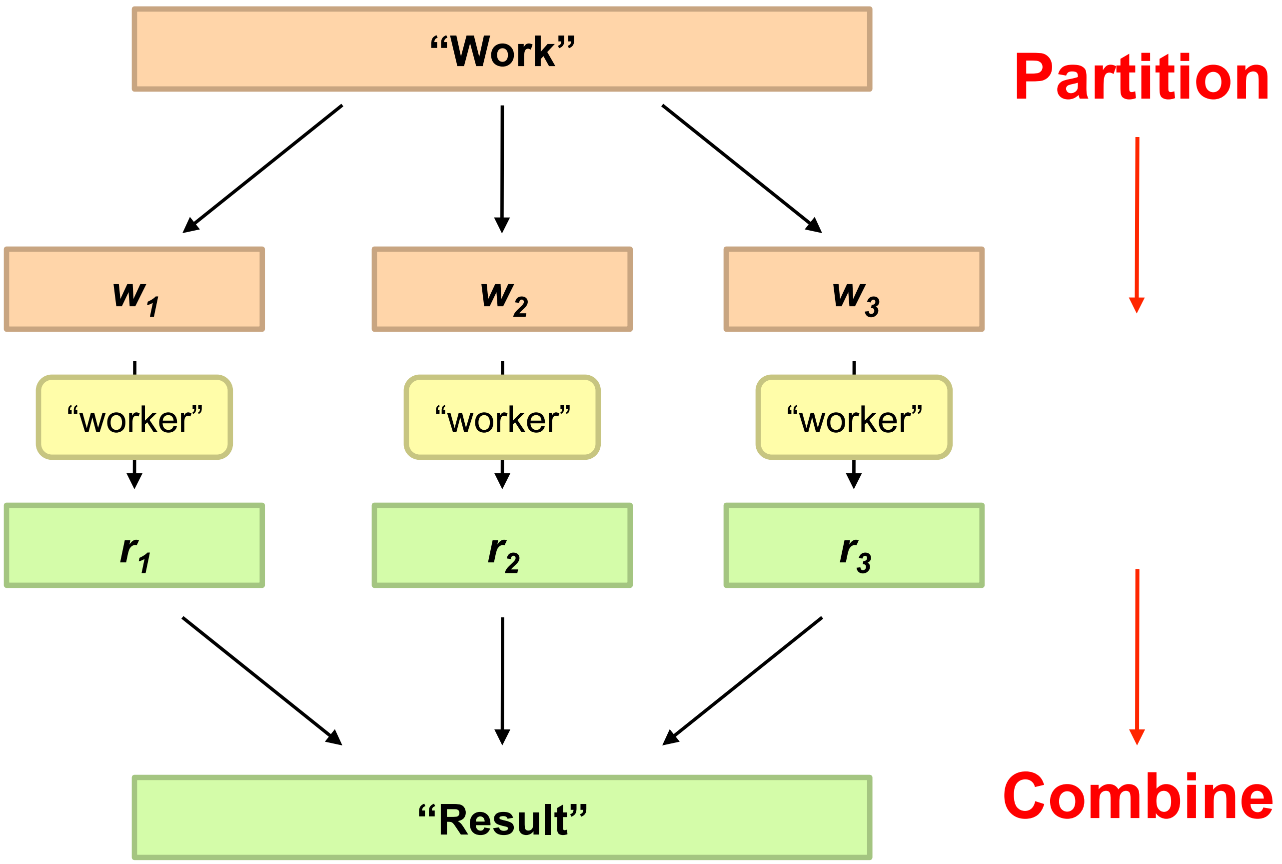
Map-Reduce and Related Systems

Acknowledgement

The slides used in this chapter are adapted from the following sources:

- CS246 Mining Massive Data-sets, by Jure Leskovec, Stanford University, http://www.mmds.org
- ENGG4030 Web-Scale Information Analytics, by Wing Cheong Lau, The Chinese University of Hong Kong,

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

Managing Multiple Workers

- Difficult because
  - We don’t know the order in which workers run
  - We don’t know when workers interrupt each other
  - We don’t know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

What's the point?

- It's all about the right level of abstraction
  - The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

The datacenter *is* the computer!

“Big Ideas”

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

Google MapReduce

- Framework for parallel processing in large-scale shared-nothing architecture
- Developed initially (and patented) by Google to handle Search Engine's webpage indexing and page ranking in a more systematic and maintainable fashion
- Why NOT using existing Database (DB)/ Relational Database Management Systems (RDMS) technologies?

  Mismatch of Objectives

  - DB/ RDMS were designed for high-performance transactional processing to support hard guarantees on consistencies in case of MANY concurrent (often small) updates, e.g. ebanking, airline ticketing ; DB Analytics were "secondary" functions added on later ;
  - For Search Engines, the documents are never updated (till next Web Crawl) and they are Read-Only ; It is ALL about Analytics !
  - Import the webpages, convert them to DB storage format is expensive
  - The Job was simply too big for prior DB technologies !

Typical BigData 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

(Dean and Ghemawat, OSDI 2004)

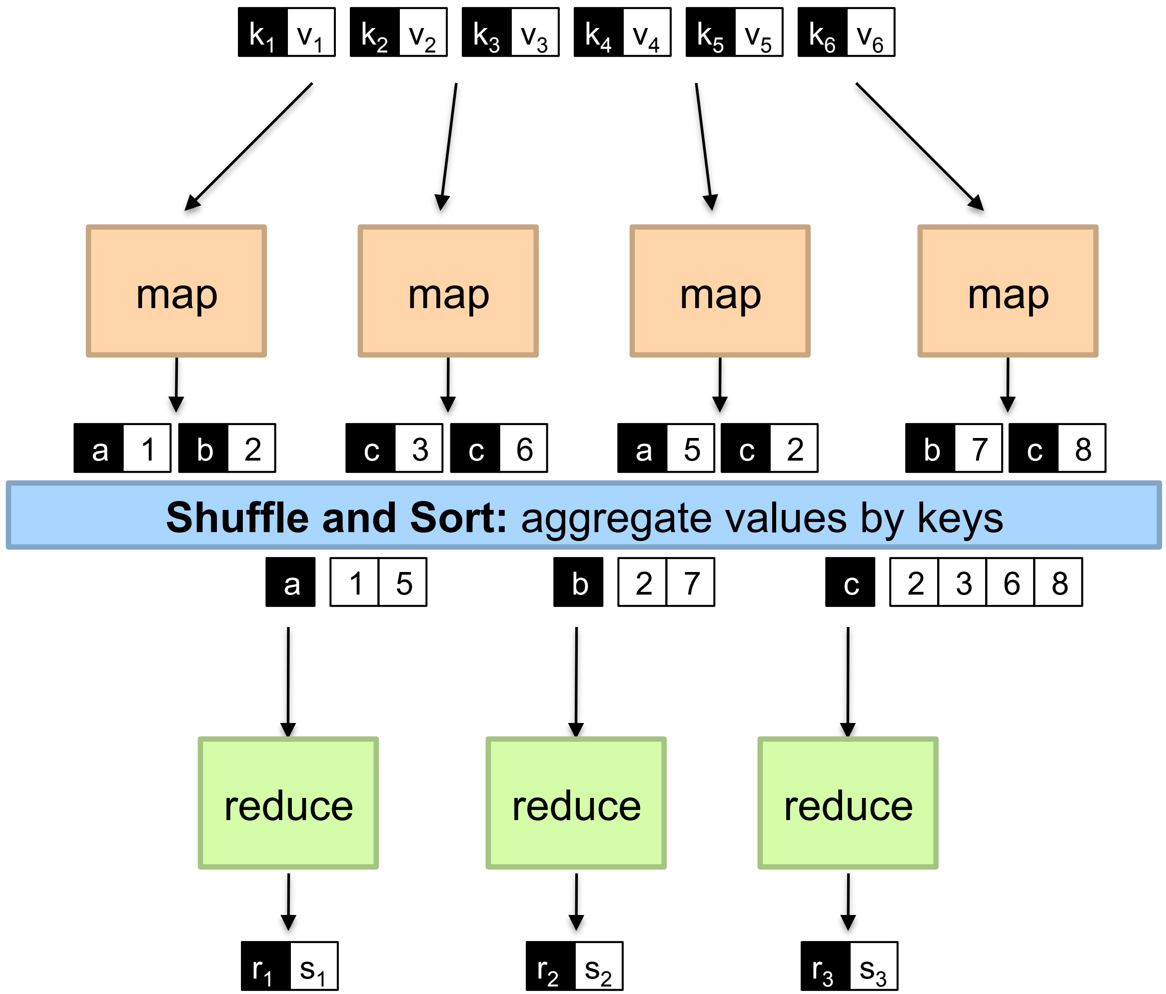
MapReduce

- Programmers specify two functions:

  map (k, v) → <k’, v’>*

  reduce (k’, v’) → <k’’, v’’>*

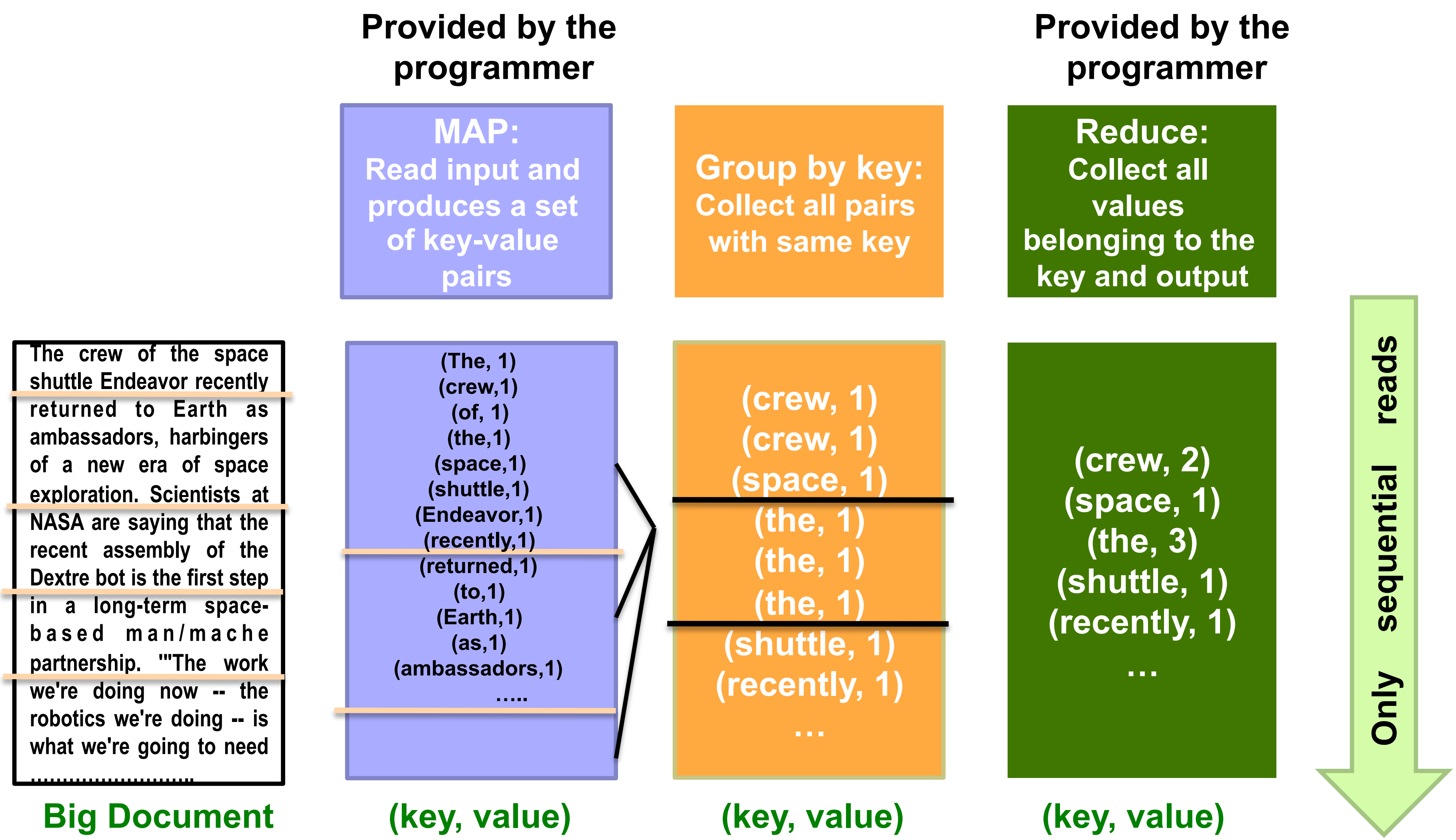
  - All values with the same key are sent to the same reducer
  - <a,b>* means a list of tuples in the form of (a,b)
- The execution framework handles everything else…


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

"Hello World" Task for MapReduce: Word Counting

- Unix/Linux shell command to Count occurrences of words in a file named `doc.txt`:
  - **`words(doc.txt) | sort | uniq -c`**
    - where **`words`** takes a file and outputs the words in it, one per a line
    - "**`uniq`**" stands for unique, is a true Unix command ; see its manpage to find out what "**`uniq -c`**" does
- The above "Unix/Linux-shell command" captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Word Counting



“Hello World”: Pseudo-code for Word Count

```
Map(String docid, String text):
// docid: document name, i.e. the input key ;
// text:  text in the document, i.e. the input value
    for each word w in text:
        EmitIntermediate(w, 1);

Reduce(String term, Iterator<Int> Ivalues):
// term: a word, i.e. the intermediate key,  also happens to be the output key here ;
// Ivalues: an iterator over counts (i.e. gives the list of intermediate values from Map)
    int sum = 0;
    for each v in Ivalues:
        sum += v ;
    Emit(term, sum);
```

// The above is pseudo-code only ! True code is a bit more involved: needs to define how the input key/values are divided up and accessed, etc).

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 restarts
- Everything happens on top of a distributed File System (later)

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k'', v''>*

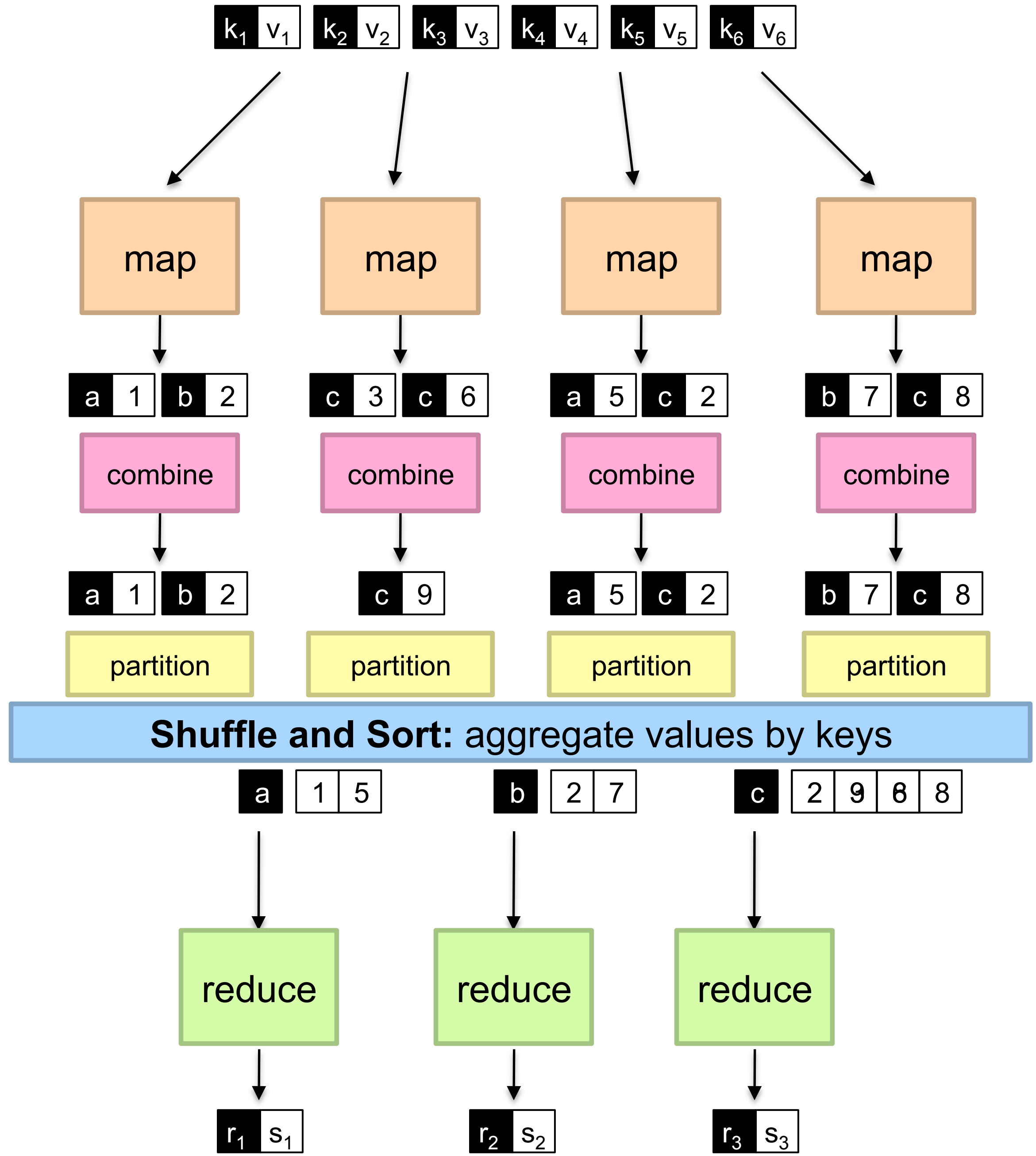
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  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
  - **Sometimes useful to override the hash function:**
    - e.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

  combine (k', v') → <k', v'>*

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic
  - Works only if Reduce function is Commutative and Associative



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 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Hadoop Streaming

- To enjoy the convenience brought by Hadoop, one has to implement mapper and reducer in Java
  - Hadoop defines a lot of data types and complex class hierarchy
  - There is a learning curve
- Hadoop streaming allows you to use any language to write the mapper and reducer

Hadoop Streaming

- Using Hadoop Streaming, you need to write
  - Mapper
    - Read input from standard input (STDIN)
    - Write map result to standard output (STDOUT)
      - Key value are separated using tab
  - Group by key
    - Done by Hadoop
  - Reducer
    - Read input (Mapper's output) from standard input (STDIN)
    - Write output (Final result) to standard output (STDOUT)

Hadoop Streaming

- Allows you to start writing MapReduce application that can be readily deployed without having to learn Hadoop class structure and data types
- Speed up development
- Utilize rich features and handy libraries from other languages (Python, Ruby)
- Efficiency critical application can be implemented in efficient language (C, C++)

Hadoop Streaming: Word Count Mapper

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Hadoop Streaming: Word Count Reducer

```
#!/usr/bin/env python
from operator import itemgetter
import sys
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    try:
        count = int(count)
    except ValueError:
        continue
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Hadoop Streaming: How to Run?

- To run the sample code

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
-input inputPathonHDFS \
-output outputPathonHDFS \
-file pathToMapper.py \
-mapper mapper.py \
-file pathToReducer.py \
-reducer reducer.py
```

- -file caches the argument to every tasktracker
- The above command distribute the mapper.py and reducer.py to every tasktracker

Hadoop Streaming: Word Count

```
#!/usr/bin/env python
"""A more advanced Mapper, using Python iterators and generators."""

import sys
def read_input(file):
    for line in file:
        yield line.split()
def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_input(sys.stdin)
    for words in data:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        for word in words:
            print '%s%s%d' % (word, separator, 1)
if __name__ == "__main__":
    main()
```

Hadoop Streaming: Word Count

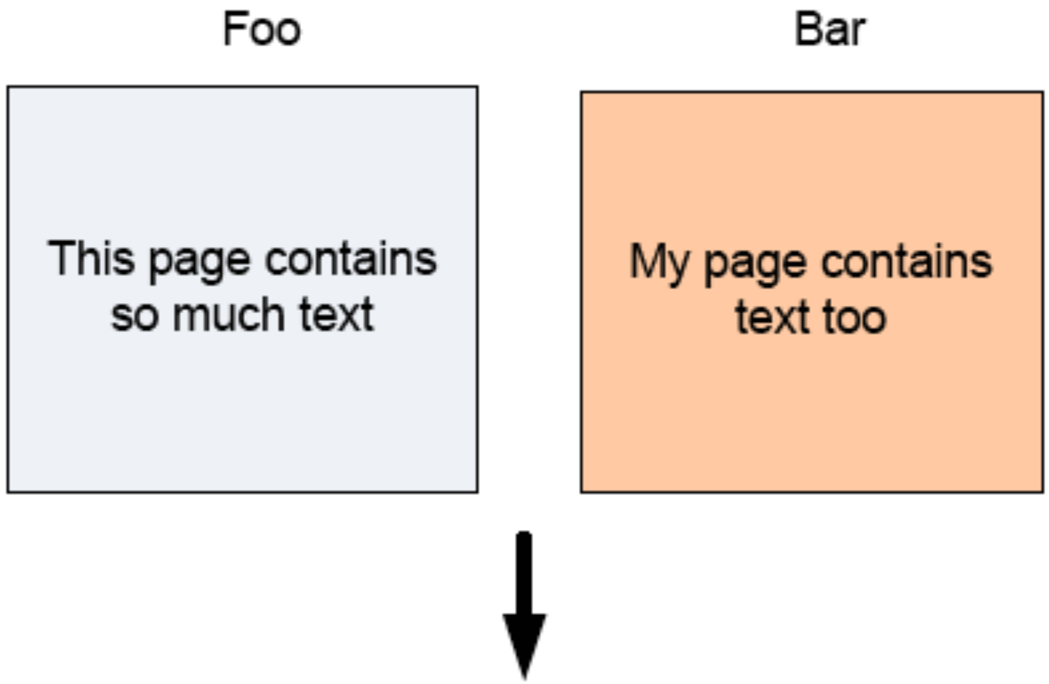
```
#!/usr/bin/env python
"""A more advanced Reducer, using Python iterators and generators."""

from itertools import groupby
from operator import itemgetter
import sys
def read_mapper_output(file, separator='\t'):
    for line in file:
        yield line.rstrip().split(separator, 1)
def main(separator='\t'):
    # input comes from STDIN (standard input)
    data = read_mapper_output(sys.stdin, separator=separator)
    # groupby groups multiple word-count pairs by word,
    # and creates an iterator that returns consecutive keys and their group:
    #   current_word - string containing a word (the key)
    #   group - iterator yielding all ["&lt;current_word&gt;", "&lt;count&gt;"] items
    for current_word, group in groupby(data, itemgetter(0)):
        try:
            total_count = sum(int(count) for current_word, count in group)
            print "%s%s%d" % (current_word, separator, total_count)
        except ValueError:
            # count was not a number, so silently discard this item
            pass
if __name__ == "__main__":
    main()
```

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

Example 2: Inverted Index (for a Search Engine)



contains: Foo, Bar
much: Foo
My: Bar
page : Foo, Bar
so : Foo
text: Foo, Bar
This : Foo
too: Bar

Inverted Index with MapReduce

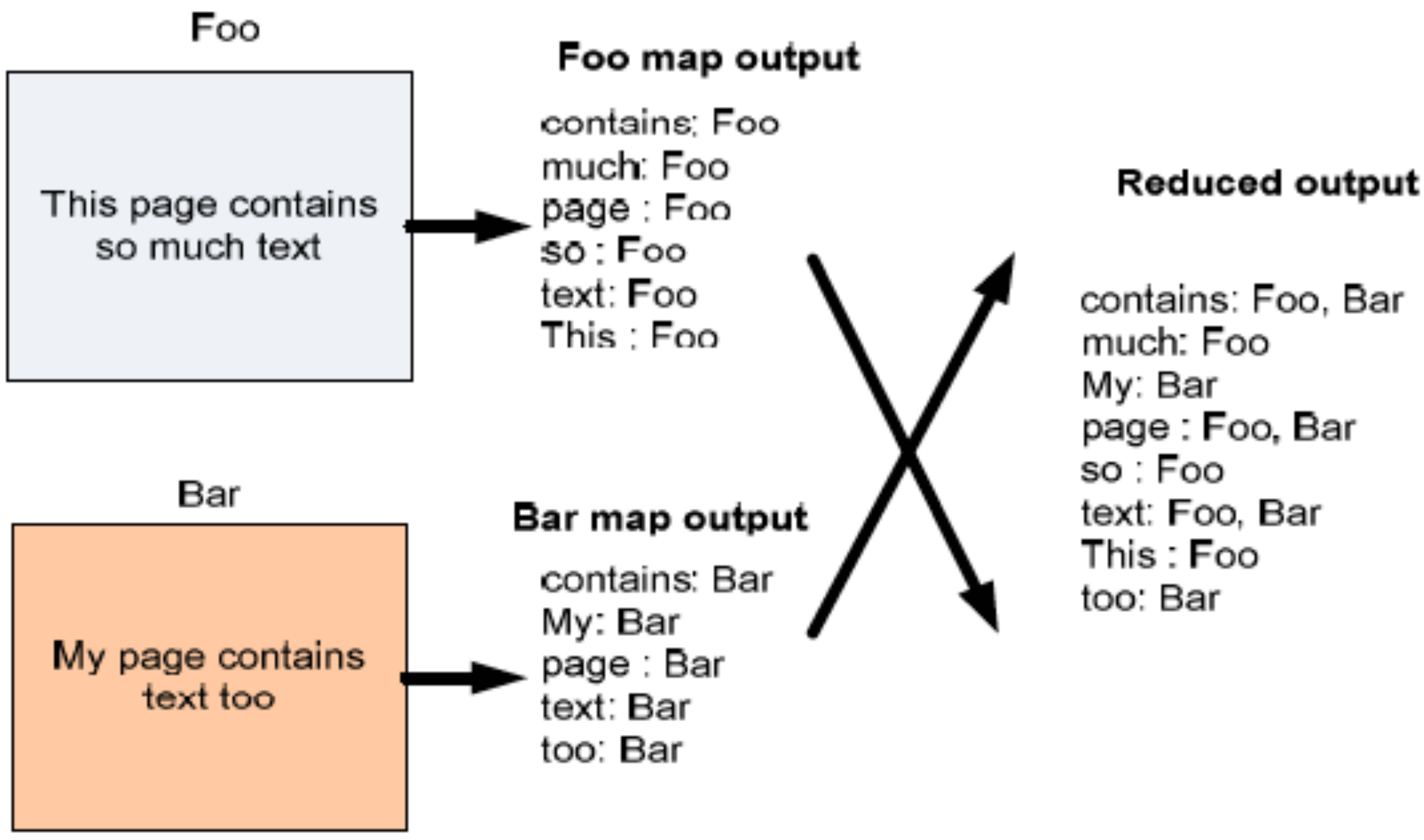
- Mapper:
  - Key: PageName   // URL of webpage
  - Value: Text   // text in the webpage

  foreach word w in Text

    EmitIntermediate(w, PageName)

- Reducer:
  - Key: word
  - Values: all URLs for word
  - … Just the Identity function

Inverted Index Data flow w/ MapReduce



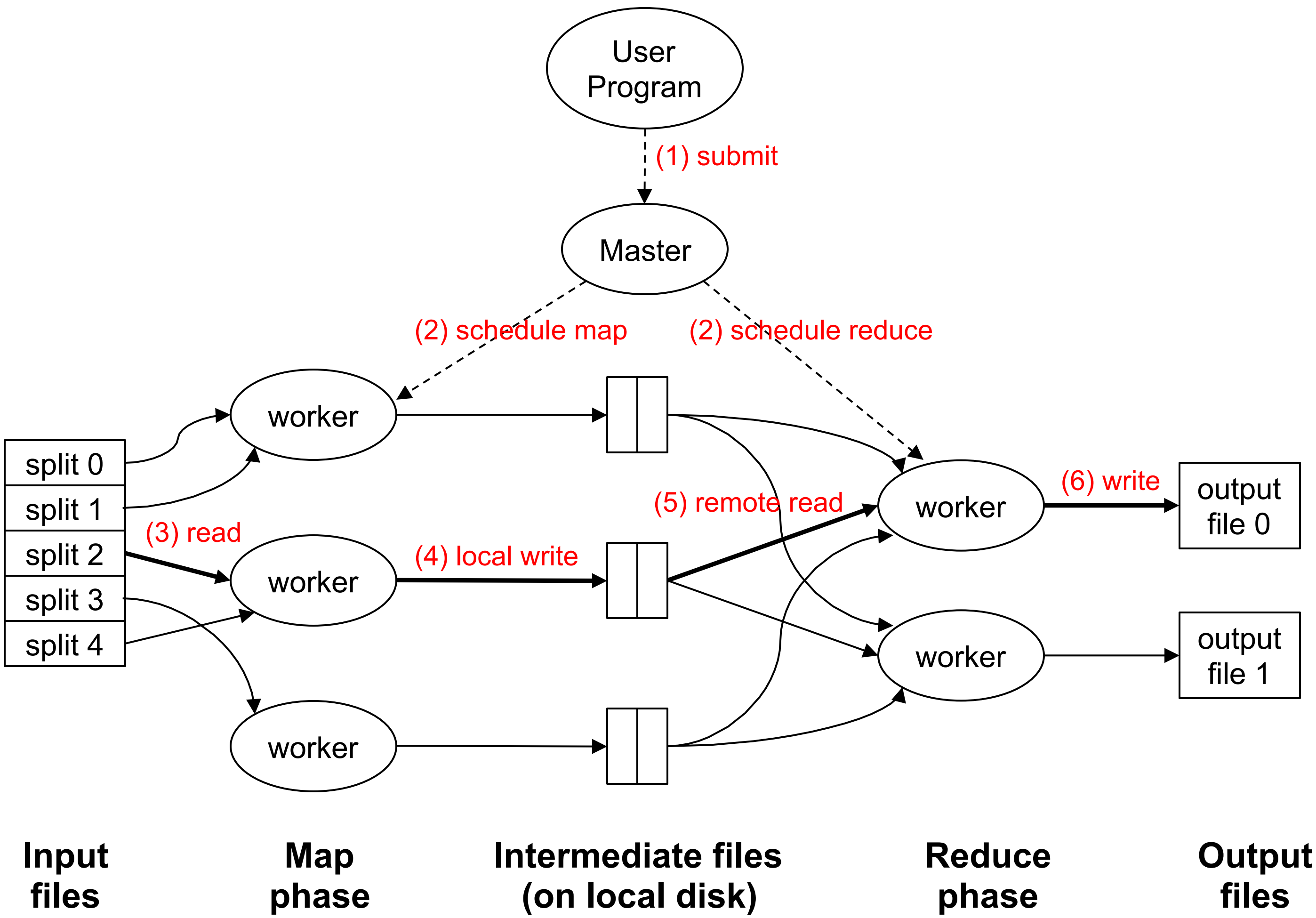
MapReduce can refer to…

- 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.



Adapted from (Dean and Ghemawat, OSDI 2004)

Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data

- **Intermediate results are stored on local FS of Map and Reduce workers**

- **Output is often input to another MapReduce task**

Coordination: Master

- **Master node takes care of coordination:**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Dealing with Failures

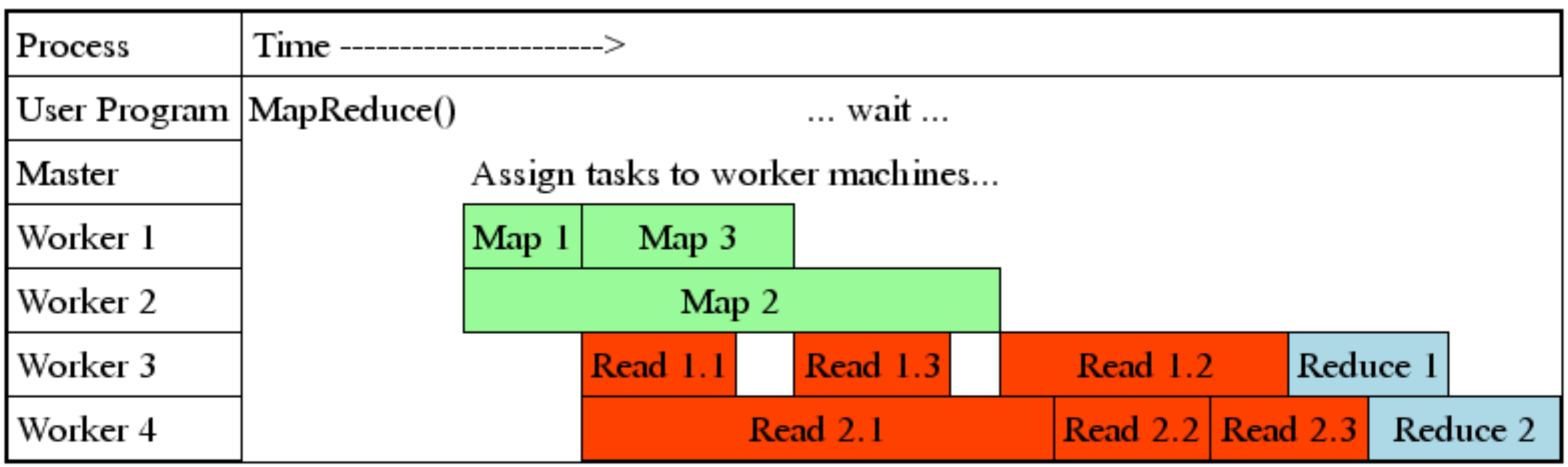
- **Map worker failure**
  - Map tasks completed (Why ??) or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk (64 Mbyte each by default) per mapper is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Task Granularity & Pipelining

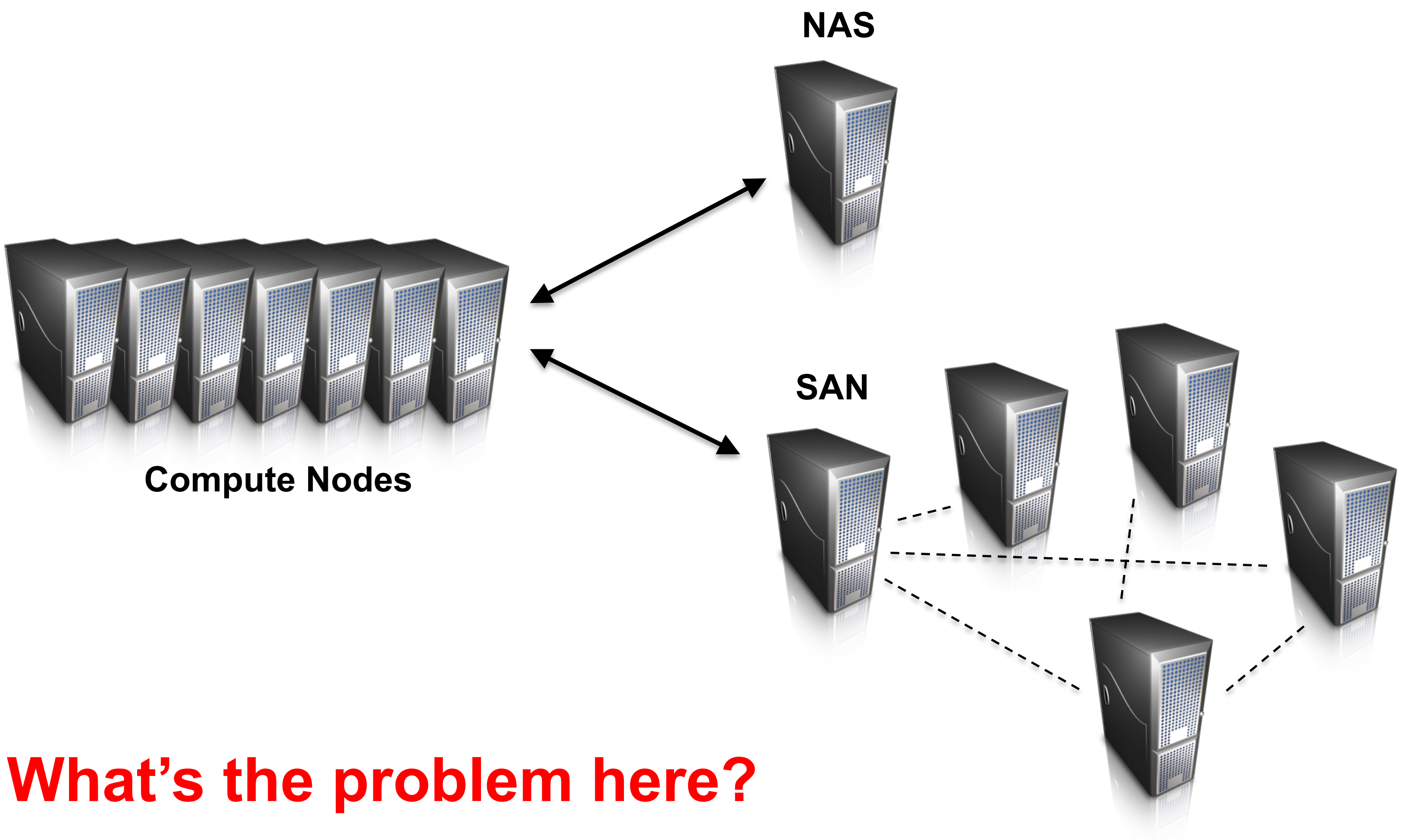
- **Fine granularity tasks:** # of map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing
  - e.g. For 2000 processors, M = 200,000 ; R = 5000



Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first "wins"
- **Effect**
  - Dramatically shortens job completion time

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
  - Non-starters
    - Lustre (high bandwidth, but no replication outside racks)
    - Gluster (POSIX, more classical mirroring, see Lustre)
    - NFS/AFS/whatever - doesn't actually parallelize

GFS: Assumptions

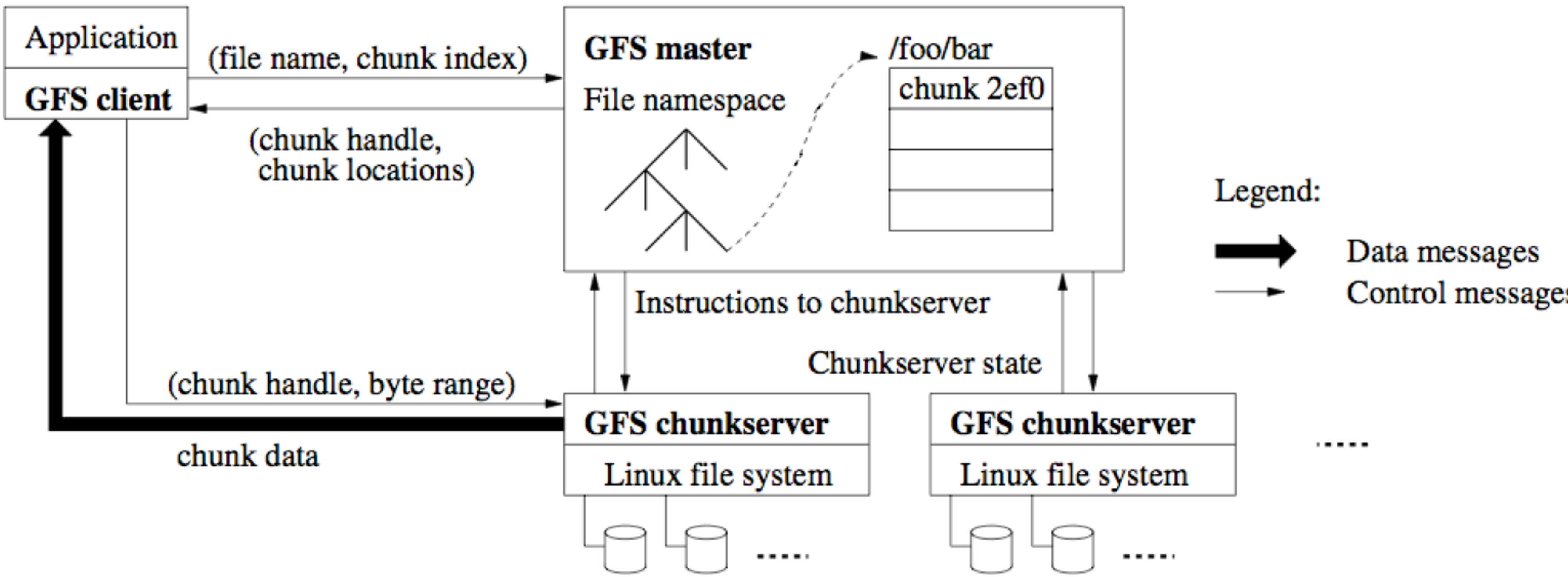
- Commodity hardware over "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- 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)

Google File System

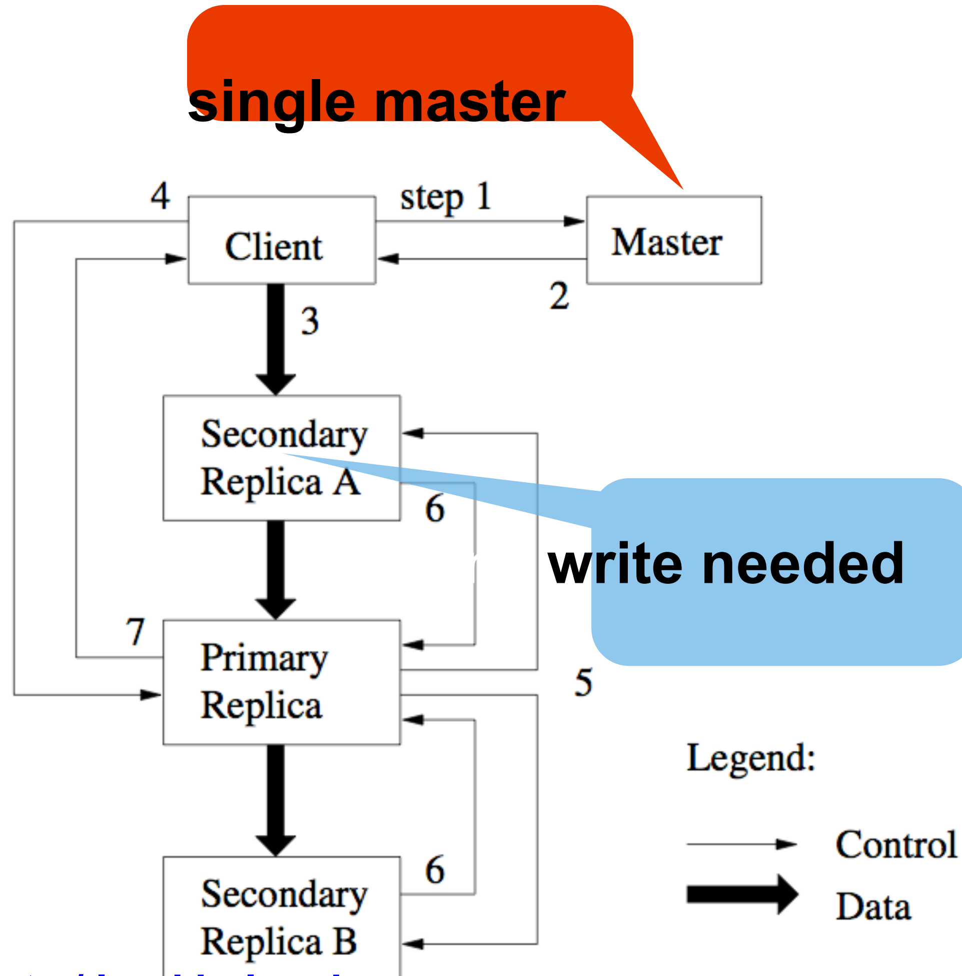


Ghemawat, Gobioff, Leung, 2003

- Chunk servers hold blocks of the file (64MB per chunk)
- Replicate chunks (chunk servers do this autonomously). More bandwidth and fault tolerance
- Master distributes, checks faults, rebalances (Achilles heel)
- Client can do bulk read / write / random reads

Google File System /HDFS

1. **Client requests chunk from master**
2. **Master responds with replica location**
3. **Client writes to replica A**
4. **Client notifies primary replica**
5. **Primary replica requests data from replica A**
6. **Replica A sends data to Primary replica (same process for replica B)**
7. **Primary replica confirms write to client**



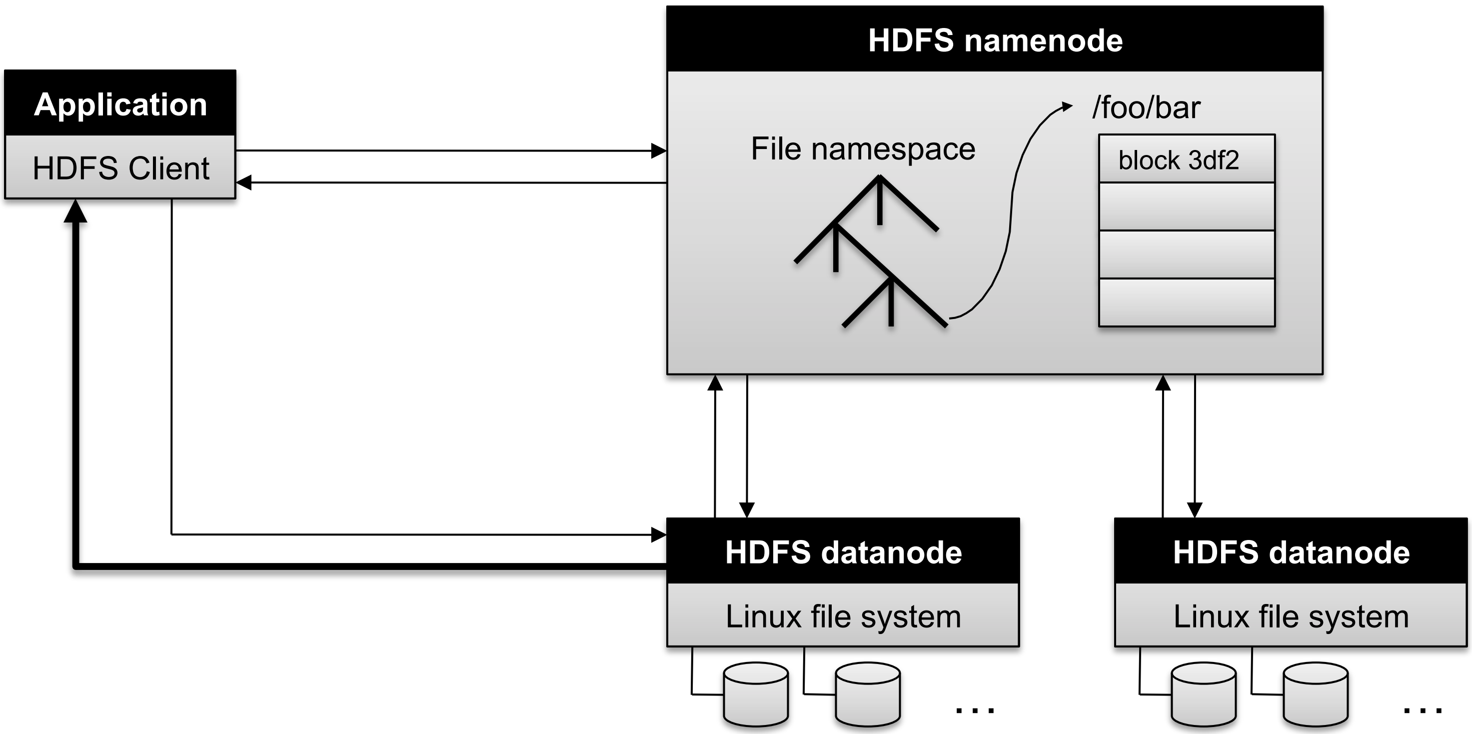
- **Master ensures nodes are live**
- **Chunks are checksummed**
- **Can control replication factor for hotspots / load balancing**
- **Deserialize master state by loading data structure as flat file from disk (fast) ;** *See Section 4.1 of GFS SOSP2003 paper for details*

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Functional differences:
  - Initially, no file appends in HDFS (the feature has been added recently)
    - http://blog.cloudera.com/blog/2009/07/file-appends-in-hdfs/
    - http://blog.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/
  - HDFS performance is (likely) slower

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

HDFS Architecture

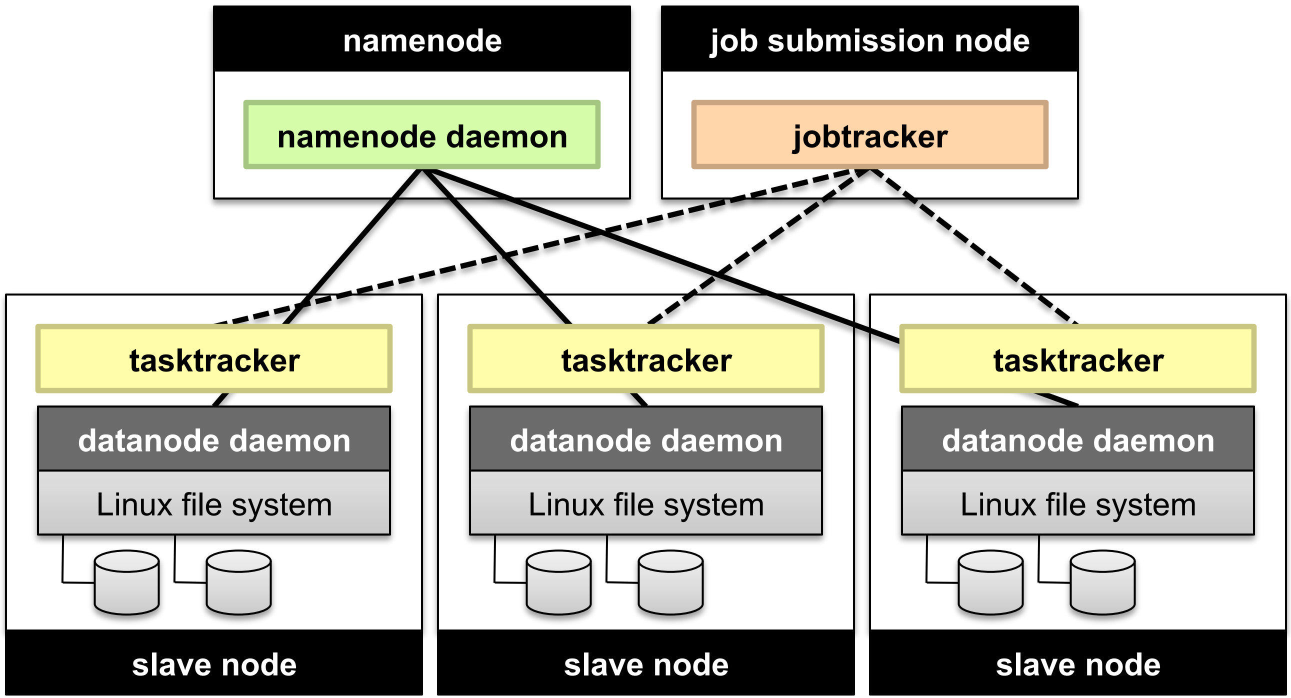


Adapted from (Ghemawat et al., SOSP 2003)

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
- Namenode can be Archille's heel – Single point of failure or bottleneck of scalability for the entire FS:
  - Need to have a Backup Namenode HDFS (or Master in GFS)
  - Compared to the fully-distributed approach in Ceph

Putting everything together…



Sample Use of MapReduce

More MapReduce Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host
- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms
  - More later in the course…

Another Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **With MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join

- **Total communication cost**
  $= O(|R|+|S|+|R \bowtie S|)$
- **Elapsed communication cost** $= O(s)$
  - We're going to pick ***k*** and the number of Map processes so that the I/O limit ***s*** is respected
  - We put a limit ***s*** on the amount of input or output that any one process can have. ***s* could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost

MapReduce is good for…

- *Embarrassingly Parallel* algorithms
- Summing, grouping, filtering, joining
- Off-line batch jobs on massive data sets
- Analyzing an entire large data set
  - New higher level languages/systems have been developed to further simplify data processing using MapReduce
    - Declarative description (NoSQL type) of processing task can be translated automatically to MapReduce functions
    - Control flow of processing steps (Pig)

MapReduce is OK, (and only ok) for…

- Iterative jobs (e.g. Graph algorithms like Pagerank)
  - Each iteration must read/write data to disk
  - I/O and latency cost of an iteration is high

MapReduce is NOT good for…

- Jobs that need shared state/ coordination
  - Tasks are shared-nothing
  - Shared-state requires scalable state store
- Low-latency jobs
- Jobs on small datasets
- Finding individual records

For some of these, we will introduce alternative computational models/ platforms, e.g. GraphLab, Spark, later in the course

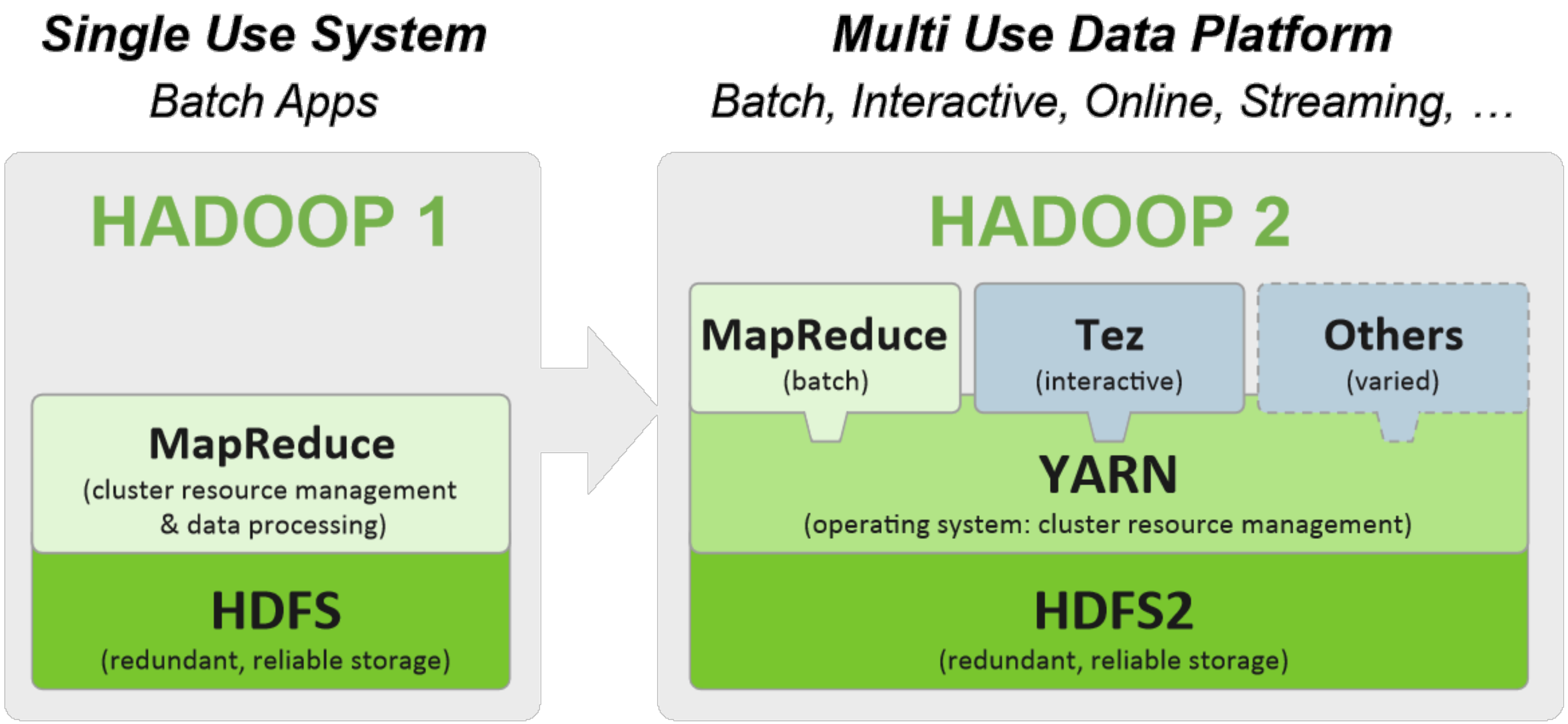
Scalability/Flexibility Issues of the MapReduce/ Hadoop 1.0 Job Scheduling/Tracking

- The MapReduce Master node (or Job-tracker in Hadoop 1.0) is responsible to monitor the progress of ALL tasks of all jobs in the system and launch backup/replacement copies in case of failures
  - For a large cluster with many machines, the number of tasks to be tracked can be huge

  => Master/Job-Tracker node can become the performance bottleneck
- Hadoop 1.0 platform focuses on supporting MapReduce as its only computational model ; may not fit all applications
- Hadoop 2.0 introduces a new resource management/ job-tracking architecture, YARN [1], to address these problems

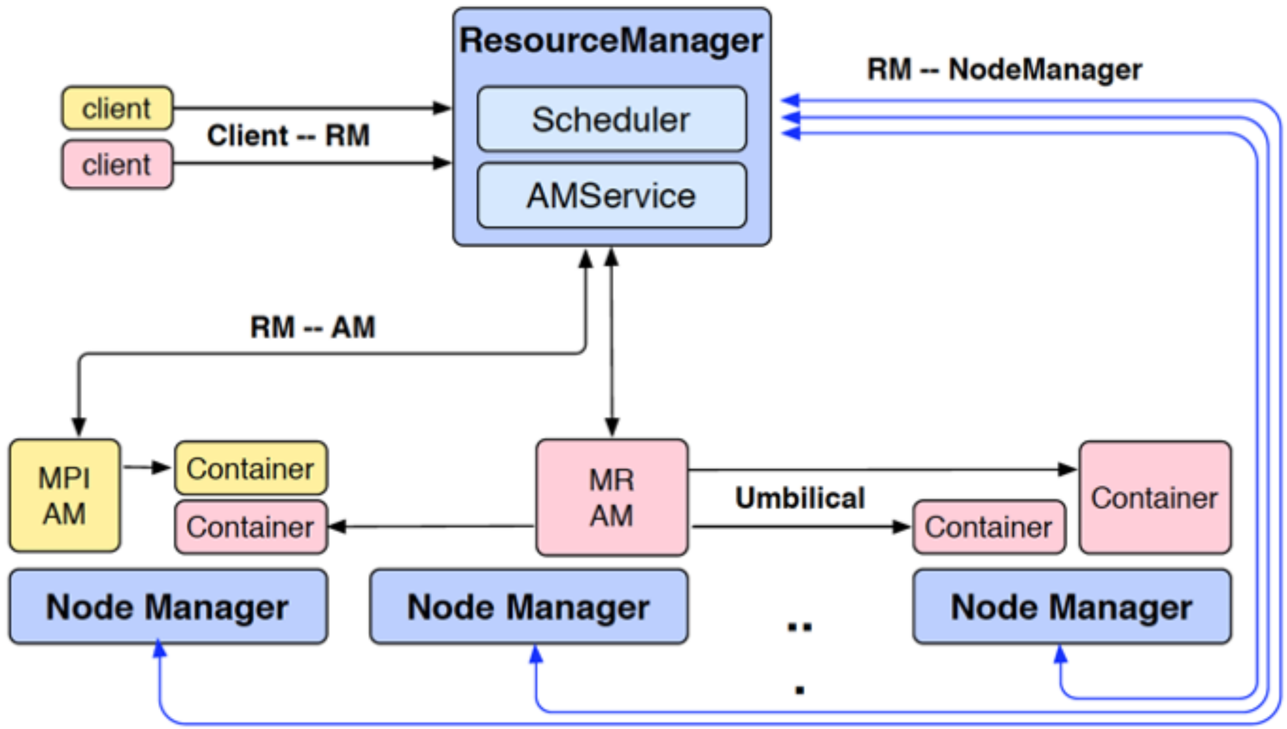
[1] V.K. Vavilapalli, A.C.Murthy, "Apache Hadoop YARN: Yet Another Resource Negotiator," ACM Symposium on Cloud Computing 2013.

YARN for Hadoop 2.0



- YARN provides a resource management platform for general Distributed/ Parallel Applications beyond the MapReduce computational model.

YARN for Hadoop 2.0



- Multiple frameworks (Applications) can run on top of YARN to share a Cluster, e.g. MapReduce is one framework (Application), MPI, or Storm are other ones.
- YARN splits the functions of JobTracker into 2 components: resource allocation and job-management (e.g. task-tracking/ recovery):
  - Upon launching, each Application will have its own Application Master (AM), e.g. MR-AM in the figure above is the AM for MapReduce, to track its own tasks and perform failure recovery if needed
  - Each AM will request resources from the YARN Resource Manager (RM) to launch the Application's jobs/tasks (Containers in the figure above) ;
  - The YARN RM determines resource allocation across the entire cluster by communicating with/ controlling the Node Managers (NM), one NM per each machine.