CME 323: Distributed Algorithms and Optimization, Spring 2017
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 15, 5/22/2017. Scribed by D. Penner, A. Shoemaker, and A. Santucci.

Data is growing faster than processing speeds – a solution is to parallelize on large clusters.

15.1 Data flow vs. traditional network programming

Historically, data has grown faster than processing speeds, leading to the development of data flow models, in which algorithms are parallelized on large clusters. Data flow models were developed to replace the traditional network programming paradigm, in which various problems arise at scale:

- The programmer has to manage locality of the data and the code across the network.
- Some hardware will inevitably fail, resulting in lost data and the need to restart computation.
- Some hardware will inevitably run slower than other hardware ("stragglers")
- Reading data from a distributed file-system is slower than loading data from memory
- Communicating data over the network is slow

Because the programmer has to manage all the nitty-gritty, writing to code to handle machine failure (or other performance issues) is not feasible. The first popular Data Flow Model, MapReduce (Google, pre-2005), offered the solution to most of these issues, and was the programmer's choice for distributed computing up until around 2012.

15.1.1 Advantages of data flow models

Such models compromise the programmer's ability to control functionality in order to handle the problems listed above. In the Map-Reduce model, the user provides two functions (*map* and *reduce*), which get distributed. *map* must output key-value pairs, and as a result *reduce* is guaranteed that its input is partitioned by key across machines.

To handle hardware failure, data is repilcated several times across different machines. To improve performance, code is shipped to where the data sits. In summary, data flow engines are useful because

- They save time and effort on the part of the programmer by providing abstractions
- They scale well
- Many algorithms have been redesigned to fit into the paradigm
- They have become common on clusters

What we've tried so far: MapReduce We've so far only looked at one data flow paradigm: map reduce. The problem was to split our computations across many nodes, and we wanted to be able to deal with machine failures and bottleneck machines. To address bottleneck machines, we looked at complexity measures of mappers and reducers. We also need to deal with the difference in speed when reading data from disk vs. ram, and network bottlenecks.

Motivating Spark The main thing that Spark provides is a distributed array, called a Resilient Distributed Dataset, which is a vector distributed across a cluster. It is open source and not specific to any programming language - there are API's in Python, Java, and R; it is written in Scala. RDD's are immutable (i.e. they cannot be modified after creation, and they are statically typed, so we have type safety.

All programming languages come with a vector and associated operations, e.g. intersect two arrays or finding a particular element within an array. Spark's key abstraction is the RDD. If an RDD dies, the recipe for its computation sticks around, i.e. we have fault-tolerance.

15.1.2 Shortcomings of Map Reduce

Why do we need anything better than MapReduce? Although MapReduce is no longer the dominant data flow system, the 'cottage industry' of MapReduce-friendly algorithm implementations lives on, because it can mostly be adapted to faster frameworks, namely, Spark. [1]

- Very restrictive programming interface.

  We have to frame everything in terms of mappers and reducers.

- There is too much I/O onto distributed disk, i.e. iterations are handled very poorly.

  When there are multiple map reduces, reading and writing to disk is a bottleneck.[2]

Of course, MapReduce is not as expressive as sequential computing, but languages such as Pig and Hive have been developed to provide the functionality of SQL translated under the hood into MapReduce. The main issue is that, while MapReduce is great for one-pass computations, it slows substantially in the case of multi-pass algorithms.

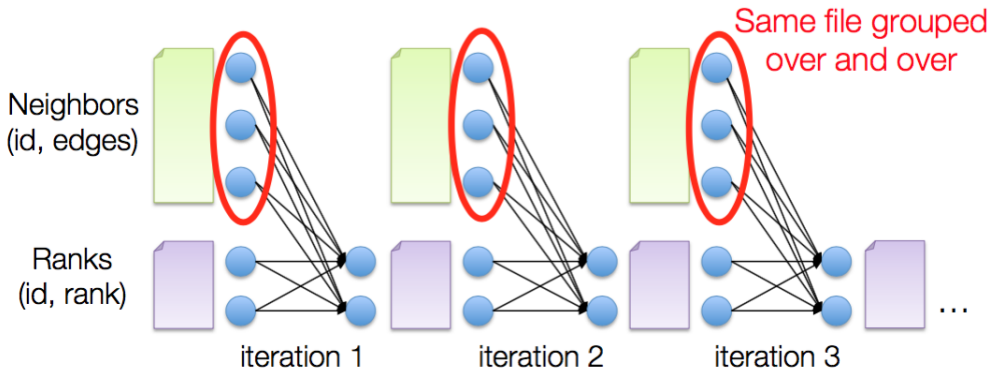
Example: gradient descent For example, if implementing gradient descent in MapReduce, each iteration will take one Map-Reduce sequence. As a result, at each iteration the machines will

[1] **Disclaimer:** "Take everything I say with a grain of salt; large parts of Spark are written by me, so I'm completely biased. Things I say are one way of doing things but not the only way. There are various competitors to Spark (Apache Beam, Apache Flink, etc). The jury is still out on who wins." - Reza

[2] MapReduce always reads to and writes to disk with each map and reduce. The reason for this is due to fault tolerance. Once a mapper is done, you want it to write to fault-tolerant and distributed disk such that its work is saved. But recall that writing and reading to disk is expensive, and some mapper operations are trivial; this causes us to spend excessive amounts of time on disk input and output. For example, consider power iteration for finding eigenvalues. Each iteration requires a matrix-vector multiply. This iterative algorithm suffers in mapreduce since with each iteration we are forced to read and write to disk.

read all the data from disk and write the results to disk, such that often 90% of time is spent on I/O. This is true even if, theoretically, there are enough machines to keep the data in memory. This is because of the method used by MapReduce to guarantee fault tolerance: write data to disk.

Example: page rank We have a matrix A about the size of the web, and we want to compute its eigenvalues by the power method: apply A repeatedly to a vector v containing the current guesses for ranks. Matrix A is stored in coo format, i.e. (i, j, value). Our guess for ranks v is also stored in sparse format (i, value). Each matrix-vector multiply needs a join, which can be implemented in one Map-Reduce. Even though the algorithm converges quickly, if it requires ten iterations, we need to read/write to disk ≈10 times, so asymptotically I/O dominates the computation.



15.2 What does an RDD look like?

RDD is a data-structure that is not language specific The concept of an RDD is not language specific; having a distributed array is a general concept. Spark is written in Scala, but there are also Python and Java bindings. The reason Scala is used is because it has great support for serializing closures (more on this in a bit). Let's look at an example of an RDD. Note that an RDD is *typed*: its signature is of the form `RDD[T]` where `T` is the type e.g. `string` or `int`.

A note on what types an RDD can support We remark that the type T of the RDD can itself be a vector, but each element of type T is stored locally on a machine, and thus each element must be able to fit on the machine's memory. For example, in the case of a matrix implemented as a vector of vectors, only the 'outer' dimension will be distributed.

Caching an RDD in memory Also note: Spark provides the option to cache an RDD, indicating that it should be kept in memory. This is useful in cases where the data will be accessed frequently (e.g. training data).

15.2.1 An applied example: filtering for error messages

RDD's come into the world in very specific ways. It has to be created via the Spark context. In the below Scala example, the file "log.txt" is stored in a distributed file system.

```
val sc = new SparkContext((;
val lines = sc.textFile("log.txt");                // This creates an RDD of string
val errors = lines.filter(_.sartswith("Error"))    // RDD[String]
val messages = errors.map(_.split("\t")(2))        // Still an RDD[String]
```

The _ refers to a parameter. The `Filter` takes a closure (which is a function passed as parameter).[3] `Map` is another functional: it takes as argument a function which gets applied to each element in the RDD.[4]

At this point, realize that Spark hasn't yet kicked off any computations. When Spark goes to actually run code, it creates a computational DAG such that it may optimize how computations are distributed. That is, Spark up until this point has performed only *lazy evaluation*. When it gets time to ship this code to mappers, each closure above gets serialized and sent to each mapper. The following line would kick off computation:

```
messages.SaveAsTextFile("Errors.txt")           // Kicks off computation
```

15.3 How RDD Achieves Fault Tolerance

15.3.1 Walking through lazy transformations

RDD's don't come into the world willy-nilly. They are tracked as soon as they enter the world, and the recipe for them is always saved. And the recipe is always very small. If one machine dies, we can just re-do the work for that one machine based on the recipe. Occasionally, we can checkpoint if we want; more on this later.

Suppose we have an RDD of words, and we just want to perform a simple word count.

```
words // An RDD[String]
words.map(x => (x, 1))
```

The shorthand above, `x =>` is syntactic sugar to define a function. Everything to the left of `=>` defines the parameters of the function, and everything to the right of `=>` defines the procedure to be applied to said arguments.

```
["CME", "CME", "CS", "Stats"].map(x => (x, 1))

= [ ("CME", 1), ("CME", 1), ("CS", 1), ("Stats", 1)]
```

We can then reduce by key to count the number of occurrences for each word. So, picking up where we left off

```
words.map(x => (x, 1)).reduce(_+_)
```

[3] This is the essence of functional programming.

[4] Example: `[0, 1, 2, 3, 4].map(_^2) = [0, 1, 4, 9, 16]`.

15.3.2 Comparing `GroupByKey` with `ReduceByKey`

In Spark, there is also a `GroupByKey`. Reduce by key will sum all values for pairs that have the same key. `GroupByKey` would just give you all of the pairs, and you would have to do something with them. However, we prefer reduce by key because we can take advantage of combiners. We've seen examples of this in our homework. Returning to our example, this leaves us with

```
[("CME", 2), ("CS", 1), ("Stats", 1)]
```

Now let's say we want to filter to observations that have at least a count of two.

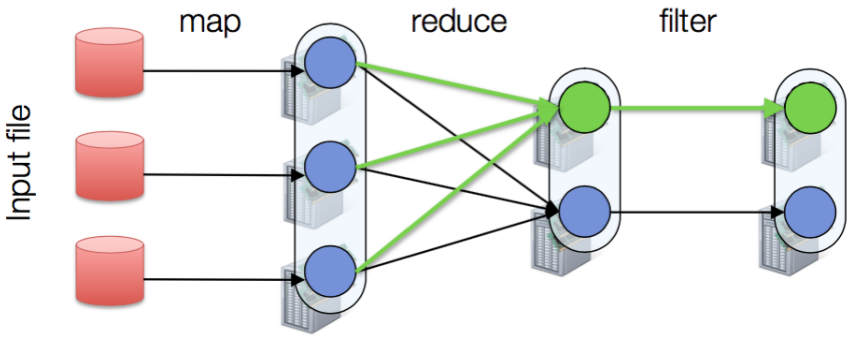
```
words.map(x => (x, 1)).reduce(_+_).Filter(_._2 > 2)
```

Filter: a functional This Filter functional takes an input argument, takes the second component of that input argument, and checks if its value is greater than two. In our case, the input argument is a key value pair, and the second component is the value (i.e. the word count). So after applying this filter, we are left with

```
[("CME", 2)]
```

Lazy creation allows fault-tolerance The lazy creation allows the us to distribute a record the process of how they are created, so the code can be re-run in the case of failure. In this way, Spark uses a different kind of fault tolerance than MapReduce, one which does not rely on writing to disk.

As an example, if we want to execute the sequence of operations, *map-reduce-filter*, and one machine in the reduce step fails, we don't need to start over - we can just recompute that step and continue, because the code is distributed to the cluster with the data.



When is Spark not appropriate? It's also important to know when you shouldn't use Spark. Spark runs on top of the JVM, Java Virtual Machine. The JVM abstracts away hardware specific details in order to facilitate portability across machines. When there's a GPU on the machine, it's hard to take advantage of it through Spark; that is, Spark may not be the best tool to use for numerically intensive computations.

Spark Libraries On top of the idea of an RDD, Spark also contains libraries backed by RDD's. Some such libraries are

- `MLlib`: a machine learning library
- `SparkSQL`: SQL translated into Spark
- `SparkStreaming`: Mini-batches of data coming in
- `GraphX`: for graph operations

Out of all these, `MLlib` is the most mature, and the one we will focus on the most. `SparkSQL` is being replaced by RDD backed data-frames.

15.4 Examples of Spark Operations

Two types of operations: transformations (lazy) and actions (kick off computations). Transformations take as input an RDD and yield a new RDD; they don't actually kick off computation, they just register the computation to happen when its needed. So, for example `map`, `reduceByKey`, and `Filter` are all transformations; they all transform an RDD into another RDD.

Transformations `map, flatMap, filter, Join, reduceByKey, groupByKey` are all transformations that you should know for your final exam.

Actions `reduce, collect, count, first, takeSample, saveAsTextFile` and more. Actions output something that can be inspected directly as opposed to creating another RDD. E.g. `reduce` could be used to sum up all the elements in the RDD; in this case it returns a single number which can be inspected on the driver machine. You should always prefer ReduceByKey, because you never have to materialize the group of tuples that represent the key, you just apply the reduce operation on them; i.e. it is more memory efficient and performant.

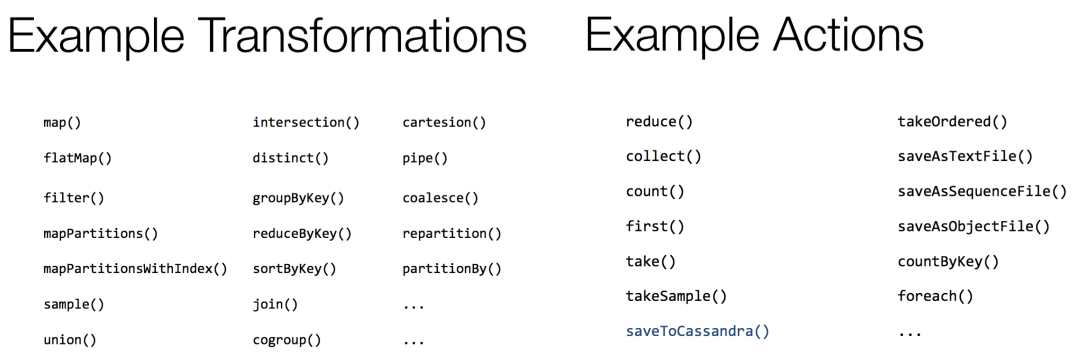
Communication Costs Some have narrow dependencies, e.g. `Filter, map`; in each case we either apply an operation to an element or filter it out locally (no communication is required). This is essentially embarassingly parallel.

With a union operation, we also don't need to incur any communication overhead. All we have to do is redress the two RDD's and tell a new RDD that it in fact points to two other existing RDDs. No computation needs to even happen, we just think of these two RDDs as one.

There are other operations that incur high communication cost. For example, a `GroupByKey` or `join` incurs an all-to-all communication, and is very expensive. A join is the most expensive operation.

15.5 Spark computing engine

Spark code is divided into two classes of operations: 'transformations' and 'actions'. Transformations are executed lazily, i.e. they don't get distributed to the cluster on their own; actions initiate parallel computations. Below is a list of some transformations and actions.



The operations used to create an RDD are transformations, and so RDD's are created lazily - they are only shipped to the cluster when actions occur.

References

[1] MapReduce-Combiners. Retrieved from `http://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm`.

[2] Broadcast Variables. Retrieved from `https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-broadcast.html`.