Since I started using Spark, shuffles and joins have become the bane of my life. We frequently host group therapy sessions at my company where we share Spark stories, and try to answer questions like: when shall I stop waiting for a task (that might never end) and unplug the chord on it? Is there a life after I lose an executor?

Recently I have been doing some joins that were taking too long, and every single time it would finish the vast majority of the tasks rapidly but spend most of the time (like 99% of the time) finishing the last few. If you ever experienced this, then what you have is a badly skewed join.

This is briefly explained in the Databricks talk in https://www.youtube.com/watch?t=3215&v=HG2Yd-3r4-M. The talk only explains how to diagnose the problem, but does not put forward any way of fixing it, apart from “picking a different algorithm” or “restructuring the data”. So last time I had this problem, I came up with a solution that might or might not be the right thing to do for every case, but it did the job for me.

For the purpose of demonstration, we will create a skewed rdd and a smaller non-skewed rdd and do the join between them. I start by creating the large skewed rdd in a similar manner to the Databricks talk by doing:

```
from math import exp
from random import randint
from datetime import datetime

def count_elements(splitIndex, iterator):
    n = sum(1 for _ in iterator)
    yield (splitIndex, n)

def get_part_index(splitIndex, iterator):
    for it in iterator:
        yield (splitIndex, it)

num_parts = 16
# create the large skewed rdd
skewed_large_rdd = sc.parallelize(range(0,num_par

print "skewed_large_rdd has %d partitions."%skewe
print "The distribution of elements across partit
%str(skewed_large_rdd.mapPartitionsWithIndex(lamb

# put it in (key, value) form
skewed_large_rdd = skewed_large_rdd.mapPartitions
skewed_large_rdd.count()
```

```
skewed_large_rdd has 17 partitions.
The distribution of elements across partitions is: [(0, 1), (1, 2), (2, 7), (3, 20), (4
```

The rdd is made out of tuples (num_partition, number) where num_partition is the index of the partition and number is a number between 0 and exp(num_partition). This creates an rdd with an artificial skew, where each partition has an exponentially large number of items in it. So partition 0 has 1 item, partition 1 has floor(exp(1))=2 items, partition 2 has floor(exp(2))=7 items and so on.

We create a second dummy dataset which has keys 1 to num_parts and the value is ‘b’.

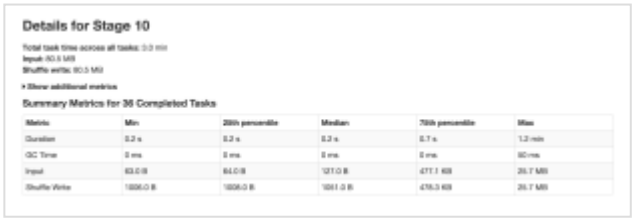
```
small_rdd = sc.parallelize(range(0,num_parts), num
small_rdd.count()
```

Both rdds are cached in memory, now we go and do what we came for: the join.

```
t0 = datetime.now()
result = skewed_large_rdd.leftOuterJoin(small_rdd)
result.count()
print "The direct join takes %s"%(str(datetime.now
```

```
The direct join takes 0:01:46.539335
```

In this particular example, it took ~20 seconds to do 31 tasks out of 32 and the remaining time to finish off the last one. This is a symptom of a skew. We can diagnose the skew by looking at the Spark webUI and checking the time taken per task. Some tasks will take very little time, while others will straggle behind for a significantly longer time.



One quick fix for a skewed join is to simply drop the largest items in the rdd, which may contain outliers. However, let's imagine we don't want to throw away data. In that case, I came up with a semi-quick fix that deals with the skew at the expense of having some data replication. And it works like this:

- We replicate the data in the small rdd N times by creating a new key (original_key, v) where v takes values between 0 and N. The value does not change, i.e. it is the same value that was associated to the original key.
- We take the large skewed rdd and modify the key to add some randomness by doing (original_key, random_int) where random_int takes a value between 0 and N. Note that in this case we are NOT replicating the data in the large rdd. We are simply splitting the keys so that values associated to the same original key are now split into N buckets.
- Finally, we perform the join between these datasets.
- We remove the random_int from the key to have the final result of the join.

```
1  N = 100 # parameter to control level of data repl
2  small_rdd_transformed = small_rdd.cartesian(sc.pa
3  small_rdd_transformed.count()
4  skewed_large_rdd_transformed = skewed_large_rdd.m
5  skewed_large_rdd_transformed.count()
6
7  t0 = datetime.now()
8  result = skewed_large_rdd_transformed.leftOuterJo
9  result.count()
10 print "The hashed join takes %s"%(str(datetime.no
```

```
The hashed join takes 0:00:04.893182
```

So as you can see, using this trick, we have decreased the time necessary for the join from 1 mins 50s to 5 seconds!