BIG DATA ANALYTICS

Spark

Resilient Distributed Datasets (RDDs)

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs)

- Spark provides Structured APIs (like DataFrames) for most use cases. However, when advanced control over data is needed, we use lower-level APIs.
- The key lower-level APIs in Spark include:
  1. RDDs: For manipulating distributed data.
  2. Shared Variables: Includes accumulators and broadcast variables for custom data sharing.

When to Use Low-Level APIs?

- When we need precise control over data distribution across the cluster.
- For maintaining legacy code that uses RDDs.
- For custom shared variable handling.

Resilient Distributed Datasets (RDDs)

Why Understand These APIs?

- All operations in Spark, even on DataFrames, ultimately translate to RDD transformations.
- Knowing these helps in debugging complex workloads.

Accessing Low-Level APIs

- We can access RDDs and shared variables via SparkContext, available through:

```
// python
spark.sparkContext
```

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): About RDDs

About RDDs

- RDDs were the primary API in Spark 1.X and are still available in 2.X, though they are now less commonly used.
- All Spark code, including DataFrames and Datasets, eventually compiles down to RDDs.
- RDDs represent an immutable, partitioned collection of records that can be processed in parallel.

RDDs vs. DataFrames

- RDDs allow storing any Java, Scala, or Python objects, offering flexibility but requiring manual optimization and manipulation.
- DataFrames provide automatic optimizations, such as compressed storage and efficient execution plans.

Resilient Distributed Datasets (RDDs): About RDDs

Types of RDDs

- Two common types of RDDs:
    1. Generic RDDs: A collection of objects.
    2. Key-Value RDDs: Allows custom partitioning and aggregation by key.

Properties of RDDs

- Five key properties define an RDD:
    1. List of partitions.
    2. Function to compute each partition.
    3. Dependencies on other RDDs.
    4. Optional Partitioner for key-value RDDs.
    5. Optional list of preferred locations for computing splits.

Resilient Distributed Datasets (RDDs): About RDDs

When to Use RDDs?

- Use RDDs when you need fine-grained control over data distribution or custom partitioning.
- RDDs lack many optimizations present in the Structured APIs, so prefer DataFrames for most tasks.

Performance Considerations

- Scala and Java RDDs perform similarly, but Python RDDs can be slower due to serialization overhead.
- It's recommended to use the Structured APIs in Python unless RDDs are absolutely necessary.

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Creating RDDs

1. Interoperating Between DataFrames, Datasets, and RDDs

- Convert DataFrames or Datasets to RDDs using the `rdd` method:

```
// Scala: Convert Dataset[Long] to RDD[Long]
spark.range(500).rdd

# Python: Convert DataFrame to RDD[Row]
spark.range(10).rdd
```

Resilient Distributed Datasets (RDDs): Creating RDDs

1. Interoperating Between DataFrames, Datasets, and RDDs

- Extract values from RDD[Row] for manipulation:

```
// Scala

spark.range(10).toDF().rdd.map(rowObject =>
rowObject.getLong(0))


# Python

spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

Resilient Distributed Datasets (RDDs): Creating RDDs

1. Interoperating Between DataFrames, Datasets, and RDDs

- Convert an RDD back to a DataFrame:

```
// Scala
spark.range(10).rdd.toDF()


# Python
spark.range(10).rdd.toDF()
```

Resilient Distributed Datasets (RDDs): Creating RDDs

1. Interoperating Between DataFrames, Datasets, and RDDs

- Extract values from RDD[Row] for manipulation:

```
// Scala

spark.range(10).toDF().rdd.map(rowObject =>
rowObject.getLong(0))


# Python

spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

Resilient Distributed Datasets (RDDs): Creating RDDs

2. From a Local Collection

- Create an RDD from a local collection using `parallelize`:

```
// Scala

val myCollection = "Spark The Definitive Guide : Big Data
Processing Made Simple".split(" ")

val words = spark.sparkContext.parallelize(myCollection,
2)

# Python

myCollection = "Spark The Definitive Guide : Big Data
Processing Made Simple".split(" ")

words = spark.sparkContext.parallelize(myCollection, 2)
```

Resilient Distributed Datasets (RDDs): Creating RDDs

2. From a Local Collection

- Name the RDD to track it in the Spark UI:

```scala
// Scala
  words.setName("myWords")
  words.name // myWords
```

```python
# Python
  words.setName("myWords")
  words.name() # myWords
```

Resilient Distributed Datasets (RDDs): Creating RDDs

3. From Data Sources

- Create an RDD from text files using `textFile`:

```
// scala
spark.sparkContext.textFile("/some/path/withTextFiles")
```

- Alternatively, read each file as a single record with `wholeTextFiles`:

```
// scala
spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")
```

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Manipulating RDDs

- Similar to DataFrames, but with raw Java/Scala objects instead of structured types.
- Fewer built-in functions-requires defining custom functions for filters, maps, aggregations, etc.
- Example: Using the previously created `words` RDD for further manipulations.

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Transformations

- Most transformations in RDDs work similarly to those in DataFrames and Datasets.
- We apply transformations to one RDD to create a new one, defining dependencies between them and manipulating the data.

distinct

- A `distinct` method call on an RDD removes duplicates from the RDD:

```
words.distinct().count()
```

This gives a result of 10.

Resilient Distributed Datasets (RDDs): Transformations

filter

- Filtering in RDDs is like a SQL `` `WHERE` `` clause, allowing us to keep records that match a certain condition (predicate function). The function returns a Boolean to decide whether to keep each row.
- Example: Filtering words that start with the letter "S"

```scala
// scala
def startsWithS(individual: String) = {
  individual.startsWith("S")
}
words.filter(word => startsWithS(word)).collect()
```

Resilient Distributed Datasets (RDDs): Transformations

filter

- Example: Filtering words that start with the letter "S"

```
// python
def startsWithS(individual):
  return individual.startswith("S")
words.filter(lambda word: startsWithS(word)).collect()
```

- This returns: `*Spark*` and `*Simple*`. The result is native types, with no need to convert the data.

Resilient Distributed Datasets (RDDs): Transformations

map

- Mapping applies a function to each record in the RDD, returning the desired values.
- In this example, we map each word to a tuple containing the word, its first letter, and whether it starts with "S."

```
// scala
val words2 = words.map(word => (word, word(0),
word.startsWith("S")))


// python
words2 = words.map(lambda word: (word, word[0],
word.startswith("S")))
```

Resilient Distributed Datasets (RDDs): Transformations

map

- We can then filter based on whether the word starts with "S":

```
// scala
words2.filter(record => record._3).take(5)


// python
words2.filter(lambda record: record[2]).take(5)
```

This returns tuples like `("Spark", "S", true)` and `("Simple", "S", true)`.

Resilient Distributed Datasets (RDDs): Transformations

map

flatMap

- `flatMap` is used when each input row should return multiple rows.
- For example, to break words into characters, we use `flatMap`:

```
// scala
words.flatMap(word => word.toSeq).take(5)


// python
words.flatMap(lambda word: list(word)).take(5)
```

This returns: `` `S, P, A, R, K` ``.

Resilient Distributed Datasets (RDDs): Transformations

sort

- To sort an RDD we must use the `sortBy` method, and just like any other RDD operation, we do this by specifying a function to extract a value from the objects in our RDDs and then sort based on that.
- For instance, the following example sorts by word length from longest to shortest:

```
// in Scala
words.sortBy(word => word.length() * -1).take(2)


# in Python
words.sortBy(lambda word: len(word) * -1).take(2)
```

Random Splits

- We can also randomly split an RDD into an `Array` of RDDs by using the `randomSplit` method, which accepts an `Array` of weights and a random seed:.

```
// in Scala
val fiftyFiftySplit = words.randomSplit(Array[Double](0.5, 0.5))

# in Python
fiftyFiftySplit = words.randomSplit([0.5, 0.5])
```

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Actions

- Just as we do with DataFrames and Datasets, we specify actions to kick off our specified transformations.
- Actions either collect data to the driver or write to an external data source.
  - reduce
  - count
    - countApprox
    - countApproxDistinct
    - countByValue
    - countByValueApprox
  - first
  - max and min
  - take

Resilient Distributed Datasets (RDDs): Actions

reduce

- The `reduce` method aggregates an RDD into a single value.
- For example, we can reduce a set of numbers to their sum by defining a function that combines two values.

```
// scala
spark.sparkContext.parallelize(1 to 20).reduce(_ + _) // 210
// python
spark.sparkContext.parallelize(range(1, 21)).reduce(lambda
x, y: x + y) # 210
```

Resilient Distributed Datasets (RDDs): Actions

reduce

- To find the longest word in an RDD, define a function that compares word lengths:

```
// scala
def wordLengthReducer(leftWord:String, rightWord:String):
String = {
  if (leftWord.length > rightWord.length)
    return leftWord
  else
    return rightWord
}
words.reduce(wordLengthReducer)
```

Resilient Distributed Datasets (RDDs): Actions

reduce

- To find the longest word in an RDD, define a function that compares word lengths:

```
# python
def wordLengthReducer(leftWord, rightWord):
 if len(leftWord) > len(rightWord):
   return leftWord
 else:
   return rightWord
words.reduce(wordLengthReducer)
```

This function can return either "definitive" or "processing" (both length 10), depending on how the data is processed.

Resilient Distributed Datasets (RDDs): Actions

count

- This method is fairly self-explanatory.
- Using it, we could, for example, count the number of rows in the RDD:

```
words.count()
```

Resilient Distributed Datasets (RDDs): Actions

count

1. countApprox

- An approximate version of the `count` method that runs within a set time limit.
- May return incomplete results if it exceeds the timeout.
- Confidence level (between 0 and 1) indicates the likelihood that the result is close to the true count.
  - Example: With a confidence of 0.9, 90% of the results should be accurate.
- Example usage:

```scala
// scala
  val confidence = 0.95
  val timeoutMilliseconds = 400
  words.countApprox(timeoutMilliseconds, confidence)
```

Resilient Distributed Datasets (RDDs): Actions

count

2. countApproxDistinct

- Estimates distinct counts using two methods based on the "HyperLogLog in Practice: Algorithmic Engineering of a State-of-the-Art Cardinality Estimation Algorithm."

2.1. Basic Implementation

- Pass in a relative accuracy value (smaller values use more memory).
- Example:

```
// scala
words.countApproxDistinct(0.05)
```

2.2. Advanced Implementation

- Specify precision (`p`) and sparse precision (`sp`) for more control.
- Helps reduce memory use and improve accuracy for small datasets.
- Example:

```
// scala
words.countApproxDistinct(4, 10)
```

Resilient Distributed Datasets (RDDs): Actions

count

3. countByValue

- Counts occurrences of each value in an RDD.
- Loads the result into the driver's memory, so it's best for small result sets.
- Use when the dataset is either small or has few distinct items.
- Example usage:

```
// scala
words.countByValue()
```

Resilient Distributed Datasets (RDDs): Actions

count

4. countByValueApprox

- Similar to `countByValue`, but returns an approximate result within a given timeout.
- May return incomplete results if it exceeds the time limit.
- Confidence level (between 0 and 1) represents the likelihood that the result is close to the true count.
    - Example: With confidence 0.9, 90% of the results should be accurate.
- Example usage:

```
// scala
words.countByValueApprox(1000, 0.95)
```

Resilient Distributed Datasets (RDDs): Actions

first

- The `first` method returns the first value in the dataset:

```
words.first()
```

max and min

- `max` and `min` return the maximum values and minimum values, respectively:

```
spark.sparkContext.parallelize(1 to 20).max()
spark.sparkContext.parallelize(1 to 20).min()
```

Resilient Distributed Datasets (RDDs): Actions

take

- Retrieves a specified number of values from an RDD.
- Scans one partition at a time and estimates how many more are needed to meet the limit.

Variations:

1. takeOrdered – Returns the smallest values in order.
2. top – Returns the largest values in order.
3. takeSample – Returns a random sample from the RDD.
   - You can specify whether to sample with replacement, how many values to take, and a random seed.

Resilient Distributed Datasets (RDDs): Actions

take

Variations: cont'd.

- Examples:

```
// scala
words.take(5)
words.takeOrdered(5)
words.top(5)
val withReplacement = true
val numberToTake = 6
val randomSeed = 100L
words.takeSample(withReplacement, numberToTake,
randomSeed)
```

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Saving Files

Saving Files in Spark

1. saveAsTextFile

- Saves RDD data to a plain-text file.
- Specify the file path, and optionally, a compression codec.
- Example:

```
// scala
words.saveAsTextFile("file:/tmp/bookTitle")
import org.apache.hadoop.io.compress.BZip2Codec
words.saveAsTextFile("file:/tmp/bookTitleCompressed",
classOf[BZip2Codec])
```

Resilient Distributed Datasets (RDDs): Saving Files

Saving Files in Spark

2. SequenceFiles

- Stores data as binary key-value pairs, often used in Hadoop's MapReduce.
- Example:

```
//scala
words.saveAsObjectFile("/tmp/my/sequenceFilePath")
```

3. Hadoop Files

- Supports saving to various Hadoop file formats with customizable options like output formats, compression, and configurations.
- Useful for legacy Hadoop jobs.

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching 
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Caching

Caching in Spark

- We can cache or persist RDDs to store data in memory for faster access.
- By default, caching stores data in memory.
- Use `setName` to give the cached RDD a name.

```
// scala
words.cache()
```

- We can choose a storage level (memory, disk, or off-heap) using `StorageLevel`.
- Example of checking the storage level:

```
// in Scala
words.getStorageLevel

# in Python
words.getStorageLevel()
```

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Check pointing

Checkpointing in Spark

- Saves an RDD to disk, so future computations use the saved version instead of recomputing from the source.
- Similar to caching, but stores data on disk, not in memory.
- Useful for iterative computations.

```
// in Scala
spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")
words.checkpoint()
```

- Future references to this RDD will come from the checkpoint, improving performance.

Spark Resilient Distributed Datasets (RDDs): *Outline*

- Introduction to RDDs
- Creating RDDs
- Manipulating RDDs
- Transformations
- Actions
- Saving Files
- Caching
- Check pointing
- Pipe RDDs to System Commands

Resilient Distributed Datasets (RDDs): Pipe RDDs to System Commands

- The `pipe` method allows us to pipe elements of an RDD to an external process for each partition.
- Each input partition is sent to the external process's `stdin` as lines, and the `stdout` output forms the new partition.
- Example:

```
// python
words.pipe("wc -l").collect() # Example: returns line count per partition
```

Resilient Distributed Datasets (RDDs): Pipe RDDs to System Commands

1. mapPartitions

- `mapPartitions` allows us to apply a function to each partition (not row-wise, but partition-wise).
- Useful for operations on subdatasets, like custom machine learning algorithms.
- Example:

```
// Scala
words.mapPartitions(part => Iterator[Int](1)).sum()  // 2


# Python
words.mapPartitions(lambda part: [1]).sum()  # 2
```

Resilient Distributed Datasets (RDDs): Pipe RDDs to System Commands

1. mapPartitions cont'd.

mapPartitionsWithIndex

- This function works like `mapPartitions`, but also provides the partition index, useful for debugging.
- Example:

```
// Scala
def indexedFunc(partitionIndex:Int, withinPartIterator: Iterator[String]) =
{
      withinPartIterator.toList.map(
          value => s"Partition: $partitionIndex => $value").iterator
}
words.mapPartitionsWithIndex(indexedFunc).collect()

# Python
def indexedFunc(partitionIndex, withinPartIterator):
      return ["partition: {} => {}".format(partitionIndex, x) for x in
withinPartIterator]
words.mapPartitionsWithIndex(indexedFunc).collect()
```

Resilient Distributed Datasets (RDDs): Pipe RDDs to System Commands

2. foreachPartition

- Similar to `mapPartitions`, but without needing a return value. Commonly used for operations like writing partition data to a database.
- Example:

```
// scala
words.foreachPartition { iter =>
  import java.io._
  import scala.util.Random
  val randomFileName = new Random().nextInt()
  val pw = new PrintWriter(new File(s"/tmp/random-file-
${randomFileName}.txt"))
  while (iter.hasNext) {
      pw.write(iter.next())
  }
   pw.close()
}
```

Resilient Distributed Datasets (RDDs): Pipe RDDs to System Commands

3. glom

- `glom` converts each partition into an array. Useful when collecting data to the driver, but may cause issues if partitions are large.
- The term `glom` doesn't have a formal abbreviation in Spark, but it is often interpreted as an informal word meaning "to gather" or "to group." In the context of Spark, the `glom` function groups all elements within a partition into a list or array.
- Example:

```
// Scala
spark.sparkContext.parallelize(Seq("Hello", "World"), 2).glom().collect()
// Array(Array(Hello), Array(World))

# Python
spark.sparkContext.parallelize(["Hello", "World"], 2).glom().collect()
# [['Hello'], ['World']]
```