

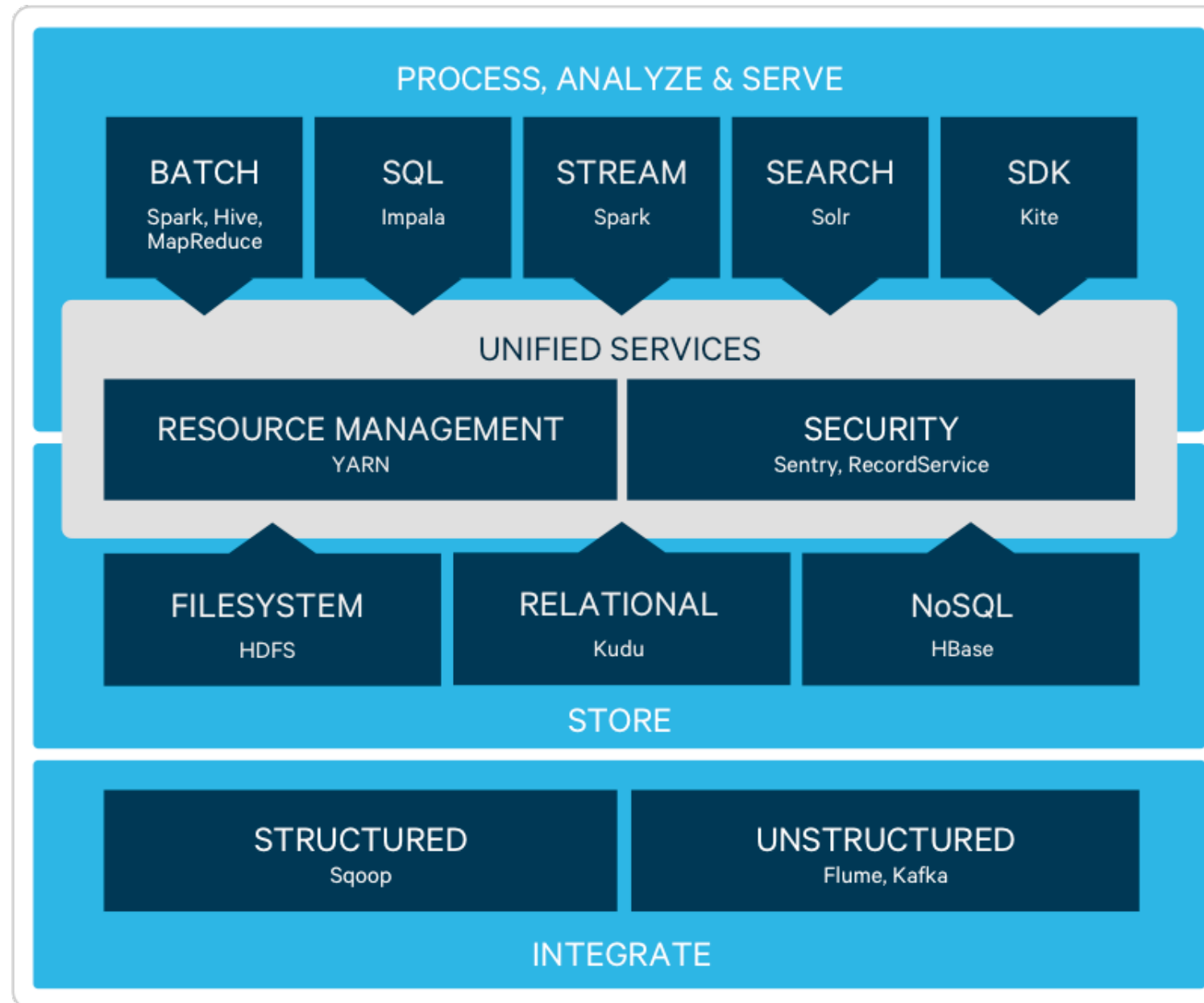


- Components
- How to Hadoop
- Examples/Patterns
- Labs



- Google File System, 2003:
<http://research.google.com/archive/gfs.html>
- Map Reduce, 2004:
<http://research.google.com/archive/mapreduce.html>
- Doug Cutting, Yahoo: 2004. Now with Cloudera

Hadoop



What is Hadoop/MapReduce?

- Programming model for expressing distributed computations at a massive scale
- Execution framework for organizing and performing such computations
- Open-source implementation called Hadoop



MapReduce can refer to...

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

Usage is usually clear from context!

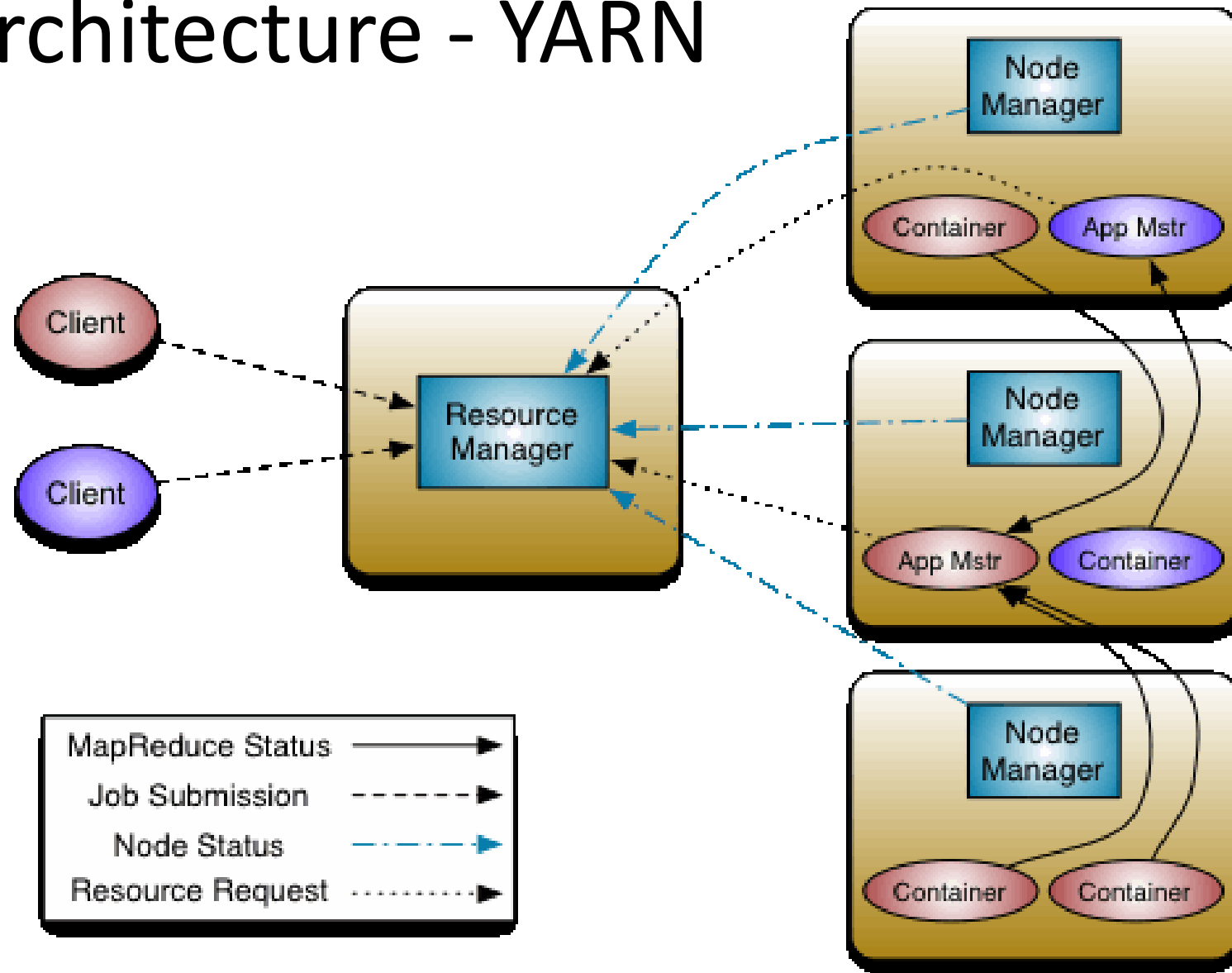
Apache Hadoop

- Apache Hadoop Project : <http://hadoop.apache.org/docs/current/>

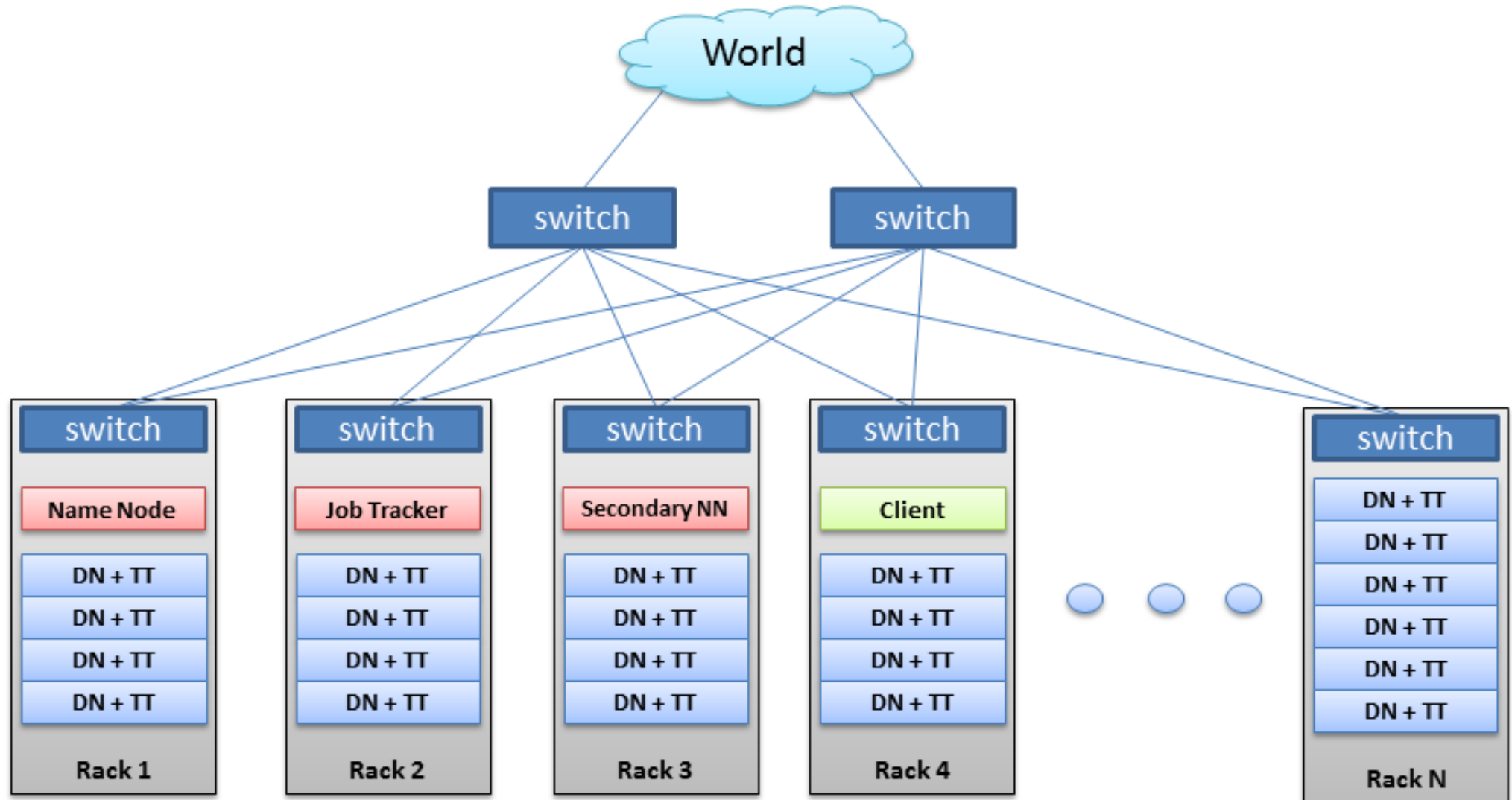
A software stack for reliable, scalable, distributed computing

- Core components
 - Hadoop Common
 - Hadoop Distributed File System - HDFS
 - Hadoop YARN
 - Hadoop MapReduce
- Hadoop Related projects: Ambari, Avro, Cassandra, HBase, Hive, Mahout, Pig, Spark, Tez, ZooKeeper, Hue, ...?

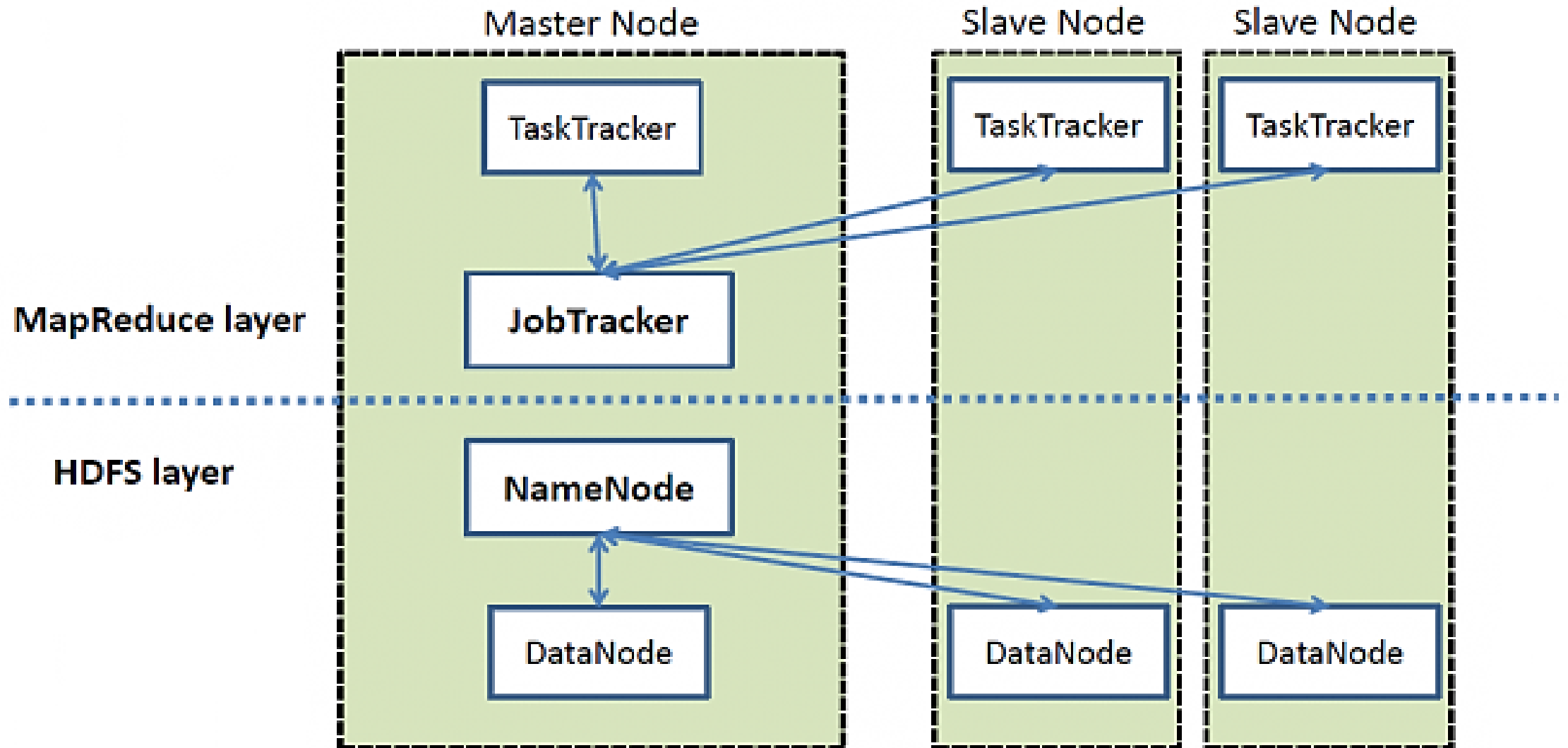
Hadoop Architecture - YARN



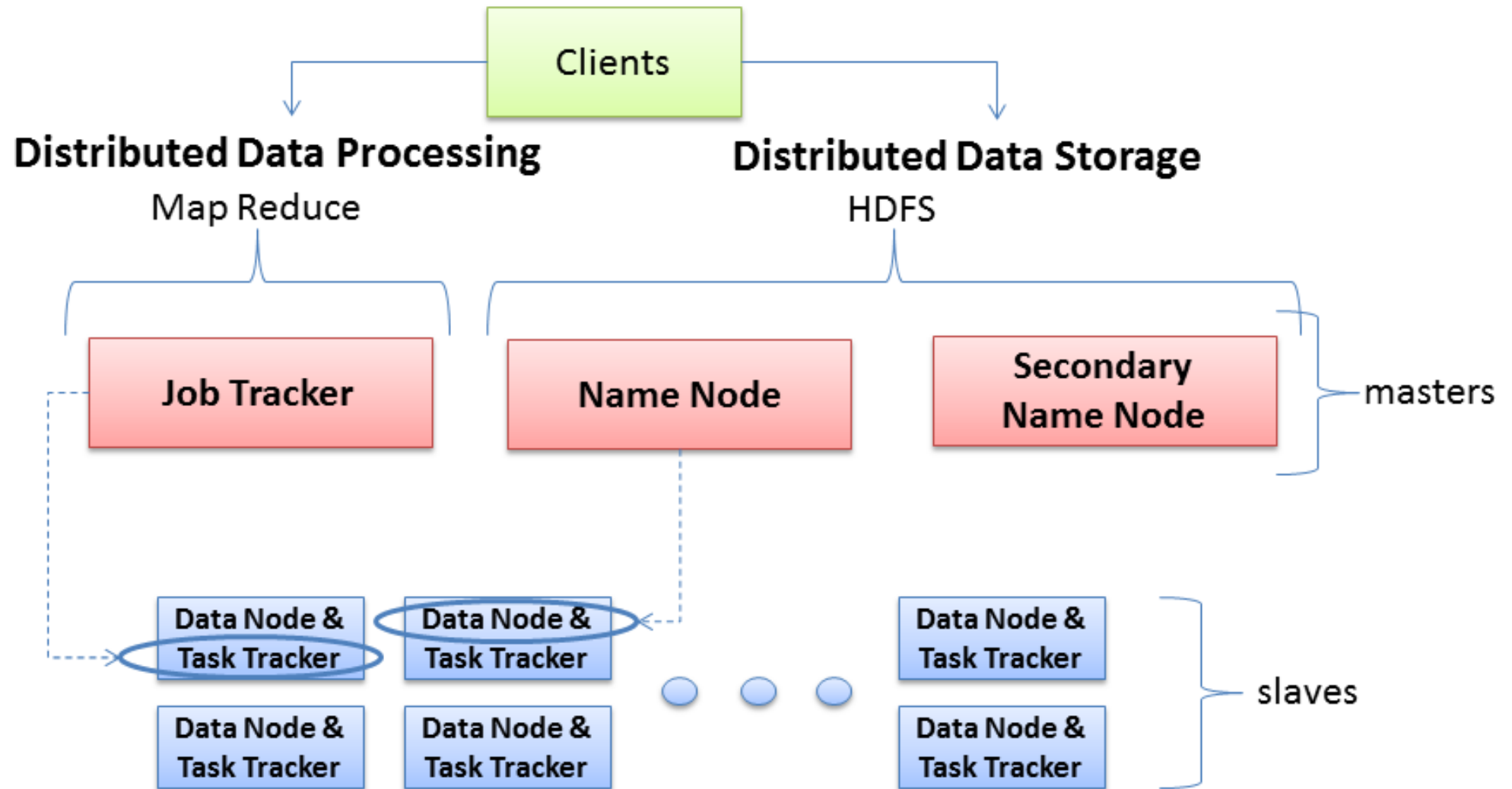
Hadoop Architecture/ YARN



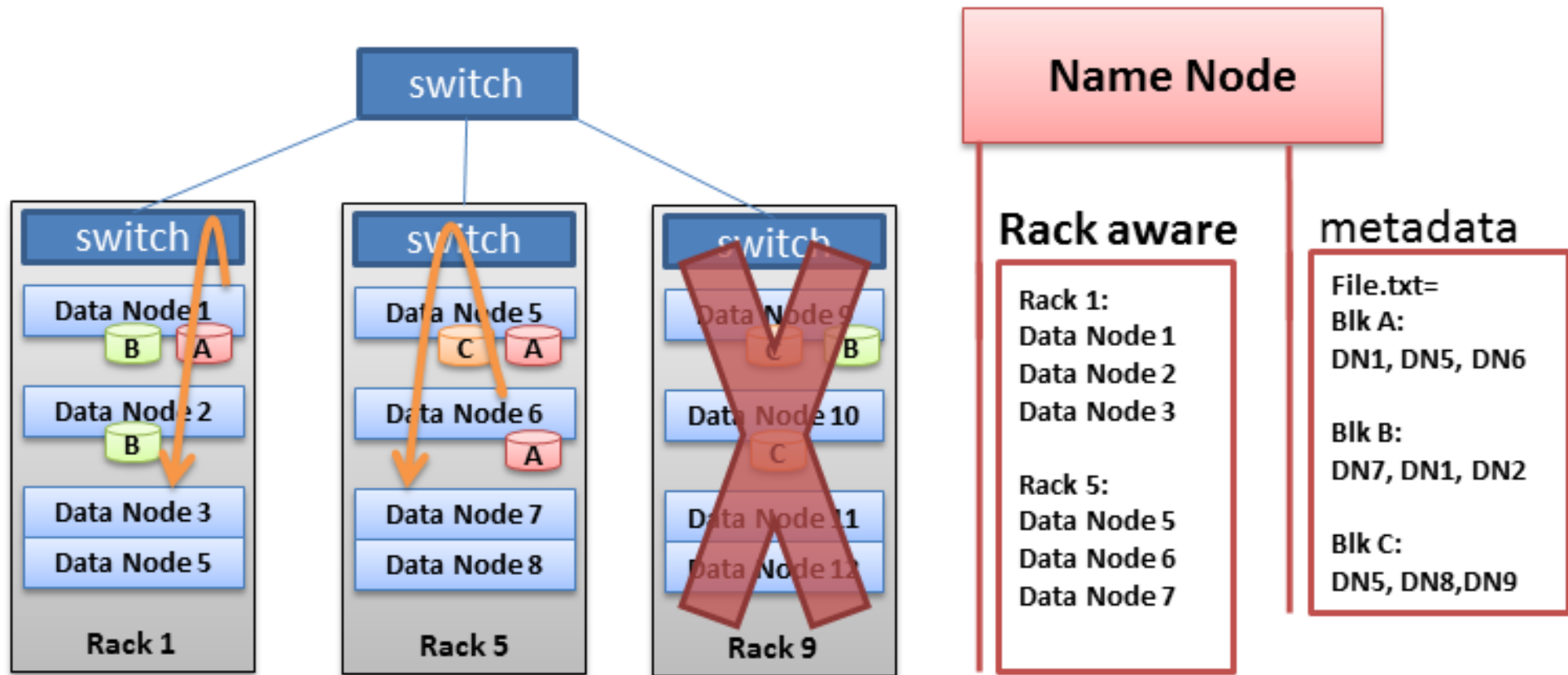
Hadoop Architecture/HDFS, MapReduce



Hadoop Architecture / HDFS



Hadoop Architecture / HDFS



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

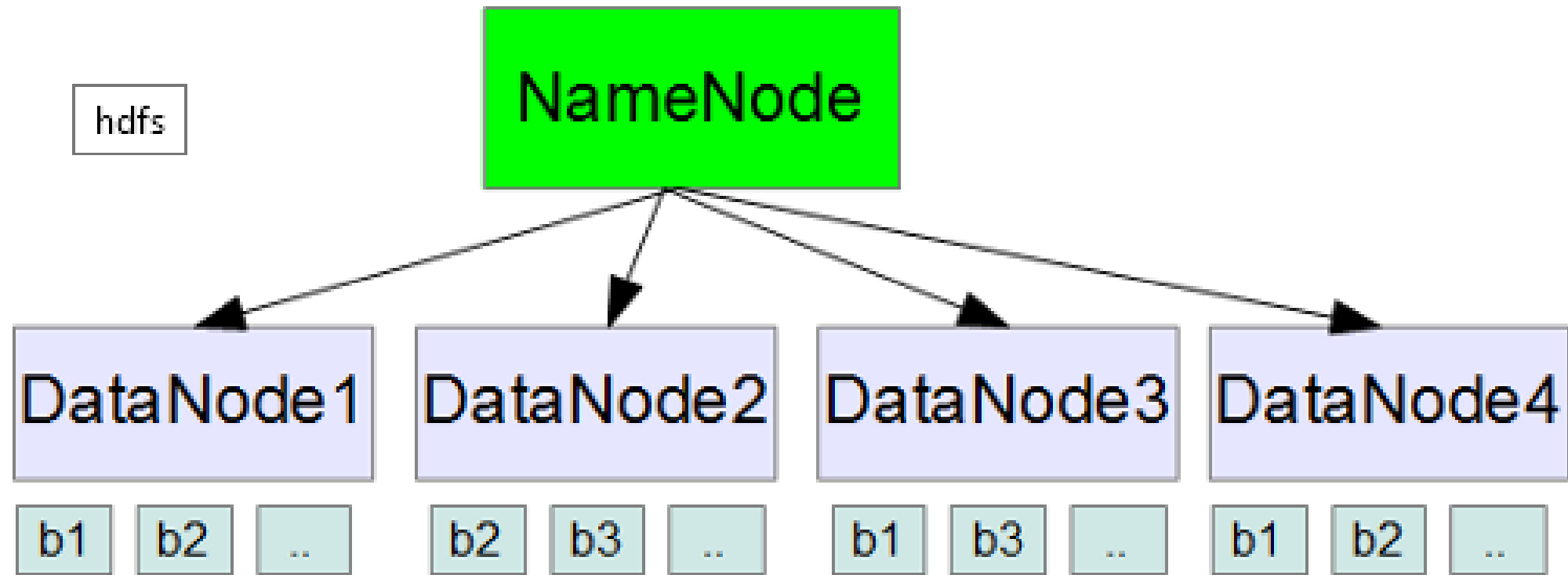
GFS: Assumptions

- 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)

Hadoop Architecture / HDFS



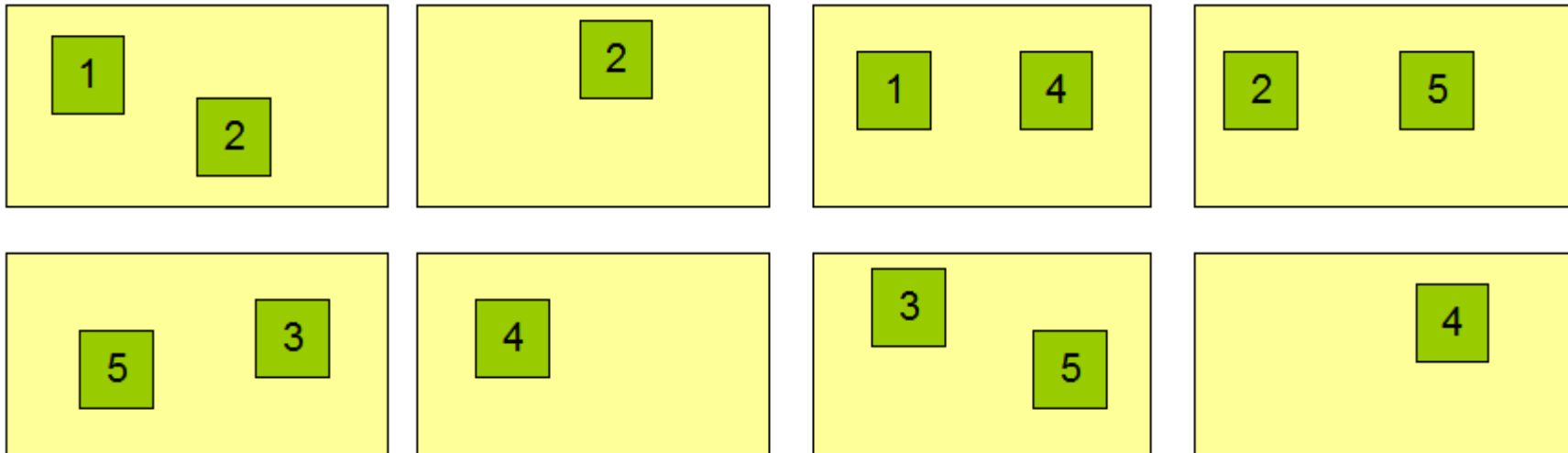
* Google File System, 2003 <http://research.google.com/archive/gfs.html>

Hadoop - HDFS

Block Replication

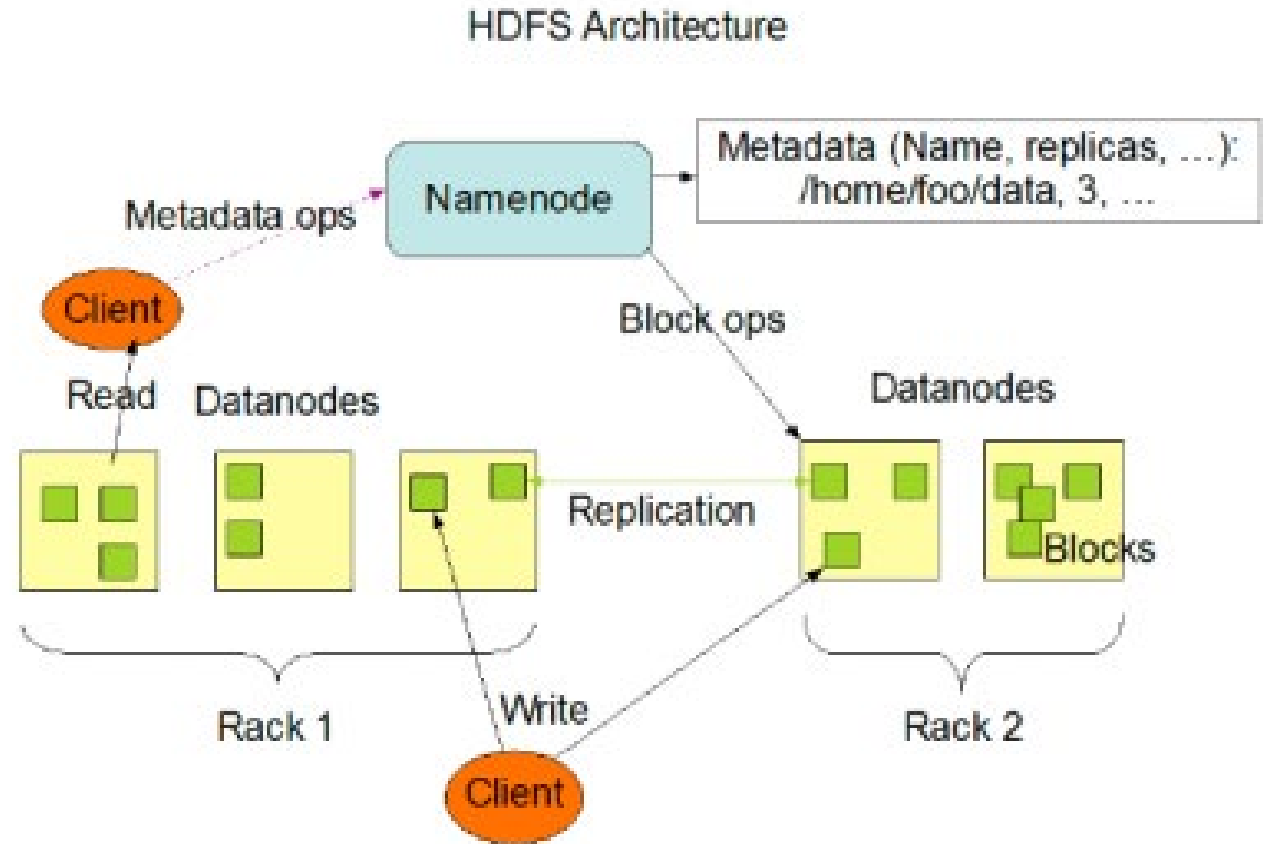
Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...

Datanodes



HDFS Terminology

- Namenode
- Datanode
- DFS Client
- Files/Directories
- Replication
- Blocks
- Rack-awareness



Exploring HDFS command line

- Attempt to re-create new dir :
 - `$ hadoop dfs -mkdir /user/foo`
- Create a destination directory using implicit path:
 - `$ hadoop dfs -mkdir bar`
- Auto-create nested destination directories:
 - `$ hadoop dfs -mkdir dir1/dir2/dir3`
- Remove dir:
 - `$ hadoop dfs -rmr /user/foo`
- Remove dir:
 - `$ hadoop dfs -rmr bar dir1`

HDFS Example: Import access log data

- Load access log into hdfs:
 - `$ hadoop dfs -put /var/log/apache2/access.log input/access.log`
- Verify it's in there:
 - `$ hadoop dfs -ls input/access.log`
- View the contents:
 - `$ hadoop dfs -cat input/access.log`

Browse HDFS using web UI

- Open <http://<hadoopIP>:50070>

MapReduce

Hadoop Programming

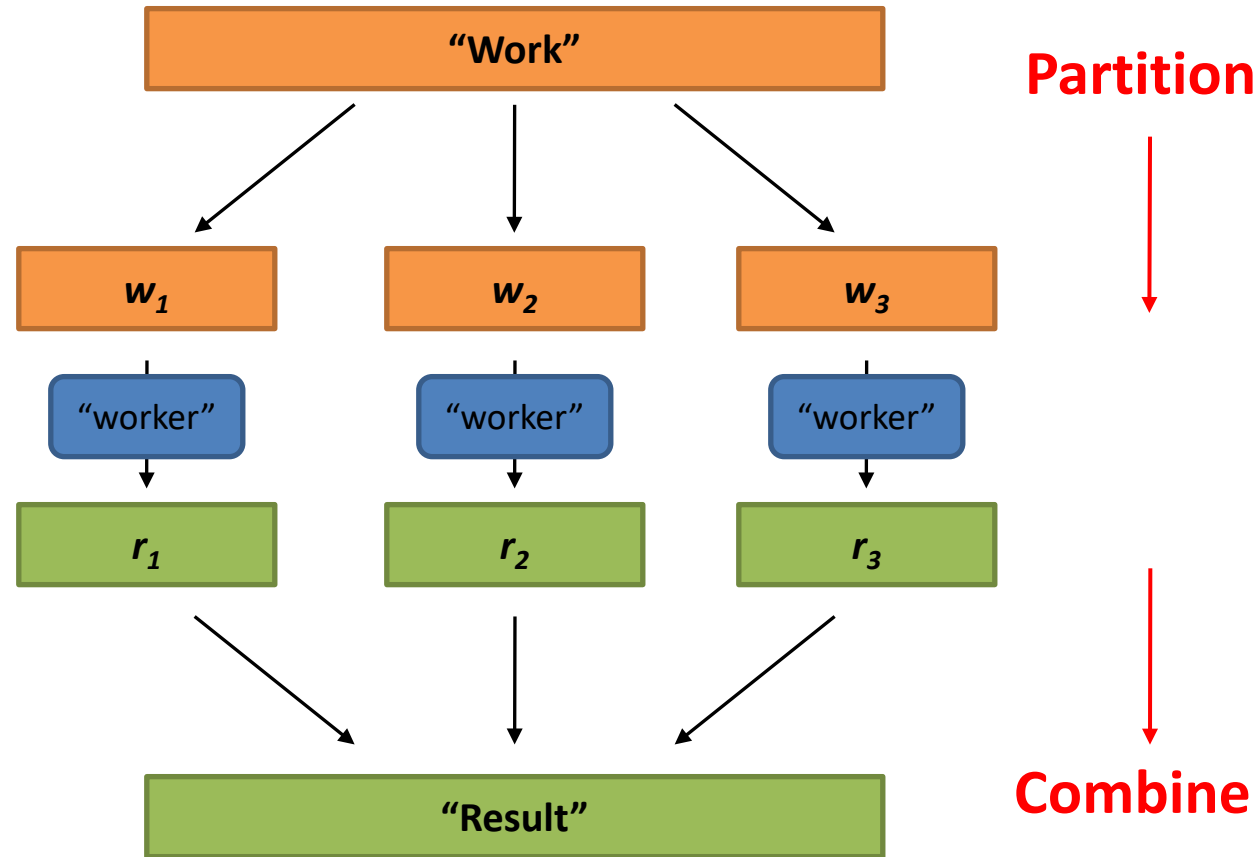
- “strong Java programming” as pre-requisite?
- Hadoop Streaming: ability to use an arbitrary language to define a job’s map and reduce processes
- this class is *not* about programming!
 - Focus on “thinking at scale” and algorithm design
 - We expect you to pick up Hadoop (quickly)
- How do I learn Hadoop?
 - This session: brief overview
 - White’s book
 - **RTFM, RTFC(!)**

Next...

- Map Reduce
 - Functional Programming
 - Map Reduce / Components
 - I/O
- Creating/Running MR Programs
 - Java
 - Streaming / Command Line
- Patterns
 - Word Count
 - Filtering
 - Joins
 - Top K
 - Binning
 - Bloom Filters
 - Page Rank

Hadoop/MapReduce

Divide and Conquer



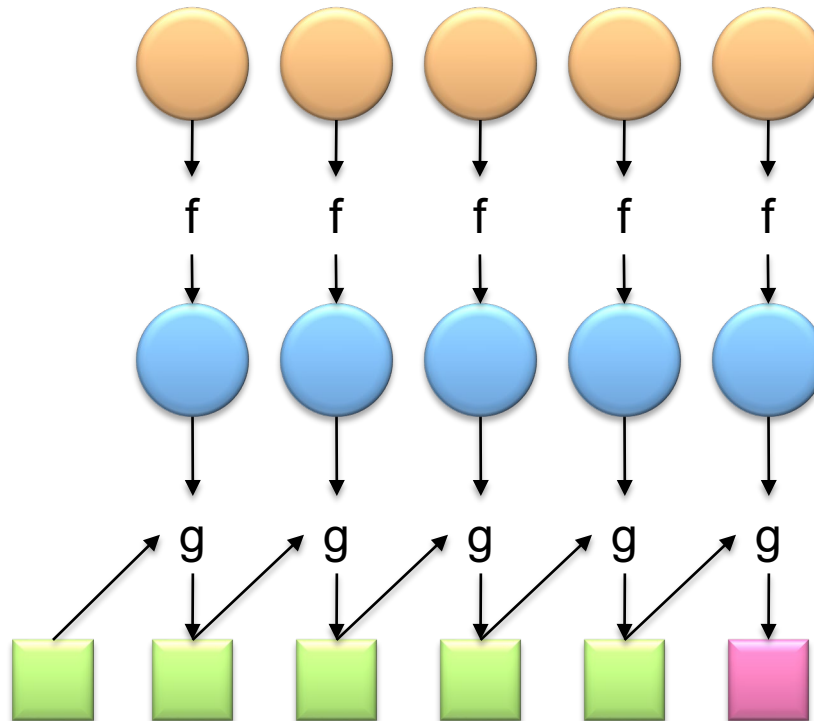
MapReduce

- 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 FS (HDFS)

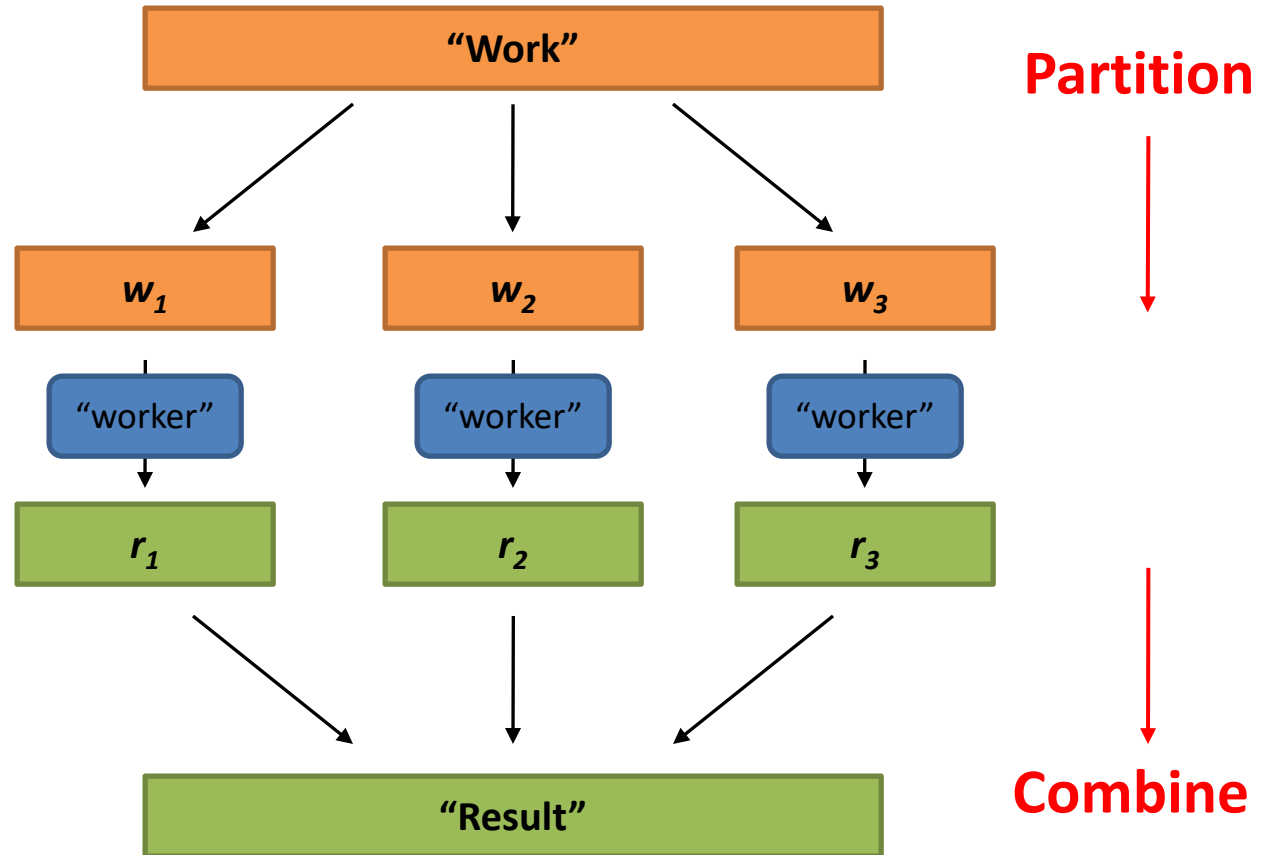
Roots in Functional Programming

Map

Fold



Map Reduce Framework



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

Typical Large-Data Problem

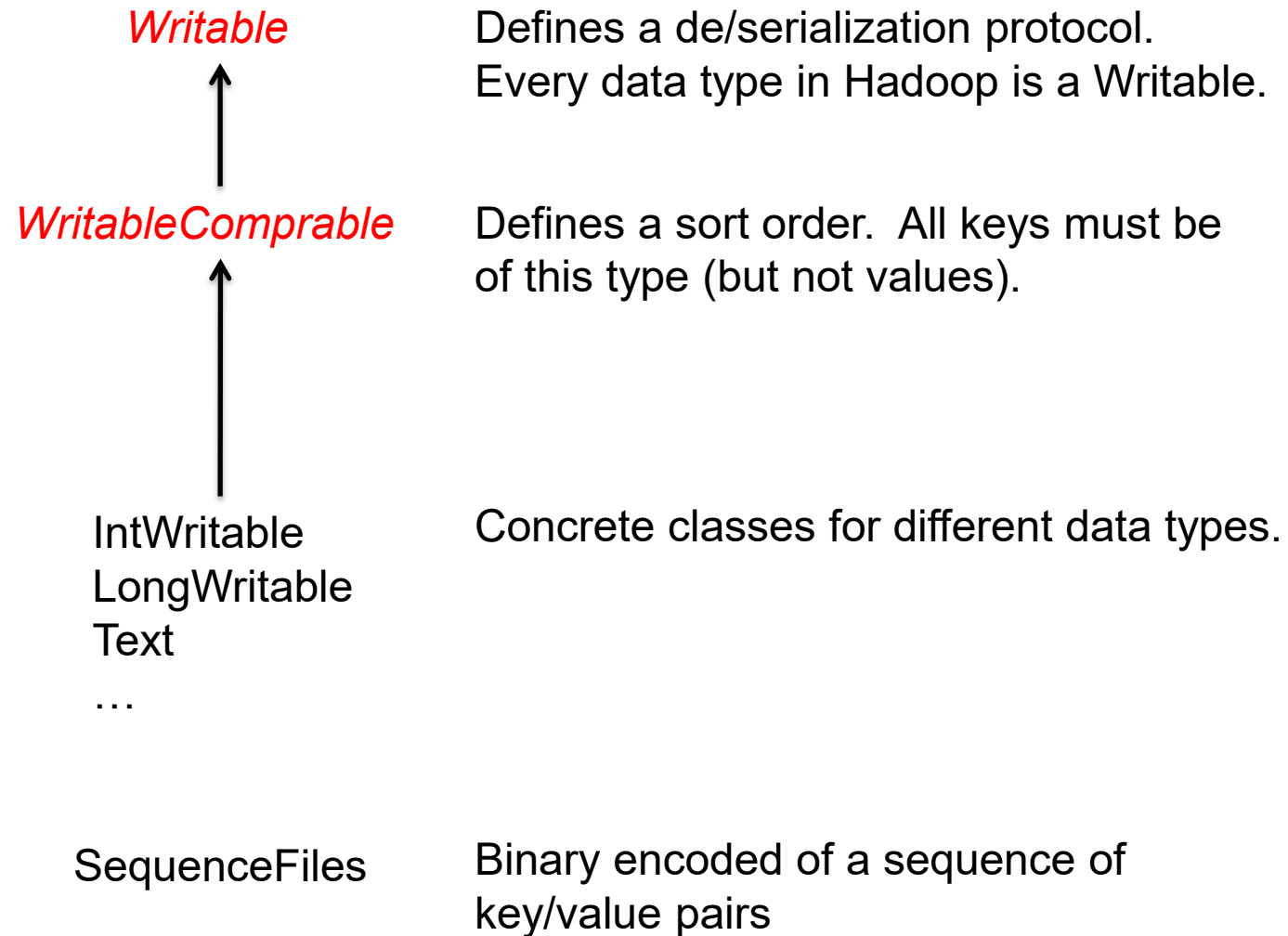
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

Reduce

Key idea: provide a functional abstraction for these two operations

Data Types in Hadoop



Big Data

Complex Data Types in Hadoop

- How do you implement complex data types?
- The easiest way:
 - Encoded it as Text, e.g., (a, b) = “a:b”
 - Use regular expressions to parse and extract data
 - Works, but pretty hack-ish
- The hard way:
 - Define a custom implementation of WritableComparable
 - Must implement: readFields, write, compareTo
 - Computationally efficient, but slow for rapid prototyping
- Alternatives:
 - Cloud⁹ offers two other choices: Tuple and JSON
 - (Actually, not that useful in practice)

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...

MapReduce

- Programmers specify two functions:

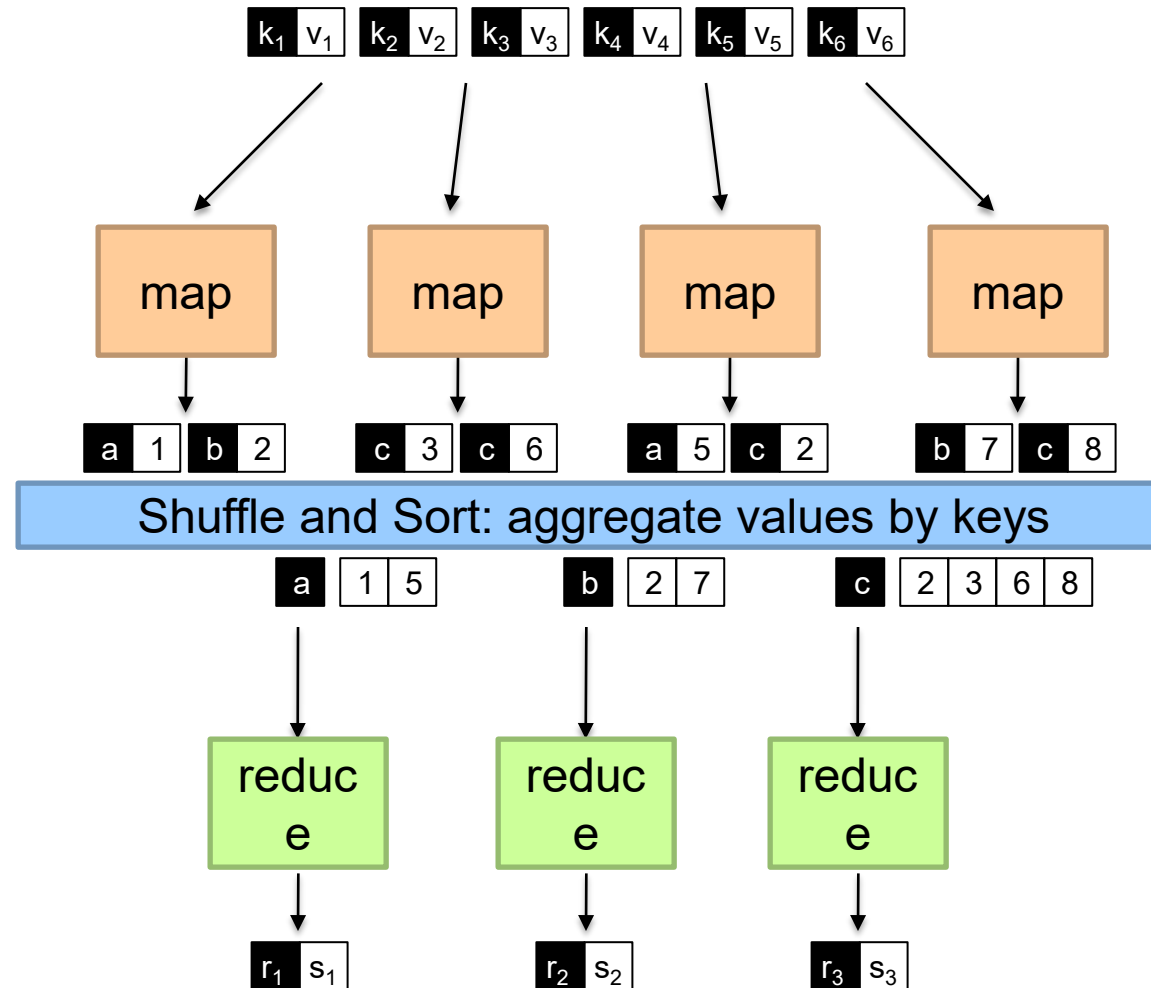
map $(k, v) \rightarrow \langle k', v' \rangle^*$

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

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 reduced together

- The execution framework handles everything else...

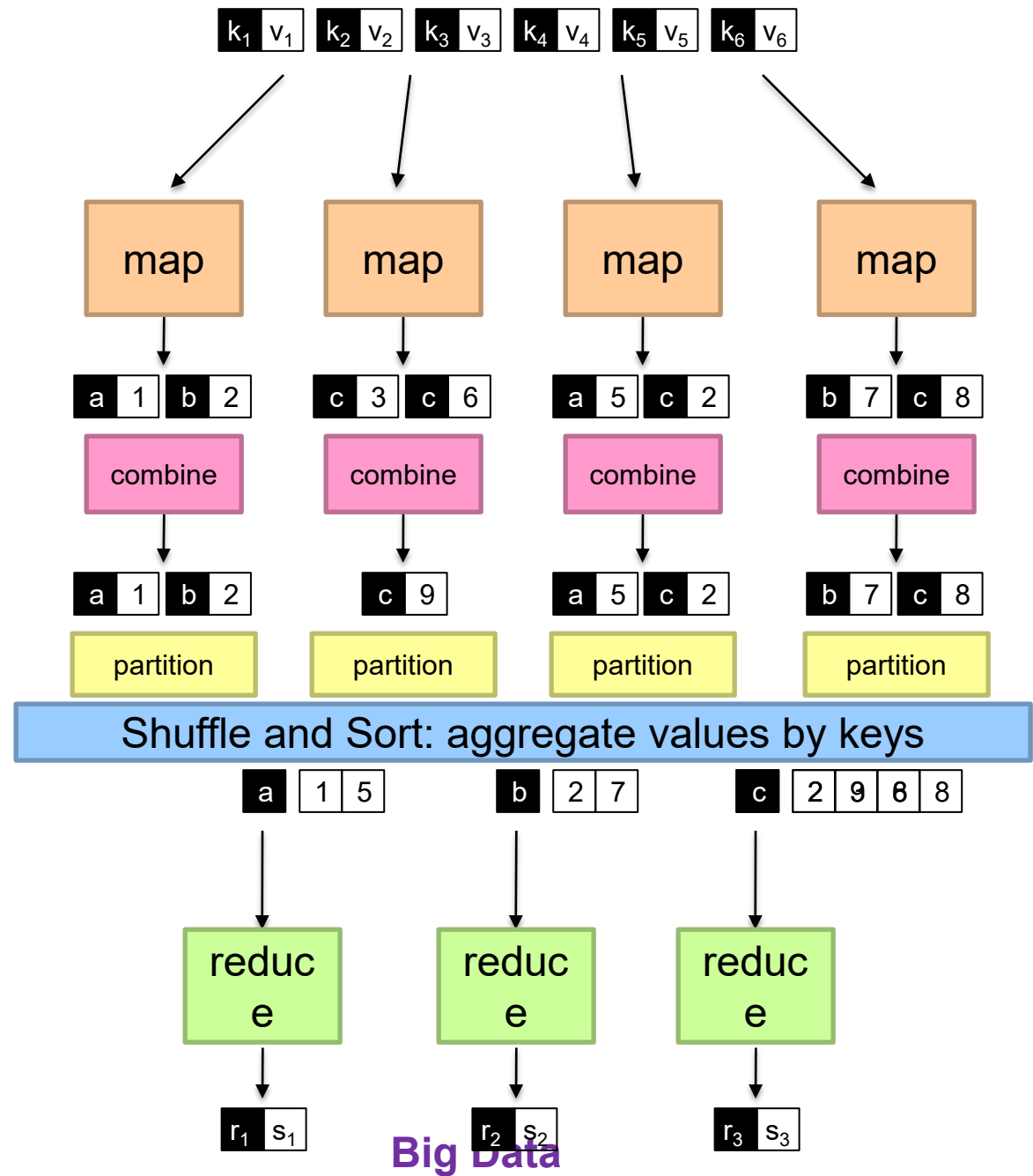
- Not quite...usually, programmers also specify:

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 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.

“Hello World”: Word Count

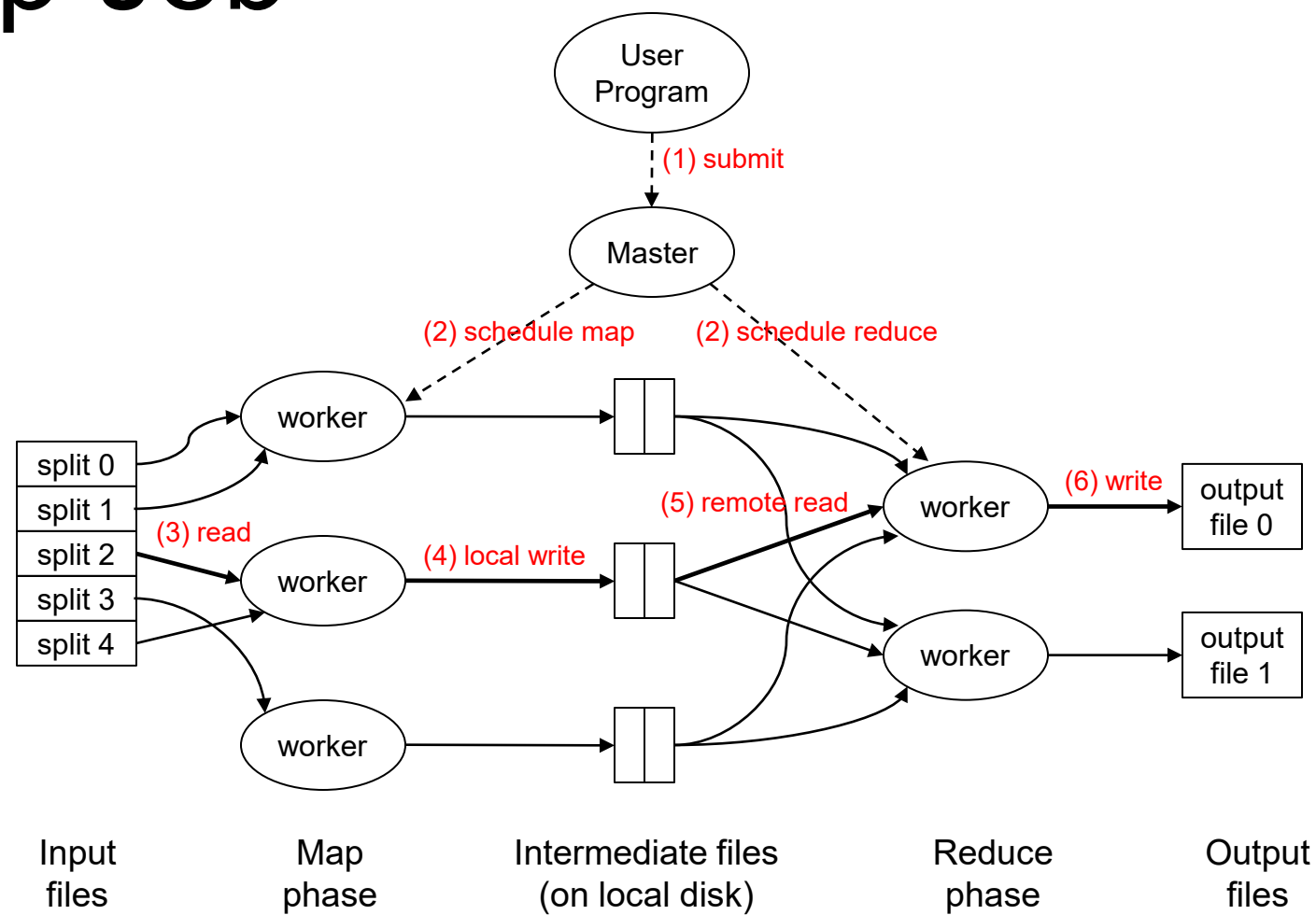
```
Map(String docid, String text):  
  for each word w in text:  
    Emit(w, 1);
```

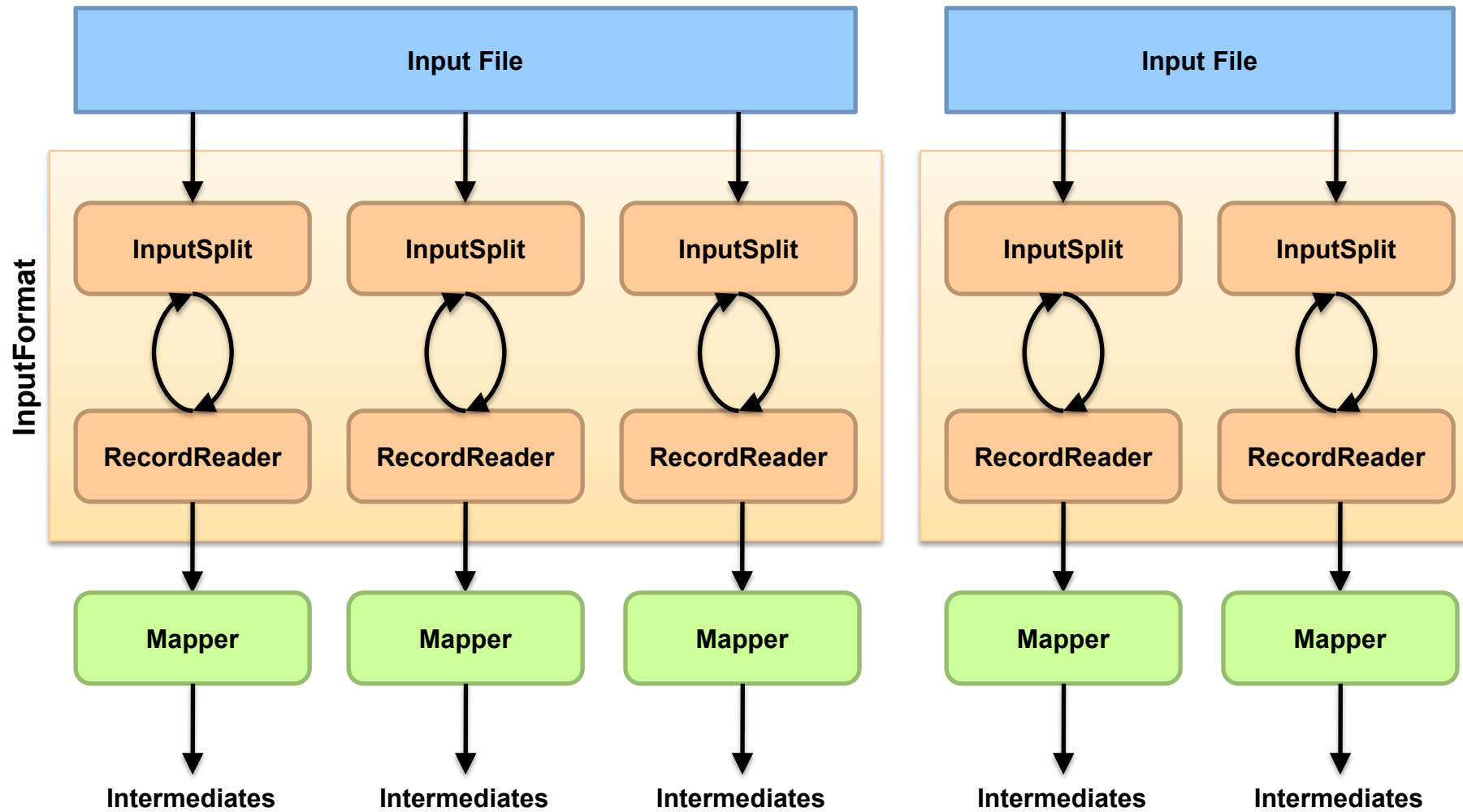
```
Reduce(String term, Iterator<Int> values):  
  int sum = 0;  
  for each v in values:  
    sum += v;  
  Emit(term, value);
```

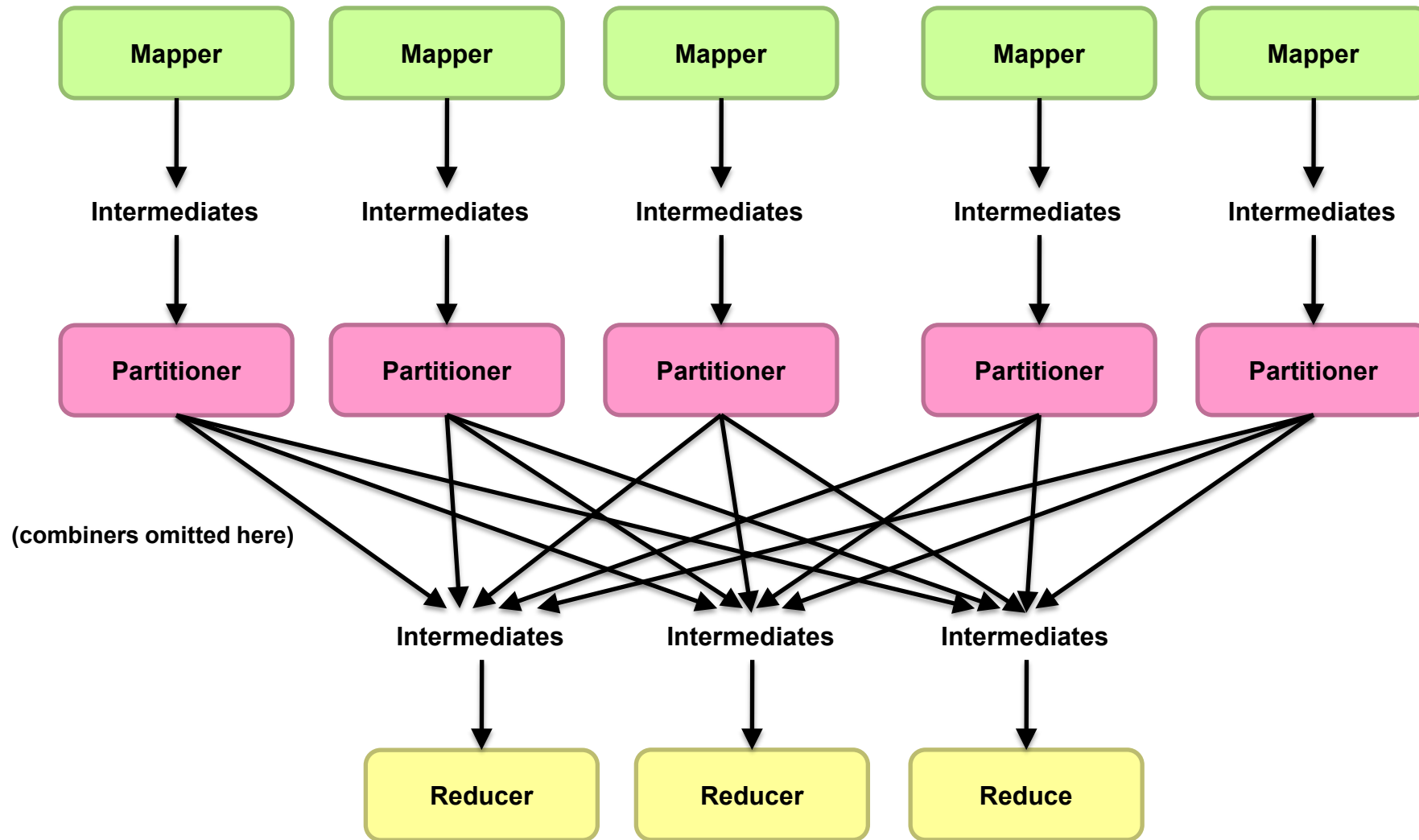

Anatomy of a Job

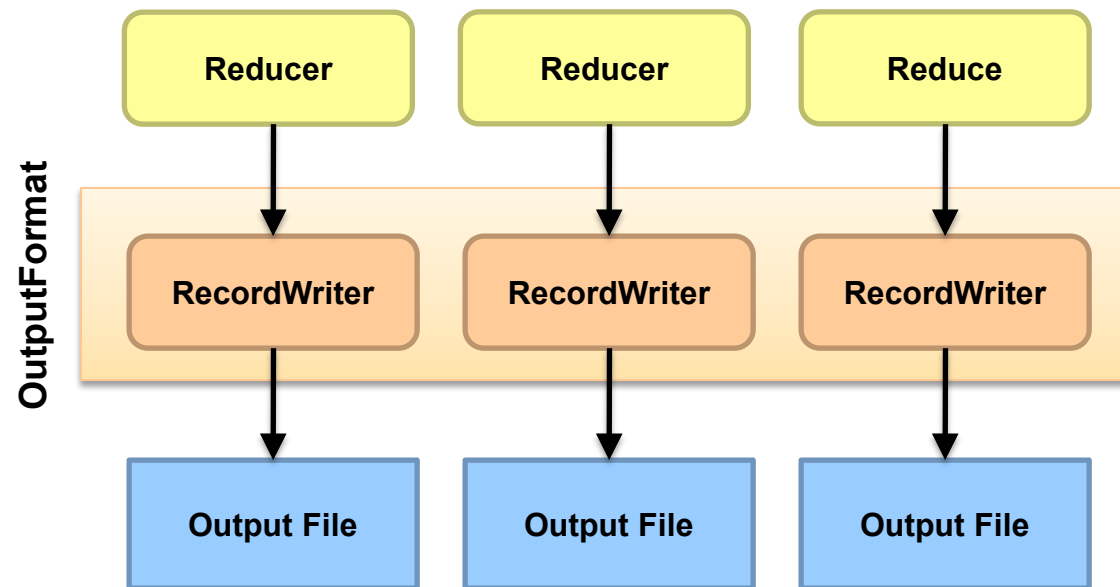
- MapReduce program in Hadoop = Hadoop job
 - Jobs are divided into map and reduce tasks
 - An instance of running a task is called a task attempt
 - Multiple jobs can be composed into a workflow
- Job submission process
 - Client (i.e., driver program) creates a job, configures it, and submits it to job tracker
 - JobClient computes input splits (on client end)
 - Job data (jar, configuration XML) are sent to JobTracker
 - JobTracker puts job data in shared location, enqueues tasks
 - TaskTrackers poll for tasks
 - Off to the races...

Hadoop Job









Basic Hadoop API*

○ Mapper

- void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter)
- void configure(JobConf job)
- void close() throws IOException

○ Reducer/Combiner

- void reduce(K2 key, Iterator<V2> values, OutputCollector<K3,V3> output, Reporter reporter)
- void configure(JobConf job)
- void close() throws IOException

○ Partitioner

- void getPartition(K2 key, V2 value, int numPartitions)

“Hello World”: 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, value);
```

Three Gotchas

- Avoid object creation, at all costs
- Execution framework reuses value in reducer
- Passing parameters into mappers and reducers
 - DistributedCache for larger (static) data

Input and Output

- InputFormat:
 - TextInputFormat
 - KeyValueTextInputFormat
 - SequenceFileInputFormat
 - ...
- OutputFormat:
 - TextOutputFormat
 - SequenceFileOutputFormat
 - ...

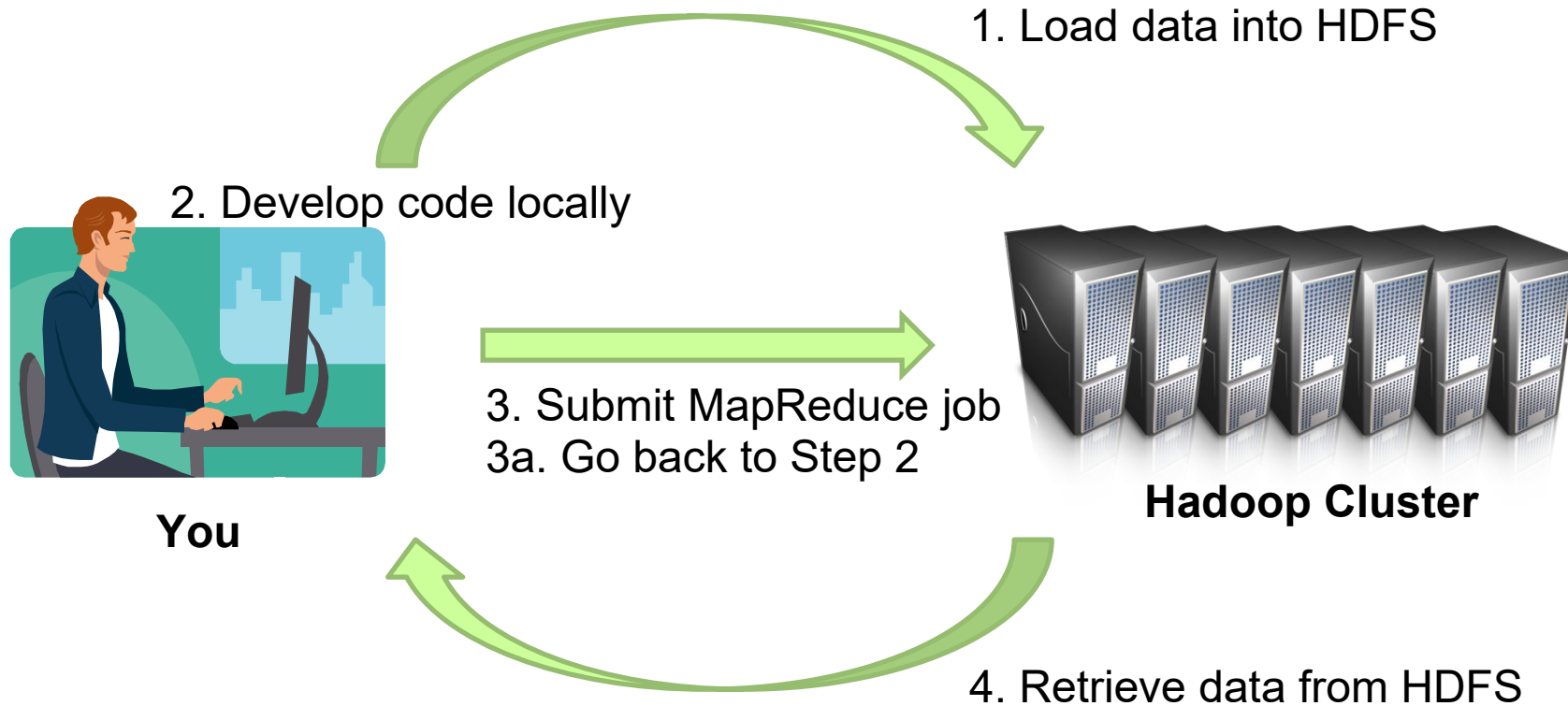
Recap

- Why large data?
- Cloud computing and MapReduce
- Large-data processing: “big ideas”
- What is MapReduce?
- Importance of the underlying distributed file system

Shuffle and Sort in Hadoop

- Probably the most complex aspect of MapReduce!
- Map side
 - Map outputs are buffered in memory in a circular buffer
 - When buffer reaches threshold, contents are “spilled” to disk
 - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here
- Reduce side
 - First, map outputs are copied over to reducer machine
 - “Sort” is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs here
 - Final merge pass goes directly into reducer

Hadoop Workflow



Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Strategies
 - Learn to use the webapp
 - Where does println go?
 - Don't use println, use logging
 - Throw RuntimeExceptions

Recap

- Hadoop data types
- Anatomy of a Hadoop job
- Hadoop jobs, end to end
- Software development workflow

MapReduce: Recap

- Programmers must specify:

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

- Optionally, also:

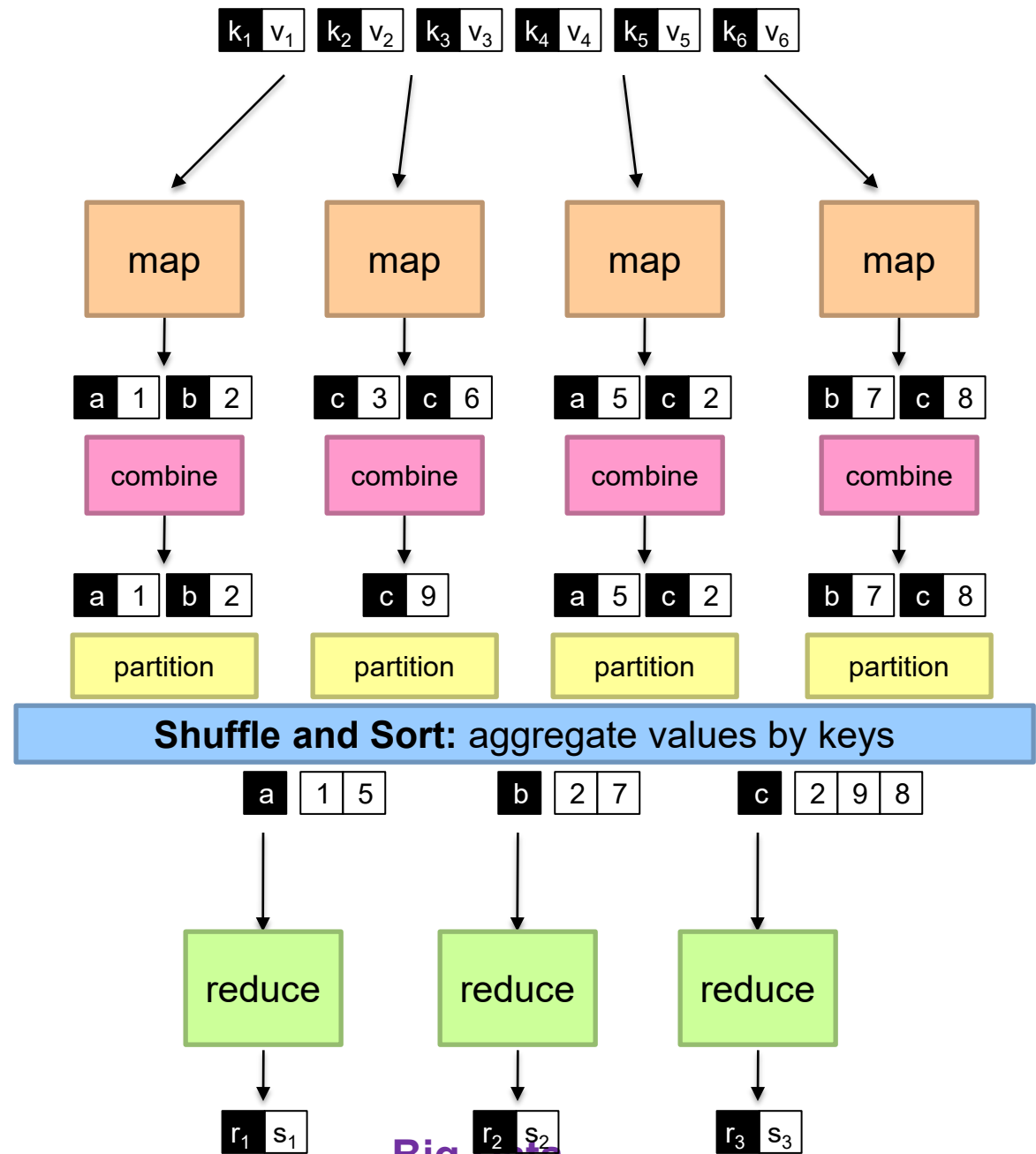
partition $(k', \text{number of partitions}) \rightarrow \text{partition for } k'$

- 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

- The execution framework handles everything else...



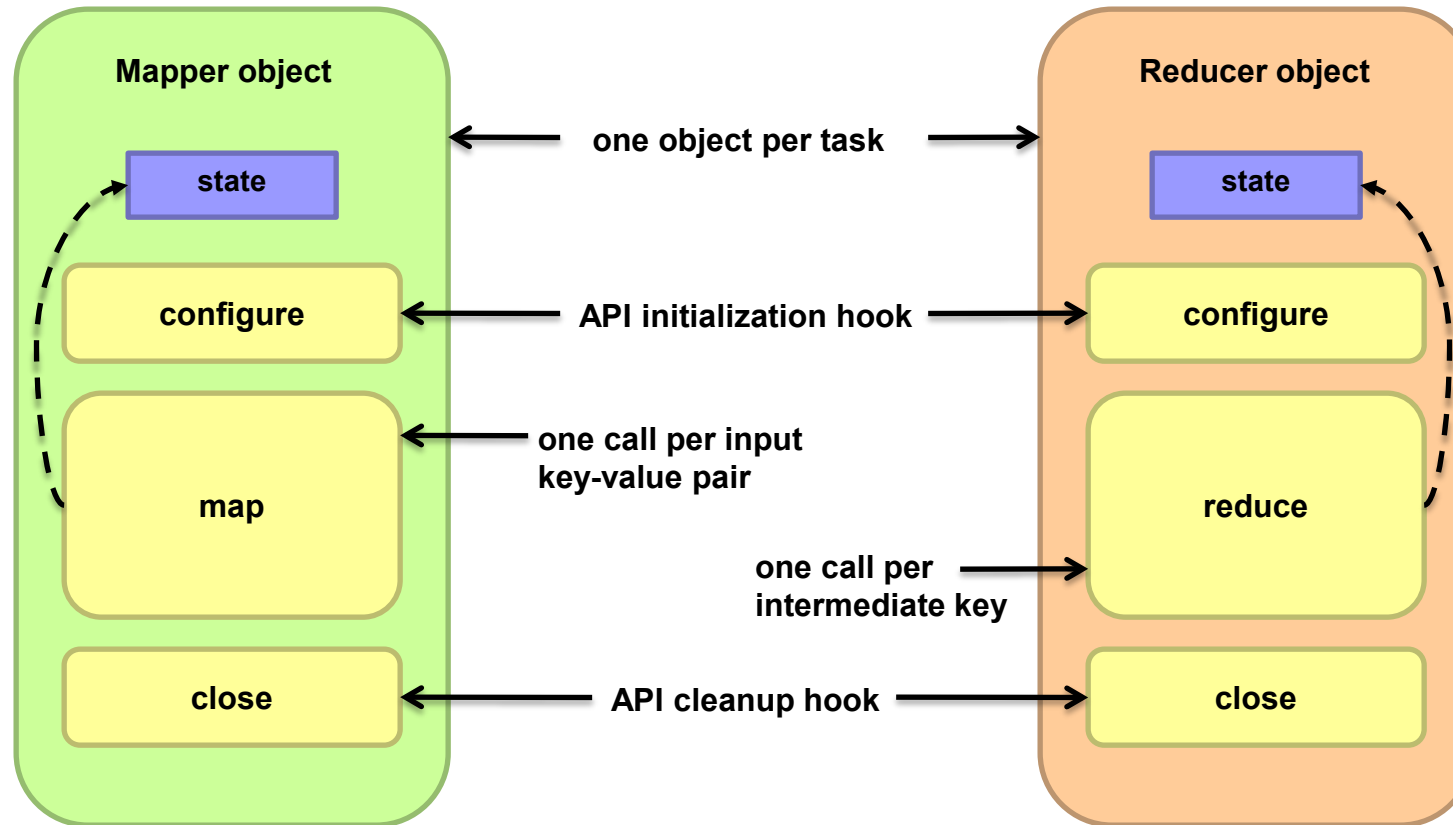
“Everything Else”

- The execution framework handles everything else...
 - Scheduling: assigns workers to map and reduce tasks
 - “Data distribution”: moves processes to data
 - Synchronization: gathers, sorts, and shuffles intermediate data
 - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
 - All algorithms must be expressed in m, r, c, p
- You don’t know:
 - Where mappers and reducers run
 - When a mapper or reducer begins or finishes
 - Which input a particular mapper is processing
 - Which intermediate key a particular reducer is processing

Tools for Synchronization

- Cleverly-constructed data structures
 - Bring partial results together
- Sort order of intermediate keys
 - Control order in which reducers process keys
- Partitioner
 - Control which reducer processes which keys
- Preserving state in mappers and reducers
 - Capture dependencies across multiple keys and values

Special Topics - Preserving State



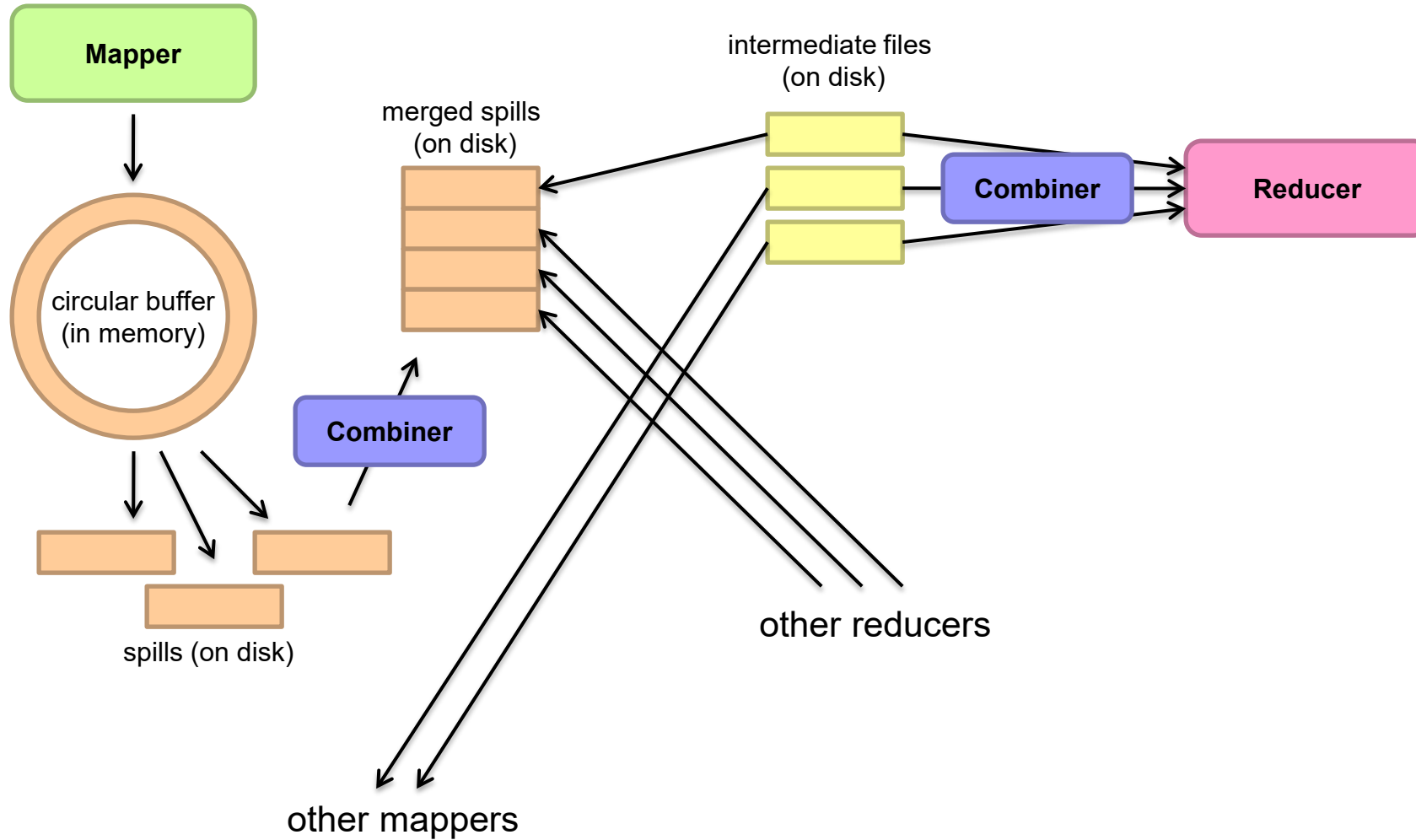
Scalable Hadoop Algorithms: Themes

- Avoid object creation
 - Inherently costly operation
 - Garbage collection
- Avoid buffering
 - Limited heap size
 - Works for small datasets, but won't scale!

Importance of Local Aggregation

- Ideal scaling characteristics:
 - Twice the data, twice the running time
 - Twice the resources, half the running time
- Why can't we achieve this?
 - Synchronization requires communication
 - Communication kills performance
- Thus... avoid communication!
 - Reduce intermediate data via local aggregation
 - Combiners can help

Shuffle and Sort



Design Pattern for Local Aggregation

- “In-mapper combining”
 - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
 - Speed
 - Why is this faster than actual combiners?
- Disadvantages
 - Explicit memory management required
 - Potential for order-dependent bugs

Combiner Design

- Combiners and reducers share same method signature
 - Sometimes, reducers can serve as combiners
 - Often, not...
- Remember: combiner are optional optimizations
 - Should not affect algorithm correctness
 - May be run 0, 1, or multiple times
- Example: find average of all integers associated with the same key

Recap: Tools for Synchronization

- Cleverly-constructed data structures
 - Bring data together
- Sort order of intermediate keys
 - Control order in which reducers process keys
- Partitioner
 - Control which reducer processes which keys
- Preserving state in mappers and reducers
 - Capture dependencies across multiple keys and values

Issues and Tradeoffs

- Number of key-value pairs
 - Object creation overhead
 - Time for sorting and shuffling pairs across the network
- Size of each key-value pair
 - De/serialization overhead
- Local aggregation
 - Opportunities to perform local aggregation varies
 - Combiners make a big difference
 - Combiners vs. in-mapper combining
 - RAM vs. disk vs. network

Debugging at Scale

- Works on small datasets, won't scale... why?
 - Memory management issues (buffering and object creation)
 - Too much intermediate data
 - Mangled input records
- Real-world data is messy!
 - Word count: how many unique words in Wikipedia?
 - There's no such thing as "consistent data"
 - Watch out for corner cases
 - Isolate unexpected behavior, bring local