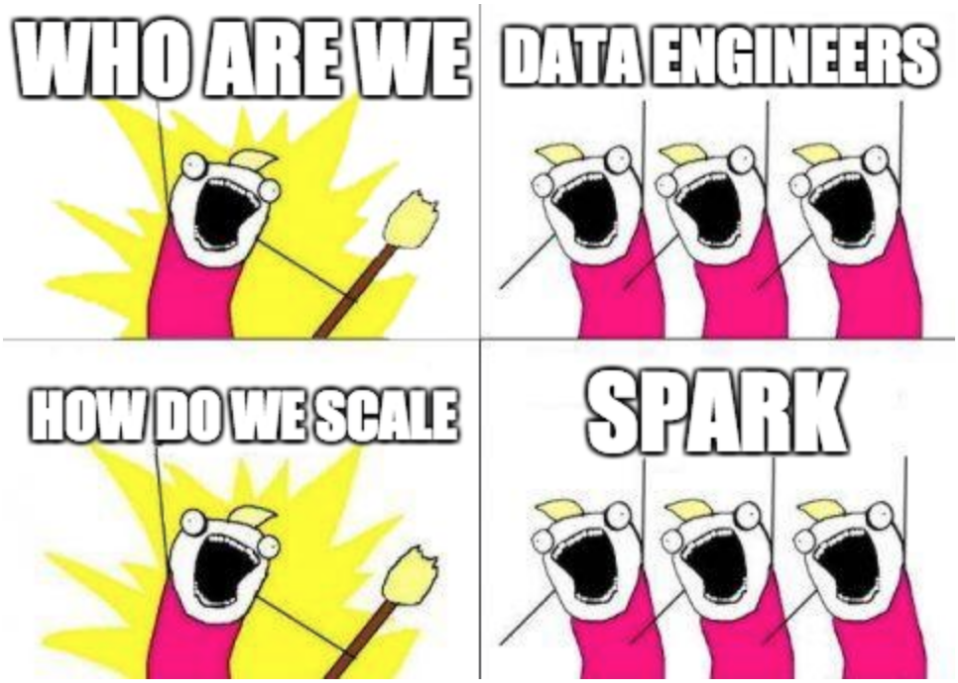
Spark Protip: Joining on skewed dataframes

At Simpl, most of our data and ML pipelines are highly dependant on Spark. And we all love Spark here. We really do! To an extent that all our data processing codes are now spark pipeline transformers.

Spark—Answer to all big data processing problems

With that out of the way, we readily admit that making the shift wasn't easy. Moving from Pandas to Spark for scalable data wrangling and processing had its learning curve. Like many others, we started with assumption that by just using Spark, we could solve for most of our scalability issues.



Spark—Answer to all data scalability problems

However, we quickly realised that a deep knowledge of the functions we were using was essential if we wanted to use Spark and scale it.

This post is the first of many where we will delve into interesting findings and a-ha moments as we learned to work with Spark. Many of these lessons are things we wish we knew when we started out. So let's get started.

Joining on skewed datasets in spark

While using Spark for our pipelines, there was a use-case where we had to join 2 data-frames, one of which was highly skewed on the join column and the other was an evenly distributed data-frame. Both of these dataframes were fairly large (millions of records. And the job was getting stuck at the last stage (say at 199/200 steps) and stayed that way . On the Spark Web App UI, we saw this:

	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	S
+details	2018/02/18 15:12:11	47 h	199/200			

PySpark Job stuck at last stage—For illustration purposes

Initially, we thought that it just takes a long time to merge huge datasets. But 47 hours later, there had to be something wrong in what we were doing. Eventually, with a fair bit of research and brainstorming(and hair-pulling), we were finally able to get to the bottom of it.

So why is this happening?

Let's say that we have asked Spark to join two dataframes—A and B. When Spark gets this instruction with its default configuration, it performs the default Shuffle Hash Join. In this process, Spark hashes the join column and sorts it. And then it tries to keep the records with same hashes in both partitions on the same executor.

Highly skewed dataframes will pose problems with the above approach. A skewed dataframe looks something like this.

```
+---+----+
| id|city|
+---+----+
|  1|  c1|
|  2|  c1|
|  3|  c1|
|  4|  c2|
|  5|  c1|
|  6|  c1|
|  7|  c2|
|  8|  c1|
|  9|  c1|
| 10|  c3|
+---+----+
```

In case one of these dataframes is highly skewed i.e. a particular value in the column appears a lot more as compared to others, it becomes impossible for Spark to fit all the records with the matching hash on a single partition. Therefore, Spark gets into a continuous loop of shuffling and garbage collection with no success.

How do you fix this?

There are multiple ways to optimise for this. We will list all of it below. Most of these ways are nothing but following standard Spark good practices.

Copy the smaller of the 2 datasets in all nodes of the spark cluster

Basically, this means that Spark will now not need to do shuffle across the network and can refer to the local copy of the file. Fortunately, to achieve this, we just need to add a couple of lines of code.

```
from pyspark.sql.functions import broadcast

result = broadcast(A).join(B,["join_col"],"left")
```

The above assumes that A is the smaller dataframe and can fit entirely into each of the executors.

Use repartitions gracefully

As a standard best practice in Spark, it's always recommended to repartition your data with optimal repartition factor right before a huge operation.

Cache dataframes and use take(1) to execute the cache

Caching the dataframes before heavy operations also optimises performance to great extent. You can use take(1) to make sure that the cache gets executed.

In upcoming posts, we will bring a few more interesting findings and ways to solve for it.