PART 1.
LARGE SCALE DATA ANALYSIS
USING MAPREDUCE

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics
- FAQs
- Anatomy of a MapReduce

Job submission process in Classic framework
- Asks the jobtracker for a new jobID
  - Step 2
- Checks the output specification of the job
- Computes the input splits for the job
- Copies the resources to the jobtracker’s filesystem in a directory named after the jobID
  - The job JAR file, the configuration file, and the computed input splits
- Tells the jobtracker that the job is ready for execution by calling submitJob() on JobTracker
  - Step 4

Job Initialization in the Classic mode
- JobTracker puts the job into an internal queue
  - Creating an object to represent the job being run
  - Bookkeeping information to keep track of the status and progress of its tasks
- Creating the list of tasks
  - The job scheduler retrieves the input splits
  - The job scheduler creates a map task for each split
  - The number of reduce tasks to create
    - mapred.reduce.tasks property in the Job
    - setNumReduceTasks()
  - A job setup task and a job cleanup task are created

Task assignment
- Jobtracker select a job to run
  - Default algorithm
  - Based on the priority list of jobs
- Tasktrackers have a fixed number of slots for map tasks and reduce tasks
  - These are set independently
  - These are selected based on the number of cores and the memory
- The default scheduler fills empty map task slots first
  - Before it fills the reduce task slots
  - Scheduling the reduce task does not need to consider data locality
- Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker
Task execution

- Tasktracker
  - Copies the job jar to the tasktracker’s file system
  - Any files needed from the distributed cache to the local file system
  - Creates a local working directory
  - Creates instance of TaskRunner

- TaskRunner
  - Launches a new Java Virtual Machine to run each task
  - Any failed map or reduce does not affect the tasktracker

Streaming and Pipe

- Runs special map and reduce tasks
  - To launch the user supplied executable
  - To communicate with it

- Streaming task communicates with the process using standard input and output streams

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper /bin/cat \
    -reducer /bin/wc
```

Hadoop Pipes task

- Allows C++ application code to be used in MapReduce program through Pipes

- Passes the C++ process a port number in its environment via socket
  - On startup, the C++ process establish a persistent socket connection back to the parent Java Pipes task

Progress and status updates

- A Job and each of its tasks have a status
  - State of the job or task
    - E.g. running, successfully complete, failed
  - The progress of maps and reduces
  - The values of the job’s counters
  - A status message or description
Progress and status updates [2/4]

- Progress of a task
  - The proportion of the task completed
- Map task
  - The proportion of the input that has been processed
- Reduce task
  - Divides the total progress into 3 parts (copy/sort/reduce)
  - If the task has run the reducer on half its input
    - $\frac{1}{3}$ (copy) + $\frac{1}{3}$ (sort) + a half of $\frac{1}{3}$ (reduce phase) = $\frac{5}{6}$

Progress and status updates [3/4]

- Tasks
  - have a set of counters
  - Count various events at the task run
    - e.g. the number of map output records written
  - If a task reports progress
    - It sets a flag to indicate that the status change should be sent to the tasktracker
    - Checked every 3 seconds
- Tasktracker
  - Tasks notify the current task status to the tasktracker
    - if the flag is set
  - Tasktracker sends heartbeats to the jobtracker every 5 seconds (minimum)

Progress and status updates [2/3]

- Jobtracker
  - Combines updates to produce a global view
- Job
  - Receives the latest status by polling the jobtracker every second
  - Prints job statistics and counters to the console

YARN Framework

YARN (MapReduce 2)

- To provide the scalability to MapReduce
  - Splitting responsibility of the jobtracker
    - Scheduling
    - Task progress monitoring
- MapReduce is one type of YARN application

YARN (MapReduce 2)

- Resource manager
  - Manages the use of resources across the cluster
- Node manager
  - Launches and monitors the compute containers on machines in the cluster
- Application master
  - Manages the lifecycle of applications running on the cluster
    - Application master negotiates with the resource manager for cluster resources
      - Number of container and certain memory limit
      - Node managers oversee containers not to use more resources than allocated
### A MapReduce job using YARN

**Job Initialization**

- **submitApplication()**
  - Resource manager will hand off the request to the scheduler
  - The scheduler allocates a container
  - The resource manager then launches the application master’s process in the container

### Job Initialization (1/3)

- Application master
  - The application master for MapReduce jobs
    - MRAppMaster
    - Initializes the job by creating bookkeeping objects
    - To keep track of the job’s progress
  - Retrieves the input splits
    - Creates a map task object for each split
    - Creates reduce task object
    - Mapreduce.job.reduces property

### Job Initialization (2/3)

- Uber task
  - The overhead of allocating and running tasks in new container outweighs the gain to be had if running them in parallel, compared to running them sequentially on one node

### Job Initialization (3/3)

- What is a small job?
  - A small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block
Task assignment [1/2]

- Application Master requests container for all the map and reduce tasks in the job
  - From the resource manager (Step 8)
- All the requests includes information about each map tasks' locality
  - Host and corresponding racks that the input split resides on
- Scheduler attempts to place tasks on data-local nodes in the ideal case
  - If it is not possible, the scheduler prefers rack-local placement
  - Job is running on a node in the same rack

Task assignment [2/2]

- Requests specify required memory
  - 1024MB (by default)
  - This is configurable
    - mapreduce.map.memory.mb
    - mapreduce.reduce.memory.mb
- In YARN, resources are managed more fine-grained
  - Applications may request a memory capability that is anywhere between the minimum allocation and a maximum allocation
    - yarn.scheduler.capacity.minimum-allocation-mb
    - Default minimum: 1024MB
    - yarn.scheduler.capacity.maximum-allocation-mb
    - Default maximum: 10240 MB
  - Tasks can request any memory allocation between 1 and 10GB (default) in multiple of 1GB
    - mapreduce.map.memory.mb and mapreduce.reduce.memory.mb

Task execution

- Application master starts the container by contacting node manager
- The task is executed by YarnChild
- YarnChild runs in a dedicated JVM

Progress and status updates

- Task reports its progress and status back to its application master
  - Every 3 seconds over the umbilical interface
- The client polls the application master every second
  - mapreduce.client.progressmonitor.pollinterval

Failures
Failures in Classic MapReduce

- The child task fails
  - Runtime exception from the user code
  - Child JVM reports the error back to its parent before it exits
  - Written in the user logs
  - Tasktracker marks the task attempt as failed
  - Frees a slot to run another task
  - Sudden exit of the child JVM
  - Tasktracker notices that the process has exited and marks the attempt as failed

- Hanging tasks
  - If there is no progress update for a while
  - Mark the task as failed
  - Timeout period is normally 10 minutes
  - `mapred.task.timeout`

Tasktracker failure in Classic MR

- Tasktracker stops sending heartbeats
  - Jobtracker will notice if it hasn’t received one for 10 minutes (configurable)
  - Remove it from the pool of tasktrackers
  - Jobtracker arranges tasks including the completed jobs
  - Because the output may not be accessible
  - Tasktracker can also be blacklisted if more than four tasks from the same job fail (set by `mapred.max.tracker.failures`)
  - Blacklisted tasktrackers are not assigned tasks.
  - Until faults expire

Jobtracker failure in Classic MR

- The most serious failure mode
  - Hadoop has no mechanism for dealing with jobtracker failure
  - Single point of failure
  - All running jobs fail

- After restarting a jobtracker
  - Job should be resubmitted

  - This is improved with YARN

Task failure in YARN

- Failure of the running task is similar to the classic case
  - Runtime exception and sudden exit of the JVM are propagated back to the application master
  - The task attempt is marked as failed
  - Hanging tasks are noticed by the application manager by the absence of a ping over the umbilical channel

Application master failure in YARN

- No heartbeats to the resource manager from the application master
  - The resource manager will detect the failure and start a new instance of the master running in a new container
  - All tasks will be rerun (default)
  - Recovery can be enabled

- Client will access resource manager to get the new address of the application master

Node manager failure in YARN

- Resource manager will stop getting heartbeats
  - Remove the failed node manager from the pool of available nodes

- Any task or application master running on the failed node manager will be recovered
Resource manager failure
- After a crash and a new resource manager instance is brought up (by administrator)
  - It recovers from the saved state
    - Check points saved in the persistent storage
    - Non-completed jobs are included

Hadoop Combiner
- Minimizes the data transferred between map and reduce tasks
- Users can specify a combiner function
  - To be run on the map output
  - To replace the map output with the combiner output
- Hadoop does not guarantee how many times it will call combiner for a particular map output record

Example: Find the maximum temperature
- First map produces
  - (1950, 0)
  - (1950, 20)
  - (1950, 10)
  - \(\Rightarrow (1950, 20)\)
- Second map produces
  - (1950, 25)
  - (1950, 15)
  - \(\Rightarrow (1950, 25)\)
- Input to the reduce function
  - (1950, [0, 20, 10, 25, 15])
- Output
  - (1950, 25)

If a combiner finds the maximum temperature for each map output:
- First map produces
  - (1950, 0)
  - (1950, 20)
  - (1950, 10)
  - \(\Rightarrow (1950, 20)\)
- Second map produces
  - (1950, 25)
  - (1950, 15)
  - \(\Rightarrow (1950, 25)\)
- Input to the reduce function
  - (1950, [0, 20, 10, 25, 15])
  - \(\Rightarrow (1950, [20, 25])\)
- Output
  - (1950, 25) \(\Rightarrow (1950, 25)\)

Can we use a combiner function for finding mean value?
• mean (0, 20, 10, 25, 15) = 14
• mean (mean (0, 20, 10), mean (25, 15)) = mean (10, 20) = 15

• Can combiners replace reducers?
  - No. Combiner is run only on the map output

Specifying a combiner function

```java
Public class MaxTemperatureWithCombiner{
    ...
    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);
    ...
}
```