15640 Project 3 Report

Building a Map-Reduce Facility

Zichang Feng, Siyuan Zhou

zfeng, siyuanz

What we have done

- We provide two examples along with scripts to deploy our system and run the examples.
- We wrote and generated java docs(in folder doc/index.html) and guideline for system administrator (in Administrator Guideline.pdf)
- We implemented a distributed file system that support read, write, ls and delete operations. We implemented a heartbeat protocol between name node and data node to keep track of health status of data nodes. We also maintained replicas in the DFS to handle data node failures.
- The name node and data node periodically write image file to disk such that the system is in the same state after restarts.
- We implemented an interactive DFS client to operate on files in DFS including list all the files, list files with a given prefix, upload file to DFS, download file from DFS, delete file and shutdown the file system.
- We implemented a Map-Reduce facility that allows user to submit jobs from any node.
- We designed two way to submit a job, either by submitting jar file or by writing a java program that calls our API. The second submission approach allows user to execute part of the program other than Map-Reduce. It also allows user to do some setup for each Map and Reduce task.
- We schedule map and reduce tasks to exploit locality by trying to assign a task to task tracker with input local to that task tracker.
- Map tasks and reduce tasks can run in parallel in each phase to gain performance.
- Reduce task shuffles the output of map task by external sorting to ensure ordering of reduce output.
- In case of task tracker failure, job tracker is able to recognize failure and reassign the tasks to other task trackers if necessary.

- If a task fails, job tracker will schedule it on other task trackers. If a task fails more than a threshold, the entire job fails.
- We provide a JobClient to start job by jar file, shutdown the system and monitor job progress.
- Our DFS and Map-Reduce facility can read user-defined configuration from configuration file. Default values will be used if properties are missing.

What we have not done:

- Mirror name node and mirror job tracker.
- We only allow one job running at the same time. Job scheduling is on a FIFO policy.

Design

Distributed File System

The DFS can be mainly divided to three pars: name node, data node and client.

Name node

Name node is the coordinator of DFS. It stores the file system meta data, including data nodes, files, blocks corresponding to each file and the location of each block. When a client wants to do operations on files, it first asks name node for the blocks of the file and data nodes which store the block. Then it communicates to data node to perform the operations.

The life cycle of a name node can be described as follows:

On start up, a new instance of Namenode is created either by loading configuration file or from image file written by previous session of Namenode if exists. Then it creates RMI registry on local host and bind itself to that registry. This Namenode object in registry will respond to coming request from client or data node. A data node register itself by calling register method of Namenode. Namenode will assign it an id and keep track of the data node. When user sends read request, name node looks up the metadata about files and respond with the block ids and data nodes containing these blocks. On write request, name node gives three random data nodes on each block to be written. On delete request, name node updates the metadata and keeps a list of delete commands for data nodes. When data node heartbeats to name node, it pulls all delete commands since last heartbeat. Name node also keeps track of the time of last heartbeat of every data node and regard a data node as dead if no heartbeat for a long time. In such case, name node will update the metadata to avoid respond with this data node to client. Name node also periodically write the metadata no an image file for use when it reboot. Finally, when user sends

shutdown to name node, it waits for heartbeat for all alive data nodes for heartbeat to notify them to shutdown. Then it shutdown it self.

Data node

A data node stores blocks of data where size of block is defined by configuration file. After registering to name node, a data node is like a file server that accepts connections from clients for fetching and writing blocks. It periodically sends heartbeat to name node to report its status and pull commands(delete or shutdown) from name node. When shut down command is pulled, a data node writes its metadata to image file on disk and exit the JVM.

Client

The DFS client is an interactive command line interface to DFS. It parses user input, sends request to name node to retrieve meta data and communicate with data node to do operations. The supported commands are:

ls list all files in the DFS

get <file path in DFS> <local destination path> fetch the file from DFS to local disk

put <local file path> <DFS destination path> upload the file to DFS

delete <DFS file path> delete file in DFS

shutdown shut down the entire DFS

quit exit the client CLI

Map-Reduce Facility

Similar to DFS, our Map-Reduce facility has three parts: job tracker, task tracker and client, where job tracker acts as the coordinator of the system, task tracker manages tasks on worker nodes and client sends commands to job tracker to manage the system.

Job Tracker

The job tracker keeps track of all task trackers, jobs, tasks, job progresses, and is responsible for scheduling jobs, creating tasks for a job, assigning tasks to task trackers, reporting progress and reschedule failed tasks.

The communication between job tracker and task tracker is done by RMI calls. On start up, job tracker starts an RMI registry server and bind itself to the registry. Then it waits for task tracker registering and client job submissions. When a job is received, the job tracker fetch the metadata about the blocks of input files and construct map tasks according to the blocks. The map tasks to be run are kept in queue and waits for task tracker heartbeats to pull them. Reduce tasks are also generated but will not run until all map tasks are completed.

There are two types of failures we handled: task tracker failure and task failure. The job tracker has a thread to periodically check the last heartbeat time of all task trackers. If a task tracker times out for a number of counts, it is marked as failed are all map tasks on it should be rescheduled to other task trackers. In case of a task failure, the task will be retried on other task trackers. If a single task fails more than a threshold of times, the entire job is marked as failed.

When all tasks of a job are finished, the job is regarded as finished, and the job tracker starts handle the next queued job. We only run one job at a time. The jobs are scheduled by FIFO.

When client calls shutdown on the job tracker, the job tracker waits for all alive task tracker heartbeats to notify them of shutdown. If all task trackers are notifies, the job tracker exits JVM.

Task Tracker

A task tracker calls register method on job tracker to register itself and periodically heartbeat to job tracker to report finished tasks, failed tasks, available slots and pulls new tasks to be run. A task can be either map task or reduce task.

Map Task:

A map task takes a block on DFS as input, do user-defined setup method, read input by record reader, feed input to map function, partition the output and write output to local disk. Every output file of map task is sorted according to key.

Reduce Task

A reduce task is much complicated than map task. It gets a list of task tracker ids that may have map outputs needed by this reduce task. The reduce task communicated with task tracker listener(explained later) of these task trackers requesting map output files. There may be more than one files on one task tracker, so the task tracker first need to send the number of such files to reduce task. Then for each file, first send the size of file and the content of file using output stream. The files are transferred in terms of byte array in buffer, so they don't need to fit in memory.

After all files are received from all task trackers, the reduce task shuffles them by external sort. This is done by putting the heads of all files into a priority queue. Then pop one record from queue and write to disk and read one record from the file where the popped record belonged to. The pop and read step iterates until all files reach the end and the priority queue is empty.

Next, reduce task do user-defined setup function and read the merged file using record reader and feed key-value pairs to reducer method. After the file reaches the end, we upload the local output file to DFS. Since we did shuffle on map outputs, the reduce output files are ordered by key.

When a task completes, the task will report complete to task tracker. If any exception occurs during the execution, the task will report failure to task tracker. All completed tasks and failed tasks will then be reported to job tracker on heartbeat.

Client

The client create new jobs and submit them by RMI on job tracker. The new job can be created by calling JobClient through command line and specifying parameters like jar file path, delimiter and input path. The new job can also be created by user written java code and API calls. The mapper and reducer classes can be set by class or jar file. The map task and reduce task will load the jar file and look for class that extends mapper and reducer.

Easy Way to Run Our System

We provide several scripts to make TA's testing easier:

To run steps including compiling, starting the system, uploading input files, running examples and stopping the system in a **single** command:

```
cd bin; ./run-all.sh
```

Or run the following commands step by step

To compile the source code:

```
cd src
```

```
make
```

To deploy the system

```
cd bin
```

```
./start-all.sh
```

To upload input files for examples to DFS:

```
./putInput.sh
```

To run example 1(Degree Count):

```
./run_example1.sh
```

To run example 2(Word Count):

```
./run_example2.sh
```

To stop the system:

```
./stop-all.sh
```

Example1:

Description

In this example we write a program to count the degrees of nodes in a directed graph. In the map phase, each mapper reads set of edges from input file and outputs two records (u, 1), (v, 1) for each edge (u, v). In reduce phase, reducer reads records with the same node id and accumulates the values to get the degree.

Input

We manually construct 6 small graphs each with 4 to 6 edges. The graph is stored in format of edge list, therefore every line in the file contains two ids u and v, which represents a edge (u, v) in the graph.

Output

Every line in the output file contains two numbers id and count, which represents the id of a node and its degree respectively.

How to Run This Example

In this example, we only write mapper and reducer, so we run this example by using the command line interface of JobClient.

First we need to change the working directory to bin. Assuming the input files have been uploaded to /graphs in DFS(this can be done using scripts in previous section) and the delimiter used in input file is comma, we then type the following command to run this example:

```
java -cp . mapredClient.Client -jar DegreeCount.jar -input /
graphs -output /output-degree -reducenum 2 -delim ,
```

After typing this command in terminal, the job will be submitted to MapReduce system to execute. Next client will report the progress of job periodically until it finishes.

The output file will be in DFS under directory /output-degree. To fetch the file to local disk, type

```
java -cp . dfsClient.Client <name node host> <name node port>
get /output-degree/part0 ./DCpart0
get /output-degree/part1 ./DCpart1
```

The output is now in ./DCpart0 and ./DCpart1

Example2:

Description

In this example we write a program to count the number of occurrence of non-stop words in the given texts. In the map phase, each mapper will load the dictionary of stop words in the setup method and then reads lines from input file to do the map operations. In reduce phase, reducer reads records with the same word as the key and accumulates the values to get the number of occurrence.

Input

We use the report for this project as input file.

Output

Every line in the output file contains a word and its number of occurrence.

How to Run This Example

In this example, we not only write mapper and reducer, but also write a main method which uses the API provide by Job and JobClient to set and submit job. Therefore, we only need to execute this java program by typing the following command to run the example (assuming the input files have been uploaded to /WCDocuments in dfs, this can be done using scripts in previous section)

```
java -cp . example2.WordCount /WCDocuments /output-wc
```

The output file will be in DFS under directory /output-wc. To fetch the file to local disk, type

```
java -cp . dfsClient.Client <name node host> <name node port>
get /output-wc/part0 ./wcpart0
```

The output is now in ./wcpart0