Spark 2.0.2

Extending Spark RDD(Resilient Data Distribution) Rich APIs

Author : Pooshan Singh

Organization : Talentica Software Pvt. Ltd.

Department : (R & D) + (Talasecurity team).

Version : 1.0v

Date : 5-FEB-2019

|  |
| --- |
| **Contents :**   * Abstract. * Introduction. * Extending Spark Rich API : Custom RDD. * Graphical Representation : A Explanation. * Case Study 1: Through Range partitioner in Alphabetic sequence. * Case Study 2: A sphosticated example of studen’s enrollement in college.(code available in github) |

**Abstract** :

Presenting the plug-in facility of external custom datasource model into Spark Resilient Distributed Datasets (RDD) ecosystem. The main fatures of this customized creation of spark RDD is that it allows the user a flixiblily to use extended Spark API according to his/her needs. Although, we can create the RDD directly from the different datasource, we may end up compromizing the performance due to not properly utilizing(also not aware of the rich Spark extended APIs) the leverage which is what Spark actually provides. Plug-in the data source at the granular level has verious advantages. RDDs mainly support *coarse-grained transformations* which means any transformation is applied to whole set of RDD whereas if we do plug-in utility through spark extended API, these will sequencially be applied to each partitions (set of rows) while loading from data source at initial phase.

Currently these research has been well tested on *Spark verson 2.0.2*. We have to verify the same with spark *latest version 2.4.0*.

**Introduction**

Resilient data distribution is the heart of the spark designed to support in-memory data storage, distributed across the nodes in cluster manner in which it provide fault-tolerant. Fault tolerance is achieved by *lineage graph* of transformation applied to coarse-grained sets of data. Moreover, parallelization of the processing across the nodes in cluster attain the efficient.

Two operations are performed once data is loaded into the RDD :

|  |
| --- |
| 1. **Transformation** : Create new RDD by processing as mapping, filtering, and many more. |
| 1. **Actions** : It does not create new RDD rather do some actual intended task such as sum, count, save (to file / hdfs) etc. |

During entire life-cycle of the RDD, transformation from RDD1, RDD2...RDDn, can be repeated in the event of data loss or failure of the node. Spark does elegant recovery of the data from the parent RDD through lineage graph, which it maintain, rather than seeking entire data from initial phase.

Transformation are *lazily evaluated* which means it is not executed until *action* applied. Hence, it improves the performance by avoiding unnecessary data processing.

**Extending Spark rich API : Custom RDD**

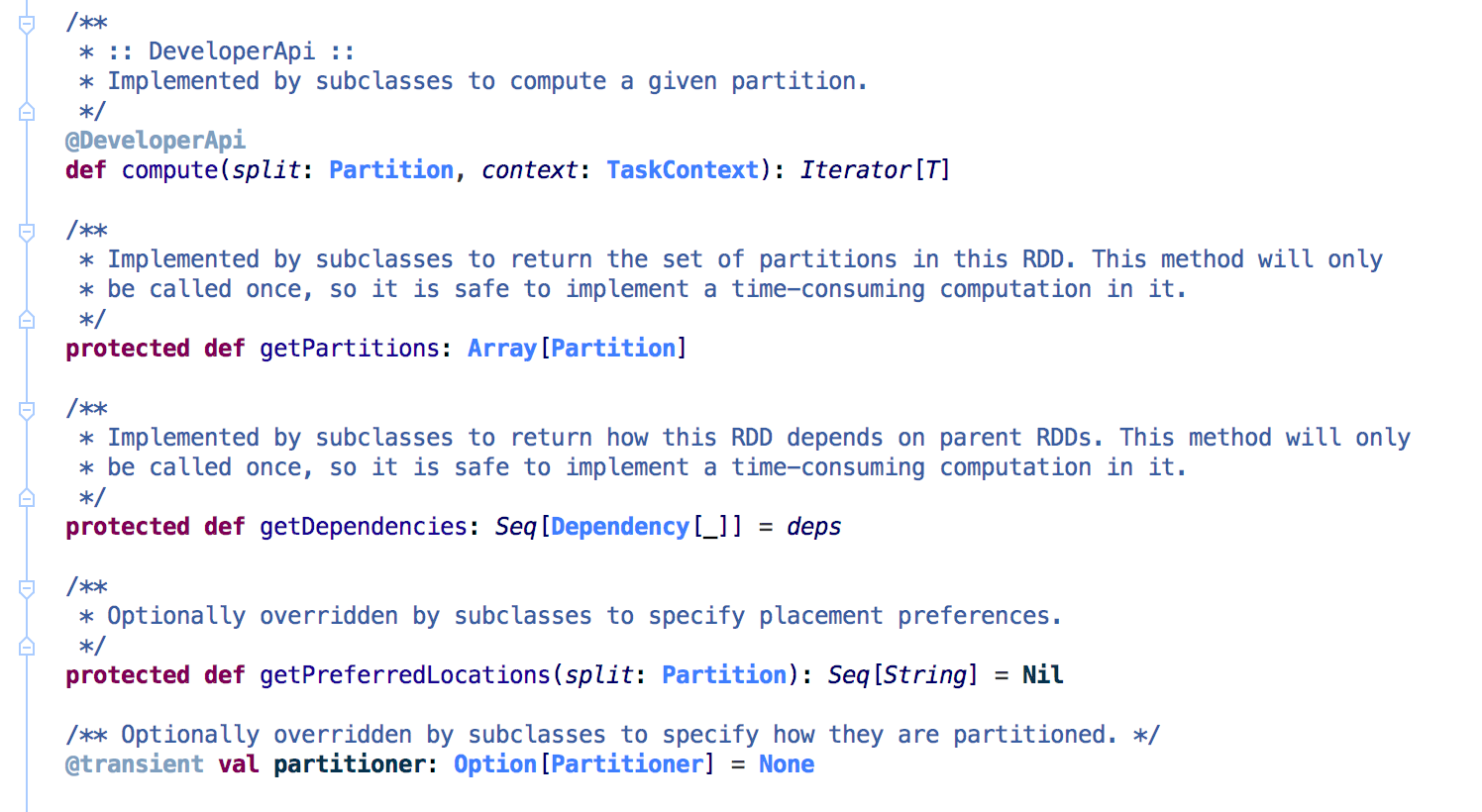
Spark RDD provide many rich APIs extending which custom RDD can be created.Let us dive deep into those APIs to understand how to proceed to write our own custom RDD. Many of such RDD are extended by the spark itself such as Block RDD, JDBCRDD, Pairwise RDD etc.

*Internally RDD is characterised by five main properties :*

|  |
| --- |
| 1. List of partitions. 2. Function for computing each split. 3. Lists of dependencies on other RDDs. 4. Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned). 5. Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HHFS file or HDFS file). |

Let us now explore each of properties of the RDD with suitable case studies in Java and Scala.

**RDD.scala**



*Brief description about these five properties.*

|  |
| --- |
| 1. List of partitions :   Method : **getPartitions**  Return type : Partition[]  Arguments : none  Spark allow developer to specify the new partitions for RDD.  *Description* : It is the block of the data partitioned by some strategy such as partitioner or existing block with parent partitioner none. In other words, it is logical/physical set of rows which form the partition. |

|  |
| --- |
| 2. Compute :  Method : **compute**  Return type : Iterator[T]  Argument : Partition and TaskContext.  *Description* : Its start the computation over each partition for associated task  and return the Iterator for computation.In other words, spark returns the Iterator  of the row going to process. |

|  |
| --- |
| 3. Lists of dependencies on other RDDs.  None to specify if their is no parent RDD. |

|  |
| --- |
| 4. Optionally, a Partitioner :  Method : **partitioner**  Return type : Option  Argument : Partitioner  *Description* : It is optional and if developer wants to define how actually the  RDD should be partitioned based on some Partition algorithm such as *Hash*  *Partition* or *Range Partition*. Range Partition is the case such as SortByKey  whereas Hash Partition is used in other cases for various transformation function. |

|  |
| --- |
| 5. Optionally, a list of preferred locations :  Method : **getPreferredLocations**  Return type : Seq<String>  Argument : Partition  *Description* : Spark provide developer to override this method to specify  the placement preferences for partition. These are the locations for the  partition where actually partition is eligible to get executed if sufficient  resources amongst these preferences is not available. Nonetheless, if resources  are not available in particular time duration, it may get executed in any  of locations/nodes in cluster. |

**Graphical Representation : A Explanation:**

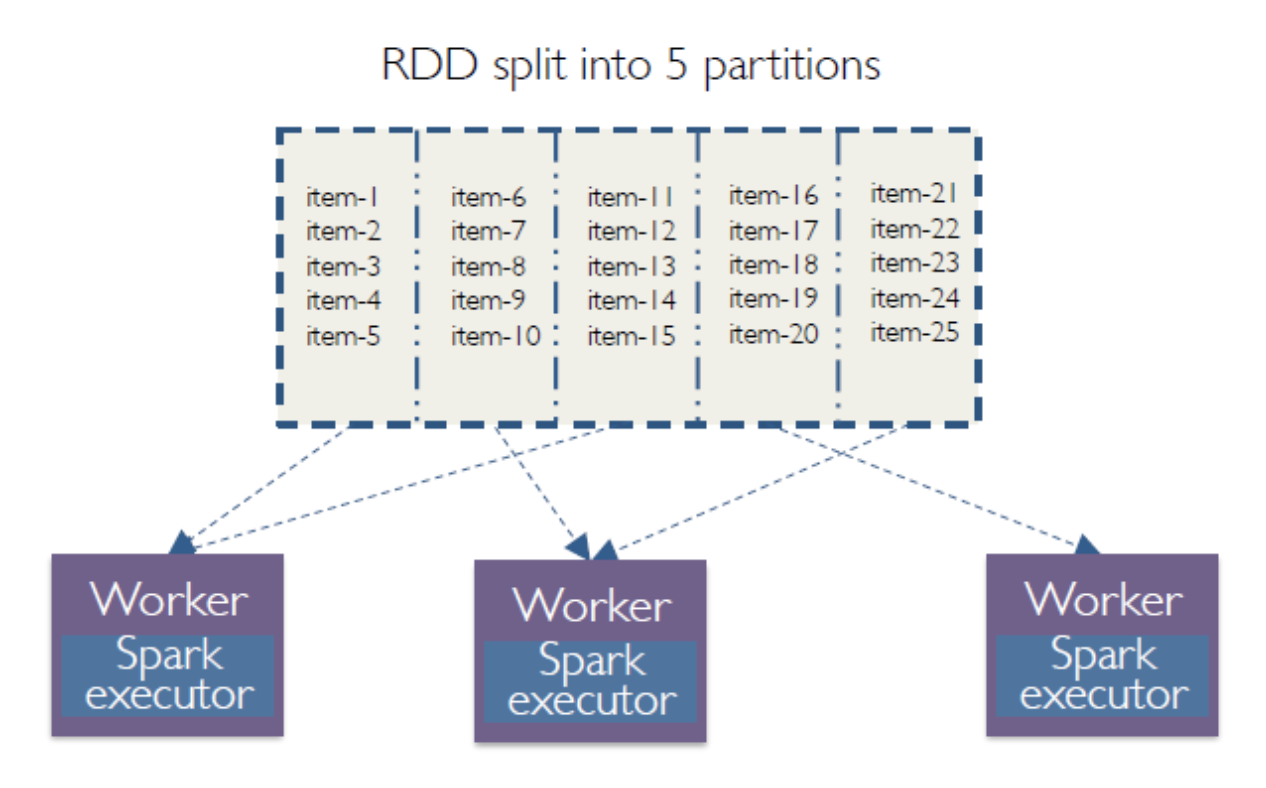


Fig: 1

You can clearly see that one RDD is basically the collections of number of partitions and each partition is collections of number records. Therefore, parallelism can be achieved if partitions are made based on number of Spark executor. You can clearly see that one Spark executor(last one) is performing execution only on one partition. If we could have divided RDD into multiple of number of spark executor, we could have achieved the better performance.

In addition to that, custom implementation of Spark RDD provide the flixibility of registering the partitions under **getPartitions** method**.** Therefore, you don’t need to do any hash partitioning (by default it does while we load file sc.textFile(“datasource”) internally) or range partitioning initially. However, Spark provide the facility as per user’s whims and fancy to create it’s own partitoner. The method **partitioner** provides such facility from RDD. This partitioner will be used during transformation in which another RDD is created from it’s parent RDD (i.e in our case custom RDD).

Let’s take the simple example to understand the different functionality offered by Spark.

**Case Study 1:**

|  |
| --- |
| Suppose we have alphabets from A to Z (A,B,C...Z) and we want to collect the two ranges from A to C (A,B,C) and W to Z (W,X,Y,Z) and final result would be A,B,C,W,X,Y,Z. |

**Code Snippet :**

|  |
| --- |
| import org.apache.spark.Dependency;  import org.apache.spark.Partition;  import org.apache.spark.SparkConf;  import org.apache.spark.SparkContext;  import org.apache.spark.TaskContext;  import org.apache.spark.api.java.JavaSparkContext;  import org.apache.spark.rdd.RDD;  import scala.collection.AbstractIterator;  import scala.collection.Iterator;  import scala.collection.mutable.ArrayBuffer;  import scala.reflect.ClassManifestFactory$;  import scala.reflect.ClassTag; |

|  |
| --- |
| **Partition :**  public static class **AlphabetRangePartition** implements Partition {  private static final long serialVersionUID = 1L;  private int index;  private char from;  private char to;  public AlphabetRangePartition(int index, char c, char d) {  this.index = index;  this.from = c;  this.to = d;  }  @Override  public int index() {  return index;  }  @Override  public int hashCode() {  return index();  }  @Override  public boolean equals(Object obj) {  if (!(obj instanceof AlphabetRangePartition)) {  return false;  }  return ((AlphabetRangePartition) obj).index != index;  }  } |

|  |
| --- |
| **Iterator :**  public static class **CharacterIterator** extends AbstractIterator<String> {  private char next;  private char last;  public CharacterIterator(char from, char to) {  next = from;  this.last = to;  }  @Override  public boolean hasNext() {  return next <= last;  }  @Override  public String next() {  return Character.toString(next++);  }  } |

|  |
| --- |
| **Main:**  public class **CustomRDDMain** {  private static final ClassTag<String> STRING\_TAG =  ClassManifestFactory$.MODULE$.fromClass(String.class);  public static void **main**(final String[] args) {  SparkConf conf = new SparkConf().setMaster("local[2]").setAppName("CustomRDDApp");  try (JavaSparkContext sc = new JavaSparkContext(conf)) {  System.out.println(new AlphabetRDD(sc.sc()).toJavaRDD().collect());  }  }  public static class AlphabetRDD extends RDD<String> {  private static final long serialVersionUID = 1L;  public AlphabetRDD(SparkContext sc) {  super(sc, new ArrayBuffer<Dependency<?>>(), STRING\_TAG);  }  @Override  public Iterator<String> **compute**(Partition partition, TaskContext taskContext) {  AlphabetRangePartition p = (AlphabetRangePartition) partition;  return new CharacterIterator(p.from, p.to);  }  @Override  public Partition[] **getPartitions**() {  return new Partition[] {new AlphabetRangePartition(0, 'A', 'C'),  new AlphabetRangePartition(1, 'W', 'Z')};  }  }  } |

**Case Study 2:**

Let’s take some more example.

Conside the case where we have detailed records of the students in the college/university wherein the student are required to pay the semester fee at the time of admission/enrollement. At the same time it must be ensured that student belonging to the *different category* should have to pay the fee at consession (a kind of scholarship).

You can find above programe in below github repository.

https://github.com/pooshans/SparkRDD

**Reference**:

* + - 1. Picture 1 is taken from online sources(googled it).
      2. <https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>