



UNIVERSITY OF TARTU

INSTITUTE OF COMPUTER SCIENCE



Basics of Cloud Computing – Lecture 4

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 Creative Commons Attribution 3.0 License)

Outline

- MapReduce
- Hadoop

Economics of Cloud Providers – Failures

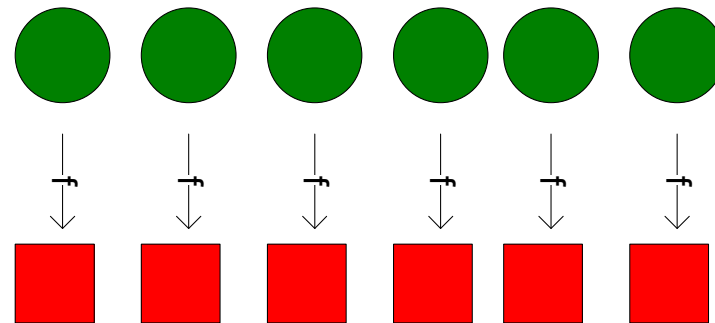
- Cloud Computing providers bring a shift from high reliability/availability servers to commodity servers
 - At least one failure per day in large datacenter
- Caveat: User software has to adapt to failures
- Solution: Replicate data and computation
 - MapReduce & Distributed File System
- MapReduce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

Functional Programming -> MapReduce

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

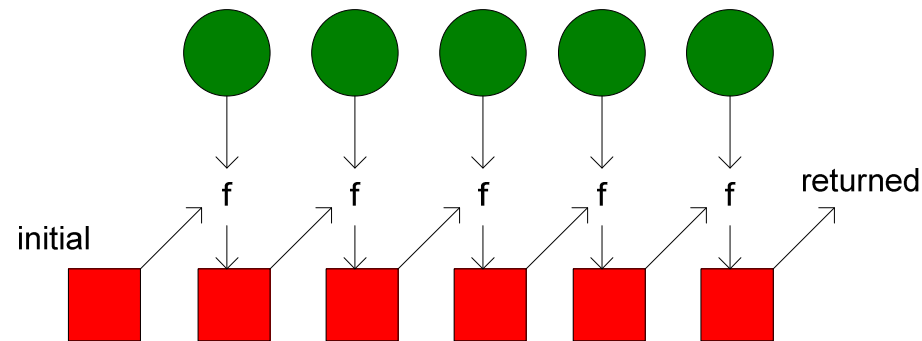
Map

- Map is a higher-order function
- How map works:
 - Function is applied to every element in a list
 - Result is a new list
- $\text{map } f \text{ lst: } ('a \rightarrow 'b) \rightarrow ('a \text{ list}) \rightarrow ('b \text{ list})$
 - 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 intermediate results
- Aggregate intermediate results
- Generate final output

Reduce

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

MapReduce

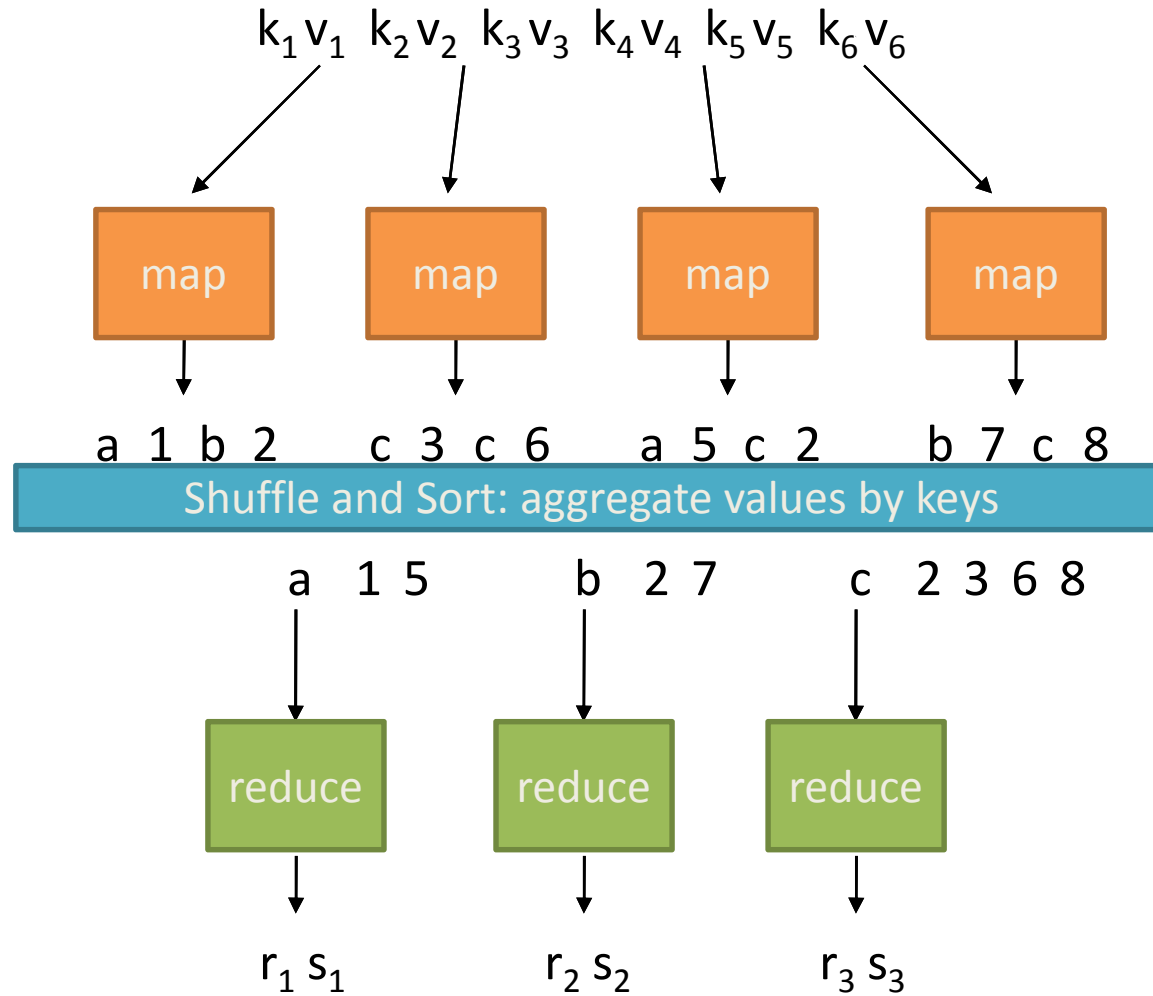
- Programmers specify two functions:
 - map** $(k, v) \rightarrow \langle k', v' \rangle^*$
 - reduce** $(k', [v']) \rightarrow \langle k'', v'' \rangle^*$
 - All values with the same key are sent to the same reducer
- The execution framework handles everything else...

“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



MapReduce

- Programmers specify two functions:
 - map** $(k, v) \rightarrow \langle k', v' \rangle^*$
 - reduce** $(k', [v']) \rightarrow \langle k'', v'' \rangle^*$
 - All values with the same key are sent to the same reducer
- The execution framework handles everything else...

What's “everything else”?

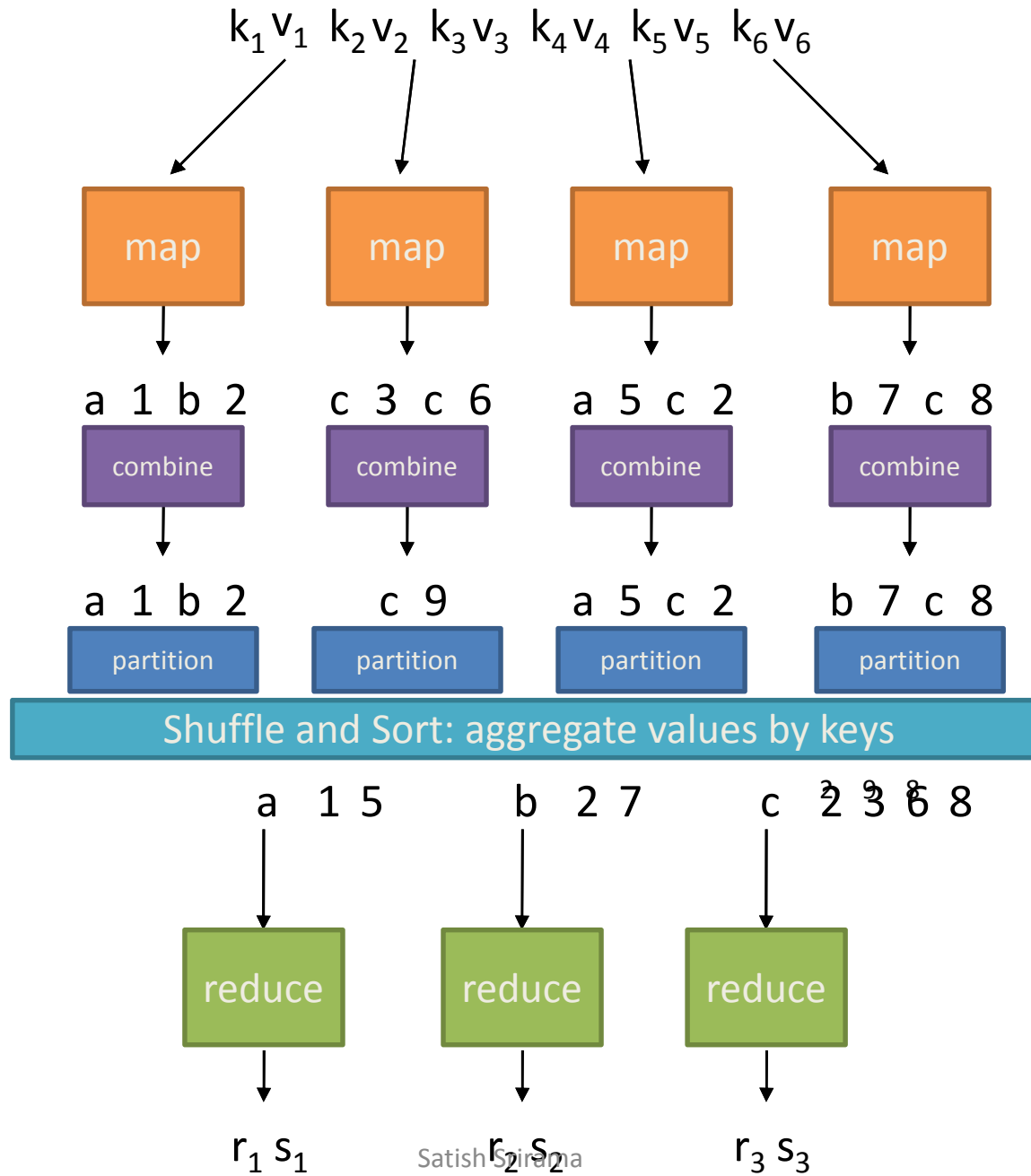
MapReduce “Runtime”

- 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

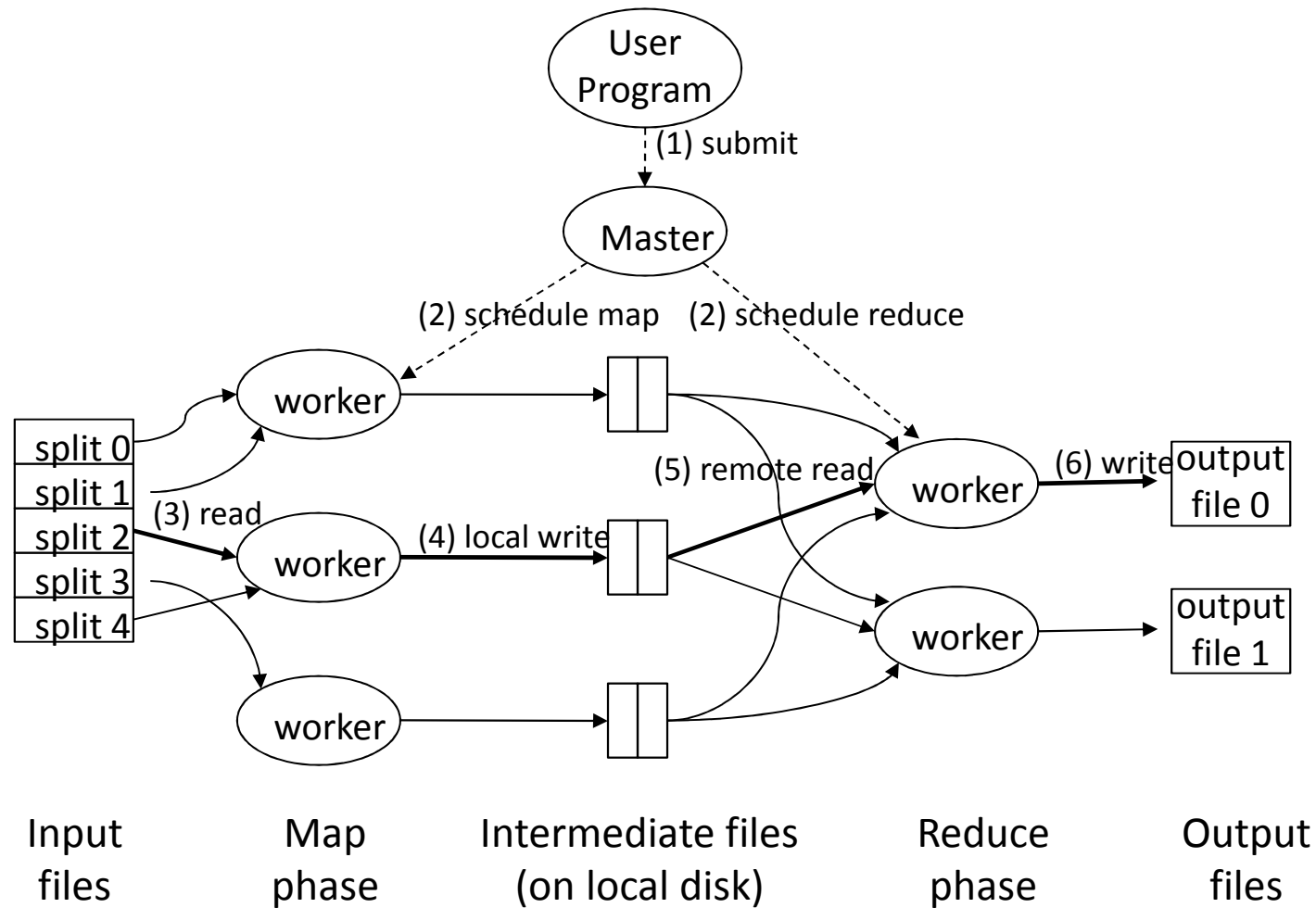
- Programmers specify two functions:
 - map** $(k, v) \rightarrow \langle k', v' \rangle^*$
 - reduce** $(k', [v']) \rightarrow \langle k'', v'' \rangle^*$
 - All values with the same key are reduced together
- The execution framework handles everything else...
- Not quite...usually, programmers also specify:
 - partition** $(k', \text{number of partitions}) \rightarrow \text{partition for } k'$
 - Often a simple hash of the key, e.g., $\text{hash}(k') \bmod n$
 - Divides up key space for parallel reduce operations
 - combine** $(k', v') \rightarrow \langle k'', v'' \rangle^*$
 - Mini-reducers that run in memory after the map phase
 - Used as an optimization to reduce network traffic



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)

MapReduce can refer to...

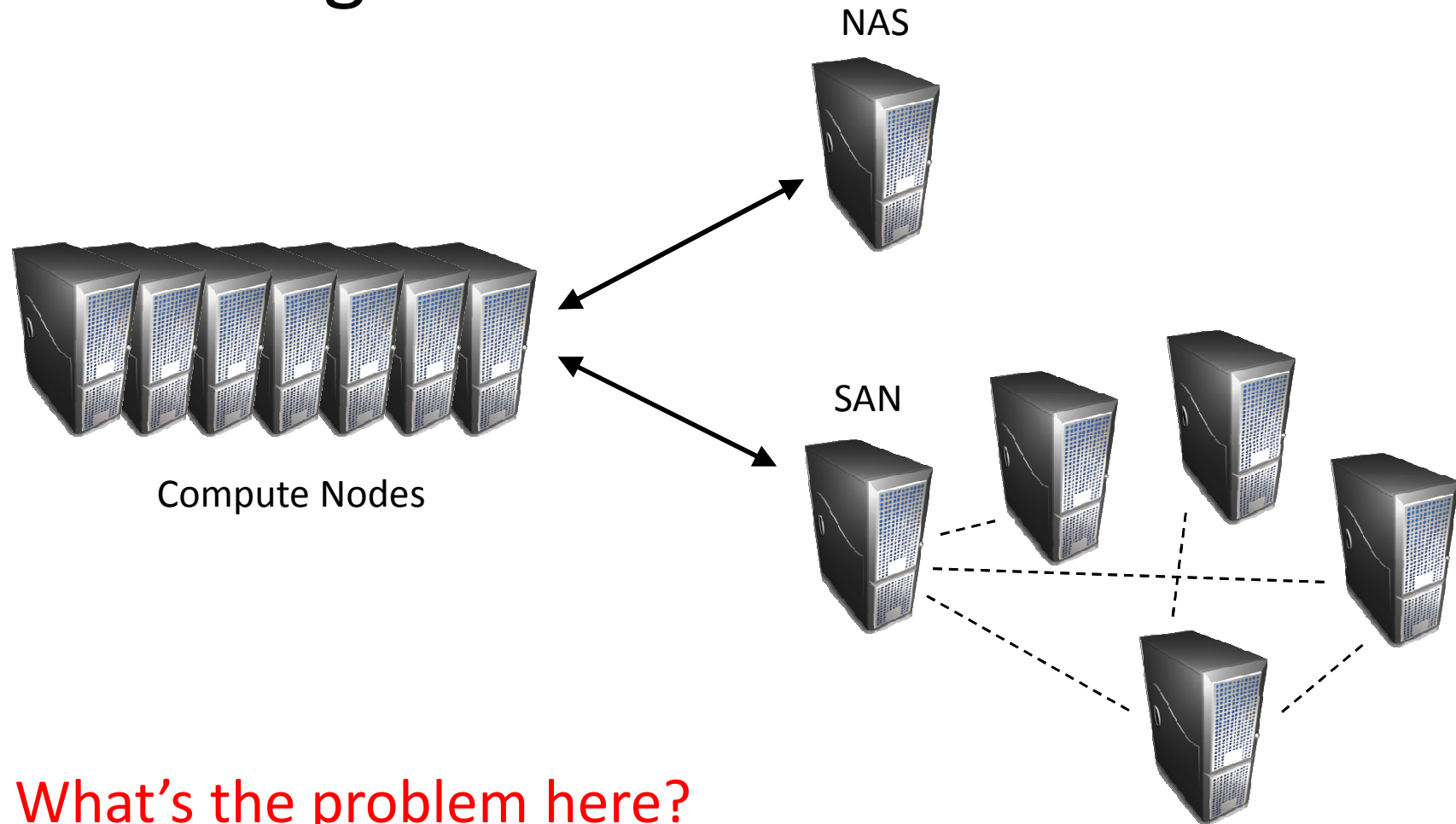
- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

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
- Lots of custom research implementations
 - For GPUs, cell processors, etc.

Cloud Computing Storage, or how do we get data to the workers?



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)

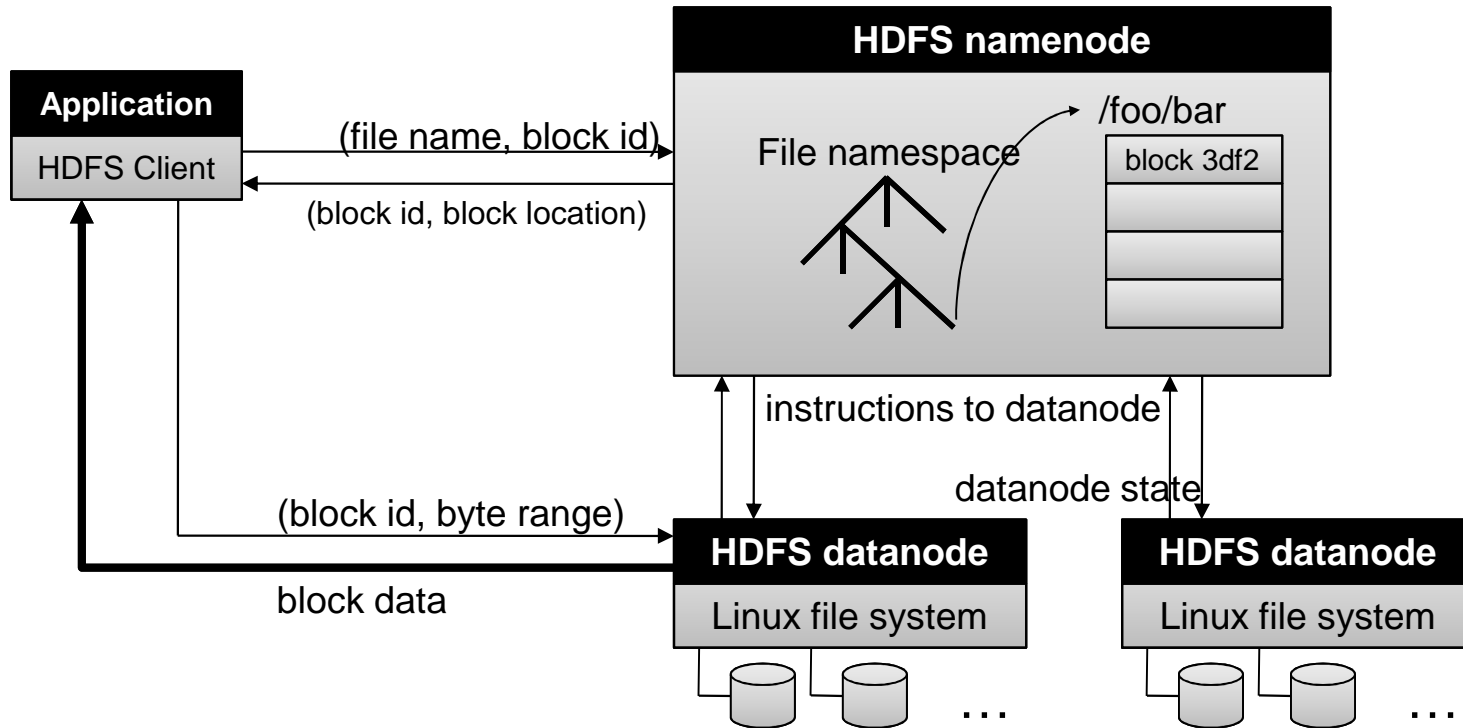
HDFS = GFS clone (same basic ideas implemented in Java)

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

Hadoop Processing Model

- Create or allocate a cluster
- Put data onto the file system
 - Data is split into blocks
 - Replicated and stored in the cluster
- Run your job
 - Copy Map code to the allocated nodes
 - Move computation to data, not data to computation
 - Gather output of Map, sort and partition on key
 - Run Reduce tasks
- Results are stored in the HDFS

Some MapReduce Terminology w.r.t. Hadoop

- Job – A “full program” - an execution of a Mapper and Reducer across a data set
- Task – An execution of a Mapper or a Reducer on a data chunk
 - Also known as Task-In-Progress (TIP)
- Task Attempt – A particular instance of an attempt to execute a task on a machine

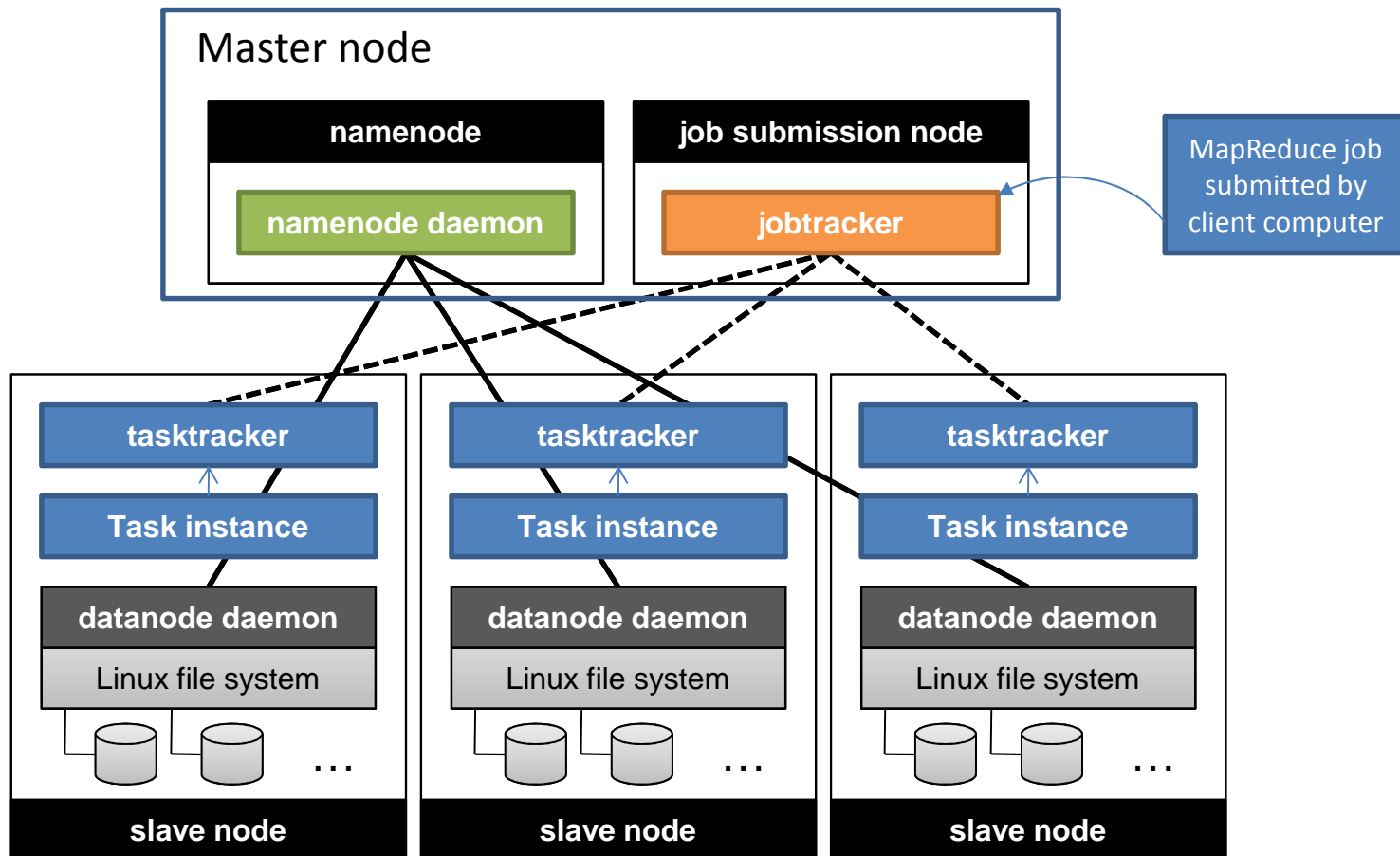
Terminology example

- Running “Word Count” across 20 files is one job
- 20 files to be mapped imply 20 map tasks + some number of reduce tasks
- At least 20 map task attempts will be performed
 - more if a machine crashes or slow, etc.

Task Attempts

- A particular task will be attempted at least once, possibly more times if it crashes
 - If the same input causes crashes over and over, that input will eventually be abandoned
- Multiple attempts at one task may occur in parallel with speculative execution turned on
 - Task ID from *TaskInProgress* is not a unique identifier; don't use it that way

MapReduce: High Level



MapReduce/GFS Summary

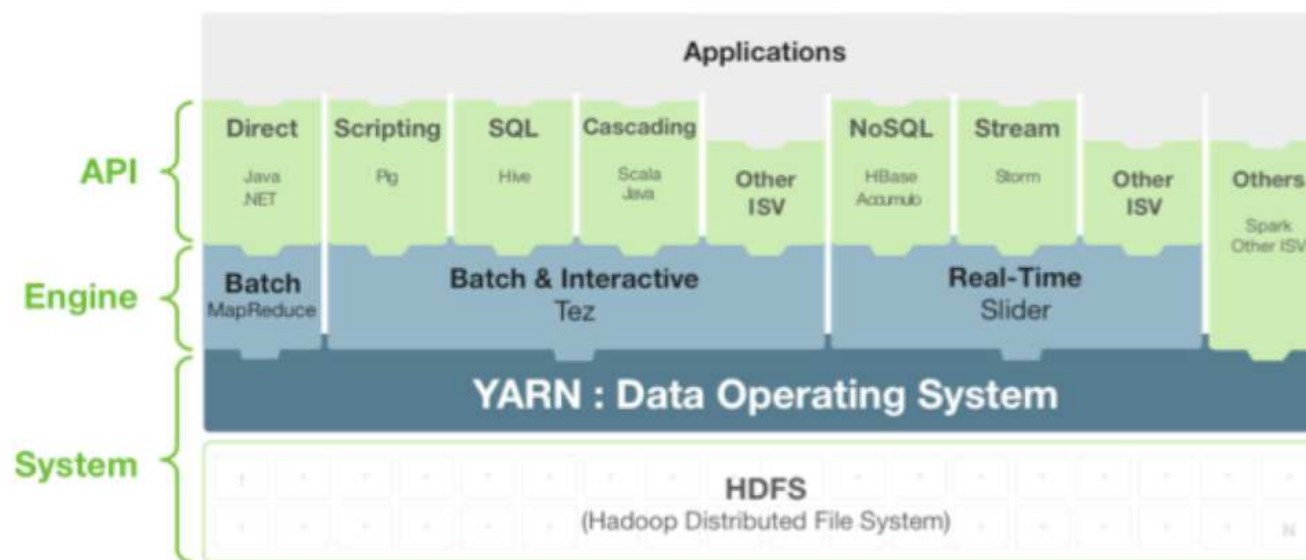
- Simple, but powerful programming model
- Scales to handle petabyte+ workloads
 - Google: six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers
 - Yahoo!: 16.25 hours to sort 1PB on 3,800 computers
- Incremental performance improvement with more nodes
- Seamlessly handles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

Limitations with MapReduce V1

- Scalability
 - Maximum Cluster Size – 4000 Nodes
 - Maximum Concurrent Tasks – 40000
- Coarse synchronization in Job Tracker
 - Single point of failure
 - Failure kills all queued and running jobs
- Jobs need to be resubmitted by users
 - Restart is very tricky due to complex state
- Problems with resource utilization

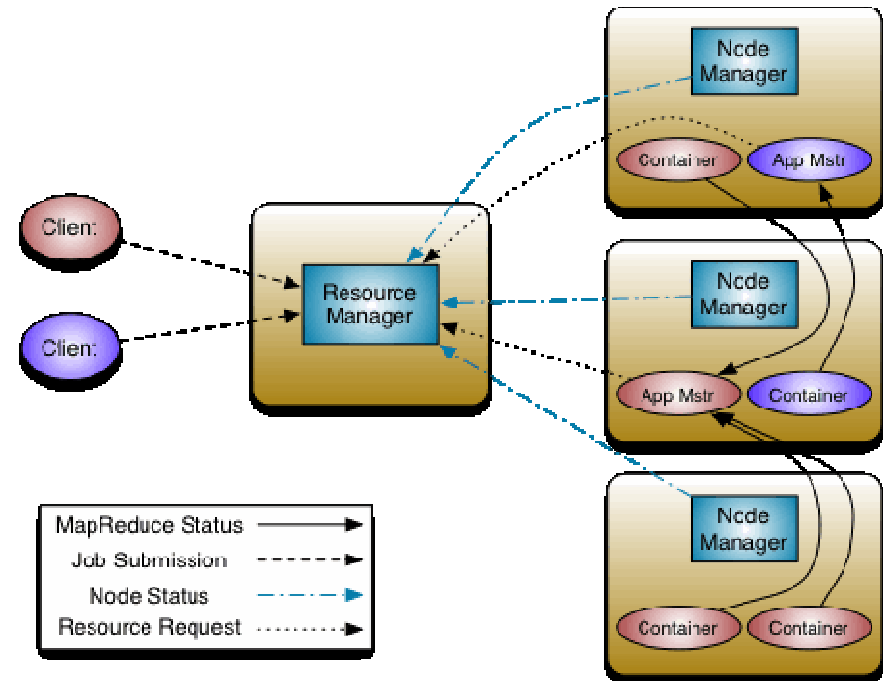
MapReduce NextGen aka YARN aka MRv2

- New architecture introduced in hadoop-0.23
- Divides two major functions of the JobTracker
 - Resource management and job life-cycle management are divided into separate components
- An application is either a single job in the sense of classic MapReduce jobs or a DAG of such jobs



YARN Architecture

- **ResourceManager:**
 - Arbitrates resources among all the applications in the system
 - Has two main components: Scheduler and ApplicationsManager
- **NodeManager:**
 - Per-machine slave
 - Responsible for launching the applications' containers, monitoring their resource usage
- **ApplicationMaster:**
 - Negotiate appropriate resource containers from the Scheduler, tracking their status and monitoring for progress
- **Container:**
 - Unit of allocation incorporating resource elements such as memory, cpu, disk, network etc.
 - To execute a specific task of the application
 - Similar to map/reduce slots in MRv1



Execution Sequence with YARN

- A client program submits the application
- ResourceManager allocates a specified container to start the ApplicationMaster
- ApplicationMaster, on boot-up, registers with ResourceManager
- ApplicationMaster negotiates with ResourceManager for appropriate resource containers
- On successful container allocations, ApplicationMaster contacts NodeManager to launch the container
- Application code is executed within the container, and then ApplicationMaster is responded with the execution status
- During execution, the client communicates directly with ApplicationMaster or ResourceManager to get status, progress updates etc.
- Once the application is complete, ApplicationMaster unregisters with ResourceManager and shuts down, freeing its own container process

Lab related to the lecture

- Work with MapReduce jobs
- Try out WordCount example
 - MapReduce in hadoop-2.x maintains **API compatibility** with previous stable release (hadoop-1.x)
 - MapReduce jobs still run unchanged on top of YARN with just a recompile

Next Lecture

- Let us have a look at some MapReduce algorithms

References

- Hadoop wiki <http://wiki.apache.org/hadoop/>
- Hadoop MapReduce tutorial
http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html
- Data-Intensive Text Processing with MapReduce

Authors: Jimmy Lin and Chris Dyer

<http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf>

- White, Tom. Hadoop: the definitive guide. O'Reilly, 2012.
- J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI'04: Sixth Symposium on Operating System Design and Implementation, Dec, 2004.
- Apache Hadoop YARN -
<https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html>