Hadoop & Map-Reduce

Riccardo Torlone

Università Roma Tre

Credits: Jimmy Lin (University of Maryland)

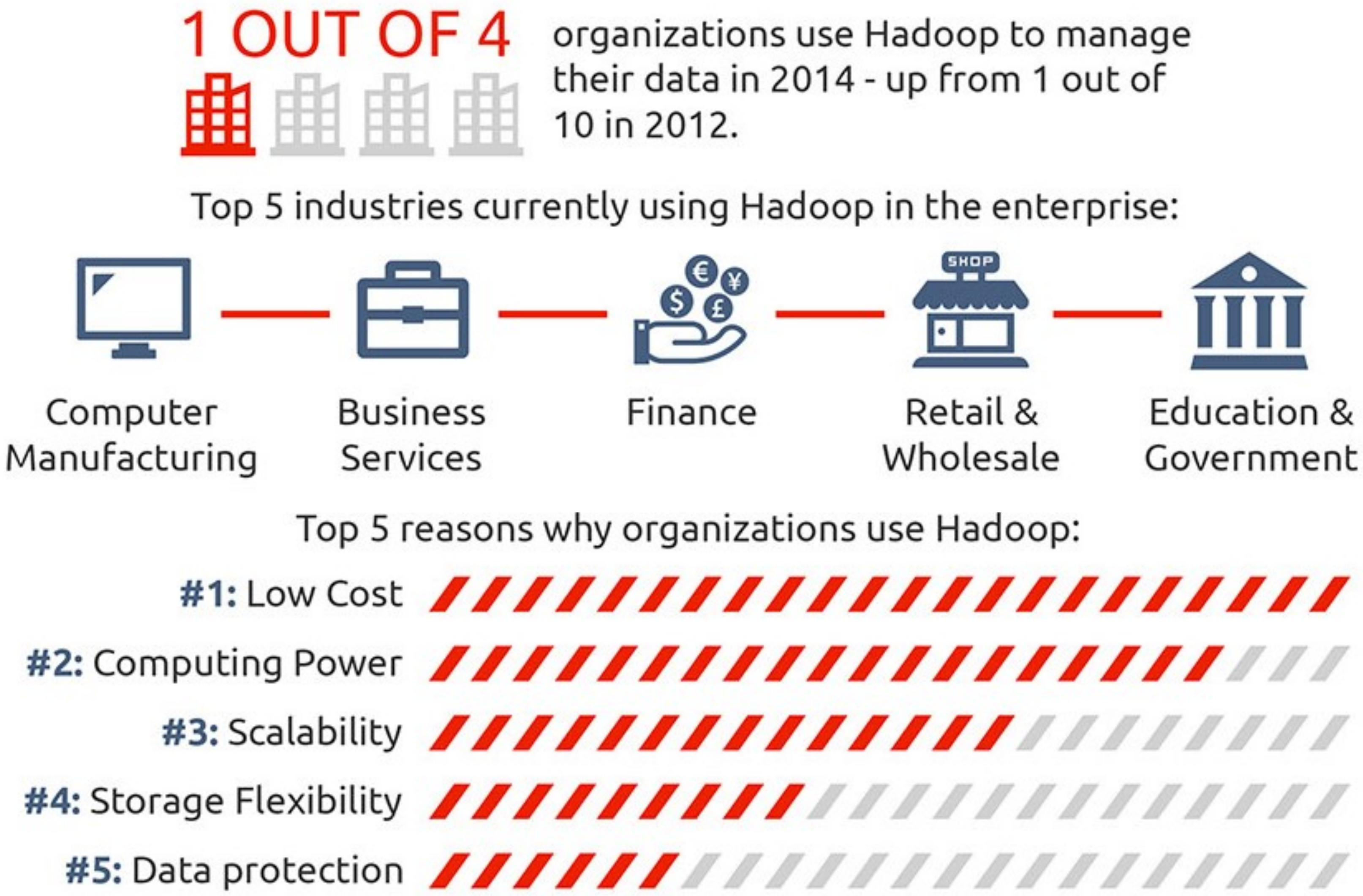
Hadoop in a nutshell

- An open-source software framework (Apache project)
- Originally developed by Yahoo (but originates from Google)
- Goal: storage and processing of data-sets at massive scale
- Infrastructure: clusters of commodity hardware
- Core (Hadoop 1.0):
  - HDFS, a distributed file system
  - MapReduce, a programming model for large scale data processing.
- Includes a number of related projects (Ecosystem)
  - Hive, Spark, Kafka, HBase, etc..
- Used in production by Google, Facebook, Yahoo! and many others.

Hadoop: some History

- 2003: Google publishes about its cluster architecture & distributed file system (GFS)
- 2004: Google publishes about its MapReduce programming model used on top of GFS
  - written in C++
  - closed-source, Python and Java APIs available to Google programmers only
- 2006: Apache & Yahoo! → Hadoop & HDFS (Doug Cutting and Mike Cafarella)
  - open-source, Java implementations of Google MapReduce and GFS
  - a diverse set of APIs available to public
- 2008: becomes an independent Apache project
  - Yahoo! uses Hadoop in production
- Today: used as a general-purpose storage and analysis platform for big data
  - other Hadoop distributions from several vendors including IBM, Microsoft, Oracle, Cloudera, Hortonworks, etc.
  - many users (http://wiki.apache.org/hadoop/PoweredBy)
  - research and development actively continues…

Who uses Hadoop and why?



1 OUT OF 4 organizations use Hadoop to manage their data in 2014 - up from 1 out of 10 in 2012.

Top 5 industries currently using Hadoop in the enterprise:

Computer Manufacturing — Business Services — Finance — Retail & Wholesale — Education & Government

Top 5 reasons why organizations use Hadoop:

#1: Low Cost

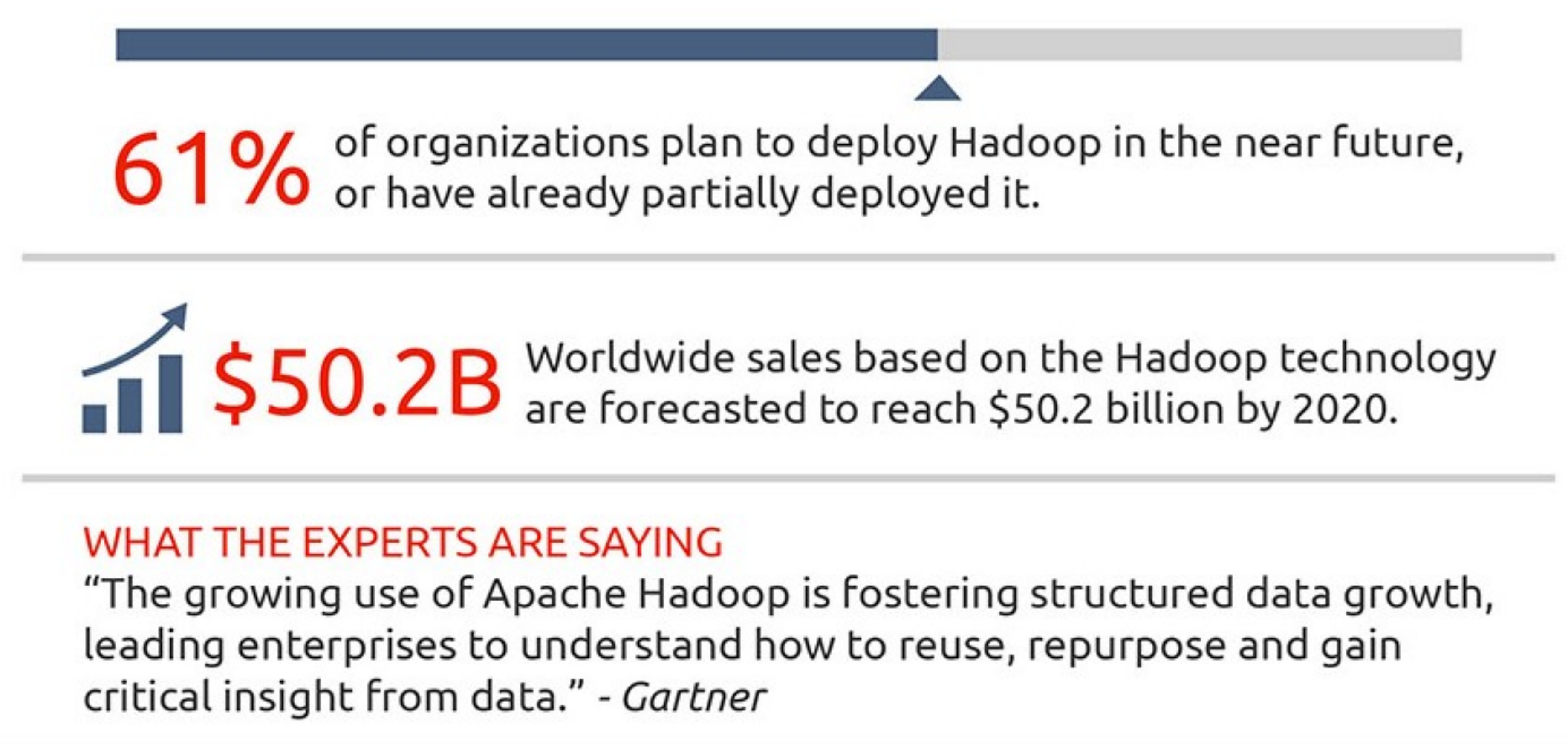
#2: Computing Power

#3: Scalability

#4: Storage Flexibility

#5: Data protection

The future of Hadoop



61% of organizations plan to deploy Hadoop in the near future, or have already partially deployed it.

$50.2B Worldwide sales based on the Hadoop technology are forecasted to reach $50.2 billion by 2020.

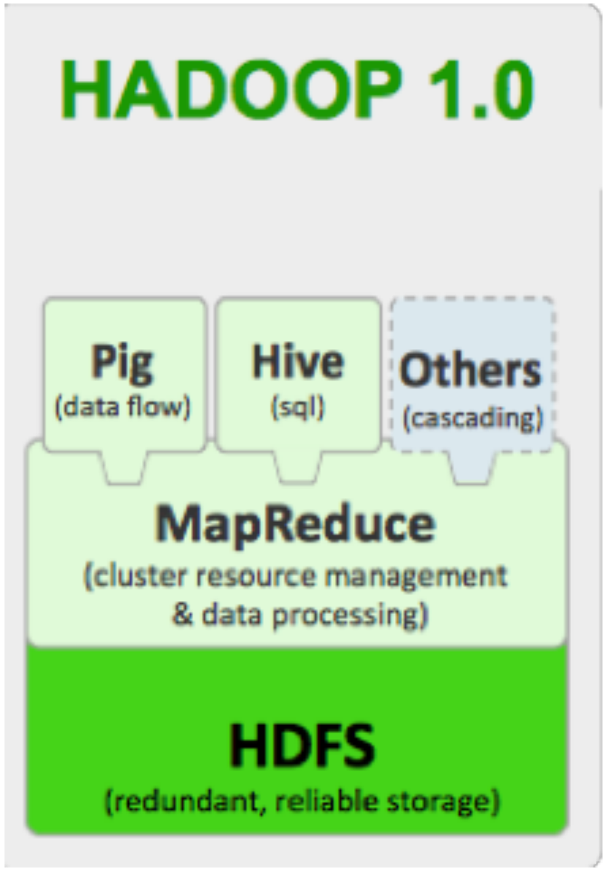
WHAT THE EXPERTS ARE SAYING

"The growing use of Apache Hadoop is fostering structured data growth, leading enterprises to understand how to reuse, repurpose and gain critical insight from data." - *Gartner*

The core of Hadoop 1.0

- HDFS
  - A distributed file systems
  - Servers can fail and not abort the computation process.
  - Data is replicated with redundancy across the cluster.
- MapReduce
  - Programming paradigm for expressing distributed computations over multiple servers
  - The powerhouse behind most of today's big data processing
  - Also used in other MPP environments and NoSQL databases (e.g., Vertica and MongoDB)
- Improving programmability: Pig, Hive, Tez
- Improving data access: HBase, Sqoop and Flume

The original Hadoop Ecosystem


HADOOP 1.0
Pig
(data flow)
Hive
(sql)
Others
(cascading)
MapReduce
(cluster resource management & data processing)
HDFS
(redundant, reliable storage)

What is MapReduce?

- Programming model for expressing distributed computations at a massive scale
- Execution framework for organizing and performing such computations
  - running the various tasks in parallel,
  - providing for redundancy and fault tolerance.
- Inspired by the map and reduce functions commonly used in functional programming
- Various implementations:
  - Google,
  - Hadoop,
  - ….
- Google has been granted a patent on MapReduce.

Good news

Ronald Coase

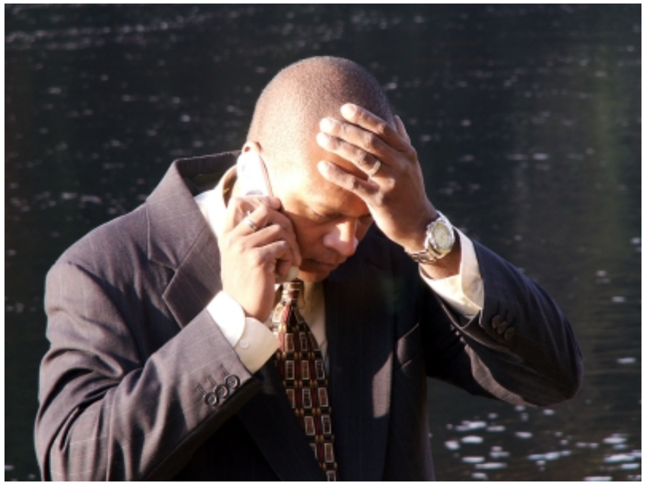
If you torture the data long enough, it will always confess

More data usually beats better algorithms

Anand Rajaman

Bad news

- The storage capacities have increased massively over the years but access speeds have not kept up:



year: 1990
size: ~1.3GB
speed: 4.4 MB/s

5 mins

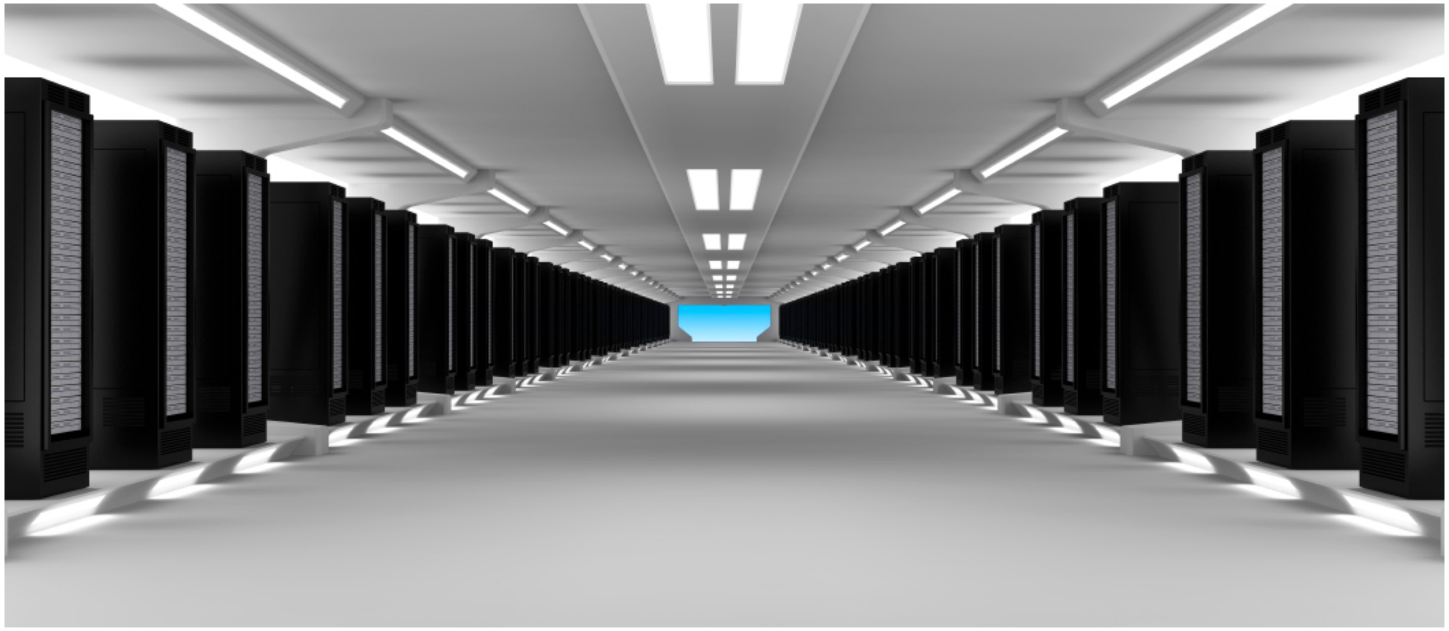
year: 2018
size: ~1TB
speed: 2 GB/s

8 minutes

year: 2019
size: ~1TB
speed: 300 MB/s

1 hours

Today solution: cluster computing



- 100 hard disks? < 1 min to read 1TB
- What about disk failures?
- Replication (RAID) … or

hadoop HDFS

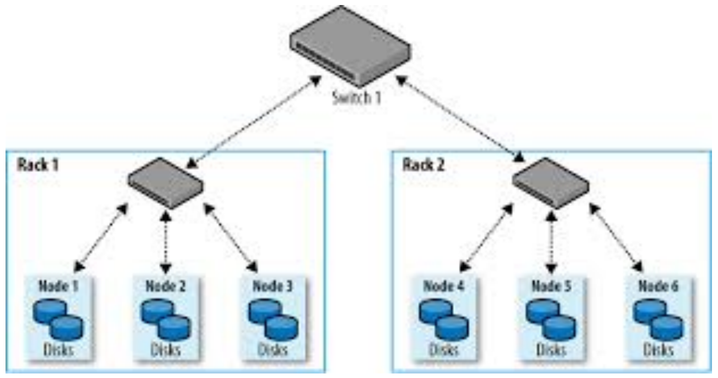
Scale up vs scale out

Scale up

Scale out

Cluster computing

- Compute nodes are stored on racks
  - 8–64 compute nodes on a rack
- There can be many racks of compute nodes
- The nodes on a single rack are connected by a network
  - typically gigabit Ethernet
- Racks are connected by another level of network or a switch
- The bandwidth of intra-rack communication is usually much greater than that of inter-rack communication
- Compute nodes can fail! Solution:
  - Files are stored redundantly
  - Computations are divided into tasks



Commodity hardware

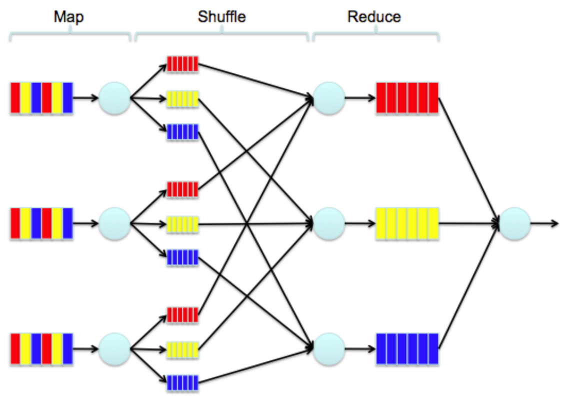
- You are not tied to expensive, proprietary offerings from a single vendor
- You can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster
- Commodity ≠ Low-end!
  - cheap components with high failure rate can be a false economy
  - but expensive database class machines do not score well on the price/performance curve

Commodity hardware

- Example typical specifications of commodity hardware:
  - Processor 2 quad-core 2-2.5GHz CPUs
  - Memory 16-24 GB ECC RAM
  - Storage 4 × 1TB SATA disks
  - Network Gigabit Ethernet
- Yahoo! has a huge installation of Hadoop:
  - > 100,000 CPUs in > 40,000 computers
  - Used to support research for Ad Systems and Web Search
  - Also used to do scaling tests to support development of Hadoop

The New Software Stack

- New programming environments designed to get their parallelism not from a supercomputer but from computing clusters
- Bottom of the stack: distributed file system (DFS)
- On the top of a DFS:
  - many different high-level programming systems
- DFS
  - We have a winner: Hadoop
- Programming system
  - The original one: MapReduce



DFS: Assumptions

- Cluster of commodity hardware
  - Scale “out”, not “up”
- Significant failure rates
  - Nodes can fail over 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

DFS: organization

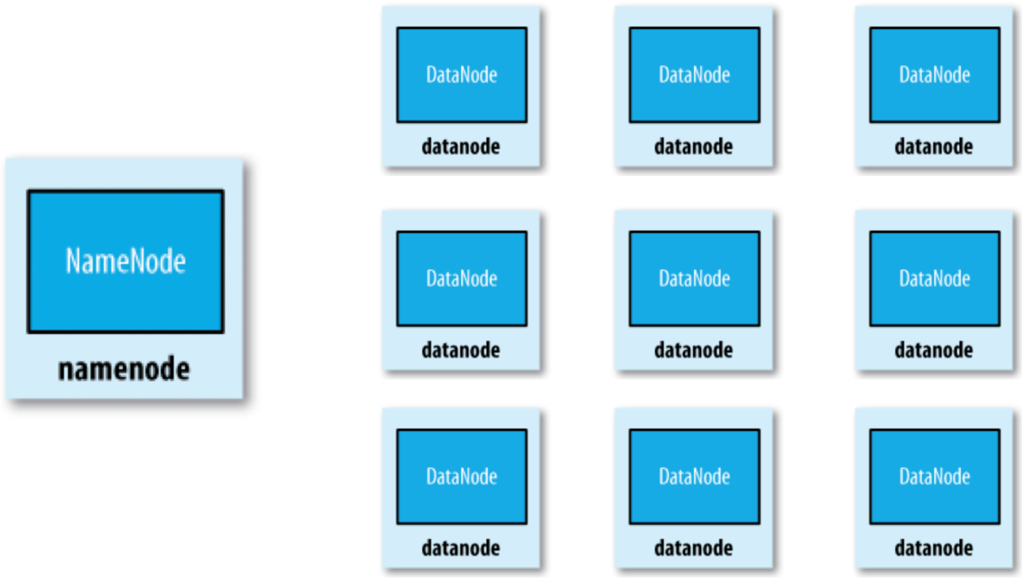
- Files are divided into chunks (or shard)
  - typically 64 megabytes in size.
- Chunks are replicated at different compute nodes (usually 3+)
- Nodes holding copies of one chunk are located on different racks
- Chunk size and the degree of replication can be decided by the user
- A special file (the master node) stores, for each file, the positions of its chuncks
- The master node is itself replicated
- A directory for the file system knows where to find the master node.
- The directory itself can be replicated.
- All participants using the DFS know where the directory copies are.

DFS implementations

- Several distributed file systems used in practice.
- Among these:
  - The Google File System (GFS), the original of the class.
  - CloudStore, an open-source DFS originally developed by Kosmix.
  - S3, a storage service offered by Amazon Web Services (AWS)
  - GPFS (IBM), Isilon (EMC), MapR File System, Ceph, …
  - Hadoop Distributed File System (HDFS), an open-source DFS used with Hadoop: the winner!
    - Master node = Namenode
    - Compute node = Datanode
    - Node: both physical and logical entity

HDFS concepts

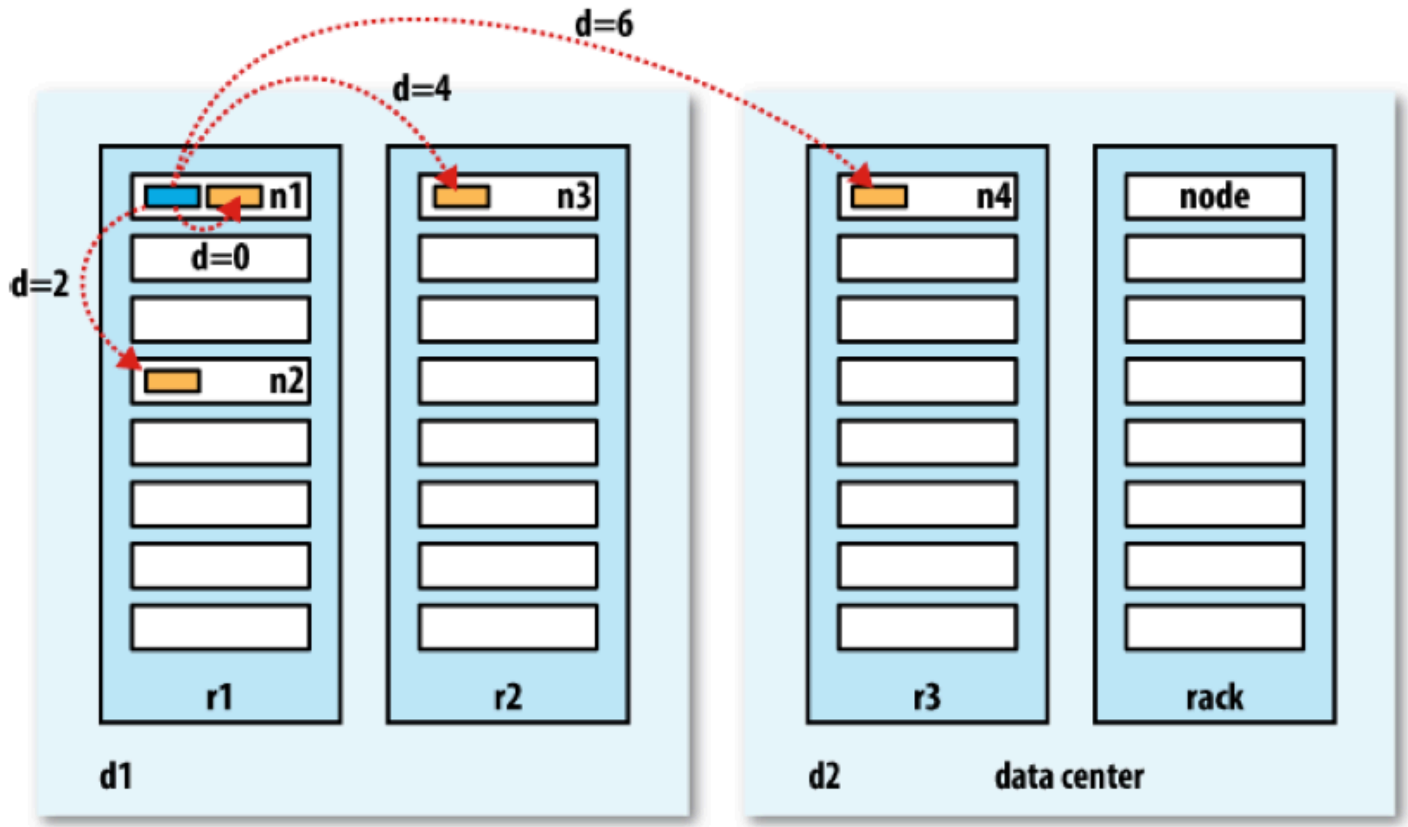
- HDFS has a master/slave architecture
- An HDFS cluster has two types of nodes:
  - The NameNode: a master server that manages the file system namespace and regulates access to files by clients.
  - Multiple DataNodes: usually one per node in the cluster, which manage storage attached to the nodes that they run on



HDFS concepts

- The namenode:
  - Manages the filesystem tree and the metadata for all the files and directories
  - Knows the datanodes on which all the blocks for a given file are located
- The datanodes:
  - Just store and retrieve the blocks when they are told to (by clients or the namenode). Internally, a file is split into one or more blocks and these blocks are stored in a set of datanodes.
- Without the namenode HDFS cannot be used
  - The NameNode machine is a single point of failure for an HDFS cluster
- It is important to make the namenode resilient to failure

Physical organization



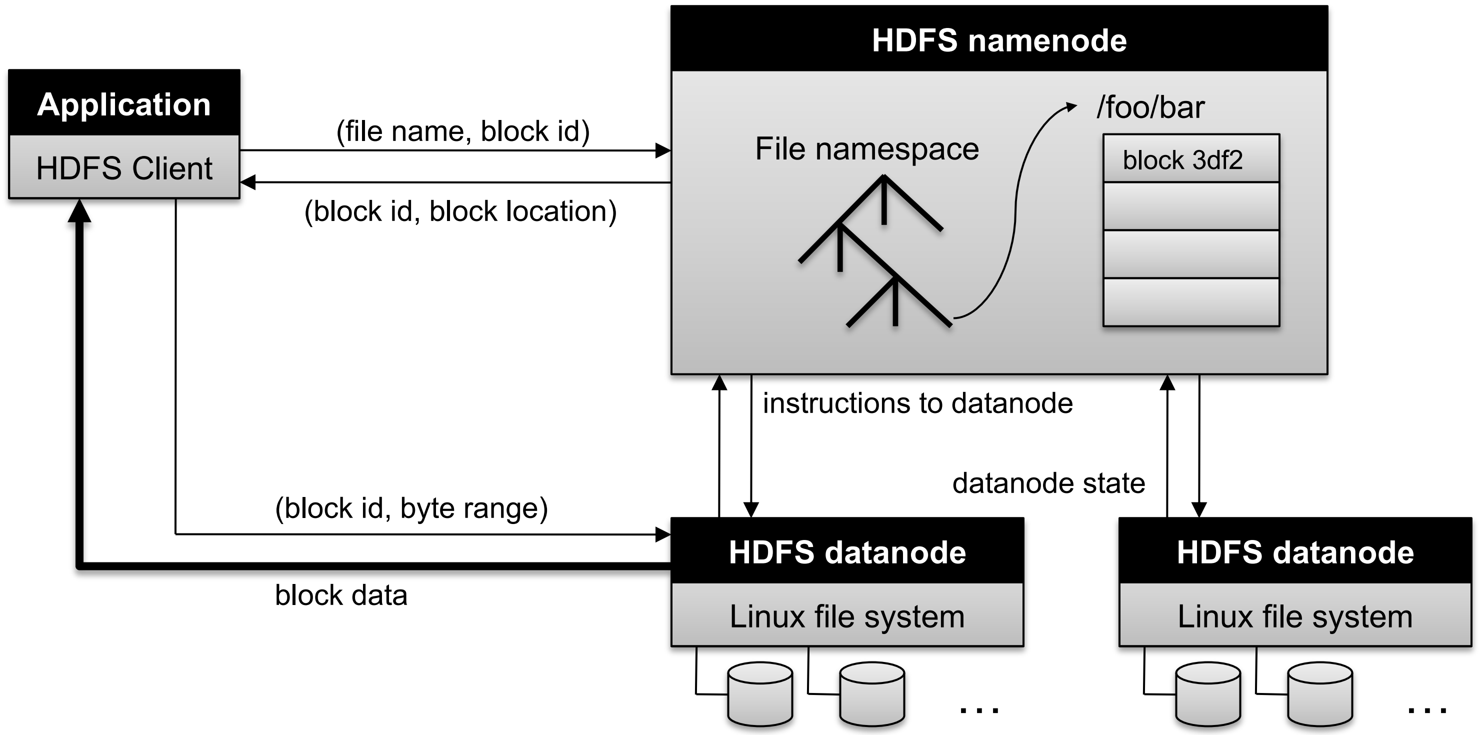
HDFS I/O

- An application client wishing to read a file (or a portion thereof) must first contact the namenode to determine where the actual data is stored
- In response to the client request the namenode returns:
  - the relevant block ids
  - the location where the blocks are held (i.e., which datanodes)
- The client then contacts the datanodes to retrieve the data
- Blocks are themselves stored on standard single-machine file systems
  - HDFS lies on top of the standard OS stack
- Important feature of the design:
  - data is never moved through the namenode
  - all data transfer occurs directly between clients and datanodes
  - communications with the namenode only involves transfer of metadata

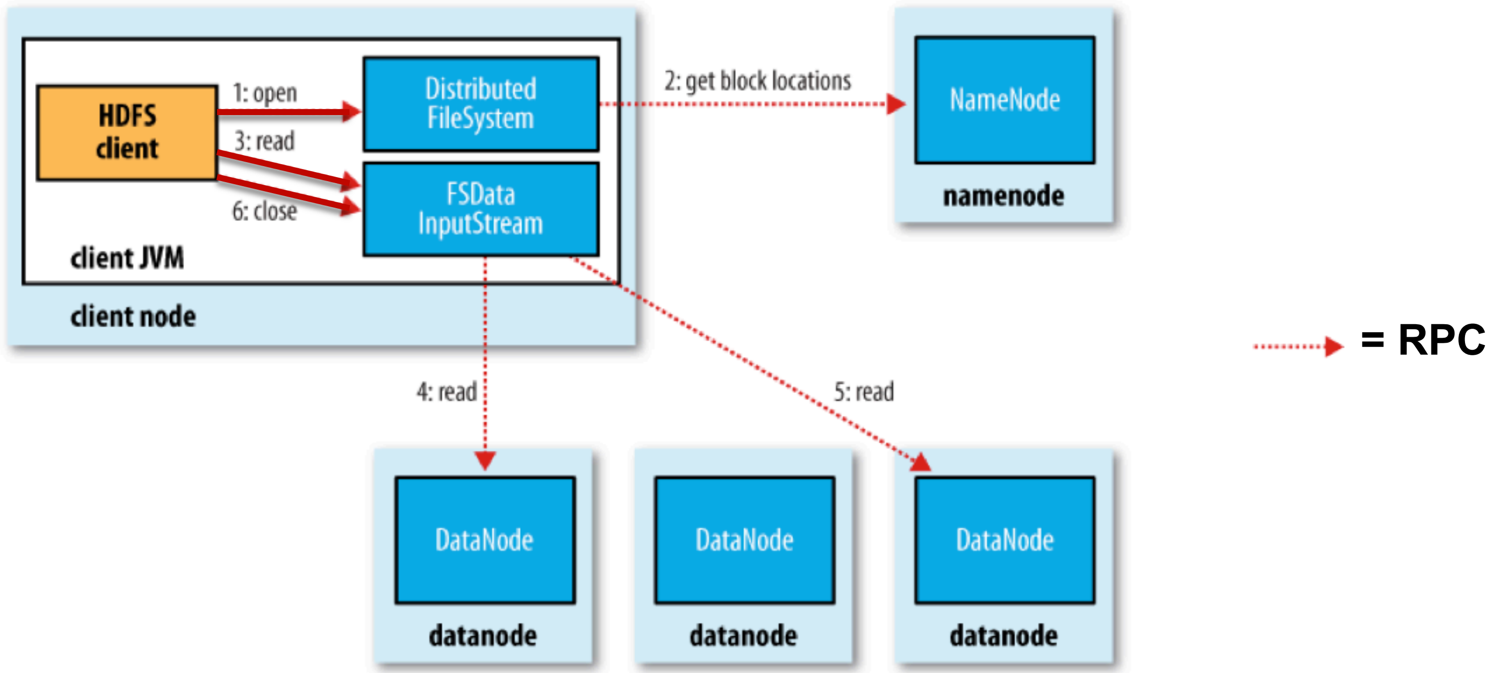
Namenode Responsibilities

- 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

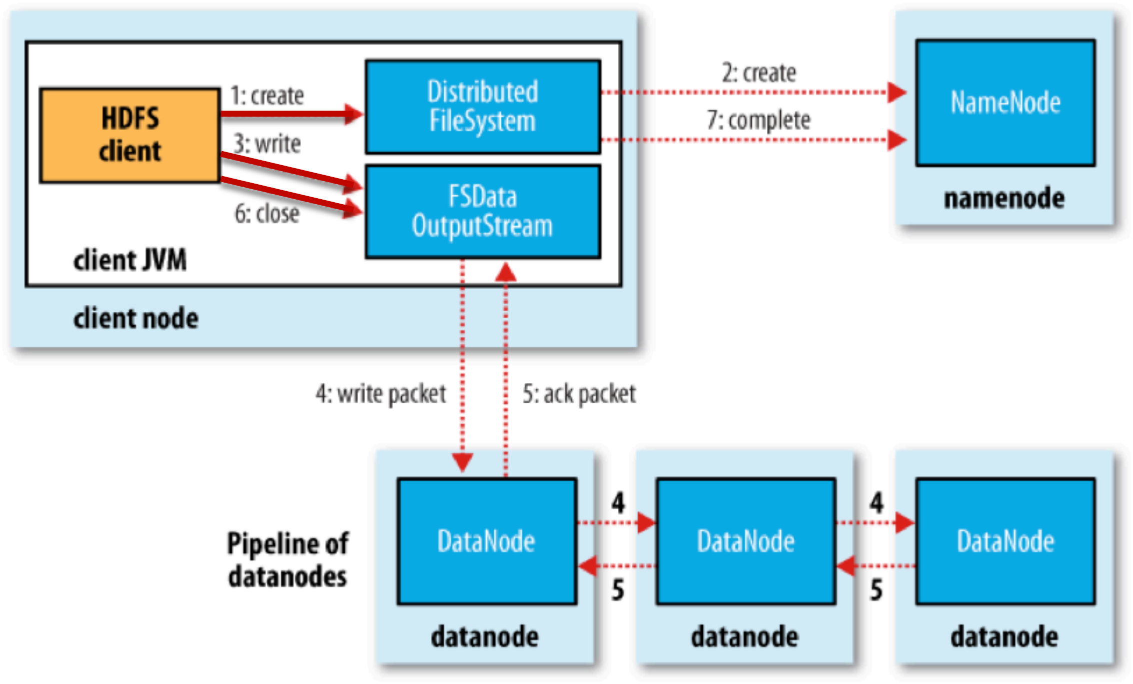
HDFS Architecture



HDFS anatomy of a file read



HDFS anatomy of a file write



= RPC

HDFS Interfaces

- HDFS provides lots of interfaces to allow users to access and manipulate the distributed file system.
  - Command line (from a Linux/UNIX shell)
  - Java API
  - Thrift proxy (supporting C++, Perl, PHP, Python, Ruby)
  - C via a Java Native Interface (JNI)
  - WebDAV for mounting file systems via HTTP
  - HTTP (read-only) & FTP (both read & write)

  and many more…

HDFS Command Line Interface

- URI scheme for accessing local files from the command line: `file://path1/path2/.../filename.ext`
- URI scheme for accessing files across different HDFS servers: `hdfs://hostname/path1/path2/…/filename.ext`
- HDFS supports most of the usual UNIX file system commands. User permissions are set similar to the POSIX file permissions

```
> hdfs dfs -ls /
> hdfs dfs -mkdir hdfs://myfiles
> hdfs dfs -copyFromLocal file://input/docs/1.txt
     hdfs://myfiles/1.txt
> hdfs dfs -cat hdfs://myfiles/1.txt
> hdfs dfs -rm hdfs://myfiles/1.txt
```

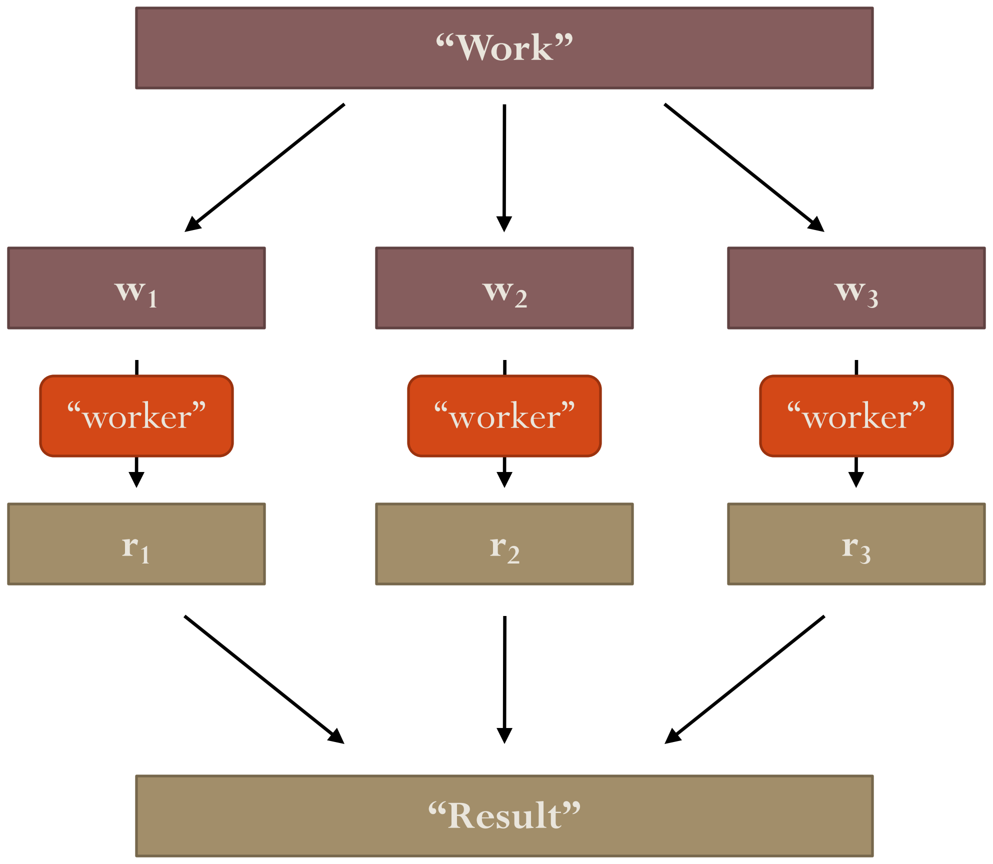
- See here for a full list of HDFS file system commands.

HDFS Java API (I)

Reading data from a Hadoop URI:

```
import java.net.URL;
…
  InputStream in = null;
  try {
     in = new URL("hdfs://host/path").openStream();
     …
     in.skip(2323324);
     …
  } finally {
     in.close();
  }
```

Distributed computing: an old idea



Divide

Conquer

Parallelization Challenges

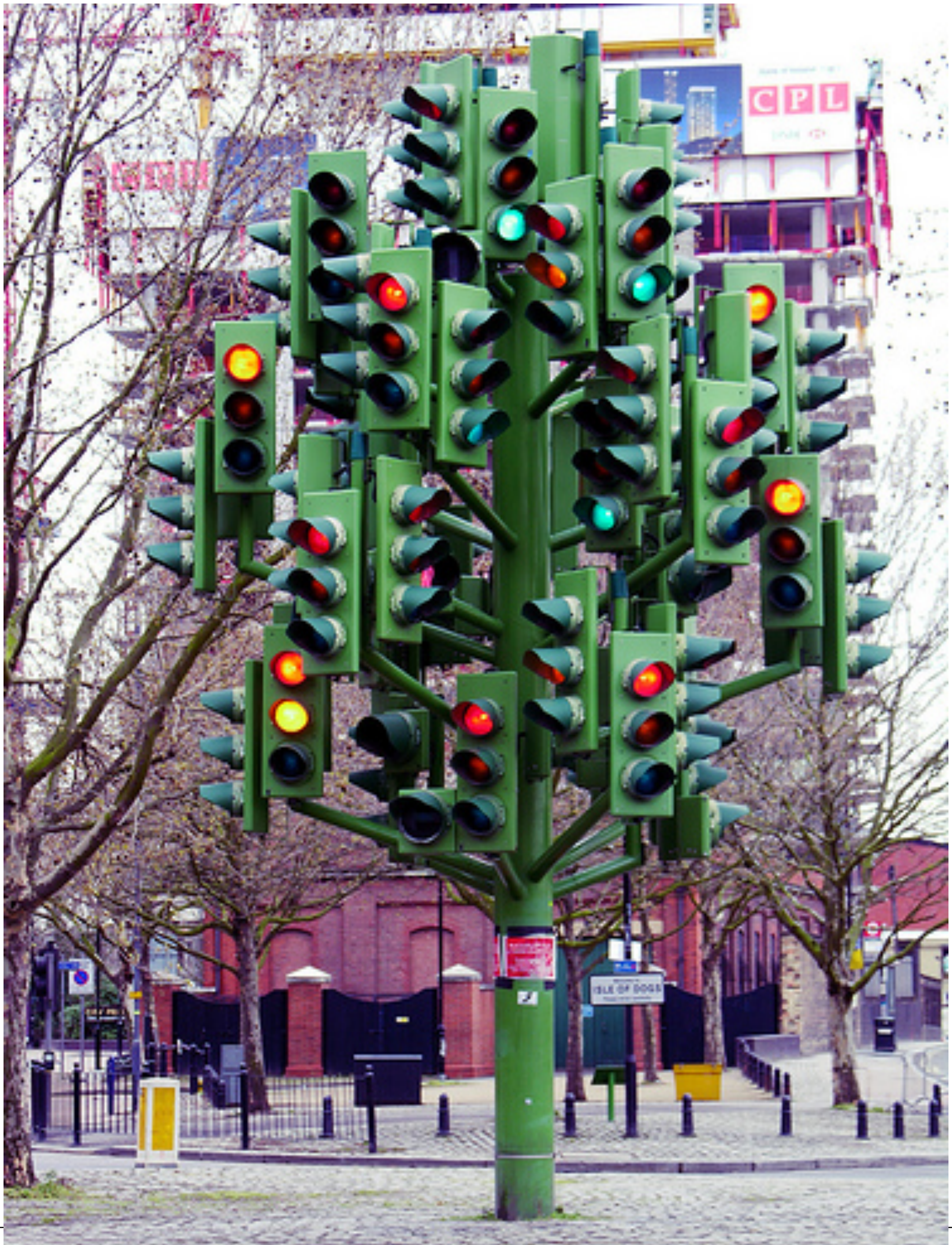
- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the risk of all of these problems?

Source: Ricardo Guimarães Herrmann

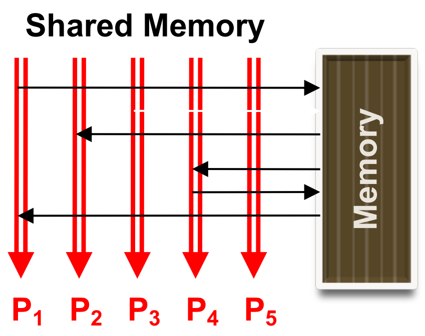
Risk?

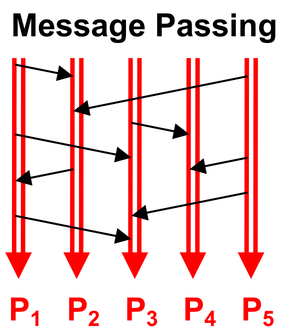
- A lot: for instance, deadlock and starvation
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

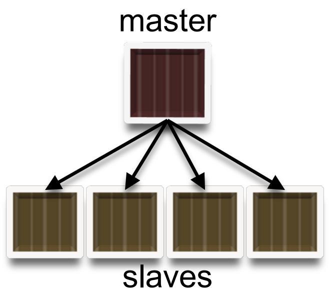

CPL
ISLE OF DOGS

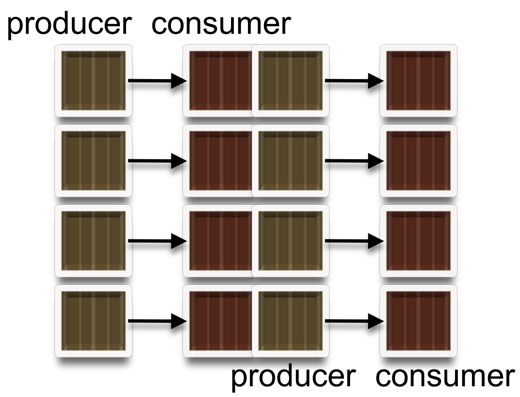
Current approaches

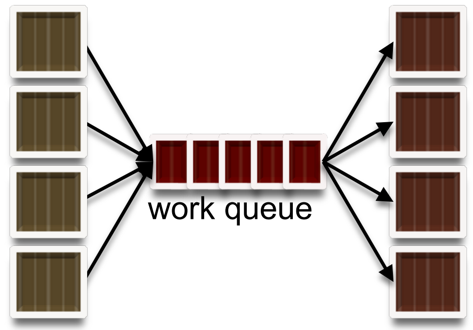
- Programming models
  - Shared memory (pthreads)
  - Shared nothing (Message passing)
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues











So, what?

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across datacenters)
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality can be hard
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything
- MapReduce approach to large scale computing:

What’s the point?

- 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” of large scale computing

- Scale “out”, not “up”
  - Limits of Symmetric Multi-Processing and large shared-memory machines
- Hide system-level details from the application developer
  - Concurrent programs are difficult to reason about and harder to debug
- 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
- Share nothing computation
  - A shared memory is always a bottleneck

Typical Large-Data Problem

Map

- Iterate over a large number of records in parallel
- Extract something of interest from each iteration
- Shuffle and sort intermediate results of different concurrent iterations
- Aggregate intermediate results
- Generate final output

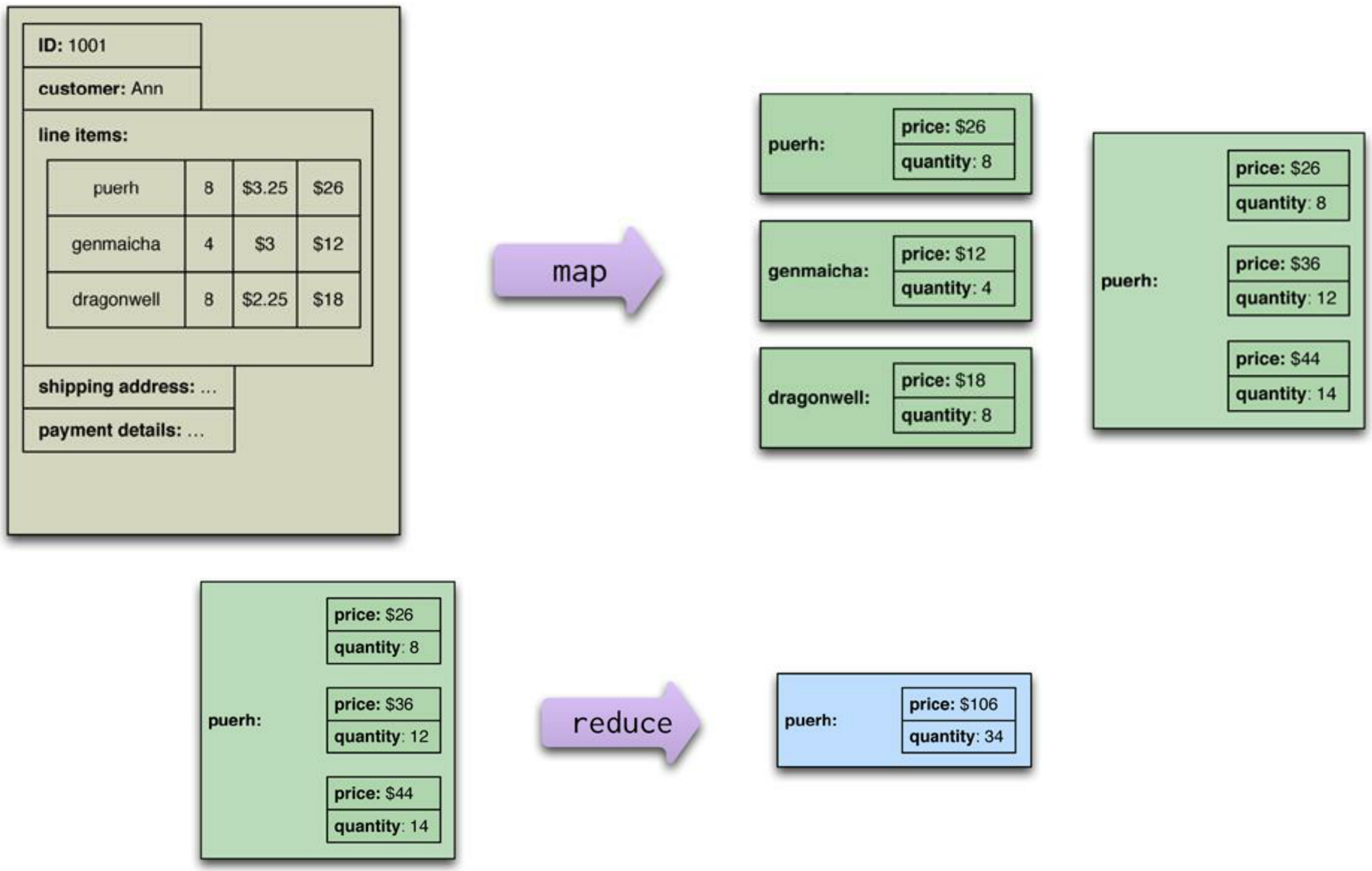
Reduce

Key idea: provide a functional abstraction for these two operations

MapReduce

- Map: takes as input an object with a key (k,v) and returns a bunch of key-value pairs: $(k_1,v_1),(k_2,v_2),\ldots,(k_n,v_n)$
- The framework collects all the pairs with the same key k and associates with k all the values for k: $(k,[v_1,..,v_n])$
- Reduce: takes as input a key and a list of values $(k,[v_1,..,v_n])$ and combine them somehow.

Example


ID: 1001
customer: Ann
line items:
puerh 8 $3.25 $26
genmaicha 4 $3 $12
dragonwell 8 $2.25 $18
shipping address: ...
payment details: ...
map
puerh: price: $26 quantity: 8
genmaicha: price: $12 quantity: 4
dragonwell: price: $18 quantity: 8
puerh: price: $26 quantity: 8 price: $36 quantity: 12 price: $44 quantity: 14
puerh: price: $26 quantity: 8 price: $36 quantity: 12 price: $44 quantity: 14
reduce
puerh: price: $106 quantity: 34

MapReduce programming

- Basic data structure: key-value pairs
- Programmers specify two functions:
  - **map** (k1, v1) → [(k2, v2)]
  - **reduce** (k1, [v1]) → [(k2, v2)]
    - (k, v) denotes a (key, value) pair
    - [...] denotes a list
    - normally the keys of input elements are not relevant
    - keys do not have to be unique: different pairs can have the same key
- The execution framework handles everything else!!

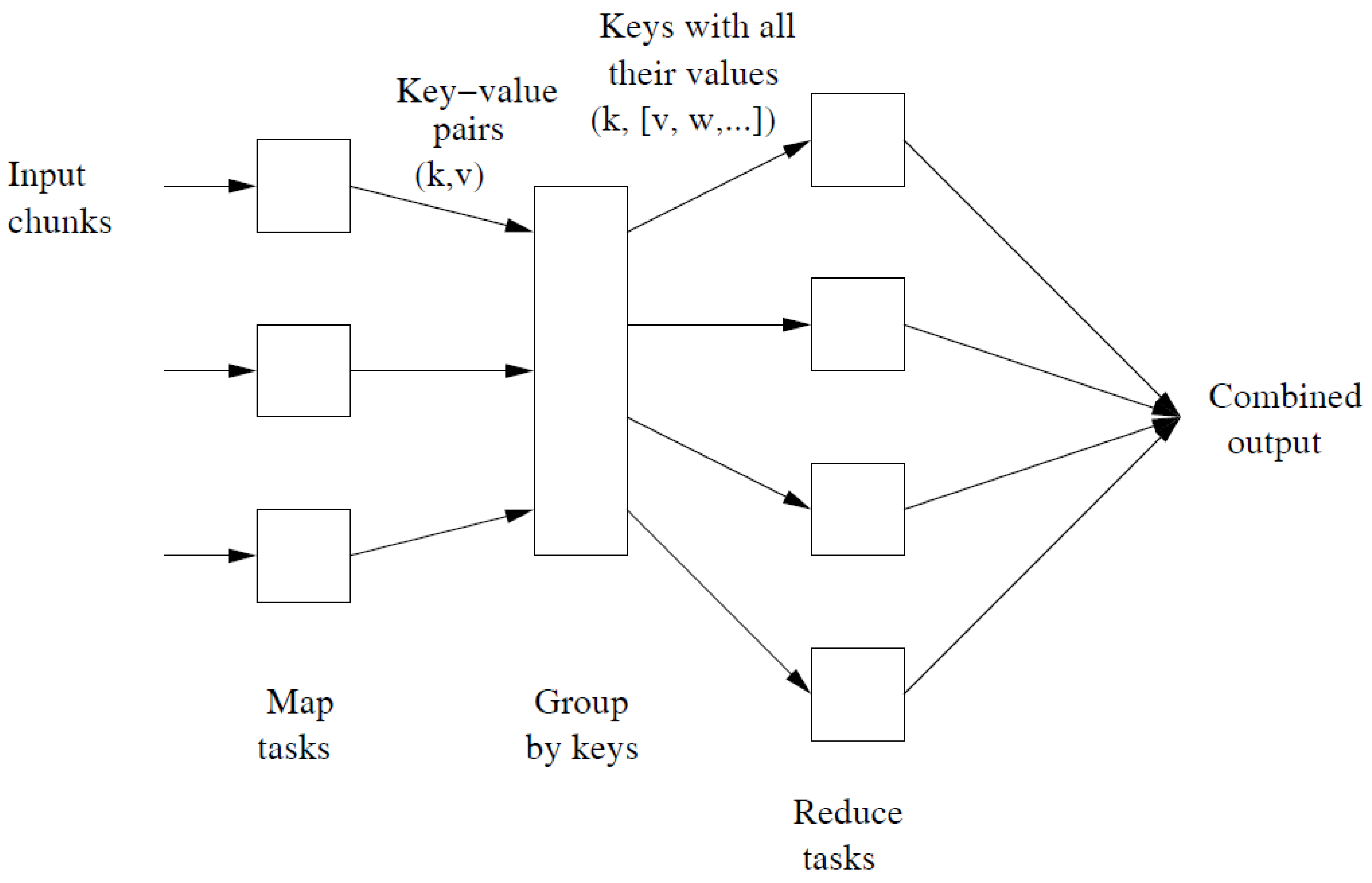
MapReduce program

- A MapReduce program, referred to as a job, consists of:
  - code for Map and a code for Reduce packaged together
  - configuration parameters (where the input lies, where the output should be stored)
  - the input, stored on the underlying distributed file system
- Each MapReduce job is divided by the system into smaller units called tasks
  - Map tasks
  - Reduce tasks
- Input and output of MapReduce jobs are stored on the underlying distributed file system

MapReduce execution process

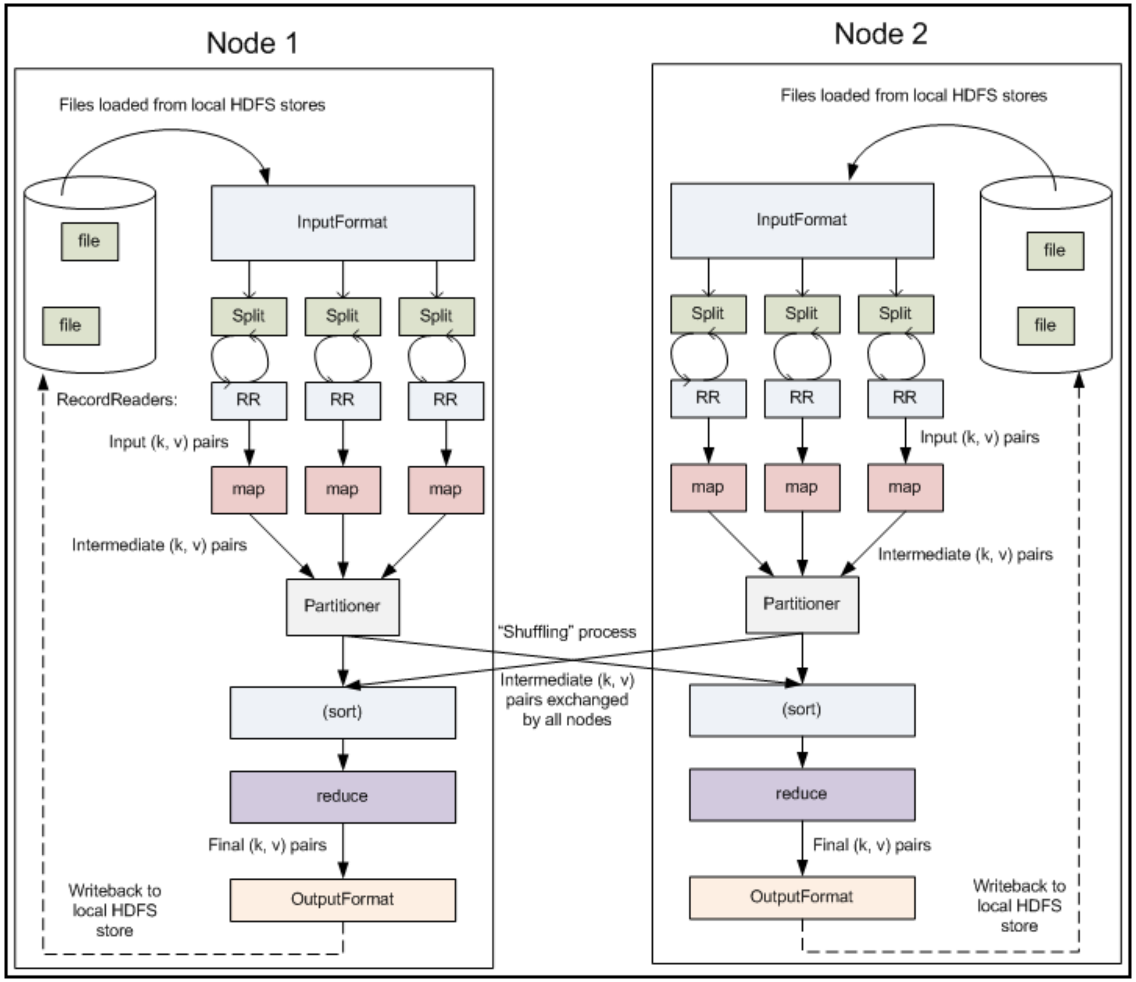
1) Some number of Map tasks each are given one or more chunks of data.
2) Each Map task turns the chunk into a sequence of key-value pairs.
   - The way key-value pairs are produced is determined by the code written by the user for the Map function.
3) The key-value pairs from each Map task are collected by a master controller and sorted and grouped by key (Shuffle and sort).
4) The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.
5) The Reduce tasks work on one key at a time and combine all the values associated with that key in some way.
   - The way values are combined is determined by the code written by the user for the Reduce function.
6) Output key-value pairs from each reducer are written persistently back onto the distributed file system
7) The output ends up in *r* files, where *r* is the number of reducers.
   - the r files often serve as input to yet another MapReduce job

MapReduce process



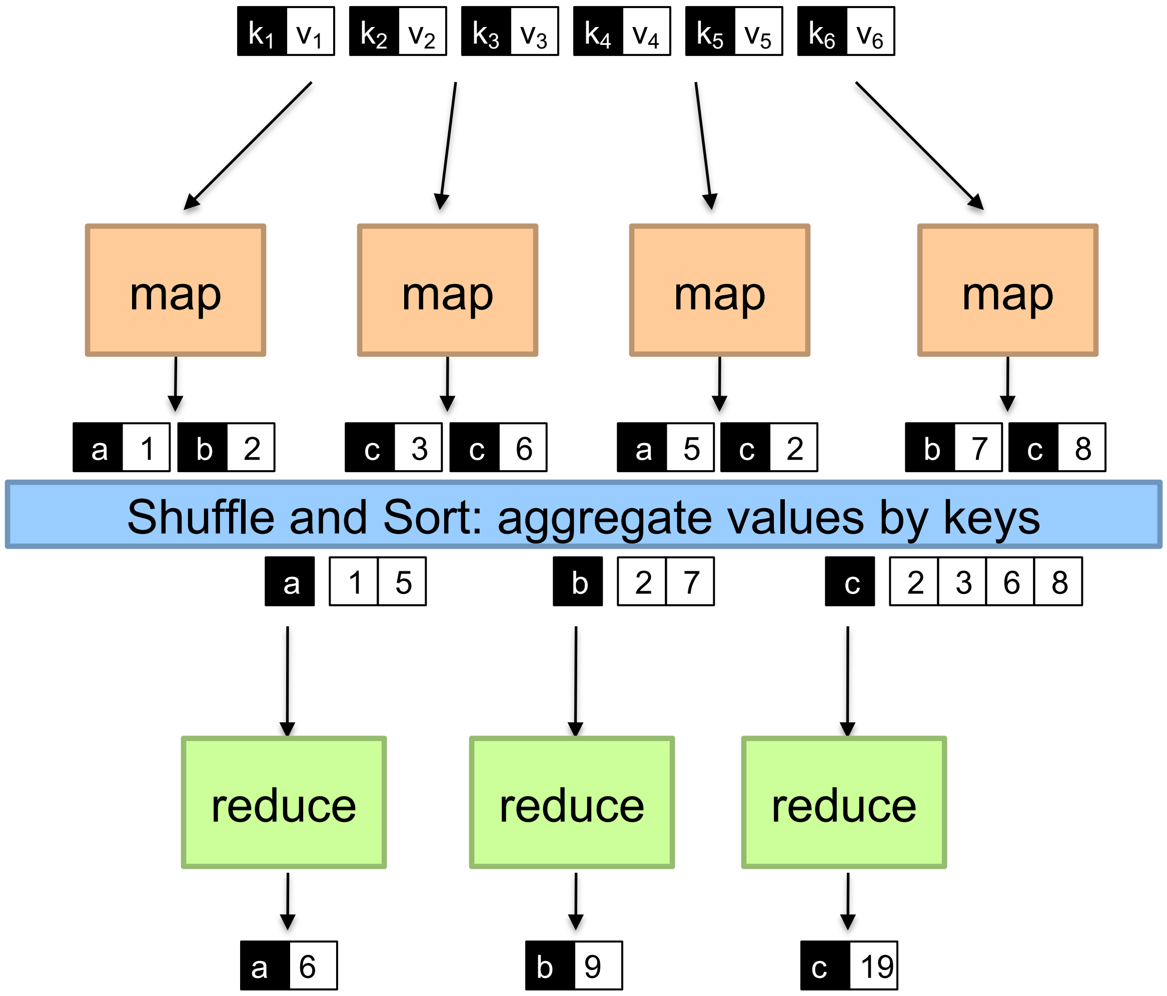
More Detailed View

- Data is loaded from and written to **HDFS**.
- Worker nodes serve as both **Mappers** and **Reducers**.
- **Synchronization** is required between the map and reduce phases.



http://developer.yahoo.com/hadoop/tutorial/

A generic example



A practical example: Word Count

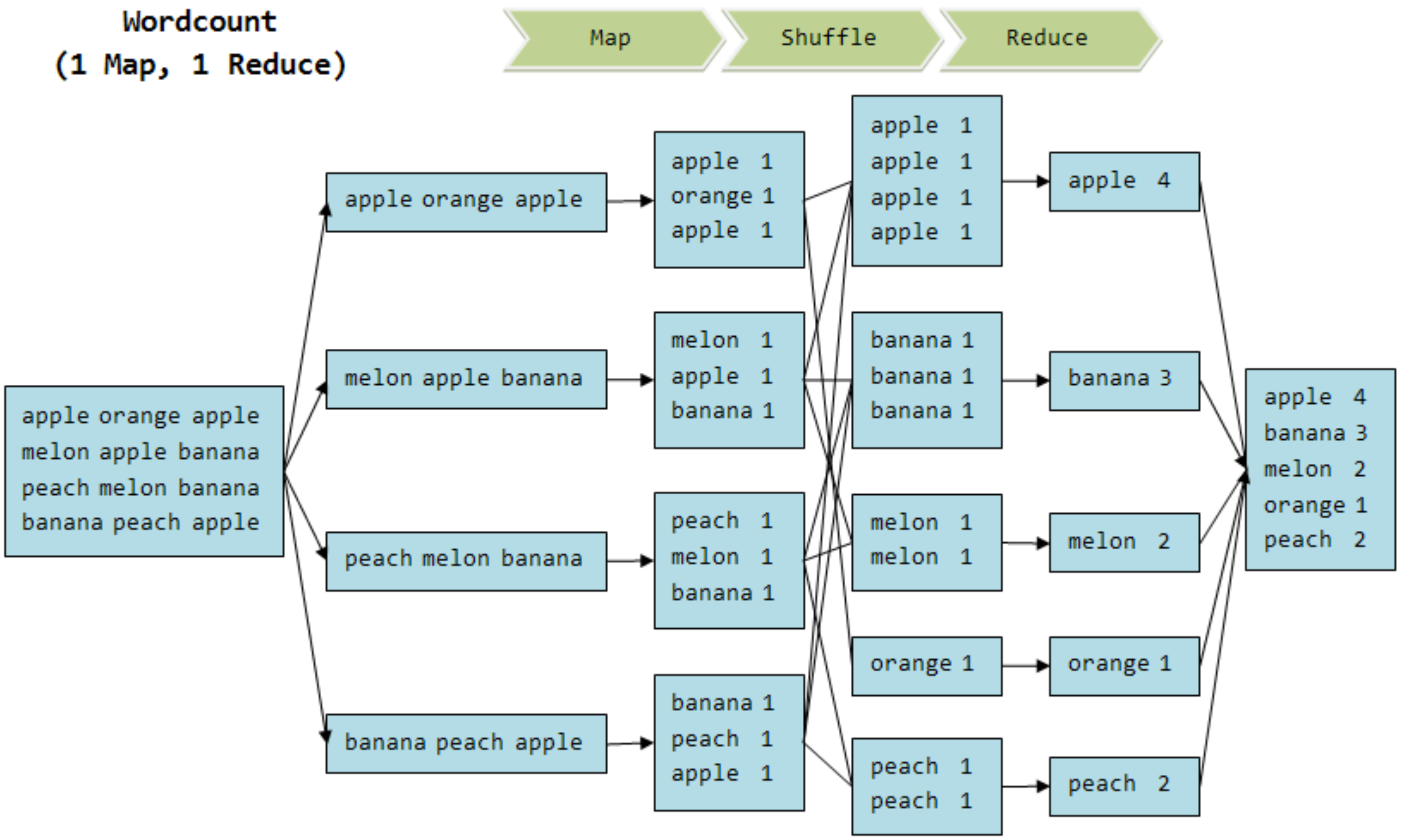
- **Problem**: counting the number of occurrences for each word in a collection of documents.
- **Input**: a repository of documents, each document is an element
- **Map**: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

  (w1, 1), (w2, 1), . . . , (wn, 1)
- **Shuffle**: groups by key and generates pairs of the form

  (w1, [1, 1, . . . , 1]) , . . . , (wn, [1, 1, . . . , 1])
- **Reduce**: adds up all the values and emits:

  (w1, k) , . . . , (wn, l)
- **Output**: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents.

Word Count in practice



Implementation in Python

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

Reduce(String term, counts[]):
    int sum = 0;
    for each c in counts:
        sum += c;
    Emit(term, sum);
```

A Map in Java

```
public static class Map extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable uno = new IntWritable(1);
    private Text parola = new Text();
    public void map(LongWritable chiave, Text testo,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String linea = testo.toString();
      StringTokenizer tokenizer = new StringTokenizer(linea);
      while (tokenizer.hasMoreTokens()) {
        parola.set(tokenizer.nextToken());
        output.collect(parola, uno);
      }
    }
}
```

A Reduce in Java

```
public static class Reduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(
         Text chiave,
         Iterator<IntWritable> valori,
         OutputCollector<Text, IntWritable> output,
         Reporter reporter)
    throws IOException {
      int somma = 0;
      while (valori.hasNext()) {
        somma += valori.next().get();
      }
      output.collect(chiave, new IntWritable(somma));
  }
}
```

Another example: Word Length Count

- **Problem**: counting how many words of certain lengths exist in a collection of documents.
- **Input**: a repository of documents, each document is an element
- **Map**: reads a document and emits a sequence of key-value pairs where the key is the length of a word and the value is the word itself:

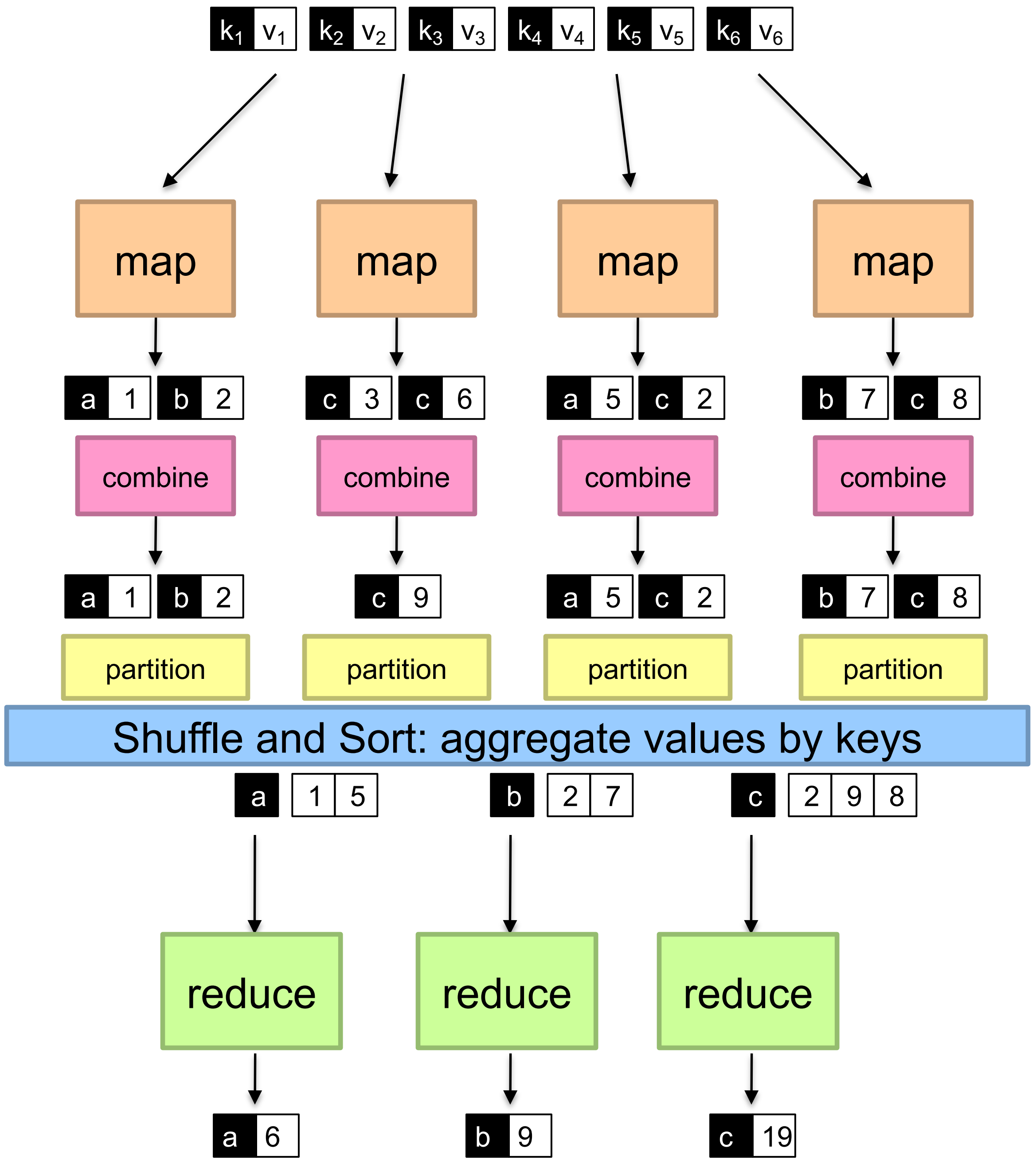
  (i,w1), . . . , (j,wn)
- **Shuffle and sort**: groups by key and generates pairs of the form

  (1, [w1, . . . , wk]) , . . . , (n, [wr, . . . , ws])
- **Reduce**: counts the number of words in each list and emits:

  (1, l) , . . . , (p, m)
- **Output**: (l,n) pairs, where l is a length and n is the total number of words of length l in the input documents.

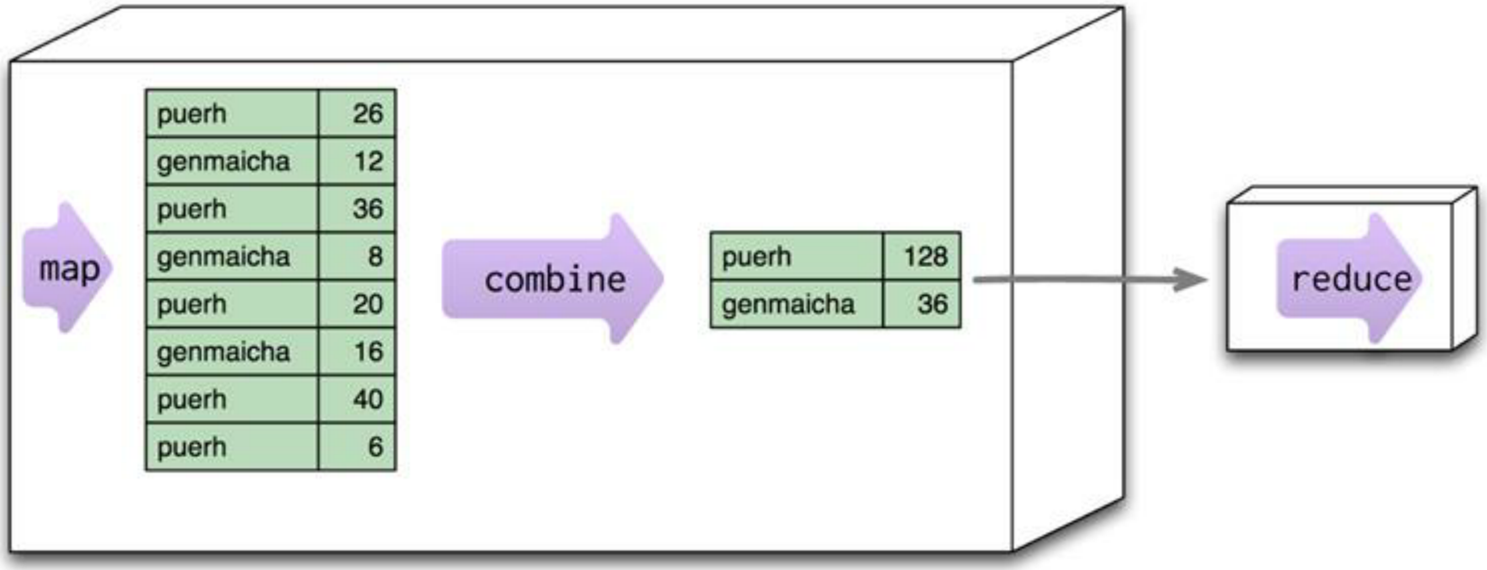
Introducing combiners

- When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks
- In this case we also apply a combiner to the Map function
- In many cases the same function can be used for combining as the final reduction
- Shuffle and sort is still necessary!
- Advantages:
  - it reduces the amount of intermediate data
  - it reduces the network traffic



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
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 9 8
reduce
reduce
reduce
a 6
b 9
c 19

An example of combiner



puerh	26
genmaicha	12
puerh	36
genmaicha	8
puerh	20
genmaicha	16
puerh	40
puerh	6

puerh	128
genmaicha	36

Example: Word Count with combiners

- **Input**: a repository of documents, each document is an element
- **Map**: reads a document and emits a sequence of key-value pairs where keys are words of the documents and values are equal to 1:

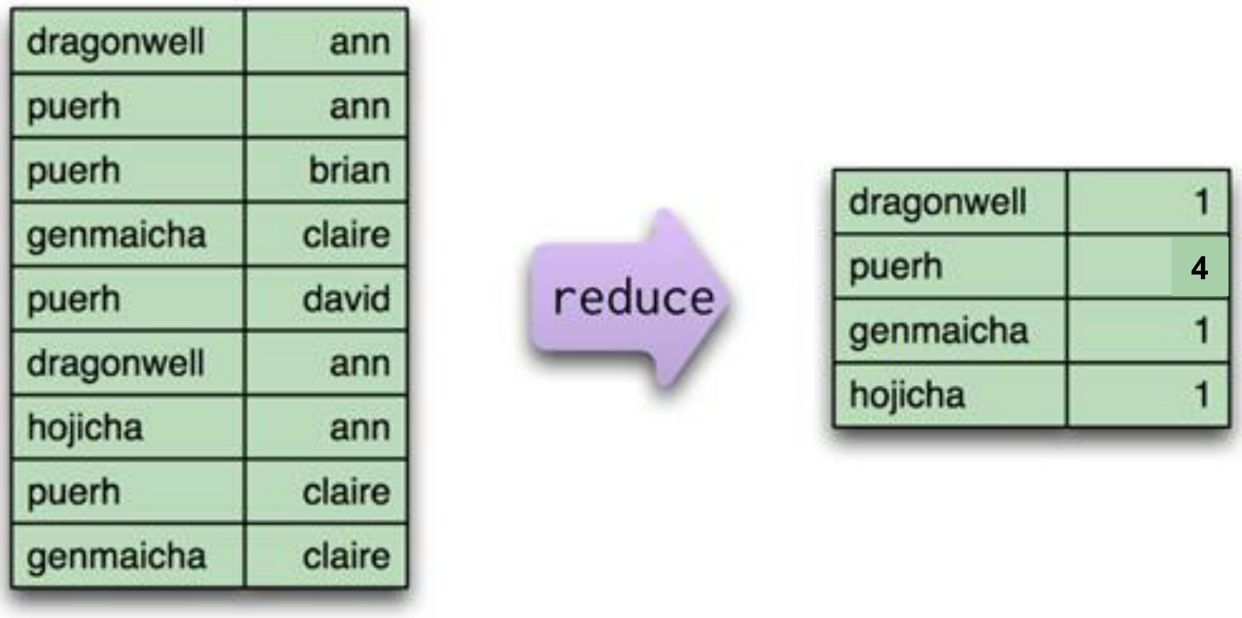
  (w1, 1), . . . , (wn, 1)
- **Combiner**: groups by key, adds up all the values and emits:

  (w1, i), . . . , (wn, j)
- **Shuffle and sort**: groups by key and generates pairs of the form

  (w1, [p, . . . , q]) , . . . , (wn, [r, . . . , s])
- **Reduce**: adds up all the values and emits:

  (w1, k) , . . . , (wn, l)
- **Output**: (w,m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents.

Reduces and combiners can differ!

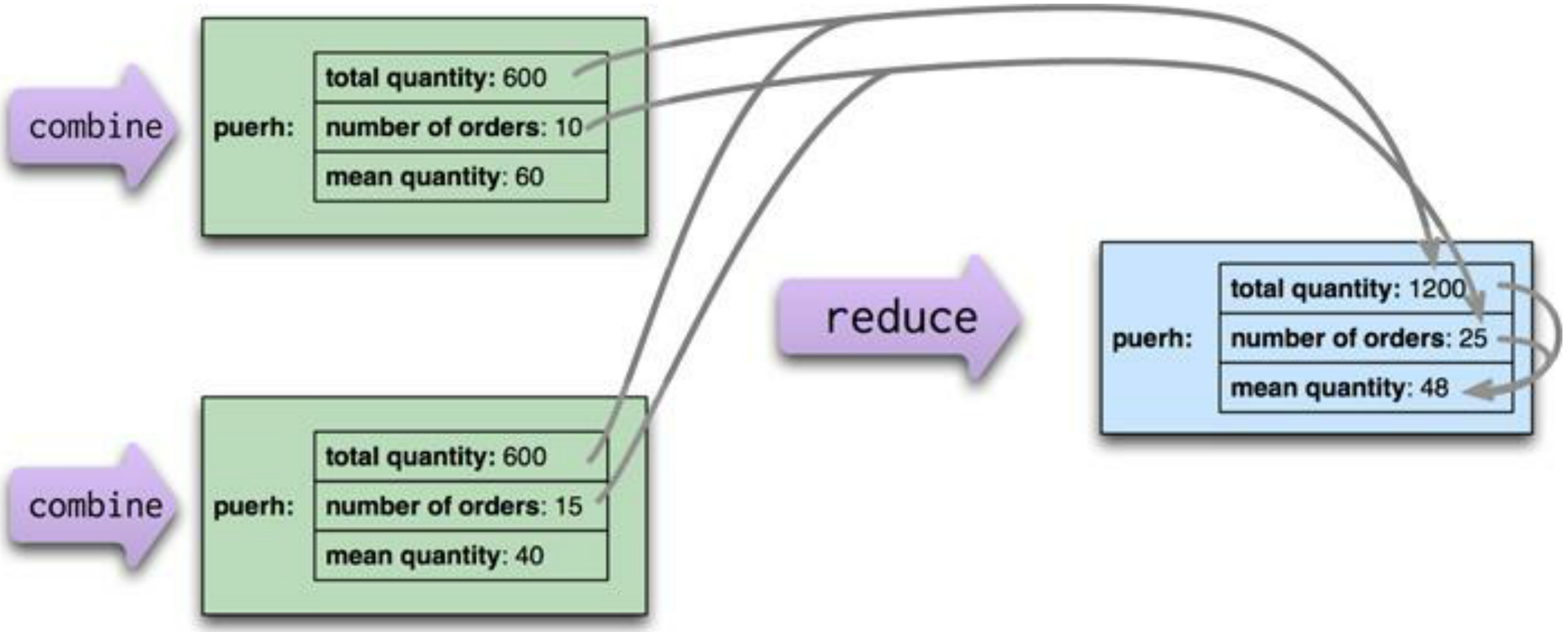


dragonwell	ann
puerh	ann
puerh	brian
genmaicha	claire
puerh	david
dragonwell	ann
hojicha	ann
puerh	claire
genmaicha	claire

dragonwell	1
puerh	4
genmaicha	1
hojicha	1

Can we make a Combiner from this Reducer?

Using combiners for calculating avarages



How many maps and reduces?

- Can be set in a configuration file (JobConfFile) or during command line execution (for maps, it is just an hint)
- Number of Maps: driven by the number of HDFS blocks in the input files. You can adjust your HDFS block size to adjust the number of maps.
- Number of Reducers: can be user defined (the default is 1 but in general is driven by the number of HDFS blocks of the intermediate files)

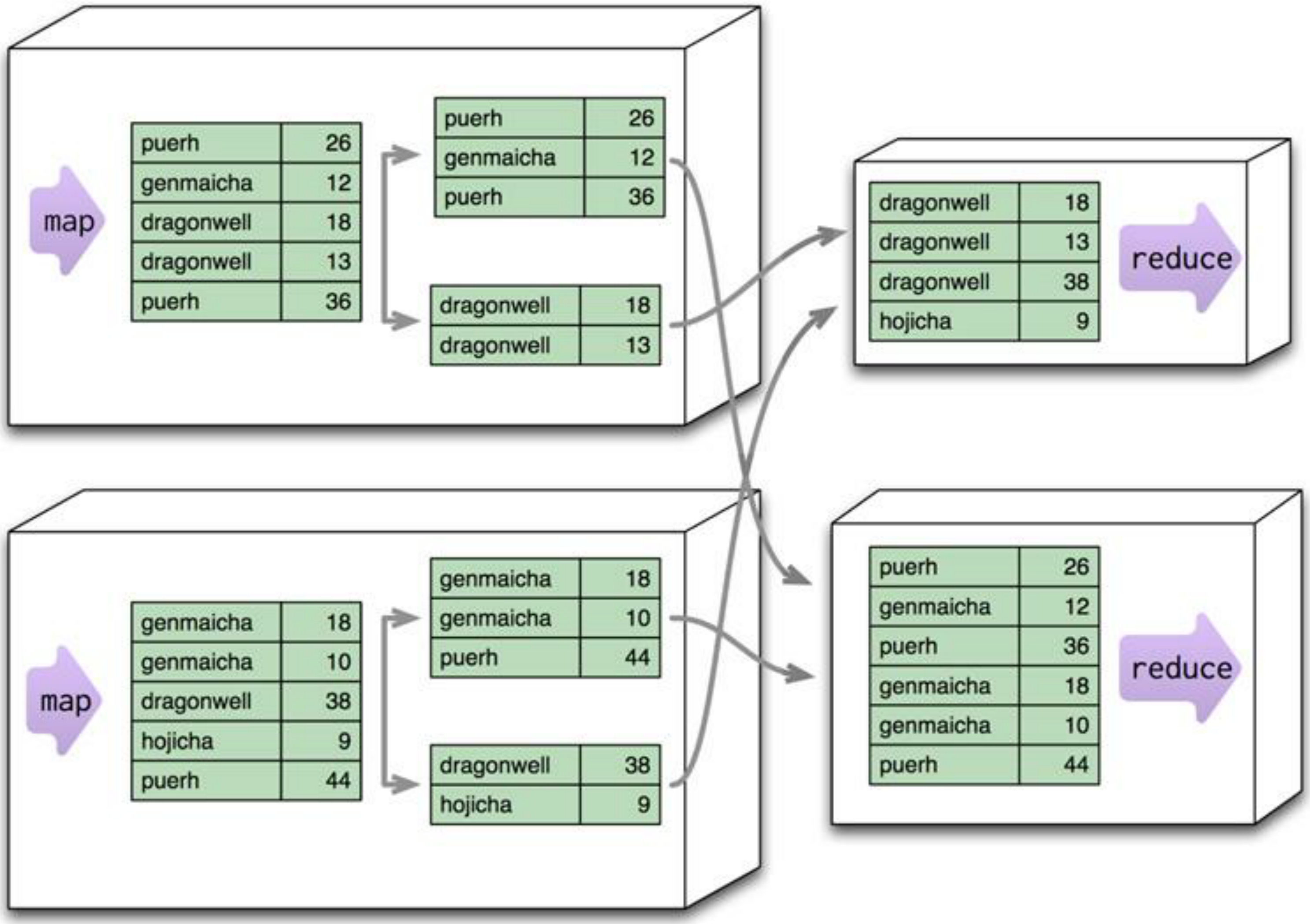
Partitioners

- The number of Reduce tasks, say r, is fixed (by the user or automatically by the system)
- The keyspace of the intermediate key-value pairs is evenly distributed over the reducers with a hash function (same keys in different mappers end up at the same reducer)
- Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files
- Local files are organized as a sequence of (key, list-of-values) pairs
- Each file is destined for one of the Reduce tasks.
- Optionally, users can specify their own hash function or other method for assigning keys to Reduce tasks.
- However, whatever algorithm is used, each key is assigned to one and only one Reduce task.

Customization of partitioners

- We can specify a partitioner that:
  - divides up the intermediate key space
  - assigns intermediate key-value pairs to reducers
  - n partitions $\rightarrow$ n reducers
- The default partitioner assigns approximately the same number of keys to each reducer.
- But partitioner only considers the key and ignores the value
- An imbalance in the amount of data associated with each key is relatively common in many text processing applications
  - In texts the frequency of any word is inversely proportional to its rank in the frequency table
  - The most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

Example of partitioners



Reduce Tasks, Compute Nodes, and Skew

- If we want maximum parallelism, then we could use one Reduce task to execute each single key and its associated value list and execute each Reduce task at a different compute node, so they would all execute in parallel.
- Skew:
  - there is often significant variation in the lengths of the value lists for different keys
  - different reducers take different amounts of time
  - significant difference in the amount of time each reduce task takes.
- Overhead and physical limitations:
  - There is overhead associated with each task we create
  - Often there are far more keys than there are compute nodes available.
- We can reduce the impact of skew by using fewer Reduce tasks.
- We can further reduce the skew by using more Reduce tasks than compute nodes. In that way, long Reduce tasks might occupy a compute node fully, while several shorter Reduce tasks might run sequentially at a single compute node.

MapReduce: the complete picture

- Programmers specify two functions:

  map (k1, v1) → [(k2, v2)]

  reduce (k2, [v2]) → [(k3, v3)]

  - All values with the same key are reduced together
- Programmers can also specify:

  combine (k2, [v2]) → [(k3, v3)]

    - Mini-reducers that run after the map phase
    - Used as an optimization to reduce network traffic

  partition (k2, number of partitions) → partition for k2

    - Divides up key space for parallel reduce operations
- The execution framework handles everything else…

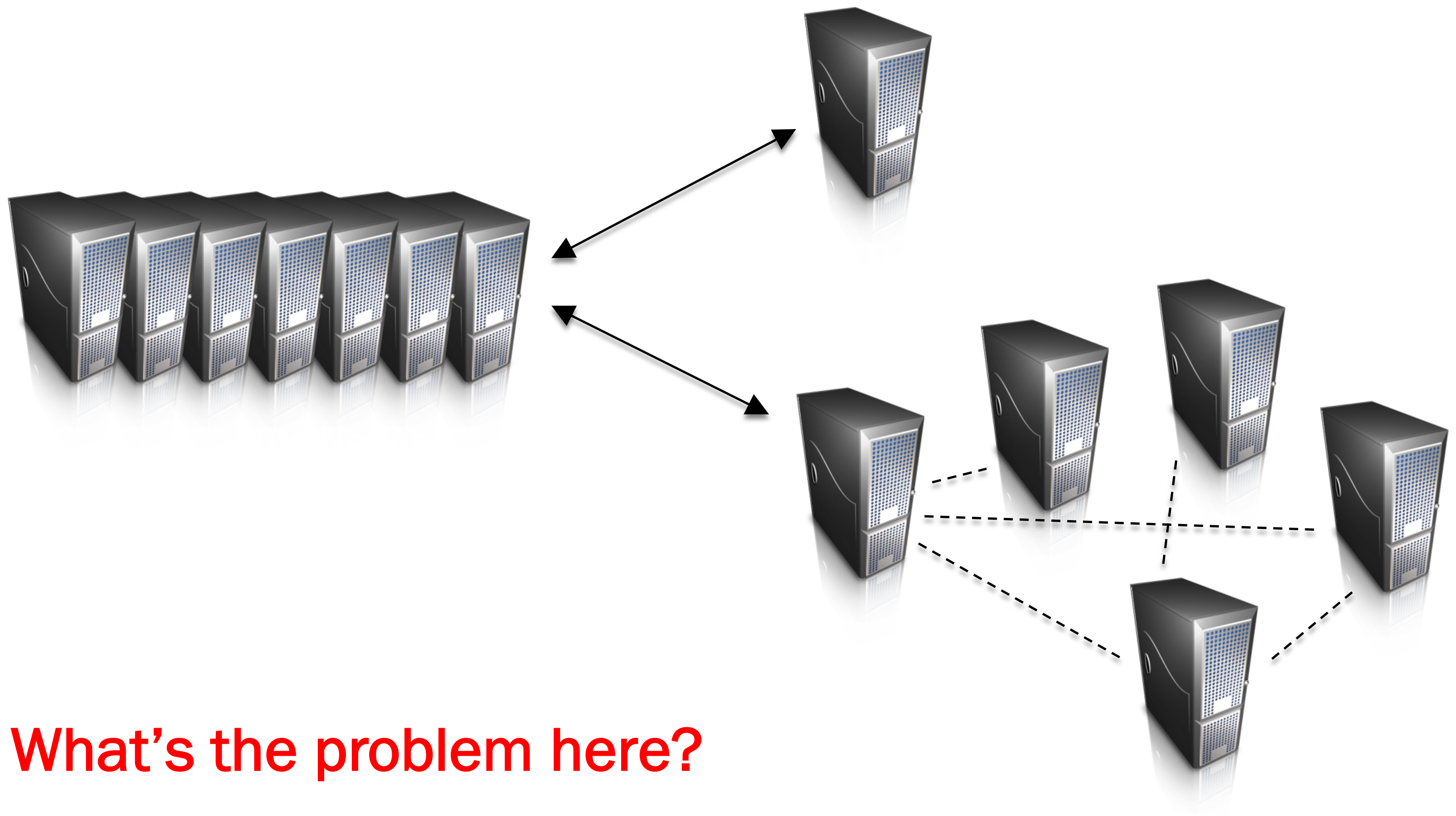
MapReduce runtime

- Important idea behind MapReduce is separating the what of distributed processing from the how
- The developer submits the job to the submission node of a cluster
- The execution framework (the "runtime") takes care of everything else:
  - it transparently handles all aspects of distributed code execution
  - on clusters ranging from a single node to a few thousand nodes

Scheduling

- It is not uncommon for Hadoop jobs to have thousands of individual tasks that need to be assigned to nodes in the cluster.
- In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently, making it necessary for the scheduler to maintain some sort of a task queue and to track the progress of running tasks so that waiting tasks can be assigned to nodes as they become available

How do we get data to the workers?



What’s the problem here?

Run-Time Principles in Hadoop

- Data locality:
  - Data and workers must be close to each other
- Shared nothing architecture
  - Each node is independent and self-sufficient

Locality enforcement

- 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
- The tasks are created from the input splits in the shared file system
  - 1 map per split
  - N reducers determined by the configuration
    - e.g.: (0.95 or 1.75) * (<no. of nodes> * <no. of maximum containers per node>)
- If this is not possible (e.g., a node is already running too many tasks)
  - new tasks will be started elsewhere
  - the necessary data will be streamed over the network
- Optimization
  - prefer nodes that are on the same rack in the datacenter as the node holding the relevant data block
  - inter-rack bandwidth is significantly less than intra-rack bandwidth!

Shared-nothing architecture

- Coordinating the process in a distributed computation is hard
  - How to distribute the load over the available nodes?
  - How to serialize the data for the transmission?
  - How to handle an unresponsive process?
- MapReduce's shared-nothing architecture means that tasks have no dependence on one other
- Programmers do not worry about the distributed computation issues
- They need only to write the Map and Reduce functions

Synchronization

- In general, synchronization refers to the mechanisms by which multiple concurrently running processes "join up", for example, to share intermediate results or otherwise exchange state information
- In MapReduce, synchronization is accomplished by a "barrier" between the map and reduce phases of processing
- Intermediate key-value pairs must be grouped by key, which is accomplished by a large distributed sort involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks
- This necessarily involves the transfer of intermediate data over the network, and therefore the process is known as "shuffle and sort"
- A MapReduce job with *m* mappers and *r* reducers involves up to *m x r* distinct transfer operations, since each mapper may have intermediate output going to every reducer

Coping With Node Failures

- The worst thing that can happen is that the compute node at which the Master is executing fails
  - In this case, the entire map-reduce job must be restarted.
- Other failures will be detected and managed by the Master, and the map-reduce job will complete eventually.
- Failure of a worker
  - the tasks assigned to this Worker will have to be redone
  - the Master:
    - sets the status of each failed tasks to idle
    - reschedules them on a Worker when one becomes available

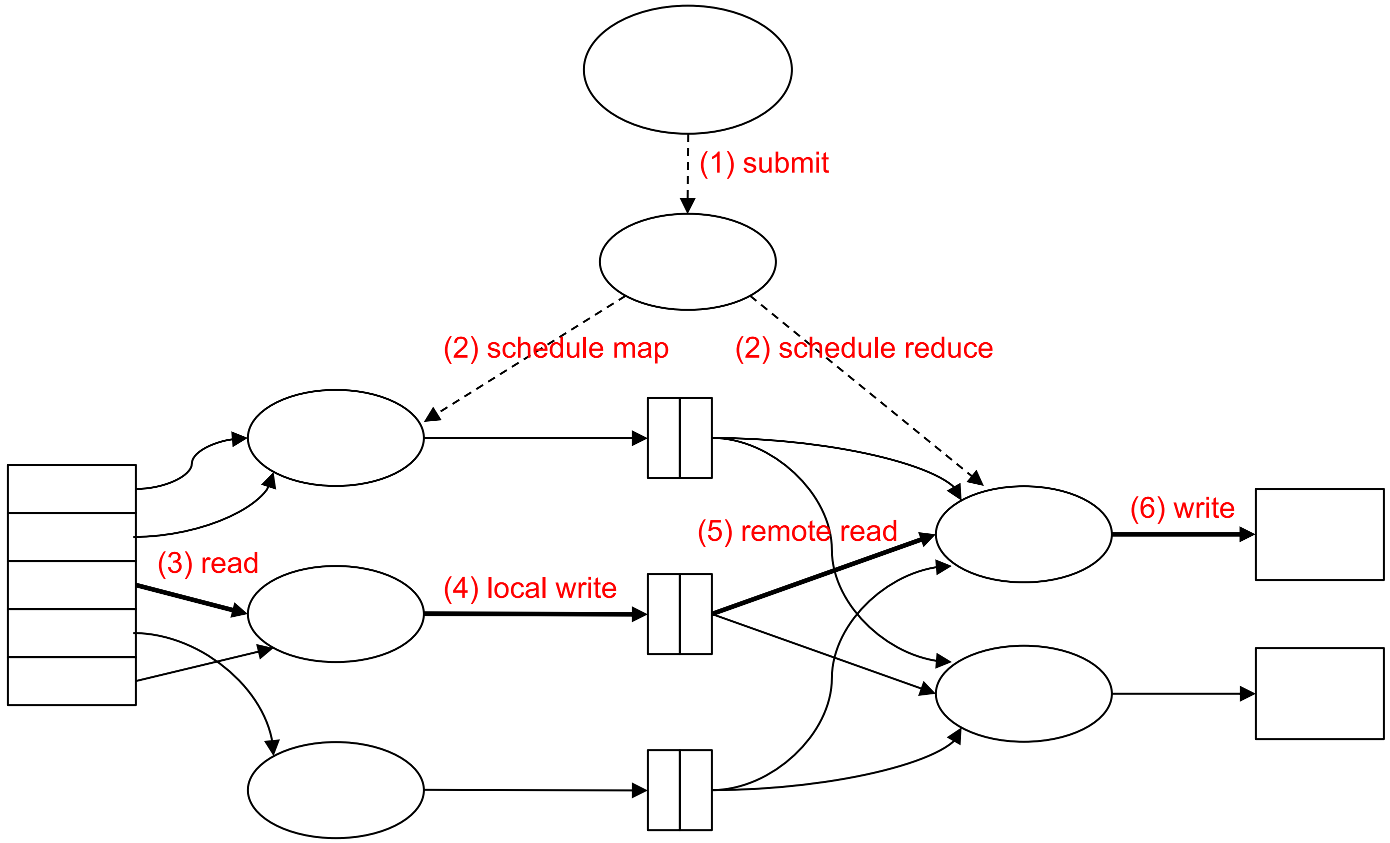
Error and fault handling

- The MapReduce execution framework must accomplish all the tasks above in a hostile environment, where errors and faults are the norm, not the exception
- Since MapReduce was explicitly designed around commodity servers, the runtime must be especially resilient. In large clusters, disk failures are common and RAM experiences more errors than one might expect
- Datacenters suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.).
- And that's just hardware. No software is bug-free…
- Furthermore, any sufficiently large dataset will contain corrupted data or records that are managed beyond a programmer's imagination, resulting in errors that one would never think to check for or trap

MapReduce at work

- Master-slave architecture
- Taking advantage of a library provided by a map-reduce system such as Hadoop, the user program generates:
  - a Master controller process
  - some number of Worker processes at different compute
- The Master process coordinates all the jobs run on the system by scheduling tasks
- The Worker processes run tasks and send progress reports to the Master process
- If a task fails, the Master process can reschedule it on a different Workers

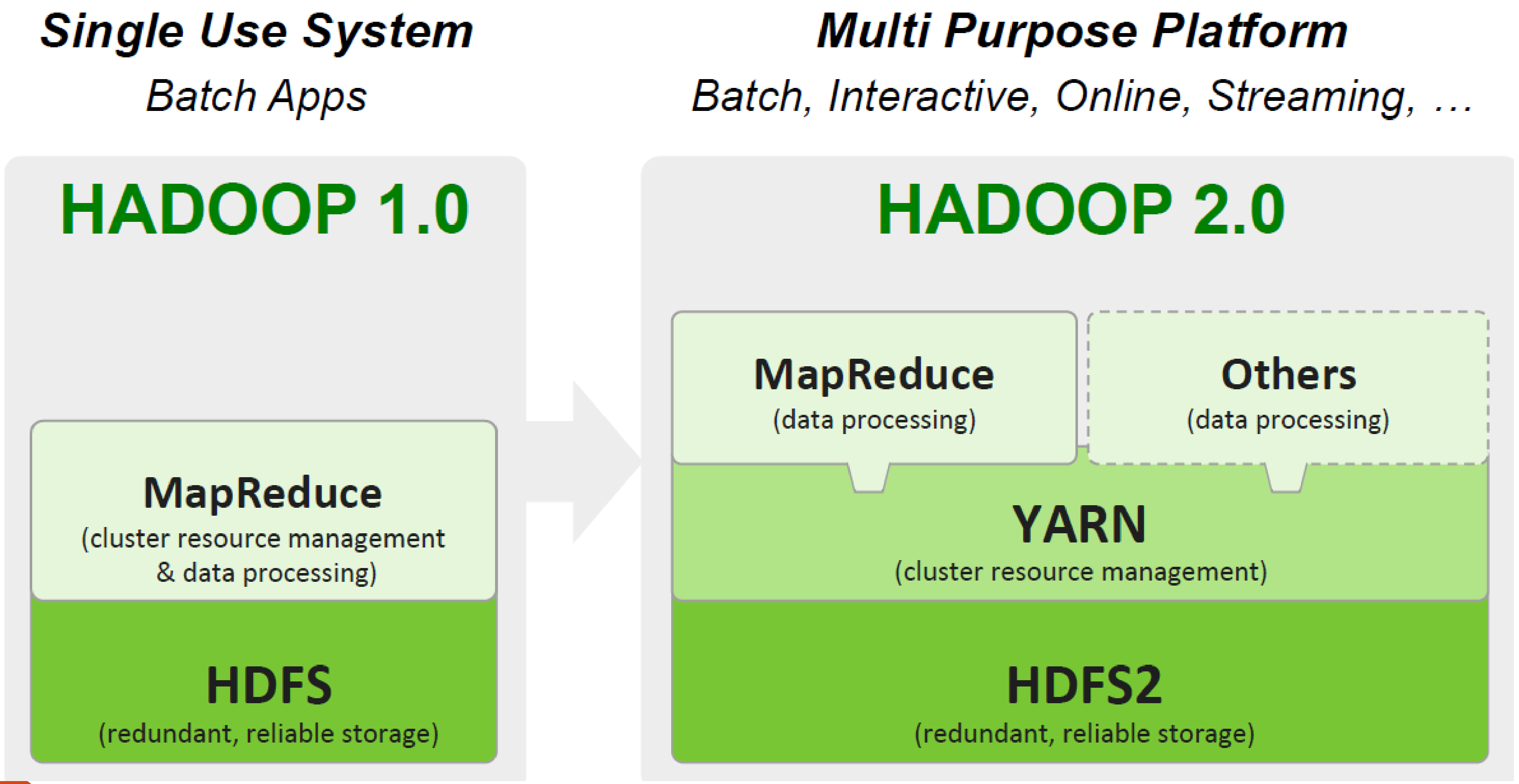
Map-Reduce Execution



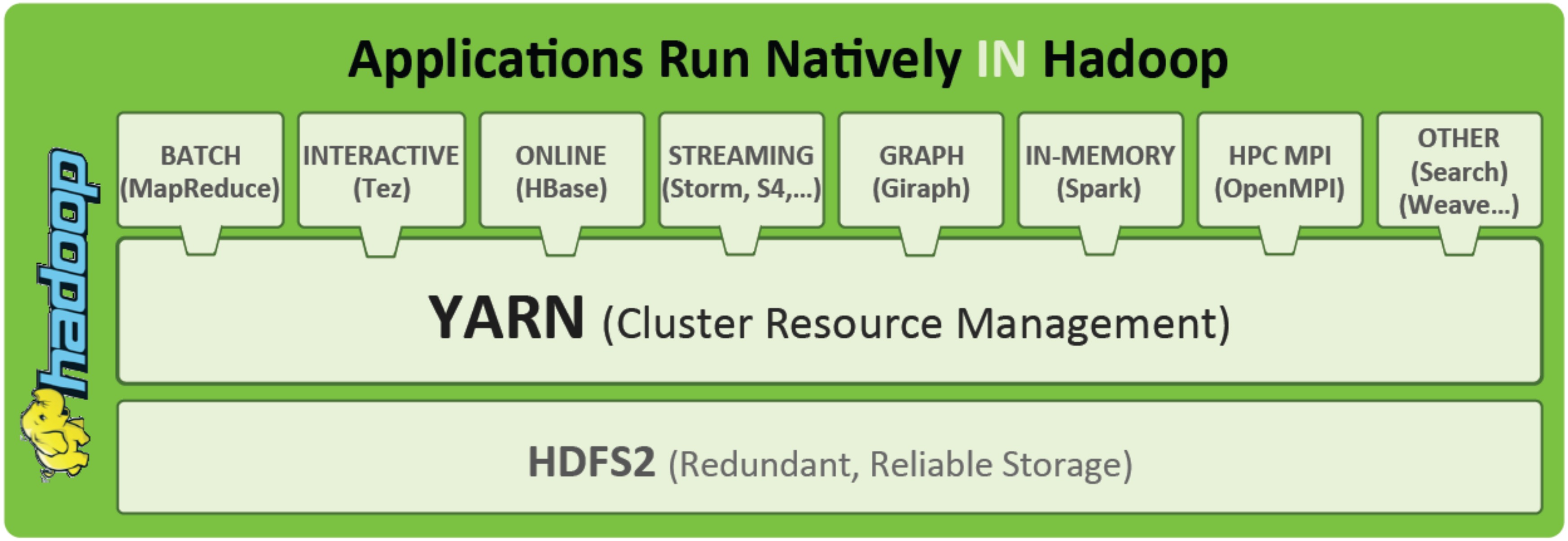
MapReduce execution

- Normally, a Worker handles either Map tasks (a Map worker) or Reduce tasks (a Reduce worker), but not both.
- The Master:
  - creates a number of Map tasks and a number of Reduce tasks, as selected by the user program.
  - assigns tasks to Workers
  - keeps track of the status of each Map and Reduce task (idle, executing at a particular Worker, completed).
- The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task and informs the Master of the location and sizes of each of these files.
- When a Reduce task is assigned by the Master to a Worker, that task is given all the files that form its input.
- The Reduce task writes its output to a file of the distributed file system.

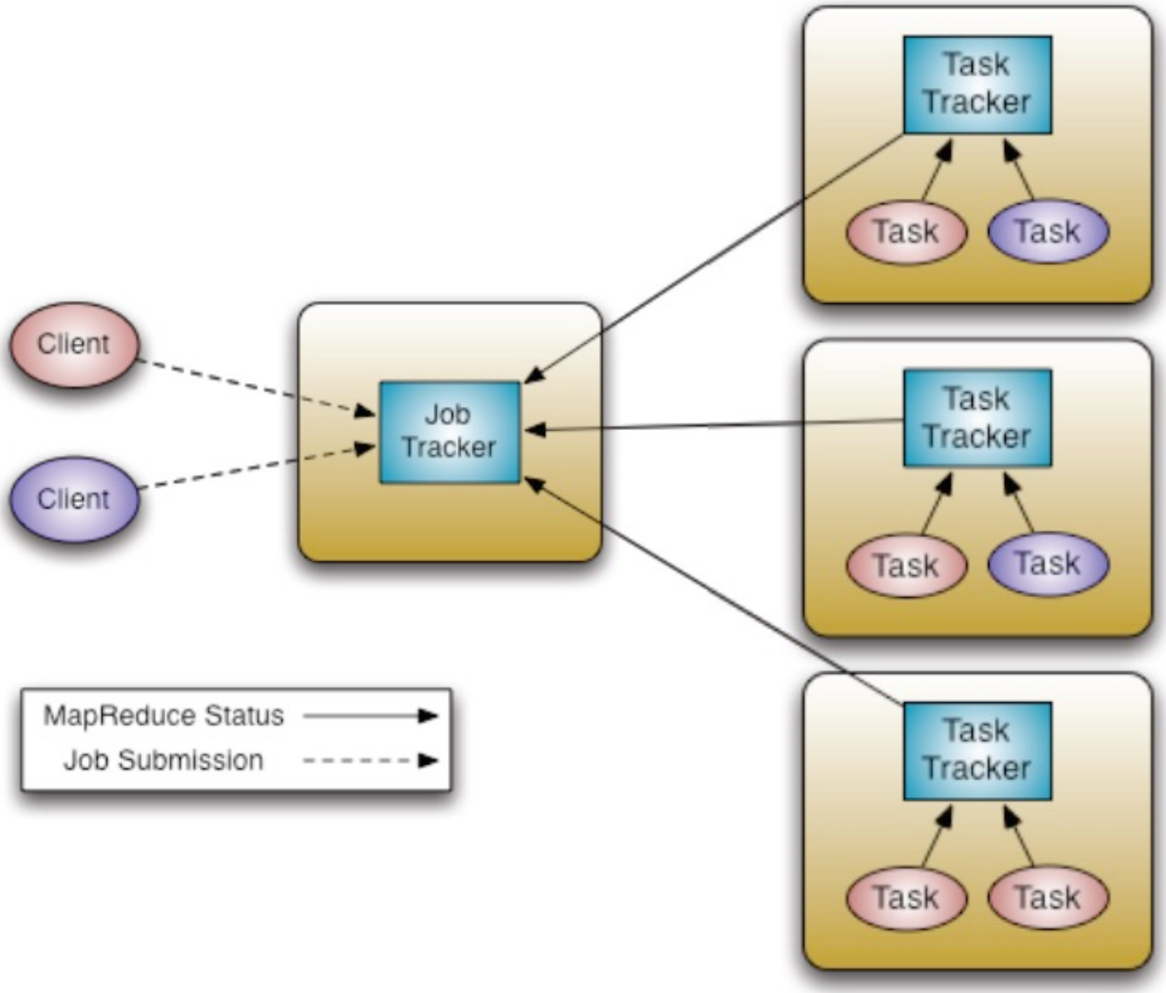
Hadoop 1.0 vs Hadoop 2.0


Single Use System
Batch Apps
Multi Purpose Platform
Batch, Interactive, Online, Streaming, …
HADOOP 1.0
MapReduce
(cluster resource management & data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS2
(redundant, reliable storage)

YARN: Taking Hadoop Beyond Batch



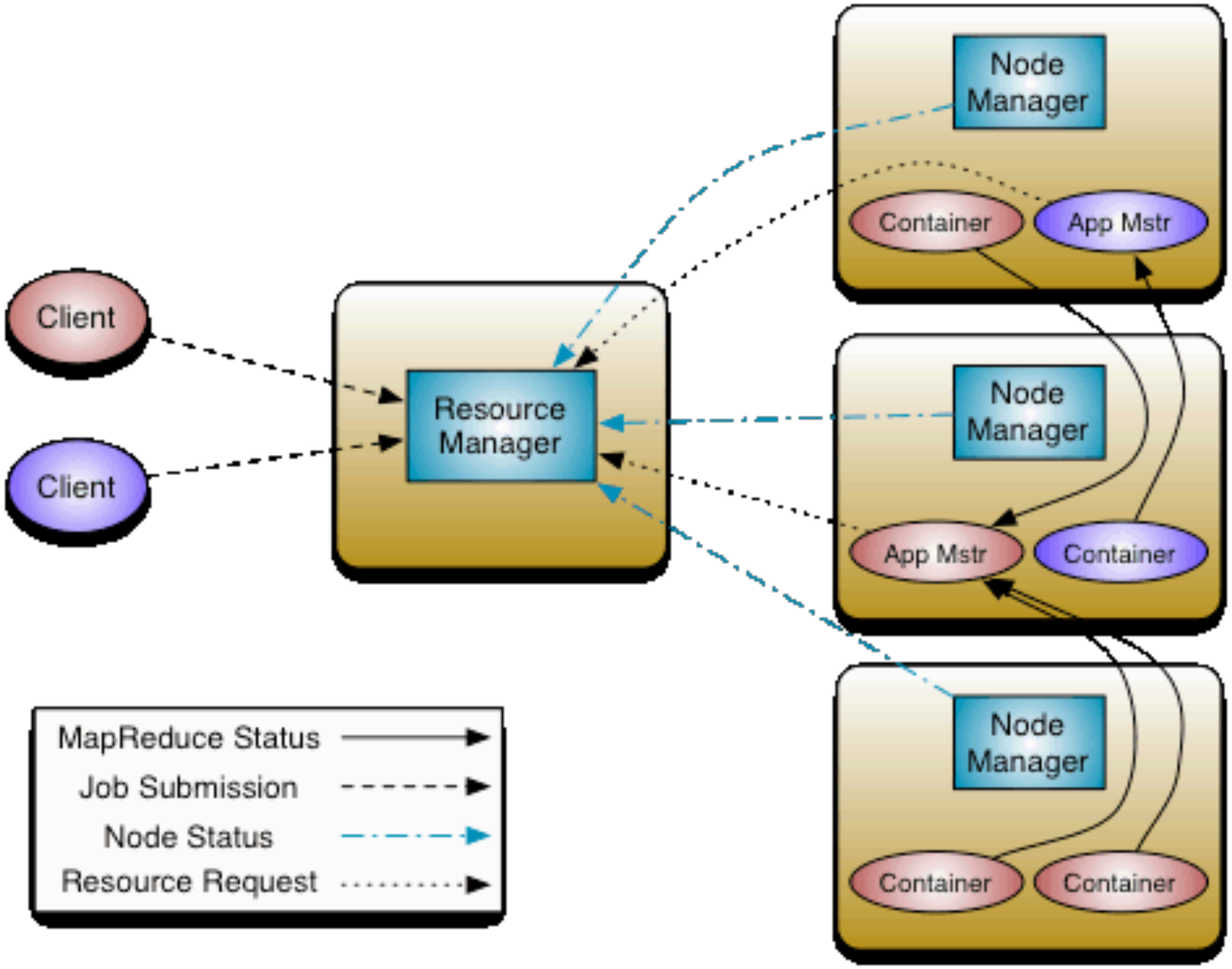
Classical Job Run Implementation (1.0)



MapReduce Classic: Limitations

- Scalability
  - Maximum Cluster size – 4,000 nodes
  - Maximum concurrent tasks – 40,000
  - Coarse synchronization in JobTracker
- Availability
  - Failure of the JobTracker kills all queued and running jobs
- Lacks support for alternate paradigms and services
  - Iterative applications implemented using MapReduce are 10x slower

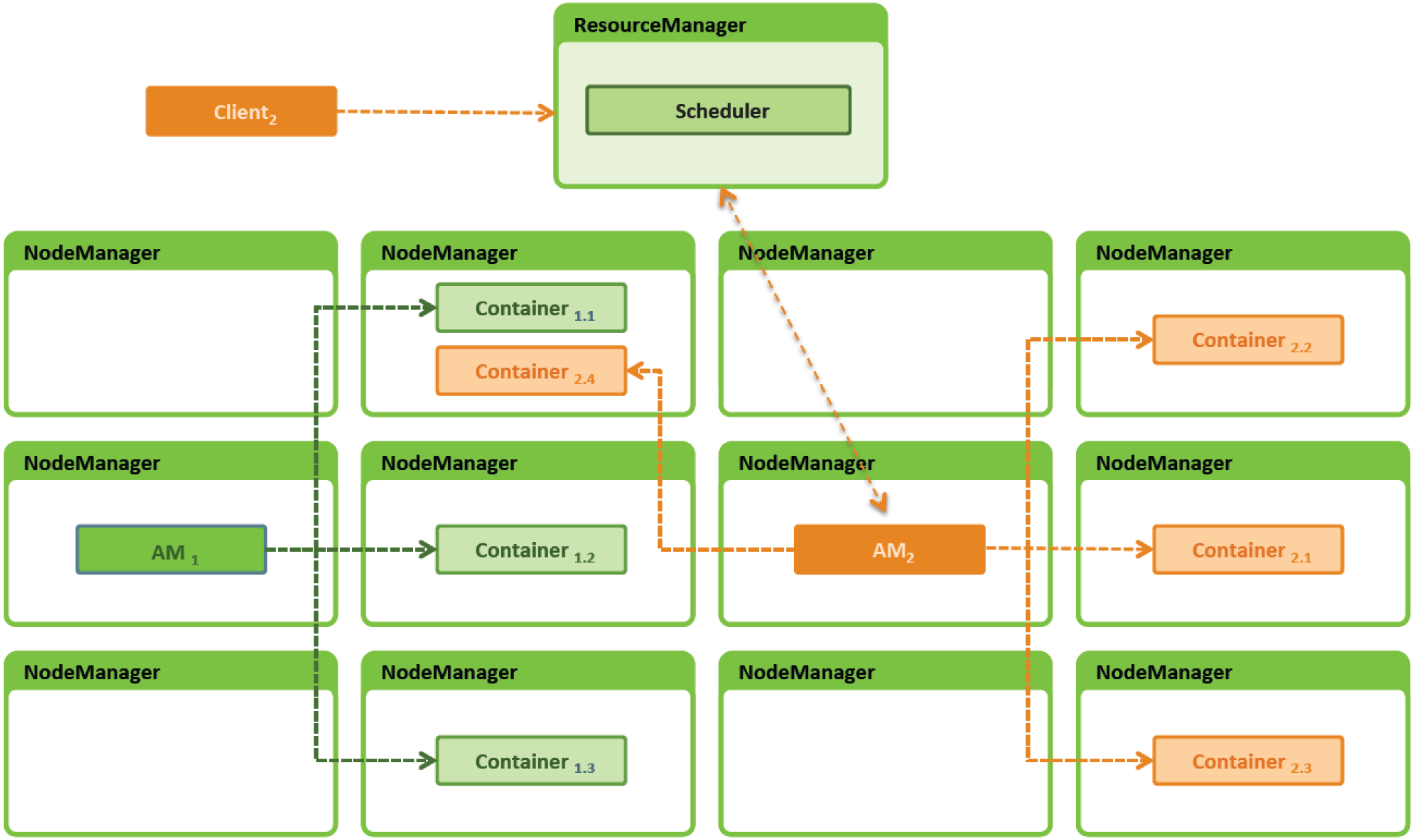
Job Run Implementation in YARN (2.0)



Job Run Implementation in YARN (2.0)

- With YARN five entities are involved:
- The **client**(s), which submit(s) the **job**
- The **ResourceManager**, which coordinates the job run; it has two main components:
  - The **ApplicationsManager**, which accepts job-submissions, negotiates the first node for executing the application specific ApplicationMaster and restarts elsewhere the ApplicationMaster on failure;
  - The **Scheduler**, which is responsible for allocating resources to the various running applications according to the requirements of the applications.
- The **NodeManager**(s), which coordinate(s) the single node (one NM for each node)
- The **ApplicationMaster**(s), which negotiate(s) resources from the ResourceManager and work(s) with the NodeManager(s) to execute and monitor the tasks (one AM for each application)

Job run in practice



Design Patterns for MapReduce

- Let us recap the MapReduce framework
- Programmers specify two functions:

  map (k1, v1) → [(k2, v2)]

  reduce (k2, [v2]) → [(k3, v3)]

  - All values with the same key are reduced together
- Programmers can also specify:

  combine (k2, [v2]) → [(k3, v3)]

    - Mini-reducers that run after the map phase
    - Used as an optimization to reduce network traffic

  partition (k2, number of partitions) → partition for k2

    - Divides up key space for parallel reduce operations
- The execution framework handles everything else
- Let's see some examples/design patterns

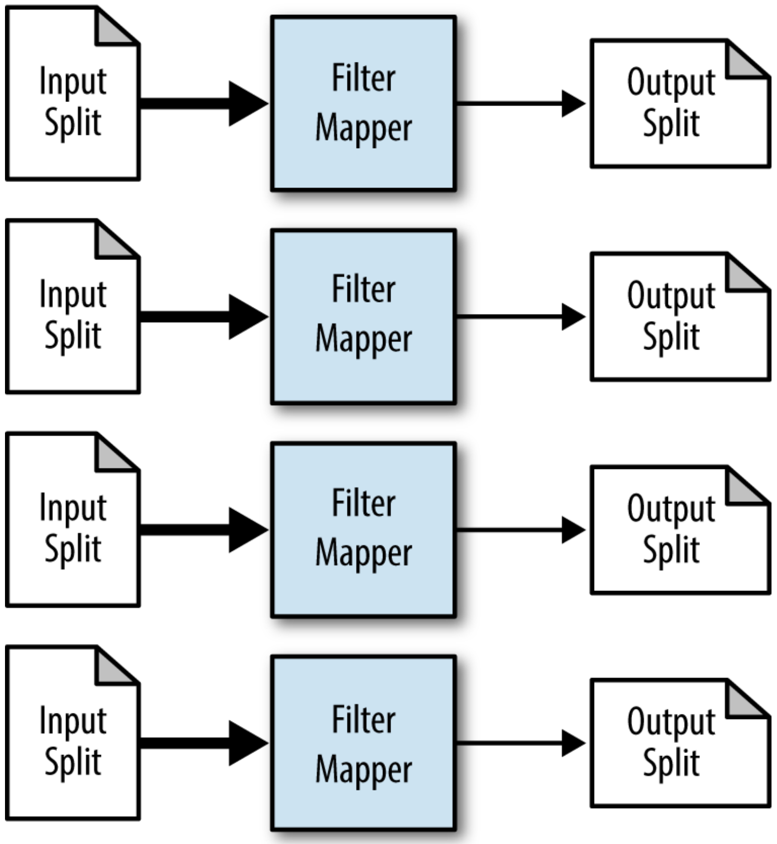
Design Patterns for MapReduce

- MapReduce is a framework not a tool
  - You have to fit your solution into the framework of map and reduce
  - It might be challenging in some situations.
- Need to take the algorithm and break it into filter/aggregate steps
  - Filter becomes part of the map function
  - Aggregate becomes part of the reduce function
- Sometimes we may need multiple MapReduce stages
- MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel!
- It makes sense only when:
  - files are very large and are rarely updated
  - We need to iterate over all the files to generate some interesting property of the data in those files
- Let's see some examples/design patterns

Filtering patterns

- Goal: Find lines/files/tuples with a particular characteristic
- Examples:
  - grep Web logs for requests to *dia.uniroma3.it/*
  - find in the Web logs the hostnames accessed by 192.168.127.1
  - locate all the files that contain the words ‘Apple’ and ‘Jobs’
- Map: filters and so it does (most of) the work
- Reduce: may simply be the identity

Structure of the filter pattern

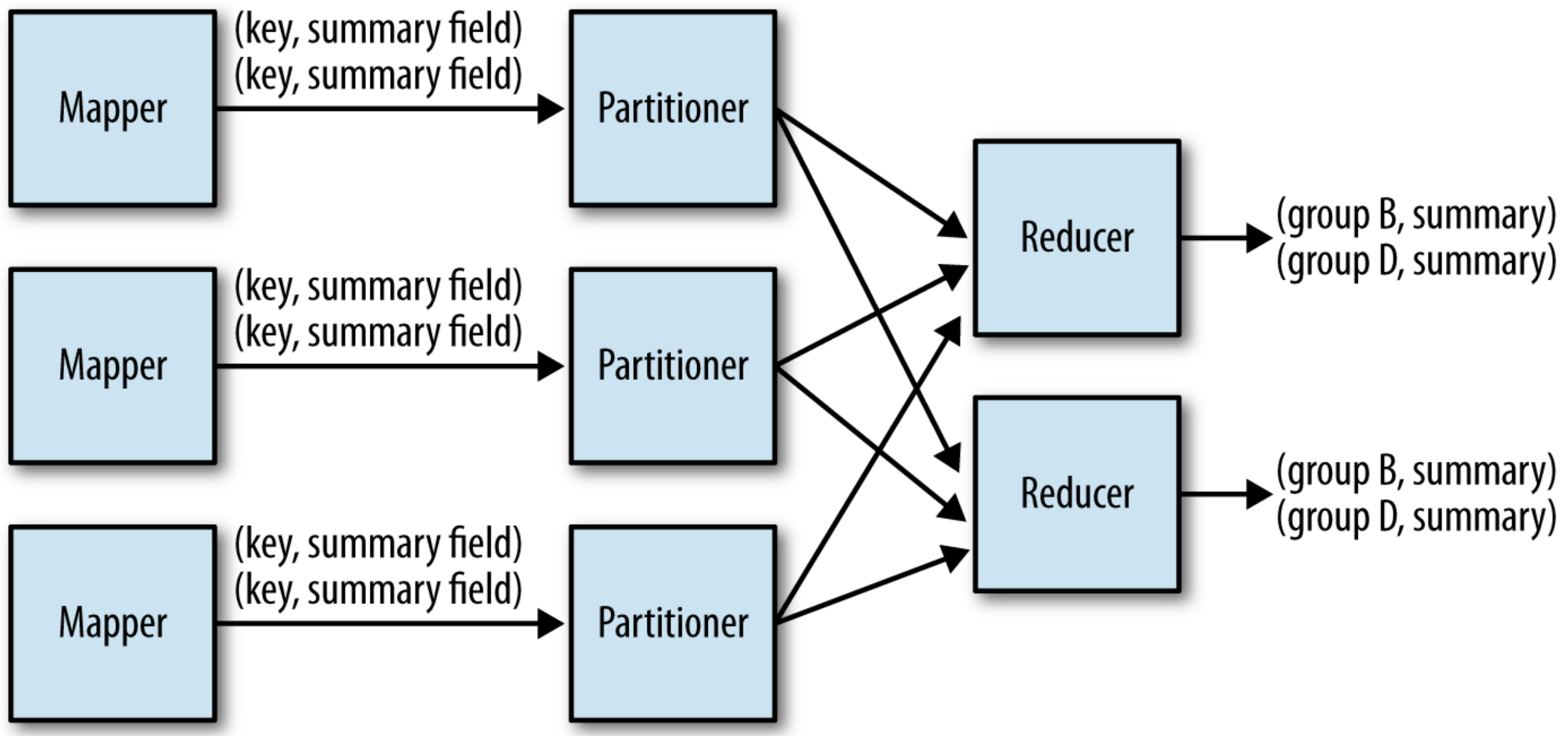


```
map(key, record):
    if we want to keep record then
        emit key,value
```

Summarization Patterns

- Goal: Compute the maximum, the sum, the average, …, over a set of values
- Examples:
  - Count the number of requests to *.dia.uniroma3.it/*
  - Find the most popular domain
  - Average the number of requests per page per Web site
- Map: filters the input and emits (k, v), where v is a summary value
  - It may be simple or the identity
- Reduce: takes (k, [v]) and applies the summarization to [v]

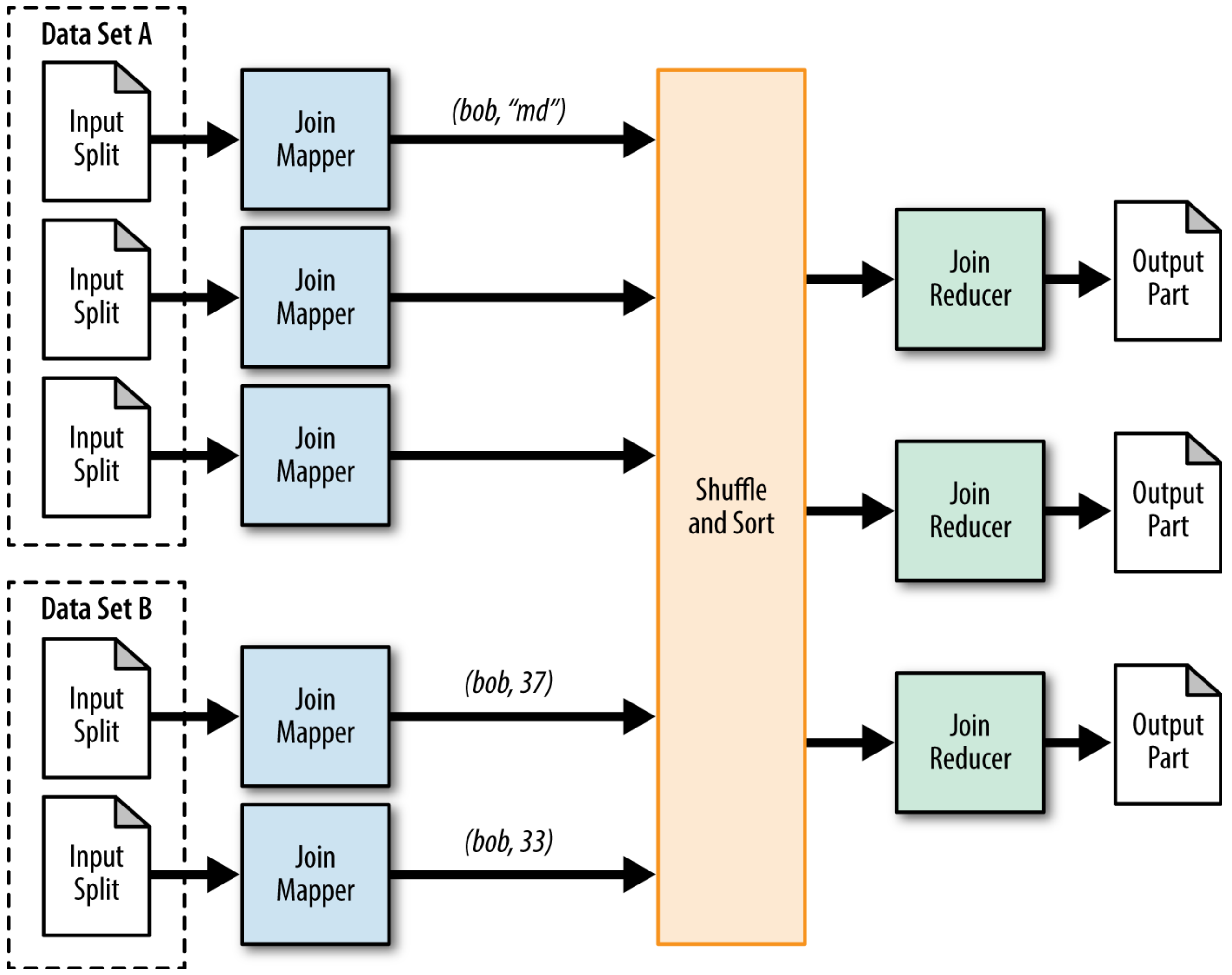
Structure of the summarizations pattern



Join Patterns

- Goal: Combine multiple inputs according to some shared values
  - Values can be equal, or meet a certain predicate
- Examples:
  - Find all documents with the words "big" and/or "data" given an inverted index
  - Find all professors and students in common courses and return the pairs <professor,student> for those cases
- Map: generates a pair (k,e) for each element e in the input where k is the value to share
- Reduce: generates a sequence of pairs [(k1,r1),..,(kn,rn)] for each k,[e1,..,ek] in the input

Structure of the join pattern



Example: Relational Union and Intersection

- Union
  - Map: Turn each input tuple t of R into a key-value pair (t, R) and turn each input tuple t' of S into a key-value pair (t', S).
  - Reduce: Associated with each key t there will be one or two values. Produce output (t, t) in either case.

- Intersection
  - Map: Turn each input tuple t of R into a key-value pair (t, R) and turn each input tuple t' of S into a key-value pair (t', S).
  - Reduce: Associated with each key t there will be either one or two values. Produce output (t, t) only if there are two values.

Example: Relational Join

- R(AB) JOIN S(BC)
- Map: For each tuple (a, b) of R, produce the pair (b, (R, a)). For each tuple (b, c) of S, produce the pair (b, (S, c)).
- Reduce: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). For each pair (R, a) and (S, c) in the list generate a pair (k,(a, b, c)).

- Example:
  - R(AB)={(a,b),(c,b)} JOIN S(BC)={(b,e),(f,g)}
  - Map: (b,(R,a)),(b,(R,c)),(b,(S,e)),(f,(S,g))
  - Shuffle and sort: (b,[(R,a),(R,c),(S,e)]) (f,[(S,g)])
  - Reduce: (k1,(a,b,e)),(k2,(c,b,e))

Sorting Pattern

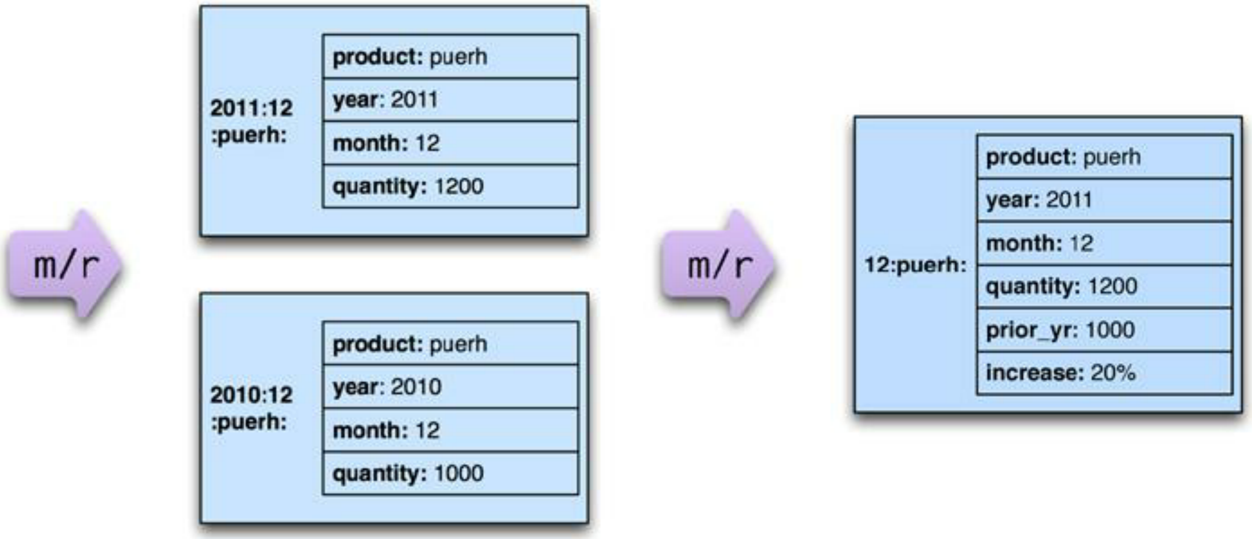
- Goal: Sort input
- Examples:
  - Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages
- The programming model does not support this per se, but the implementations do
  - The Shuffle stage groups and orders!
- In general:
  - The Map and the Reduce might do nothing
  - If we have a single reducer, we will get sorted output
  - If we have multiple reducers, we can get partly sorted output but it's quite easy to write a last-pass file that merges all the files

Two stage MapReduce

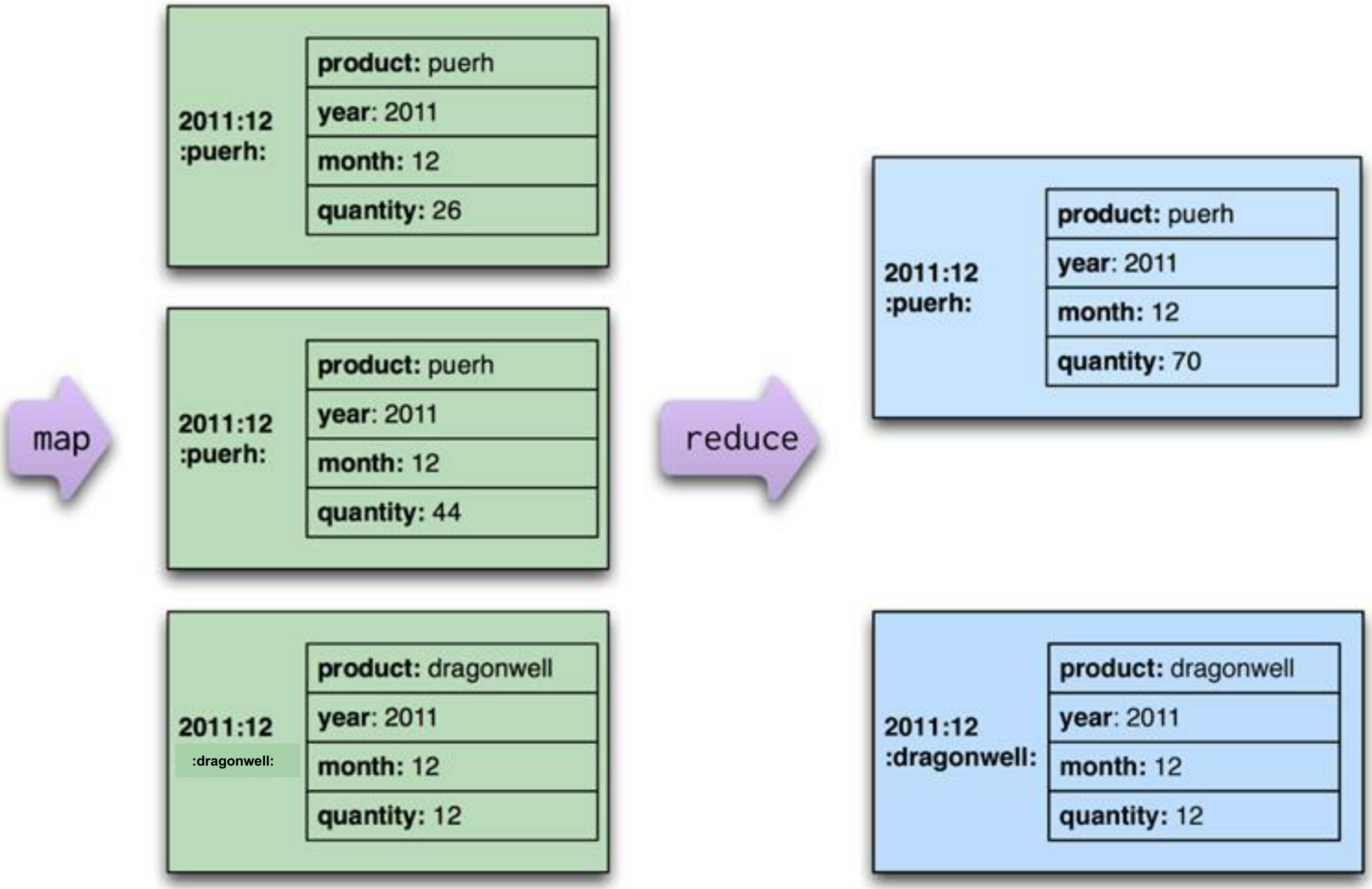
- As map-reduce calculations get more complex, it’s useful to break them down into stages, with the output of one stage serving as input to the next
- Intermediate output may be useful for different outputs too, so you can get some reuse
- The intermediate records can be saved in the data store, forming a materialized view
- Early stages of map-reduce operations often represent the heaviest amount of data access, so building and save them once as a basis for many downstream uses saves a lot of work.

Example of Two Stage MapReduce

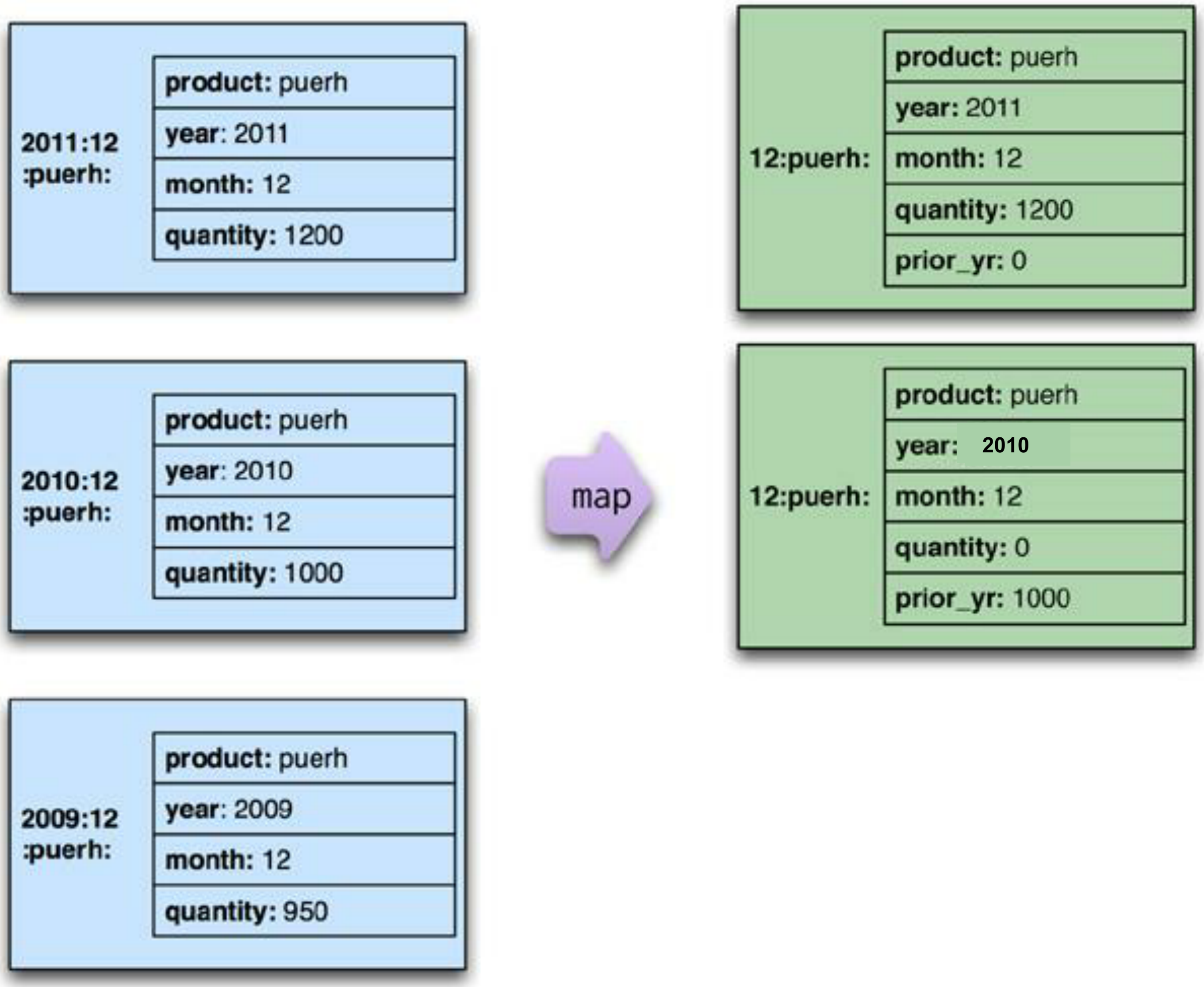
- We want to compare the sales of products for each month in 2011 to the same month of the prior year.
- First stage: produces records showing the sales for a single product in a single month of the year.
- Second stage: produces the result for a single product by comparing one month's results with the same month in the prior year



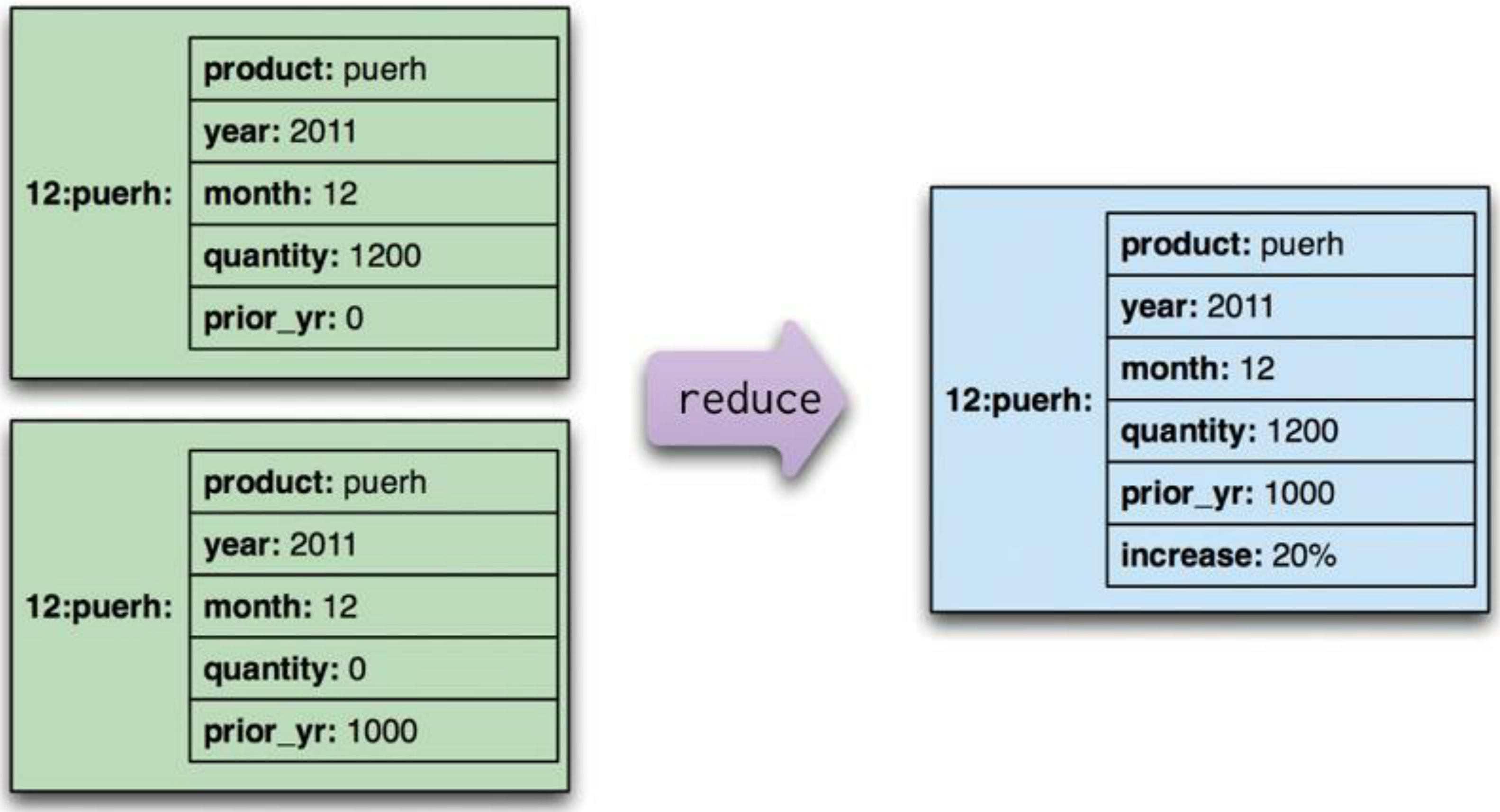
First stage


map
2011:12 :puerh:
product: puerh
year: 2011
month: 12
quantity: 26
2011:12 :puerh:
product: puerh
year: 2011
month: 12
quantity: 44
2011:12 :dragonwell:
product: dragonwell
year: 2011
month: 12
quantity: 12
reduce
2011:12 :puerh:
product: puerh
year: 2011
month: 12
quantity: 70
2011:12 :dragonwell:
product: dragonwell
year: 2011
month: 12
quantity: 12

Second stage mapper



Second stage reducer



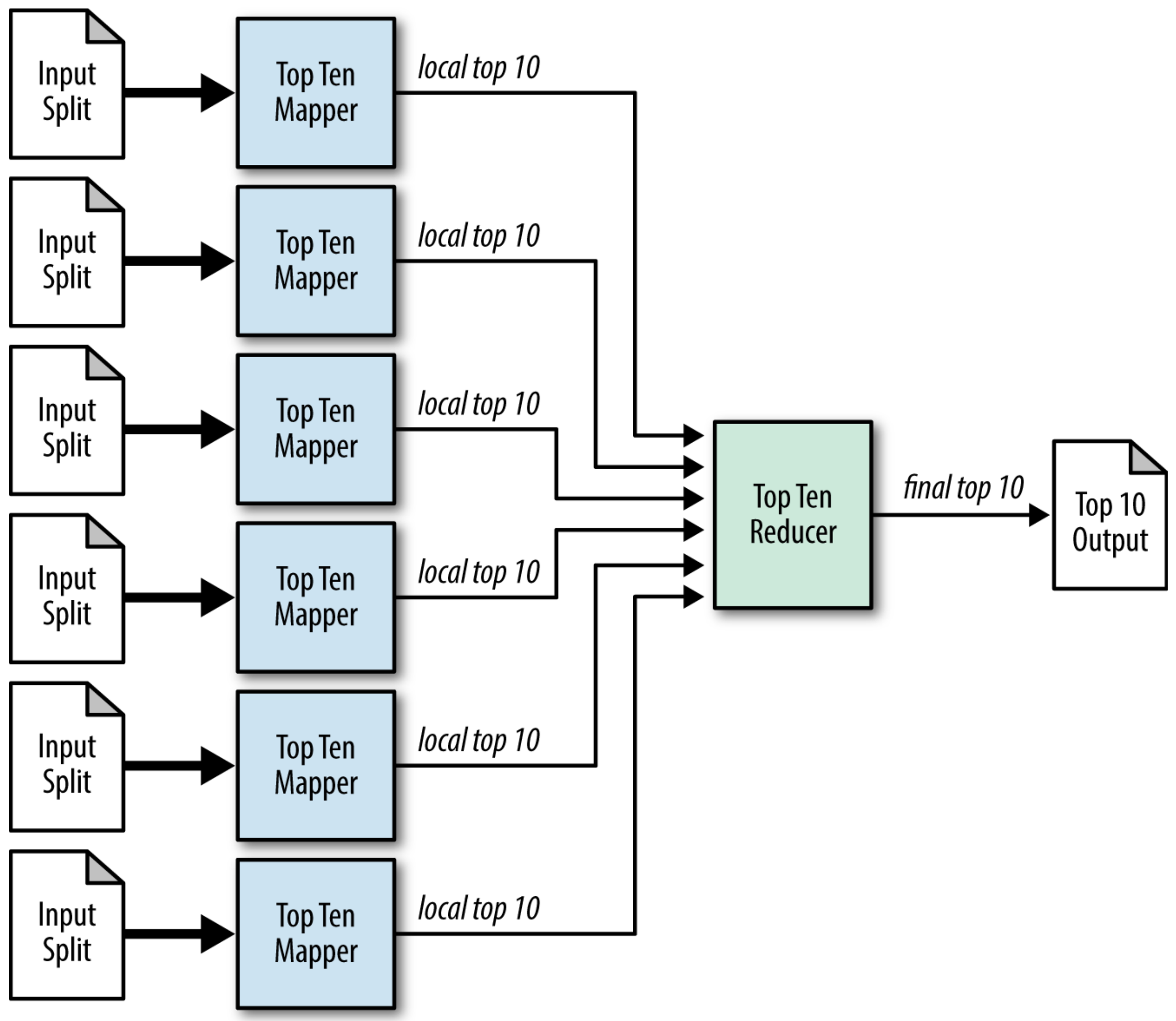
Exercise 1: Top-K

- Given a set of records

  [Name: n, Age: g, Salary, s]

  Find the top 10 employees younger than 30 with the highest salaries

Structure of the solution (with a combiner)



Solution

```
map(key, record):
  insert record into a top ten sorted list
  if length of list is greater-than 10 then
    truncate list to a length of 10
    emit (key, record with the values of the list)

reduce(key, record):
  sort record
  truncate record to top 10
  emit record
```

Exercise 2: Inverted index

- Given a set of Web pages

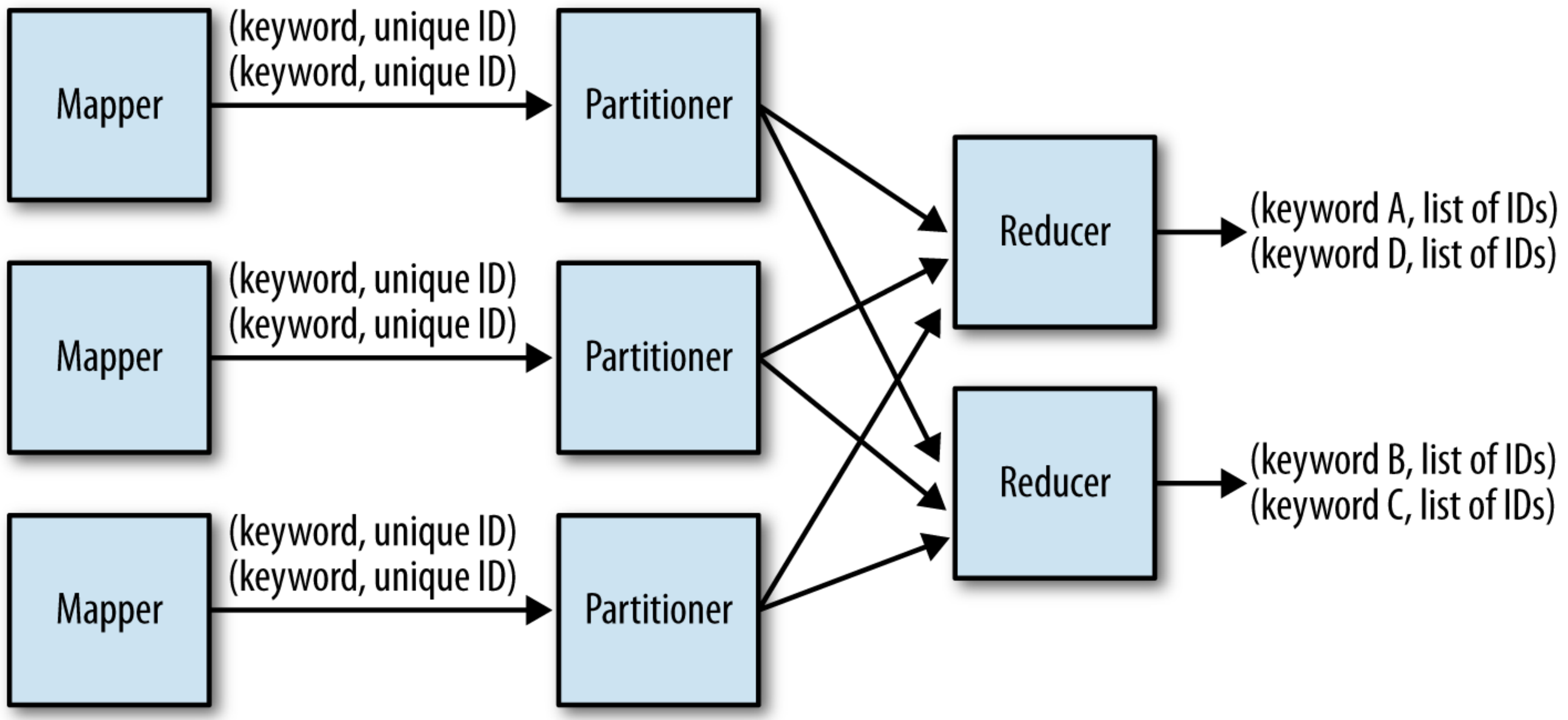
  [URL: ID, Content: page]

  build an inverted index

  [(k1,[u1,…uk]),….,(kn,[u1,…,ul])]

  where k is a keyword and u1,…uk are the URLs of the Web pages containing the keyword k

Structure of the solution



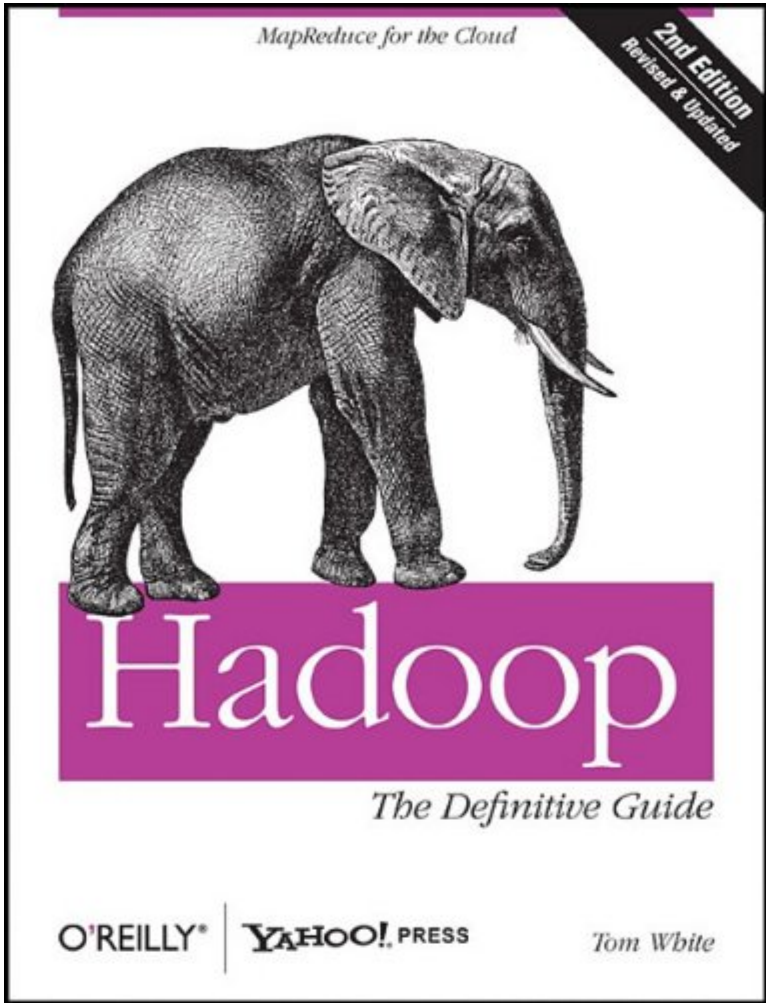
Exercise 3: Average

- Given a set of records

  [Name: n, Age: a, Salary, s]

  Find the min, the max, and the average salary of employees of the same age

References


MapReduce for the Cloud
2nd Edition
Revised & Updated
Hadoop
The Definitive Guide
O'REILLY
YAHOO! PRESS
Tom White

Building Effective Algorithms and Analytics for Hadoop and Other Systems
MapReduce Design Patterns
O'REILLY
Donald Miner & Adam Shook