15-440/640 Distributed Systems

Lab 3 Report: Map-Reduce Engine

Team:

Abhishek Bhowmick (abhowmi1),

Neil Rajesh Dhruva (ndhruva)

Date:

11/22/2014

Contents

Problem Definition

Distributed File System

Client API

Distributed Map Reduce

Tips for the System Administrator

Examples

Conclusion

Problem Definition

The problem deals with designing a map-reduce framework similar to Hadoop for parallel execution of user tasks. The framework should be able to deploy such tasks in a distributed environment on multiple nodes, with the capacity to start and monitor jobs deployed by a user.

In addition to the parallel execution of tasks, the framework should provide a distributed file system abstraction that keeps track of input and output files corresponding to user jobs. And, while the engine is not required to sort and combine the final output, it should be able to parallelize user jobs efficiently and distribute the work for concurrent execution to improve job latency compared to using a single machine.

The framework should implement an effective scheduler that distributes task load efficiently. Additionally, both the distributed file system, and the map-reduce engine should be capable of recovering from node failure. Additionally, the framework should provide a way for differentiating between records in user files so that users can deal with each record individually through map and reduce functions.

The framework should provide an API for the user, i.e., the application programmer, to submit tasks, and documentation for the system administrator to effectively use the map-reduce engine. Finally, two examples should be provided in order to demonstrate the functionality of the system.

Distributed File System (with Namenode and Datanodes)

The distributed file system implemented as part of the map-reduce framework acts as an abstraction for a global file system available across all nodes participating in the framework. It, in many ways, is similar to the way HDFS is implemented in the Hadoop framework.

The key feature of the distributed file system (henceforth, DFS) implemented in this project is how it emulates an actual file system with physical disk space. It provides the user with the ability to add files to the DFS, view the structure of the DFS at any given time, delete files present on the DFS and get (download) files from the DFS. The abstraction provided is in the form of a representation of a file system, which acts like a regular file system, but distributes the actual data provided by the user across different machines (datanodes) present in the map-reduce engine rather than storing it on one machine.

Working:

The DFS is implemented on the Namenode, i.e., the master node of the map-reduce framework. The user communicates with it through the Client API. In fact, the client API is the user's gateway to the map-reduce engine. As will be discussed later, the user adds, downloads and deletes files to and from the DFS using the client API, and also submits map-reduce jobs using the same.

The client API provides the user a clean and simple interface to the DFS by abstracting away all network I/O, and displaying only the abstraction provided by the DFS. It also hides the details of how user files are stored across datanodes in the framework. The user deals with files in a manner similar to dealing with local file system files. Hence, the client API, to a great extent, reduces the effort that the user has to put into worrying about processing data in huge files.

The client API itself communicates with the DFS hosted on the namenode using Java's RMI facility. This makes it very easy for system administrators or system developers to add more functionality to the DFS as and when

required. All calls to the DFS are made through the DFS service hosted on an RMI registry on the namenode. Hence, a major overhead of sockets and network I/O is completely removed from the process. The system developer can thus focus on the functionality provided by adding more features to the DFS, rather than worry about sockets and network messages.

The actual working of the DFS involves converting the user input files to fixed-sized blocks, in order to allow the map-reduce engine to work on smaller blocks of a file, concurrently. Additionally, the DFS maintains a replication factor for each file added by every user. The replication factor (if greater than one,) ensures that a particular block belonging to a file is available across different datanodes. In short, the number of datanodes that the block is available on is equal to the replication factor.

When the user adds a local file from the user's file system to DFS, the DFS identifies certain datanodes (according to the replication factor) that can be used to store the file blocks. This is done using a round-robin algorithm to distribute the data across the different nodes. This way, we ensure that the next block goes to a datanode with lesser load (according to space occupied on the datanode) than others. Once the DFS identifies these nodes, it forwards the list to the client API that does the actual work of splitting the user input file into blocks (using an input split criterion provided by the user), and sending these blocks across the network to the datanodes. The split criterion provided by the user can be in the form of a delimiting string or a certain number of bytes per record. However, for now we assume that unless the user specifies the number of bytes per record, a newline character will separate every record.

Once the input file is split, the blocks are sent as byte streams across the network to the datanodes using RMI. To enable this, and various other features, each datanode has an RMI service (called the DN-service) bound to an RMI registry on each such machine. The communication to these services on various datanodes is done through a Node API provided for the framework. The client API uses the node API in order to send blocks to the datanodes. Once the file transfer is complete, the user can invoke the map and reduce functionality of the engine.

Additional facilities of file download and deletion are also provided to the user through the client API. When the user requests a file from the DFS, the DFS sends a list of datanodes to the client API which constitute blocks of the file requested by the user. The client API communicates with the node API (which, as described above, is an interface for the datanode RMI service) in order to download the corresponding blocks to the local file system of the user. A file deletion command from the user causes the client API to forward the request to the DFS service on the namenode that deletes the file blocks from the datanodes that store these blocks, using the node API. Finally, the user is also provided the service of printing the current directory structure of the DFS using the appropriate API call on the client API. The DFS does *level-order traversal* of its structure, described next, and sends it back to the user.

The DFS structure is represented on the namenode using a *trie data structure*. Each node in the data structure represents a virtual directory that points to zero or more subdirectories, and also points to files stored in that directory. The user always specifies file paths according to the base directory of the DFS. The base directory is the root of the trie. The directory is traversed every time a file is added, requested or deleted, and the corresponding data structures that represent the nodes in the trie, as well as the data structures representing the file itself, are accordingly added/altered/deleted. Each data structure that represents a file in the trie, stores the list datanodes on which the blocks of the file are present, which in turn is used to do operations on the physical blocks stored across the datanodes.

The question that now remains is, what if a datanode fails? Does the user lose the files uploaded to DFS? The answer is, no. As long as the user, i.e. the application programmer, maintains a replication factor greater than 1, there will always be a copy of a file block on more than one node. This is possible because of the round-robin algorithm used for assigning datanodes to store file blocks. The algorithm makes sure that as long as there are more than 1 nodes active, and the replication factor is greater than 1, we have a copy of the block on more than one nodes. However, what if more than 1 node goes down? Well, we have provided a solution for such situations too. Every time a node goes down and detected by the job tracker (periodically),

the namenode/DFS is notified of the same. When this happens, the namenode immediately identifies all the blocks of all files that were present on that node, and all the nodes that have a duplicate copy. These nodes are then asked to replicate the corresponding block(s) to one or more nodes that do not already have the block(s). This way, the replication factor is maintained. And, this is also done in the round robin manner described above. It is important to note that the transfer between datanodes is done entirely by the datanodes, and the namenode is not involved in routing the physical file block form one datanode to another.

However, if the replication factor is more than the number of available datanodes, then only one copy of each block is sent to the various datanodes. This is to avoid unnecessary usage of space on the datanodes. The main reason behind not maintaining multiple copies of the same block on a particular datanode is that most of the time (and the only possibility with our framework as of now) the user only reads from the uploaded file, and does not alter it. Hence, it is not necessary to maintain multiple copies of the same block on a datanode.

Finally, additional features are provided to the DFS like checking the validity of the path on the DFS to add/get/delete a file. Also, we provide the facility of making sure that a block has successfully been received by a datanode, as directed by the namenode/DFS. If the block was supposed to be sent, but was not successfully transferred to a datanode, then the DFS maintains a note of this fact and makes sure that the block is never requested from that datanode.

Summary

The DFS is an abstraction for a system-wide file system for the map-reduce framework. It is maintained using different representations for directories and files on the DFS. It is hosted at a central location, the master node (namenode) to be more precise. In order to conduct map-reduce operations on data, the user has to add these files to the DFS. The client API provides various functionalities to the user to add, download and delete files from the DFS.

When the user adds files to the DFS, the DFS distributes the file blocks across various datanodes, and maintains a replication factor for each of the blocks while guaranteeing fault tolerance. Java's RMI facility is used extensively to communicate between the client, DFS and the datanodes, as well as for transfer of file blocks. And, corresponding API's are provided for the application programmer to interact with the DFS in an intuitive manner.

Client API

The client API provides the application programmer (user) of the map-reduce framework, the ability to interact with the Distributed File System (DFS), and run map-reduce jobs on the datanodes. Following are some of the functionalities available to the user:

1. Add files to DFS: When the user adds a file to the DFS, it is separated into blocks of fixed sizes and replicated across datanodes. By this, we mean that the actual file is converted to blocks that are scattered and stored on the local file system of several datanodes. *The user must supply a single file ending in ".txt" when adding it to the DFS.* This limitation ensures that the user adds only one file at a time, and not a directory. However, the limitation can be modified easily in the future. The user does not have to stay connected to the DFS in order to access files added during a session. The files stay on the DFS as long as the DFS is running as a service on the namenode. The user can now submit map-reduce jobs to run on the added files.

2. View DFS structure: While the DFS service is running on the namenode, the user can, through the client API, query the DFS to get the current file structure, and the files added by all the users. This can also be easily curtailed for security purposes by showing only user specific files in the future.

3. Get file or directory from DFS: The user can specify the name of a file on DFS and the client API will download it (in the form of blocks) from the datanodes. The blocks are not combined as of now, inn our implementation. The user can also ask to download all the files in a directory on DFS. This function is mainly used when the user wants the reducer outputs after a map-reduce job is complete.

4. Delete file from DFS: When the user deletes a file from DFS, the namenode deletes all the blocks of the file from the currently active

nodes on which any of those blocks are present. The file is also deleted from the DFS, i.e., the link to the corresponding data structures in the DFS trie is removed.

5. Check if a file exists: Finally, the user can check if a file has already been uploaded to the DFS. In that case, the user shall not upload it again to prevent file transfer overhead. In addition, the user can also ask to overwrite a file on the DFS. In that case, the file is deleted and then the add file method is called by the client API.

Distributed Map-Reduce (with JobTracker and TaskTrackers)

The distributed map-reduce is the component that handles the distributed processing of user provided job. A master-slave architecture similar to the Hadoop framework has been implmented. Henceforth, we refer to the distributed map-reduce framework as MapRed.

MapRed not only increases the throughput of the job through parallel processing over a cluster, it also enables us to run jobs with huge (of the order of terabytes) inputs, which would have been impossible to run otherwise on a monolithic machine. This is made possible with the help of a distributed file-system such as DFS, that spreads and manages the storage of machines across the local storage capacities of all machines in the cluster.

Computing

The framework consists of a node designated as the master (henceforth called JobTracker). This component is responsible for managing the distributed computing of the entire cluster. All the remaining nodes in the cluster are slaves (henceforth called TaskTracker) which do the actual computing (map or reduce tasks). Here, we provide a very high level overview of the functioning each of these node types.

1. **JobTracker** : This node takes in a user supplied mapreduce job and stores all the associatd metadata in addition to the state of the cluster and tasks in progress. It maintains a table of all jobs running on the system (henceforth called the 'state table'). Each entry of this table further contains 2 table, one each for map tasks and reduce tasks that will be deployed on the cluster. Every user job submitted to the JobTracker is processed by a separate thread that identifies the job, its parameters, calculates the number of mappers/reducers it will need, and queues them up for execution. The JobTracker runs a separate dispatcher that contains a scheduler. The scheduler chooses one of the queued tasks, determines the node to which it should be dispatched making use of locality information and return this information to the

dispatcher. The dispatcher then tries dispatch this task to the determined node. If it fails, it picks up a new map task and tries to dispatch on a slave node.

2. **TaskTracker** : This node runs the tasks that are sent to it by the dispatcher from master, for computation. It has an upper threshold on the maximum number of tasks it can run though, and and any tasks requested above this limit are rejected by the TaskTracker. For every new task it receives, it spawns off a new thread to execute this task. When the task execution finishes, it indicates its completion to the master that in turn marks the task as 'done' in its 'state table'. The TaskTracker also has a state table similar to one on master, to keep track of the tasks (potentially beloning to different jobs) that are currently running on this node. We ensure that we hide the state table (and the ability to modify it) from the task execution thread, so that malicious user code for map/reduce cannot affect the state of the TaskTracker. Thus, we achieve some of the isolation that we could have achieved by spawning separate JVMs for each task.

Fault Tolerance

Fault Tolerance is primarily achieved by dedicated background threads at both the master and slave nodes. The JobTracker runs a separate polling thread in the background that periodically polls all the slave nodes in the cluster and based on their replies, updates their status to “up” / “down” in its 'state table'. The TaskTracker runs a similar thread that listens for these queries from the master and replies to indicate it's status.. Note that this background thread is configured as a daemon thread, so that it stops running if the TaskTracker goes down.

Since, the JobTracker periodically polls all the nodes in the cluster (whether dead or alive), it can therefore detect any dead nodes that come alive in the future. It also periodically notifies the NameNode component of the DFS about the nodes that are currently alive in the system.

Communication

The communication between the different modules of the MapRed framework are accomplished by a message passing protocol. The socket port numbers for various required connections are read off from a system configuration file. Low network overhead while communicating through messages over TCP sockets gives it a considerable performance advantage over RMI (remote method invocation). Given the large volume of communication between master and slaves in the MapRed framework, we chose messaging for this part. However, we use RMI for communication in the DFS, due to the simplicity of implementation.

As a result of the above decision choices, our MapRed modules communicate with the components of the DFS (NameNode and DataNode) through RMI.

Task Scheduler

The scheduler that we use for dispatching tasks on the slave nodes follows a simple Round-Robin policy across scheduled jobs. We access information about the locations of file blocks by querying the NameNode from DFS. For instance, while scheduling a map task, for every file block we wish to work on, we acquire the list of nodes across which it has been replicated and among those, choose the one with least number of running tasks. For a reduce task, we simply choose the node that has the least load.

Note that the above logic though simple, achieves fairness and also tries to ensure that all slave nodes are working on full capacity. We also retain the availability of adding more complex scheduler logic through any object that implements the 'Scheduler' interface.

Tips for the System Administrator

The system administrator (sysadmin) plays an important role in determining the specifics of the map-reduce engine. The sysadmin has to make various choices ranging from the location of the namenode and datanodes, the ports for hosting the different services, the location of user file blocks on datanodes, the replication factor and so on. To make life easy for the sysadmin, we have defined a configuration file ("ConfigFile") that the sysadmin can change in order to control various parameters.

Some of the parameters that the sysadmin can control through the ConfigFile are defined below:

- DFS and Datanode machines and registry ports.
- Replication factor of the blocks generated from the user file.
- Base directory on the local file system on datanodes where the file blocks will be physically stored.
- The size of the block that user files are split into.
- Various ports used for communication between the user, task trackers and job trackers.
- And finally, the maximum tasks (map or reduce) that can run per datanode.

In addition to configuring these parameters, the sysadmin is responsible for starting and monitoring the system, and shutting it down when required. This is done through the command line and the specifics on how to do this are provided in the README file provided with the code.

Examples

We provide two examples, described below, to show the functionality provided by our map-reduce engine:

1. Word Count: This is the standard word count example, that computes the word frequencies of all words in a large document and displays them in lexicographically sorted order. The input is in the form of fixed size records with size 60 characters, and each record contains a list of names (CMU faculty and Hollywood actresses) delimited by commas. The size of the data file is approximately 128 MB and has been manually generated.

   Testing

   This example was tested to work correctly on the GHC with no splitting of the input file and one TaskTracker. We have also implemented the logic for running map jobs with Input files split into multiple blocks and with multiple TaskTrackers, but these features haven't been tested on the cluster machines. We have also implemented and tested Fault Tolerance on a single machine with multiple JVMs for the master and slaves.

2. Geo Location: DBpedia provides a dump of geographical data associated with various locations mentioned in Wikipedia articles. These locations range from cities and countries to physical location of objects like bridges, monuments etc. The data is provided in a CSV format with the name of the place, the latitude and longitude of the place, and some additional information about the article.

   Each of the locations is in the form of a separate line, with comma-separated details pertaining to it on that single line. In this example, the mapper extracts the latitude and longitude for each unique place, and group the locations into 4 categories: North-East, North-West, South-East, South-West.

   The grouping is done based on the latitude and longitude values of the location, such that the latitude forms the horizontal axis and longitude,

the vertical axis. If both latitude and longitude are positive, for instance, then the label for that record is “North-East”, and so on. The reducer then combines the mapped output and sums up the number of articles for each category.

Data source (contains subdirectories 3.1/en, 3.2/en, 3.3/en, 3.4/en from which the data was obtained): http://downloads.dbpedia.org/

The file name in each directory is “geo_en.csv.bz2”. We combined four such files to give a total of close to 450MB of data.

Test

The only way in which this example differs is that unlike the previous one, we do not have records have fixed size here and hence is more representative of a real world input. We have implemented the logic to deal with such a scenario as well, however we haven’t tested it properly.

Comments

Though we have implemented most of the expected features such as running jobs to completion, kill jobs, get status reports, fault tolerance etc – we haven’t tested them thoroughly on the cluster machines. However, we have tested all the components through unit tests and integration tests in the process of development, using multiple JVMs on single machines. Thus we replicate most of the challenges associated with building such a system in a distributed environment and believe that with more testing/debugging effort, we can have a fully functional, robust MapReduce system.

Conclusion

The distributed system described above implements a multi-node master/slave framework that caters to a user's commands of creating and migrating processes between nodes, as well as generating a report of the activity at any given time in the system. While the system works reasonably well according to the test cases that we performed, there are a lot of additions that can be performed (like ACK) to make the system more robust.

However, we believe that this system can deal with requests from a single user efficiently, and also perform efficient and frequent bookkeeping in order to generate useful reports for the user.