Basics of Cloud Computing – Lecture 3

Introduction to MapReduce

Satish Srirama

Some material adapted from slides by Jimmy Lin, Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet, Google Distributed Computing Seminar, 2007 (licensed under Creation Commons Attribution 3.0 License)

Outline

- Functional programming review
- MapReduce
- Hadoop
- HDFS (Hadoop Distributed File System)

Economics of Cloud Providers – Failures - Recap

- Cloud Computing providers bring a shift from high reliability/availability servers to commodity servers
  - At least one failure per day in large datacenter
- Caveat: User software has to adapt to failures
- Solution: Replicate data and computation
  - MapReduce & Distributed File System
- MapReduce = functional programming meets distributed processing on steroids
  - Not a new idea… dates back to the 50's (or even 30's)

Functional programming

- What is functional programming?
  - Computation as application of functions
  - Theoretical foundation provided by lambda calculus
- How is it different?
  - Traditional notions of “data” and “instructions” are not applicable
  - Data flows are implicit in program design
  - Different orders of execution are possible
- Exemplified by LISP and ML

Functional Programming Review

- Lists are primitive data types
- Functions = lambda expressions bound to variables
  - f = lambda x : 2 * x
    print f(2)
- Higher-order functions
  - Functions that take other functions as arguments
- Recursion is your friend

Functional Programming - features

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

FP features - continued

- fun foo(l: int list) =

  sum(l) + mul(l) + length(l)
  - Order of sum() and mul(), etc does not matter – they do not modify l
- Functional Updates Do Not Modify Structures
  - fun append(x, lst) =

    let lst' = reverse lst in

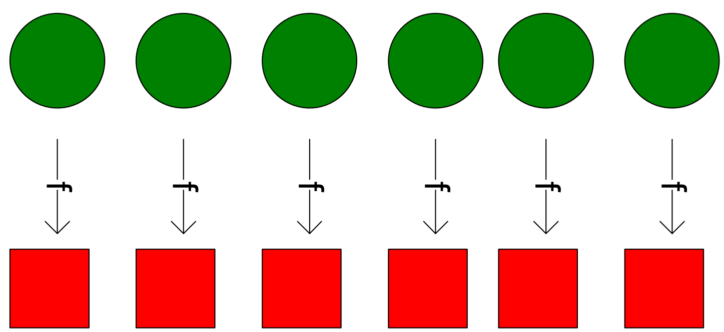
    reverse (x :: lst')
  - The append() function reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item.
  - But it *never modifies lst*!
- Functions Can Be Used As Arguments
  - fun DoDouble(f, x) = f (f x)

Functional Programming -> MapReduce

- Two important concepts in functional programming
  - Map: do something to everything in a list
  - Fold: combine results of a list in some way

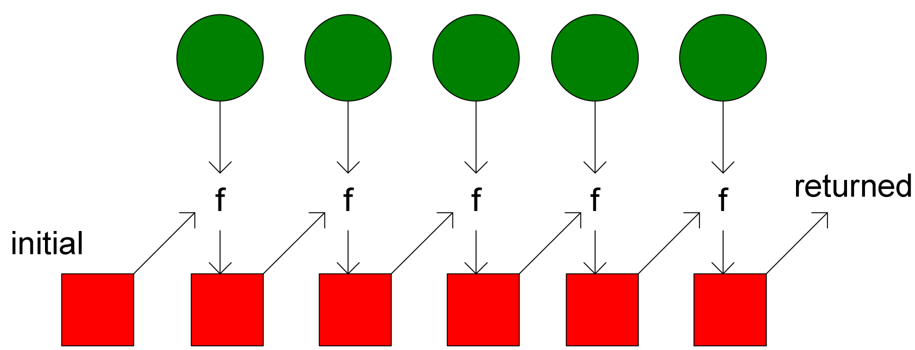
Map

- Map is a higher-order function
- How map works:
  - Function is applied to every element in a list
  - Result is a new list
- map f lst: ('a->'b) -> ('a list) -> ('b list)
  - Creates a new list by applying f to each element of the input list; returns output in order.



Fold

- Fold is also a higher-order function
- How fold works:
  – Accumulator set to initial value
  – Function applied to list element and the accumulator
  – Result stored in the accumulator
  – Repeated for every item in the list
  – Result is the final value in the accumulator



Map/Fold in Action

- Simple map example:

```
(map (lambda (x) (* x x))
     '(1 2 3 4 5))
 → '(1 4 9 16 25)
```

- Fold examples:

```
(fold + 0 '(1 2 3 4 5)) → 15
(fold * 1 '(1 2 3 4 5)) → 120
```

- Sum of squares:

```
(define (sum-of-squares v)
  (fold + 0 (map (lambda (x) (* x x)) v)))
(sum-of-squares '(1 2 3 4 5)) → 55
```

Implicit Parallelism In map

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
- If order of application of *f* to elements in list is *commutative*, we can reorder or parallelize execution
- Let's assume a long list of records: imagine if...
  - We can parallelize map operations
  - We have a mechanism for bringing map results back together in the fold operation
- This is the "secret" that MapReduce exploits
- Observations:
  - No limit to map parallelization since maps are independent
  - We can reorder folding if the fold function is commutative and associative

Typical Large-Data Problem

Map erate over a large number of records

- Extract something of interest from each
- Shuffle and sort inter **Reduce** results
- Aggregate intermediate results
- Generate final output

Key idea: provide a functional abstraction for these two operations – MapReduce

MapReduce

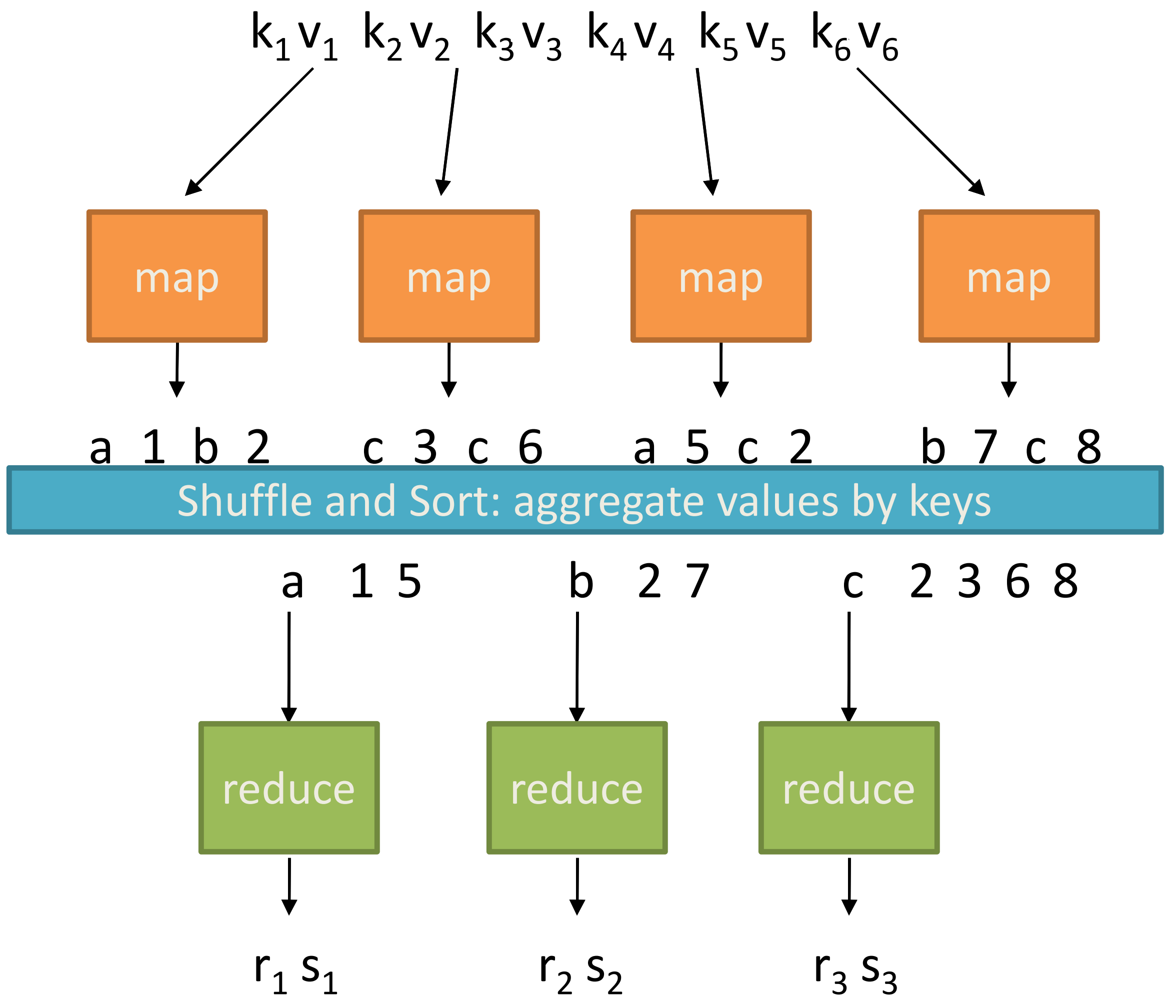
- Programmers specify two functions:

  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…

MapReduce



$k_1 v_1$ $k_2 v_2$ $k_3 v_3$ $k_4 v_4$ $k_5 v_5$ $k_6 v_6$
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
$r_1 s_1$
$r_2 s_2$
$r_3 s_3$

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”?

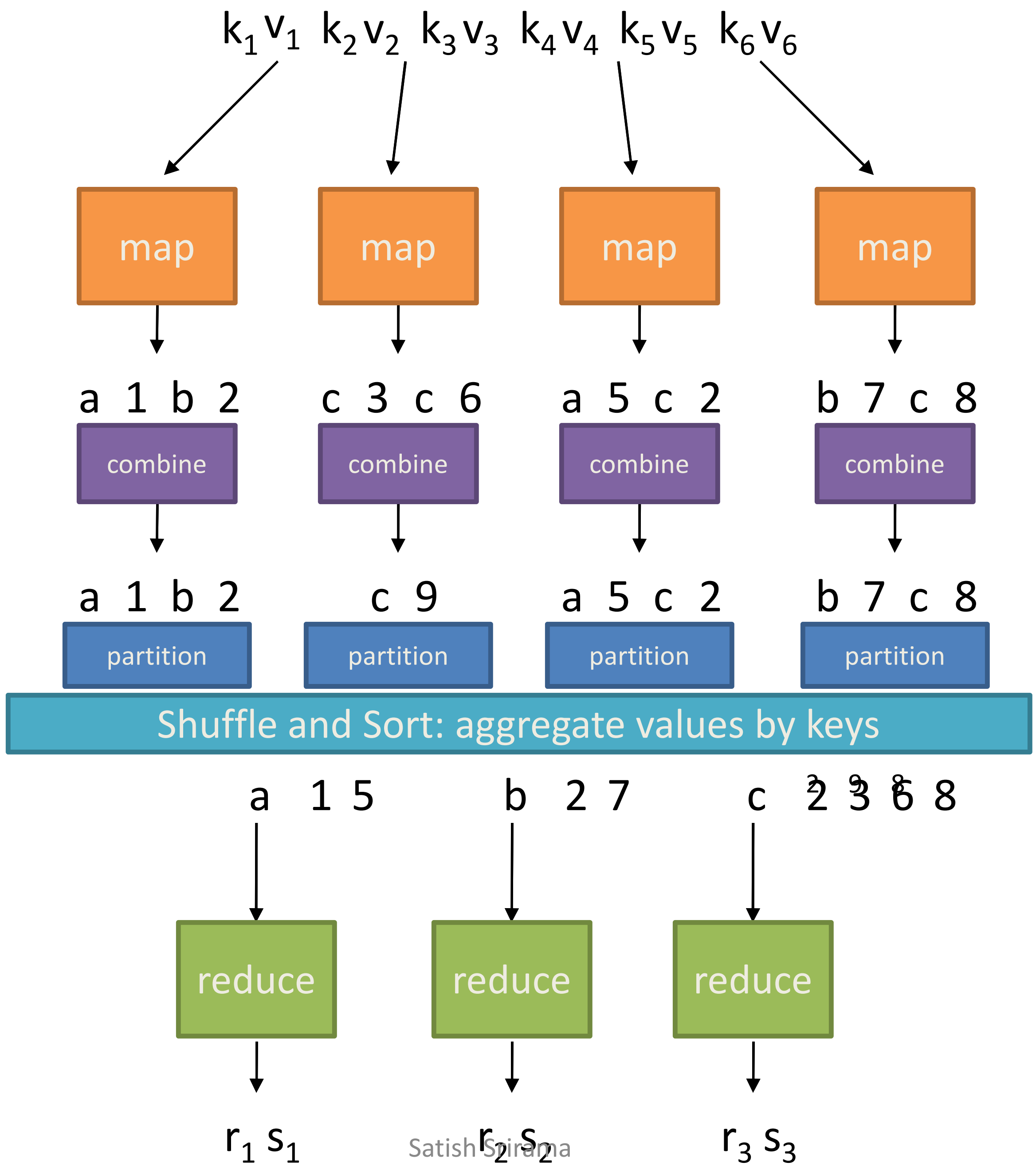
MapReduce "Runtime"

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and automatically restarts
- Handles speculative execution
  - Detects "slow" workers and re-executes work
- Everything happens on top of a distributed FS (later)

Sounds simple, but many challenges!

MapReduce - extended

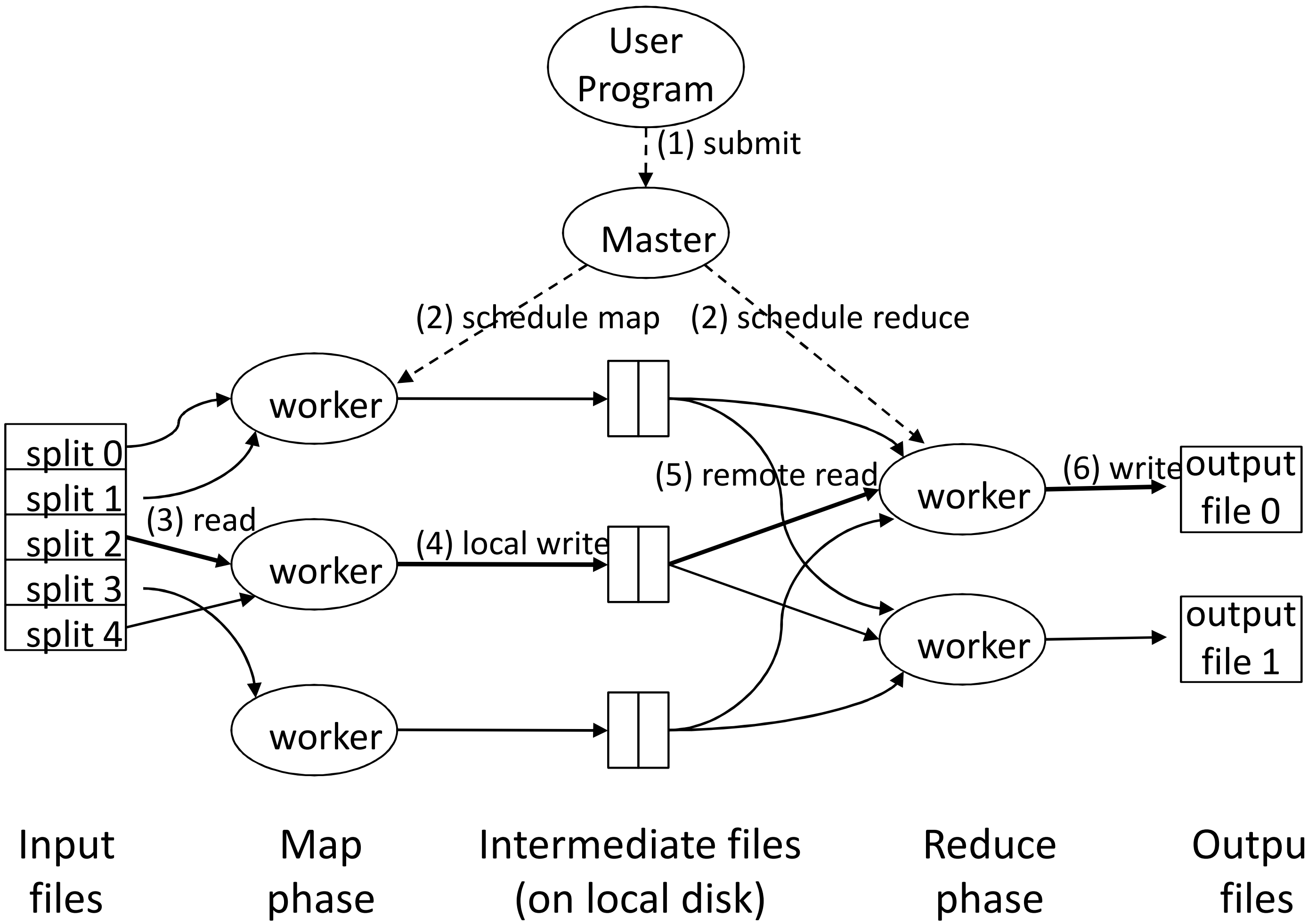
- Programmers specify two functions:
  map (k, v) → <k', v'>*
  reduce (k', v') → <k', v'>*
  – All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:
  partition (k', number of partitions) → partition for k'
  – Often a simple hash of the key, e.g., hash(k') mod n
  – Divides up key space for parallel reduce operations
  combine (k', v') → <k', v'>*
  – Mini-reducers that run in memory after the map phase
  – Used as an optimization to reduce network traffic


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

MapReduce Overall Architecture



Adapted from (Dean and Ghemawat, OSDI 2004)

"Hello World" Example: Word Count

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
    Emit(term, sum);
```

MapReduce can refer to…

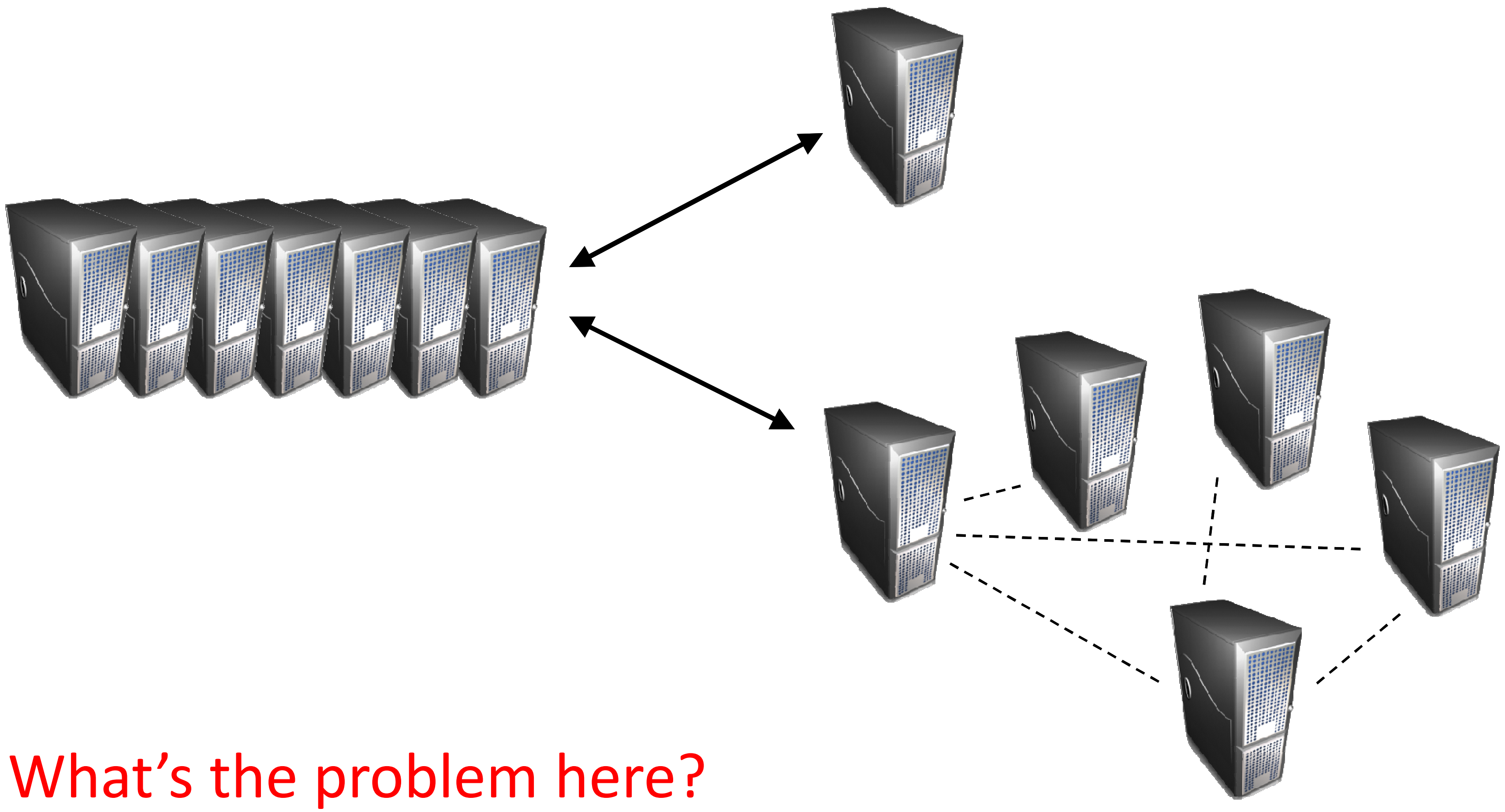
- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually clear from context!

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
  - Rapidly expanding software ecosystem, but still lots of room for improvement (e.g., OSDI 2008, Nexus)
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

Cloud Computing Storage, or 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?
  - Network bisection bandwidth is limited
  - 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: Assumptions

- Choose commodity hardware over “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

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)

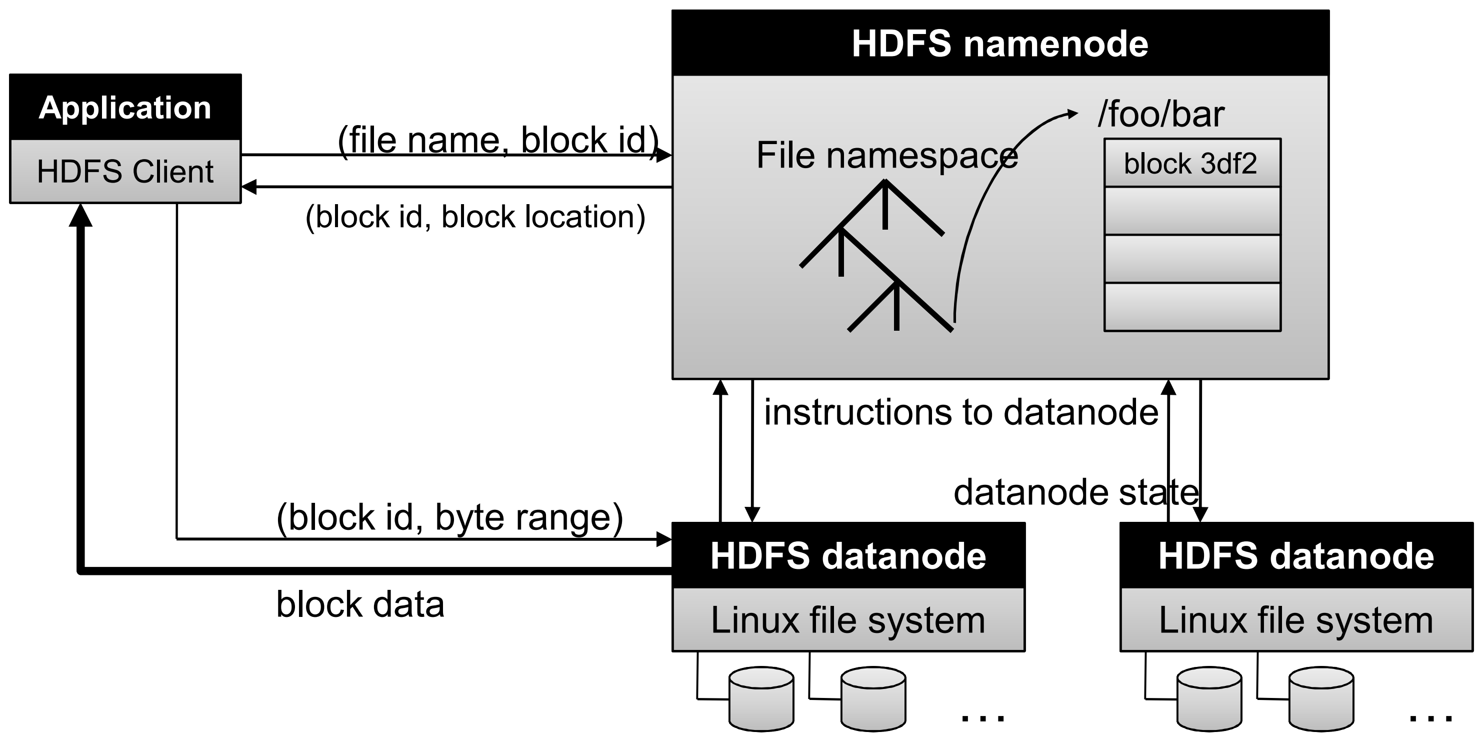
HDFS = GFS clone (same basic ideas implemented in Java)

From GFS to HDFS

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

For the most part, we'll use the Hadoop terminology…

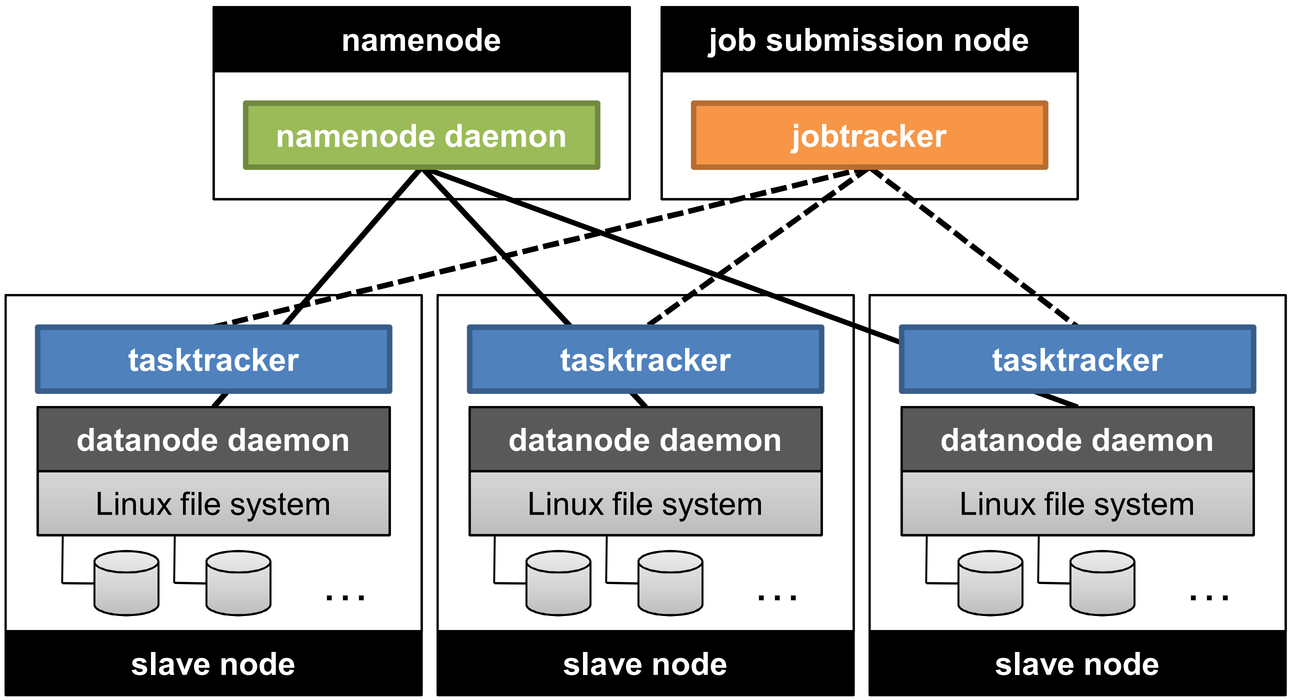
HDFS Architecture



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…



MapReduce/GFS Summary

- Simple, but powerful programming model
- Scales to handle petabyte+ workloads
  - Google: six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers
  - Yahoo!: 16.25 hours to sort 1PB on 3,800 computers
- Incremental performance improvement with more nodes
- Seamlessly handles failures, but possibly with performance penalties

Next Lectures

- Deeper look at Hadoop
- MapReduce in different domains
- Let us have a look at some algorithms

References

- Hadoop wiki http://wiki.apache.org/hadoop/
- Cloudera – Hadoop training http://www.cloudera.com/developers/learn-hadoop/training/
- J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI'04: Sixth Symposium on Operating System Design and Implementation, Dec, 2004.
- Todo:
  - Work with SciCloud Hadoop setup and Cloudera virtual machine