Wadoop Design and Implementation Report

Author: Guanjie Chen (guanjiec)
Wenhan Lu (wenhanl)

1. Introduction

Wadoop is a fault-tolerant distributed framework supporting running concurrent MapReduce job on it. Every Wadoop cluster consist of two type of nodes: Master and Slave. There are two layers inside each node. A distributed file system we called “WHFS” is the first layer which is responsible for store file distributedly. The second layer is a MapReduce programming framework which allow user to run MapReduce job on it.

The first layer inside Master is called “NameNode” and in Slave called “DataNode” (same notation with HDFS). In second layer, it is called “JobTracker” for Master and “TaskTracker” for Slave. (Also same notation with Hadoop)

2. WHFS

2.1 Design description

WHFS is the underlying distributed file system that supports Wadoop Mapreduce framework. Its basic infrastructure includes one NameNode and several DataNodes.

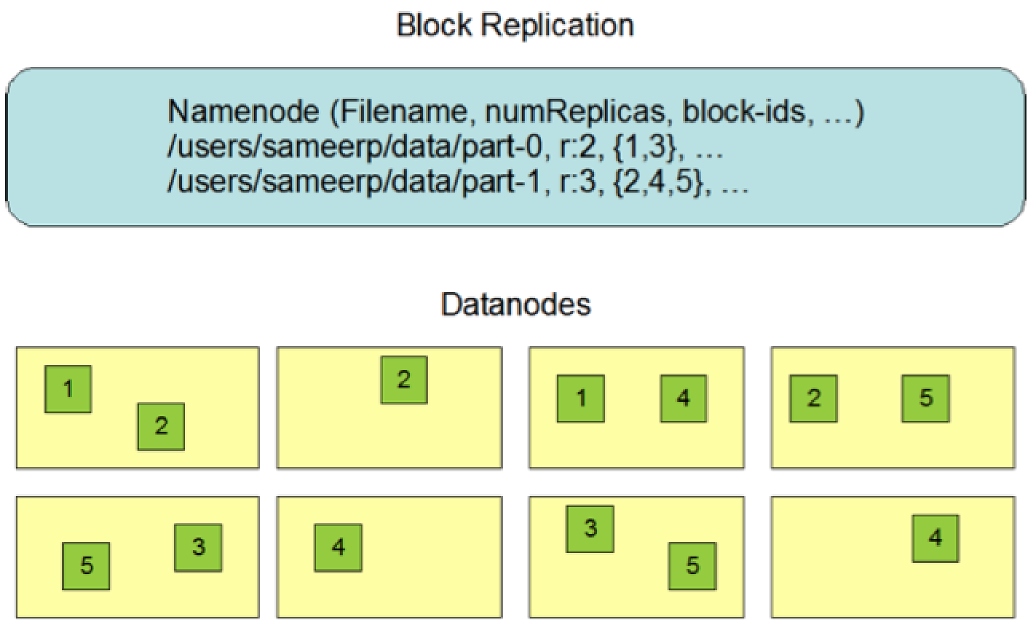
Block Replication
Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

Figure 1. WHFS archtecture

DataNodes store data blocks and in order to be more fault-tolerant, usually each block is replicated in configurable number of machines (3 replica by default, must be no larger than number of Slaves in cluster).

NameNode stores metadata for all DataNodes, including mapping data blocks to DataNodes, replica registry of each block, liveliness of DataNodes. NameNode should also maintains FS namespace operations like opening, closing, renaming files and directories. But for simplicity in this project, we do not involve directory structure inside WHFS. All files imported to WHFS is like under a same base directory, and files are uniquely identified by file name.

DataNodes periodically send heartbeat to NameNode informing its availability and data block information. Upon failure, NameNode will realize failed node by not receiving heartbeat, and replicate missing data from other replica to other available nodes.

Assumptions for simplicity:

- Data blocks are separated by lines instead of bytes like HDFS
- Data blocks are small and each MapReduce splits will take integer number of blocks.
- No directory structure. All files is like under a same base directory uniquely identified by filename.

2.2 Fault tolerance

Fault tolerance mainly realized by replication. When some DataNode dies, its data blocks are usually replicated in other nodes.

In our design, data blocks are replicated in pack on entire machine. That is to say, data blocks originally stored in DataNode A are entirely replicated in other DataNodes like B and C. In this design, our mapper task can be rerun on exactly one node instead of several after a node failure. In this design, our system can tolerate NUM_REPLICA - 1 node failure in worst case. NUM_REPLICA is normally smaller than total number of DataNodes, otherwise replication is meaningless.

2.3 Rebalancing

After DataNode failure, there are some data blocks which have fewer replica than configured NUM_REPLICA. And when a DataNode newly connected or recovered from failure or network partition, it is a waste if no data blocks are known by NameNode in this node.

In both cases, rebalancing is a better design decision. Rebalancing after failure makes all blocks have NUM_REPLICA replications, which make system more tolerant to further failure. Rebalancing after node recovery makes data more evenly distributed. Rebalancing is not supported in this version of systm, but surely it is included in our future work.

3. MapReduce framework

3.1 Design description

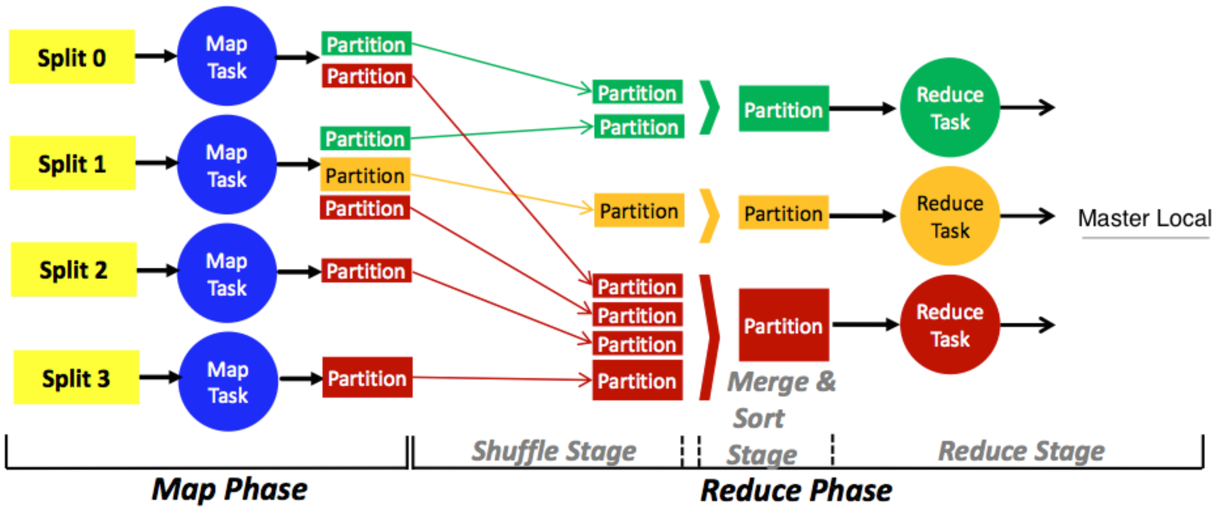

Figure 2. Wadoop Mapreduce workflow

As shown in figure 2, when a mapreduce job starts, the JobTracker first separate job to mapper tasks and reducer tasks and assign tasks to TaskTrackers. It consists of three steps:

- **Mapper step:** Each slave applies the "map()" function to the local data, and writes the output to a temporary storage.
- **Shuffle step:** Slave nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same reducer.
- **Reduce step:** Slave nodes now process each group of output data, per key, in parallel.

Assumption for simplicity:

- All the MapReduce job must use row number as the first parameter of Mapper(which means we split file by line. And In many cases this variable is never used just like the hadoop).
- Each slave node has one Mapper for each job and also one reducer for this job (mapper slot and reducer slot are one).
- We assume no failure during the data transmission between datanodes (We use reliable transfer protocol TCP).
- NUM_MAPPER and NUM_REDUCER are not configurable, they are number of slave nodes by default, one mapper and one reducer per machine.

3.2 Concurrent use support
Our facility support concurrency. There are two type of concurrency which we've considered. First for a single job, there can be many mappers and reducers across many datanode the run concurrently. For reduce phase, it will not start until all the mappers complete and report to the Jobtracker. This is mainly achieved by using thread-safe data structure like java.io.ConcurrentHashMap to prevent concurrent write to critical resource.

Second, we can enable running many jobs concurrently. We assign a unique job id for each job. And each task in the system also has a distinct task id. The jobtracker use these id to identify a task and dispatch it. So our system can support running many jobs at the same time without the confliction.

Finally, low level network facility under this system is also important to support concurrency. We developed our own net/ package based on Java Non-blocking I/O (NIO). This is a well-encapsulated network package which makes our system good in performance and robust in race conditions.

3.3 Scheduling strategy
After you import a file to whfs, it will be splitted and store in different nodes. Our mapreduce job pay a lot of attention to data locality to avoid heavy intermediate data transmission. We assign mapper task to every datanode that has the splitted file of the original file. And the tasktracker will launch a mapper task when received a message from the jobtracker. For example, a large file is splitted and distributedly store in 100 nodes. Then the jobtracker will assign mapper task to all these 100 datanodes. And all the mapper and reducer tasks are schedule one time.

But the same design does not apply to the reduce phase. After all the mapper task of a certain job complete, the jobtracker will dispatch reduce job of which the number is configured by the users. And when a tasktracker receive a reduce task it will first send a partition message to all the datanode which to ask for the files it needs and the reduce task in this node will start after all the required files are reveived successfully.

3.4 Failure and recovery
We periodically ping every datanode and update the amount of non-response time. If a certain node does not respond in a certain amount of time, we will assume that this datanode encounters failure(disconnected from NameNode or accidentally shut down). Then we will remove this node from the datanode list.

The recovery part is a little tricky, if the failed datanode is running some mapper and reducer task, we definitely should reassign these tasks. The way we do that is we rerun all the job from beginning to simplify our design. So we do not need find whether the failure happens in map phase or the reduce phase. The jobtracker keeps a record of all the jobs and if a failure happens all the running jobs will restart again.

4. Limitation and future work

This system works well in general, but still have several places to improve. In WHFS level, rebalancing should be supported to tolerate more node failure. And now node hostnames are configured statically, which requires user to start cluster carefully in order with what is in configuration. Hostname information should be maintained dynamically and update from master to all slaves in future work.

Furthermore, now machines are not used efficiently because no multi-thread used in slave nodes, which means each node only run one task at a time. In future it should support MAPPER_SLOT and REDUCER_SLOT and make NUM_MAPPER and NUM_REDUCER configurable. These will use machines more efficiently and provide more feasibility to our users.

5. Appendix: Source Code Structure

/config

This package contains only one file Config.java. You should properly set the variables in this file before you compile and run our program. You can see setting guide of all variable in our administrator document.

/example

This package contains the examples of our facility and you can check the existing the wordcount example and the user api and the administrator guide we provide you to learn how to write a mapreduce program.

/mapr

This package implements all the function of the mapreduce process.

- MPCoordinator is the central coordinator the tasks which run in the facility. The coordinator will receive message and get the tasks needed to assign from the jobtracker and schedule the tasks to each datanode. It is also responsible for receive and process the update messages such as a task is completed from each data nodes.
- The Mapper and Reducer class is a abstract class which every mapreduce job needs to implement.
- The MapReduceJob also an abstract class which every mapreuce program needs to extend and implement the mapper and reducer in it.
- Record is the data type we use in our facility
- The Task represents a mapper or reduce task of our program. The MapperTask and ReducerTask extends the Task class which represents a task of map and reduce. The MPCoordinator will distinguish what type a specific task is and dispatch it.

- The MapTaskProcessor and ReduceTaskProcessor are the processor of the map and reduce phase. After the datanode receive a task, it will launch a mapper or reducer processor to do the map and redcue. And the processor also is responsible for informing MPCoordinator that a certain map or reduce task has completed.

/Tracker
This package contains two trackers: JobTracker and TaskTracker

- Jobtracker runs on the the master node. Its job involves receive command from user input and launch a new mapreduce job. We receive from users that a new mapreduce job needs to be run, it will specify the map and reduce task and make a new mapreduce job and then send message to the MPCoordinator.
- TaskTracker run on the datanode. And its job is to run a task when receive message from the coordinator and ensure the mapper and the reducer have the required files to run. Before the reduce starts, the task tracker will ask for all the requested files from other datanodes.

/msg
This package contains all the message type we use in our program.

- MPMessageManager is responsible for send and receive a messgae.
- MPPartitionMessage is the message that the reducer will use to ask for the required files from other datanodes.
- MPTaskMessage is the message which send by jobtracker to tasktracker to start a new map or reduce task.
- TaskUpdateMessage is the message to inform coordinator that a certain task is completed or not.

/whfs
Distributed file system facility package for MapReduce framework.

/net
Low level network facility for this system. Built based on Java NIO, it can be used by a easy interface without knowing how it is implemented. (Good in teamwork) This package is unit-tested in TestServer.java and TestClient.java

Master
The master node class.

Slave
The slave node class.

Console
Receive input from users and send to jobtracker and namenode.