Data-Parallel to Distributed Data-Parallel

- *Shared Memory*: data-parallel programming model; data partitioned in memory and operated upon in parallel.
- *Distributed*: same programming model; although, data is partitioned between machines, network in between, operated upon in parallel.

Distribution

- Partial Failure: crash failures of a subset of the machines involved in a dstributed computation;
- Latency: certain operations have a much higher latency than other operations due to network communication.

  network roundtrips are expensive, you typically want to reduce the amout of network communication that your job cause.

Important Latency Numbers

It's 1.000.000x slower Sending a packet round trip over a long distance than reference something that exists in main memory.

Memory > Disk > Network

Why Spark over Hadoop?

- Fault-tolerance in Hadoop/MapReduce comes at a cost; between each map and reduce step, in order to recover from potential failures, Hadoop shuffles its data and write intermediate data to disk.
- Spark keep all data immutable and in-memory; all operations on data are just functional transformations; faul tolerance is achieved by replaying functional transformations over original dataset.

Hadoop = Disk + Network; Spark = Memory + Network; => 100x faster.

Spark <3 Data Science * Lazy Transformations; * Eager Actions that kick off staged transformations; * In-memory computations = lower latencies; * No need to read/write to disk!

Resilient Distributed Datasets (data parallel model)

A nice paper on RDD's: https://cs.stanford.edu/~matei/papers/2012/nsdi_spark.pdf > a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

Using RDDs in Spark feels a lot like normal Scala sequential/parallel collections, with the added knowledge that your data is distributed accross several machines.

```
abstract class RDD[T] {
  def map[U](f: T => U): RDD[U] = ...
  def flatMap[U](f: T => TraversableOnce[U]): RDD[U] = ...
}
```

```
// count the words in a large file
val count = spark.textFile("hdfs://...")
                 .flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
```

There is two ways of creating RDDs:

- Transforming an existing RDD ~ just like a call to map on a List returns a new List, many higher-order functions defined on RDD return a new RDD.
- From a SparkContext (represents the connection between the Spark cluster and the runnin application) either with `parallelize` (convert a local Scala collection to an RDD) or with `textFile` (read a file from HDFS and return an RDD of String)

Transformations and Actions

- Transformations ~ Return new RDDs as results; (They are *lazy*, their result RDD is not immediately computed.
- Actions ~ Compute a result based on an RDD, and either returned or saved to an external storage system. (They are *eager*, their result is immediately computed.

By relying on these lazy transformations, Spark aggressively reduce the amout of network communications.

```
1. val largeList: List[String] = ...
2. val wordsRdd = sparkCtx.parallelize(largeList)
3. val lengthsRdd = wordsRdd.map(_.length)
4. val totalChars = lengthsRdd.reduce(_ + _)
```

Nothing happens on the cluster untill invoking the `reduce` action on line #4.

Common Actions

```
collect(): Array[T]
count(): Long
take(num: Int): Array[T]
reduce(op: (A, A) => A): A
foreach(f: T => Unit): Unit
```

Caching and Persistence

RDDs are recomputed each time we run an action on them. This can be very expensive (in time) if we need to use a dataset more than once.

Although, we can tell Spark to cache an RDD in memory, simply by calling `persist()` or `cache()` on it!

A scenario where we can benefit from caching RDDs is listed below:

```
val logsWithErrors = lastYearsLogs.filter(_.contains("ERROR")).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
val numOfErrors = logsWithErrors.count()
```

If we don't persist the filter result, we would execute that same filter twice, on `take(10)` and `count()`.

There are manyb ways to configure how data is persisted: * in memory as regular objects; * on disk as regular objects; * in memory as serialized objects; (saves memory but it costs a little more of compute time to serialize/unserialize objects) * on disk as serialized objects; * both in memory and on disk (spill over to disk, when we don't have any more memory available, to avoid re-computation!).

Spark default is memory only, cache().

Note ~ One of the most common performance bottlenecks of newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used!

Cluster Topology

- Master(Driver Program -> Spark Context) ~ This is the node we're interacting with when we're writing Spark programs!
- Workers(Worker Node -> Executor) ~ These are the nodes actually executing the jobs!

Master and Worker communicate via a *Cluster Manager*, that allocates resources across the cluster and manages scheduling.

If we take a look at the example below:

```
val people: RDD[Person] = ...
people.foreach(println)
```

The `foreach` is an `action`, with return type `Unit`. It's `lazy` on the Driver. Although, it is `eagerly` executed on the Executors!

Distributed Key-Value Pairs -> Pair RDDs

Large datasets are often made up of unfathomably large numbers of complex, nested data records. To be able to work with such datasets, it's often desirable to project down these complex datatypes into key-value pairs.

It's useful to organize data into key-value pairs since it allows to act on each key in parallel or regroup data across the network.

```
val rdd: RDD[WikipediaPage] = ...
// The following map produce a type RDD[(String, String)]
val pairRdd = rdd.map(page => (page.title, page.text))
```

```
def groupBy[K](f: A => K): Map[K, Traversable[A]]
def groupByKey(): RDD[(K, Iterable[V])]
def reduceByKey(f: (V, V) => V): RDD[(K, V)]
def mapValues[U](f: V => U): RDD[(K, U)]
def countByKey(): Map[K, Long]
```

Example where we calculate the average budget per event organizer using both `reduceByKey` and `mapValues`:

```
val avgBudgets = events.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => v1._1 + v2._1, v1._2 + v2._2)
    .mapValues {
        case (budget, numberOfEvents) => budget / numberOfEvents
    }
    .collect()
```

`reduceByKey` is more efficient that using each `groupByKey` and `reduce` separately.

Joins

Joins are another sort of transformation on Pair RDDs. They're used to combine multiple datasets. There are two kinds of joins: `inner` and `outer` joins.

`Inner joins` return a new RDD containing combined pairs whose **keys are present in both input RDDs**. Other keys will be dropped from the result.

```
def join[W](other RDD[(K,W)]): RDD[(K, (V,W))]
```

`Outer joins` return a new RDD containing combined pairs whose keys don't have to be present in both input RDDs. With `outer joins`, we can decide which RDD's keys are most essential to keep the left, or the right RDD in the join expression.

```
def leftOuterJoin[W](other: RDD[(K,W)]): RDD[(K, (V,Option[W])]
def rightOuterJoin[W](other: RDD[(K,W)]): RDD[(K, (Option[V],W)]
```

Shuffling

Suffling is when we have to move data from one node to another to be "grouped with" its key. These shuffles can be an enormous hit because it means that Spark must send data from one node to another.

```
purchases.map(p => (p.customerId, p.price))
         .groupByKey()
         .map(p => (p._1, (p._2.size, p._2.sum)))
         .count()
```

By reducing the dataset first, the amount of data sent over the network during the shuffle is greatly reduced. This kind of trick can result in non-trivial gains in performance!

```
purchases.map(p => (p.customerId, (1, p.price))
         .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
         .count()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine. For determining which key-value pair should be sent to which machine, Spark uses *hash partitioning*.

Partitions

The data within an RDD is split into several *partitions*. There are two kinds of available partitioning in Spark: Hash and Range partitioning.

Properties of partitions: * Partitions never span multiple machines; * Each machine in the cluster contains one or more partitions; * The number of partitions is configurable. (by default it equals the total number of cores on all executor nodes)

Hash Partitioning

Given a Pair RDD that should be grouped, `groupByKey` first computes per tuple `(k,v)` its partition `p`:

```
p = k.hashCode() % numPartitions
```

Then, all tuples in the same partition `p` are sent to the machine hosting `p`.

Summary

It's crucial to have an understanding how Spark works under the hood in order to make effective use of RDDs. It's not always immediately obvious upon first

glance on what part of the cluster a line of code might run on. We need to know by ourselves where the code is running!

Resources

- https://jaceklaskowski.gitbooks.io/mastering-apache-spark/