Introduction

Challenges of Big Data

- Volume: Scale of data
  - Impacts performance, cost, reliability, and algorithm design complexity
- Velocity: Speed of streaming/real-time data
  - Impacts performance, cost, reliability, and algorithm design complexity
- Variety: Different formats of data
  - Each new data format $\rightarrow$ New system for handling
- Veracity: Uncertainty of data
  - E.g., Dirty and noisy data, Data provenance/source, Data uncertainty

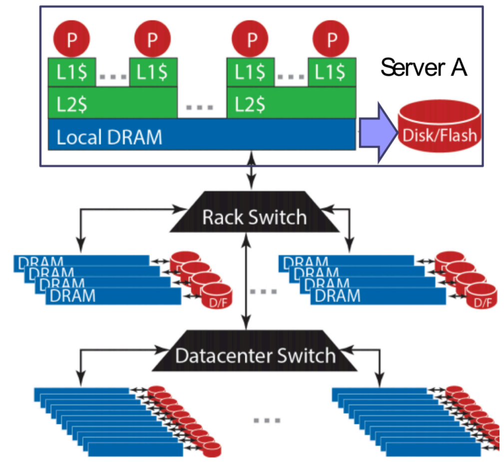
Infrastructure for Big Data

- Utility Computing: Computing resources as a metered service
  - Pros: Scalability (Infinite capacity), Elasticity (Scale up or down on demand)
- To enable utility computing:
  - Virtual Machines: Enable sharing of hardware resources by running each application in isolated virtual machine
    - High overhead: Each VM has its own OS
  - Containers: Lightweight sharing of resources by isolating applications, while sharing the same OS
- Infrastructure as a Service (IaaS): User rents VM and decides what to run (e.g., EC2)
- Platform as a Service (PaaS): Platform to host, while taking care of hardware for you (e.g., Google App Engine)
- Software as a Service (SaaS): Existing app that does everything for you (e.g., Gmail)

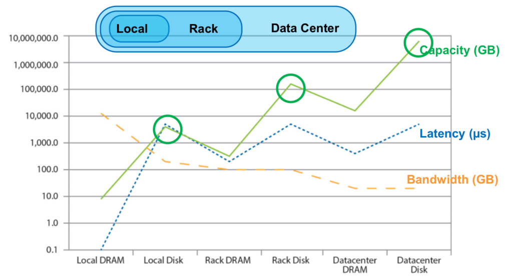
Bandwidth vs. Latency

- Bandwidth: Maximum amount of data that can be transmitted per unit time
  - Throughput: Amount of data *actually* transmitted per unit time
  - Bandwidth of whole path $\approx$ Minimum bandwidth along the path
- Latency: Time taken for 1 packet to go from A to B
  - Latency of whole path $\approx$ Sum of latency along the path

Storage Hierarchy in Data Center



- Local server $\rightarrow$ Rack $\rightarrow$ Datacenter
- Dynamic Random Access Memory (DRAM): Fast but limited capacity
  - Data in disk must first be loaded into DRAM for usage
- Disk: Slow but large capacity
- Flash: In between DRAM and disk



- Disk's capacity $>$ DRAM's capacity
- Capacity increases as we go from local $\rightarrow$ rack $\rightarrow$ datacenter, since the capacity sums up
- Disk reads have higher latency and lower bandwidth than DRAM
- DRAM latency increases as we go from local $\rightarrow$ rack $\rightarrow$ datacenter, due to switch latency
  - Go out of local server $\rightarrow$ Slows by several magnitudes due to switch latency, but slows within same magnitude when going around multiple machines
- Disk latency similar because disk latency is much larger than switch latency
- Bandwidth decreases as we go from local $\rightarrow$ rack $\rightarrow$ datacenter, since bounded by switch

Big Ideas in Data Centers

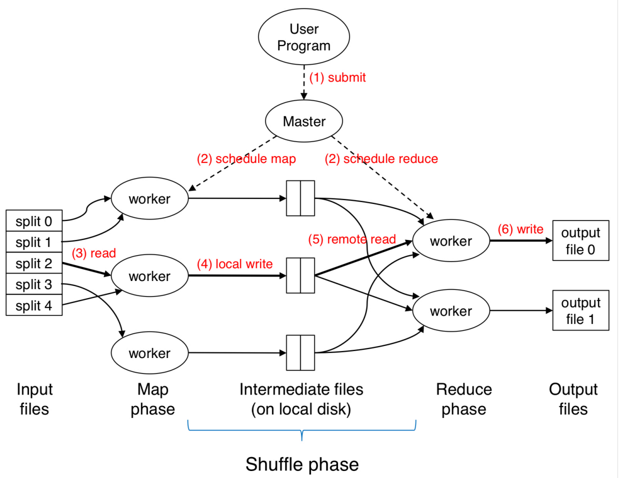
- Scale out, not up
  - Pros: Better reliability, Lower cost

- Seamless scalability: Ideally, if task scales linearly to number of machines, use more machines for better performance
- Move processing to the data, rather than data to the task, via task scheduling
- Process data sequentially and avoid random access

MapReduce

- Motivation: MapReduce provides functional abstraction for 2 operations to solve typical big data problems that include:
  1. Map: Iterating over a large number of records and extracting something from each record
  2. Shuffle: Shuffle and sort intermediate results
  3. Reduce: Aggregate intermediate results
- Interfaces for the user to specify:
  - $\text{map}(k_1, v_1) \to \text{List}(k_2, v_2)$
    - Input: Key-value pair representing a record
  - $\text{reduce}(k_2, \text{List}(v_2)) \to \text{List}(k_3, v_3)$
    - Input: All values with same key are grouped and sent to same reduce task

Implementation



1. Submit: User submits program (`map()`, `reduce()`, and configurations like number of workers)
2. Schedule: Master node schedules resources for tasks and does not handle any data
3. Read: Input files split into tasks of 128MB for workers to execute tasks 1 at a time
   - Can be local or remote read, but local preferred
   - Map phase: Worker iterates and computes over each key-value tuple
4. Local write: Map worker writes outputs of `map()` to intermediate files on local disk
   - Files can be partitioned into chunks depending on number of reduce tasks
   - Each chunk is sorted by key
   - Can be in DRAM, but it's a trade-off between capacity and performance
5. Remote read: Reduce worker responsible for 1 or more keys
   - Reducer processes keys in sorted order
   - For each key, worker reads the needed key-value pairs from corresponding partition of each mapper's local disk
   - Reduce phase: Reducer receives needed key-value pairs and computes reduce function on values of each key
6. Write: Output of `reduce()` written to HDFS

Details

- Choice of split size:
  - Too big: Limited parallelism
  - Too small: High overhead since master node can be overwhelmed by scheduling, Devolves into random access, instead of sequential
- Barrier between map and reduce phases, since we must finish map before starting reduce

Partitioner and Combiner

- Optional functions in local write step that optimize disk and network traffic
- Partitioner: Custom partition defined by user to better spread load among reducers
  - Motivation: By default, assignment of keys to reducers determined by hash function. What if some keys have more values than others?
- Combiner: Locally aggregate output from mappers to reduce disk writes
  - Motivation: Writing `map()` output without aggregating is expensive
  - "Mini-reducers": Can reuse reducer function
  - Ensuring correctness: Reduction function must be binary operation that is associative ($a + (b + c) = (a + b) + c$) and commutative ($a + b = b + a$)
    - E.g., `sum()`, `min()`, `max()`
    - Incorrect: Mean, Minus
  - Possible to maintain state within same map task (e.g., hash table to count words and counts across all lines in a task) $\to$ Trade-off between reducing disk/memory I/O and increasing the memory working set

Performance Guidelines for Algorithm Design

- Linear scalability: More nodes can do more work in same time
- Minimize disk I/O: Sequential vs. Random
- Minimize network I/O: Send in bulk vs. Send in small chunks
- Reduce memory working set (Portion of memory actively being used during execution): Reduce needed memory and out-of-memory errors