FAQs
- PA1 deadline has been extended for 24 hrs
  - Two day late period is still provided
  - Late penalty (10% per day) will be applied
- Team assignment is available in Canvas
  - Plan your regular meeting for the TP proposal

Topics discussed in this lecture
- How MapReduce works
  - YARN
  - Combiner and Partitioner
  - Failures

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

YARN cluster 101

- Multi-tenancy
  - Many users launch many different applications
- Scalability
  - A cluster scheduler needs to scale to large clusters running many applications

Scheduling in YARN

- The scheduler is a part of the Resource Manager
  - matching honoring organizational policies on sharing resources
- YARN uses queues to share resources
  - The ApplicationMaster (AM) tracks each task’s resource requirements and coordinates container requests
  - allows better scaling
  - the RM/scheduler doesn’t need to track all containers running on the cluster

FAIR Scheduler

- Hadoop supports
  - Capacity, priority, and Fair scheduler
- A popular choice
  - recommended by Cloudera
  - Shares resources fairly among all jobs running on the cluster
- Queues
  - Organizing structure for YARN schedulers
  - Hierarchical Queues
    - Root queue
    - Non-root queues are either child of the root queue or other queue
    - Queues can be correspond to the concepts such as user, department or priorities

Representing queues in YARN

Queue weights and top-down scheduling

The marketing queue has a weight of 3.0
The sales queue has a weight of 4.0
The datasciencequeue has a weight of 1.0
So, the allocation from the root will be 15% to marketing, 20% to sales, and 65% to data science.
Queue weights and top-down scheduling

<?xml version="1.0"?>
<queue>
  <queue name="short_jobs">
    <weight>0.0</weight>
  </queue>
  <queue name="est_effort_jobs">
    <weight>100.0</weight>
  </queue>
  <queue name="data_intensive">
    <weight>10.0</weight>
  </queue>
  <queue name="intensive_short_jobs">
    <weight>100.0</weight>
  </queue>
  <queue name="best_effort_jobs">
    <weight>1.0</weight>
  </queue>
</queue>

What if there is no job queued in the dataScience queue?

All of the queue's allocation goes to the short_jobs queue. If there is no job assigned to the short_jobs queue, the jobs in the best_effort_jobs queue are allocated resources.

Other MapReduceSchedulers

- NLong
- Maplet
- CoMoR
- ARIA
- CSRA
- CSRA
- ARIA
- CSRA

Job Initialization

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

• Application master
  - The application master for MapReduce jobs
    - Hadoop cluster
      - Initiates 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 [3/3]

- Application master
  - Plans job execution
    - If the job is "small", Application Master will run the tasks in the same JVM as itself
- Uber task
  - The overhead of allocating and running tasks in new container outweighs the gain to be had it running them in parallel, compared to running them sequentially on one node
  - mapreduce.job.ubertask.enable (default = false)
  - mapreduce.job.ubertask.maxmap (default=9)
  - mapreduce.job.ubertask.maxreduce (default=1)
  - mapreduce.job.ubertask.maxbytes (default=HDFS Block Size)

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.schedular.capacity.minimum- allocation-mb
    - yarn.schedular.capacity.maximum-allocation-mb
    - Default minimum: 1024MB
    - 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 (Not the Resource Manager)
  - `mapreduce.client.progressmonitor.pollinterval`

---

Shuffle and Sort

- **Sort**
  - MapReduce makes the guarantee that the input to every reducer is sorted by key

- **Shuffle**
  - MapReduce transfers the map outputs to the reducers as inputs

---

In a map task

- Each map task has a circular memory buffer
  - For output
    - 100MB by default
    - `io.sort.mb`

- When the contents of the buffer reaches the threshold, a background thread starts `spill the contents to disk`
  - Default 0.80
  - Spills are written in round-robin-fashion to the local directory
    - `mapred.local.dir`
Partitioning

- Before the data is written to the local disk, data is divided into partitions corresponding to the reducers
- The background thread performs an in-memory sort by key
  - Within each partition
- Each time the memory buffer reaches the spill threshold, a new spill file is created
  - There can be several spill files after the last output record is written
  - The spill files are merged into a single partitioned and sorted output file

Output file

- Combiner is run
- Output files from map can be compressed
- The output file’s partitions are made available to the reducers over HTTP

Execution of Combiner

- Hadoop does not guarantee on how many times a combiner will be called for each output key

Properties of a Combiner

- Commutative
- Associative
- Input types (key/value pair) and the output types must be the same
  - Same as output of mapper as well

Copy phase

- The reduce task needs the map output from several map tasks across the cluster
- Copy phase
  - The reduce task starts copying their outputs as soon as each completes
  - The map tasks may finish at different times
  - Merges them into larger and sorted files
  - Decompresses the compressed files

Sort phase

- Sort phase
  - All of the map outputs should be moved and copied to the reduce task
  - Merging and sorting the map outputs
  - Sorting is done in rounds
    - If there are 50 map outputs and the merge factor was 10
      - 5 intermediate files
      - 5 rounds of merging will be required
    - Final round
      - A mixture of in-memory and on-disk segments
      - Directly feeds the reduce function
      - Without writes a single sorted file to disk
Failures in Classic MapReduce

- The child task fails
  - Runtime exception from the user code
  - The child JVM reports the error back to its parent before it exits
  - Written in the user logs
  - Tasktracker marks the task attempt as failed
  - Finals a slot for 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
  - Targeted Tasks (Timeouts)

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
  - Unless 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

Questions?