CME 323: Distributed Algorithms and Optimization, Spring 2016
http://stanford.edu/~rezab/dao.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 12, 5/4/2016. Scribed by Daniel Penner & Amy Shoemaker.

12 Overview of Distributed Computing

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

12.1 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:

1. The programmer has to manage locality of the data and the code across the network.
2. Some hardware will inevitably fail, resulting in lost data and/or the need to restart the algorithm.
3. Some hardware will inevitably run slower than other hardware ("stragglers")
4. If each machine reads data from disk and not from memory, and the data is communicated over a network, it is very slow.
5. 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.

Data Flow 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

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

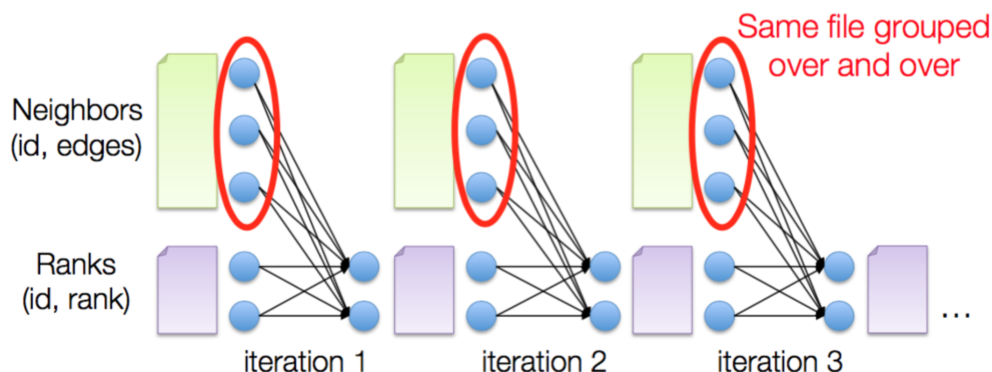
12.2 Limitations of MapReduce

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.

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 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. 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.



12.3 Spark computing engine

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]

Spark is a data flow system based on the concept of a resilient distributed dataset (RDD), a vector distributed across a cluster. It is open source and not specific to a language – there are APIs in Java, Python, R, etc (it is written in Scala). RDD's are immutable (that is, they can not be modified), and spread out across the cluster. They are statically typed, so we have type safety: we write $RDD[T]$ for a data type T (e.g. string, integer, etc).

[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

Note: 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.

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).

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.

Example Transformations

map()	intersection()	cartesion()
flatMap()	distinct()	pipe()
filter()	groupByKey()	coalesce()
mapPartitions()	reduceByKey()	repartition()
mapPartitionsWithIndex()	sortByKey()	partitionBy()
sample()	join()	...
union()	cogroup()	...

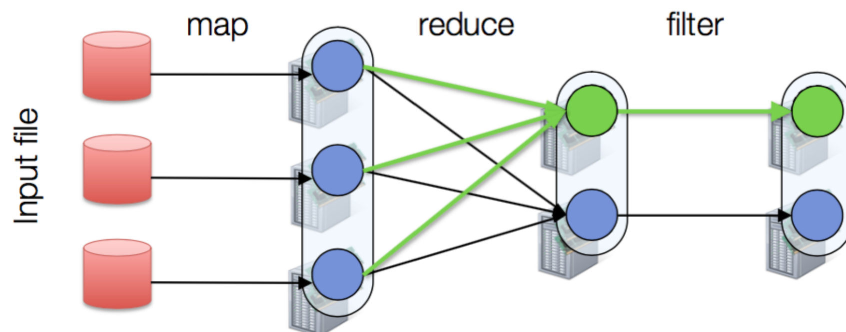
Example Actions

reduce()	takeOrdered()
collect()	saveAsTextFile()
count()	saveAsSequenceFile()
first()	saveAsObjectFile()
take()	countByKey()
takeSample()	foreach()
saveToCassandra()	...

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.

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.



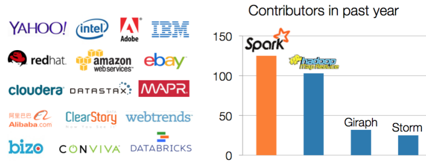
12.4 Current State of Spark Ecosystem

Spark is open-source, and has an active and growing community of users and contributors.

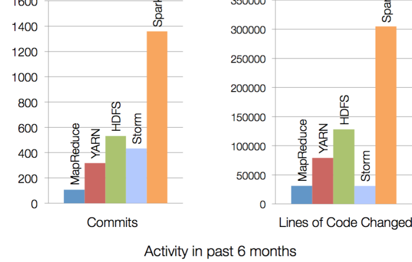
Spark Community

Most active open source community in big data

200+ developers, 50+ companies contributing



Project Activity



12.5 Built-in Libraries

Spark contains many libraries that provide abstractions for common applications:

1. **MLib** provides a framework for machine learning and optimization algorithms.
2. **GraphX** attempts to provide a useful abstraction for graph algorithms, but was poorly implemented. As a result, communication costs dominate and performance is awful.
3. **Spark Streaming** simulates streaming computation as a series of small batch jobs, which is useful when the stream is too big to process sequentially.
4. **Spark SQL** allows loading and querying structured data.

12.6 Tips for Using Spark Efficiently

Spark also provides support for tuples, called PairRDD's, and provides two transformations for grouping tuples: reduceByKey and groupByKey. The former applies *reduce* locally in each machine, and passes and combines the results across machines, and thus is only possible if *reduce* is an associative operator. The latter is more general, but as a result much slower, because the data is shuffled across the machines, and then grouped by key, so that all the data has to be passed across the network.

Similarly, embarassingly parallel operations can be distributed effectively with little cost (e.g. map, filter, union, and join for co-partitioned data). On the other hand, operations with dependencies incur the costs all-to-all or all-to-some communication between machines (e.g. groupByKey, join with inputs not co-partitioned).

References

[1] R. Zadeh. *Introduction to Distributed Optimization.*

[2] R. Zadeh. *Distributed Computing with Spark and MapReduce.*