CS 455: INTRODUCTION TO DISTRIBUTED SYSTEMS [SPARK]

Transformations: Narrow and Wide

Though their numbers are few
Don't let them beguile you
Innocuous though
they may seem
The wrong invocation
Is all it takes
To amplify inefficiencies
And protract computations

Shrideep Pallickara
Computer Science
Colorado State University

1

Topics covered in this lecture

- Pair RDDs
- Data Frames
- Dependencies and Transformations
- Partitioners

2

TRANSFORMATIONS ON PAIR RDDS

Pair RDDs

- RDDs that contain **key/value pairs**
- Expose partitions that allow you to act on each key in parallel or regroup data across the network

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 }

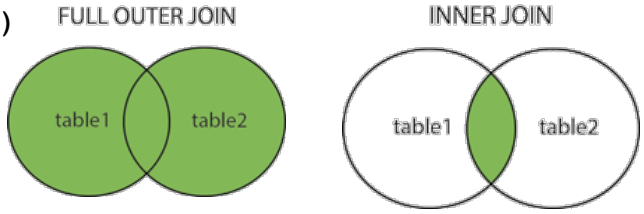
TRANSFORMATIONS ON TWO PAIR RDDS

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) }

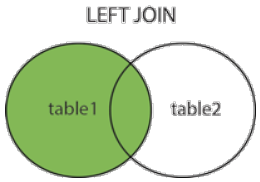
Transformations on two Pair RDDs [2/5]

- **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)) }



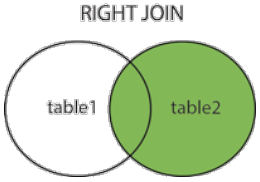
Transformations on two Pair RDDs [3/5]

- **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])) }

15

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))
```


16

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 `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/2015-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")
```

TUNING THE LEVEL OF PARALLELISM

Tuning the level of parallelism

- Every RDD has a **fixed number of partitions**
  - Determine the degree of parallelism when executing operations
- During aggregations or grouping operations, you can ask Spark to use a specific number of partitions
  - This will override defaults that Spark uses

Example: Tuning the level of parallelism

```
data = [("a", 3), ("b", 4), ("a", 1)]

sc.parallelize(data).
       reduceByKey(lambda x, y: x + y) #default

sc.parallelize(data).
       reduceByKey(lambda x, y: x + y, 10) #Custom
```


What if you want to tune parallelism outside of grouping and aggregation operations?

- There is `repartition()`
  - **Shuffles data across the network** to create a new set of partitions
  - Very **expensive operation!**
- There is the `coalesce()` operation
  - Allows avoiding data movement
    - But only if you are **decreasing** the number of partitions
  - Check `rdd.getNumPartitions()` and make sure you are coalescing to fewer partitions than current

WIDE AND NARROW TRANSFORMATIONS

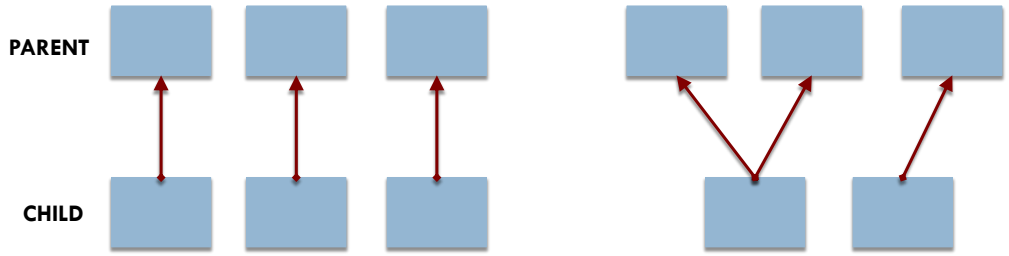
Transformations and Dependencies

- Two categories of **dependencies**
  - Narrow
    - Each partition of the parent RDD is used by **at most one partition of the child** RDD
  - Wide
    - Multiple child RDD partitions may depend on a single parent RDD partition
- The narrow versus wide distinction has significant implications for the way Spark evaluates a transformation and, consequently, for its performance

Narrow Transformations

- Narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD
- Can be **determined at design time**, irrespective of the values of the records in the parent partitions
- Partitions in narrow transformations can either depend on:
  - One parent (such as in the `map` operator), or
  - A unique subset of the parent partitions that is known at design time (`coalesce`)
- Narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions.

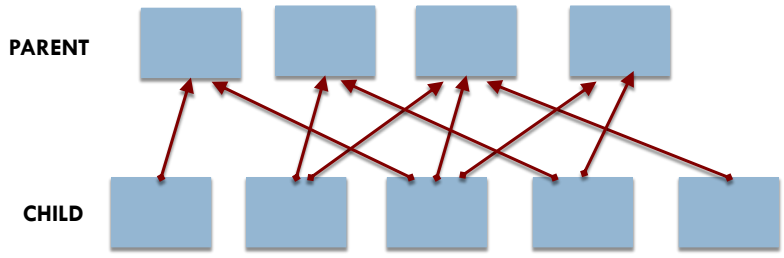
Dependencies between partitions for narrow transformations



Wide Transformations

- Transformations with **wide dependencies** cannot be executed on arbitrary rows
- Require the data to be partitioned in a particular way, e.g., according the **value** of their key
  - In sort, for example, records have to be partitioned so that keys in the same range are on the same partition
- Transformations with wide dependencies include `sort`, `reduceByKey`, `groupByKey`, `join`, and anything that calls the `rePartition` function

Dependencies between partitions for wide transformations



Wide dependencies **cannot be known fully before the data is evaluated**

The dependency graph for any operations that cause a **shuffle** (such as `groupByKey`, `reduceByKey`, `sort`, and `sortByKey`) follows this pattern

PAIR RDDS: WHAT TO WATCH FOR

53

Despite their utility, key/value operations can lead to a number of performance issues

- Most expensive operations in Spark fit into the key/value pair paradigm
  - Because **most wide transformations** are key/ value transformations,
    - And most require some fine tuning and care to be performant

54

In particular, operations on key/ value pairs can cause …

1. Out-of-memory errors in the driver
2. Out-of-memory errors on the executor nodes
3. Shuffle failures
4. "Straggler tasks" or partitions, which are especially slow to compute

- The last three performance issues are all most often caused by **shuffles associated with the wide transformations**

Memory errors in the driver, are usually caused by actions

- Several key/value actions (including `countByKey`, `countByValue`, `lookUp`, and `collectAsMap`) return data to the driver
- In most instances they return unbounded data since the number of keys and the number of values are unknown
- In addition to number of records, the size of each record is an important factor in causing memory errors

Preventing out-of-memory errors with aggregation operations [1/2]

- `combineByKey` and all of the aggregation operators built on top of it (`reduceByKey, foldLeft, foldRight, aggregateByKey`) may lead to memory errors if they cause the accumulator to become too large for one key
- What about `groupByKey`?
  - It is actually implemented using `combineByKey` where the accumulator is an iterator with all the data.

Preventing out-of-memory errors with aggregation operations [2/2]

- Use functions that implement **map-side combinations**
  - Meaning that records with the same key are combined before they are shuffled
  - This can greatly reduce the shuffled read
- The following four functions are implemented to use map-side combinations
  - `reduceByKey`
  - `treeAggregate`
  - `aggregateByKey`
  - `foldByKey`

Two primary techniques to avoid performance problems associated with shuffles

- Shuffle Less
- Shuffle Better

Shuffle Less

- Preserve partitioning across narrow transformations to avoid reshuffling data
- Use the same partitioner on a sequence of wide transformations. This can be particularly useful:
  - To avoid shuffles during joins and …
  - To reduce the number of shuffles required to compute a sequence of wide transformations

Shuffle Better [1/2]

- Sometimes, computation cannot be completed without a shuffle
- However, not all wide transformations and not all shuffles are equally expensive or prone to failure

61

Shuffle Better [2/2]

- By using wide transformations such as `reduceByKey` and `aggregateByKey` that can preform map-side reductions and that do not require loading all the records for one key into memory?
  - You can prevent memory errors on the executors and
  - Speed up wide transformations, particularly for aggregation operations
- Lastly, shuffling data in which **records are distributed evenly throughout the keys**, and which contain a **high number of distinct keys**?
  - Prevents out-of-memory errors on the executors and "straggler tasks"

62

PARTITIONERS

63

Partitioners

- The partitioner defines **how records will be distributed** and thus which records will be completed by each task
- Practically, a partitioner is actually an interface with two methods
  - `numPartitions` that defines the number of partitions in the RDD after partitioning
  - `getPartition` that defines a mapping from a key to the integer index of the partition where records with that key should be sent.

64

There are two implementations for the partitioner object provided by Spark

- HashPartitioner
  - Determines the index of the child partition based on the hash value of the key
- RangePartitioner
  - Assigns records whose keys are in the same range to a given partition
  - Required for **sorting** since it ensures that by sorting records within a given partition, the entire RDD will be sorted
- It is possible to define a custom partitioner

Partitioners and transformations

- Unless a transformation is known to only change the value part of the key/value pair in Spark
  - The resulting RDD will **not have a known** partitioner
    - Even if the partitioning has not changed

Using narrow transformations that preserve partitioning

- Some narrow transformations, such as `mapValues`, **preserve the partitioning** of an RDD if it exists
- Common transformations like `map` and `flatMap` can change the key
  - So even if your function does not change the key, the resulting RDD will not have a known partitioner.
  - Instead, if you don't want to modify the keys, call the `mapValues` function (defined only on pair RDDs)
    - It keeps the keys, and therefore the partitioner, exactly the same.
    - The `mapPartitions` function will also preserve the partition if the `preservesPartitioning` flag is set to true.

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