CS X55: DISTRIBUTED SYSTEMS [SPARK]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Where are replicas of Spark partitions stored?
- Rdd1→Rdd2→Rdd3→Rdd4 if no action is invoked on any of these RDDs, where are they stored?
- Say Rdd1→Rdd2 … if an action is invoked on Rdd2; what happens to Rdd1?

Topics covered in this lecture

- Actions on RDDs
- Pair RDDs
- Data Frames

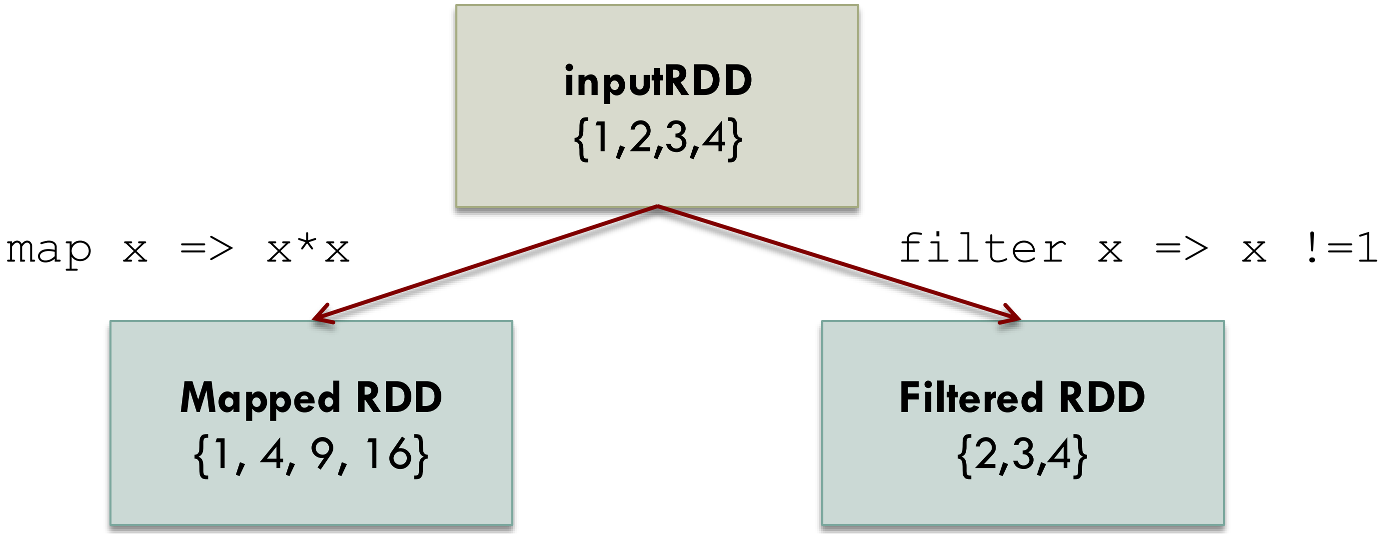
Common transformations and Actions

Element-wise transformations: `filter()`

- Takes in a function and returns an RDD that only has elements that pass the `filter()` function

Element-wise transformations: `map()`

- Takes in a function and applies it to each element in the RDD
- Result of the function is the new value of each element in the resulting RDD



Things that can be done with `map()`

- Fetch website associated with each URL in collection to just squaring numbers
- `map()`'s return type does not have to be the same as its input type
- Multiple output elements for each input element?
  - Use `flatMap()`

```
lines=sc.parallelize([“hello world”, “hi”])
words=lines.flatMap(lambda line: line.split(“ “) )
words.first()    # returns hello
```

Difference between `map` and `flatMap`

RDD1
{“coffee panda”, “happy panda”, “happiest panda party”}

RDD1.map(tokenize)

mappedRDD
{[“coffee”, “panda”], [“happy”, “panda”], [“happiest”, “panda”, “party”]}

RDD1.flatMap(tokenize)

flatMappedRDD
{“coffee”, “panda”, “happy”, “panda”, “happiest”, “panda”, “party”}

Psuedo set operations

- RDDs support many of the operations of mathematical sets such as union, intersection, etc.
  - Even when the RDDs themselves are not properly sets

Some simple set operations

RDD1
{coffee, coffee, panda, tiger, tea}

RDD2
{coffee, tiger, snake}

RDD1.distinct()
{coffee, tiger, panda, tea}

RDD1.union(RDD2)
{coffee, coffee, coffee, panda, tiger, tiger, tea, snake}

RDD1.intersection(RDD2)
{coffee, tiger}

RDD1.subtract(RDD2)
{panda, tea}

Cartesian product between two RDDs

RDD1
{User1, User2, User3}

`cartesian`

RDD2
{Venue(“Betabrand”),
Venue(“Asha Tree House”),
Venue(“Ritual”)}

RDD1.cartesian(RDD2)
{ (User1, Venue(“Betabrand”)),
(User1,Venue(“Asha Tree House”)),
(User1,Venue(“Ritual”)),
(User2, Venue(“Betabrand”)),
(User2,Venue(“Asha Tree House”)),
(User2,Venue(“Ritual”)),
(User3, Venue(“Betabrand”)),
(User3,Venue(“Asha Tree House”)),
(User3,Venue(“Ritual”)) **}**

Common Actions

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

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}

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)

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)`

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

PairRDDs: Working with Key/Value Pairs

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

Transformations on Pair RDDs

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

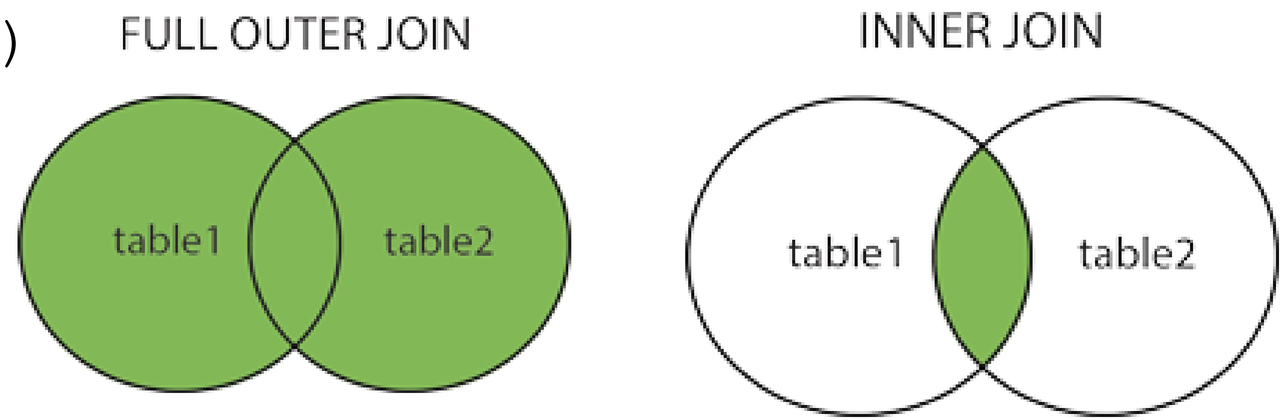
Transformations on two Pair RDDs [1/5]

- **`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) }

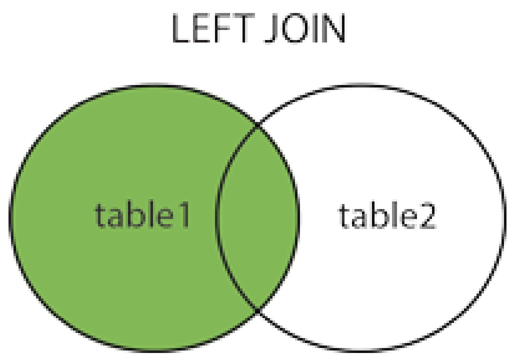
Transformations on two Pair RDDs [2/5]

- **`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]

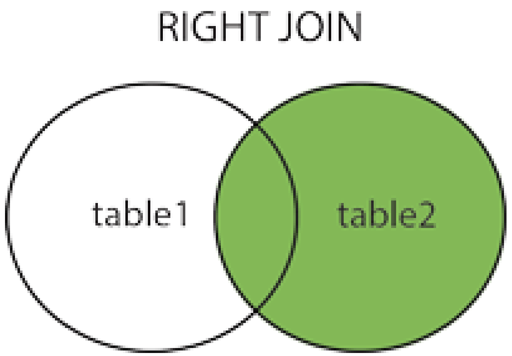
- **`rdd`** = {(1,2), (3,4), (3,6) }   **`other`** = {(3,9)}
- **`leftOuterJoin()`**
  - Perform a join between two RDDs where the key must be present in the first RDD
  - Value associated with each key is a tuple of the value from the source and an Option for the value from the `other` pair RDD
    - 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)
```

DataFrames

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 `DataFrame`s 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")
```

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]