Spark Basics 2

Lazy evaluation

We can combine map and reduce operations to perform more complex operations.

Suppose we want to compute the sum of the squares [Math Processing Error]
where the elements [Math Processing Error]
are stored in an RDD.

Create an RDD

In [2]:
```
B=sc.parallelize(range(4))
B.collect()
```

Out[2]:
```
[0, 1, 2, 3]
```

Sequential syntax

Perform assignment after each computation

In [3]:

```
Squares=B.map(lambda x:x*x)
Squares.reduce(lambda x,y:x+y)
```

Out[3]:

```
14
```

Cascaded syntax

Combine computations into a single cascaded command

In [4]:
```
B.map(lambda x:x*x)\
   .reduce(lambda x,y:x+y)
```

Out[4]:
```
14
```

Both syntaxes mean the same thing

The only difference:

- In the sequential syntax the intermediate RDD has a name `Squares`
- In the cascaded syntax the intermediate RDD is *anonymous*

The execution is identical!

Sequential execution

The standard way that the map and reduce are executed is

- perform the map
- store the resulting RDD in memory
- perform the reduce

Disadvantages of Sequential execution

1. Intermediate result (Squares) requires memory space.
2. Two scans of memory (of B, then of Squares) - double the cache-misses.

Pipelined execution

Perform the whole computation in a single pass. For each element of B

1. Compute the square
2. Enter the square as input to the reduce operation.

Advantages of Pipelined execution

1. Less memory required - intermediate result is not stored.
2. Faster - only one pass through the Input RDD.

Lazy Evaluation

This type of pipelined evaluation is related to **Lazy Evaluation**. The word **Lazy** is used because the first command (computing the square) is not executed immediately. Instead, the execution is delayed as long as possible so that several commands are executed in a single pass.

The delayed commands are organized in an **Execution plan**

An instructive mistake

Here is another way to compute the sum of the squares using a single reduce command. What is wrong with it?

In [5]:
```
C=sc.parallelize([1,1,1])
C.reduce(lambda x,y: x*x+y*y)
```

Out[5]:
```
5
```

1 1 1

getting information about an RDD

RDD's typically have hundreds of thousands of elements. It usually makes no sense to print out the content of a whole RDD. Here are some ways to get manageable amounts of information about an RDD

In [6]:
```
n=1000000
B=sc.parallelize([0,0,1,0]*(n/4))
```

In [7]:
```
#find the number of elements in the RDD
B.count()
```
Out[7]:
```
1000000
```

In [8]:

```python
# get the first few elements of an RDD
print 'first element=',B.first()
print 'first 5 elements = ',B.take(5)
```

```
first element= 0
first 5 elements =  [0, 0, 1, 0, 0]
```

Sampling an RDD

- RDDs are often very large.
- Aggregates, such as averages, can be approximated efficiently by using a sample.
- Sampling is done in parallel and it keeps the data local.

In [9]:

```
# get a sample whose expected size is m
m=5.
B.sample(False,m/n).collect()
```

Out[9]:

```
[1, 0, 1, 0, 0, 0]
```

filtering an RDD

The method RDD.filter(func) Return a new dataset formed by selecting those elements of the source on which func returns true.

In [10]:

```
# How many positive numbers?
B.filter(lambda n: n > 0).count()
```

Out[10]:

```
250000
```

Removing duplicate elements from an RDD

The method RDD.distinct(numPartitions=None) Returns a new dataset that contains the distinct elements of the source dataset

- The number of partitions is specified through the **numPartitions** argument. Each of this partitions is potentially on different machine.

In [11]:

```
# Remove duplicate element in DuplicateRDD, we get distinct RDD
DuplicateRDD = sc.parallelize([1,1,2,2,3,3])
DistinctRDD = DuplicateRDD.distinct()
DistinctRDD.collect()
```

Out[11]:

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

flatmap an RDD

The method RDD.flatMap(func) is similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).

In [12]:

```
text=["you are my sunshine","my only sunshine"]
text_file = sc.parallelize(text)
# map each line in text to a list of words
print 'map:',text_file.map(lambda line: line.split(" ")).collect()
# create a single list of words by combining the words from all of the lines
print 'flatmap:',text_file.flatMap(lambda line: line.split(" ")).collect()
```

```
map: [['you', 'are', 'my', 'sunshine'], ['my', 'only', 'sunshine']]
flatmap: ['you', 'are', 'my', 'sunshine', 'my', 'only', 'sunshine']
```

Set operations

In this part, we explore set operations including **union**,**intersection**,**subtract**, **cartesian** in pyspark

In [13]:

```
rdd1 = sc.parallelize([1, 1, 2, 3])
rdd2 = sc.parallelize([1, 3, 4, 5])
```

1. union(other)
    - Return the union of this RDD and another one.

In [14]:
```
rdd1.union(rdd2).collect()
```

Out[14]:
```
[1, 1, 2, 3, 1, 3, 4, 5]
```

1. intersection(other)
    - Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.Note that this method performs a shuffle internally.

In [15]:
```
rdd1.intersection(rdd2).collect()
```

Out[15]:
```
[1, 3]
```

1. subtract(other, numPartitions=None)
    - Return each value in self that is not contained in other.

In [16]:

```
rdd1.subtract(rdd2).collect()
```

Out[16]:

```
[2]
```

1. cartesian(other)
    - Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where **a** is in **self** and **b** is in **other**.

In [17]:

```
print rdd1.cartesian(rdd2).collect()
```

```
[(1, 1), (1, 3), (1, 4), (1, 5), (1, 1), (1, 3), (1, 4), (1, 5), (2, 1), (2, 3), (2, 4), (2, 5), (3, 1), (3, 3), (3, 4), (3, 5)]
```