01_Spark-RDD

November 20, 2018

0.1 Spark RDDs

```
In [2]: # Check for Spark Context
        sc

Out[2]: <SparkContext master=local[*] appName=PySparkShell>
```

0.2 Creating RDDs

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

There are two ways to create RDDs: - parallelizing an existing collection in your driver program, or - referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

Creating SparkContext

To execute any operation in spark, you have to first create object of SparkContext class.

A SparkContext class represents the connection to our existing Spark cluster and provides the entry point for interacting with Spark.

We need to create a SparkContext instance so that we can interact with Spark and distribute our jobs.

0.3 Example 1

```
In [3]: a = range(100)

        data = sc.parallelize(a)
```

we can count() the number of elements in the RDD.

```
In [4]: data.count()

Out[4]: 100
```

we can access the first few elements on our RDD.

```
In [5]: data.take(15)

Out[5]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

0.4 Example 2

```
In [4]: #from pyspark import SparkContext
        #sc=SparkContext()
        data=range(1,10)
        rdd=sc.parallelize(data)
        rdd.collect() # display elements
```

```
Out[4]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

0.5 Example 3

```
In [2]: data1 = ['Hello', 'Social', 'Prachar','Social']
        rdd=sc.parallelize(data1)
        rdd1 = rdd.map(lambda x: (x,1))
        rdd1.collect()
```

```
Out[2]: [('Hello', 1), ('Social', 1), ('Prachar', 1), ('Social', 1)]
```

0.6 Example 4

```
In [6]: import random
        num_samples = 1000
        def inside(p):
            x, y = random.random(), random.random()
            return x*x + y*y < 1
        count = sc.parallelize(range(0, num_samples)).filter(inside).count()
        pi = 4 * count / num_samples
        print(pi)
        sc.stop()
```

```
3.164
```

0.7 Getting help

Alternatively, you can use Python's help() function to get an easier to read list of all the attributes, including examples, that the sc object has.

```
In [2]: # Use help to obtain more detailed information
        help(sc)
```

```
Help on SparkContext in module pyspark.context object:

class SparkContext(builtins.object)
 |  Main entry point for Spark functionality. A SparkContext represents the
 |  connection to a Spark cluster, and can be used to create L{RDD} and
 |  broadcast variables on that cluster.
 |
|  Methods defined here:
|
|  __enter__(self)
|      Enable 'with SparkContext(...) as sc: app(sc)' syntax.
|
|  __exit__(self, type, value, trace)
|      Enable 'with SparkContext(...) as sc: app' syntax.
|
|      Specifically stop the context on exit of the with block.
|
|  __getnewargs__(self)
|
|  __init__(self, master=None, appName=None, sparkHome=None, pyFiles=None, environment=None, ba
|      Create a new SparkContext. At least the master and app name should be set,
|      either through the named parameters here or through C{conf}.
|
|      :param master: Cluster URL to connect to
|             (e.g. mesos://host:port, spark://host:port, local[4]).
|      :param appName: A name for your job, to display on the cluster web UI.
|      :param sparkHome: Location where Spark is installed on cluster nodes.
|      :param pyFiles: Collection of .zip or .py files to send to the cluster
|             and add to PYTHONPATH.  These can be paths on the local file
|             system or HDFS, HTTP, HTTPS, or FTP URLs.
|      :param environment: A dictionary of environment variables to set on
|             worker nodes.
|      :param batchSize: The number of Python objects represented as a single
|             Java object. Set 1 to disable batching, 0 to automatically choose
|             the batch size based on object sizes, or -1 to use an unlimited
|             batch size
|      :param serializer: The serializer for RDDs.
|      :param conf: A L{SparkConf} object setting Spark properties.
|      :param gateway: Use an existing gateway and JVM, otherwise a new JVM
|             will be instantiated.
|      :param jsc: The JavaSparkContext instance (optional).
|      :param profiler_cls: A class of custom Profiler used to do profiling
|             (default is pyspark.profiler.BasicProfiler).
|
|
|      >>> from pyspark.context import SparkContext
|      >>> sc = SparkContext('local', 'test')
|
|      >>> sc2 = SparkContext('local', 'test2') # doctest: +IGNORE_EXCEPTION_DETAIL
|      Traceback (most recent call last):
|          ...
|      ValueError:...
|
|  __repr__(self)
|      Return repr(self).
```

```
|
|  accumulator(self, value, accum_param=None)
|      Create an L{Accumulator} with the given initial value, using a given
|      L{AccumulatorParam} helper object to define how to add values of the
|      data type if provided. Default AccumulatorParams are used for integers
|      and floating-point numbers if you do not provide one. For other types,
|      a custom AccumulatorParam can be used.
|
|  addFile(self, path, recursive=False)
|      Add a file to be downloaded with this Spark job on every node.
|      The C{path} passed can be either a local file, a file in HDFS
|      (or other Hadoop-supported filesystems), or an HTTP, HTTPS or
|      FTP URI.
|
|      To access the file in Spark jobs, use
|      L{SparkFiles.get(fileName)<pyspark.files.SparkFiles.get>} with the
|      filename to find its download location.
|
|      A directory can be given if the recursive option is set to True.
|      Currently directories are only supported for Hadoop-supported filesystems.
|
|      >>> from pyspark import SparkFiles
|      >>> path = os.path.join(tempdir, "test.txt")
|      >>> with open(path, "w") as testFile:
|      ...    _ = testFile.write("100")
|      >>> sc.addFile(path)
|      >>> def func(iterator):
|      ...    with open(SparkFiles.get("test.txt")) as testFile:
|      ...        fileVal = int(testFile.readline())
|      ...        return [x * fileVal for x in iterator]
|      >>> sc.parallelize([1, 2, 3, 4]).mapPartitions(func).collect()
|      [100, 200, 300, 400]
|
|  addPyFile(self, path)
|      Add a .py or .zip dependency for all tasks to be executed on this
|      SparkContext in the future.  The C{path} passed can be either a local
|      file, a file in HDFS (or other Hadoop-supported filesystems), or an
|      HTTP, HTTPS or FTP URI.
|
|  binaryFiles(self, path, minPartitions=None)
|      .. note:: Experimental
|
|      Read a directory of binary files from HDFS, a local file system
|      (available on all nodes), or any Hadoop-supported file system URI
|      as a byte array. Each file is read as a single record and returned
|      in a key-value pair, where the key is the path of each file, the
|      value is the content of each file.
|
|      .. note:: Small files are preferred, large file is also allowable, but
|          may cause bad performance.
|
|  binaryRecords(self, path, recordLength)
|      .. note:: Experimental
|
|      Load data from a flat binary file, assuming each record is a set of numbers
|      with the specified numerical format (see ByteBuffer), and the number of
|      bytes per record is constant.
|
|      :param path: Directory to the input data files
|      :param recordLength: The length at which to split the records
|
|  broadcast(self, value)
|      Broadcast a read-only variable to the cluster, returning a
|      L{Broadcast<pyspark.broadcast.Broadcast>}
|      object for reading it in distributed functions. The variable will
|      be sent to each cluster only once.
|
|  cancelAllJobs(self)
|      Cancel all jobs that have been scheduled or are running.
|
|  cancelJobGroup(self, groupId)
|      Cancel active jobs for the specified group. See L{SparkContext.setJobGroup}
|      for more information.
|
|  dump_profiles(self, path)
|      Dump the profile stats into directory `path`
|
|  emptyRDD(self)
|      Create an RDD that has no partitions or elements.
|
|  getConf(self)
|
|  getLocalProperty(self, key)
|      Get a local property set in this thread, or null if it is missing. See
|      L{setLocalProperty}
|
|  hadoopFile(self, path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConve
|      Read an 'old' Hadoop InputFormat with arbitrary key and value class from HDFS,
|      a local file system (available on all nodes), or any Hadoop-supported file system URI.
|      The mechanism is the same as for sc.sequenceFile.
|
|      A Hadoop configuration can be passed in as a Python dict. This will be converted into a
|      Configuration in Java.
|
|      :param path: path to Hadoop file
|      :param inputFormatClass: fully qualified classname of Hadoop InputFormat
|              (e.g. "org.apache.hadoop.mapred.TextInputFormat")
|       :param keyClass: fully qualified classname of key Writable class
|              (e.g. "org.apache.hadoop.io.Text")
|       :param valueClass: fully qualified classname of value Writable class
|              (e.g. "org.apache.hadoop.io.LongWritable")
|       :param keyConverter: (None by default)
|       :param valueConverter: (None by default)
|       :param conf: Hadoop configuration, passed in as a dict
|              (None by default)
|       :param batchSize: The number of Python objects represented as a single
|              Java object. (default 0, choose batchSize automatically)
|
|   hadoopRDD(self, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=No
|       Read an 'old' Hadoop InputFormat with arbitrary key and value class, from an arbitrary
|       Hadoop configuration, which is passed in as a Python dict.
|       This will be converted into a Configuration in Java.
|       The mechanism is the same as for sc.sequenceFile.
|
|       :param inputFormatClass: fully qualified classname of Hadoop InputFormat
|              (e.g. "org.apache.hadoop.mapred.TextInputFormat")
|       :param keyClass: fully qualified classname of key Writable class
|              (e.g. "org.apache.hadoop.io.Text")
|       :param valueClass: fully qualified classname of value Writable class
|              (e.g. "org.apache.hadoop.io.LongWritable")
|       :param keyConverter: (None by default)
|       :param valueConverter: (None by default)
|       :param conf: Hadoop configuration, passed in as a dict
|              (None by default)
|       :param batchSize: The number of Python objects represented as a single
|              Java object. (default 0, choose batchSize automatically)
|
|   newAPIHadoopFile(self, path, inputFormatClass, keyClass, valueClass, keyConverter=None, valu
|       Read a 'new API' Hadoop InputFormat with arbitrary key and value class from HDFS,
|       a local file system (available on all nodes), or any Hadoop-supported file system URI.
|       The mechanism is the same as for sc.sequenceFile.
|
|       A Hadoop configuration can be passed in as a Python dict. This will be converted into a
|       Configuration in Java
|
|       :param path: path to Hadoop file
|       :param inputFormatClass: fully qualified classname of Hadoop InputFormat
|              (e.g. "org.apache.hadoop.mapreduce.lib.input.TextInputFormat")
|       :param keyClass: fully qualified classname of key Writable class
|              (e.g. "org.apache.hadoop.io.Text")
|       :param valueClass: fully qualified classname of value Writable class
|              (e.g. "org.apache.hadoop.io.LongWritable")
|       :param keyConverter: (None by default)
|       :param valueConverter: (None by default)
|       :param conf: Hadoop configuration, passed in as a dict
|              (None by default)
|       :param batchSize: The number of Python objects represented as a single
|              Java object. (default 0, choose batchSize automatically)
|
|   newAPIHadoopRDD(self, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConver
|       Read a 'new API' Hadoop InputFormat with arbitrary key and value class, from an arbitrar
|       Hadoop configuration, which is passed in as a Python dict.
|       This will be converted into a Configuration in Java.
|       The mechanism is the same as for sc.sequenceFile.
|
|       :param inputFormatClass: fully qualified classname of Hadoop InputFormat
|              (e.g. "org.apache.hadoop.mapreduce.lib.input.TextInputFormat")
|       :param keyClass: fully qualified classname of key Writable class
|              (e.g. "org.apache.hadoop.io.Text")
|       :param valueClass: fully qualified classname of value Writable class
|              (e.g. "org.apache.hadoop.io.LongWritable")
|       :param keyConverter: (None by default)
|       :param valueConverter: (None by default)
|       :param conf: Hadoop configuration, passed in as a dict
|              (None by default)
|       :param batchSize: The number of Python objects represented as a single
|              Java object. (default 0, choose batchSize automatically)
|
|   parallelize(self, c, numSlices=None)
|       Distribute a local Python collection to form an RDD. Using xrange
|       is recommended if the input represents a range for performance.
|
|       >>> sc.parallelize([0, 2, 3, 4, 6], 5).glom().collect()
|       [[0], [2], [3], [4], [6]]
|       >>> sc.parallelize(xrange(0, 6, 2), 5).glom().collect()
|       [[], [0], [], [2], [4]]
|
|   pickleFile(self, name, minPartitions=None)
|       Load an RDD previously saved using L{RDD.saveAsPickleFile} method.
|
|       >>> tmpFile = NamedTemporaryFile(delete=True)
|       >>> tmpFile.close()
|       >>> sc.parallelize(range(10)).saveAsPickleFile(tmpFile.name, 5)
|       >>> sorted(sc.pickleFile(tmpFile.name, 3).collect())
|       [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
|
|   range(self, start, end=None, step=1, numSlices=None)
|       Create a new RDD of int containing elements from `start` to `end`
|       (exclusive), increased by `step` every element. Can be called the same
|       way as python's built-in range() function. If called with a single argument,
|       the argument is interpreted as `end`, and `start` is set to 0.
|
 |      :param start: the start value
 |      :param end: the end value (exclusive)
 |      :param step: the incremental step (default: 1)
 |      :param numSlices: the number of partitions of the new RDD
 |      :return: An RDD of int
 |
 |      >>> sc.range(5).collect()
 |      [0, 1, 2, 3, 4]
 |      >>> sc.range(2, 4).collect()
 |      [2, 3]
 |      >>> sc.range(1, 7, 2).collect()
 |      [1, 3, 5]
 |
 |  runJob(self, rdd, partitionFunc, partitions=None, allowLocal=False)
 |      Executes the given partitionFunc on the specified set of partitions,
 |      returning the result as an array of elements.
 |
 |      If 'partitions' is not specified, this will run over all partitions.
 |
 |      >>> myRDD = sc.parallelize(range(6), 3)
 |      >>> sc.runJob(myRDD, lambda part: [x * x for x in part])
 |      [0, 1, 4, 9, 16, 25]
 |
 |      >>> myRDD = sc.parallelize(range(6), 3)
 |      >>> sc.runJob(myRDD, lambda part: [x * x for x in part], [0, 2], True)
 |      [0, 1, 16, 25]
 |
 |  sequenceFile(self, path, keyClass=None, valueClass=None, keyConverter=None, valueConverter=N
 |      Read a Hadoop SequenceFile with arbitrary key and value Writable class from HDFS,
 |      a local file system (available on all nodes), or any Hadoop-supported file system URI.
 |      The mechanism is as follows:
 |
 |          1. A Java RDD is created from the SequenceFile or other InputFormat, and the key
 |             and value Writable classes
 |          2. Serialization is attempted via Pyrolite pickling
 |          3. If this fails, the fallback is to call 'toString' on each key and value
 |          4. C{PickleSerializer} is used to deserialize pickled objects on the Python side
 |
 |      :param path: path to sequncefile
 |      :param keyClass: fully qualified classname of key Writable class
 |             (e.g. "org.apache.hadoop.io.Text")
 |      :param valueClass: fully qualified classname of value Writable class
 |             (e.g. "org.apache.hadoop.io.LongWritable")
 |      :param keyConverter:
 |      :param valueConverter:
 |      :param minSplits: minimum splits in dataset
 |             (default min(2, sc.defaultParallelism))
 |      :param batchSize: The number of Python objects represented as a single
```

```
|              Java object. (default 0, choose batchSize automatically)
|
|  setCheckpointDir(self, dirName)
|      Set the directory under which RDDs are going to be checkpointed. The
|      directory must be a HDFS path if running on a cluster.
|
|  setJobDescription(self, value)
|      Set a human readable description of the current job.
|
|  setJobGroup(self, groupId, description, interruptOnCancel=False)
|      Assigns a group ID to all the jobs started by this thread until the group ID is set to a
|      different value or cleared.
|
|      Often, a unit of execution in an application consists of multiple Spark actions or jobs.
|      Application programmers can use this method to group all those jobs together and give a
|      group description. Once set, the Spark web UI will associate such jobs with this group.
|
|      The application can use L{SparkContext.cancelJobGroup} to cancel all
|      running jobs in this group.
|
|      >>> import threading
|      >>> from time import sleep
|      >>> result = "Not Set"
|      >>> lock = threading.Lock()
|      >>> def map_func(x):
|      ...     sleep(100)
|      ...     raise Exception("Task should have been cancelled")
|      >>> def start_job(x):
|      ...     global result
|      ...     try:
|      ...         sc.setJobGroup("job_to_cancel", "some description")
|      ...         result = sc.parallelize(range(x)).map(map_func).collect()
|      ...     except Exception as e:
|      ...         result = "Cancelled"
|      ...     lock.release()
|      >>> def stop_job():
|      ...     sleep(5)
|      ...     sc.cancelJobGroup("job_to_cancel")
|      >>> supress = lock.acquire()
|      >>> supress = threading.Thread(target=start_job, args=(10,)).start()
|      >>> supress = threading.Thread(target=stop_job).start()
|      >>> supress = lock.acquire()
|      >>> print(result)
|      Cancelled
|
|      If interruptOnCancel is set to true for the job group, then job cancellation will result
|      in Thread.interrupt() being called on the job's executor threads. This is useful to help
|      ensure that the tasks are actually stopped in a timely manner, but is off by default due
```

```
 |      to HDFS-1208, where HDFS may respond to Thread.interrupt() by marking nodes as dead.
 |  
 |  setLocalProperty(self, key, value)
 |      Set a local property that affects jobs submitted from this thread, such as the
 |      Spark fair scheduler pool.
 |  
 |  setLogLevel(self, logLevel)
 |      Control our logLevel. This overrides any user-defined log settings.
 |      Valid log levels include: ALL, DEBUG, ERROR, FATAL, INFO, OFF, TRACE, WARN
 |  
 |  show_profiles(self)
 |      Print the profile stats to stdout
 |  
 |  sparkUser(self)
 |      Get SPARK_USER for user who is running SparkContext.
 |  
 |  statusTracker(self)
 |      Return :class:`StatusTracker` object
 |  
 |  stop(self)
 |      Shut down the SparkContext.
 |  
 |  textFile(self, name, minPartitions=None, use_unicode=True)
 |      Read a text file from HDFS, a local file system (available on all
 |      nodes), or any Hadoop-supported file system URI, and return it as an
 |      RDD of Strings.
 |  
 |      If use_unicode is False, the strings will be kept as `str` (encoding
 |      as `utf-8`), which is faster and smaller than unicode. (Added in
 |      Spark 1.2)
 |  
 |      >>> path = os.path.join(tempdir, "sample-text.txt")
 |      >>> with open(path, "w") as testFile:
 |      ...    _ = testFile.write("Hello world!")
 |      >>> textFile = sc.textFile(path)
 |      >>> textFile.collect()
 |      ['Hello world!']
 |  
 |  union(self, rdds)
 |      Build the union of a list of RDDs.
 |  
 |      This supports unions() of RDDs with different serialized formats,
 |      although this forces them to be reserialized using the default
 |      serializer:
 |  
 |      >>> path = os.path.join(tempdir, "union-text.txt")
 |      >>> with open(path, "w") as testFile:
 |      ...    _ = testFile.write("Hello")
```

```
|      >>> textFile = sc.textFile(path)
|      >>> textFile.collect()
|      ['Hello']
|      >>> parallelized = sc.parallelize(["World!"])
|      >>> sorted(sc.union([textFile, parallelized]).collect())
|      ['Hello', 'World!']
|
|  wholeTextFiles(self, path, minPartitions=None, use_unicode=True)
|      Read a directory of text files from HDFS, a local file system
|      (available on all nodes), or any  Hadoop-supported file system
|      URI. Each file is read as a single record and returned in a
|      key-value pair, where the key is the path of each file, the
|      value is the content of each file.
|
|      If use_unicode is False, the strings will be kept as `str` (encoding
|      as `utf-8`), which is faster and smaller than unicode. (Added in
|      Spark 1.2)
|
|      For example, if you have the following files::
|
|        hdfs://a-hdfs-path/part-00000
|        hdfs://a-hdfs-path/part-00001
|        ...
|        hdfs://a-hdfs-path/part-nnnnn
|
|      Do C{rdd = sparkContext.wholeTextFiles("hdfs://a-hdfs-path")},
|      then C{rdd} contains::
|
|        (a-hdfs-path/part-00000, its content)
|        (a-hdfs-path/part-00001, its content)
|        ...
|        (a-hdfs-path/part-nnnnn, its content)
|
|      .. note:: Small files are preferred, as each file will be loaded
|          fully in memory.
|
|      >>> dirPath = os.path.join(tempdir, "files")
|      >>> os.mkdir(dirPath)
|      >>> with open(os.path.join(dirPath, "1.txt"), "w") as file1:
|      ...    _ = file1.write("1")
|      >>> with open(os.path.join(dirPath, "2.txt"), "w") as file2:
|      ...    _ = file2.write("2")
|      >>> textFiles = sc.wholeTextFiles(dirPath)
|      >>> sorted(textFiles.collect())
|      [('.../1.txt', '1'), ('.../2.txt', '2')]
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
```

```
|
|  getOrCreate(conf=None) from builtins.type
|      Get or instantiate a SparkContext and register it as a singleton object.
|
|      :param conf: SparkConf (optional)
|
|  setSystemProperty(key, value) from builtins.type
|      Set a Java system property, such as spark.executor.memory. This must
|      must be invoked before instantiating SparkContext.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  applicationId
|      A unique identifier for the Spark application.
|      Its format depends on the scheduler implementation.
|
|      * in case of local spark app something like 'local-1433865536131'
|      * in case of YARN something like 'application_1433865536131_34483'
|
|      >>> sc.applicationId  # doctest: +ELLIPSIS
|      'local-...'
|
|  defaultMinPartitions
|      Default min number of partitions for Hadoop RDDs when not given by user
|
|  defaultParallelism
|      Default level of parallelism to use when not given by user (e.g. for
|      reduce tasks)
|
|  startTime
|      Return the epoch time when the Spark Context was started.
|
|  uiWebUrl
|      Return the URL of the SparkUI instance started by this SparkContext
|
|  version
|      The version of Spark on which this application is running.
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
 |  PACKAGE_EXTENSIONS = ('.zip', '.egg', '.jar')
```

```
In [3]: # Help can be used on any Python object
        help(map)
```

```
Help on class map in module builtins:

class map(object)
 |  map(func, *iterables) --> map object
 |
 |  Make an iterator that computes the function using arguments from
 |  each of the iterables.  Stops when the shortest iterable is exhausted.
 |
 |  Methods defined here:
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __reduce__(...)
 |      Return state information for pickling.
```

0.7.1 Create a Python collection of integers in the range of 1 .. 1001

We will use the xrange() function to create a list() of integers. xrange() only generates values as they are needed. This is different from the behavior of range() which generates the complete list upon execution. Because of this xrange() is more memory efficient than range(), especially for large ranges.

```
In [8]: data = range(1, 1001)
```

```
In [9]: # Data is just a normal Python list
        # Obtain data's first element
        data[0]
```

```
Out[9]: 1
```

```
In [10]: # We can check the size of the list using the len() function
         len(data)

Out[10]: 1000
```

0.7.2 Distributed data and using a collection to create an RDD

In Spark, datasets are represented as a list of entries, where the list is broken up into many different partitions that are each stored on a different machine. Each partition holds a unique subset of the entries in the list. Spark calls datasets that it stores "Resilient Distributed Datasets" (RDDs).

One of the defining features of Spark, compared to other data analytics frameworks (e.g., Hadoop), is that it stores data in memory rather than on disk. This allows Spark applications to run much more quickly, because they are not slowed down by needing to read data from disk.

```
In [13]: # Parallelize data using 8 partitions
         # This operation is a transformation of data into an RDD
         # Spark uses lazy evaluation, so no Spark jobs are run at this point
         rangeRDD = sc.parallelize(data, 8)

In [19]: rangeRDD.take(5) #collect entries from all partitions

Out[19]: [1, 2, 3, 4, 5]
```

1 What is Transformation and Action?

Spark has certain operations which can be performed on RDD. An operation is a method, which can be applied on a RDD to accomplish certain task. RDD supports two types of operations, which are Action and Transformation. An operation can be something as simple as sorting, filtering and summarizing data.

Transformation: Transformation refers to the operation applied on a RDD to create new RDD. Filter, groupBy and map are the examples of transformations.

Actions: Actions refer to an operation which also applies on RDD, that instructs Spark to perform computation and send the result back to driver. This is an example of action.

1.1 Transformation: map and flatMap

When applying the map() transformation, each item in the parent RDD will map to one element in the new RDD.

```
In [17]: wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
         # Not parallelize instantly
         wordsRDD = sc.parallelize(wordsList, 4)

         # Previous transformation will actually executed here
         wordsRDD.collect()

Out[17]: ['cat', 'elephant', 'rat', 'rat', 'cat']

In [23]: d= wordsRDD.distinct()
         print(d)
```

```
PythonRDD[50] at RDD at PythonRDD.scala:53
```

```
In [19]: wordPairs = wordsRDD.map(lambda w: (w, 1))
         wordPairs.collect()
```

```
Out[19]: [('cat', 1), ('elephant', 1), ('rat', 1), ('rat', 1), ('cat', 1)]
```

flatMap() is similar to map(), except that with flatMap() each input item can be mapped to zero or more output elements.

```
In [27]: # Use map
         singularAndPluralWordsRDDMap = wordsRDD.map(lambda x: (x, x + 's'))

         # Use flatMap
         singularAndPluralWordsRDD = wordsRDD.flatMap(lambda x: (x, x + 's'))

         # View the results
         print (singularAndPluralWordsRDDMap.collect())
         print (singularAndPluralWordsRDD.collect())

         # View the number of elements in the RDD
         print (singularAndPluralWordsRDDMap.count())
         print (singularAndPluralWordsRDD.count())
```

```
[('cat', 'cats'), ('elephant', 'elephants'), ('rat', 'rats'), ('rat', 'rats'), ('cat', 'cats')]
['cat', 'cats', 'elephant', 'elephants', 'rat', 'rats', 'rat', 'rats', 'cat', 'cats']
5
10
```

```
In [28]: sc.parallelize([3,4,5]).map(lambda x: [x,  x*x]).collect()
```

```
Out[28]: [[3, 9], [4, 16], [5, 25]]
```

```
In [29]: sc.parallelize([3,4,5]).flatMap(lambda x: [x, x*x]).collect()
```

```
Out[29]: [3, 9, 4, 16, 5, 25]
```

1.2 Transformation: filter

We can use a "filter" transformation which will return a new RDD containing only the elements that satisfy given condition(s).

```
In [30]: x = sc.parallelize([1,2,3,4,5,6,7,8,9,10], 2)

         # filter operation
         y = x.filter(lambda x: x % 2 == 0)
         y.collect()
         # [2, 4, 6, 8, 10]
```

```
Out[30]: [2, 4, 6, 8, 10]
```

1.3 Transformation: groupByKey / reduceByKey

We can apply the "groupByKey" / "reduceByKey" transformations on (key,val) pair RDD. The "groupByKey" will group the values for each key in the original RDD. It will create a new pair, where the original key corresponds to this collected group of values.

```
In [55]: rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
         sorted(rdd.groupByKey().mapValues(len).collect())

Out[55]: [('a', 2), ('b', 1)]

In [56]: sorted(rdd.groupByKey().mapValues(list).collect())

Out[56]: [('a', [1, 1]), ('b', [1])]

In [37]: wordCountsCollected = wordPairs.reduceByKey(lambda x, y: x + y)
         wordCountsCollected.collect()

Out[37]: [('cat', 2), ('elephant', 1), ('rat', 2)]

In [59]: from operator import add
         rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
         sorted(rdd.reduceByKey(add).collect())

Out[59]: [('a', 2), ('b', 1)]
```

1.4 Action: Reduce

A reduce action is use for aggregating all the elements of RDD by applying pairwise user function.

```
In [3]: num_rdd = sc.parallelize(range(1,1001))
        num_rdd.reduce(lambda x,y: x+y)

Out[3]: 500500
```

1.5 Action: count

The count action will count the number of elements in RDD

```
In [4]: num_rdd.count()

Out[4]: 1000
```

1.6 Action: max, min, sum, variance and stdev

```
In [7]: num_rdd.max(), num_rdd.min()

Out[7]: (1000, 1)

In [8]: num_rdd.sum(),num_rdd.variance(),num_rdd.stdev()

Out[8]: (500500, 83333.25, 288.67499025720952)
```

1.7 Action: getNumPartitions

With "getNumPartitions", we can find out that how many partitions exist in our RDD.

```
In [9]: num_rdd.getNumPartitions()
```

```
Out[9]: 1
```

1.8 Transformation: distinct

We can apply "distinct" transformation on RDD to get the distinct elements.

```
In [25]: wd = wordsRDD.distinct()
         len(wd.collect())
```

```
Out[25]: 3
```

1.9 Creating RDD from Files

```
In [26]: rdd = sc.textFile("dil.txt")
         rdd.collect()
```

```
Out[26]: ['101 ramesh', '102 suresh', '103 kamesh']
```

1.10 Example: HDFS WordCount

```
In [2]: # load file from hdfs
        contentRDD =sc.textFile("hdfs://localhost:9000/input/sample.txt")
```

```
In [3]: # omitting empty lines
        nonempty_lines = contentRDD.filter(lambda x: len(x) > 0)

        # Split the content based on space
        words = nonempty_lines.flatMap(lambda x: x.split(' '))

        # Perform a Map and Reduce Task
        wordcount = words.map(lambda x:(x,1)).reduceByKey(lambda x,y: x+y).map(lambda x: (x[1],

        # collect words and print in descending order
        for word in wordcount.collect():
            print(word)
```

```
(1, 'hello')
(1, 'how')
(1, 'r')
(1, 'u')
(1, 'social')
(1, 'prachar')
```

```
In [ ]:
```