehicle
- They helped in framing what we really want in a big data system
- Spark came as a new system designed from scratch to satisfy the real need of big data
- A distributed **shared-nothing** system
- Uses a functional programming paradigm

Apache Spark

- Apache Spark is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing
- The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos.
- There is also a local mode. The driver and executors are simply processes, which run (as threads) on your individual computer instead of a cluster

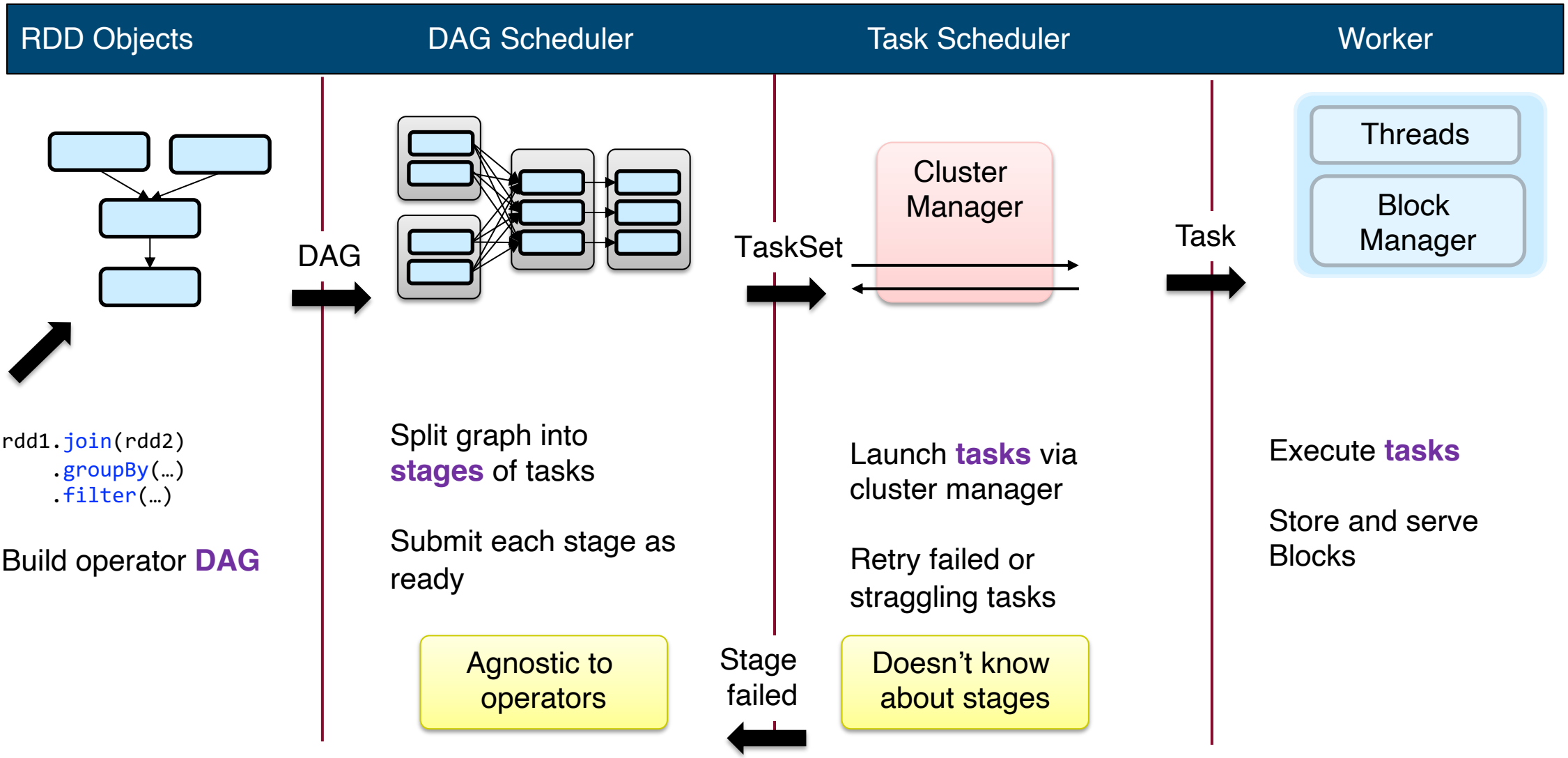




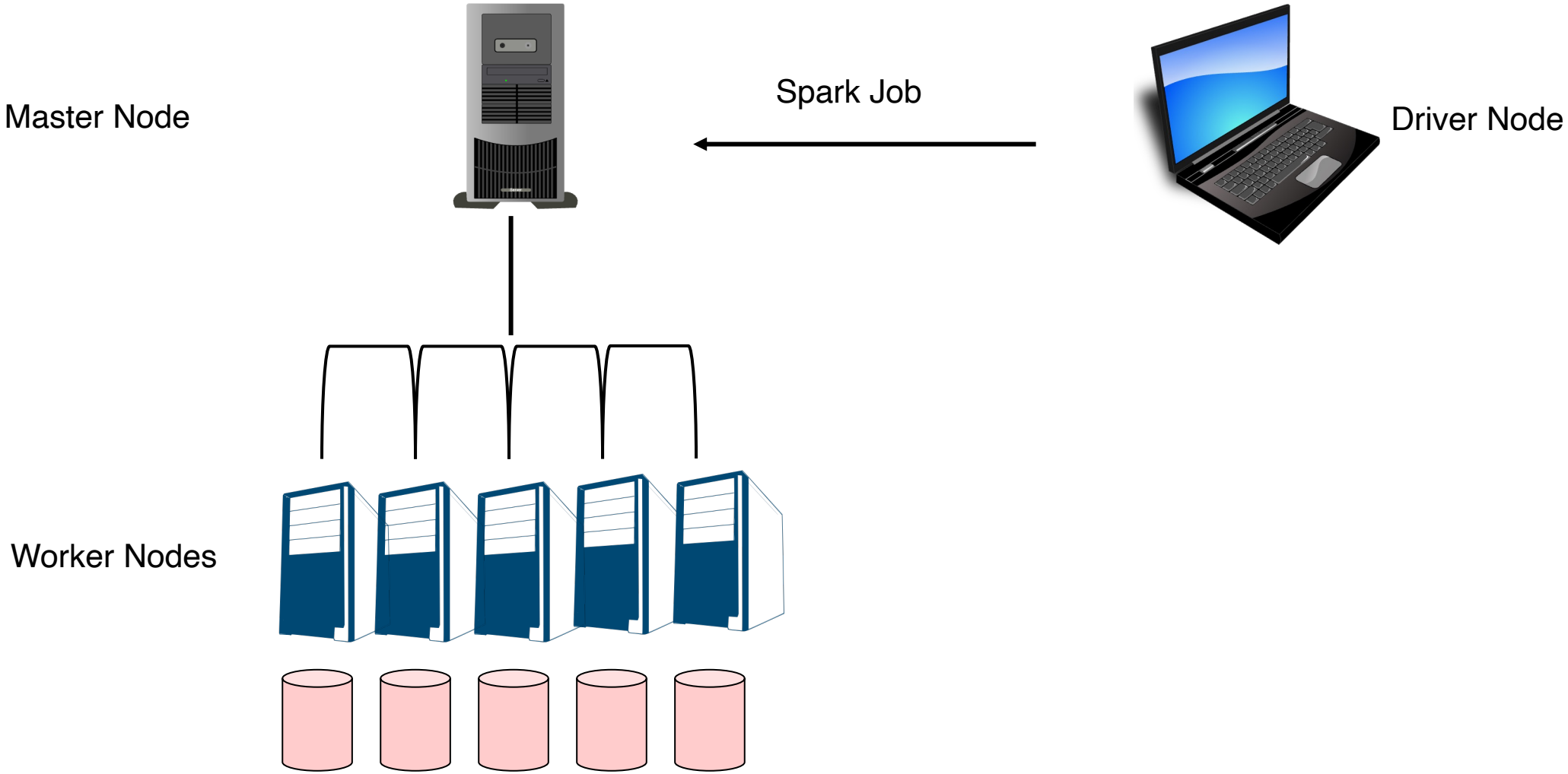
Spark Terminologies

- **Application**: A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.
- **SparkSession**: An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs. In an interactive Spark shell, the Spark driver instantiates a SparkSession for you, while in a Spark application, you create a SparkSession object yourself.
- **Job**: A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()).
- **Stage**: Each job gets divided into smaller sets of tasks called stages that depend on each other.
- **Task**: A single unit of work or execution that will be sent to a Spark executor.

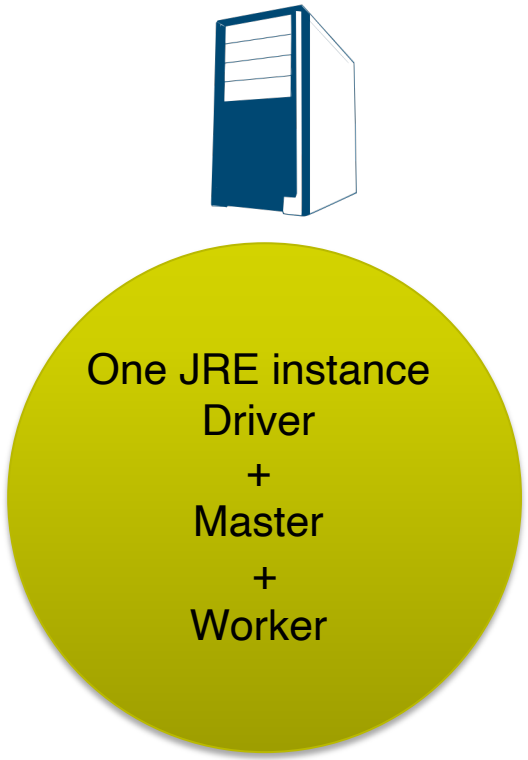
How Spark Works



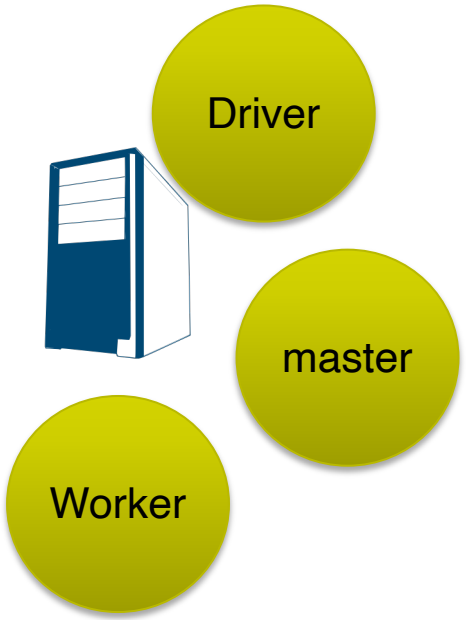
Spark Overview



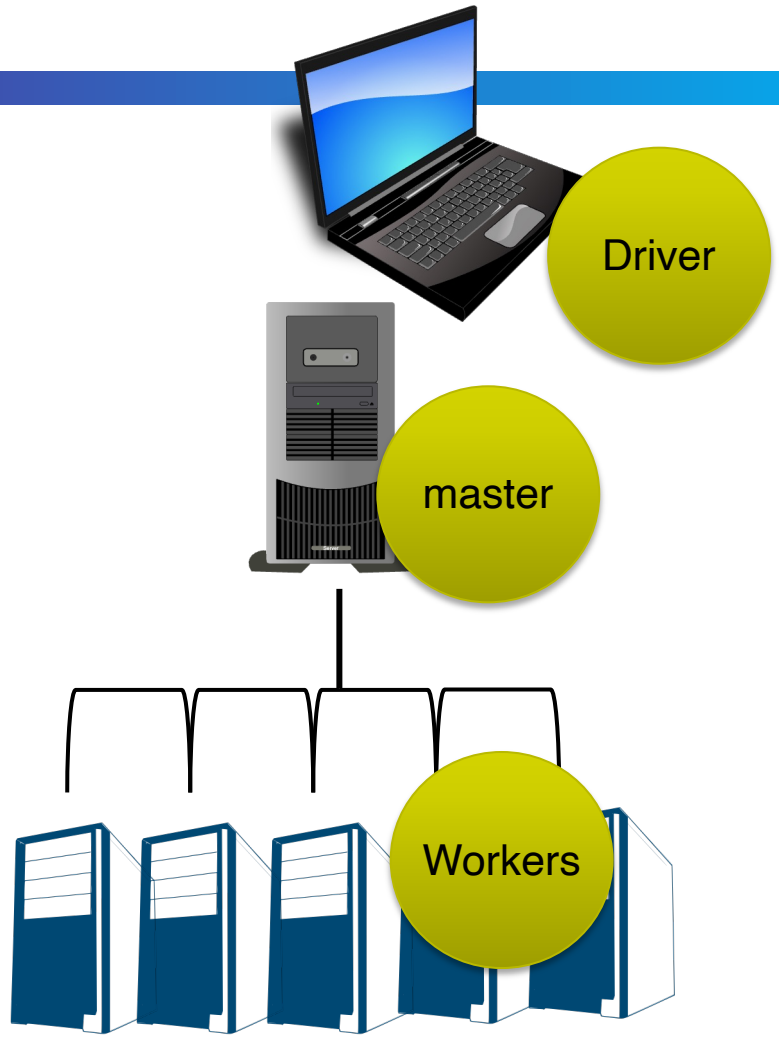
Spark Operation Modes



Local Mode



Standalone Mode



Cluster Mode

Spark Operation/Deployment Modes

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Learning Spark Lightning-Fast Data Analytics, Jules S. Damji et al

Apache Spark Language APIs

- Spark is primarily written in Scala, making it Spark's "default" language.
- **Java** - You can also write Spark code in Java
- **Python** supports nearly all constructs that Scala supports.
- **SQL** - Spark supports ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to leverage the big data powers of Spark.
- **R** - Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as a R community driven package (sparklyr)

Running Spark from the Console

- We control our Spark Application through a driver process. This driver process manifests itself to the user as an object called the SparkSession
- When run on a cluster, each part of this range of numbers exists on a different executor

- **Install Spark**
  - https://sparkbyexamples.com/spark/apache-spark-installation-on-windows/
  - https://sparkbyexamples.com/spark/install-apache-spark-on-mac/

```
spark # start spark from the console

my_range = spark.range(1000).toDF("number") # create a range of numbers. "number" is the column name
```

Running Spark from the Console

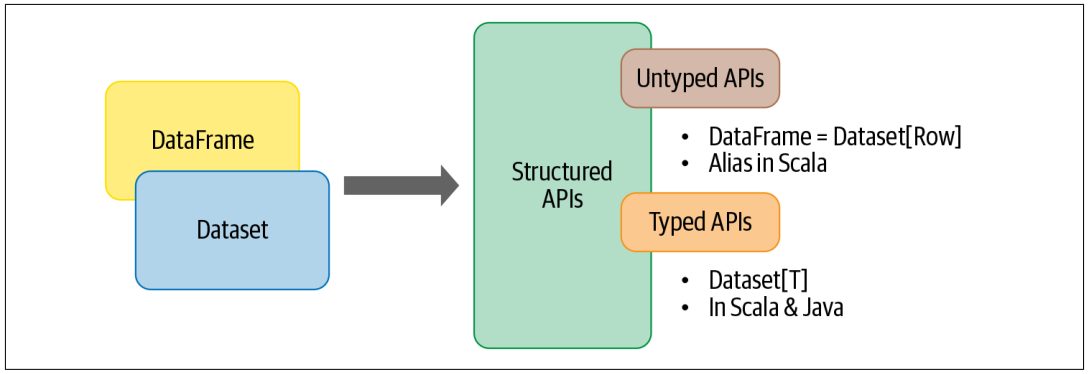
```
(base) fitzroi@Fitz-MacBook-Pro-2021 dask % spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(
newLevel).
22/10/04 20:56:18 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
Spark context Web UI available at http://fitz-mbp-2021.lan:4040
Spark context available as 'sc' (master = local[*], app id = local-1664931379043).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.3.0
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 19)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Running Spark from Jupyter

```
# pip install pyspark

Edit bashrc or bash_profile and add the following

export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'

./bin/pyspark # suppose you installed pyspark here
```

Spark Abstractions

- Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs)
- **RDD** (the most basic abstraction in Spark) is a pointer to a distributed dataset
- A **DataFrame** in Scala is an alias for a collection of generic objects, Dataset[Row], where a **Row** is a generic untyped JVM object that may hold different types of fields.
- A **Dataset**, by contrast, is a collection of strongly typed JVM objects in Scala or a class in Java.



Language	Typed and untyped main abstraction	Typed or untyped
Scala	`Dataset[T]` and DataFrame (alias for `Dataset[Row]`)	Both typed and untyped
Java	`Dataset<T>`	Typed
Python	DataFrame	Generic Row untyped
R	DataFrame	Generic Row untyped

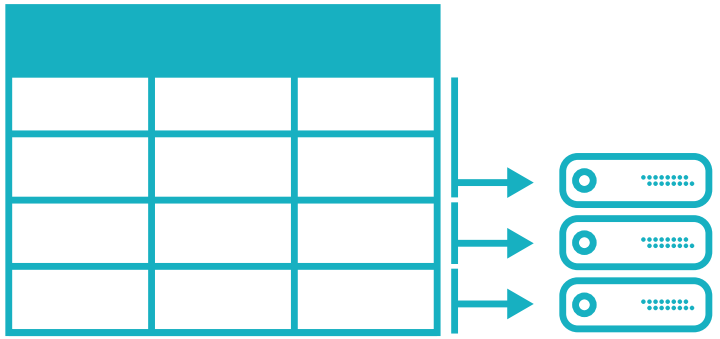
Learning Spark Lightning-Fast Data Analytics, Jules S. Damji et al

Spark DataFrames

- Since Spark has language interfaces for Python, it's quite easy to convert Pandas (Python) DataFrames to Spark DataFrames

Spreadsheet on a single machine

Table or DataFrame partitioned across servers in data center



Spark Partitions

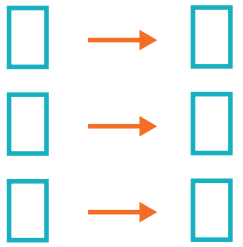
- Spark breaks up the data into **chunks**, called **partitions**.
- A partition is a collection of rows that sit on one physical machine in our cluster.
- A DataFrame's partitions represent how the data is physically distributed across your cluster of machines during execution

Spark Transformations & Actions (Example 1)

- In Spark, the core data structures are immutable
- In order to "change" a DataFrame, you will have to instruct Spark how you would like to modify the DataFrame
- Spark also uses **Lazy** evaluation -- it waits until the very last moment to execute the graph of computation instructions

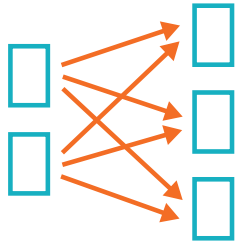
```
./bin/pyspark # Start spark from the console

my_range = spark.range(1000).toDF("number")

# narrow transformation on a DataFrame
divis_by2 = my_range.where("number % 2 = 0")

divis_by2.count() # An action. We see the result (500).
# There are also actions to view, collect, and write output
```

Narrow Transformations
1 to 1



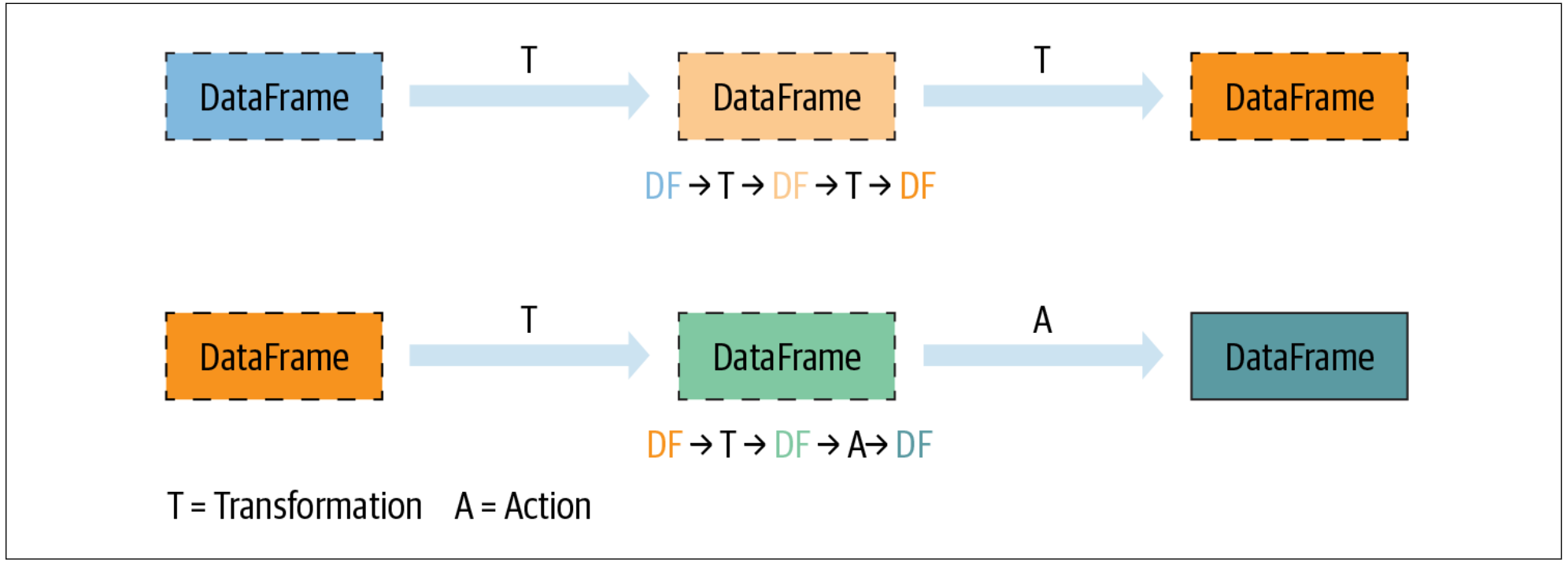
Wide Transformations (shuffles)

1 to many



Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

Lazy Transformations and Eager Actions



Learning Spark Lightning-Fast Data Analytics, Jules S. Damji et al

More Spark Transformations

Essential Core & Intermediate Spark Operations

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

Essential Core & Intermediate PairRDD Operations

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure

- partitionBy

Learning Apache Spark with Python, Wenqiang Feng

More Spark Actions

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

- keys
- values

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

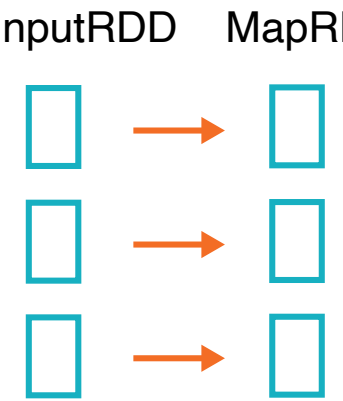
Imitating Map Reduce with Spark RDD (Example 2)

```
lines = sc.textFile("data.txt")
line_lengths = lines.map(lambda s: len(s))
total_length = line_lengths.reduce(lambda a, b: a + b)
```

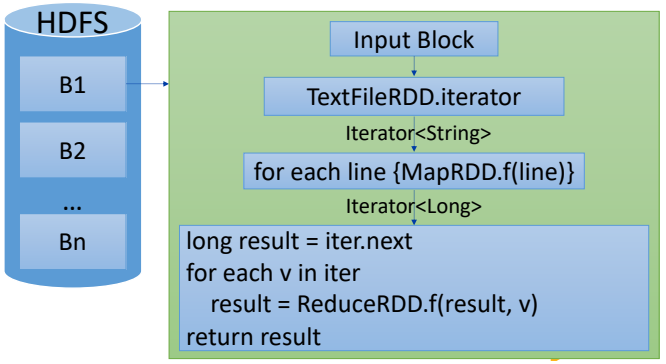
- The partitions are defined based on the metadata
- The file is not opened

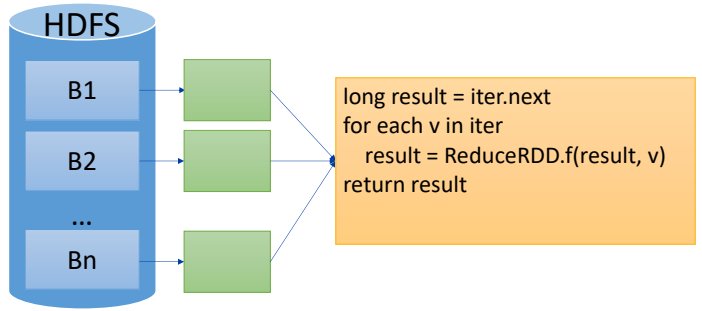
PartitionID: 0→n-1
File: Path
Offset: Long
Length: Long
Locations: String[]

- Transformation



- Reduce Action (a+b)





Imitating Map Reduce with Spark RDD (Example 3)

```
words_file = 'dataset.txt'

# read data from text file and split each line into words
words = sc.textFile(words_file).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("data/count")
```

Reduce vs ReduceByKey

REDUCE	REDUCEBYKEY
Action	**Transformation**
reduce must pull the entire dataset down into a single location because it is reducing to one final value	reduceByKey reduces one value for each key
reduce() is a function that operates on an RDD of objects.	reduceByKey() is a function that operates on an RDD of key-value pairs
reduce() function is a member of RDD[T] class	reduceByKey() is a member of the PairRDDFunctions[K, V] class
Output is a collection not an RDD and is is not added to DAG .	Output is an RDD and is added to DAG The reduce cannot result in an RDD simply because it is a single value as output.
def reduce(f: (T, T) => T): T	def reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]
reduce is an action which Aggregate the elements of the dataset using a function func (which takes two arguments and returns one)	reduceByKey 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.

Spark Example – Data Filtering in Python (Example 4)

- This program counts the total number of lines vs the number of lines that have the word python in a file named my_dataset.

```
import pyspark
sc = pyspark.SparkContext('local[*]')

txt = sc.textFile('file:////my_dataset')
print(txt.count())

python_lines = txt.filter(lambda line: 'python' in line.lower())
print(python_lines.count())
```

Spark RDD Example – Word Count in Java (Example 5)

- This program counts the total number of lines in a file named words.

```
SparkSession spark = SparkSession
        .builder()
        .appName("Dr. Fitz's App")
        .master("local[*]")
        .getOrCreate();



JavaRDD<String> textFileRDD = sparkContext.textFile("./src/data/words");

long count = textFileRDD.count();
System.out.println("Number of lines is " + count);
```

PySpark Errors

```
 9 # save the counts to output
10 wordCounts.saveAsTextFile("data/imdb_countries")
11 wordCounts
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
<ipython-input-47-7c6e4cce09b5> in <module>
      8
      9 # save the counts to output
---> 10 wordCounts.saveAsTextFile("data/imdb_countries")
     11 wordCounts

~/opt/anaconda3/lib/python3.8/site-packages/pyspark/rdd.py in saveAsTextFile(self, path, compressionCodecClass)
   2203             keyed._jrdd.map(self.ctx._jvm.BytesToString()).saveAsTextFile(path, compressionCodec)
   2204         else:
-> 2205             keyed._jrdd.map(self.ctx._jvm.BytesToString()).saveAsTextFile(path)
   2206
   2207     # Pair functions

~/opt/anaconda3/lib/python3.8/site-packages/py4j/java_gateway.py in __call__(self, *args)
   1319
   1320         answer = self.gateway_client.send_command(command)
-> 1321         return_value = get_return_value(
   1322             answer, self.gateway_client, self.target_id, self.name)
   1323

~/opt/anaconda3/lib/python3.8/site-packages/pyspark/sql/utils.py in deco(*a, **kw)
    188     def deco(*a: Any, **kw: Any) -> Any:
    189         try:
--> 190             return f(*a, **kw)
    191         except Py4JJavaError as e:
    192             converted = convert_exception(e.java_exception)

~/opt/anaconda3/lib/python3.8/site-packages/py4j/protocol.py in get_return_value(answer, gateway_client, target_id, nam
    324             value = OUTPUT_CONVERTER[type](answer[2:], gateway_client)
    325             if answer[1] == REFERENCE_TYPE:
--> 326                 raise Py4JJavaError(
    327                     "An error occurred while calling {0}{1}{2}.\n".
    328                     format(target_id, ".", name), value)

Py4JJavaError: An error occurred while calling o334.saveAsTextFile.
: org.apache.hadoop.mapred.FileAlreadyExistsException: Output directory file:/Users/fitzroi/Library/Mobile Documents/co
imdb_countries already exists
        at org.apache.hadoop.mapred.FileOutputFormat.checkOutputSpecs(FileOutputFormat.java:131)
        at org.apache.spark.internal.io.HadoopMapRedWriteConfigUtil.assertConf(SparkHadoopWriter.scala:299)
        at org.apache.spark.internal.io.SparkHadoopWriter$.write(SparkHadoopWriter.scala:71)
        at org.apache.spark.rdd.PairRDDFunctions.$anonfun$saveAsHadoopDataset$1(PairRDDFunctions.scala:1091)
        at scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.java:23)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:406)
        at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopDataset(PairRDDFunctions.scala:1089)
        at org.apache.spark.rdd.PairRDDFunctions.$anonfun$saveAsHadoopFile$4(PairRDDFunctions.scala:1062)
        at scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.java:23)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
        at org.apache.spark.rdd.RDD.withScope(RDD.scala:406)
        at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.scala:1027)
        at org.apache.spark.rdd.PairRDDFunctions.$anonfun$saveAsHadoopFile$3(PairRDDFunctions.scala:1009)
        at scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.java:23)
        at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
```

Spark RDD Example – Word Count & Filtering in Java (Example 6)

- This program counts the total number of lines that start with the letter ‘A’ in a file named words.

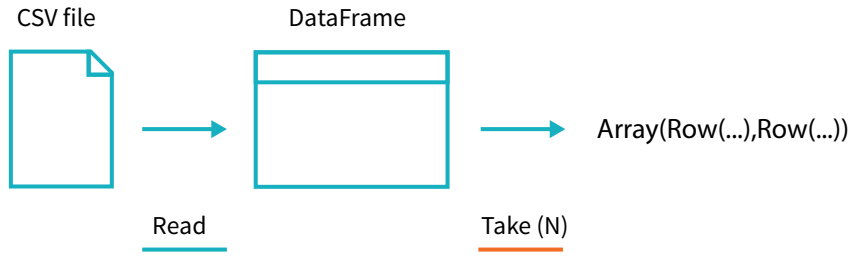
```
SparkSession spark = SparkSession
        .builder()
        .appName(”Dr. Fitz’s App")
        .master("local[*]")
        .getOrCreate();


JavaRDD<String> textFileRDD = sparkContext.textFile("./src/data/words");

JavaRDD<String> wordsStartingWithA = textFileRDD.filter((Function<String, Boolean>) s ->
        s.substring(0,1).equalsIgnoreCase("A"));

System.out.println(wordsStartingWithA.count());
```

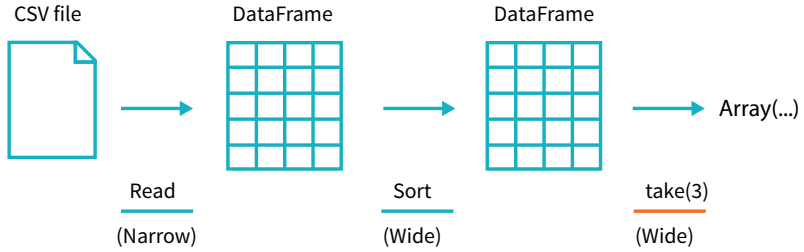
Spark RDD Example – Log Filtering in Java Practice

- Given a SSH log file https://raw.githubusercontent.com/logpai/loghub/master/OpenSSH/SSH_2k.log
- Count the number of lines containing “failed password”

Spark Example – the "take" action

- To read data, we use a DataFrameReader that is associated with our SparkSession
- DataFrames from a read operation have a set of columns with an unspecified number of rows because reading data is a transformation.
- If we perform the **take action** on the DataFrame, we will be able to see results
- Sort does not modify the DataFrame
- We can call explain on any DataFrame object to see the DataFrame's lineage (or how Spark will execute this query).






```
./bin/pyspark # Start spark from the console

flight_data_2015 = spark\
.read\
.option("inferSchema", "true")\
.option("header", "true")\
.csv("2015-summary.csv") # Locate dataset on Canvas

flight_data_2015.take(3)
# Array([United States,Romania,15], [United States,Croatia...

flight_data_2015.sort("count").explain()
```

When to use DataFrames vs Datasets

- If you want to tell Spark what to do, not how to do it, use DataFrames or Datasets. If you want rich semantics, high-level abstractions, and DSL operators, use DataFrames or Datasets.
- If you want strict compile-time type safety and don't mind creating multiple case classes for a specific Dataset[T], use Datasets.
- If your processing demands high-level expressions, filters, maps, aggregations, computing averages or sums, SQL queries, columnar access, or use of relational operators on semi-structured data, use DataFrames or Datasets.
- If your processing dictates relational transformations similar to SQL-like queries, use DataFrames.
- If you want to take advantage of and benefit from Tungsten's efficient serialization with Encoders, use Datasets.
- If you want unification, code optimization, and simplification of APIs across Spark components, use DataFrames.
- If you are an R user, use DataFrames.
- If you are a Python user, use DataFrames and drop down to RDDs if you need more control.
- If you want space and speed efficiency, use DataFrames.

Learning Spark Lightning-Fast Data Analytics, Jules S. Damji et al

When to Use RDDs

- Are using a third-party package that's written using RDDs
- Can forgo the code optimization, efficient space utilization, and performance benefits available with DataFrames and Datasets
- Want to precisely instruct Spark *how to do* a query

Learning Spark Lightning-Fast Data Analytics, Jules S. Damji et al

SPARK RDD OPERATIONS

Spark Narrow and Wide

- In Spark RDD, you can generally think of these two rules
  - Narrow dependency➔Local processing
  - Wide dependency➔Network communication

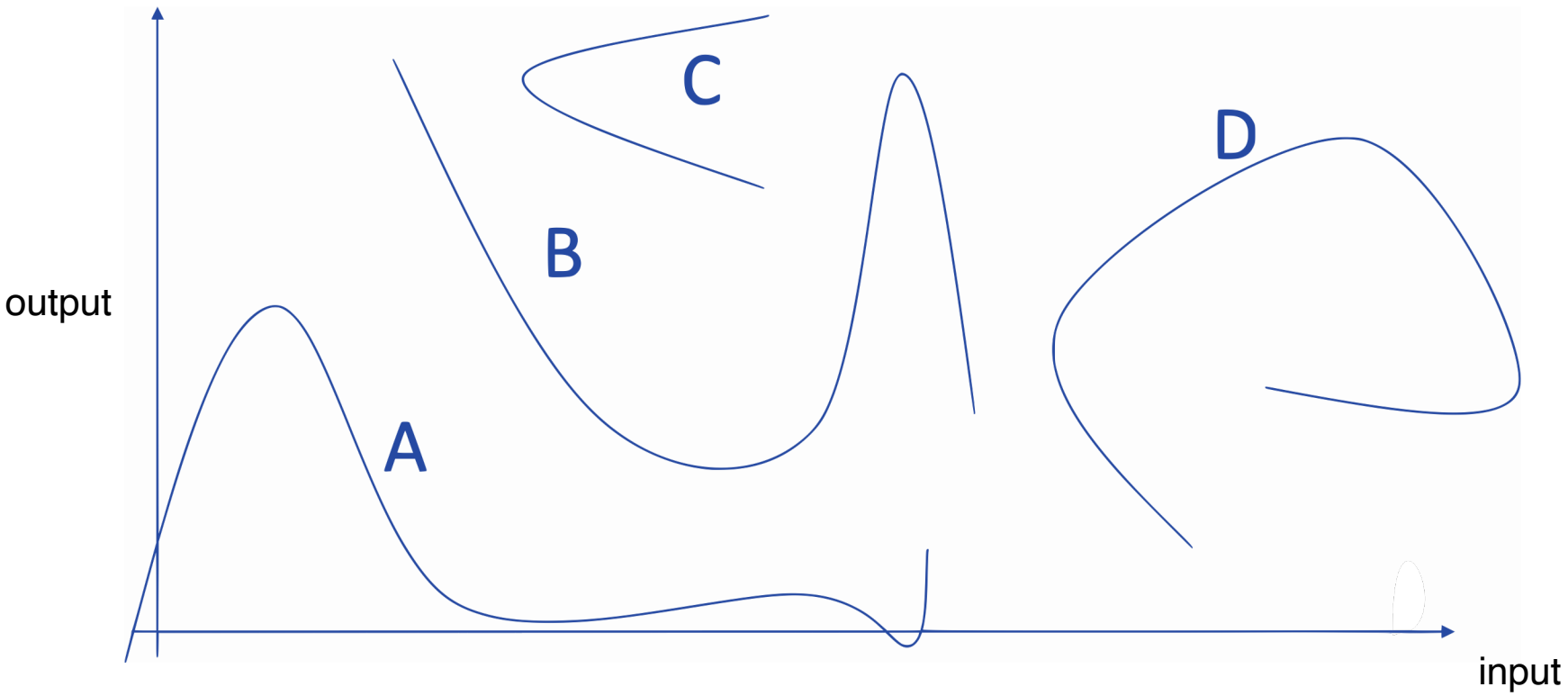
Local Processing in RDD

- A simple abstraction for local processing
- Based on functional programming

```
public void LocalProcessing(input: Iterator<T>, output:
Writer<U>) {
    ... // output.write(U)
}
```

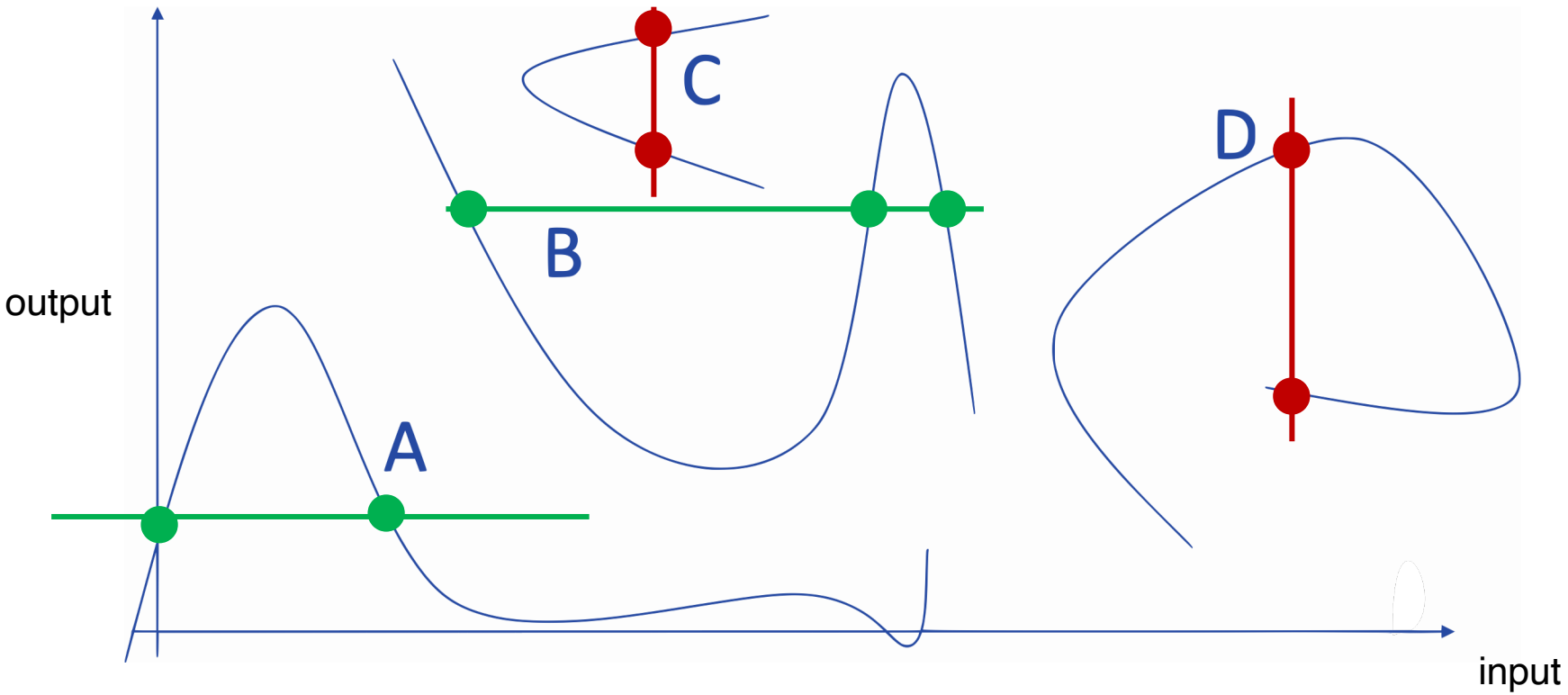
Functional Programming

- RDD is a functional programming paradigm
- Which of these are functions?



Functional Programming

- RDD is a functional programming paradigm
- Which of these are functions?



Functional Programming

Relations That Are Not Functions

A function is a relation between domain and range such that **each value in the domain corresponds to only one value in the range.**

Relations that are not functions violate this definition.
They feature at least one value in the domain that corresponds to two or more values in the range.

Function Limitations

- For one input, the function should return one output
- The function should be memoryless
  - Should not remember past input
- The function should be stateless
  - Should not change any state when called
- It is up to the developer to enforce these properties

Examples

```
Function1(x){
    return x + 5;
}
```

```
RNG random;
Function3(x) {
    random.randomInt(0, x);
}
```

```
Int sum
Function2(x) {
 sum += x;
 return sum;
}
```

```
Map<String, Int> lookuptable;
Function4(x) {
  return lookuptable.get(x);
}
```

Network Communication

- a.k.a. shuffle operation
- Given a record r and n partitions:
  - Assign the record to one of the partitions [0, n – 1]

Spark Operations

- Spark is rich with operations
- Sometimes, you can do the same logic in multiple ways
- In the subsequent slides, we will explain how different RDD operations work
- The goal is to understand the performance implications of these operations and choose the most efficient one

RDD<T>#filter

- Filter is a `function{T→Boolean}`
- Applies the predicate function on each record and produces that tuple only of the predicate returns true
- Result `RDD<T>` with same or fewer records than the input

```
Local Processing {
    for each (t in input) {
        if (func(t))
            output.write(t)
    }
}
```

RDD<T>#map(func)

- `func: T→U`
- Applies the map function to each record in
- the input to produce one record
- Results in RDD<U> with the same number of records as the input

```
Local Processing {
    for each (t in input)
        output.write(func(t))
}
```

RDD<T>#flatMap(func)

- `func: T→Iterator<V>`
- Applies the map function to each record and add all resulting values to the output RDD
- Result: RDD<V>
- This is the closest function to the Hadoop map function

```
Local Processing {
    Iterator<V> results = func(input);
    for (V result : results)
        output.write(result)
}
```

RDD<T>#mapPartition(func)

- func: Iterator<T>→Iterator<U>
- Applies the map function to a list of records in one partition in the input and adds all resulting values to the output RDD
- Can be helpful in two situations
  - If there is a costly initialization step in the function
  - If many records can result in one record
- Result: RDD<U>

```
Local Processing {
    results = func(input)
    for-each (v in results)
        output.write(v);
}
```

RDD<T>#mapPartitionWithIndex(func)

- `func: (Integer, Iterator<T>)→ Iterator<U>`
- Similar to mapPartition but provides a unique index for each partition
- To achieve this in Spark, the partition ID is passed to the function

RDD<T>#sample(r, f, s)

- r: Boolean: With replacement (true/false)
- f: Float: Fraction [0,1]
- s: Long: Seed for random number generation
- Returns RDD<T> with a sample of the records in the input RDD
- Can be implemented using mapPartitionWithIndex as follows
  - Initialize the random number generator based on seed and partition index
  - Select a subset of records as desired
  - Return the sampled records

RDD<T>#reduce(func)

- `func: (T, T)→T`
- Reduces all the records to a single value by repeatedly applying the given function
- The function should be associative and commutative
- Result: T
- This is an action

- **Shuffle**: assign all records to one partition
- Collect partial results and apply the same function again

```
mapPartition {
    T result = input.next
    for each (r in input)
        result = reduce(result, r)
    return result
}
```

RDD<K,V>#reduceByKey(func)

- func: (V, V)→V
- Similar to reduce but applies the given function to each group separately
- Since there could be many groups, this operation is a transformation that can be followed by further transformations and actions
- Result: RDD<K,V>
- By default, number of reducers is equal to number of input partitions but can be overridden

```
mapPartition{
    Map<K,V> results;
    for each ((k,v) in input) {
        if (results.contains(k))
            results[k] = reduce(results[k], v);
        else
            results[k] = v;
    }
}
```

```
mapPartition{
    // All input records have the same key
    V result = value.next
    for each (v in values)
        result = reduce(result, v)
    output.write(k, v)
}
```

RDD<T>#distinct()

- Removes duplicate values in the input RDD
- Returns RDD<T>
- Implemented as follows:

```
map(x => (x, null))
    .reduceByKey((x, y) => x, numPartitions)
    .map(_._1)
```

Limitation of Reduce Methods

- Both reduce methods have a limitation is that they have to return a value of the same type as the input.
- Let us say we want to implement a program that operates on an RDD<Integer> and returns one of the following values
  - 0: Input is empty
  - 1: Input contains only odd values
  - 2: Input contains only even values
  - 3: Input contains a mix of even and odd values

RDD<T>#aggregate(zero, seqOp, combOp)

- zero: U - Zero value of type U
- seqOp: (U, T)→U – Combines the aggregate
- value with an input value
- combOp: (U, U)→U – Combines two aggregate values
- Like reduce, aggregate is an action
- Returns U
- Similarly, aggregateByKey is a transformation that takes RDD<K,V> and returns RDD<K,U>

```
mapPartition {
    U partialResult = zero
    for each (t in input)
        result = seqOp(partialResult, t)
    return partialResult
}
```

- Collect all partial results into one partition

```
mapPartition
{
    U finalResult = input.next
    for each(u in input)
        finalResult = combOp(finalResult, u)
    return finalResult
}
```

RDD<K,V>#groupByKey()

- Groups all values with the same key into the same partition
- Closest to the shuffle operation in Hadoop
- Returns RDD<K, Iterator<V>>
- Performance notice: By default, all values are kept in memory so this method can be very memory consuming.
- Unlike the reduce and aggregate methods, this method does not run a combiner step, i.e., all records get shuffled over the network

MORE ON DATAFRAMES

Basic Python Data Types in Spark

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Python Structured Data Types in Spark

Data type	Value assigned in Python	API to instantiate
`BinaryType`	`bytearray`	`BinaryType()`
`TimestampType`	`datetime.datetime`	`TimestampType()`
`DateType`	`datetime.date`	`DateType()`
`ArrayType`	List, tuple, or array	`ArrayType(dataType, [nullable])`
`MapType`	`dict`	`MapType(keyType, valueType, [nullable])`
`StructType`	List or tuple	`StructType([fields])`
`StructField`	A value type corresponding to the type of this field	`StructField(name, dataType, [nullable])`

Schemas and Creating DataFrames

- A *schema* in Spark defines the column names and associated data types for a Data- Frame
- Two ways to define a schema

```
from pyspark.sql.types import *

schema = StructType([StructField("author", StringType(), False),
                     StructField("title", StringType(), False),
                     StructField("pages", IntegerType(), False)])



schema = "author STRING, title STRING, pages INT"
```

Using the Schema with Static Data

```
# Create our static data
data = [[1, "Jules", "Damji", "https://tinyurl.1", "1/4/2016", 4535, ["twitter", "LinkedIn"]],
       [2, "Brooke", "Wenig", "https://tinyurl.2", "5/5/2018", 8908, ["twitter", "LinkedIn"]],
       [3, "Denny", "Lee", "https://tinyurl.3", "6/7/2019", 7659, ["web", "twitter", "FB", "LinkedIn"]],
       [4, "Tathagata", "Das", "https://tinyurl.4", "5/12/2018", 10568,  ["twitter", "FB"]],
       [5, "Matei", "Zaharia", "https://tinyurl.5", "5/14/2014", 40578, ["web","twitter", "FB", "LinkedIn"]],
       [6, "Reynold", "Xin", "https://tinyurl.6", "3/2/2015", 25568, ["twitter", "LinkedIn"]]
       ]
```

```
spark = (SparkSession
        .builder
        .appName("Spark DataFrame Example")
        .getOrCreate())

 # Create a DataFrame using the schema defined above
blogs_df = spark.createDataFrame(data, schema)
# Show the DataFrame; it should reflect our table above
blogs_df.show()
# Print the schema used by Spark to process the DataFrame
print(blogs_df.printSchema())
```

Reading in Data from A JSON File

```
# Create a DataFrame by reading from the JSON file
# with a predefined schema
blogs_df = spark.read.schema(schema).json(jsonFile)


# Show the DataFrame schema as output
blogs_df.show(false)
```

SPARK SQL

Structured Data Processing

- Spark RDD is good for general-purpose processing
- For (semi-)structured data, you need to provide your own parser and logic
- Due to the popularity of (semi-) structured data processing, SparkSQL was added to facilitate this task

SparkSQL

- Supports all the popular relational operators
- Can be intermixed with RDD operations
- Uses the Dataframe API as an enhancement to the RDD API
- DataFrame = RDD + Schema

SparkSQL

- SparkSQL's counterpart to relations or tables in RDMBS
- Consists of rows and columns
- A dataframe is NOT in 1NF (Why?)
- Can be created from various data sources
  - CSV file
  - JSON file
  - MySQL database
  - Hive
  - Parquet column-format files

Spark DataFrame vs RDD

DataFrame

- Lazy execution
- Spark is aware of the data model
- Spark is aware of the query logic
- Can optimize the query

RDD

- Lazy execution
- The data model is hidden from Spark
- The transformations and actions are black boxes
- Cannot optimize the query

Built-in Operations in SprkSQL

- Filter (Selection)
- Select (Projection)
- Join
- GroupBy(Aggregation)
- Load/Store in various formats
- Cache
- Conversion between RDD (back and forth)

SPARKSQL EXAMPLES

SparkSQL Example 1

```
SparkSession spark = SparkSession
        .builder()
        .appName(”Dr. Fitz’s App")
        .master("local[*]")
        .getOrCreate();
```

```
Dataset<Row> log_file = spark
        .read()
        .option("delimiter", "\t")
        .option("header", "true")
        .option("inferSchema", "true")
        .csv("nasa_log.tsv");
log_file.show();
```

```
Dataset<Row> errorLines =
        log_file.filter("");
long error_count = errorLines.count();
System.out.println("Number of error lines is" + error_count);
```

RDD vs SQL/DataFrame

```
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average
agesRDD = (dataRDD
        .map(lambda x: (x[0], (x[1], 1)))
        .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
        .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg
# Create a DataFrame using SparkSession
spark = (SparkSession
        .builder
        .appName("AuthorsAges")
        .getOrCreate())
   # Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),
                                 ("TD", 35), ("Brooke", 25)], ["name", "age"])

# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

SparkSQL Example 2

```
./bin/pyspark # Start spark from the console

sql_way = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1) FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")

data_frame_way = flight_data_2015\
                      .groupBy("DEST_COUNTRY_NAME")\
                      .count()
sql_way.explain()
data_frame_way.explain() # Explain the DataFrame's lineage (or how Spark will execute this query)

############
spark.sql("SELECT max(count) from flight_data_2015").take(1)

from pyspark.sql.functions import max

flight_data_2015.select(max("count")).take(1)
```

SparkSQL Example 3

Version 1

```
max_sql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

max_sql.collect()
```

Version 2

```
from pyspark.sql.functions import desc
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.collect()
```

The End