Group 17: Ruochen Shen (rs20), Jun Luo (junluo2)

MP4 Report

Design

In this MP, we design a Hadoop-like MapReduce framework, which allows users to execute MapReduce tasks based on the distributed file system made in MP3. We offer two types of tasks, maple and juice, which are similar to the map and reduce in Hadoop.

Architecture

we have several roles in this MP:

1. **dcli**: a binary that handles user input. We can use this dcli to trigger a maple or juice task
2. **RM**: dispatch tasks to workers and supervise all tasks, re-dispatch tasks in case of failures
3. **Worker**: execute maple or juice tasks

Workflow

We have a command handler - *dcli*, to process the command input by users from the command line. It parses the command, wraps the command and related arguments and then sends wrapped the result to the local *dnode* process. The *dnode* process is our core process which has the membership list for failure detector (MP2) and the file list for the distributed file system (MP3)

After the local *dnode* process receives the wrapped user command, it redirects the command to the RM node, which has the smallest id.

After receiving the user command, the RM put the command into a queue. There is a dedicated goroutine that pulls command from the queue and to start a new maple or juice task.

To start a maple task, the RM first gets all filenames under an SDFS folder, gets alive *dnode* on the system based on the membership list and then assigns different filenames to different *dnode*. After this step, the RM dispatches maple tasks to the corresponding *dnode* worker.

To execute a maple task, a worker retrieves all files and executable file it needs from SDFS, get maple function from executable file and feed the maple function with 10 lines at a time. Each output from the maple function will be stored in local file systems, different files will be created for different keys. After all the files are processed, each key file will be put to the corresponding file master. But these key files are just intermediate. Because for one key, the file master may receive many different parts from different workers. These different parts will be merge after all workers finish their jobs.

With all the completion of all workers' tasks, the RM will ask all file masters to merge all intermediate files. Then Maple is done.

The Juice process is similar, the RM dispatches tasks to Juice workers, the workers fetch files from SDFS and execute the Juice function with files got from SDFS, send intermediate files to file master. RM will send merging file signals to all file masters after all Juice tasks complete.

Programming framework

We use go plugin package to dynamically load custom Maple or Juice function. We use a certain signature for Maple and Juice function:

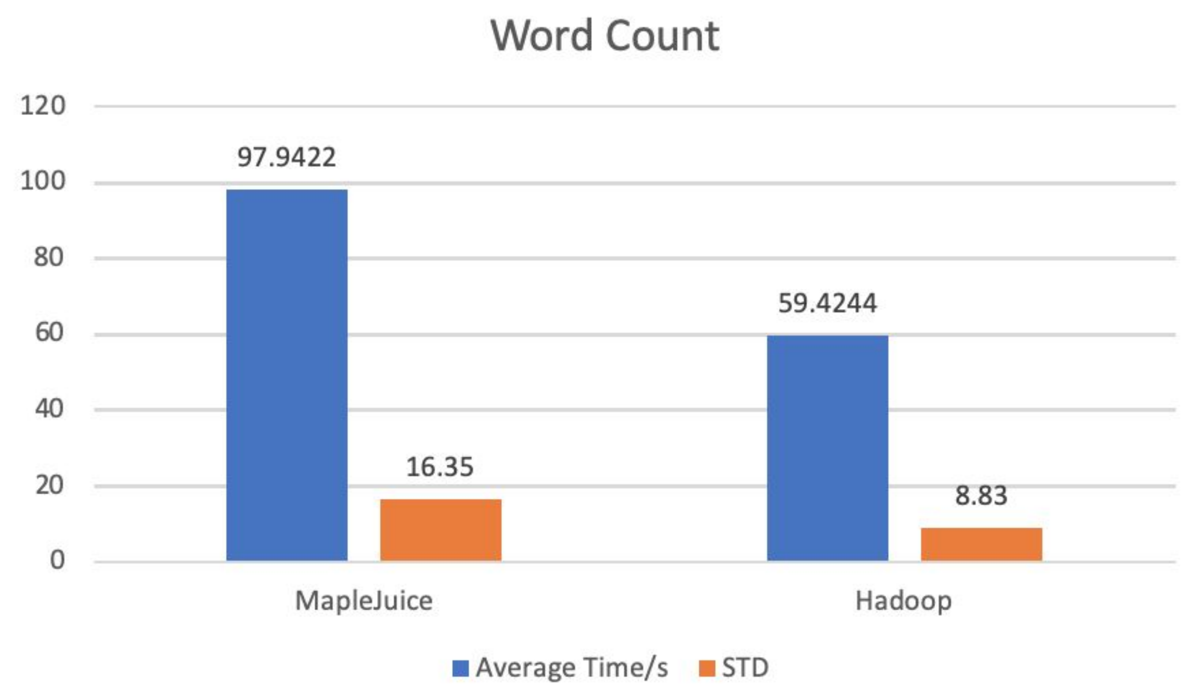
```
func Maple(lines []string) map[string]string

func Juice(key string, lines []string) map[string]string
```

After a user writes their Maple or Juice functions, they can use go build -buildmode=plugin to build a so file and put the so file to SDFS. When they run a task, our framework will load the so file from SDFS and extract the Maple or Juice function and feed the function with lines.

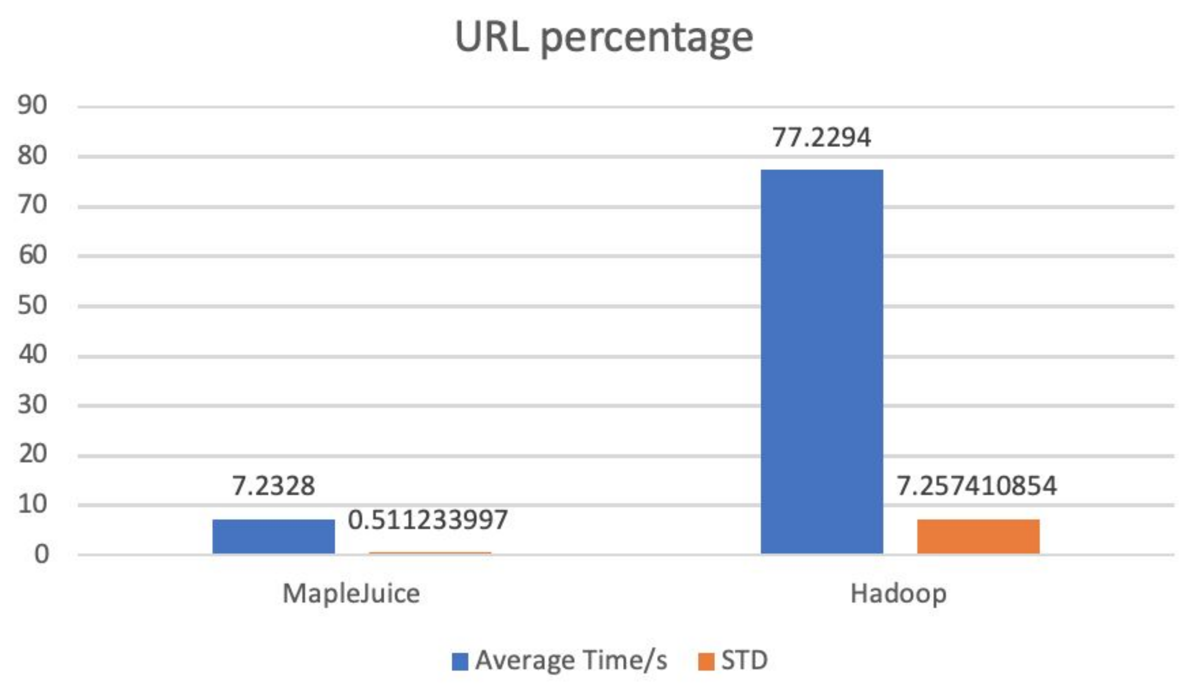
Plot

Word Count



As the above fig show, our MapleJuice for word count task is slower than Hadoop, the result is expected because there are many small files generated. It takes a long time to put thousands of small files into the SDFS.

URL percentage



As the above fig shows, our MapleJuice for URL percentage App is much fast than Hadoop. And the result is expected. In this application, the dataset is pretty small, only around 9000+ URL, so there are only a few intermediate files that need to be put into SDFS. So our MapleJuice is quite fast for this application.

The reason why Hadoop is slower is that Hadoop takes a lot of time to set up. And we need to chain two MapReduce together to complete this task, and each MapReduce takes 30+ second to set up, so the overall time is much longer than our MapleJuice.