CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 4, 04/08/2015. Scribed by Eric Lax, Andreas Santucci, Charles Zheng.

Lecture Overview

Today we will talk about what happens when you submit a job to a Spark cluster. What does this involve?

- Splitting up your data into as many parallelizable chunks as possible
- Serializing and shipping code to the clusters
- Scheduling computations which have been serialized to run where data is stored, to minimize network communication costs
- Shuffling
- Broadcasting
- Bindings

Overview - How Spark Ships Code

To see a brief overview of the life of a Spark Program, consult the lecture slides, page 4. The general idea is that we first create some input RDD's from external data, we then transform them to define new RDD's using transformations such as `filter()` or `map()`, we cache any intermediate results that will need to be reused, and then we launch actions such as `count()` or `collect()` which actually kick off computations. Because these actions are lazily evaluated, Spark optimizes them ahead of time.

Most actions are optimized by a sort, especially those which involve all-to-all communication. If we can sort our data by machine, we can then perform group-by and reduce-by operations efficiently.

Representing an RDD How does the process flow through Spark? When an RDD is created by a *driver*, we implicitly construct a *Directed Acyclic Graph*. The DAG is typically very small, on the order of several kilobytes, and consists of pointers to files and closures which have been serialized. Within the DAG, some tasks are embarassingly parallel, but others depend on each other. Spark figures out which parts can be computed in parallel, and which must wait for others to finish. See slide 10.

Drivers and Executors When Spark optimizes code internally, it splits it into *stages*, where each stage consists of many little *tasks*. The stages are determined via standard graph algorithms. Every task for a given stage is a single-threaded atom of computation consisting of exactly the same code, just applied to a different set of data. This task level is where *fault tolerance* is built in. If a machine dies, the task can be recreated and redone. It's also possible to build in cluster-manager fault tolerance. In general, if a driver fails, it must be restarted; it's preferred to run the drivers from a reliable machine.

Cluster Managers The driver begins making requests to the *cluster manager*, which has control over all machines, and coordinates resources such that they are used efficiently. The cluster manager informs the driver which worker they may assign, and assigns *executors*. Examples of cluster managers included Yarn, Mesos, and the Spark standalone manager. Different companies sometimes use their own proprietary cluster managers.

The cluster manager assigns machines to tasks based on where the data lies, and in doing so minimizes the network communication cost. If this cannot be accomplished, the system is willing to incur some communication costs.

Scope of a Task Tasks can perform non-trivial operations. It may start serving all-to-all communications. Sometimes a task can be a bunch of small computations involving no network communication, and other times it can be non-trivial and involve setting up a small server which ships data that its responsible for.

One point worth noting is that RDD's are immutable, which is one of the reasons why Spark is implemented in Scala. The RDD knows the I.P. addresses for where the data sits. When the driver requests resources from the cluster manager, it starts with requests which match workers to tasks which can be performed on their local data store. The workers themselves never communicate with the Spark context directly.

The whole process we have described can also happen locally on a machine, which is what happens when Spark is run on a single machine.

Now, we discuss how communication patterns are implemented. From the application designers point of view, this is the interesting part. Certain actions such `groupByKey` and joins with inputs which are not co-partitioned have *wide dependencies*.

Shuffling

Sorting has been the bread-and-butter of Computer Science for decades. We utilize these advances in distributed computing by framing any operation which involves all-to-all communications as a sort. Currently, TimSort is used, because it efficiently manages streams of data between memory and disk.

Estimating Distribution of Data Across Workers In a distributed environment, we have a giant data set split across machines. We'd like to sort it such that machine 1 gets the first chunk

of data, machine 2 gets the second chunk of data, etc. To do this, we need to know about the *distribution of the data*, such that we may determine optimal splitting points in a way that facilitates uniform data distribution across machines. We desire equally balanced partitions, because in a distributed computing framework, computation time is dependent on the bottleneck worker. To do this, we take a uniform sample of each machines' allocated data, send this information to the driver, which then determines optimal binning points.

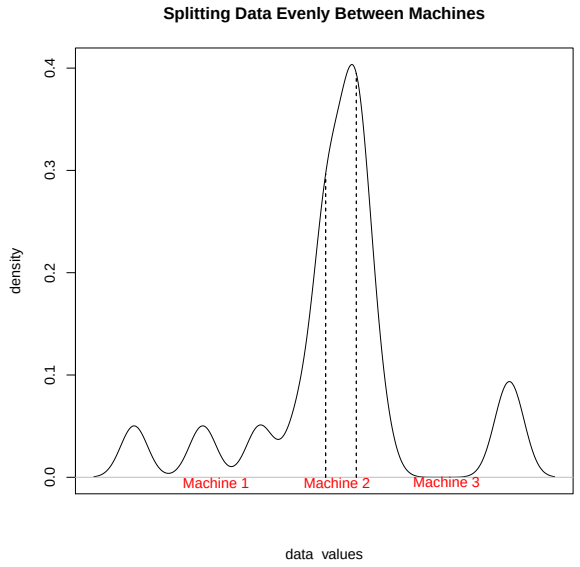

Figure 1: Estimating Optimal Binning Points

We now have boundaries which determine which machines are responsible for which sets of data. Each machine sorts their local data using TimSort, and in doing so also builds an inverted index which describe where the data for each machine "starts and stops". Since the data is sorted, these partitions are stored *contiguously* in disk. When it comes time to serve the data to other machines, the data can be quickly read from disk with minimal seek time.

Each machine knows all split points, and each machine has sorted their local data. Suppose machine 5 is querying data, it uses an *index* which serves for the network. It describes, "for machine 1, your part of the data starts here on disk, and stops here", using pointers. Note that during the shuffle process, the amount of data each machine holds can double, because machines are serving and querying data at the same time. Each machine pages to disk as much as possible, since data can't be stored in memory on any one machine, but we have sorted our data in a way that lets us write to disk in contiguous bins.

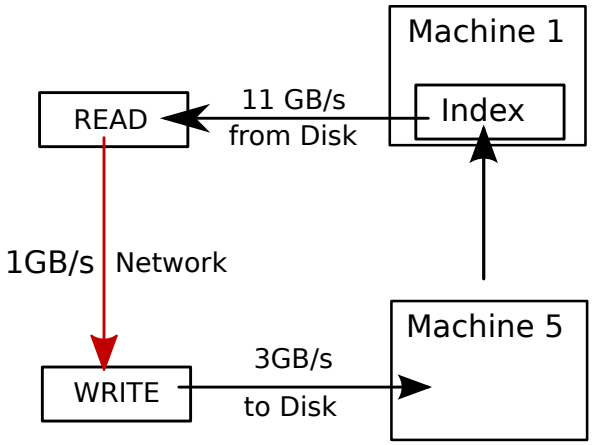

Figure 2: Distributed Sort, when done properly, is bottle-necked by network transmission rate

Joins and Sorts What are we doing in a join? We try to find keys that are the same. How do we do this? We dump the data from different sources into one area, sort them, and then the keys that are the same show up next to each other. Ideally, we'd like to avoid shuffling our RDD's if one of them is small. We will next look at interesting communication patterns which are not all-to-all.

Map Side Join If one side of the join is *really* small (e.g. 100 megabytes), we could have the entire array of data sent to each machine, which can then merge the results with their larger RDD. That is, we look at all the little parts of an RDD and intersect each locally with the entire other array. When the size of data being transmitted is very small, we use *broadcasting*.

Broadcasting Take any Scala object which is not too large, which is serializable, and send it to all machines. These items are immutable by construction. Broadcasting lets us work on a *per-machine* basis. Typically, each machine contains all data and everything it needs. Broadcasting makes it at the node level. Such that each machine only gets one copy of the data being sent around. This is useful in machine learning, when models can sometimes be as large as a gigabyte.

Bit Torrent Broadcasting The naive way of broadcasting is having one machine communicate all other machines. This method leads to network saturation and increased wait times. Instead, start out with data sitting on the driver, then send out the data to as many nodes as the network can tolerate, and each of the recipients repeats this process, leading to exponential growth. The branching process depends on how much network capacity is available. This is called *bit-torrent* broadcasting.

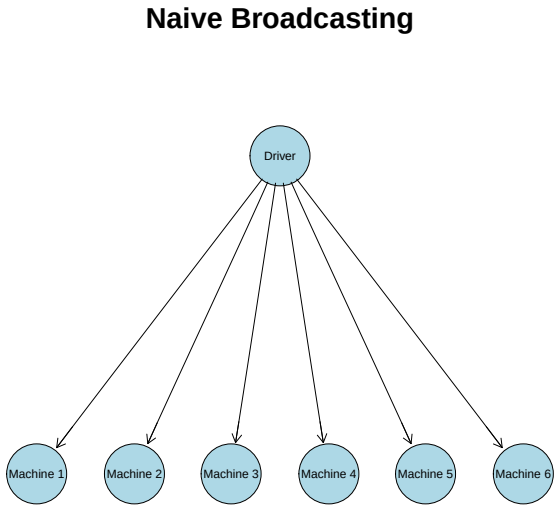

Figure 3: Naive Broadcasting

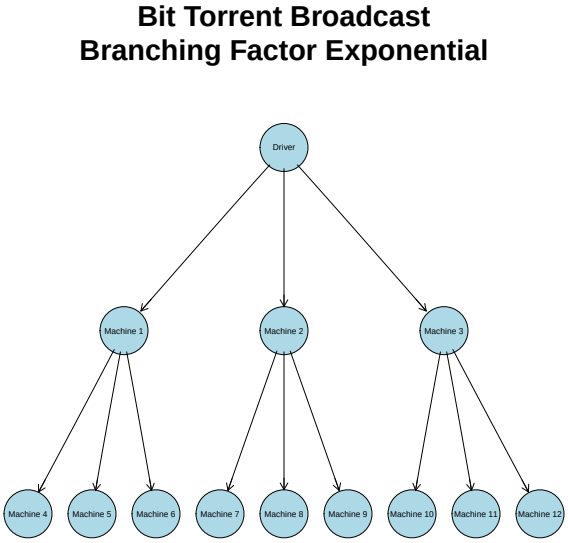

Figure 4: Bit-Torrent Broadcasting

Broadcast Rules Broadcast objects must be created using a Spark context, and they must be immutable. The broadcast is accessed within a closure using a `.value` notation.

Replicated Join Suppose we have a small RDD, which is so small it's just an array. We may broadcast this data to all machines. Suppose we have another piece of data which is very large,

which must be stored in an RDD. See slide 21. With a small array, each machine gets the entire array available to them. They can then easily see where the intersection is with their data and the entire array. There is no communication cost involved once the action is kicked off.

Broadcasting for Optimization See slide 23. Take an optimization problem we saw in Lecture 2. If we don't broadcast, each machine as part of the task must obtain a *copy* of the entire array, since the closure must contain all necessary information to perform the task. Since we often have giant models, sending out all this information creates unnecessarily large tasks. Therefore, we may broadcast the model to each machine, which saves huge amounts of space on each machine executing a task. From then on, each machine simply has a pointer to the broadcast variable, rather than storing the whole model locally. The data is then sent back to the driver using an "all-to-one" operation.

Piping Spark

It's possible to use command line arguments to pipe operations from Spark into other API's. This lets us do very complicated things using legacy code we might have. Each worker, for example, can start up their own instance of Python to help execute their task. Slide 25.

References

[1] R. Zadeh *Lecture 4: Shuffling, Communication Patterns.*