Types of spark operations

There are Three types of operations on RDDs: Transformations, Actions and Shuffles.

- The most expensive operations are those the require communication between nodes.
Transformations: RDD $\rightarrow$ RDD.

- Examples map, filter, sample, More
- No communication needed.
Actions: RDD → Python-object in head node.

- **Examples:** reduce, collect, count, take, [More](#)
- **Some** communication needed.
Shuffles: RDD → RDD, shuffle needed

- **Examples**: repartition, sortByKey, reduceByKey, join More
- **A LOT** of communication needed.
Key/value pairs

- A python dictionary is a collection of *key/value* pairs.
- The **key** is used to find a set of pairs with the particular key.
- The **value** can be anything.
- Spark has a set of special operations for *(key,value)* RDDs.
Creating (key,value) RDDs

Method 1: parallelize a list of pairs.

In [2]:
```python
pair_rdd = sc.parallelize([(1,2), (3,4)])
print(pair_rdd.collect())
```
```
[(1, 2), (3, 4)]
```
**Method 2:** `map()` a function that returns a key/value pair.

```python
In [3]:
regular_rdd = sc.parallelize([1, 2, 3, 4, 2, 5, 6])
pair_rdd = regular_rdd.map(lambda x: (x, x**2))
print(pair_rdd.collect())
```

```
[(1, 1), (2, 4), (3, 9), (4, 16), (2, 4), (5, 25), (6, 36)]
```
Some important Key-Value Transformations

1. reduceByKey(func): Apply the reduce function on the values with the same key.

```python
In [8]:
rdd = sc.parallelize([(1,2), (2,4), (2,6)])
print "Original RDD :", rdd.collect()
print "After transformation : ", rdd.reduceByKey(lambda a, b: a+b).collect()

Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation : [(1, 2), (2, 10)]
```

Note that although it is similar to the reduce function, it is implemented as a transformation and not as an action because the dataset can have very large number of keys. So, it does not return values to the driver program. Instead, it returns a new RDD.
2. sortByKey():

Sort RDD by keys in ascending order.

```
In [9]:

rdd = sc.parallelize([(2,2), (1,4), (3,6)])
print "Original RDD :", rdd.collect()
print "After transformation : ", rdd.sortByKey().collect()
```

Original RDD : [(2, 2), (1, 4), (3, 6)]
After transformation : [(1, 4), (2, 2), (3, 6)]

**Note:** The output of sortByKey() is an RDD. This means that RDDs do have a meaningful order, which extends between partitions.
3. **mapValues(func):**

Apply func to each value of RDD without changing the key.

```
In [10]:
rdd = sc.parallelize([(1,2), (2,4), (2,6)])
print "Original RDD :", rdd.collect()
print "After transformation :", rdd.mapValues(lambda x: x*2).collect()
```

Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation : [(1, 4), (2, 8), (2, 12)]
4. `groupByKey()`:

Returns a new RDD of (key,<iterator>) pairs where the iterator iterates over the values associated with the key.

**Iterators** are python objects that generate a sequence of values. Writing a loop over n elements as

```python
for i in range(n):
    ##do something
```

is inefficient because it first allocates a list of n elements and then iterates over it. Using the iterator `xrange(n)` achieves the same result without materializing the list. Instead, elements are generated on the fly.

To materialize the list of values returned by an iterator we will use the list comprehension command:

```python
[a for a in <iterator>]
```

```python
In [11]:
rdd = sc.parallelize([(1,2), (2,4), (2,6)])
print "Original RDD:" rdd.collect()
```
5. `flatMapValues(func)`:

`func` is a function that takes as input a single value and returns an iterator that generates a sequence of values. The application of `flatMapValues` operates on a key/value RDD. It applies `func` to each value, and gets an list (generated by the iterator) of values. It then combines each of the values with the original key to produce a list of key-value pairs. These lists are concatenated as in `flatMap`

```python
In [14]:
rdd = sc.parallelize([(1,2), (2,4), (2,6)])
print "Original RDD :", rdd.collect()
# the lambda function generates for each number i, an iterator that produces i,i+1
print "After transformation : ", rdd.flatMapValues(lambda x: xrange(x,x+2)).collect()

Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation : [(1, 2), (1, 3), (2, 4), (2, 5), (2, 6), (2, 7)]
```
Transformations on two Pair RDDs

In [16]:

```
rdd1 = sc.parallelize([(1,2),(2,1),(2,2)])
rdd2 = sc.parallelize([(2,5),(3,1)])
a = rdd1.collect()
b = rdd2.collect()
print a,b

[(1, 2), (2, 1), (2, 2)] [(2, 5), (3, 1)]
```
1. `subtractByKey`:

Remove from RDD1 all elements whose key is present in RDD2.

In [17]:

```python
print "RDD1:", a
print "RDD2:", b
print "Result:", rdd1.subtractByKey(rdd2).collect()
```

RDD1: [(1, 2), (2, 1), (2, 2)]
RDD2: [(2, 5), (3, 1)]
Result: [(1, 2)]
2. join:

- A fundamental operation in relational databases.
- Assumes two tables have a **key** column in common.
- Merges rows with the same key.

Suppose we have two (key, value) datasets

<table>
<thead>
<tr>
<th>dataset 1</th>
<th></th>
<th>dataset 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>key=name</td>
<td>(gender, occupation, age)</td>
<td>key=name</td>
</tr>
<tr>
<td>John</td>
<td>(male, cook, 21)</td>
<td>Jill</td>
</tr>
<tr>
<td>Jill</td>
<td>(female, programmer, 19)</td>
<td>Grace</td>
</tr>
<tr>
<td>John</td>
<td>(male, kid, 2)</td>
<td>John</td>
</tr>
<tr>
<td>Kate</td>
<td>(female, wrestler, 54)</td>
<td></td>
</tr>
</tbody>
</table>
When `Join` is called on datasets of type `(Key, V)` and `(Key, W)`, it returns a dataset of `(Key, (V, W))` pairs with all pairs of elements for each key. Joining the 2 datasets above yields:

<table>
<thead>
<tr>
<th>key = name</th>
<th>(gender,occupation,age), haircolor</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>((male, cook, 21), black)</td>
</tr>
<tr>
<td>John</td>
<td>((male, kid, 2), black)</td>
</tr>
<tr>
<td>Jill</td>
<td>((female, programmer, 19), blond)</td>
</tr>
</tbody>
</table>
In [18]:

```python
print "RDD1:" , a
print "RDD2:" , b
print "Result:" , rdd1.join(rdd2).collect()
```

RDD1: [(1, 2), (2, 1), (2, 2)]
RDD2: [(2, 5), (3, 1)]
Result: [(2, (1, 5)), (2, (2, 5))]
### Actions on Pair RDDs

```python
In [21]:
rdd = sc.parallelize([(1,2), (2,4), (2,6)])
a = rdd.collect()
```
1. `countByKey()`: Count the number of elements for each key. Returns a dictionary for easy access to keys.

In [22]:

```python
print "RDD: ", a
result = rdd.countByKey()
print "Result: ", result
```

RDD: [(1, 2), (2, 4), (2, 6)]
Result: defaultdict(<type 'int'>, {1: 1, 2: 2})
2. `collectAsMap()`:

Collect the result as a dictionary to provide easy lookup.

```
In [23]:
    print "RDD: ", a
    result = rdd.collectAsMap()
    print "Result: ", result

RDD: [(1, 2), (2, 4), (2, 6)]
Result: {1: 2, 2: 6}
```
3. lookup(key):

Return all values associated with the provided key.

In [24]:
   print "RDD: ", 
   result = rdd.lookup(2)
   print "Result: ", result

RDD: [(1, 2), (2, 4), (2, 6)]
Result: [4, 6]