1. **Broadcast variable and accumulator**

**Broadcast variable:**

Broadcast variables are pretty simple in concept. They're variables that we want to share throughout our cluster. However there are a couple of caveats that are important to understand. Broadcast variables have to be able to fit in memory on one machine. That means that they definitely should NOT be anything super large, like a large table or massive vector. Secondly, broadcast variables are immutable, meaning that they cannot be changed later on. This may seem inconvenient but it truly suits their use case. If you need something that can change, I'd certainly point you to accumulators which will be covered in another post. So now

**we know that broadcast variables are:**

1. Immutable
2. Distributed to the cluster
3. Fit in memory

val hoods = Seq((1, "Mission"), (2, "SOMA"), (3, "Sunset"), (4, "Haight Ashbury"))

val checkins = Seq((234, 1),(567, 2), (234, 3), (532, 2), (234, 4))

val hoodsRdd = sc.parallelize(hoods)

val checkRdd = sc.parallelize(checkins)

Now that we've set those up, we need to broadcast the first table.

val broadcastedHoods = sc.broadcast(hoodsRdd.collectAsMap())

Now that that's our there across our cluster, let's go ahead and join the two!

val checkinsWithHoods = checkRdd.mapPartitions({row =>

row.map(x => (x.\_1, x.\_2, broadcastedHoods.value.getOrElse(x.\_2, -1)))

}, preservesPartitioning = true)

checkinsWithHoods.take(5)

// res3: Array[(Int, Int, Any)] = // Array((234,1,Mission), (567,2,SOMA), (234,3,Sunset), (532,2,SOMA), (234,4,Haight Ashbury))

You may have noticed that whole "preserve partitioning argument and that's to prevent the shuffle of data!

preservesPartitioning indicates whether the input function preserves the partitioner, which should be false unless this is a pair RDD and the input function doesn't modify the keys.

**Accumulator:**

As you might assume from the name, Accumulators are variables which may be added to through associated operations. There are many uses for accumulators including implementing counters or sums. Spark supports the accumulation of numeric types easily, but programmers can add support for other types. If there is a particular name for an accumulator in code, it is usually displayed in the Spark UI, which will be useful in understanding the running stage progress.

Accumulators are created from an initial value v; i.e. SparkContext.accumulator(v). Then the tasks running in the cluster can be added to it using the known “add method” or += operator in Scala. They cannot, however, read the value of it. The driver program has the ability to read the value of the accumulator, using the value method as shown below

scala> val accum = sc.accumulator(0, "Accumulator Example")

accum: spark.Accumulator[Int] = 0

scala> sc.parallelize(Array(1, 2, 3)).foreach(x => accum += x)

scala> accum.value

res4: Int = 6