Spark Tutorial: Learning Apache Spark

This tutorial will teach you how to use Apache Spark (http://spark.apache.org/), a framework for large-scale data processing, within a notebook. Many traditional frameworks were designed to be run on a single computer. However, many datasets today are too large to be stored on a single computer, and even when a dataset can be stored on one computer (such as the datasets in this tutorial), the dataset can often be processed much more quickly using multiple computers. Spark has efficient implementations of a number of transformations and actions that can be composed together to perform data processing and analysis. Spark excels at distributing these operations across a cluster while abstracting away many of the underlying implementation details. Spark has been designed with a focus on scalability and efficiency. With Spark you can begin developing your solution on your laptop, using a small dataset, and then use that same code to process terabytes or even petabytes across a distributed cluster.

During this tutorial we will cover:

Part 1: Basic notebook usage and Python (https://docs.python.org/2/) integration

Part 2: An introduction to using Apache Spark (https://spark.apache.org/) with the Python pySpark API (https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD) running in the browser

Part 3: Using RDDs and chaining together transformations and actions

Part 4: Lambda functions

Part 5: Additional RDD actions

Part 6: Additional RDD transformations

Part 7: Caching RDDs and storage options

Part 8: Debugging Spark applications and lazy evaluation

Part 1: Basic notebook usage and Python (https://docs.python.org/2/) integration

(1a) Notebook usage

A notebook is comprised of a linear sequence of cells. These cells can contain either markdown or code, but we won't mix both in one cell. When a markdown cell is executed it renders formatted text, images, and links just like HTML in a normal webpage. The text you are reading right now is part of a markdown cell. Python code cells allow you to execute arbitrary Python commands just like in any Python shell. Place your cursor inside the cell below, and press "Shift" + "Enter" to execute the code and advance to the next cell. You can also press "Ctrl" + "Enter" to execute the code and remain in the cell. These commands work the same in both markdown and code cells.

In [1]:
```
# This is a Python cell. You can run normal Python code here...
```
```
The sum of 1 and 1 is 2
```

In [2]:
```
# Here is another Python cell, this time with a variable (x) declaration and an
x = 42
y = 10
if x > 40:
    print 'The sum of 1 and 2 is {0}'.format(1+2)
if y == 10:
```
```
The sum of 1 and 2 is 3
match
```

(1b) Notebook state

As you work through a notebook it is important that you run all of the code cells. The notebook is stateful, which means that variables and their values are retained until the notebook is detached (in Databricks Cloud) or the kernel is restarted (in IPython notebooks). If you do not run all of the code cells as you proceed through the notebook, your variables will not be properly initialized and later code might fail. You will also need to rerun any cells that you have modified in order for the changes to be available to other cells.

In [3]:
```
# This cell relies on x being defined already.
# If we didn't run the cells from part (1a) this code would fail.
```
```
84
```

(1c) Library imports

We can import standard Python libraries (modules (https://docs.python.org/2/tutorial/modules.html)) the usual way. An `import` statement will import the specified module. In this tutorial and future labs, we will provide any imports that are necessary.

In [4]:
```
# Import the regular expression library
import re
m = re.search('(?<=abc)def', 'abcdef')
```

Out[4]:
```
'def'
```

In [5]:
```
# Import the datetime library
import datetime
```

```
This was last run on: 2015-08-07 14:38:08.008935
```

Part 2: An introduction to using Apache Spark (https://spark.apache.org/) with the Python pySpark API (https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD) running in the browser

Spark Context

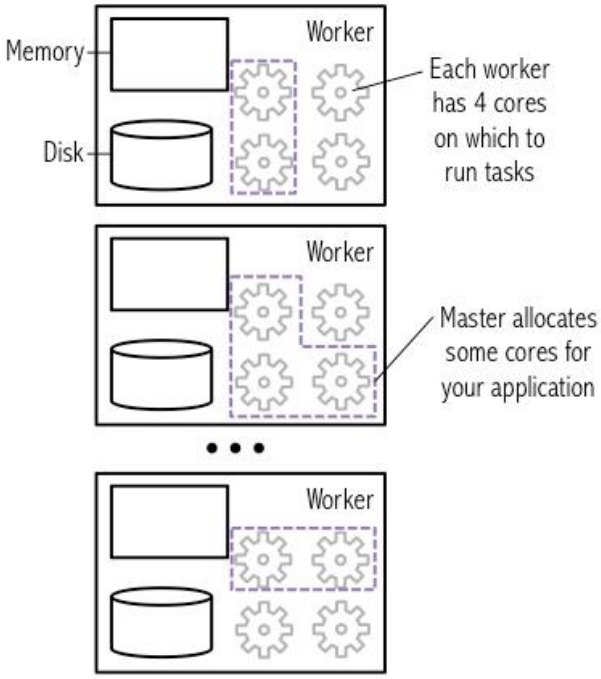
In Spark, communication occurs between a driver and executors. The driver has Spark jobs that it needs to run and these jobs are split into tasks that are submitted to the executors for completion. The results from these tasks are delivered back to the driver.

In part 1, we saw that normal python code can be executed via cells. When using Databricks Cloud this code gets executed in the Spark driver's Java Virtual Machine (JVM) and not in an executor's JVM, and when using an IPython notebook it is executed within the kernel associated with the notebook. Since no Spark functionality is actually being used, no tasks are launched on the executors.

In order to use Spark and its API we will need to use a `SparkContext`. When running Spark, you start a new Spark application by creating a SparkContext (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext). When the `SparkContext` is created, it asks the master for some cores to use to do work. The master sets these cores aside just for you; they won't be used for other applications. When using Databricks Cloud or the virtual machine provisioned for this class, the `SparkContext` is created for you automatically as `sc`.

(2a) Example Cluster

The diagram below shows an example cluster, where the cores allocated for an application are outlined in purple.



You can view the details of your Spark application in the Spark web UI. The web UI is accessible in Databricks cloud by going to "Clusters" and then clicking on the "View Spark UI" link for your cluster. When running locally you'll find it at localhost:4040 (http://localhost:4040). In the web UI, under the "Jobs" tab, you can see a list of jobs that have been scheduled or run. It's likely there isn't any thing interesting here yet because we haven't run any jobs, but we'll return to this page later.

At a high level, every Spark application consists of a driver program that launches various parallel operations on executor Java Virtual Machines (JVMs) running either in a cluster or locally on the same machine. In Databricks Cloud, "Databricks Shell" is the driver program. When running locally, "PySparkShell" is the driver program. In all cases, this driver program contains the main loop for the program and creates distributed datasets on the cluster, then applies operations (transformations & actions) to those datasets.

Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster. A Spark context object (`sc`) is the main entry point for Spark functionality. A Spark context can be used to create Resilient Distributed Datasets (RDDs) on a cluster.

Try printing out `sc` to see its type.

In [6]:
```
# Display the type of the Spark Context sc
```

Out[6]:
```
pyspark.context.SparkContext
```

(2b) `SparkContext` attributes

You can use Python's dir() (https://docs.python.org/2/library/functions.html?highlight=dir#dir) function to get a list of all the attributes (including methods) accessible through the `sc` object.

In [7]:
```
# List sc's attributes
```

Out[7]:
```
['PACKAGE_EXTENSIONS',
 '__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__enter__',
 '__exit__',
 '__format__',
 '__getattribute__',
 '__getnewargs__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_accumulatorServer',
 '_active_spark_context',
 '_batchSize',
 '_callsite',
 '_checkpointFile',
 '_conf',
 '_dictToJavaMap',
 '_do_init',
 '_ensure_initialized',
 '_gateway',
 '_getJavaStorageLevel',
 '_initialize_context',
 '_javaAccumulator',
 '_jsc',
 '_jvm',
 '_lock',
 '_next_accum_id',
 '_pickled_broadcast_vars',
 '_python_includes',
 '_temp_dir',
 '_unbatched_serializer',
 'accumulator',
 'addFile',
 'addPyFile',
 'appName',
 'binaryFiles',
 'binaryRecords',
 'broadcast',
 'cancelAllJobs',
 'cancelJobGroup',
 'clearFiles',
 'defaultMinPartitions',
 'defaultParallelism',
 'dump_profiles',
 'environment',
 'getLocalProperty',
 'hadoopFile',
 'hadoopRDD',
 'master',
 'newAPIHadoopFile',
 'newAPIHadoopRDD',
 'parallelize',
 'pickleFile',
 'profiler_collector',
 'pythonExec',
```

(2c) Getting help

Alternatively, you can use Python's help() (https://docs.python.org/2/library/functions.html?highlight=help#help) function to get an easier to read list of all the attributes, including examples, that the `sc` object has.

In [8]:
```
# Use help to obtain more detailed information
```

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

class SparkContext(__builtin__.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, batchSize=0, serializer=PickleSerializer(), conf=None, gat
eway=None, jsc=None, profiler_cls=<class 'pyspark.profiler.BasicProfiler'>)
 |      Create a new SparkContext. At least the master and app name should b
e 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 U
I.
 |      :param sparkHome: Location where Spark is installed on cluster nodes
.
 |      :param pyFiles: Collection of .zip or .py files to send to the clust
er
 |             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 sing
le
 |             Java object. Set 1 to disable batching, 0 to automatically ch
oose
 |             the batch size based on object sizes, or -1 to use an unlimit
ed
 |             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_EXCEPTIO
N_DETAIL
 |      Traceback (most recent call last):
 |          ...
 |      ValueError:...
 |
 |  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 intege
```

In [9]:
```
# After reading the help we've decided we want to use sc.version to see what ve
#sc.version
```

Out[9]: 1

In [10]:
```
# Help can be used on any Python object
```

```
Help on built-in function map in module __builtin__:

map(...)
    map(function, sequence[, sequence, ...]) -> list

    Return a list of the results of applying the function to the items of
    the argument sequence(s).  If more than one sequence is given, the
    function is called with an argument list consisting of the corresponding
    item of each sequence, substituting None for missing values when not all
    sequences have the same length.  If the function is None, return a list
of
    the items of the sequence (or a list of tuples if more than one sequence
).
```

Part 3: Using RDDs and chaining together transformations and actions

Working with your first RDD

In Spark, we first create a base Resilient Distributed Dataset (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD) (RDD). We can then apply one or more transformations to that base RDD. *An RDD is immutable, so once it is created, it cannot be changed.* As a result, each transformation creates a new RDD. Finally, we can apply one or more actions to the RDDs. Note that Spark uses lazy evaluation, so transformations are not actually executed until an action occurs.

We will perform several exercises to obtain a better understanding of RDDs:

- *Create a Python collection of 10,000 integers*
- *Create a Spark base RDD from that collection*
- *Subtract one from each value using* `map`
- *Perform action* `collect` *to view results*
- *Perform action* `count` *to view counts*
- *Apply transformation* `filter` *and view results with* `collect`
- *Learn about lambda functions*
- *Explore how lazy evaluation works and the debugging challenges that it introduces*

(3a) Create a Python collection of integers in the range of 1 .. 10000

We will use the xrange() (https://docs.python.org/2/library/functions.html?highlight=xrange#xrange) function to create a list() (https://docs.python.org/2/library/functions.html?highlight=list#list) of integers. `xrange()` only generates values as they are needed. This is different from the behavior of range() (https://docs.python.org/2/library/functions.html?highlight=range#range) which generates the complete list upon execution. Because of this `xrange()` is more memory efficient than `range()`, especially for large ranges.

In [11]:
```
```

In [12]:
```
# Data is just a normal Python list
# Obtain data's first element
```

Out[12]: 1

In [13]:
```
# We can check the size of the list using the len() function
```

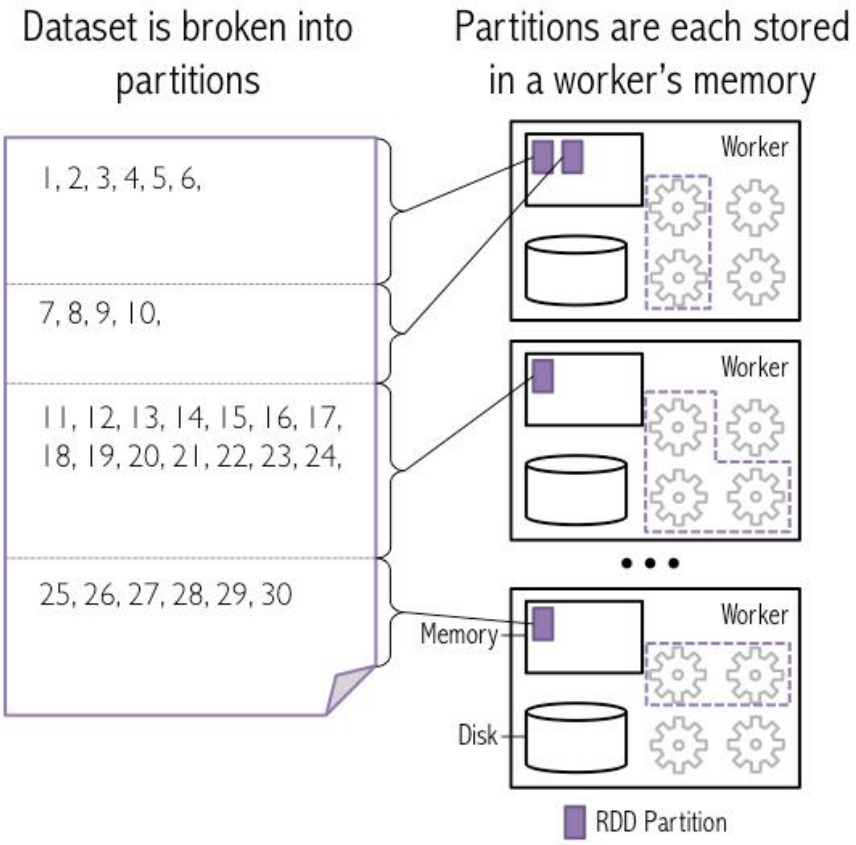
Out[13]: 10000

(3b) 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.

The figure below illustrates how Spark breaks a list of data entries into partitions that are each stored in memory on a worker.



To create the RDD, we use `sc.parallelize()`, which tells Spark to create a new set of input data based on data that is passed in. In this example, we will provide an `xrange`. The second argument to the sc.parallelize() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.SparkContext.parallelize) method tells Spark how many partitions to break the data into when it stores the data in memory (we'll talk more about this later in this tutorial). Note that for better performance when using `parallelize`, `xrange()` is recommended if the input represents a range. This is the reason why we used `xrange()` in 3a.

There are many different types of RDDs. The base class for RDDs is pyspark.RDD

In [14]:
```
# 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
```

In [15]:
```
# Let's view help on parallelize
```

```
Help on method parallelize in module pyspark.context:

parallelize(self, c, numSlices=None) method of pyspark.context.SparkContext
instance
    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]]
```

In [16]:
```
# Let's see what type sc.parallelize() returned
print 'type of xrangeRDD: {0}'.format(type(xrangeRDD))

# How about if we use a range
dataRange = range(1, 10001)
rangeRDD = sc.parallelize(dataRange, 8)
```

```
type of xrangeRDD: <class 'pyspark.rdd.PipelinedRDD'>
type of dataRangeRDD: <class 'pyspark.rdd.RDD'>
```

In [17]:
```
# Each RDD gets a unique ID
print 'xrangeRDD id: {0}'.format(xrangeRDD.id())
```

```
xrangeRDD id: 2
rangeRDD id: 1
```

In [18]:
```
# We can name each newly created RDD using the setName() method
```

Out[18]:
```
My first RDD PythonRDD[2] at RDD at PythonRDD.scala:43
```

In [19]:
```
# Let's view the lineage (the set of transformations) of the RDD using toDebugSt
```

```
(8) My first RDD PythonRDD[2] at RDD at PythonRDD.scala:43 []
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:392 []
```

In [20]:
```
# Let's use help to see what methods we can call on this RDD
```

```
Help on PipelinedRDD in module pyspark.rdd object:

class PipelinedRDD(RDD)
 |  Pipelined maps:
 |
 |  >>> rdd = sc.parallelize([1, 2, 3, 4])
 |  >>> rdd.map(lambda x: 2 * x).cache().map(lambda x: 2 * x).collect()
 |  [4, 8, 12, 16]
 |  >>> rdd.map(lambda x: 2 * x).map(lambda x: 2 * x).collect()
 |  [4, 8, 12, 16]
 |
 |  Pipelined reduces:
 |  >>> from operator import add
 |  >>> rdd.map(lambda x: 2 * x).reduce(add)
 |  20
 |  >>> rdd.flatMap(lambda x: [x, x]).reduce(add)
 |  20
 |
 |  Method resolution order:
 |      PipelinedRDD
 |      RDD
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |
 |  __init__(self, prev, func, preservesPartitioning=False)
 |
 |  id(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from RDD:
 |
 |  __add__(self, other)
 |      Return the union of this RDD and another one.
 |
 |      >>> rdd = sc.parallelize([1, 1, 2, 3])
 |      >>> (rdd + rdd).collect()
 |      [1, 1, 2, 3, 1, 1, 2, 3]
 |
 |  __getnewargs__(self)
 |
 |  __repr__(self)
 |
 |  aggregate(self, zeroValue, seqOp, combOp)
 |      Aggregate the elements of each partition, and then the results for a
ll
 |      the partitions, using a given combine functions and a neutral "zero
 |      value."
 |
 |      The functions C{op(t1, t2)} is allowed to modify C{t1} and return it
 |      as its result value to avoid object allocation; however, it should n
ot
 |      modify C{t2}.
 |
 |      The first function (seqOp) can return a different result type, U, th
an
 |      the type of this RDD. Thus, we need one operation for merging a T in
to
 |      an U and one operation for merging two U
 |
 |      >>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
 |      >>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
 |      >>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)
 |      (10, 4)
 |      >>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)
```

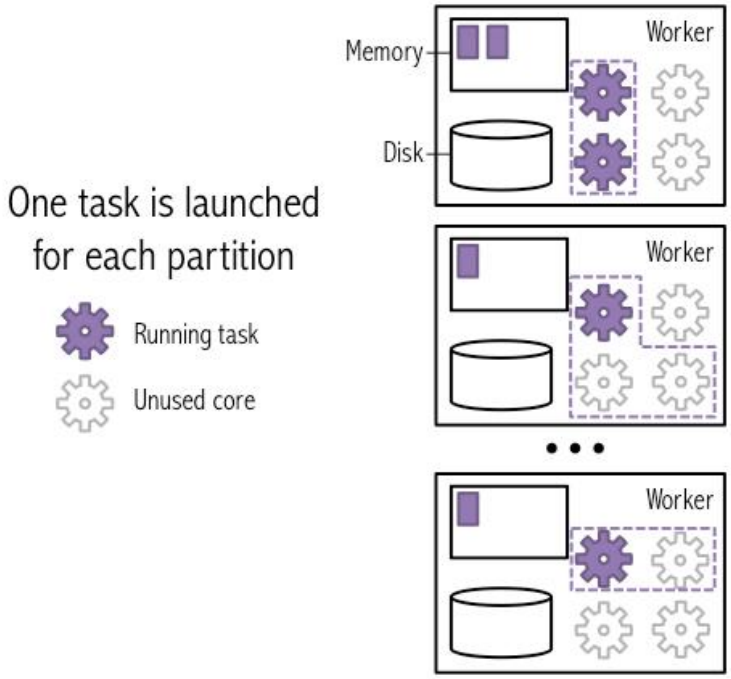
In [21]:
```
# Let's see how many partitions the RDD will be split into by using the getNumPa
```

Out[21]: 8

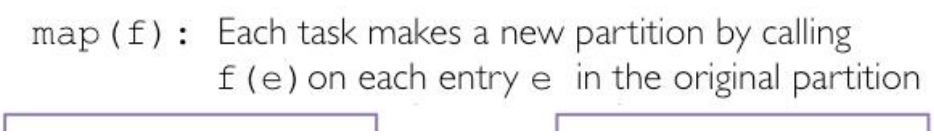
(3c): Subtract one from each value using `map`

So far, we've created a distributed dataset that is split into many partitions, where each partition is stored on a single machine in our cluster. Let's look at what happens when we do a basic operation on the dataset. Many useful data analysis operations can be specified as "do something to each item in the dataset". These data-parallel operations are convenient because each item in the dataset can be processed individually: the operation on one entry doesn't effect the operations on any of the other entries. Therefore, Spark can parallelize the operation.

`map(f)`, the most common Spark transformation, is one such example: it applies a function `f` to each item in the dataset, and outputs the resulting dataset. When you run `map()` on a dataset, a single *stage* of tasks is launched. A *stage* is a group of tasks that all perform the same computation, but on different input data. One task is launched for each partitition, as shown in the example below. A task is a unit of execution that runs on a single machine. When we run `map(f)` within a partition, a new *task* applies `f` to all of the entries in a particular partition, and outputs a new partition. In this example figure, the dataset is broken into four partitions, so four `map()` tasks are launched.



The figure below shows how this would work on the smaller data set from the earlier figures. Note that one task is launched for each partition.



In [22]:
```
# Create sub function to subtract 1
def sub(value):
    """Subtracts one from `value`.

    Args:
       value (int): A number.

    Returns:
        int: `value` minus one.
    """
    return (value - 1)

# Transform xrangeRDD through map transformation using sub function
# Because map is a transformation and Spark uses lazy evaluation, no jobs, stage
# or tasks will be launched when we run this code.
subRDD = xrangeRDD.map(sub)

# Let's see the RDD transformation hierarchy
```

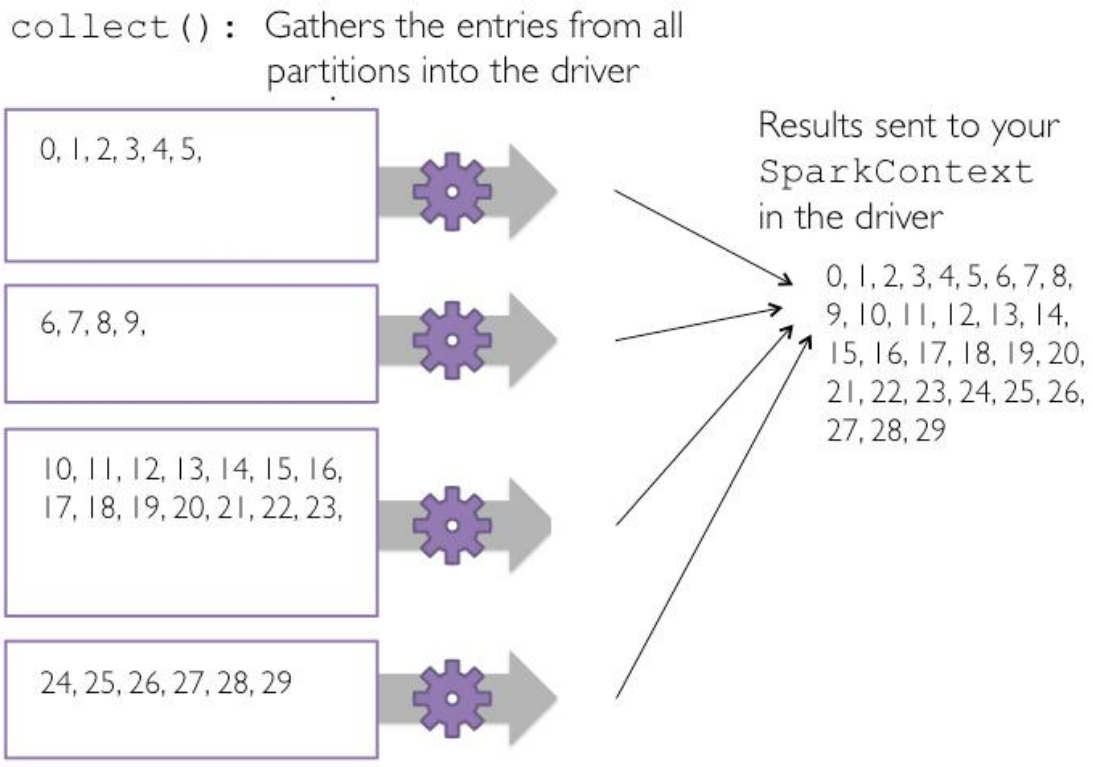
```
(8) PythonRDD[3] at RDD at PythonRDD.scala:43 []
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:392 []
```

(3d) Perform action `collect` to view results

To see a list of elements decremented by one, we need to create a new list on the driver from the the data distributed in the executor nodes. To do this we call the `collect()` method on our RDD. `collect()` is often used after a filter or other operation to ensure that we are only returning a *small* amount of data to the driver. This is done because the data returned to the driver must fit into the driver's available memory. If not, the driver will crash.

The `collect()` method is the first action operation that we have encountered. Action operations cause Spark to perform the (lazy) transformation operations that are required to compute the RDD returned by the action. In our example, this means that tasks will now be launched to perform the `parallelize`, `map`, and `collect` operations.

In this example, the dataset is broken into four partitions, so four `collect()` tasks are launched. Each task collects the entries in its partition and sends the result to the SparkContext, which creates a list of the values, as shown in the figure below.



The above figures showed what would happen if we ran `collect()` on a small example dataset with just four partitions.

Now let's run `collect()` on `subRDD`.

In [23]:
```
# Let's collect the data
```

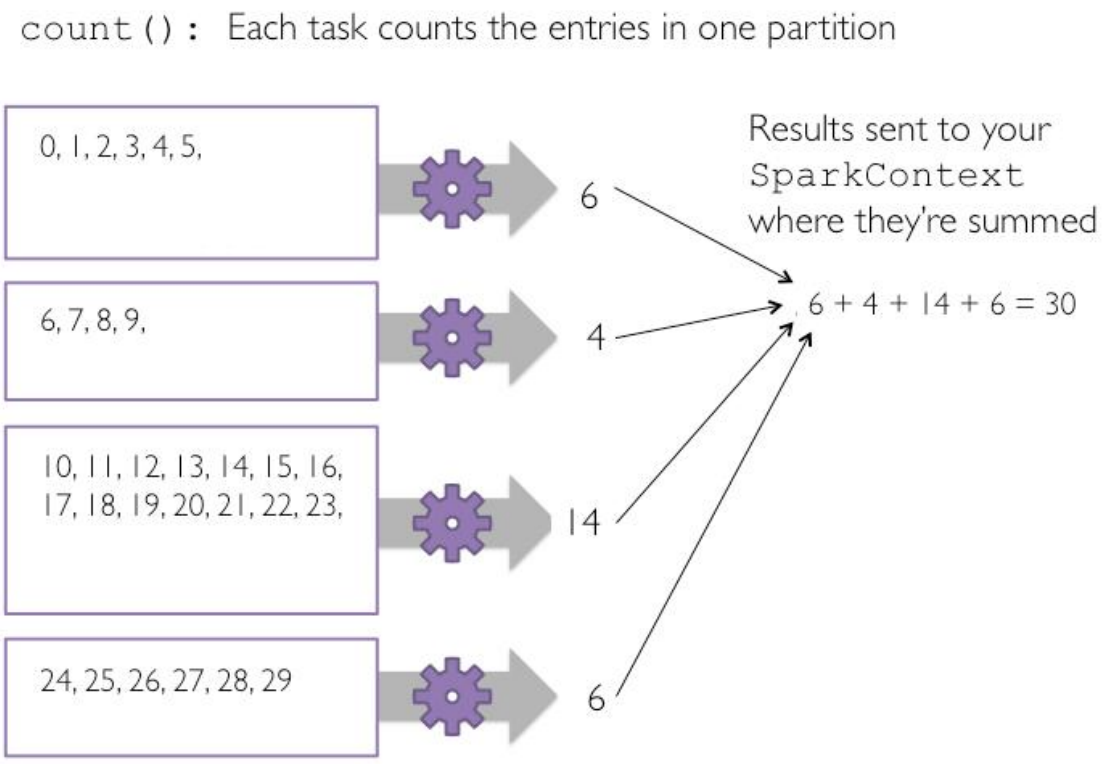
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
1, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 4
0, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 5
9, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 7
8, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 9
7, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,
113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 1
28, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 14
3, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158
, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173,
 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188,
189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 2
04, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 21
9, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234
, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249,
 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264,
265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 2
80, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 29
5, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310
, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325,
 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340,
341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 3
56, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 37
1, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386
, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401,
 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416,
417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 4
32, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 44
7, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462
, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477,
 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492,
493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 5
08, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 52
3, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538
, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553,
 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568,
569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 5
84, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 59
9, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614
, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629,
 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644,
645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 6
60, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 67
5, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690
, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705,
 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720,
721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 7
36, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 75
1, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766
, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781,
 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796,
797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 8
12, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 82
7, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842
, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857,
 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872,
873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 8
88, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 90
3, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918
, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933,
 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948,
949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 9
64, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 97
9, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994
, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007, 1
008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020,
 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029, 1030, 1031, 1032, 103
```

(3d) Perform action `count` to view counts

One of the most basic jobs that we can run is the `count()` job which will count the number of elements in an RDD using the `count()` action. Since `map()` creates a new RDD with the same number of elements as the starting RDD, we expect that applying `count()` to each RDD will return the same result.

Note that because `count()` is an action operation, if we had not already performed an action with `collect()`, then Spark would now perform the transformation operations when we executed `count()`.

Each task counts the entries in its partition and sends the result to your SparkContext, which adds up all of the counts. The figure below shows what would happen if we ran `count()` on a small example dataset with just four partitions.



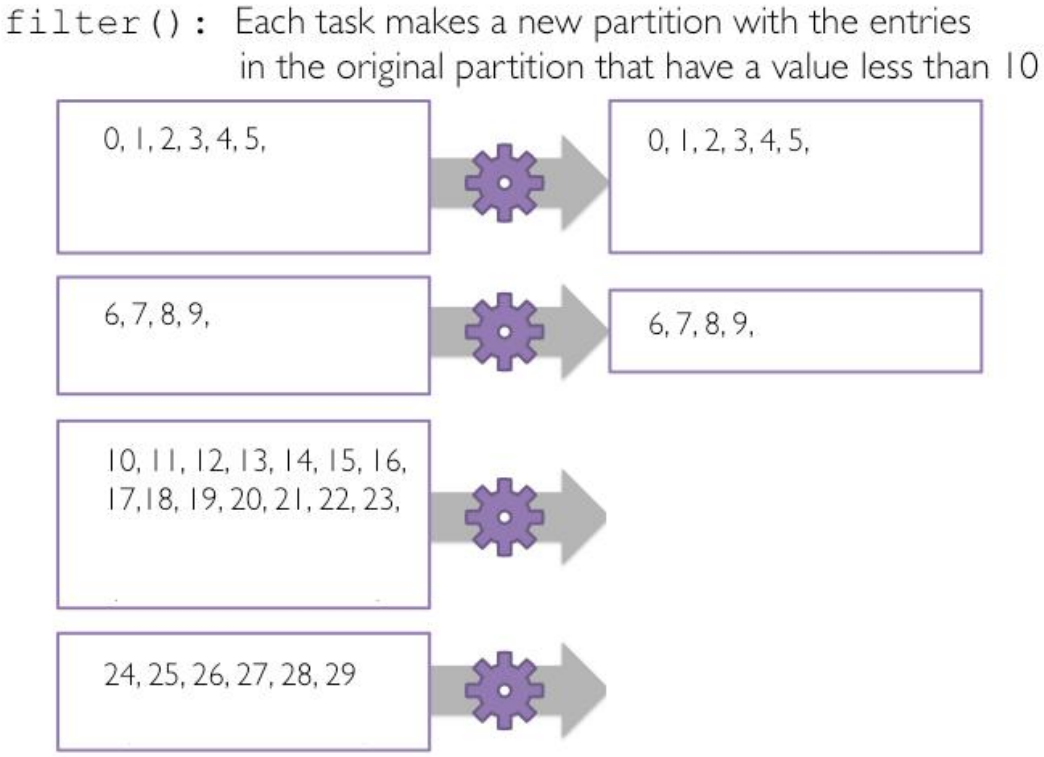
In [24]:
```
print xrangeRDD.count()
```

```
10000
10000
```

(3e) Apply transformation `filter` and view results with `collect`

Next, we'll create a new RDD that only contains the values less than ten by using the `filter(f)` data-parallel operation. The `filter(f)` method is a transformation operation that creates a new RDD from the input RDD by applying filter function `f` to each item in the parent RDD and only passing those elements where the filter function returns `True`. Elements that do not return `True` will be dropped. Like `map()`, filter can be applied individually to each entry in the dataset, so is easily parallelized using Spark.

The figure below shows how this would work on the small four-partition dataset.



To filter this dataset, we'll define a function called `ten()`, which returns `True` if the input is less than 10 and `False` otherwise. This function will be passed to the `filter()` transformation as the filter function `f`.

To view the filtered list of elements less than ten, we need to create a new list on the driver from the distributed data on the executor nodes. We use the `collect()` method to return a list that contains all of the elements in this filtered RDD to the driver program.

In [25]:
```
# Define a function to filter a single value
def ten(value):
    """Return whether value is below ten.

    Args:
        value (int): A number.

    Returns:
        bool: Whether `value` is less than ten.
    """
    if (value < 10):
        return True
    else:
        return False
# The ten function could also be written concisely as: def ten(value): return va

# Pass the function ten to the filter transformation
# Filter is a transformation so no tasks are run
filteredRDD = subRDD.filter(ten)

# View the results using collect()
# Collect is an action and triggers the filter transformation to run
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Part 4: Lambda Functions

(4a) Using Python `lambda()` functions

Python supports the use of small one-line anonymous functions that are not bound to a name at runtime. Borrowed from LISP, these `lambda` functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Remember that `lambda` functions are a matter of style and using them is never required - semantically, they are just syntactic sugar for a normal function definition. You can always define a separate normal function instead, but using a `lambda()` function is an equivalent and more compact form of coding. Ideally you should consider using `lambda` functions where you want to encapsulate non-reusable code without littering your code with one-line functions.

Here, instead of defining a separate function for the `filter()` transformation, we will use an inline `lambda()` function.

In [26]:
```
lambdaRDD = subRDD.filter(lambda x: x < 10)
```

Out[26]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [27]:
```
# Let's collect the even values less than 10
evenRDD = lambdaRDD.filter(lambda x: x % 2 == 0)
```

Out[27]:
```
[0, 2, 4, 6, 8]
```

Part 5: Additional RDD actions

(5a) Other common actions

Let's investigate the additional actions: first() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.first), take() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.take), top() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.top), takeOrdered() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.takeOrdered), and reduce() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.reduce)

One useful thing to do when we have a new dataset is to look at the first few entries to obtain a rough idea of what information is available. In Spark, we can do that using the `first()`, `take()`, `top()`, and `takeOrdered()` actions. Note that for the `first()` and `take()` actions, the elements that are returned depend on how the RDD is *partitioned*.

Instead of using the `collect()` action, we can use the `take(n)` action to return the first n elements of the RDD. The `first()` action returns the first element of an RDD, and is equivalent to `take(1)`.

The `takeOrdered()` action returns the first n elements of the RDD, using either their natural order or a custom comparator. The key advantage of using `takeOrdered()` instead of `first()` or `take()` is that `takeOrdered()` returns a deterministic result, while the other two actions may return differing results, depending on the number of partions or execution environment. `takeOrdered()` returns the list sorted in *ascending order*. The `top()` action is similar to `takeOrdered()` except that it returns the list in *descending order.*

The `reduce()` action reduces the elements of a RDD to a single value by applying a function that takes two parameters and returns a single value. The function should be commutative and associative, as `reduce()` is applied at the partition level and then again to aggregate results from partitions. If these rules don't hold, the results from `reduce()` will be inconsistent. Reducing locally at partitions makes `reduce()` very efficient.

In [28]:
```
# Let's get the first element
print filteredRDD.first()
# The first 4
print filteredRDD.take(4)
# Note that it is ok to take more elements than the RDD has
```

```
0
[0, 1, 2, 3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [29]:
```
# Retrieve the three smallest elements
print filteredRDD.takeOrdered(3)
# Retrieve the five largest elements
```

```
[0, 1, 2]
[9, 8, 7, 6, 5]
```

In [30]:
```
# Pass a lambda function to takeOrdered to reverse the order
```

Out[30]:
```
[9, 8, 7, 6]
```

In [31]:
```
# Obtain Python's add function
from operator import add
# Efficiently sum the RDD using reduce
print filteredRDD.reduce(add)
# Sum using reduce with a lambda function
print filteredRDD.reduce(lambda a, b: a + b)
# Note that subtraction is not both associative and commutative
print filteredRDD.reduce(lambda a, b: a - b)
print filteredRDD.repartition(4).reduce(lambda a, b: a - b)
# While addition is
```

```
45
45
-45
21
45
```

(5b) Advanced actions

Here are two additional actions that are useful for retrieving information from an RDD: takeSample() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.takeSample) and countByValue() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.countByValue)

The `takeSample()` action returns an array with a random sample of elements from the dataset. It takes in a `withReplacement` argument, which specifies whether it is okay to randomly pick the same item multiple times from the parent RDD (so when `withReplacement=True`, you can get the same item back multiple times). It also takes an optional `seed` parameter that allows you to specify a seed value for the random number generator, so that reproducible results can be obtained.

The `countByValue()` action returns the count of each unique value in the RDD as a dictionary that maps values to counts.

In [32]:
```
# takeSample reusing elements
print filteredRDD.takeSample(withReplacement=True, num=6)
# takeSample without reuse
```

```
[0, 7, 8, 5, 9, 0]
[5, 1, 7, 3, 0, 8]
```

In [33]:
```
# Set seed for predictability
print filteredRDD.takeSample(withReplacement=False, num=6, seed=500)
# Try reruning this cell and the cell above -- the results from this cell will
```

```
[0, 2, 5, 3, 6, 9]
```

In [34]:
```
# Create new base RDD to show countByValue
repetitiveRDD = sc.parallelize([1, 2, 3, 1, 2, 3, 1, 2, 1, 2, 3, 3, 3, 4, 5, 4,
```

```
defaultdict(<type 'int'>, {1: 4, 2: 4, 3: 5, 4: 2, 5: 1, 6: 1})
```

Part 6: Additional RDD transformations

(6a) `flatMap`

When performing a `map()` transformation using a function, sometimes the function will return more (or less) than one element. We would like the newly created RDD to consist of the elements outputted by the function. Simply applying a `map()` transformation would yield a new RDD made up of iterators. Each iterator could have zero or more elements. Instead, we often want an RDD consisting of the values contained in those iterators. The solution is to use a flatMap() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.flatMap) transformation, `flatMap()` is similar to `map()`, except that with `flatMap()` each input item can be mapped to zero or more output elements.

To demonstrate `flatMap()`, we will first emit a word along with its plural, and then a range that grows in length with each subsequent operation.

In [35]:
```
# Let's create a new base RDD to work from
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)

# 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()
```

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

In [36]:
```
simpleRDD = sc.parallelize([2, 3, 4])
print simpleRDD.map(lambda x: range(1, x)).collect()
```

```
[[1], [1, 2], [1, 2, 3]]
[1, 1, 2, 1, 2, 3]
```

(6b) `groupByKey` and `reduceByKey`

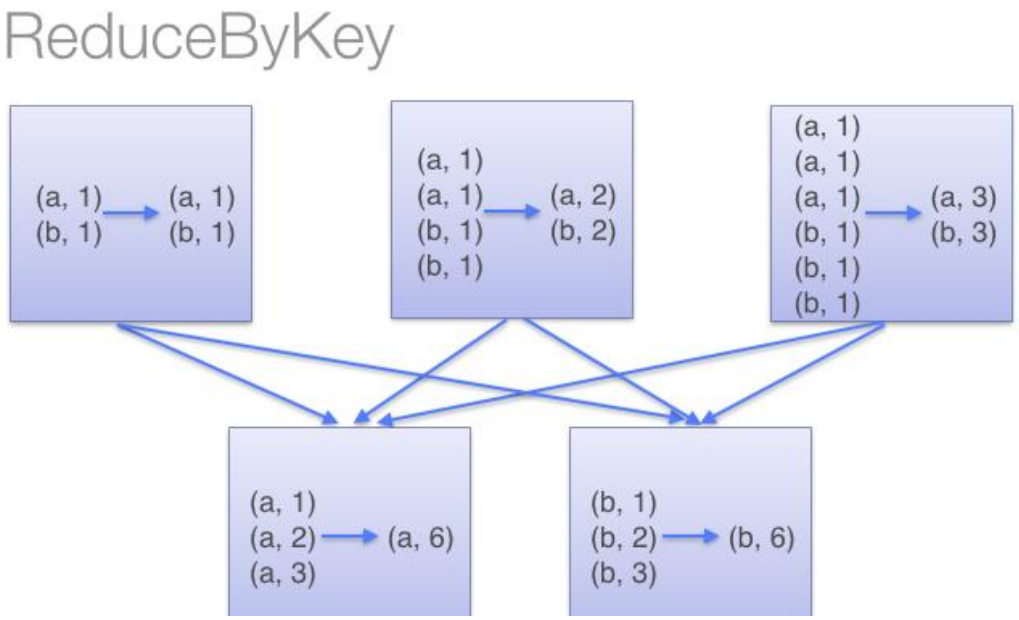
Let's investigate the additional transformations: groupByKey() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.groupByKey) and reduceByKey() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.reduceByKey).

Both of these transformations operate on pair RDDs. A pair RDD is an RDD where each element is a pair tuple (key, value). For example, `sc.parallelize([('a', 1), ('a', 2), ('b', 1)])` would create a pair RDD where the keys are 'a', 'a', 'b' and the values are 1, 2, 1.

The `reduceByKey()` transformation gathers together pairs that have the same key and applies a function to two associated values at a time. `reduceByKey()` operates by applying the function first within each partition on a per-key basis and then across the partitions.

While both the `groupByKey()` and `reduceByKey()` transformations can often be used to solve the same problem and will produce the same answer, the `reduceByKey()` transformation works much better for large distributed datasets. This is because Spark knows it can combine output with a common key on each partition *before* shuffling (redistributing) the data across nodes. Only use `groupByKey()` if the operation would not benefit from reducing the data before the shuffle occurs.

Look at the diagram below to understand how `reduceByKey` works. Notice how pairs on the same machine with the same key are combined (by using the lamdba function passed into reduceByKey) before the data is shuffled. Then the lamdba function is called again to reduce all the values from each partition to produce one final result.



In [37]:
```
pairRDD = sc.parallelize([('a', 1), ('a', 2), ('b', 1)])
# mapValues only used to improve format for printing
print pairRDD.groupByKey().mapValues(lambda x: list(x)).collect()

# Different ways to sum by key
print pairRDD.groupByKey().map(lambda (k, v): (k, sum(v))).collect()
# Using mapValues, which is recommended when they key doesn't change
print pairRDD.groupByKey().mapValues(lambda x: sum(x)).collect()
# reduceByKey is more efficient / scalable
```

```
[('a', [1, 2]), ('b', [1])]
[('a', 3), ('b', 1)]
[('a', 3), ('b', 1)]
[('a', 3), ('b', 1)]
```

(6c) Advanced transformations [Optional]

Let's investigate the advanced transformations: mapPartitions() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.mapPartitions) and mapPartitionsWithIndex() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.mapPartitionsWithIndex)

The `mapPartitions()` transformation uses a function that takes in an iterator (to the items in that specific partition) and returns an iterator. The function is applied on a partition by partition basis.

The `mapPartitionsWithIndex()` transformation uses a function that takes in a partition index (think of this like the partition number) and an iterator (to the items in that specific partition). For every partition (index, iterator) pair, the function returns a tuple of the same partition index number and an iterator of the transformed items in that partition.

In [38]:
```
# mapPartitions takes a function that takes an iterator and returns an iterator
print wordsRDD.collect()
itemsRDD = wordsRDD.mapPartitions(lambda iterator: [','.join(iterator)])
```

```
['cat', 'elephant', 'rat', 'rat', 'cat']
['cat', 'elephant', 'rat', 'rat,cat']
```

In [39]:
```
itemsByPartRDD = wordsRDD.mapPartitionsWithIndex(lambda index, iterator: [(index
# We can see that three of the (partitions) workers have one element and the fou
# elements, although things may not bode well for the rat...
print itemsByPartRDD.collect()
# Rerun without returning a list (acts more like flatMap)
itemsByPartRDD = wordsRDD.mapPartitionsWithIndex(lambda index, iterator: (index,
```

```
[(0, ['cat']), (1, ['elephant']), (2, ['rat']), (3, ['rat', 'cat'])]
[0, ['cat'], 1, ['elephant'], 2, ['rat'], 3, ['rat', 'cat']]
```

Part 7: Caching RDDs and storage options

(7a) Caching RDDs

For efficiency Spark keeps your RDDs in memory. By keeping the contents in memory, Spark can quickly access the data. However, memory is limited, so if you try to keep too many RDDs in memory, Spark will automatically delete RDDs from memory to make space for new RDDs. If you later refer to one of the RDDs, Spark will automatically recreate the RDD for you, but that takes time.

So, if you plan to use an RDD more than once, then you should tell Spark to cache that RDD. You can use the `cache()` operation to keep the RDD in memory. However, if you cache too many RDDs and Spark runs out of memory, it will delete the least recently used (LRU) RDD first. Again, the RDD will be automatically recreated when accessed.

You can check if an RDD is cached by using the `is_cached` attribute, and you can see your cached RDD in the "Storage" section of the Spark web UI. If you click on the RDD's name, you can see more information about where the RDD is stored.

In [44]:
```
# Name the RDD
filteredRDD.setName('My Filtered RDD')
# Cache the RDD
filteredRDD.cache()
# Is it cached
```

```
True
```

(7b) Unpersist and storage options

Spark automatically manages the RDDs cached in memory and will save them to disk if it runs out of memory. For efficiency, once you are finished using an RDD, you can optionally tell Spark to stop caching it in memory by using the RDD's `unpersist()` method to inform Spark that you no longer need the RDD in memory.

You can see the set of transformations that were applied to create an RDD by using the `toDebugString()` method, which will provide storage information, and you can directly query the current storage information for an RDD using the `getStorageLevel()` operation.

Advanced: Spark provides many more options for managing how RDDs are stored in memory or even saved to disk. You can explore the API for RDD's persist() (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.persist) operation using Python's help() (https://docs.python.org/2/library/functions.html?highlight=help#help) command. The `persist()` operation, optionally, takes a pySpark StorageLevel (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.StorageLevel) object.

In [41]:
```
# Note that toDebugString also provides storage information
```

```
(8) My Filtered RDD PythonRDD[6] at collect at <ipython-input-25-2e6525e1a0c
2>:23 [Memory Serialized 1x Replicated]
 |  ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:392 [Memory S
erialized 1x Replicated]
```

In [45]:
```
# If we are done with the RDD we can unpersist it so that its memory can be rec
filteredRDD.unpersist()
# Storage level for a non cached RDD
print filteredRDD.getStorageLevel()
filteredRDD.cache()
# Storage level for a cached RDD
```

```
Serialized 1x Replicated
Memory Serialized 1x Replicated
```

Part 8: Debugging Spark applications and lazy evaluation

How Python is Executed in Spark

Internally, Spark executes using a Java Virtual Machine (JVM). pySpark runs Python code in a JVM using Py4J (http://py4j.sourceforge.net). Py4J enables Python programs running in a Python interpreter to dynamically access Java objects in a Java Virtual Machine. Methods are called as if the Java objects resided in the Python interpreter and Java collections can be accessed through standard Python collection methods. Py4J also enables Java programs to call back Python objects.

Because pySpark uses Py4J, coding errors often result in a complicated, confusing stack trace that can be difficult to understand. In the following section, we'll explore how to understand stack traces.

(8a) Challenges with lazy evaluation using transformations and actions

Spark's use of lazy evaluation can make debugging more difficult because code is not always executed immediately. To see an example of how this can happen, let's first define a broken filter function.

Next we perform a `filter()` operation using the broken filtering function. No error will occur at this point due to Spark's use of lazy evaluation.

The `filter()` method will not be executed *until* an action operation is invoked on the RDD. We will perform an action by using the `collect()` method to return a list that contains all of the elements in this RDD.

In [46]:
```
def brokenTen(value):
    """Incorrect implementation of the ten function.

    Note:
        The `if` statement checks an undefined variable `val` instead of `value`

    Args:
        value (int): A number.

    Returns:
        bool: Whether `value` is less than ten.

    Raises:
        NameError: The function references `val`, which is not available in the
            namespace, so a `NameError` is raised.
    """
    if (val < 10):
        return True
    else:
        return False
```

In [47]:
```
# Now we'll see the error
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
<ipython-input-47-c7025769b408> in <module>()
      1 # Now we'll see the error
----> 2 brokenRDD.collect()

/usr/local/bin/spark-1.3.1-bin-hadoop2.6/python/pyspark/rdd.py in collect(se
lf)
    711         """
    712         with SCCallSiteSync(self.context) as css:
--> 713             port = self.ctx._jvm.PythonRDD.collectAndServe(self._jrd
d.rdd())
    714         return list(_load_from_socket(port, self._jrdd_deserializer)
)
    715

/usr/local/bin/spark-1.3.1-bin-hadoop2.6/python/lib/py4j-0.8.2.1-src.zip/py4
j/java_gateway.py in __call__(self, *args)
    536         answer = self.gateway_client.send_command(command)
    537         return_value = get_return_value(answer, self.gateway_client,
--> 538                 self.target_id, self.name)
    539
    540         for temp_arg in temp_args:

/usr/local/bin/spark-1.3.1-bin-hadoop2.6/python/lib/py4j-0.8.2.1-src.zip/py4
j/protocol.py in get_return_value(answer, gateway_client, target_id, name)
    298                 raise Py4JJavaError(
    299                     'An error occurred while calling {0}{1}{2}.\n'.
--> 300                     format(target_id, '.', name), value)
    301             else:
    302                 raise Py4JError(

Py4JJavaError: An error occurred while calling z:org.apache.spark.api.python
.PythonRDD.collectAndServe.
: org.apache.spark.SparkException: Job aborted due to stage failure: Task 0
in stage 48.0 failed 1 times, most recent failure: Lost task 0.0 in stage 48
.0 (TID 221, localhost): org.apache.spark.api.python.PythonException: Traceb
ack (most recent call last):
  File "/usr/local/bin/spark-1.3.1-bin-hadoop2.6/python/pyspark/worker.py",
line 101, in main
    process()
  File "/usr/local/bin/spark-1.3.1-bin-hadoop2.6/python/pyspark/worker.py",
line 96, in process
    serializer.dump_stream(func(split_index, iterator), outfile)
  File "/usr/local/bin/spark-1.3.1-bin-hadoop2.6/python/pyspark/serializers.
py", line 236, in dump_stream
    vs = list(itertools.islice(iterator, batch))
  File "<ipython-input-46-775c9f3488e3>", line 17, in brokenTen
NameError: global name 'val' is not defined

        at org.apache.spark.api.python.PythonRDD$$anon$1.read(PythonRDD.scal
a:135)
        at org.apache.spark.api.python.PythonRDD$$anon$1.<init>(PythonRDD.sc
ala:176)
        at org.apache.spark.api.python.PythonRDD.compute(PythonRDD.scala:94)
        at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:277)
        at org.apache.spark.rdd.RDD.iterator(RDD.scala:244)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:61
)
        at org.apache.spark.scheduler.Task.run(Task.scala:64)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:
203)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecu
tor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExec
utor.java:615)
        at java.lang.Thread.run(Thread.java:745)
```

(8b) Finding the bug

When the `filter()` method is executed, Spark evaluates the RDD by executing the `parallelize()` and `filter()` methods. Since our `filter()` method has an error in the filtering function `brokenTen()`, an error occurs.

Scroll through the output "Py4JJavaError Traceback (most recent call last)" part of the cell and first you will see that the line that generated the error is the `collect()` method line. There is *nothing wrong with this line*. However, it is an action and that caused other methods to be executed. Continue scrolling through the Traceback and you will see the following error line:

```
NameError: global name 'val' is not defined
```

Looking at this error line, we can see that we used the wrong variable name in our filtering function `brokenTen()`.

(8c) Moving toward expert style

As you are learning Spark, I recommend that you write your code in the form:

```
RDD.transformation1()
RDD.action1()
RDD.transformation2()
RDD.action2()
```

Using this style will make debugging your code much easier as it makes errors easier to localize - errors in your transformations will occur when the next action is executed.

Once you become more experienced with Spark, you can write your code with the form:

```
RDD.transformation1().transformation2().action()
```

We can also use `lambda()` functions instead of separately defined functions when their use improves readability and conciseness.

In [48]:
```
# Cleaner code through lambda use
```

Out[48]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [49]:
```
# Even better by moving our chain of operators into a single line.
```

Out[49]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

(8d) Readability and code style

To make the expert coding style more readable, enclose the statement in parentheses and put each method, transformation, or action on a separate line.

In [50]:
```
# Final version
(sc
 .parallelize(data)
 .map(lambda y: y - 1)
 .filter(lambda x: x < 10)
```

Out[50]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [ ]:

In [ ]:

In [ ]: