CS x55: Distributed Systems [Spark]

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- *Before* an action is performed on an RDD it isn't "stored"? Where is it? And for how long?
- Are there performance differences between Spark when writing programs in Scala or Java?
- Where are Spark lineage graphs stored?
- Are all transformations implemented using MapReduce under the hood?

2

Topics covered in this lecture

- Actions on RDDs
- Pair RDDs
- Data Frames

3

4

Actions on Basic RDDs

- `reduce()`
  - Takes a function that operates on two elements in the RDD; returns an element of the same type
    - E.g., of such an operation? + sums the RDD

```
sum = rdd.reduce((x,y) => x + y)
```

- `fold()` takes a function with the same signature as `reduce()`, but also takes a "zero value" for initial call
  - "Zero value" is the **identity element** for initial call
  - E.g., 0 for +, 1 for *, empty list for concatenation

Both `fold()` and `reduce()` require return type of same type as the RDD elements

- The `aggregate()` removes that constraint
  - For e.g., when computing a running average, maintain both the count so far and the number of elements

Examples: Basic Actions on RDDs

Computer Science Department
Colorado State University

7

Examples: Basic actions on RDDs [1/7]

- Our RDD contains {1, 2, 3, 3}
- `collect()`
  - Return all elements from the RDD
  - Invocation: `rdd.collect()`
  - Result: {1, 2, 3, 3}

Colorado State University | Professor: Shrideep Pallickara, Computer Science Department | Spark | L31.8

8

Examples: Basic actions on RDDs [2/7]

- Our RDD contains {1, 2, 3, 3}
- `count()`
  - Number of elements in the RDD
  - Invocation: `rdd.count()`
  - Result: 4

Examples: Basic actions on RDDs [3/7]

- Our RDD contains {1, 2, 3, 3}
- `countByValue()`
  - Number of times each element occurs in the RDD
  - Invocation: `rdd.countByValue()`
  - Result: { (1,1), (2,1), (3,2) }

Examples: Basic actions on RDDs [4/7]

- Our RDD contains {1, 2, 3, 3}
- **`take(num)`**
  - Return `num` elements from the RDD
  - Invocation: `rdd.take(2)`
  - Result: { 1, 2}

Examples: Basic actions on RDDs [5/7]

- Our RDD contains {1, 2, 3, 3}
- **`reduce(func)`**
  - Combine the elements of the RDD together in parallel
  - Invocation: `rdd.reduce( (x,y) => x + y )`
  - Result: 9

Examples: Basic actions on RDDs [6/7]

- Our RDD contains {1, 2, 3, 3}
- **aggregate(zeroValue)(seqOp, combOp)**
  - Similar to `reduce()` but used to return a different type
  - Invocation:
    - ```
      rdd.aggregate ( (0,0))
                     ((x,y) => (x._1 + y, x._2 + 1),
                     (x,y) => (x._1 + y._1, x._2 + y._2))
      ```
  - Result: (9, 4)

Examples: Basic actions on RDDs [7/7]

- Our RDD contains {1, 2, 3, 3}
- **foreach(func)**
  - Apply the provided function to each element of the RDD
  - Invocation: `rdd.foreach(func)`
  - Result: Nothing
      ```

PERSISTENCE (CACHING)

15

Why persistence?

- Spark RDDs are lazily evaluated, and we may sometimes wish to use the same RDD multiple times
  - Naively, Spark will **recompute RDD and all of its dependencies** each time we call an action on the RDD
    - Super expensive for iterative algorithms
- To avoid recomputing RDD multiple times?
  - Ask Spark to **persist** the data
  - The nodes that compute the RDD, store the partitions
  - E.g.: `result.persist(StorageLevel.DISK_ONLY)`

16

Coping with failures

- If a node that has data persisted on it fails?
  - Spark recomputes lost partitions of data when needed
- Also, replicate data on multiple nodes
  - To handle node failures without slowdowns

Persistence Levels for Spark

Level	Space Used	Wall clock time	In Memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory
MEMORY_AND_DISK _SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory
DISK_ONLY	Low	High	N	Y	

What if you attempt to cache too much data that does not fit in memory?

- Spark will **evict old partitions** using a Least Recently Used Cache policy
  - For memory only storage partitions, it will be recomputed the next time they are accessed
  - For memory_and_disk ones? Write them out to disk
- RDDs also come with a method, `unpersist()`
  - Manually remove data elements from the cache

RDDs of key/value pairs

- Key/value RDDs are commonly used to perform aggregations
  - Might have to do ETL (Extract, Transform, and Load) to get data into key/value formats
- Advanced feature to control layout of pair RDDs across nodes
  - **Partitioning**

RDDs containing key/value pairs

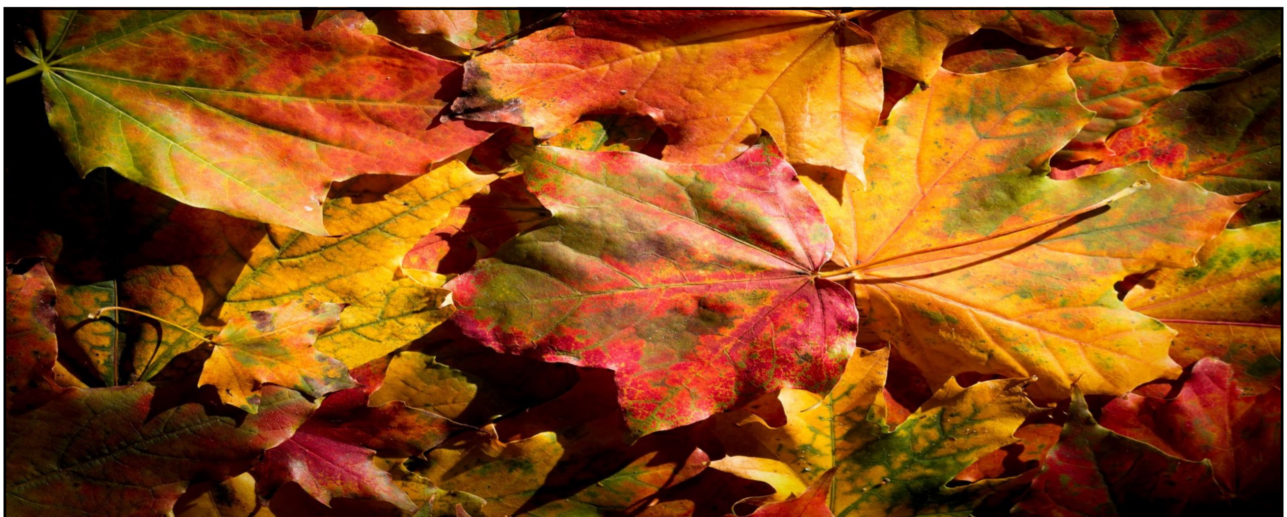
- Are called **pair RDDs**
- Useful *building block* in many programs
  - Expose operations that allow actions on each key in parallel or regroup data across network
  - `reduceByKey()` to aggregate data separately for each key
  - `join()` to merge two RDDs together by grouping elements of the same key

Creating Pair RDDs

- `pairs=lines.map(lambda x:  (x.split(" ") )[0], x))`
  - Creates a pairRDD using the first word as the key

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT SPARK L31.23

23



TRANSFORMATIONS ON PAIR RDDs

COMPUTER SCIENCE DEPARTMENT COLORADO STATE UNIVERSITY

24

Transformations on Pair RDDs [1/5]

- Pair RDD = {(1,2), (3,4), (3,6) }
- **`reduceByKey(func)`**
  - Combine values with the same key
  - Invocation: `rdd.reduceByKey((x, y) => x + y)`
  - Result: { (1, 2), (3,10) }

Transformations on Pair RDDs [2/5]

- Pair RDD = {(1,2), (3,4), (3,6) }
- **`groupByKey(func)`**
  - Group values with the same key
  - Invocation: `rdd.groupByKey()`
  - Result: { (1, [2]), (3, [4, 6]) }

Transformations on Pair RDDs [3/5]

- Pair RDD = {(1,2), (3,4), (3,6) }
- **`mapValues(func)`**
  - Apply function to each value of a pair RDD *without* changing the key
  - Invocation: `rdd.mapValues(x => x+1)`
  - Result: { (1, 3), (3, 5) , (3, 7) }

Transformations on Pair RDDs [4/5]

- Pair RDD = {(1,2), (3,4), (3,6) }
- **`values()`**
  - Return an RDD of just the values
  - Invocation: `rdd.values()`
  - Result: { 2, 4, 6 }

Transformations on Pair RDDs [5/5]

- Pair RDD = {(1,2), (3,4), (3,6) }
- **`sortByKey()`**
  - Return an RDD sorted by the key
  - Invocation: `rdd.sortByKey()`
  - Result: { (1,2), (3,4), (3,6 }

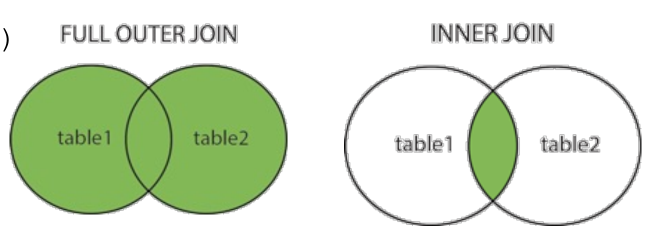
TRANSFORMATIONS ON TWO PAIR RDDS

Transformations on two Pair RDDs [1/5]

- **`rdd`** = {(1,2), (3,4), (3,6) }   **`other`** = {(3,9)}
- **`subtractByKey()`**
  - Remove elements with a key present in the `other` RDD
  - Invocation: `rdd.subtractByKey(other)`
  - Result: { (1,2) }

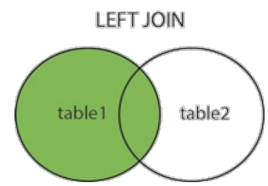
Transformations on two Pair RDDs [2/5]

- **`rdd`** = {(1,2), (3,4), (3,6) }   **`other`** = {(3,9)}
- **`join()`**
  - Perform an **inner join** between two RDDs. Only keys that are present in both pair RDDs are output
  - Invocation: `rdd.join(other)`
  - Result: { (3, (4,9)) , (3, (6,9)) }



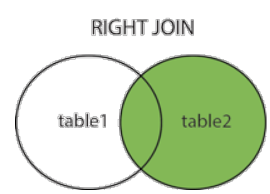
Transformations on two Pair RDDs [3/5]

- **rdd** = {(1,2), (3,4), (3,6) }   **other** = {(3,9)}
- **`leftOuterJoin()`**
  - Perform a join between two RDDs where the key must be present in the first RDD
  - Value associated with each key is a tuple of the value from the source and an Option for the value from the `other` pair RDD
    - In python if a value is not present, None is used.
  - Invocation: `rdd.leftOuterJoin(other)`
  - Result: { (1, (2,None)) , (3, (4, 9)) , (3, (6, 9)) }



Transformations on two Pair RDDs [4/5]

- **rdd** = {(1,2), (3,4), (3,6) }   **other** = {(3,9)}
- **`rightOuterJoin()`**
  - Perform a join between two RDDs where the key must be present in the `other` RDD;
  - Tuple has an option for the source rather than `other` RDD
  - Invocation: `rdd.rightOuterJoin(other)`
  - Result: { (3, (4,9)) , (3, (6,9)) }



Transformations on two Pair RDDs [5/5]

- **`rdd`** = {(1,2), (3,4), (3,6) }   **`other`** = {(3,9)}
- **`cogroup()`**
  - Group data from both RDDs using the same key
  - Invocation: `rdd.cogroup(other)`
  - Result: { (1, ([2],[])) , (3, ([4, 6], [9])) }

Example of chaining operations: Calculation of per-key average

key	value
panda	0
pink	3
pirate	3
panda	1
pink	4

mapValues

key	value
panda	(0, 1)
pink	(3, 1)
pirate	(3, 1)
panda	(1, 1)
pink	(4, 1)

reduceByKey

key	value
panda	(1, 2)
pink	(7, 2)
pirate	(3, 1)

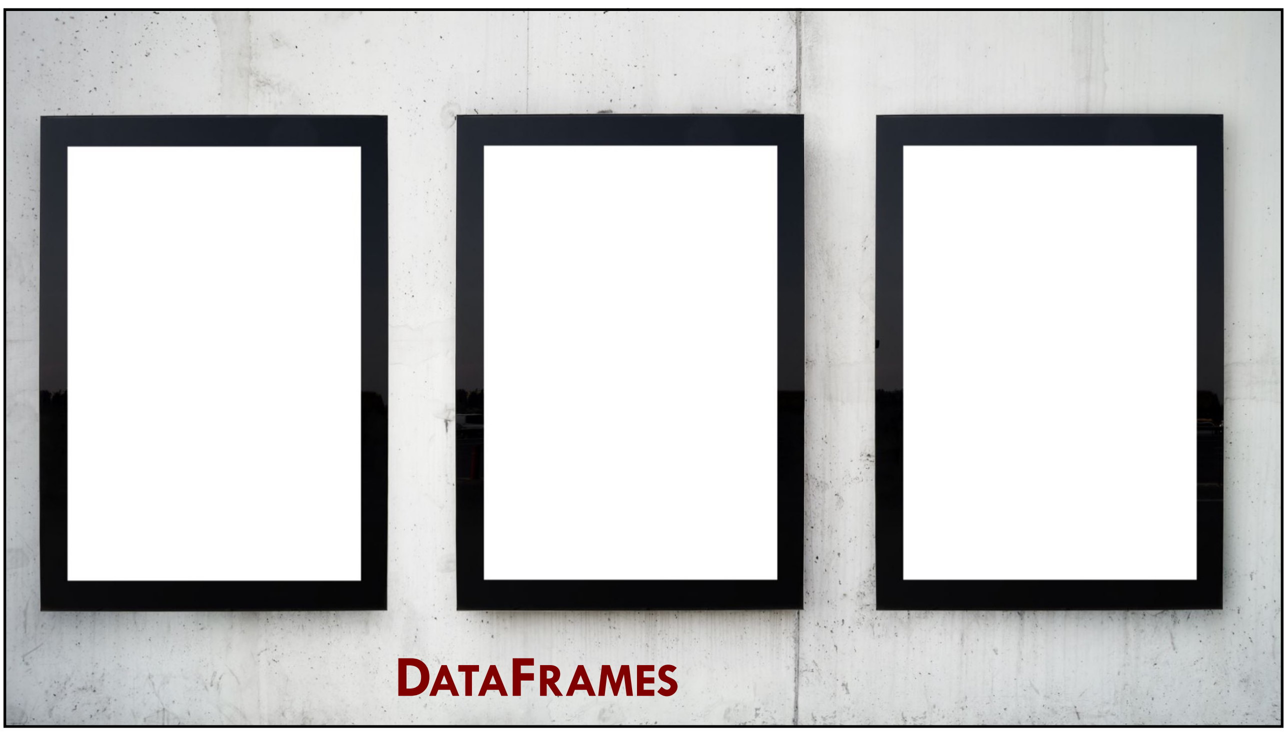
```
rdd.mapValues(x=> (x, 1)).reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
```

A word count example

- We are using `flatMap()` to produce a pair RDD of words and the number 1

```
 rdd   = sc.textfile("s3://…")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x,1)).
              reduceByKey(lambda x, y: x+y)
```


37



38

Spark DataFrame

- DataFrames consist of
  - A series of **records** (like rows in a table) that are of type `Row`
  - A number of columns (like columns in a spreadsheet)
- Rows
  - You can create rows by manually instantiating a `Row` object with the values that belong in each column
- Columns
  - You can select, manipulate, and remove columns from `DataFrames` and these operations are represented as **expressions**

Schemas

- A **schema** defines the column names and types of a `DataFrame`
- You can let a data source define the schema (called schema-on-read) or define it explicitly
- Note that only `DataFrames` have schemas
  - Rows themselves *do not* have schemas
  - If you create a `Row` manually?
    - You must specify the values *in the same order* as the schema of the `DataFrame` to which they might be appended

We can create `DataFrames` from raw data sources

- Spark has six **"core"** data sources
  - CSV
  - JSON
  - Parquet
  - ORC: Apache Optimized Row Columnar (ORC) file format
  - JDBC/ODBC connections
  - Plain-text files
- Hundreds of external data sources written by the community
  - E.g.: Cassandra, HBase, MongoDB, AWS, Redshift, XML etc.

The foundation for reading data in Spark is the `DataFrameReader`

- We access this through the `SparkSession` via the `read` attribute: `spark.read`
- After we have a `DataFrame` reader, we specify several values:
  - The format: Input data source format
  - The schema
  - The read mode {Permissive, DropMalformed, Failfast}
  - A series of options
- **The format, options, and schema each return a `DataFrameReader`** that can undergo *further* transformations and are all optional

However, at a minimum, the `DataFrameReader` must have a **`path`** from which to read

```
spark.read.format("csv")
.option("mode", "FAILFAST")
.option("inferSchema", "true")
.option("path", "path/to/file(s)")
.schema(someSchema)
.load()
```


Writing data is quite similar to that of reading data

- Instead of the `DataFrameReader` , we have the `DataFrameWriter`
- We access the `DataFrameWriter` on a per-`DataFrame` basis via the `write` attribute:

```
dataFrame.write
```

Writing Data

- After we have a `DataFrameWriter`, we specify three values:
  - The format, a series of options, and the save mode
- **At a minimum**, you must supply a `path`.
- Options may vary from data source to data source.

```
dataframe.write.format( "csv" )
                .option("mode", "APPEND")
                .option("dateFormat", "yyyy-MM-dd" )
                .option ("path", "path/to/file(s)" )
                .save ()
```


You can make any `DataFrame` into a table or view

- Done via a simple method call: `createOrReplaceTempView`
- This then allows you to query the data using SQL

```
val df = spark.read
              .format("json" )
              .load("/data/flight-data/json/2022-summary.json")

df.createOrReplaceTempView("dfTable")
```

`DataFrame` transformations

- Add rows or columns
- Remove rows or columns
- Transform a row into a column (or vice versa)
- Change the order of rows based on the values in columns

Adding Columns

- Use the **`withColumn`** method on the `DataFrame`
- For example, let's add a column that just adds the number one as a column:

```
df.withColumn("numberOne", lit(1))
```

Renaming Columns

- Done using the **`withColumnRenamed`** method.
- Will rename the column with the name of the string in the first argument to the string in the second argument:

```
df.withColumnRenamed ("DEST_COUNTRY_NAME","dest")
```


Removing Columns

- Done using a method called **`drop`**

```
df.drop("ORIGIN_COUNTRY_NAME" )
```

- We can drop multiple columns by passing in multiple columns as arguments

```
dfWithLongColName.drop("ORIGIN_COUNTRY_NAME",
                       "DEST_COUNTRY_NAME")
```

Filtering Rows

- To **filter** rows, we create an expression that evaluates to true or false
  - Those rows where the expression evaluates to `false` are filtered out

```
df.filter( col( "count" ) < 2)
```


Getting Unique Rows

- A very common use case is to extract the unique or **distinct** values in a `DataFrame`
  - These values can be in one or more columns
  - Done by using the **`distinct`** method on a `DataFrame`
    - Allows *deduplication* of any rows that are in that DataFrame.
  - Again, this is a transformation that will return a **new** DataFrame with only unique rows:

```
df.select("ORIGIN_COUNTRY_NAME","DEST_COUNTRY_NAME")
  .distinct()
```

Random Samples

- You might want to sample some random records from a `DataFrame`
- Done by using the **`sample`** method on a `DataFrame`
  - Specify a fraction of rows to extract from a `DataFrame` and whether the sample will be with or without replacement

```
val seed = 5
val withReplacement = false
val fraction = 0.5
df.sample(withReplacement, fraction, seed )
```


Random Splits

- Random splits are helpful when you need to break up a `DataFrame` into a random "splits" of the original `DataFrame`
- Often used with machine learning algorithms to create training, validation, and test sets

```
val dataFrames =
    df.randomSplit(Array (0.25, 0.75 ), seed )
```

Column Manipulations [1/4]

- **`withColumn(columnName, func)`**
  - Return an Dataframe with the additional column
  - Invocation: `df.withColumn("dogYears", df.age / 7)`
- **`dropColumn(columnName)`**
  - Return an Dataframe without the column
  - Invocation: `df.dropColumn("age")`

Column Manipulations [2/4]

- **`select(columnNames)`**
  - Return an `DataFrame` with the specified columns
  - Invocation: `df.select("firstName", "age")`
- **`describe(columnName)`**
  - Compute summary statistics over `DataFrame` columns
  - Invocation: `df.describe("age"), df.describe()`

Column Manipulations [3/4]

```
val df = Seq(
    (“Peterson”, “Marcus”, 54),
    (“Batey”, “Edward”, 36),
    (“Bruce”, "Karen", 35)
  ).toDF("lastName", “firstName”, "age”)

  df.withColumn(“dogYears”, df.age / 7.0)
  df.describe(“age”, “dogYears”)
```


Column Manipulations [4/4]

```
+-------+---------+---------+
|summary|      age| dogYears|
+-------+---------+---------+
|  count|        3|        3|
|   mean|  41.6667|  5.95238|
| stddev| 10.69268|  1.52753|
|    min|       35|        5|
|    max|       54| 7.714286|
+-------+---------+---------+
```

Dataframe joins

- **`join(other, <columnComparison>, <joinType>)`**
  - Performs a join between 2 Dataframes
  - Invocation: `df1.join(df2, Seq("id"))`

Join column comparison

- Supports a variety of criteria
  - Sequence of column names (e.g., `Seq("id", "age")`)
  - Elaborate comparison definitions (e.g., `df1("age") >= df2("age")`)

Join Type

- `DataFrames` may perform multiple styles of join
  - Inner: typical dataset join with key-to-key match
  - Outer, left-outer, right-outer: result contains all rows, filling in columns with 'null' values where data doesn't exist
  - Left-semi, right-semi: similar to outer join, but result only contains rows in specified source dataset

Example: Spark SQL

```
val df = Seq(
    ("Peterson", "Marcus", 54),
    ("Batey", "Edward", 36),
    ("Bruce", "Karen", 35)
  ).toDF("lastName", "firstName", "age")

  df.createOrReplaceTempView("people")
  spark.sql("SELECT firstName, age, age / 7.0 as dogYears
    FROM people where age < 50")
```

The contents of this slide-set are based on the following references

- *Learning Spark: Lightning-Fast Big Data Analysis. 1st Edition. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. O'Reilly. 2015. ISBN-13: 978-1449358624.* [Chapters 1-4, 10]
- Chambers, Bill, and Zaharia, Matei. Spark: The Definitive Guide: Big Data Processing Made Simple. O'Reilly Media. ISBN-13: 978-1491912218. 2018. [Chapters 5 and 9].
- SQL Joins: https://www.w3schools.com/sql/sql_join.asp
- Karau, Holden; Warren, Rachel. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media. 2017. ISBN-13: 978-1491943205. [Chapter 2]