Map Reduce

CS4230

Jay Urbain, Ph.D.

Credits:

- Jeffery Dean and Sanjay Chemawat. *MapReduce*
- Barroso and Urs Hölzle (2009)
- Jimmy Lin and Chris Dyer. *Data Intensive Text Processing with MapReduce*

The datacenter *is* the computer

Big Ideas

- **Scale *out*, not *up***
  - Limits of SMP (symmetric multi-processing) and large shared-memory machines
- **Move processing to the data**
  - Cluster has limited bandwidth
- **Process data sequentially, avoid random access**
  - Seeks are expensive, disk throughput is reasonable
- **Seamless scalability**
  - Without synchronization issues, we move from the mythical man-month to the tradable machine-hour


Source: www.robinmajumdar.com

Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

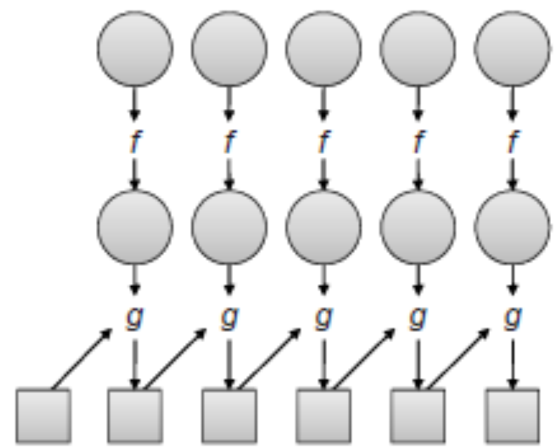
Reduce

Key idea: provide a functional abstraction for these two operations.

Roots in Functional Programming

F**unctional programming**

- Treats computation as the evaluation of mathematical functions.
- Avoids state and mutable data.
- Map and fold: map takes function *f* applies it to every element in a list, fold iteratively applies *g* to aggregate results.



MapReduce – Map + Reduce

- **"Map" step:**
  - The master node takes the input, partitions it up into smaller sub-problems, distributes to worker nodes.
  - The worker node applies a *function* to that smaller sub-problem
- **"Reduce" step:**
  - The master node then takes the answers to all the sub-problems and *aggregates* them in some way to get the output — the answer to the problem it was originally trying to solve.

MapReduce - Programmatic

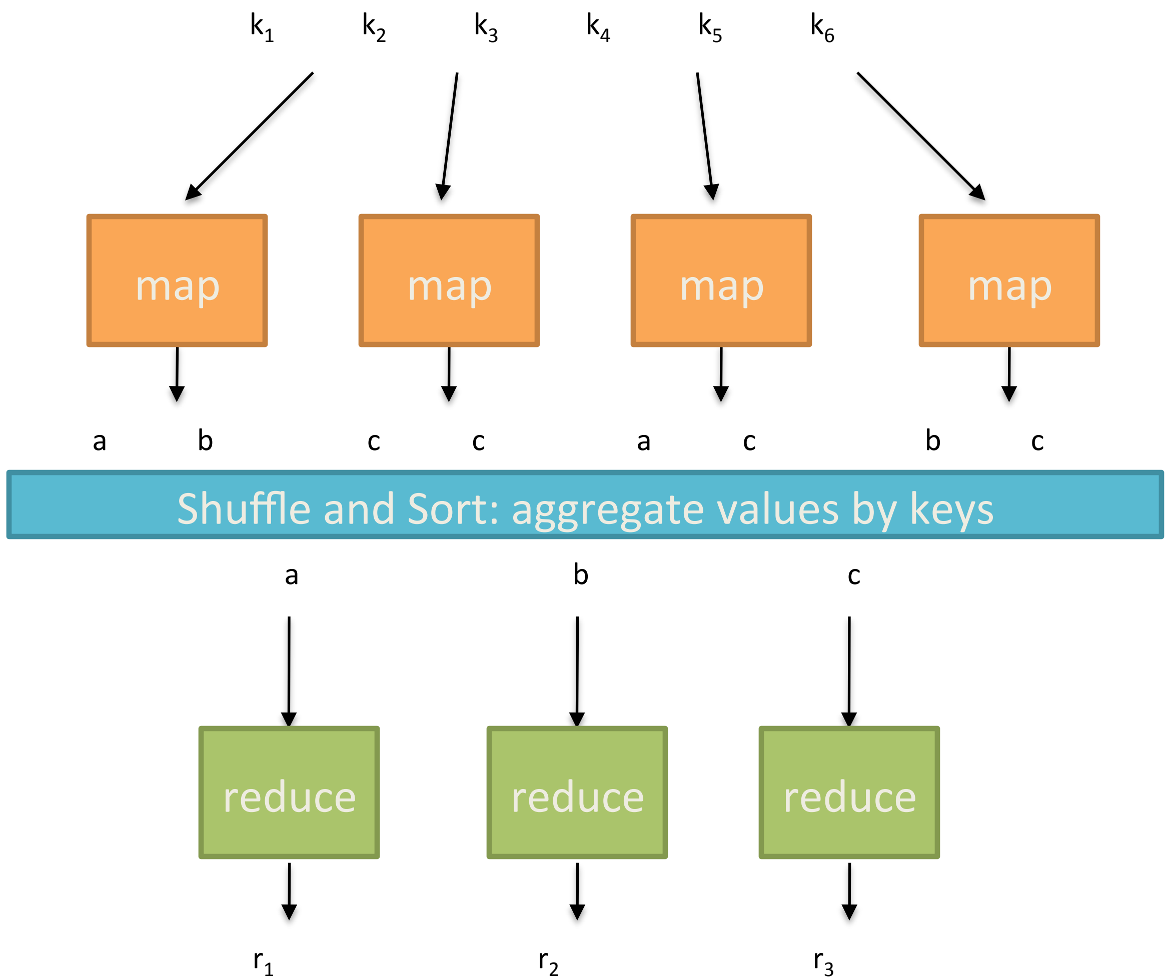
- Programmers specify two functions:
  map (k, v) → <k', v'>*
  reduce (k', v') → <k', v'>*

- Input data stored on underlying distributed file system (GFS, HDFS).
- *Mapper* is applied to every input *key-value pair (k, v)* to generate an arbitrary number of intermediate *key-value pairs <k', v'>*.*
- *Reducer* is applied to all values associated with the same intermediate key *(k', v')* to generate output key-value pairs <k', v'>*.
- Implicit between the *map* and *reduce* phases is a distributed "*group by*" operation on intermediate keys.
- Intermediate data arrive at each reducer in key sorted order.

MapReduce - Notes

Notes:

- Output key-value pairs from each *reducer* are written persistently back onto the distributed file system.
- Intermediate key-values pairs are *transient* and are not preserved.
- The output ends up in *r* files on the distributed file system where *r* is the number of reducers.



k_1 k_2 k_3 k_4 k_5 k_6
map
map
map
map
a b c c a c b c
Shuffle and Sort: aggregate values by keys
a b c
reduce
reduce
reduce
r_1 r_2 r_3

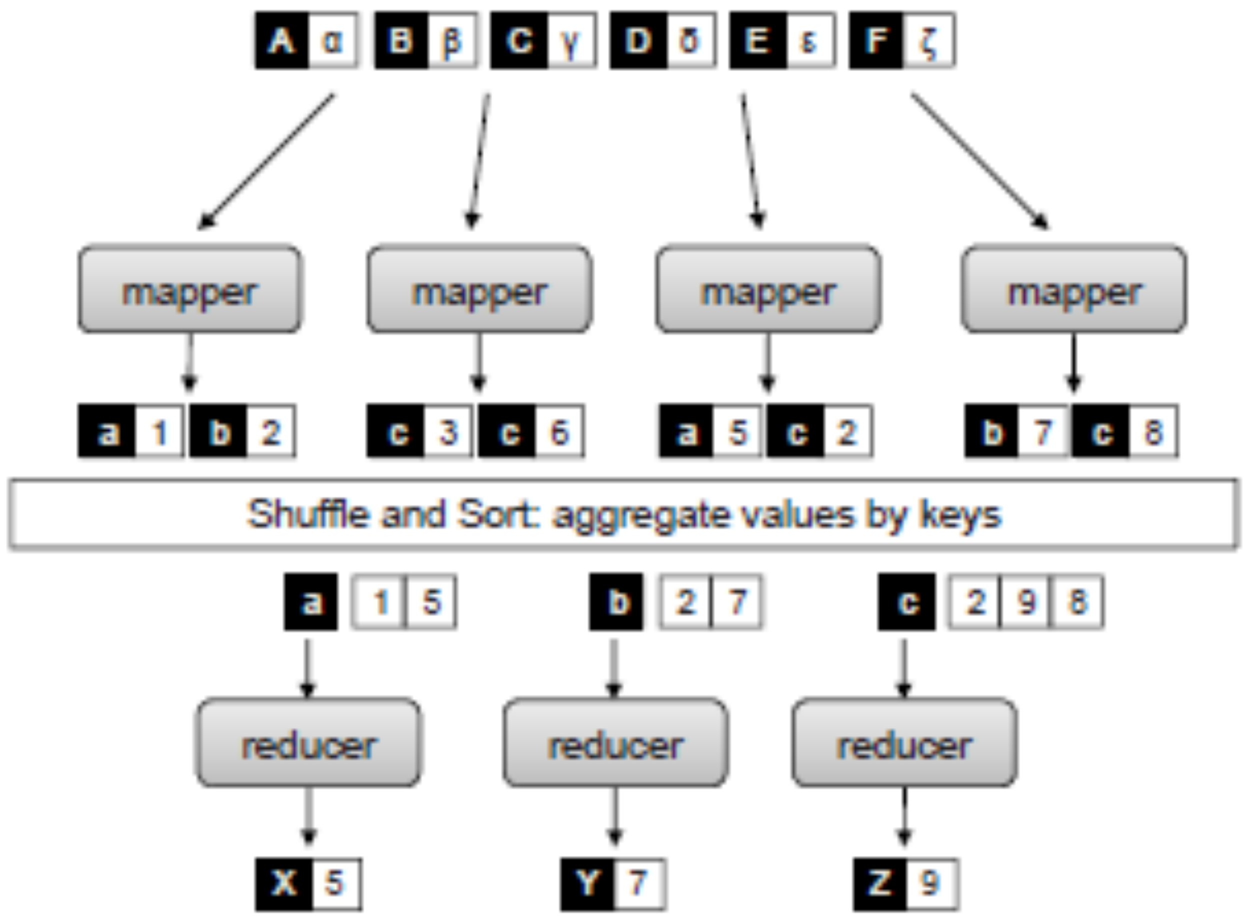
MapReduce Example

- Word count algorithm

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
class Reducer
    method Reduce(term t, counts [c1, c2, . . .])
        sum ← 0
        for all count c ∈ counts [c1, c2, . . .] do
            sum ← sum + c
        Emit(term t, count sum)
```

MapReduce Example

- Simplified MapReduce view



MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  – All values with the same key are sent to the same reducer
- The execution framework handles everything else…

What's "everything else"?

Execution Framework

- One of the most important ideas behind *MapReduce* is separating the ***what*** of distributed processing from the ***how***.
- A *MapReduce* program (job) consists of:
  - Code for *mappers* and *reducers* (as well as *combiners* and *partitioners* … later)
  - Configuration parameters (e.g., where the input lies and where the output should be stored)
- The developer submits the job to the submission node of a cluster (*jobtracker*).
- The execution framework (*runtime*) takes care of everything else.

MapReduce *Runtime*

- **Handles scheduling**
  - Assigns workers to map and reduce tasks
- **Handles "data/code distribution"**
  - Moves processes to data
- **Handles synchronization**
  - Gathers, sorts, and shuffles intermediate data
- **Handles errors and faults**
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (GFS!)

Runtime Scheduling Issues

- ***Scheduling*** involves coordination among tasks belonging to different jobs.
- ***Speculative execution optimization*** (dealing with *Stragglers*)
  - The map phase of a job is only as fast as the slowest map task.
  - Completion of a job is bounded by the running time of the slowest reduce tasks.
  - Speculative execution makes identical copy of the same task and executes on a different machine, use first task completed.
  - Shown to improve running times by 44%.

Runtime Data/code Distribution Issues

- Need to somehow ***feed data to co-located code***.
- Dependent on scheduling and underlying distributed file system.
- The scheduler starts tasks on the node that holds a particular block of data, i.e., local drive, needed by the task.
  - In effect moving the code to the data.
  - If this is not possible, it must stream data across the network.
    - An optimization here is to prefer nodes that are on the same rack in the datacenter – inter-rack BW is much slower.

Runtime Synchronization Issues

- ***Synchronization*** accomplished by a barrier between the *map* and *reduce* phases of processing.
  - Intermediate key-value pairs must be *grouped by key.*
  - Accomplished by *large distributed sort* involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks.
  - Involves copying intermediate data over the network - “shuffle and sort”.
  - *m* mappers and *r* reducers involves up to *m x r* distinct copy operations.
  - Reduce computation can not start until all mappers have finished emitting key-value pairs & all key-value pairs have been shuffled and sorted!

Runtime Error and Fault Handling

- Disk failures are common
- RAM failures
- Planned and unplanned data center outages
- Software bugs
- Large data sets contain corrupted data
- *Rely heavily on distributed file system*

Partitioners and Combiners

- Two additional elements that complete the programming model: ***partitioners*** and ***combiners***.

- ***Partitioners*** – divide up intermediate key space and assign intermediate key-value pairs to reducers.
  - Within each reducer, keys are processed in sorted order
  - The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers.

Partitioners and Combiners

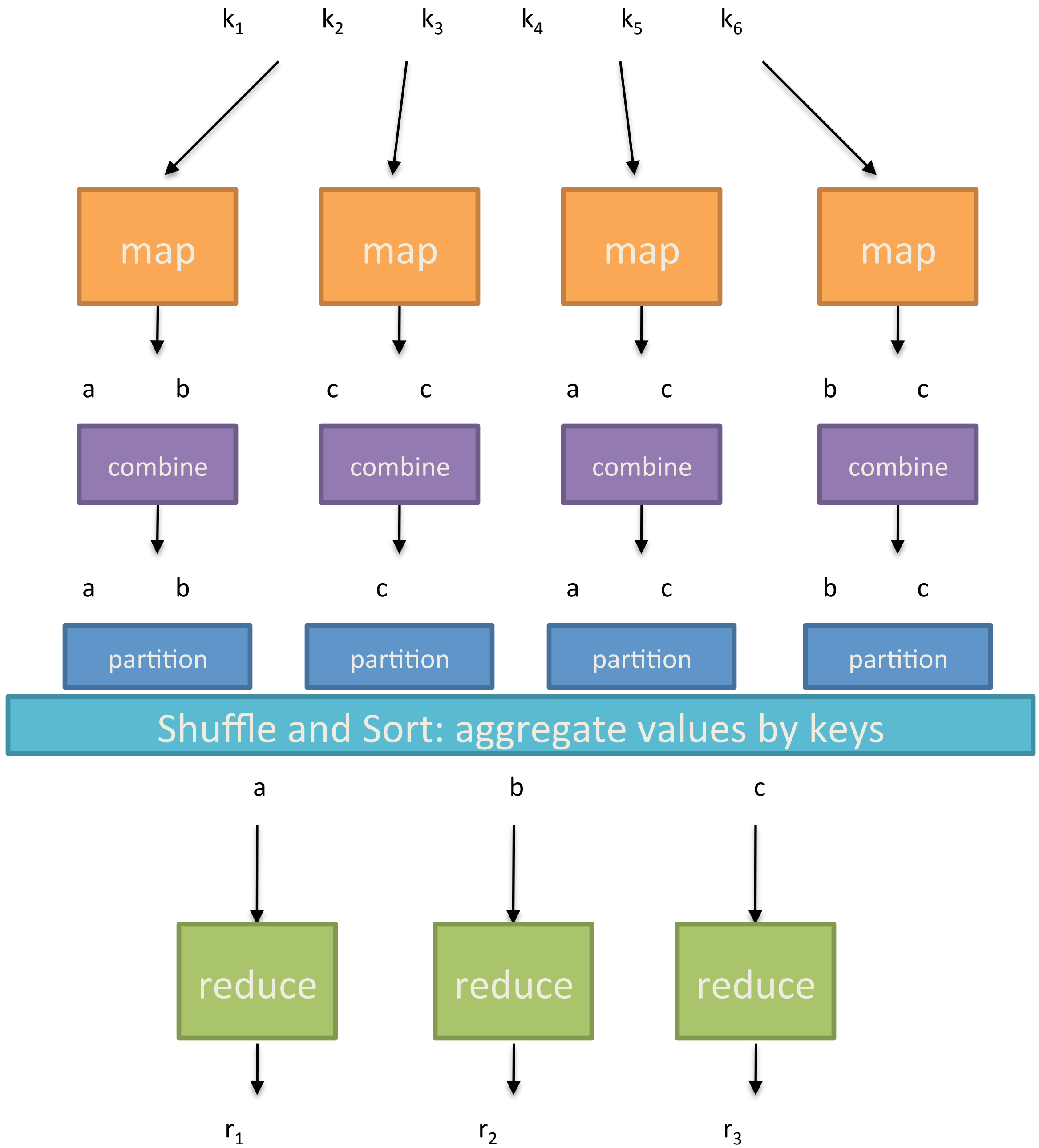
- ***Combiners*** – optimization that allows for local aggregation before the shuffle and sort phase.

Word count example:

  – Emits a key-value pair for each word in the collection!
  – Key-value pairs need to be copied across the network!
  – The amount of intermediate data will be larger than the input collection itself!!! – clearly not good.
  – E.g., perform local aggregation on the output of each mapper, i.e., perform local count for a word over all the documents processed by the mapper.
  – Number of words reduced to (at most) the number of unique words in the collection *times* the number of mappers – usually much smaller (Zipfian distribution of word occurrences).

MapReduce

- Programmers specify two functions:
  map (k, v) → <k’, v’>*
  reduce (k’, v’) → <k’, v’>*
  – All values with the same key are reduced together
  combine (k’, v’) → <k’, v’>*
  – Mini-reducers that run in memory after the map phase
  – Used as an optimization to reduce network traffic
  partition (k’, number of partitions) → partition for k’
  – Often a simple hash of the key, e.g., hash(k’) mod n
  – Divides up key space for parallel reduce operations



k_1 k_2 k_3 k_4 k_5 k_6
map
map
map
map
a b
c c
a c
b c
combine
combine
combine
combine
a b
c
a c
b c
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a
b
c
reduce
reduce
reduce
r_1
r_2
r_3

How do we get data to the workers?

What's the problem here?

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Design Decisions

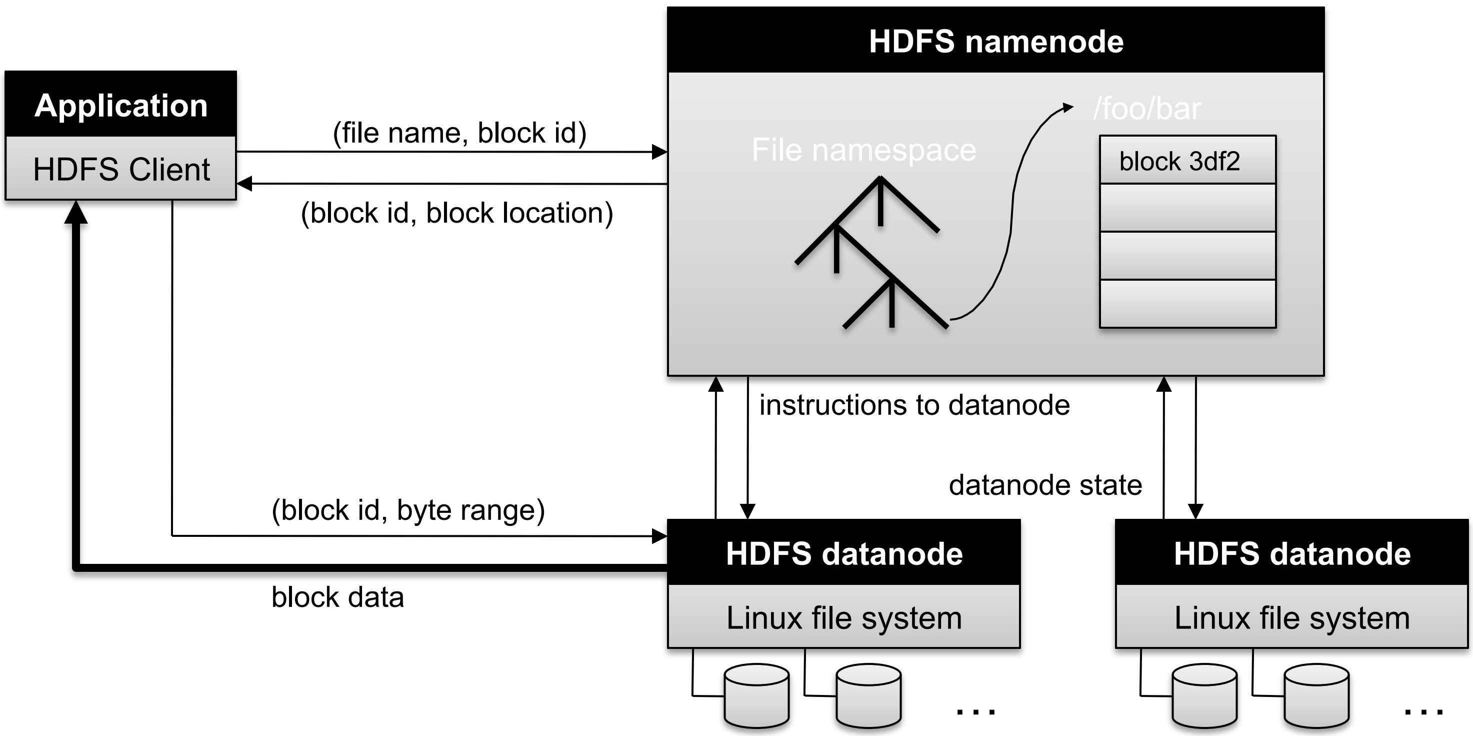
- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - No file appends in HDFS (planned feature)
  - HDFS performance is (likely) slower

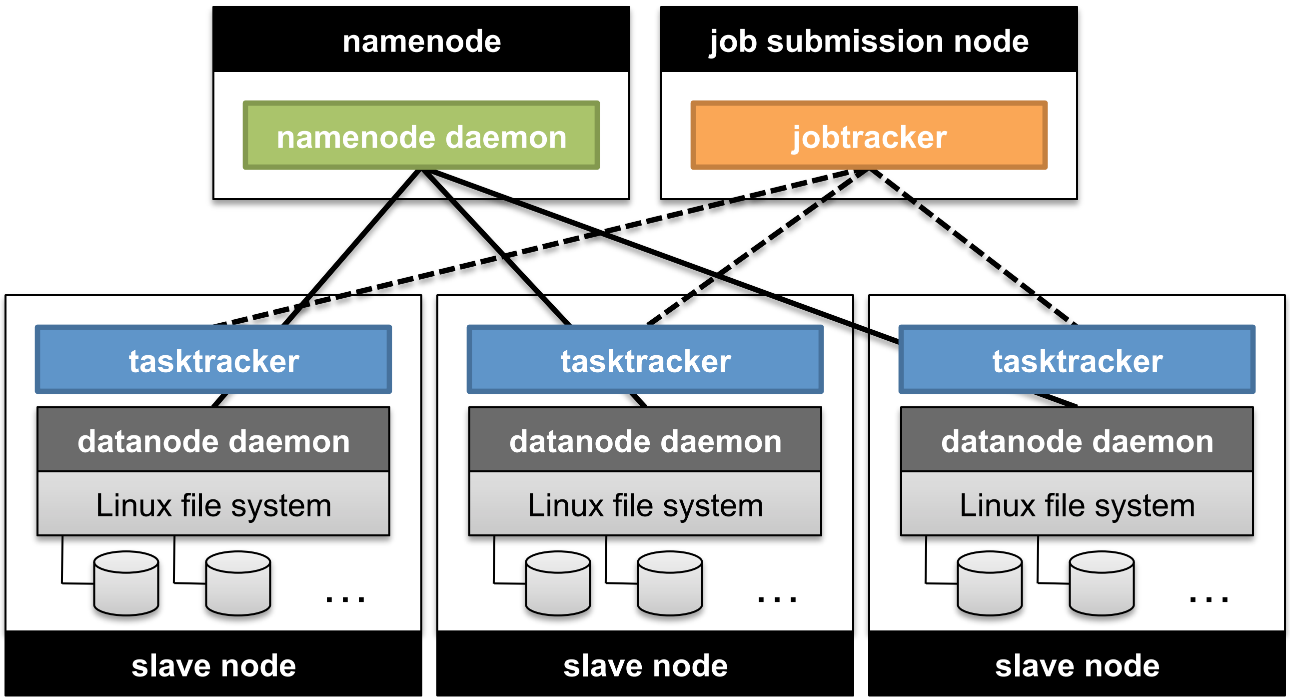
HDFS Architecture



Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together…



Cloud Resources

- Hadoop on your local machine
- Hadoop in a virtual machine on your local machine
- Hadoop in the clouds with Amazon EC2
- Hadoop on the Google/IBM cluster

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project top level project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.