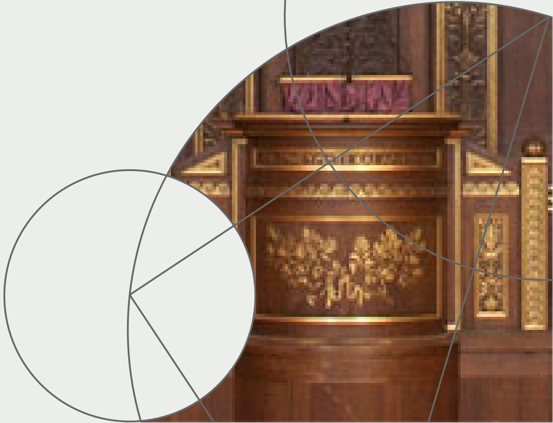
STEVENS INSTITUTE OF TECHNOLOGY

Data Acquisition and Processing I: Big Data @ Spark

Rensheng Wang,
rwang1@stevens.edu
Dept. of Electrical and Computer Engineering
Stevens Institute of Technology

November 7, 2019



Apache Spark data representations: RDD / Dataframe / Dataset

- Spark has three data representations viz RDD, Dataframe, Dataset. For each data representation, Spark has a different API.
- For example, later in this article I am going to use **ml** (a library), which currently supports only Dataframe API. Dataframe is much faster than RDD because it has metadata (some information about data) associated with it, which allows Spark to optimize query plan. The Dataframe feature in Apache Spark was added in Spark 1.3.

  Here we will not talk about Dataset as this functionality is not included in PySpark.

Introducing Spark

- At a high level, every Spark application consists of a driver program that launches various parallel operations on a cluster.
- The driver program contains your applicationâĂŹs main function and defines distributed datasets on the cluster, then applies operations to them.
- Driver programs access Spark through a `SparkContext` object, which represents a connection to a computing cluster. In the shell, a SparkContext is automatically created for you as the variable called `sc`.
- Once you have a SparkContext, you can use it to build RDDs (resilient distributed dataset).
- For example, we call `sc.textFile()` to create an RDD representing the lines of text in a file.

Understanding RDD

- RDDs (resilient distributed dataset) are the core concept in Spark.
- RDD is simply a distributed collection of elements. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.
- Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.
- An RDD in Spark is simply an immutable distributed collection of objects. Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program.

SparkContext

- To run these operations, driver programs typically manage a number of nodes called executors.ï£ij

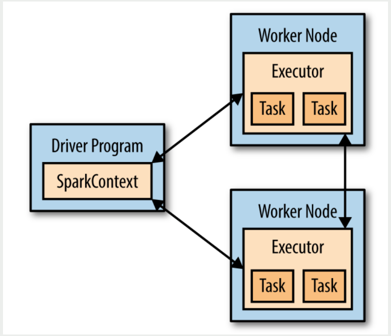


Figure: Components for distributed execution in Spark

- In summary, a quick overview of the core concepts involved in programming with Spark: a driver program creates a SparkContext and RDDs, and then runs parallel operations on them.

Creating RDD

Spark provides two ways to create RDDs: loading an external dataset and parallelizing a collection in your driver program.

- Creating an RDD of strings with `textFile()` in Python

  ```
  »> lines = sc.textFile("myData.txt")
  ```

- The simplest way to create RDDs is to take an existing collection in your program and pass it to SparkContextâĂŹs `parallelize()` method

  ```
  »> lines = sc.parallelize(["pandas", "i like pandas"])
  »> lines = sc.parallelize([1, 2,  3,  4])
  ```

Programming on RDD

- Once created, RDDs offer two types of operations: transformations and actions. Transformations construct a new RDD from a previous one.

  ```
  >> lines = sc.textFile("myDataFile.txt")
  ```

- For example, one common transformation is filtering data that matches a condition.

  ```
  >> pythonLines = lines.filter(lambda line:  "Python" in line)
  ```

- Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).
- One example of an action we called is `first()`, which returns the first element in an RDD and is demonstrated below.

  ```
  >> pythonLines.first()
  ```

Programming on RDD

- Transformations and actions are different because of the way Spark computes RDDs.
- Although you can define new RDDs any time, Spark computes them only in a **lazy fashion – that is, the first time they are used in an action**. This approach might seem unusual at first, but makes a lot of sense when you are working with Big Data.
- For instance, we read a text file and then filter the lines that include Python. If Spark were to load and store all the lines in the file as soon as we wrote **lines = sc.textFile(...)**, it would waste a lot of storage space, given that we then immediately filter out many lines.
- Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result. In fact, for the **first()** action, Spark scans the file only until it finds the first matching line; it doesnâĂŹt even read the whole file.
- Finally, SparkâĂŹs RDDs are by default recomputed each time you run an action on them. If you would like to reuse an RDD in multiple actions, you can ask Spark to persist it using **RDD.persist()**.
- After computing it the first time, Spark will store the RDD contents in memory (partitioned across the machines in your cluster), and reuse them in future actions.

Programming on RDD

- To summarize, every Spark program and shell session will work as follows:
  1. Create some input RDDs from external data.
  2. Transform them to define new RDDs using transformations like **filter()**.
  3. Ask Spark to **persist()** any intermediate RDDs that will need to be reused.
  4. Launch actions such as **count()** and **first()** to kick off a parallel computation, which is then optimized and executed by Spark.

Lazy Evaluation

- As you read earlier, transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.
- Lazy evaluation means that when we call a transformation on an RDD (for instance, calling **map()**), the operation is not immediately performed.
- Instead, Spark internally records metadata to indicate that this operation has been requested. Rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations.
- Loading data into an RDD is lazily evaluated in the same way transformations are. So, when we call **sc.textFile()**, the data is not loaded until it is necessary.
- Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together.

Common Transformations and Actions

Element-wise transformations

- The two most common transformations you will likely be using are `map()` and `filter()`

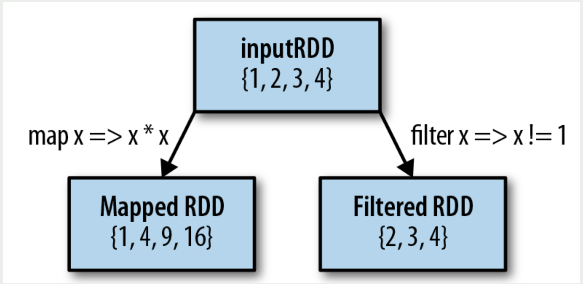


Figure: Mapped and filtered RDD from an input RDD

Common Transformations and Actions

Element-wise transformations

- For example, squaring the values in an RDD via **map()**

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x:  x * x).collect()
for num in squared:
      print "%i " % (num)
```

RDDs have a `collect()` function to retrieve the entire RDD. This is an action which is necessary before we output the results.

It can be useful if your program filters RDDs down to a very small size and youâĂŹd like to deal with it locally.

Keep in mind that your entire dataset must fit in memory on a single machine to use `collect()` on it, so `collect()` shouldnâĂŹt be used on large datasets.

Common Transformations and Actions

Element-wise transformations

- Sometimes we want to produce multiple output elements for each input element. The operation to do this is called **flatMap()**.

  As with **map()**, the function we provide to `flatMap()` is called individually for each element in our input RDD.

  Instead of returning a single element, we return an iterator with our return values. Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators.

  For example, `flatMap()` in Python, splitting lines into words

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

flatMap() vs. map()

- We illustrate the difference between `flatMap()` and `map()` in Figure below.
- You can think of `flatMap()` as âĂIJflatteningâĂİ the iterators returned to it, so that instead of ending up with an RDD of lists we have an RDD of the elements in those lists.

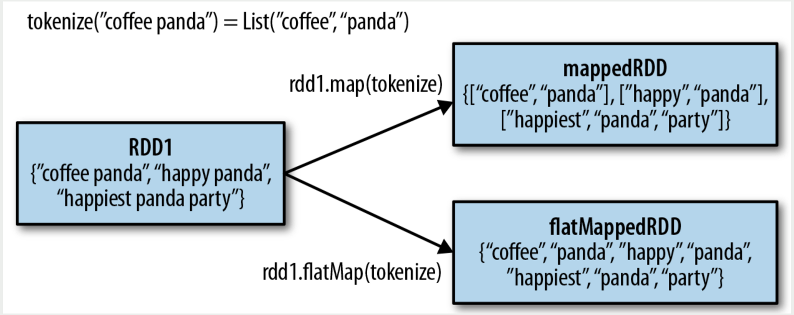

tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
rdd1.flatMap(tokenize)
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

Figure: Difference between flatMap() and map() on RDD

Actions

- The most common action on basic RDDs you will likely use is `reduce()`, which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type.
- A simple example of such a function is +, which we can use to sum our RDD. With `reduce()`, we can easily sum the elements of our RDD, count the number of elements, and perform other types of aggregations.
- Basic actions on an RDD containing $\{1,2,3,3\}$:

```
»> rdd.reduce((x, y) => x + y)   #   9 is the result
```

Working with Key/Value Pairs

- Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs.
- Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network.
- For example, pair RDDs have a **reduceByKey()** method that can aggregate data separately for each key, and a **join()** method that can merge two RDDs together by grouping elements with the same key.
- It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations.

Programming on Pair RDDs

Creating Pair RDDs

- We can do this by running a map() function that returns key/value pairs. To illustrate, we show code that starts with an RDD of lines of text and keys the data by the first word in each line.

  »> **pairs = lines.map(lambda x: (x.split(" ")[0], x))**

Transformations on Pair RDDs

- For example, transformations on one pair RDD (example: (1, 2), (3, 4), (3, 6))

  »> **rdd.reduceByKey((x,y)=> x+y)**

  The output result is `{(1, 2), (3, 10)}`

  >>> **rdd.mapValues(x => x+1)**

  `rdd.mapValues( )` apply a function to each value of a pair RDD without changing the key.

  The output result is `{(1,3), (3,5), (3,7)}`

  »> **rdd.values()**

  The output result is `{2, 4, 6}`

Aggregations on Pairs RDDs

- When datasets are described in terms of key/value pairs, it is common to want to aggregate statistics across all elements with the same key.
- For example, word count in a text file

```
rdd = sc.textFile("myData.txt")

words = rdd.flatMap(lambda x:  x.split(" "))

result = words.map(lambda x:  (x, 1)).reduceByKey(lambda x, y:  x + y)
```

Aggregations on Pairs RDDs

- We can use `reduceByKey()` along with `mapValues()` to compute the per-key average.

```
>>> rdd.mapValues(lambda x:  (x, 1)).reduceByKey(lambda x, y:  (x[0] +
y[0], x[1] + y[1]))
```

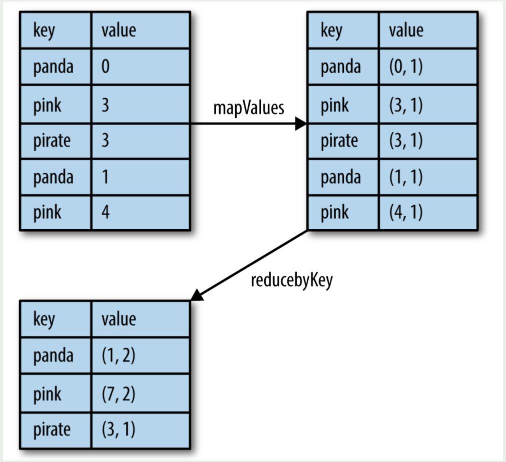


Figure: Per-key average data flow

RDD Data Transformation

- `count()`

  Return the number of elements in this RDD.

  ```
  sc.parallelize([2, 3, 4]).count()
  out[0]:  3
  ```

- `countByKey()`

  Count the number of elements for each key, and return the result to the master as a dictionary.

  ```
  rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
  sorted(rdd.countByKey().items()).
  out[0]:  [('a', 2), ('b', 1)]
  ```

RDD Data Transformation

- `distinct()`

  Return a new RDD containing the distinct elements in this RDD.

  ```
  sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())
  out[0]:  [1, 2, 3]
  ```

- `map()`

  transformation applies changes on each line of the RDD and returns the transformed RDD as iterable of iterables i.e. each line is equivalent to a iterable and the entire RDD is itself a list.

  ```
  rdd = sc.parallelize(["b", "a", "c"])
  rdd.map(lambda x:  (x, 1)).collect()
  out[0]:[('a', 1), ('b', 1), ('c', 1)]
  ```

RDD Data Transformation

- `filter(f)`

  Return a new RDD containing only the elements that satisfy a predicate.

  ```
  rdd = sc.parallelize([1, 2, 3, 4, 5])
  rdd.filter(lambda x:  x % 2 == 0).collect()
  out[0]:  [2, 4]
  ```

- `first()`

  Return the first element in this RDD.

  ```
  rdd = sc.parallelize([2, 3, 4])
  rdd.first()
  out[0]:2
  ```

Dataframe in Pyspark

- A DataFrame in Apache Spark can be created in multiple ways

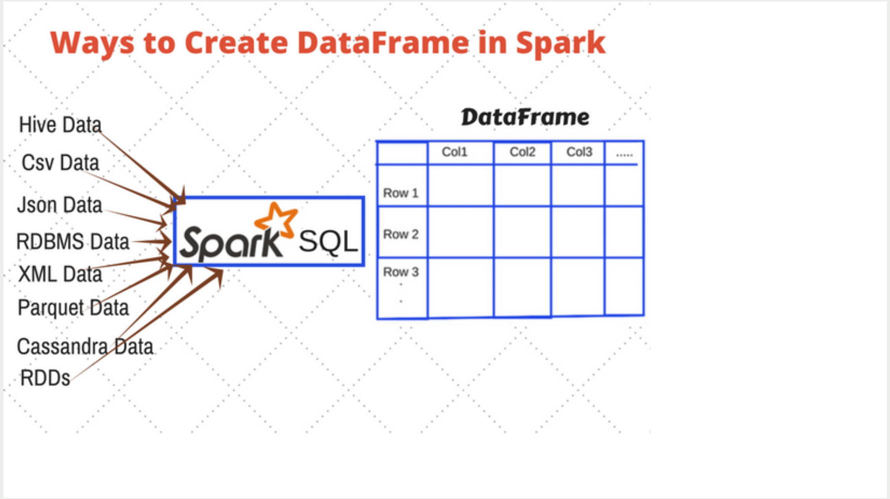


Figure: Per-key average data flow

Solving a machine learning problem

- Reading a data file (csv)

  In Apache Spark, we can read the csv file and create a Dataframe with the help of SQLContext. Dataframe is a distributed collection of observations (rows) with column name, just like a table.

  ```
  sc = sparkContext()
  sqlContext = SQLContext(sc)
  ```