Spark.js

6.824 Final Report

0 Team

Rohan Mahajan, Steven Allen, Luke Anderson, Michelle Wang

1 Introduction

While cluster computing and distributed data sets are becoming more commonplace in research and industry, the majority of people do not have easy access to resources that would allow them to efficiently process distributed data. Our project seeks to provide anyone with the ability to collaboratively process distributed data by leveraging the computing power of anyone with a web browser. Our system is run using node.js and is based on the Spark RDD model.

2 Design

2.1 System Components

Slaves

The *slaves* are the client browsers that connect to the *server* and volunteer to operate on a dataset (using the JavaScript library).

Server

The *server* is responsible for coordinating *slaves* and stores the initial RDD description and a list of known *slaves*. However it neither stores data nor runs any expensive computations. Instead, it acts as a manager and assigns tasks to slaves.

Master

The *master* is the client that uses the webapp to submit and manage an job.

2.2 Client Interface

Both master and slave clients interact through a web interface. Through this interface, a master can input code to create a job and see the peers working on the job as well as the results. The web interface gives each job a peer url that other volunteers can then connect to in order to lend their computing power. The client API for manipulating RDDs is similar to the Spark interface with some minor differences and exclusions. Supported operations are coalesce, map, filter, flatMap, mapPartitions, multiGet, and split. We allow the master to define the dependencies and partition their own RDDs. The slave interface displays job and partition information to each

volunteer, allowing them to see what they are working on as well as safely disconnect from the computing cluster when they are finished.

2.3 File System

Unlike in Spark, we do not have a file system. To solve this problem, we use an abstraction of the distributed file system. These “files” are stored in the memory of the slaves. The server knows the location of all the files in this system, but does not serve requests. For instance, let us say client A wants a file that happens to reside on client B. Rather than implementing a system that would route data through the server whenever client A wants to request a file, we chose to go with a peer to peer approach instead. Client A asks the server for the location of the file and waits for the server to respond with client B. Client A then establishes a peer to peer connection with client B, requesting the file. Thus, we are able to bypass placing an additional load on the server and reduce amount of data being transferred around the network. To implement the peer to peer connections, we spent a considerable amount of time implementing WebRTC. The server also pings each slave periodically. Thus, if a slave goes down, the server will know within a short period of time, reducing the likelihood that a non-existing file will be served.

2.4 Scheduling Jobs

Generating and Scheduling Tasks

Another key component of the server is the scheduler, which keeps track of jobs, delegates work, and monitors the slaves. Based on the RDDs, partitions, targets, and dependencies, the server builds a dependency graph for each job. It firsts prune the graph based on the tasks given. In other words, it walk up the graph and only include nodes that are necessary for the tasks to be calculated.

After the server has pruned the job graph, it breaks the graph up into tasks to assign to our slaves. There are two key abstractions that comprise a task: the source and the sink. A source indicates a node in the job graph that is already computed and stored in slave memory. It is a start point for further computation. A sink, similar to a checkpoint, is a node that should write state to the file system after it has finished computing. These serve as endpoints for computation and can later become sources for new tasks. The invariant that a sink is always someone else’s source should hold throughout because you can only read what someone else put in the file system.

The two key points in the file system are a split, where a node breaks up into more than one nodes, and a join, where more than one node joins together to form another node. For the split node, the first parent to reach it will mark the node as a sink and then pick one of the branches to keep going. For the other children, that split node will become a source. For the operation join node on node A, the server choose one of its parents, which we will call B. B will then mark all of A’s other parents as sources but keep going. A’s other parents (such as C,D,E) will not traverse

down to A but mark themselves as sinks. Thus, the task with B will keep waiting on C,D,E tasks to finish before it moves down to A. This prevents the same computation from happening twice. In the absence of failures, each node in the job graph should only be computed once by the time the job has finished.

Assigning Tasks to Workers

Once tasks have been created, the server needs to assign them to workers. The server first tries to combine really small tasks into bigger tasks, especially if they have a matching sink and source. Additionally, tasks that originate closer to the start of the graph are prioritized. Finally, we try to load balance as best as possible in terms of the number of tasks, i.e. ensuring that each worker has relatively the same number of tasks.

Once a slave receives a task, its starts computing it. Once it reaches the task's sink and writes to the file system, the server will know that the task has been completed.

2.5 Fault Tolerance and Vulnerabilities

Server

We did not implement replication for the server. Thus, in the case of server failure, jobs would have to restart from the state contained in the slaves. In the future we could implement better fault tolerance by doing some sort of logging with a system such as MongoDB or using a backup server. Additionally, the server is the only piece of hardware that we would need to actually own, so we could invest in a high quality server.

Master

Once a master has launched a job our system will continue doing that job. If the master logs off or fails while the job is still in progress, the server can let restore the current state once the master logs back in.

Slave

The slave is responsible for two parts in our system: either computing some type of task or storing files and state. Our server pings the slaves every so often to ensure that they are still functioning. Consequently, if we find that a slave is dead, the server will mark the relevant files in the internal file system to indicate that they no longer exist. The work will then be reassigned to a different slave.

2.6 Testing

We implemented automated tests to ensure that network connections are being made correctly and that all peers can communicate with each other. The tests automatically create dummy peers and an empty job. It then verifies that all slaves have been set up correctly and that messages can be sent and received.

2.6 Conclusion and Future Work

We have built a basic system that allows users to sign up, volunteer, and run various distributed computations on a web browser. In the future, we would like to improve a couple of things. First, we would like to improve reliability; i.e. improving on the availability of the server. Additionally, we would like to improve performance by improving facets of our system such as optimizing the scheduling and task assignment. Moreover, we would like to add additional RDDs operations to help improve ease of developer use. Finally, we would like to compare the results of using our distributed computation system to running the jobs locally using something like python.