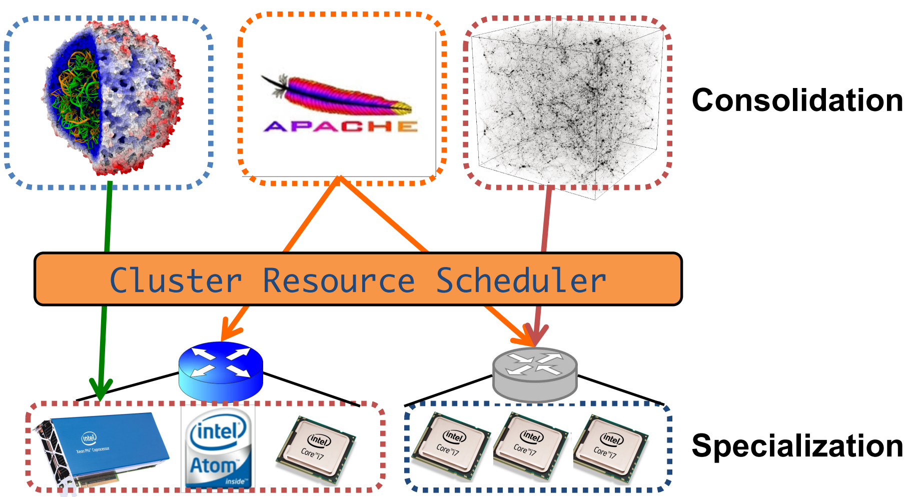
Scheduling Computation

15-719

George Amvrosiadis
Greg Ganger
Majd Sakr

Datacenters today: reality on the ground



Context: bunch of work and bunch of machines

- We've got our collection of machines
  - We'll start by assuming that they are all the same
  - We'll start by assuming that they are allocated as atomic units
- We've got our collection of "jobs" (e.g., VMs)
  - We'll start by assuming that they each want a single full machine
  - We'll start by assuming extensive user effort applied
- Lets build it up, bit by bit, relaxing assumptions as we go

Simplest case: machine checkout

- Each "job" wants any one full machine

- User looks at list of available machines and picks one
  - Edits the list to indicate that it is no longer free
  - Then, uses it

- Assumptions:
  - User owns machine fully and accesses it directly after checking it out
  - User explicitly "frees" machine when done
  - Notice: centralized service (the list) is critical to success

PDL Machines File (PDLDMFv1)

self-* | Limit Display | Home Preferences Add

foobar@baz.com: your query returned 42 result(s)

	Machine Name	Current User	Building	Room	Location	Used As	Notes	Status	Can Checkout
EDIT | CHECKOUT	ss219		CIC	DCO	Z1R8U33	selfstar		self-*	1
EDIT | CHECKOUT	ss231		CIC	DCO	Z1R8U34			self-*	1
EDIT | CHECKOUT	ss304		CIC	DCO	Z1R8U15	ESX server		self-*	1
EDIT | CHECKOUT	ss306		CIC	DCO	Z1R8U17	ESX server		self-*	1
EDIT | CHECKOUT	ss307		CIC	DCO	Z1R8U18	Xen		self-*	1
EDIT | CHECKOUT	ss308		CIC	DCO	Z1R8U19	ESX server		self-*	1
EDIT | CHECKOUT	ss309		CIC	DCO	Z1R8U20	Xen		self-*	1
EDIT | CHECKOUT	ss310		CIC	DCO	Z1R8U21	ESX server		self-*	1
EDIT | CHECKOUT	ss311		CIC	DCO	Z1R8U22	ESX server		self-*	1
EDIT | CHECKOUT	ss312		CIC	DCO	Z1R8U23	847z VMware ESX test machine		self-*	1
EDIT | CHECKOUT	ss313		CIC	DCO	Z1R8U24			self-*	1
EDIT | CHECKOUT	ss314		CIC	DCO	Z1R8U25			self-*	1
EDIT | CHECKOUT	ss315		CIC	DCO	Z1R8U26			self-*	1
EDIT | CHECKOUT	ss316		CIC	DCO	Z1R8U27			self-*	1
EDIT | CHECKOUT	ss317		CIC	DCO	Z1R8U28			self-*	1
EDIT | CHECKOUT	ss318		CIC	DCO	Z1R8U29			self-*	1
EDIT | CHECKOUT	ss319		CIC	DCO	Z1R8U30			self-*	1

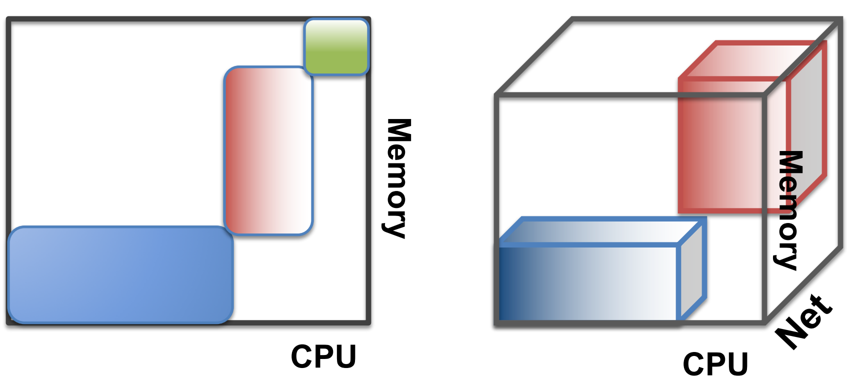
Extension #1: scheduler allocates for and runs jobs

- User submits job to system
  - might be a VM image or some executable script/program
    - depends on type of environment
- Scheduler picks machine and runs the job
  - still requires free machine list
  - also requires ability to start the job on the chosen machine
    - e.g., send to VMM or to scheduling agent that executes on the machine
- When the job finishes
  - the machine "frees itself", by telling the scheduler

Extension #2: packing multiple onto a machine

- User submits job plus resource request (parts of one machine)
  - e.g., RAM capacity (!!) and CPU fraction (in MHz or cores)
- Scheduler picks a machine with enough resources and runs job on it
  - must now track what portion of each machine is allocated vs. free
  - picking machine is somewhat akin to memory allocation
    - options like first fit, best fit, etc. apply
    - but, the physical machine boundaries make it a bit different
- Assumptions for now
  - the resource request is sufficient to the need
  - local machine agent ensures allocation fractions
  - interference among jobs on a machine can be ignored
  - can ignore unused fractions of machine

Extension #2: packing multiple onto a machine



Extension #3: packing with uncertainty

- User's resource requests can be imperfect
  - common to ask for more than needed
  - can often use more, if available, as well… e.g., to finish faster
- Overcommitting
  - monitor resource usage, identify under-utilization of allocation, and use it
  - assign more total "allocation" (e.g., RAM or CPU) to a machine than would fit
  - biggest issue: dealing with situations where resources run out
    - e.g., job tries to use its requested allocation of RAM, but there isn't enough
    - options: kill or migrate that job, kill or migrate a different job, shrink allocation
- Using slack resources
  - imagine that only ½ of the CPU has been allocated to jobs so far
  - should those jobs use the extra CPU?

Extension #4: informing decisions re: uncertainty

- User provides more information than just the resource request
  - scheduler and per-machine agent use it
- VMware extra information
  - Reservation: guaranteed minimum amount (say "no" if can't promise)
  - Limit: upper bound (so, don't use extra resources beyond certain amount)
  - Share: relative importance of different jobs (when sharing extra resources)

Extension #5: machines not all the same

- Few data centers / clouds have a single machine type
  - different amounts of RAM, different CPU speeds, core counts, etc.
  - could be special features (e.g., GPU) only present on some of them
- Scheduler still works in largely the same way
  - still track what portion of each machine is allocated vs. free, and pick
  - special features require pruning set of options considered
    - e.g., just the ones with a GPU
- Interesting nuance: exposing vs. hiding machine differences
  - remember "MHz" as a measure? Or, number of cores?
  - expose special features at all?

Ex: heterogeneity in AWS

- Amazon EC2 instance type proliferation:
  - General Purpose T2
  - Balanced M3
  - Compute Optimized C3
  - Memory Optimized R3
  - GPU G2
  - Storage Optimized I2
  - High Storage Density HS1
- Also, multiple sizes for most types: small, large, x-large, 2x-large, …
- Instance type detail matrix – doesn't fit on the screen…

Extension #6: changing previous decisions

- Free resources can become fragmented or poorly distributed
  - as jobs finish at arbitrary times that often cannot be known
  - may be enough resources for a new job, but not all together
  - over-committing or slack usage may be improvable
- Changing decisions requires work
  - the job must be moved, somehow, from one machine to another
    - inducing tradeoff between short-term cost vs. long-term benefit
  - primary options: migration or “shoot-and-restart”
    - both take time and resource from doing real work

Extension #7: non-resource constraints

- For some jobs, there are additional concerns to be addressed
  - e.g., being close to or not being close to another job
- VMware constraint examples
  - Affinity: identifies VMs that would benefit from being on same machine
    - to allow for faster communication
  - Anti-affinity: identifies VMs that must not be on same machine
    - to ensure that a machine crash does not disable both
- Constraints more generally
  - can be any machine attributes, though scheduler+user must understand
  - restricts the set of options that the scheduler can consider for a given job
  - also, affinity and anti-affinity can relate to more than just “same machine”

Extension #578: multi-machine jobs

- It is not uncommon for a request to ask for several machines at once
  - e.g., to run a Hadoop instance or a 3-tier web service
- Scheduler considers the request as a whole
  - most schedulers will wait until can schedule the entire thing
    - so, it needs to find enough free resources fitting constraints at the same time
    - some schedulers will give whatever subset it can, ASAP, rather than waiting
  - may also try to improve assignments based on knowing the full set
    - e.g., run them on same machine or rack
- Interesting nuance: to hoard or not to hoard
  - "large" requests may wait forever, if the scheduler just waits to get lucky
  - can "hold back" resources, as they become free, until enough are free
    - But, they are "wasted" while waiting

Wrap-up (for this part)

- Map collection of jobs (as they arrive) onto set of machines
- Lots of differences (in the details) among different schedulers
  - so, it's worth looking at examples that we suggested as readings
  - and, we'll talk more about scheduler architecture on Wed

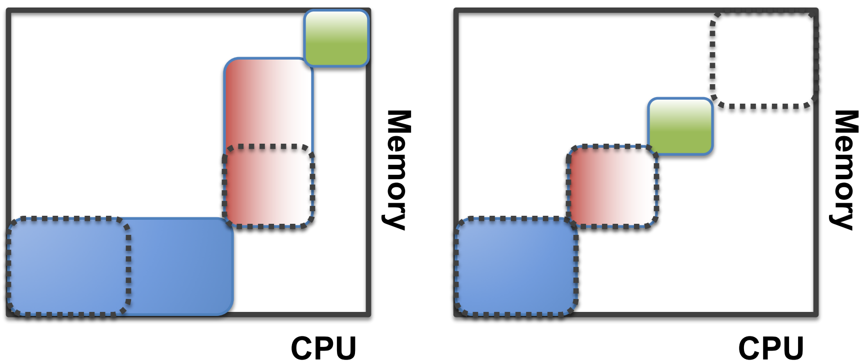
Next day plan

- Hey, not done with today! (Majd on MapReduce)
- Next time: guest lecture about Microsoft Azure!
- After that: scheduler architecture and multi-level scheduling

Wrap-up (for this part)

- Map collection of jobs (as they arrive) onto set of machines
- Basic building blocks
  - Central scheduler: receives requests, tracks and allocates resources
    - Lots of options and potential complexity in the algorithm
  - Per-machine agent: runs jobs, enforces allocations, monitors usage
- Lots of differences (in the details) among different schedulers
  - so, it's worth looking at examples that we suggested as readings
  - and, we'll talk more about scheduler architecture on Wed

Extension #3: packing with uncertainty



Extension #5: normalizing heterogeneity

Instance	EC2 Compute Units	Memory (GB)	Instance Storage (GB)	Platform	I/O Performance	Hourly Price
Small	1 (1 Virtual Core x 1 Compute Unit)	1.7	160	32-bit	Moderate	$0.10
Large	4 (2 Virtual Core x 2 Compute Unit)	7.5	850	64-bit	High	$0.40
X-Large	8 (4 Virtual Core x 2 Compute Unit)	15	1690	64-bit	High	$0.80
High-CPU Medium	5 (2 Virtual Core x 2.5 Compute Unit)	1.7	350	32-bit	Moderate	$0.20
High-CPU XL	20 (8 Virtual Core x 2.5 Compute Unit)	7	1690	64-bit	High	$0.80

Extension #8: turning off machines to save energy

- Workloads vary a lot in real clouds / data centers
  - many periods where not all resources are utilized
- Completely unused machines could be turned off until needed
  - complicating issue: allocation fragmentation
    - can use decision changes to address, packing jobs more "tightly"
  - Complicating issue: takes time to restart a machine
    - can maintain some slack and predict when the workload is growing
- Another option: "shrinking" machines
  - e.g., dynamic frequency scaling of CPUs
  - e.g., spinning down disks, turning off portions of RAM, etc.

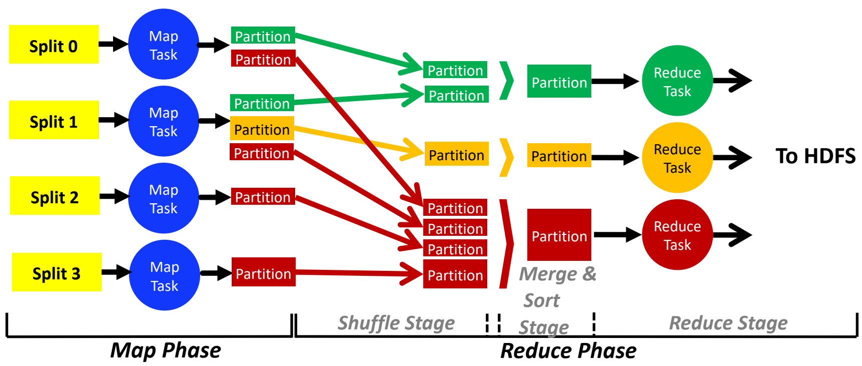
Job & Task Scheduling in MapReduce

15-719/18-709 Advanced Cloud Computing
Spring 2019

March 20, 2019

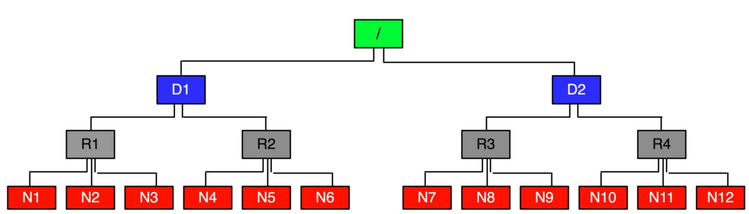
MapReduce

- Applications in MapReduce are represented as jobs
  - Each job encompasses several map and reduce tasks
  - Map and reduce tasks operate on data independently and in parallel



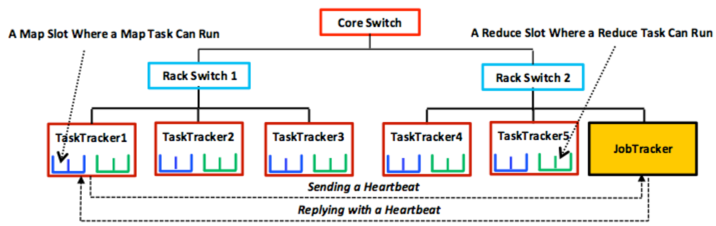
Network Topology In MapReduce

- MapReduce assumes a cluster with a tree style network topology
- Nodes are spread over different racks in one or many data centers
- The bandwidth between two nodes is dependent on their relative locations in the network topology
  - The assumption is that nodes that are on the same rack will have higher bandwidth between them as opposed to nodes that are off-rack



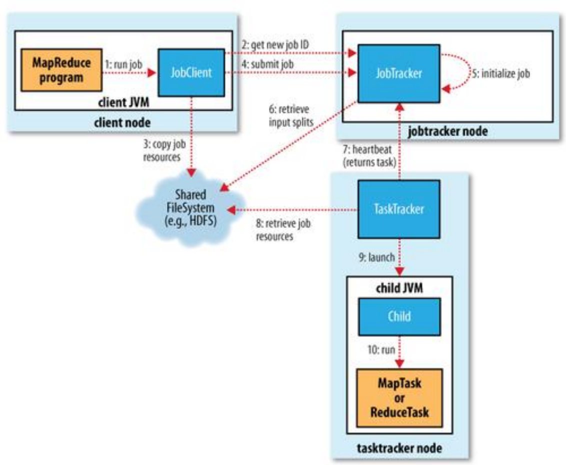
Scheduling a MapReduce Job

- In MapReduce, a job consists of tasks.
- In Hadoop 1.0, a multiple machine cluster includes
  - One master, JobTracker
  - One or many slaves, TaskTrackers
    - Configurable number of Map or Reduce task slots (default, 2M, 2R)
    - TaskTrackers send a heartbeat to JobTracker every 5 secs
    - JobTracker combines updates to produce a global view



Job Submission in MapReduce

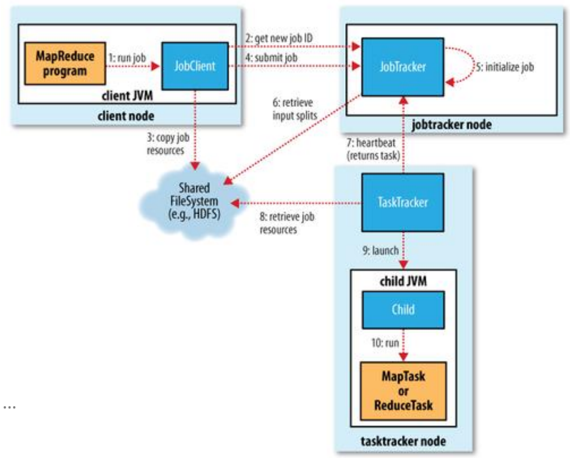
- runJob creates JobClient and calls submitJob
  - Asks JobTracker for job ID
  - Computes splits
  - Copies job resources
    - 10 replicas
- JobTracker adds to queue
  - Job scheduler to pick up and initialize
  - TaskTracker sends heartbeat to JobTracker
  - Scheduler chooses a task from job



T. White (2011). "Hadoop: The Definitive Guide" 2nd Edition." O'REILLY.

Task Assignment in MapReduce

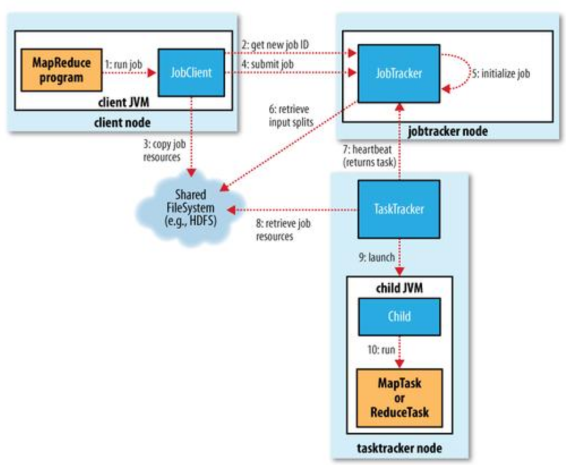
- TaskTracker
  - Fixed # slots for Map & Reduce tasks
    - Depends on resources
- Job scheduler
  - Fills Map slots before Reduce slots
    - Pick a Map task whose split is close to TaskTracker's network location
      - data-local, rack-local, …
  - If no empty map slot, choose next reduce task



T. White (2011). "Hadoop: The Definitive Guide" 2nd Edition." O'REILLY.

Task Execution in MapReduce

- TaskTracker is assigned a task
  - Copy JAR from HDFS to local filesystem
  - Creates local working directory
  - Creates TaskRunner
- TaskRunner
  - Launches child JVM
    - Reuse is possible
  - Run task in JVM
    - Bugs do not affect TT
  - Communicates progress to TaskTracker
- TaskTracker communicates progress to JobTracker



T. White (2011). "Hadoop: The Definitive Guide" 2nd Edition." O'REILLY.

Scheduling in MapReduce

- Centralized job scheduler
  - Default: FIFO
  - Others are pluggable (separate from JobTracker)
    - Fair Scheduler (Facebook)
    - Capacity Scheduler (Yahoo!)
- Task scheduling considers:
  - Data-locality
  - Variations in overall system workloads
  - Failure

FIFO Job Scheduler

- Default FIFO scheduler for jobs
  - A MapReduce job consumes all cluster resources
  - Schedules jobs in order of submission
    - Now schedules jobs with higher priority
  - Schedules tasks from a new job only when all tasks from a running job have been scheduled
  - Starvation with long-running jobs
  - No job preemption
    - A started long-running, low-priority job, cannot be preempted
  - No evaluation of job size

Fair Job Scheduler (Facebook)

- Aims to give each user fair share of cluster capacity over time
- Jobs are placed in pools
  - Each user gets a pool
    - If a single user submits many jobs
- All pools get equal share of cluster resources
  - Default setting

Fair Job Scheduler (Facebook)

- Free slots in idle pools may be allocated to other pools
- Excess capacity within a pool is shared among jobs
- Supports preemption
  - If pool has not received fair share, kills tasks in pools running over capacity
- Jobs within pool share resources equally
  - Priority may be set within pool
- Jobs that require less time can finish with long running jobs

Capacity Scheduler (Yahoo!)

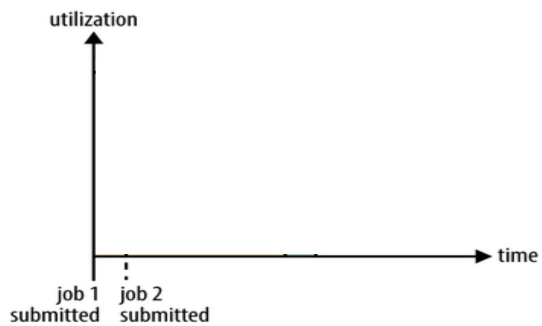
- Defined for large clusters
  - Multiple independent consumers
- Creates job queues
  - Each queue is configured with # of slots (capacity)
  - Each queue has capacity guarantees
    - Sum of all queue capacities equal cluster capacity
    - Excess capacity can be allocated to other queues
  - Within a queue, scheduling is priority based

Hadoop Job Schedulers

	FIFO	Fair	Capacity
Sharing	Limited	Yes	Yes
Starvation	Yes	No	No
Prioritization	Supported but OFF	Within Pool but OFF	Within Queue
Preemption	No	Yes	Designed, implemented?

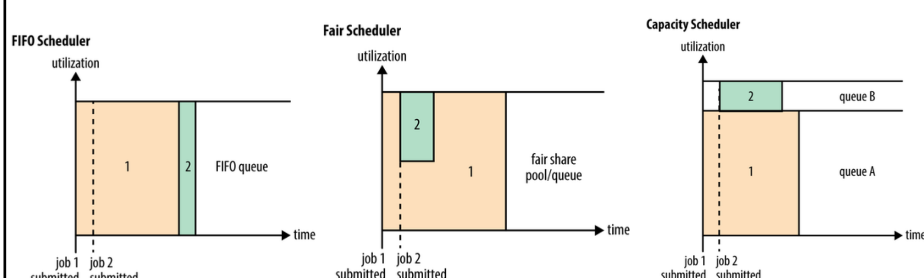
Quiz: Hadoop Job Schedulers

- Job 1, large job, submitted at time t1
- Job 2, small job, submitted at time t2



Hadoop Job Schedulers

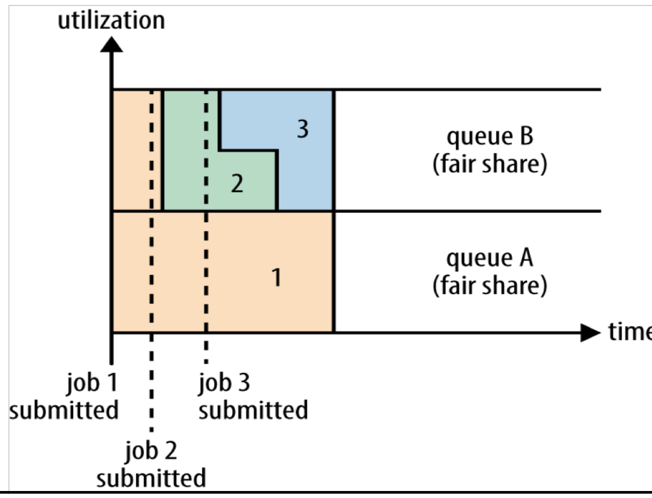
- Job 1, large job, submitted at time t1
- Job 2, small job, submitted at time t2



https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/ch04.html

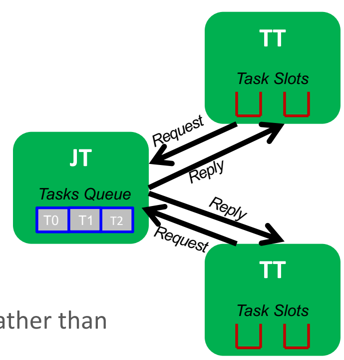
Quiz 2: Fair Scheduler

- Hadoop cluster with 2 users, User A and User B
- User A, submits job1 at t1
- User B, submits job 2 at t2
- User B submits job 3 at t3



Task Scheduling in MapReduce

- MapReduce adopts a *master-slave architecture*
- The master node in MapReduce is referred to as *Job Tracker* (JT)
- Each slave node in MapReduce is referred to as *Task Tracker* (TT)
- MapReduce adopts a *pull scheduling* strategy rather than a *push one*
    - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making requests



Map and Reduce Task Scheduling

- Every TT sends a *heartbeat message* periodically to JT encompassing a request for a map or a reduce task to run

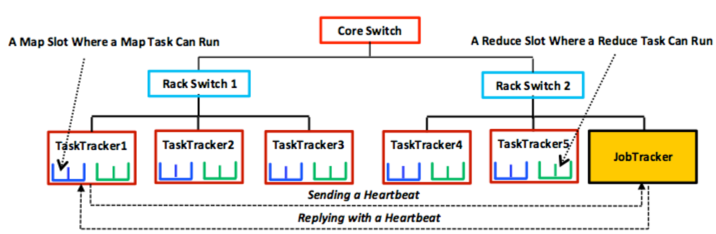
I. Map Task Scheduling:

- JT satisfies requests for map tasks via attempting to schedule mappers in the *vicinity* of their input splits (i.e., it considers locality)

II. Reduce Task Scheduling:

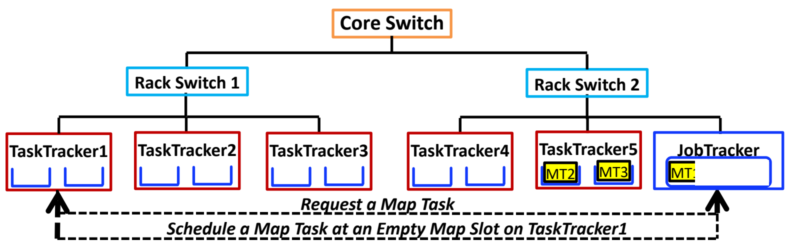
- However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT's network location and its implied effect on the reducer's shuffle time (i.e., it does not consider locality)

Task Scheduling



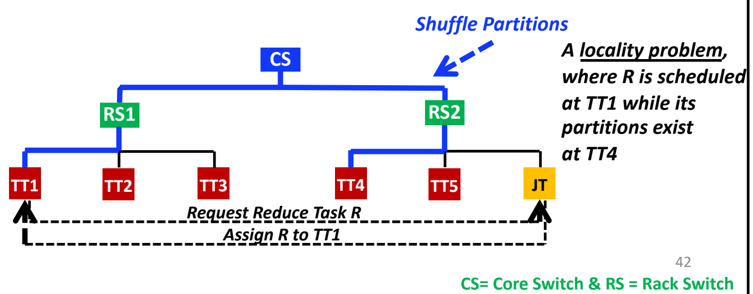
Task Scheduling in Hadoop

- A golden principle adopted by Hadoop is: "*Moving computation towards data is cheaper than moving data towards computation*"
  - Hadoop applies this principle to Map task scheduling
- With map task scheduling, once a slave (or a TaskTracker- ***TT***) polls for a map task, ***M***, at the master node (or the JobTracker- ***JT***), ***JT*** attempts to assign ***TT*** an ***M*** that has its input data local to ***TT***



Task Scheduling in Hadoop

- Hadoop does not apply the locality principle to Reduce task scheduling
- With reduce task scheduling, once a slave (or a TaskTracker- ***TT***) polls for a reduce task, ***R***, at the master node (or the JobTracker- ***JT***), ***JT*** assigns ***TT*** *any* ***R***



Fault Tolerance in Hadoop

- Data redundancy
  - Achieved at the storage layer through replicas (default is 3)
  - Stored at physically separate machines
  - Can tolerate
    - Corrupted files
    - Faulty nodes
  - HDFS:
    - Computes checksums for all data written to it
    - Verifies when reading
- Task Resiliency (task slowdown or failure)
  - Monitor tasks to detect whether faulty or slow
  - Replicate

Task Failure

- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases
- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*
- A task throws a runtime exception
  - JVM informs TT, the TT marks attempt failed and frees up the slot
- If JVM exits
  - TT marks it failed
- If a TT fails to communicate with JT for a period of time (by default, 10 minutes in Hadoop), JT will assume that TT in question has crashed
  - JT asks another TT to re-execute *all Mappers that previously ran at the failed TT*
  - JT asks another TT to re-execute *all Reducers that were in progress on the failed TT*

Task Failure

- When JT is informed by a heartbeat that a task failed
  - Reschedules task
  - Avoids same TT
    - TT is blacklisted
- If task fails > 4 times
  - Job failure
- Maximum % of tasks allowed to fail without triggering a job failure can be configured

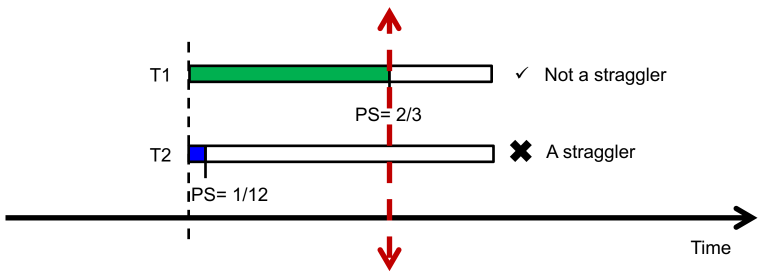
45

Speculative Execution

- A MapReduce job is dominated by the slowest task
- MapReduce attempts to locate slow tasks (*stragglers*) and run redundant (*speculative*) tasks that will optimistically commit before the corresponding stragglers
- This process is known as *speculative execution*
- Only one copy of a straggler is allowed to be speculated
  - Whichever copy of a task commits first, it becomes the definitive copy, and the other copy is killed by JT
- Task prioritization
  1. Dead tasks
  2. Normal tasks
  3. Speculative tasks

Locating Stragglers

- How does Hadoop locate stragglers?
  - Hadoop monitors each task progress using a *progress score* between 0 and 1
  - If a task's progress score ***is less than*** (average – 0.2), and the task has run for at least 1 minute, it is marked as a straggler



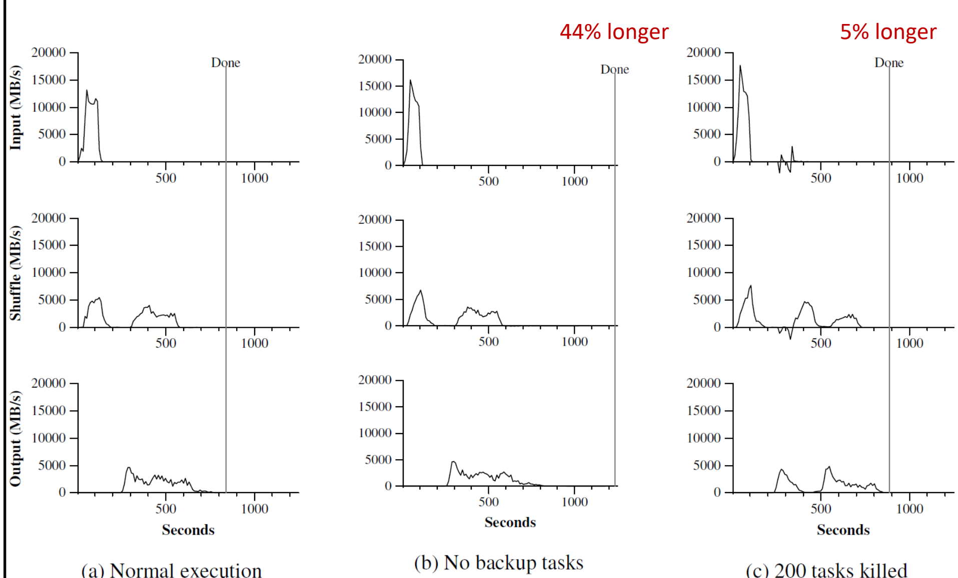


Figure 3: Data transfer rates over time for different executions of the sort program

Drawbacks of Speculative Execution

- Lots of speculative tasks
  - Heterogeneous environments (up to 80%)
  - Transient congestion
- Launches speculative tasks at TTs without checking speed of TT or load of speculative task
  - Slow TT will become slower
- Locality trumps slowness
  - If 2 speculative tasks T1 & T2
    - With stragglers ST1@70% and ST2@20%
  - If task slot is local to ST1's HDFS block, ST1 gets scheduled
- Three reduce stages treated equally
  - Shuffle stage is typically slower than the merge & sort and reduce stages

Monday, 3/25

- Guest Lecture
  - Mark Russinovich, CTO of Microsoft Azure
    - Cloud trends in Azure
    - Bring your CVs