Hadoop MapReduce Framework
Contents

- Hadoop MapReduce Framework Architecture
- Interaction Diagram of MapReduce Framework (Hadoop 1.0)
- Interaction Diagram of MapReduce Framework (Hadoop 2.0)
Motivation to Distributed SPMD
Hadoop MapReduce History

- Originally architected at Yahoo in 2008
- “Alpha” in Hadoop 2 pre-GA
  - Included in CDH4
- Yarn promoted to Apache Hadoop sub-project
  - Summer 2013
- “Production ready” in Hadoop 2 GA
  - Included in CDH5 (Beta in Oct 2013)
Master-Slave Architecture

- Master Node (Hadoop NameNode):
  - Manage resources of Hadoop cluster
  - Monitor execution of the MapReduce job

- Slave Nodes (Hadoop DataNode):
  - TaskTracker: Receives the task from Job Tracker
  - TaskTracker: Runs the task until completion

- Job Tracker:
  - Receives the task from the Job Tracker
  - Runs the task until completion
Distributed Batch-Sequential Architecture

Input: \(<k_1, v_1>\)

map \rightarrow \langle k_2, v_2 \rangle

combine * \rightarrow \langle k_2, v_2 \rangle

reduce \rightarrow \text{Output: } \langle k_3, v_3 \rangle

Job Tracker

- **Job Tracker is the master node (runs with the namenode)**
  - Receives the user’s job
  - Decides on how many tasks will run (number of mappers)
  - Decides on where to run each mapper (concept of locality)

- This file has 5 Blocks $\rightarrow$ run 5 map tasks
- Where to run the task reading block “1”
  - *Try to run it on Node 1 or Node 3*
Task Tracker

- **Task Tracker is the slave node (runs on each datanode)**
  - Receives the task from Job Tracker
  - Runs the task until completion (either map or reduce task)
  - Always in communication with the Job Tracker reporting progress

In this example, 1 map-reduce job consists of 4 map tasks and 3 reduce tasks
Key-Value Pairs

- Mappers and Reducers are users’ code (provided functions)
- Just need to obey the Key-Value pairs interface

**Mappers:**
- Consume `<key, value>` pairs
- Produce `<key, value>` pairs

**Reducers:**
- Consume `<key, <list of values>>`
- Produce `<key, value>`

**Shuffling and Sorting:**
- Hidden phase between mappers and reducers
- Groups all similar keys from all mappers, sorts and passes them to a certain reducer in the form of `<key, <list of values>>`
Deciding on what will be the **key** and what will be the **value**

⇒ **developer’s responsibility**
Example 1: Word Count

Job: Count the occurrences of each word in a data set
How does it work?

Interaction Diagram of MapReduce Framework

Hadoop 1.0
Anatomy of Running a Job in Hadoop MapReduce V1

- Job Submission
- Job Initialization
- Task Assignment
- Task Execution--Streaming and Pipes
- Progress and Status Updates
- Job Completion
How Does MapReduce Work?

A job run in classic MapReduce has four independent entities at the highest level:

- **The client**
  - Submits the MapReduce

- **The jobtracker**
  - Coordinates the job run.
    - A Java application
    - Main class is `JobTracker`

- **The tasktrackers**
  - Run the tasks that the job has been split into
    - Java applications
    - Main classes are TaskTracker

- **The distributed filesystem**
How Hadoop runs a MapReduce Job?
Step 1:
The `runJob()` method on `JobClient` is a convenience method that creates a new `JobClient` instance and calls `submitJob()` on it.
Job Submission

Step 2: Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker).
Step 3:
Copies the resources needed to run the job, including the **job JAR file**, the **configuration file** and the **computed input splits**, to the jobtracker’s filesystem in a directory named after the job ID.
Job Submission

Step 4: Tells the **jobtracker** that the job is ready for execution (by calling `submitJob()` on `JobTracker()`).
Job Initialization

Step 5: Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks’ status and progress.
Step 6:
To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the JobClient from the shared filesystem.
Task Assignment

Step 7:
1. Tasktrackers run a simple loop that **periodically sends heartbeat method** calls to the jobtracker.
2. Heartbeats tell the jobtracker that a tasktracker is alive, but they also double **as a channel for messages**.
3. The tasktracker is a part of the heartbeat. A tasktracker will **indicate whether itself is ready to run a new task**. If true, the jobtracker will allocate a task to the tasktracker.
Task Execution

Step 8:
**TaskTracker localizes** the job JAR by copying it from the shared filesystem to the tasktracker’s filesystem.

**TaskTracker also copies** any files needed from the distributed cache by the application to the local disk.
Task Execution

Step 9: **TaskRunner** launches a new Java Virtual Machine.
Task Execution

Step 10: Run each task
Issues of MapReduce Framework

- Streaming and pipes
- Progress and Status Updates
- Failures
- Job Scheduling
- Shuffle and sort
- Task Execution
Streaming and pipes

- Both Streaming and Pipes run special map and reduce tasks for the purpose of the user-supplied executable and communicating with it.
In both streaming and pipes cases, during execution of the task, the Java process input key-value to the external process.
Streaming and Pipes

The external process runs the input key/values through the user-defined map or reduce function.
Streaming and Pipes

The external process runs the input key/values through the user-defined map or reduce function.
After processing, the external process passes the output – key-value pairs task back to the Java process.
Streaming and Pipes

In the case of Streaming, the Streaming task communication with the process using standard input and output streams. The process may be written in any language.

In the case of Pipes, the Pipes task listens on a socket and passes the C++ process a port number in its environment so that on startup, the C++ process can establish a persistence socket connection back to the parent Java Pipes task.
Progress and Status Updates

- **Status**
  - The state of the job or task (e.g., running, successfully completed)
  - The progress of maps and reduces
  - The values of the job’s counters
  - Status message or description (which may be set by user code)
Progress and Status Updates

- **Progress**
  - Reading an input record (in a mapper or reducer)
  - Writing an output record (in a mapper or reducer)
  - Setting the status description on a reporter (using Reporter’s `setStatus()` method)
  - Incrementing a counter (using Reporter’s `incrCounter()` method)
  - Calling Reporter’s `progress()` method

- When a task is running, it keeps track of its **progress**, that is, **the proportion of the task** completed
  - For map tasks, this is the proportion of the input that has been processed
  - For reduce tasks, it’s a little more complex, but the system can still estimate the proportion of the reduce input processed. It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle
Job Completion

- When the jobtracker receives a notification that the last task for a job is complete, it changes the status for the job to “successful.” Then, when the JobClient polls for status, it learns that the job has completed successfully, so it prints a message to tell the user, and then returns from the runJob() method.

- The jobtracker also sends a HTTP job notification if it is configured to do so.

- Last, the jobtracker cleans up its working state for the job, and instructs tasktrackers to do the same (so intermediate output is deleted, for example).
In the real world, user code is buggy, processes crash, and machines fail. One of the major benefits of using Hadoop is its ability to handle such failures and allow your job to complete.

- Task Failure
- Tasktracker Failure
- Jobtracker Failure
Consider first the case of the child task failing. The most common way that this happens is when user code in the map or reduce task throws a runtime exception.

- Child JVM will report to parent tasktracker, before exit
- Streaming task, exit with nonzero exit code
- Hanging task (update timeout)
- Inform Jobtracker by heartbeat
Task Failure

- The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails more than four times, it will not be retried further.

- A task attempt may also be killed
  - Speculative duplicate

- For some applications it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures.
Tasktracker Failure

- If a tasktracker fails by crashing, or running very slowly, it **will stop sending heartbeats** to the jobtracker (or send them very infrequently).

- The jobtracker will notice a tasktracker that has stopped sending heartbeats (if it hasn’t received one for 10 minutes, configured via the mapred.tasktracker.expiry.interval property, in milliseconds) and **remove it from its pool of tasktrackers** to schedule tasks on.

- A tasktracker can also **be blacklisted** by the jobtracker, even if the tasktracker has not failed.
  - It is running failed frequently.
Jobtracker Failure

- Failure of the jobtracker is the most serious failure mode. Currently, Hadoop has **no mechanism for dealing with failure of the jobtracker**
  - But in the future Hadoop may try to run several jobtrackers to deal with it.
Early versions of Hadoop had a very simple approach to scheduling users’ jobs: they run in order of submission, using a **FIFO scheduler**.

Later on, the ability to set a job’s priority was added, via the mapred.job.priority property or the setJobPriority() method on JobClient.

There also a **multi-user scheduler called The Fair Scheduler**.

- Give every user a fair share of the cluster capacity over time.
- Support preemption.
- Enable it, place its JAR file on Hadoop’s classpath, by copying it from Hadoop’s contrib/fairscheduler directory to lib directory, then set its mapred.jobtracker.taskscheduler property to org.apache.hadoop.mapred.fairscheduler.
Shuffle and sort in MapReduce
The Map Side
The map function gets the input split from HDFS and produces output.
Each map task has a circular memory buffer. And the map task writes the output to the memory buffer.

The buffer is 100MB by default. When the contents of the buffer reach a certain threshold size (the default value 0.8 or 80%), a background thread will start to spill the contents to disk.
When the contents of the buffer reaches a certain threshold size a background thread will start to spill the contents to disk.

Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort.
Before the task is finished, the spill files are merged into a single partitioned and sorted output file.
The Reduce Side

Diagram showing the process flow of reduce tasks, including map, copy, sort, and reduce phases, with interactions between maps and reduces.
The reduce task starts copying their outputs as soon as each completes. This is known as the **copy phase** of the reduce task.
When all the map outputs have been copied, the reduce task moves into the sort phase (which should properly be called the merge phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering.
The Reduce Side
-Reduce Phase

During the reduce phase, the reduce function is invoked for each key in the sorted output.
The output of this phase is written directly to the output filesystem, typically HDFS. In the case of HDFS, because the node manager is also running a datanode, the first block replica will be written to the local disk.
Task Execution

- Speculative Execution
- Task JVM Reuse
- Skipping Bad Records
- The Task Execution
  - Environment Streaming environment variables
  - Task side-effect files
Speculative Execution

- Job execution time sensitive to slow-running tasks, as it takes only one slow task to make the whole job take significantly longer than it would have done otherwise.

- Hardware degradation, or software mis-configuration may be hard to detect since the tasks still complete successfully, albeit after a longer time than expected. Hadoop doesn’t try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another, equivalent, task as a backup. This is termed speculative execution of tasks.

- A speculative task is launched only after all the tasks for a job have been launched, and then only for tasks that have been running for some time (at least a minute), and have failed to make as much progress, on average, as the other tasks from the job.

- Speculative execution is an optimization, not a feature to make jobs run more reliably.
Task JVM Reuse

- The overhead of starting a new JVM for each task can take around a second, which for jobs that run for a minute or so is insignificant. However, jobs that have a large number of very short-lived tasks (these are usually map tasks) or that have lengthy initialization, can see performance gains when the JVM is reused for subsequent tasks.

- With task JVM reuse enabled, tasks do not run concurrently in a single JVM.

- Tasks that are CPU-bound may also benefit from task JVM reuse by taking advantage of runtime optimizations applied by the HotSpot JVM.
If only a small percentage of records are affected, then skipping them may not significantly affect the result. However, if a task trips up when it encounters a bad record—by throwing a runtime exception—then the task fails. Failing tasks are retried (since the failure may be due to hardware failure or some other reason outside the task’s control), but if a task fails four times, then the whole job is marked as failed.

The best way to handle corrupt records is in your mapper or reducer code. You can detect the bad record and ignore it, or you can abort the job by throwing an exception.

In rare cases, though, you can’t handle the problem because there is a bug in a third-party library that you can’t work around in your mapper or reducer. In these cases, you can use Hadoop’s optional skipping mode for automatically skipping bad records (after twice failures).
How does it work?

Interaction Diagram of MapReduce Framework

Hadoop 2.0
Hadoop MapReduce Version 2

- MapReduce version 2 uses YARN
  - Hadoop includes a MapReduce Application (MRAppMaster) to manage MR jobs
  - Each MapReduce job is a new instance of an application

https://www.slideshare.net/cloudera/introduction-to-yarn-and-mapreduce-2
The MapReduce framework consists of:
- A single master ResourceManager
- One slave NodeManager per cluster-node
- MRAppMaster per application

https://hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/YARN.html
How Hadoop MapReduce V2 Works?

The whole process of Hadoop MapReduce running a job includes five independent entities:

- The client
- The Yarn resource manager
- The Yarn node managers
- The MapReduce application master (MRAppMaster abbr)
Anatomy of Running a MapReduce2 Job
The client submits the MapReduce job.
Anatomy of Running a MapReduce2 Job

The Yarn resource manager coordinates the allocation of compute resources on the cluster.
The MapReduce application master coordinates the tasks running the MapReduce job.
Anatomy of Running a MapReduce2 Job

The distributed filesystem is used for sharing job files between the other entities.
Step 1: The `submit()` method on `Job` creates an internal `JobSubmitter` instance and calls `submitJobInternal()` on it. `JobSubmitter` checks the output specification of the job. `JobSubmitter` computes the input splits for the job.
Anatomy of Running a MapReduce2 Job

Step 2: The JobSubmitter() asks the resource manager for a new application ID. This ID is used for the MapReduce job ID.
Step 3: The JobSubmitter() copies the necessary resources to run the job. The sources including the job JAR file, the configuration file and the computed input splits.
Anatomy of Running a MapReduce2 Job

Step 4: The JobSubmitter submits the job by calling submitApplication() on the resource manager.
Anatomy of Running a MapReduce2 Job

Step 5: When the resource manager receives a call to its `submitApplication()` method, it hands off the request to the YARN scheduler.

5a: The scheduler allocates a container.

5b: The resource manager then launches the application master’s process there.
Anatomy of Running a MapReduce2 Job

Step 6: The application master for MapReduce jobs is a Java application whose main class is **MRAppMaster**.
The MRAppAMaster initializes the job by creating a number of bookkeeping objects to keep track of the job’s progress.
The MRAppAMaster will receive progress and completion reports from the tasks.
Anatomy of Running a MapReduce2 Job

Step 7: The MRAppMaster retrieves the input splits computed in the client from the shared Filesystem. Then the MRAppMaster creates a map task object for each split and a number of reduce task objects.
Anatomy of Running a MapReduce2 Job

Step 8: If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager.
Step 9: Once the resource manager’s scheduler assign resources to a task for a container on a particular node.
(9a and 9b) The application master starts the container by contacting the node manager.
Anatomy of Running a MapReduce2 Job

Step 10: The task mentioned at the step 9 is executed by a Java application whose main class is YarnChild. The YarnChild localizes the resources that the task needs. The resources include the job configuration and JAR file, and any files from the distributed cache.
Anatomy of Running a MapReduce2 Job

Step11: YarnChild runs the map or reduce task
Thank You!!

Any Questions??