CS 422 Data Mining

Lecture 6

September 27, 2018

- The midterm is on October 11, 2018 6:25 PM

❑ Introduction to MapReduce Framework

Big Data

- Mining of large datasets
  - Web mining
  - Big data

What is Web Mining?

- Discovering useful information from the World-Wide Web and its usage patterns

Web Mining v. Data Mining

- Structure (or lack of it)
  - Textual information and linkage structure
- Scale
  - Data generated per day is comparable to largest conventional data warehouses
- Speed
  - Often need to react to evolving usage patterns in real-time (e.g., merchandising)

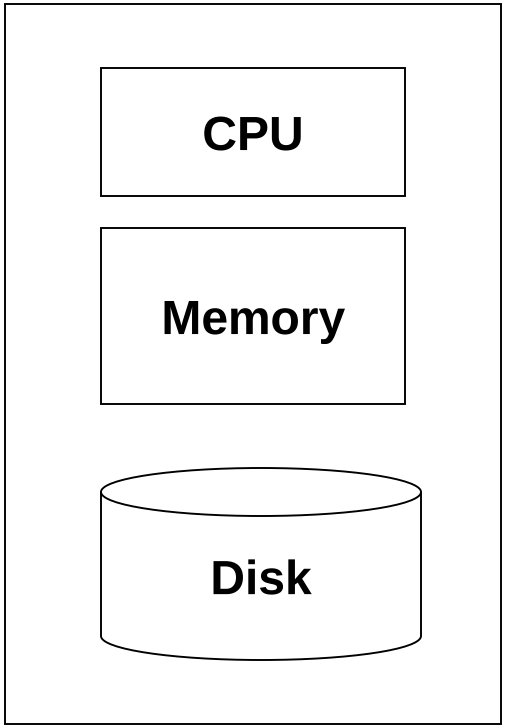
Two Approaches to Analyzing Data

- Machine Learning approach
  - Emphasizes sophisticated algorithms e.g., Decision Trees, Support Vector Machines
  - Data sets tend to be small, fit in memory
- Data Mining approach
  - Emphasizes big data sets (e.g., in the terabytes)
  - Data cannot even fit on a single disk!
  - Necessarily leads to simpler algorithms

Philosophy

- In many cases, adding more data leads to better results than improving algorithms
  - Netflix
  - Google search
  - Google ads

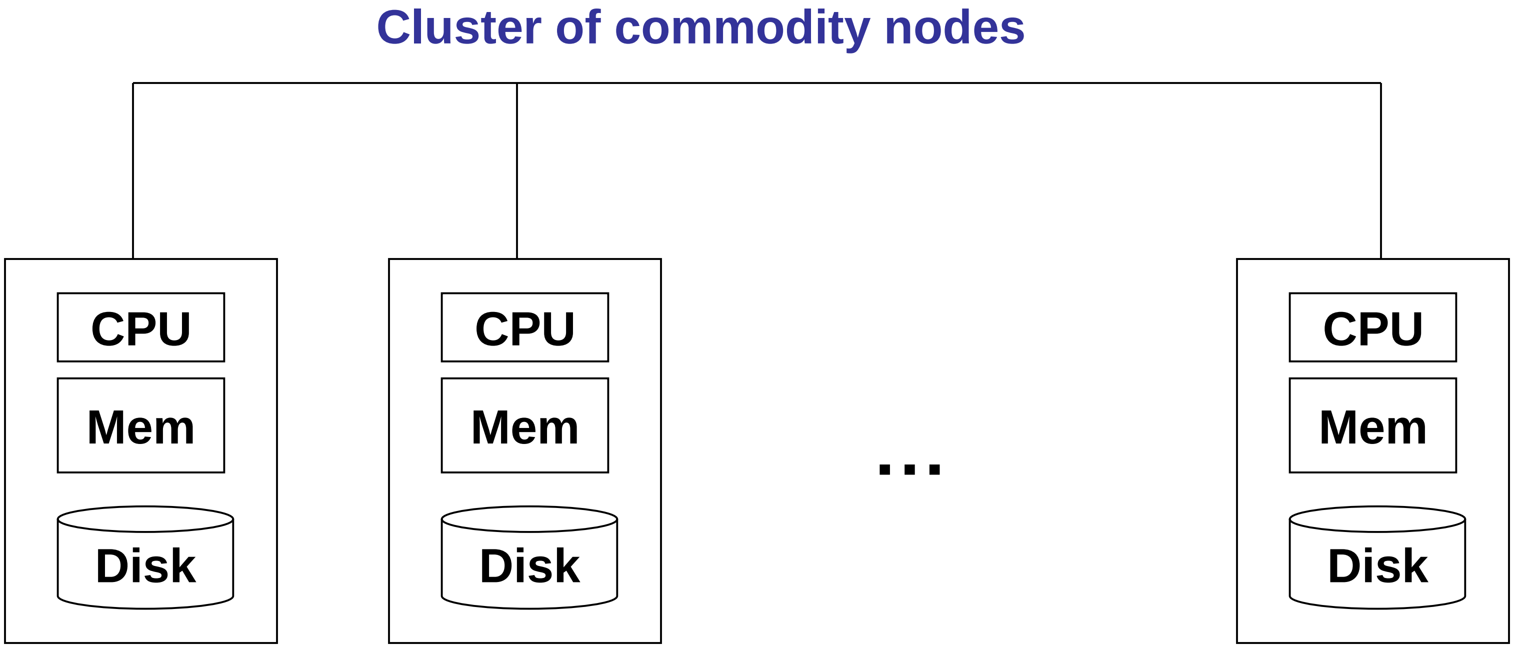
Systems architecture



Machine Learning, Statistics

“Classical” Data Mining

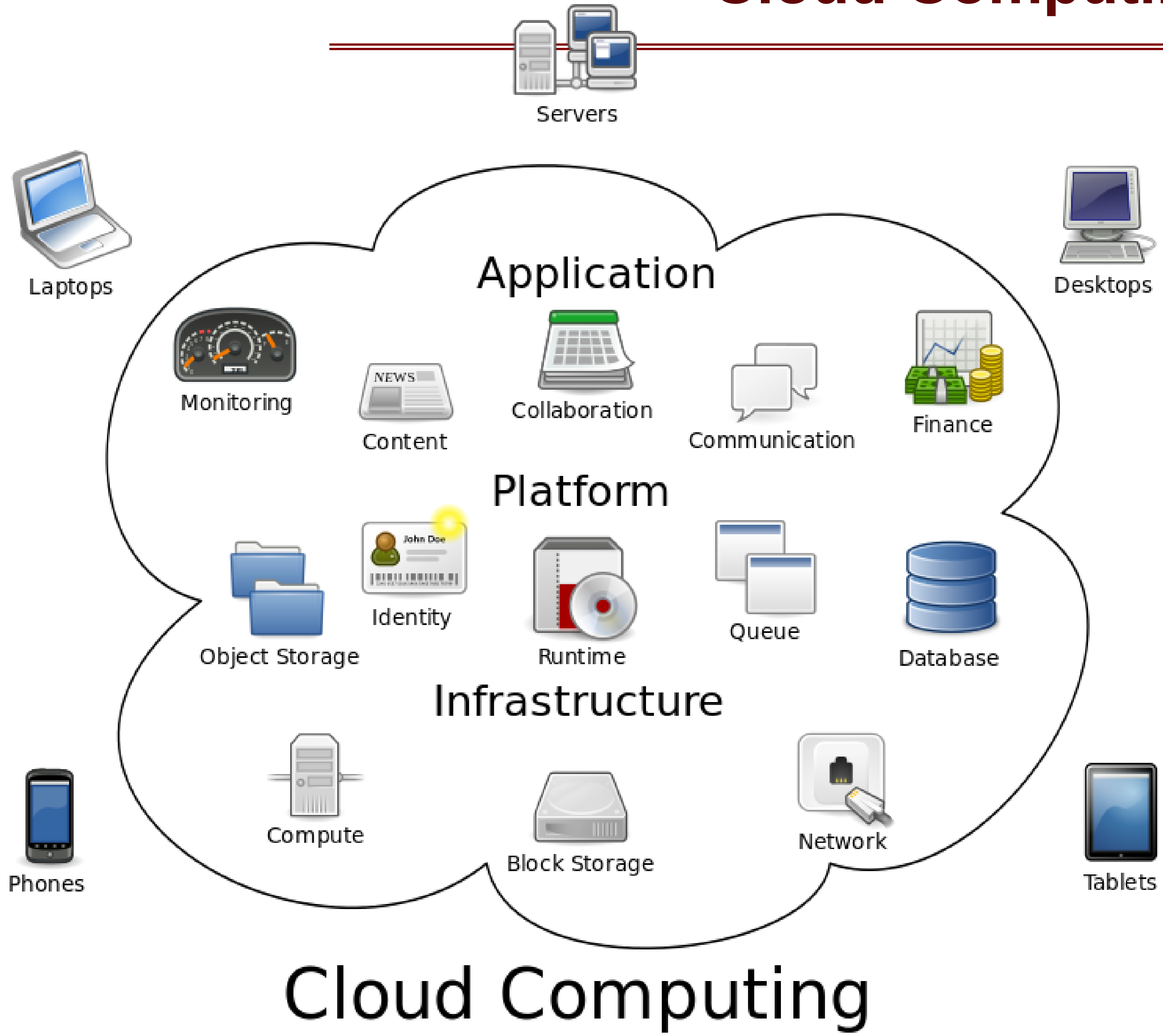
Very Large-Scale Data Mining



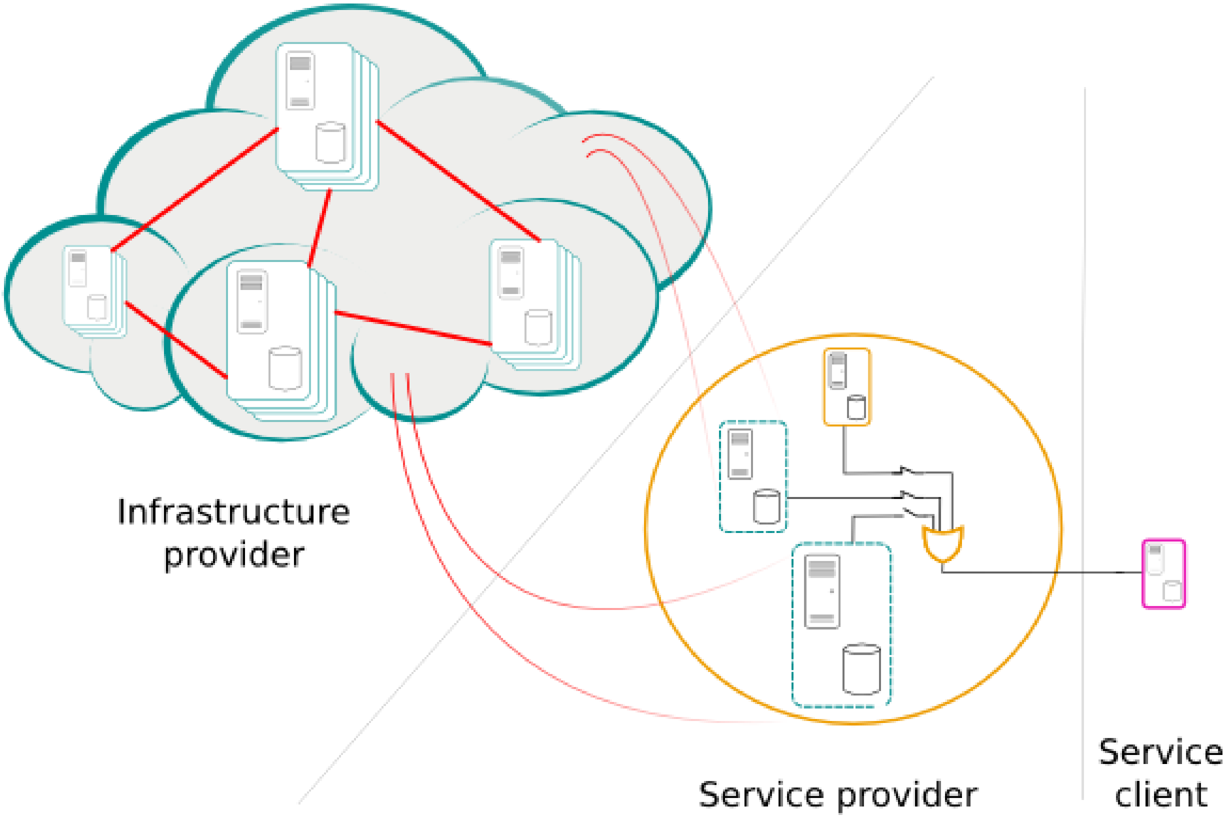
Systems Issues

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server!
  - Need large farms of servers
- How to organize hardware/software to mine multi-terabye data sets
  - Without breaking the bank!

Cloud Computing



Cloud Computing



Resources

- Infrastructure
  - Amazon AWS
    - Support for MapReduce
  - IBM SmartCloud
  - Rackspace OpenStack

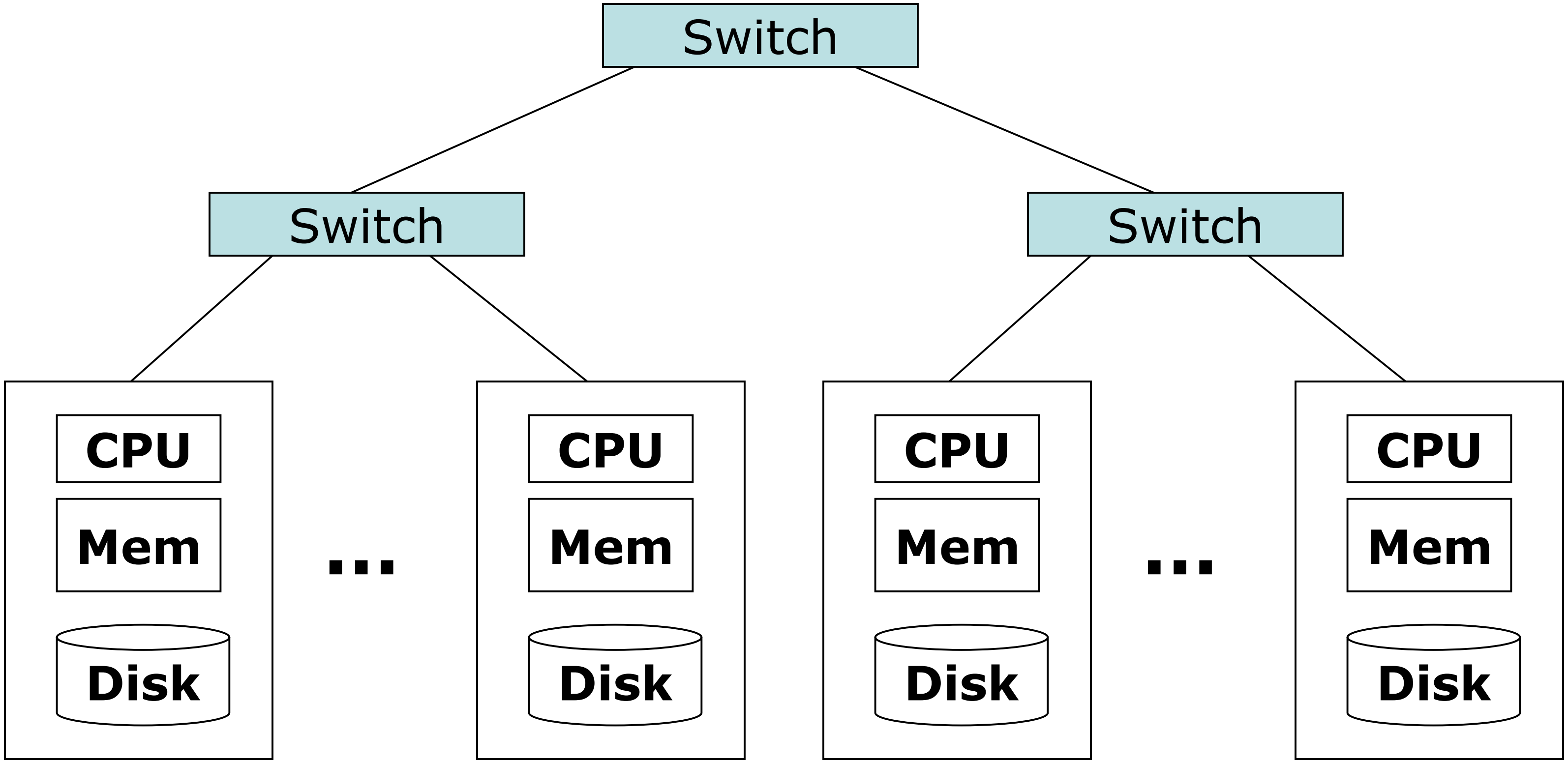
Large Scale Computation

- Distributed computation
- Server clusters
- Failures not only possilble but very likely

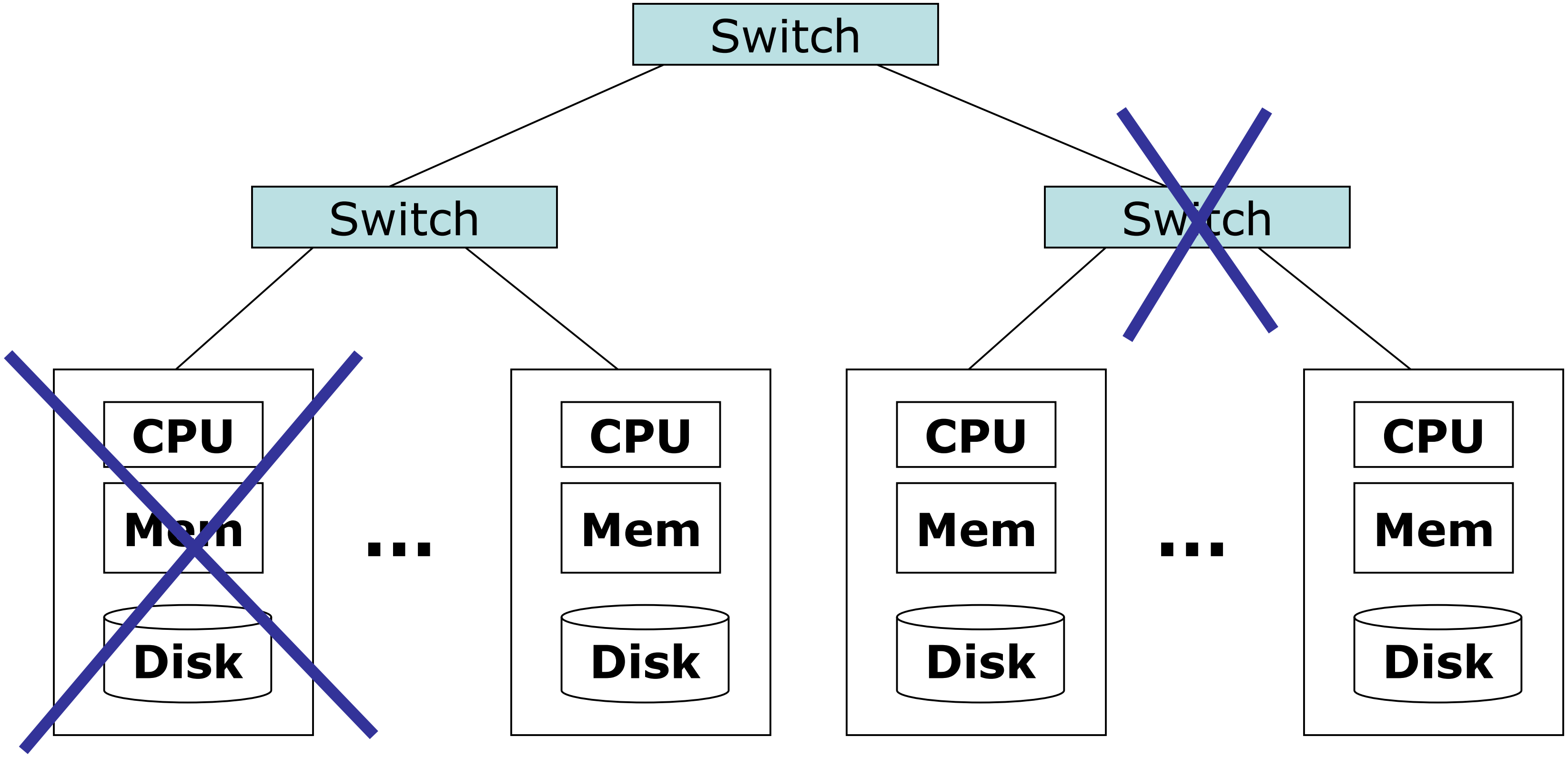
Commodity Clusters

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect

Cluster Architecture



Cluster Architecture

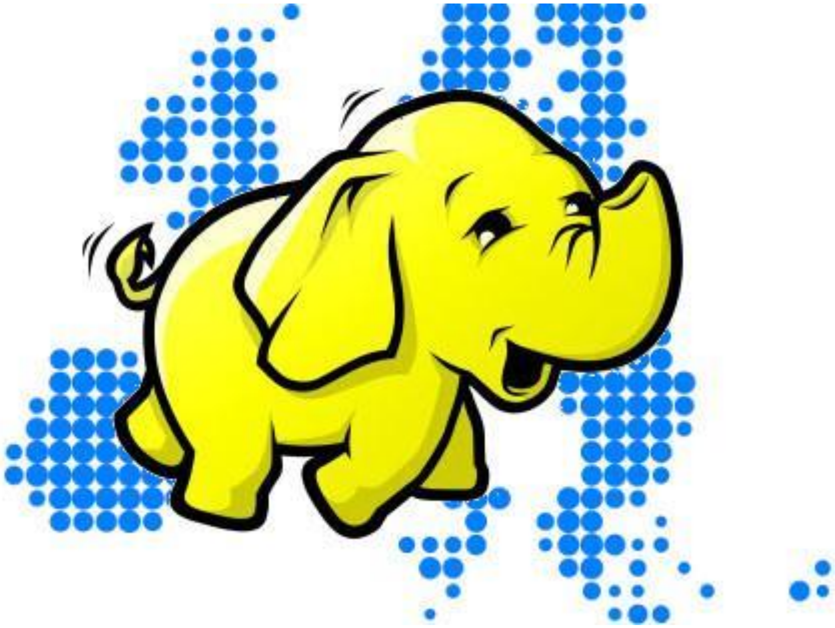


Commodity Clusters

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- How to organize computations on this architecture?
  - Mask issues such as hardware failure

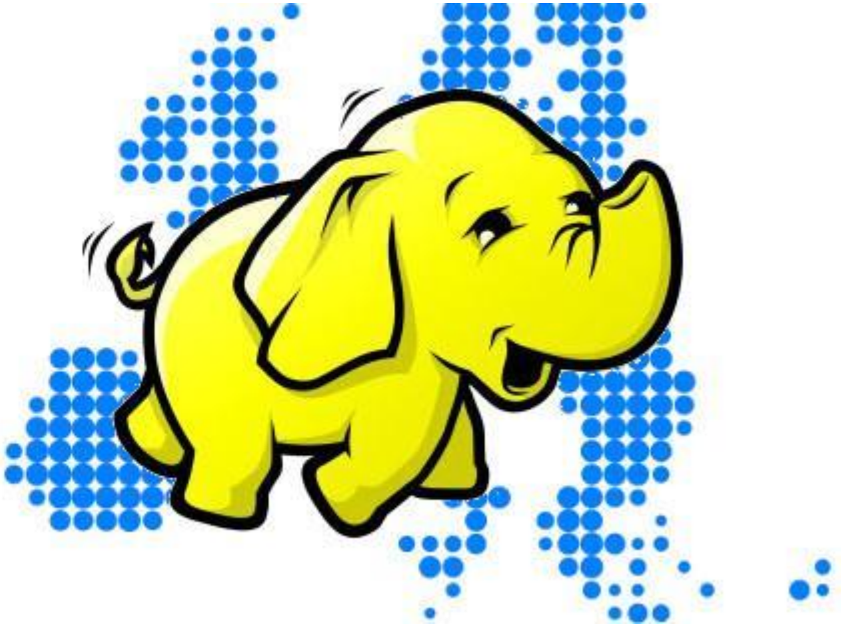
Solution?

- Enter Hadoop!



Solution?

- Enter Hadoop!



- coreservlets.com, Dima May

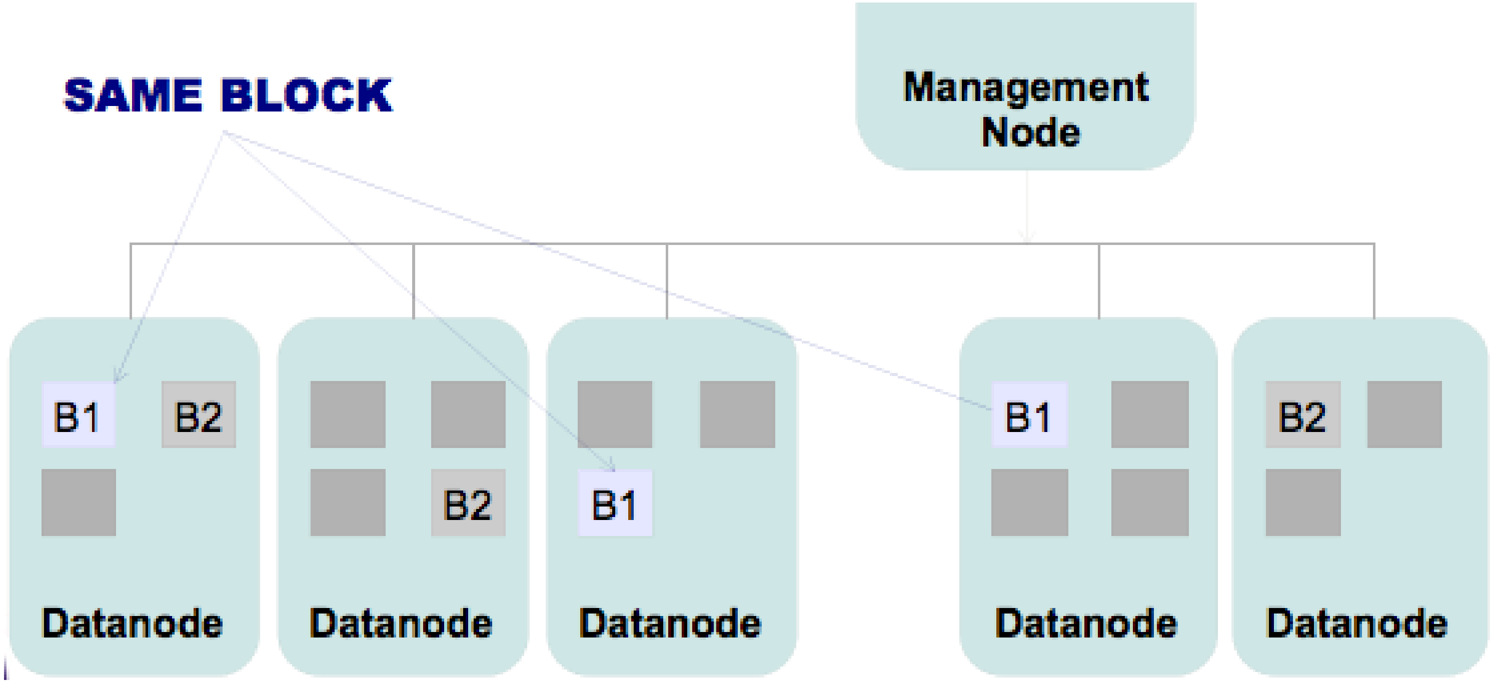
Hadoop Distributed File System

- Appears as a single disk
  - Runs on top of a native filesystem
  - Ext3,Ext4,XFS
- Based on Google's Filesystem GFS
- Fault Tolerant
  - Can handle disk crashes, machine crashes, etc

HDFS Main Idea

- Files are split into blocks (single unit of storage)
- Replicated across machines at load time

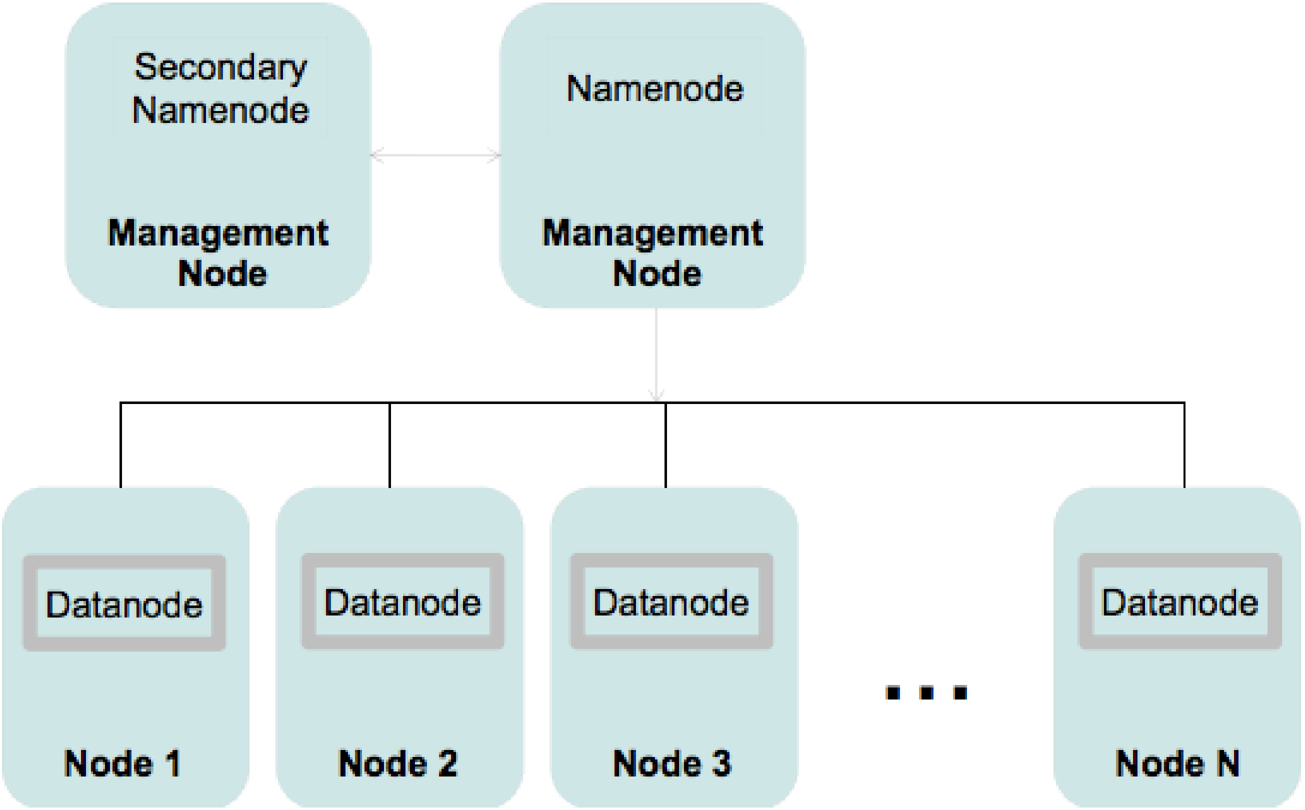
Files and Blocks



HDFS Nodes

- Name node
  - manages the File System's namespace/meta-data/file blocks
  - Runs on 1 machine to several machines
- Data node
  - Stores and retrieves data blocks
  - Reports to Namenode
  - Runs on many machines
- Secondary name node
  - Performs house keeping work so Namenode doesn’t have to
  - Requires similar hardware as Namenode machine
  - Not used for high-availability – not a backup for name node

HDFS Nodes


Secondary Namenode
Management Node
Namenode
Management Node
Datanode
Node 1
Datanode
Node 2
Datanode
Node 3
...
Datanode
Node N

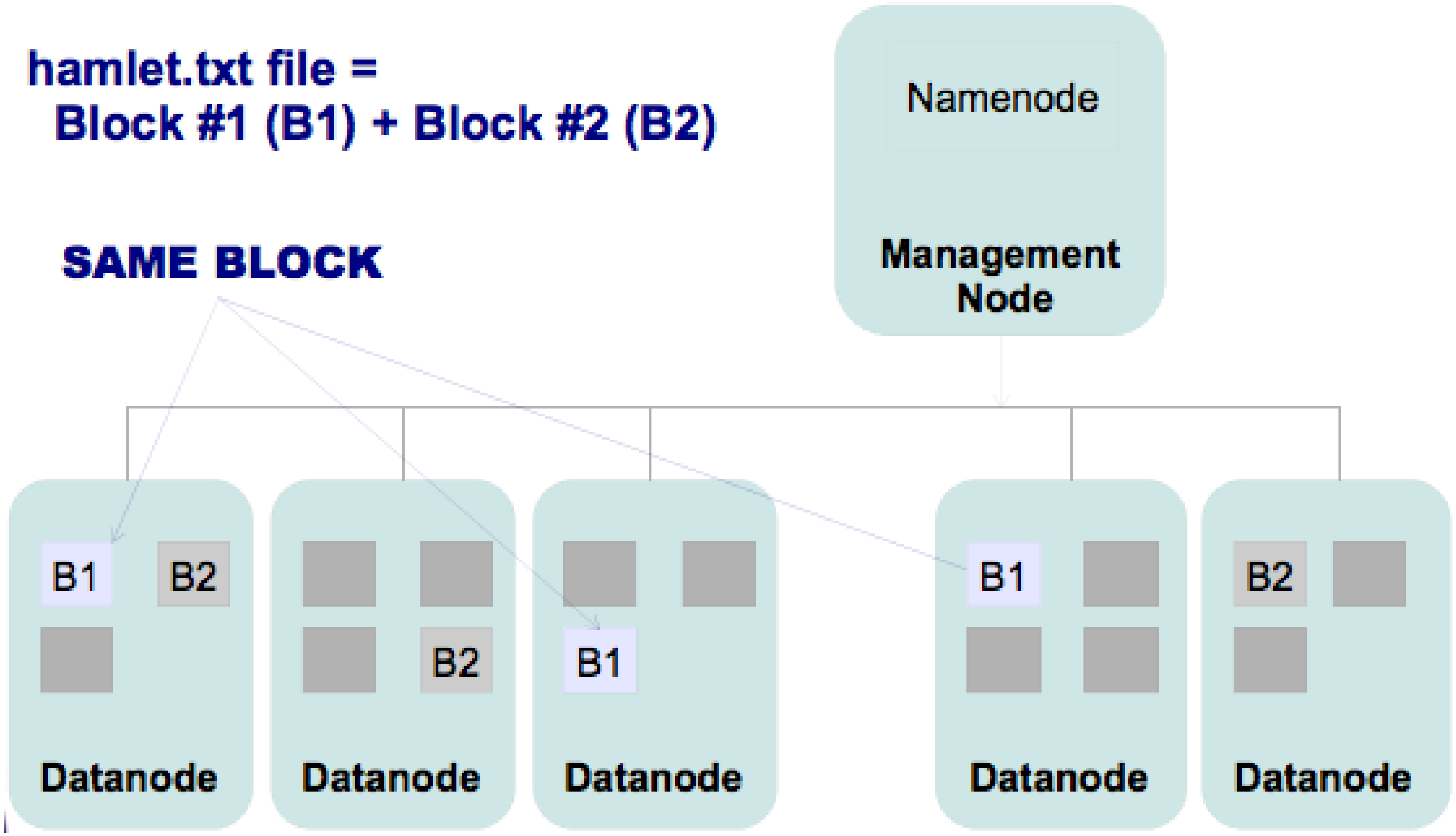
Files and Blocks

- Files are split into blocks (single unit of storage)
  - Managed by Namenode, stored by Datanode
  - Transparent to user
- Replicated across machines at load time
  - Same block is stored on multiple machines
  - Good for fault-tolerance and access
  - Default replication is 3

Files and Blocks

hamlet.txt file =
Block #1 (B1) + Block #2 (B2)

SAME BLOCK



HDFS Blocks

- Blocks are traditionally either 64MB or 128MB
  - Default is 64MB
- The motivation is to minimize the cost of seeks as compared to transfer rate
  - 'Time to transfer' > 'Time to seek'
- For example, lets say
  - seek time = 10ms
  - Transfer rate = 100 MB/s
- To achieve seek time of 1% transfer rate
  - Block size will need to be = 100MB

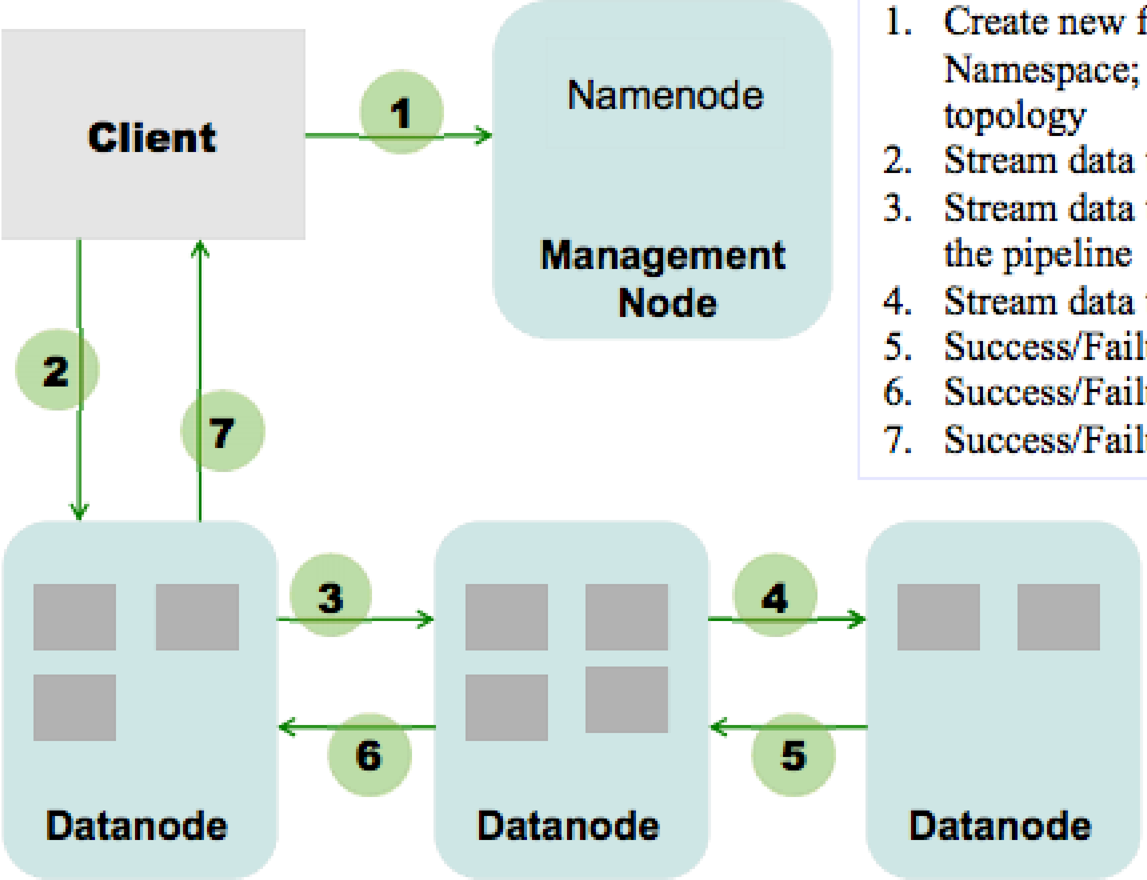
Block Replication

- Namenode determines replica placement
  - Replica placements are rack aware
  - Balance between reliability and performance
- Attempts to reduce bandwidth
- Attempts to improve reliability by putting replicas on multiple racks
  - Default replication is 3
- 1st replica on the local rack
- 2nd replica on the local rack but different machine
- 3rd replica on the different rack
- This policy may change/improve in the future

Client Interface

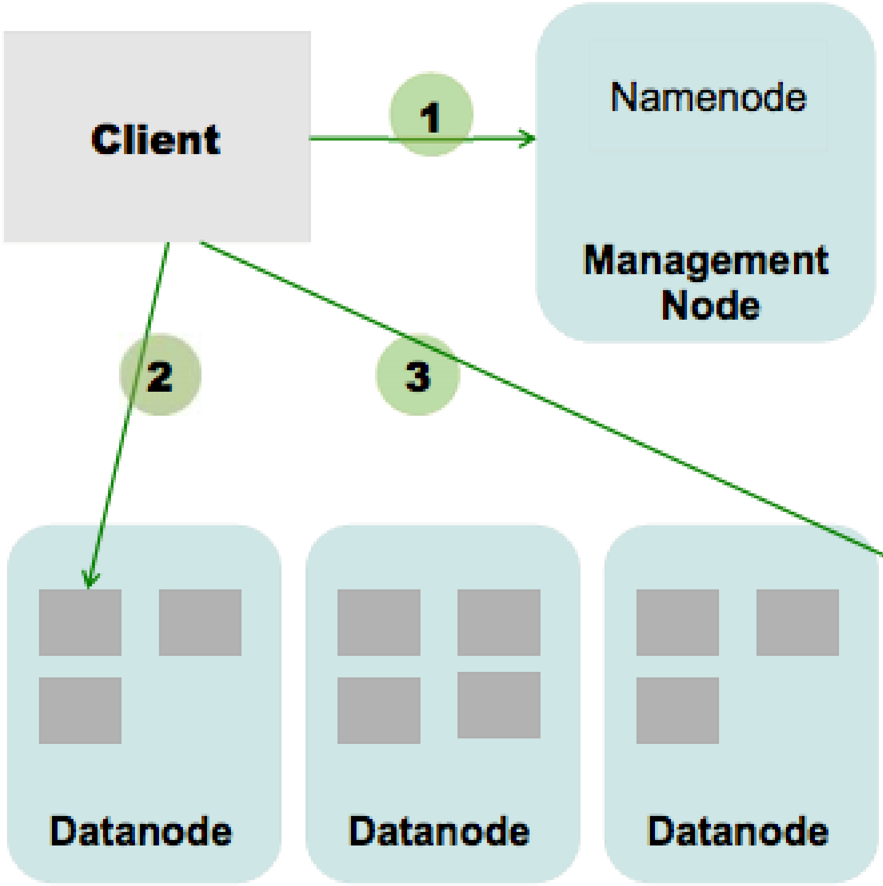
- Namenode does NOT directly write or read data
  - One of the reasons for HDFS's Scalability
- Client interacts with Namenode to update Namenode's HDFS namespace and retrieve block locations for writing and reading
- Client interacts directly with Datanode to read/write data

HDFS File Write



1. Create new file in the Namenode's Namespace; calculate block topology
2. Stream data to the first Node
3. Stream data to the second node in the pipeline
4. Stream data to the third node
5. Success/Failure acknowledgment
6. Success/Failure acknowledgment
7. Success/Failure acknowledgment

HDFS File Reads



1. Retrieve Block Locations
2. Read blocks to re-assemble the file
3. Read blocks to re-assemble the file

Name Node Memory

- For fast access Namenode keeps all block metadata in-memory
  - The bigger the cluster - the more RAM required
- Best for millions of large files (100mb or more) rather than billions
  - Will work well for clusters of 100s machines
  - Hadoop 2+
  - Namenode Federations
- Each namenode will host part of the blocks
  - Horizontally scale the Namenode
  - Support for 1000+ machine clusters
- Yahoo! runs 50,000+ machines

Name Nodes Fault Tolerance

- Namenode daemon process must be running at all times
  - If process crashes then cluster is down
- Namenode is a single point of failure
  - Host on a machine with reliable hardware (ex. sustain a diskfailure)
  - Usually is not an issue
- Hadoop 2+
  - High Availability Namenode
- Active Standby is always running and takes over in case main namenode fails
- Still in its infancy

HDFS is good for…

- Storing large files
  - Terabytes, Petabytes, etc...
  - Millions rather than billions of files
  - 100MB or more per file
- Streaming data
  - Write once and read-many times patterns
  - Optimized for streaming reads rather than random reads
  - Append operation added to Hadoop 0.21
- “Cheap” Commodity Hardware
  - No need for super-computers, use less reliable commodity hardware

Hadoop EcoSystem

Hadoop

- Started as a sub-project of Apache Nutch
  - Nutch's job is to index the web and expose it for searching
  - Open Source alternative to Google
  - Started by Doug Cutting
- In 2004 Google publishes Google File System (GFS) and MapReduce framework papers
  - Doug Cutting and Nutch team implemented Google's frameworks in Nutch
  - In 2006 Yahoo! hired Doug Cutting to work on Hadoop with a dedicated team
  - In 2008 Hadoop became Apache Top Level Project
- http://hadoop.apache.orgCloudera

Naming Conventions

- Doug Cutting drew inspiration from his family
  - Lucene: Doug’s wife’s middle name
  - Nutch: A word for "meal" that his son used as a toddler
  - Hadoop: Yellow stuffed elephant named by his son

Hadoop Eco System

- At first Hadoop was mainly known for two core products:
  - HDFS: Hadoop Distributed FileSystem
  - MapReduce: Distributed data processing framework
- Today, in addition to HDFS and MapReduce, the term also represents a multitude of products:
  - HBase: Hadoop column database; supports batch and random reads and limited queries
  - Zookeeper: Highly-Available Coordination Service
  - Oozie: Hadoop workflow scheduler and manager
  - Pig: Data processing language and execution environment
  - Hive: Data warehouse with SQL interface

Hadoop Eco System

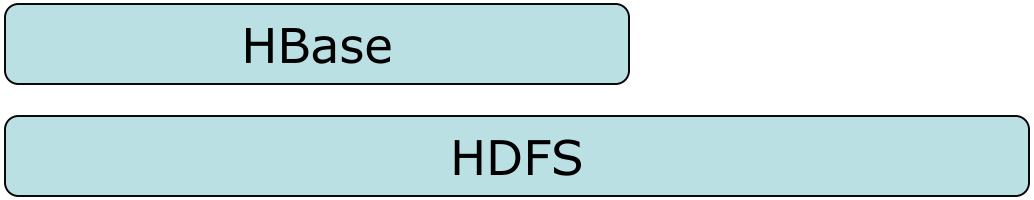
- To start building an application, you need a file system
  - In Hadoop world that would be Hadoop Distributed File System (HDFS)

Hadoop Eco System

HDFS

- To start building an application, you need a file system
  - In Hadoop world that would be Hadoop Distributed File System (HDFS)

Hadoop Eco System

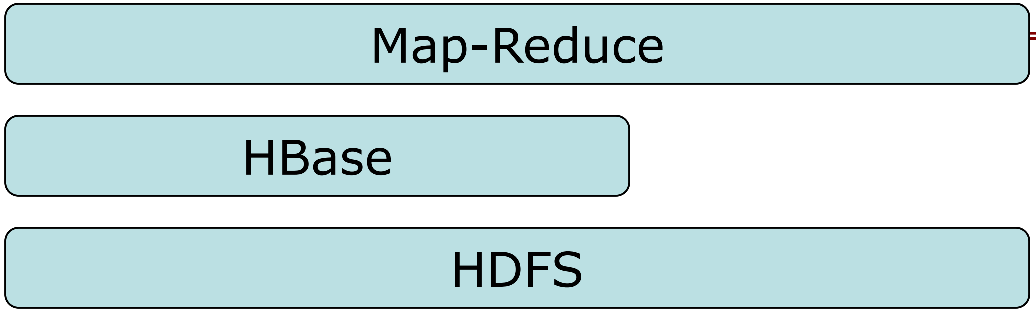


HBase

HDFS

- To start building an application, you need a file system
  - In Hadoop world that would be Hadoop Distributed File System (HDFS)
- Addition of a data store would provide a nicer interface to store and manage your data
  - HBase: A key-value store implemented on top of HDFS
  - Traditionally one could use RDBMS on top of a local file system

Hadoop Eco System



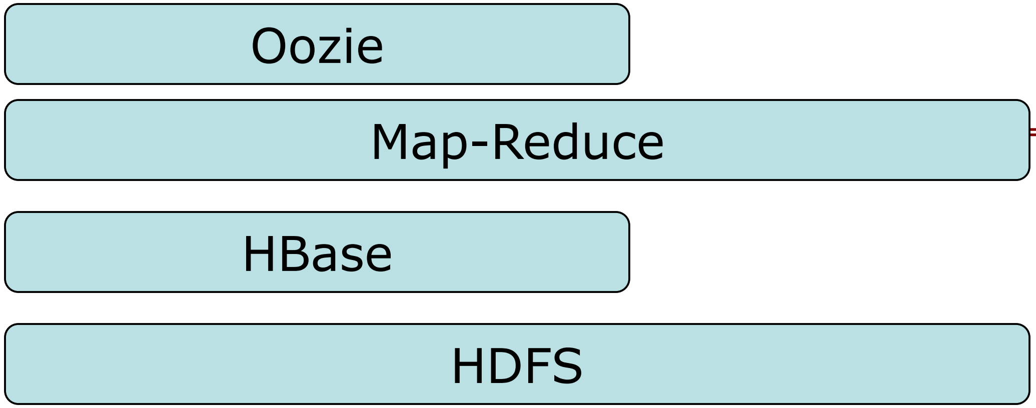
Map-Reduce

HBase

HDFS

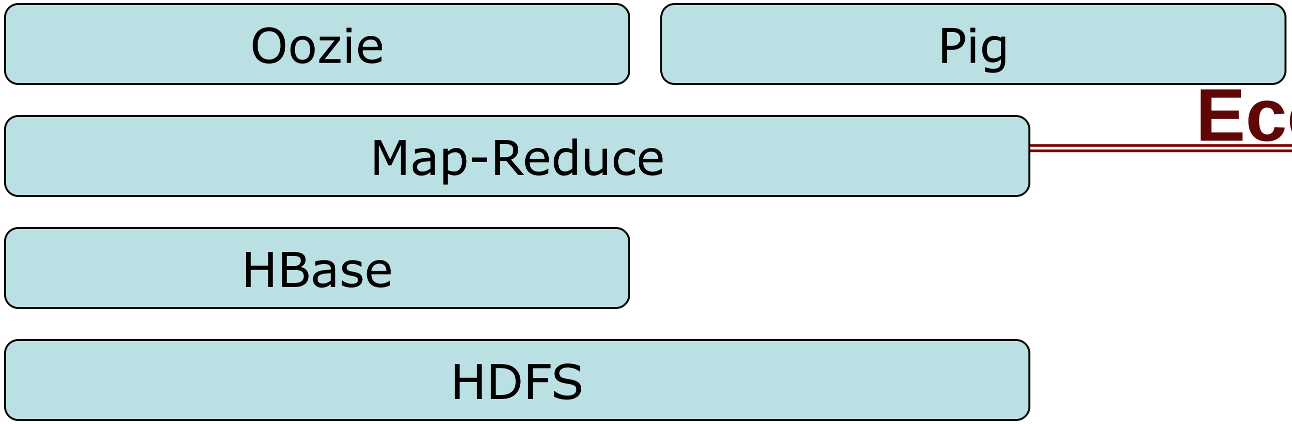
- For batch processing, you will need to utilize a framework
  - In Hadoop’s world that would be MapReduce
  - MapReduce will ease implementation of distributed applications that will run on a cluster of commodity hardware

Hadoop Eco System



- For batch processing, you will need to utilize a framework
  - In Hadoop's world that would be MapReduce
  - MapReduce will ease implementation of distributed applications that will run on a cluster of commodity hardware
- Many problems lend themselves to a MapReduce solution with multiple jobs
  - Apache Oozie is a popular MapReduce workflow and coordination product

Hadoop Eco System



Oozie

Pig

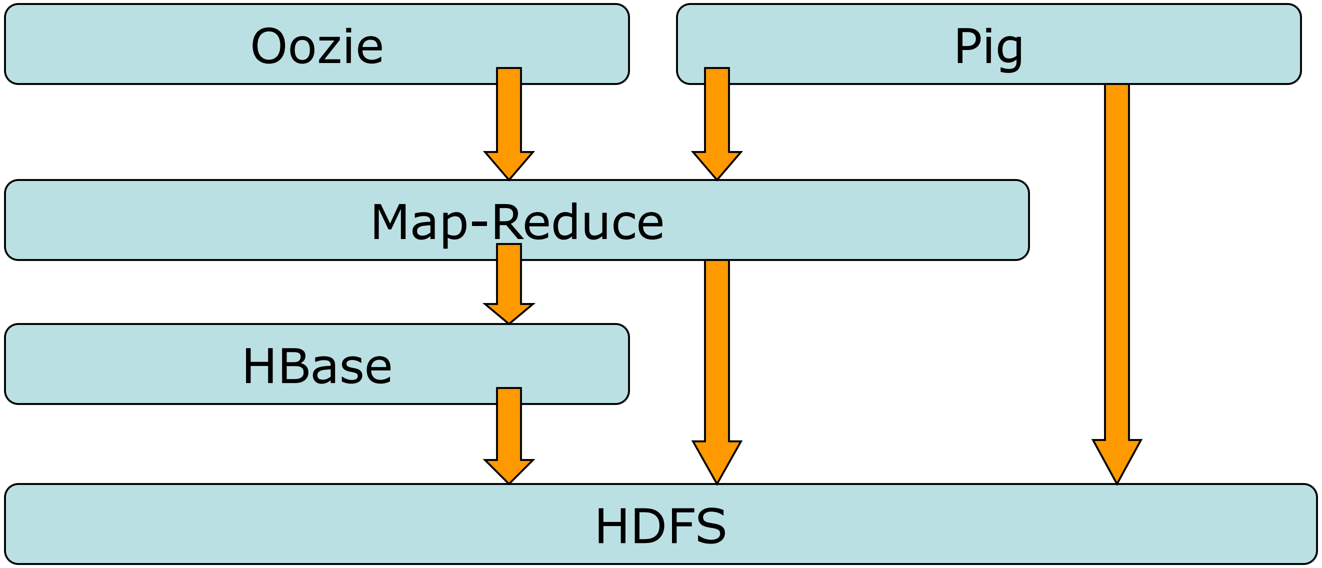
Map-Reduce

HBase

HDFS

- MapReduce paradigm may not work well for analysts and data scientists
  - Addition of Apache Pig, a high-level data flow scripting language, may be beneficial

Hadoop
Eco System



Hadoop System Principles

- Scale-Out rather than Scale-Up
- Bring code to data rather than data to code
- Deal with failures – they are common
- Abstract complexity of distributed and concurrent applications

Hadoop System Principles:
Scale-Out rather than Scale-Up

- It is harder and more expensive to scale-up
  - Add additional resources to an existing node (CPU, RAM)
  - Moore’s Law can’t keep up with data growth
  - New units must be purchased if required resources can not be added
  - Also known as scale vertically
- Scale-Out
  - Add more nodes/machines to an existing distributed application
  - Software Layer is designed for node additions or removal
- Hadoop takes this approach - A set of nodes are bonded together as a single distributed system
  - Very easy to scale down as well

Hadoop System Principles: Code to Data

- Traditional data processing architecture
  - Nodes are broken up into separate processing and storage
  - Nodes connected by high-capacity link
  - Many data-intensive applications are not CPU demanding
  - Causing bottlenecks in network
- Hadoop co-locates processors and storage
  - Code is moved to data (size is tiny, usually in KBs)
  - Processors execute code and access underlying local storage

Hadoop System Principles: Deal with failures

- Hadoop is designed to cope with node failures
  - Data is replicated
  - Tasks are retried

HDFS is NOT good for…

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data
- HBase addresses this issue
- Large amount of small files
  - Better for millions of large files instead of billions of small files
- For example each file can be 100MB or more
- Multiple Writers
  - Single writer per file
  - Writes only at the end of file, no-support for arbitrary offset

HDFS Resources

- Home Page
  - http://hadoop.apache.org
- Mailing Lists
  - http://hadoop.apache.org/mailing_lists.html
- Wiki
  - http://wiki.apache.org/hadoop
- Documentation is found across many pages and versions:
  - http://hadoop.apache.org/docs/hdfs/current
  - http://hadoop.apache.org/docs/r2.0.2-alpha/
  - HDFS Guides: http://hadoop.apache.org/docs/r0.20.2

Large Scale Computations

- Hadoop provides a scalable distributed storage infrastucture
- How to process that data?

- MapReduce

Two Approaches to Analyzing Data

- Machine Learning approach
  - Emphasizes sophisticated algorithms e.g., Decision Trees, Support Vector Machines
  - Data sets tend to be small, fit in memory
- Data Mining approach
  - Emphasizes big data sets (e.g., in the terabytes)
  - Data cannot even fit on a single disk!
  - Necessarily leads to simpler algorithms

Philosophy

- In many cases, adding more data leads to better results than improving algorithms
  - Netflix
  - Google search
  - Google ads

Large Scale Computations

- E.g. Google ngrams
  - Google Research team has been using word n-gram models for R&D projects: statistical MT, speech recognition, spelling correction, entity detection, information extraction, etc.
  - Training corpus from public Web pages
  - Processed 1,024,908,267,229 words of running text
  - Published the counts for all 1,176,470,663 five-word sequences that appear at least 40 times
  - There are 13,588,391 unique words, after discarding words that appear less than 200 times

Large Scale Computations

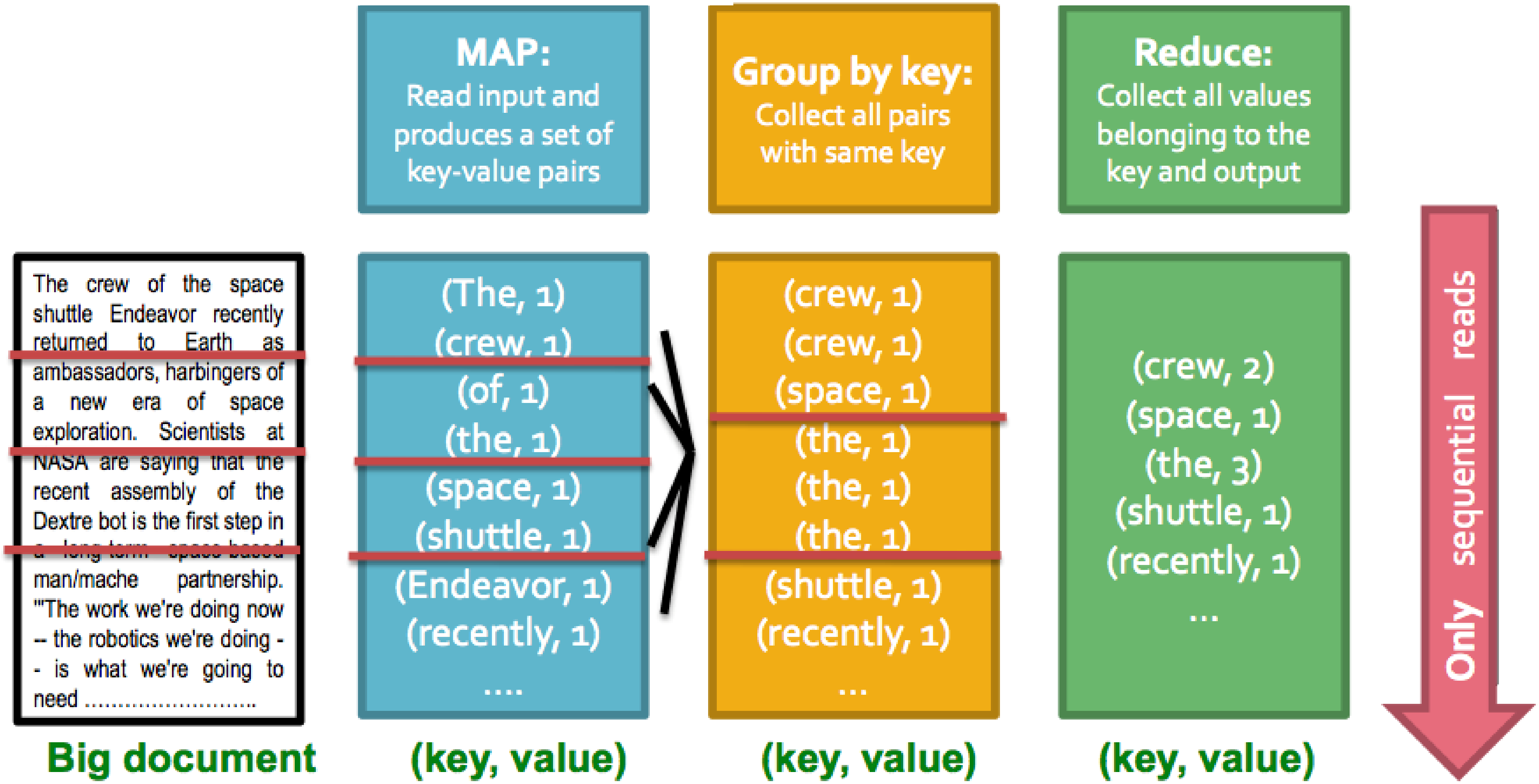
- Linguistics Data Consortium (LDC)
  - File sizes: approx. 24 GB compressed (gzip'ed) text files
  - Number of tokens: 1,024,908,267,229
  - Number of sentences: 95,119,665,584
  - Number of unigrams: 13,588,391
  - Number of bigrams: 314,843,401
  - Number of trigrams: 977,069,902
  - Number of fourgrams: 1,313,818,354
  - Number of fivegrams: 1,176,470,663

Word Count

- Large corpus of documents
- Count the number of times each distinct word appears in the corpus
- Sample application: analyze web server logs to find popular URLs
- Naturally parallelizable!

MapReduce Framework

- Word count job
- Input is a body of text from HDFS
  - Split text into tokens
  - For each first word sum up all occurrences
- Output to HDFS



MapReduce Framework

- Sequentially read a lot of data
- Map:
  - Extract something you care about
- Group by key: Sort and Shuffle
- Reduce:
  - Aggregate, summarize, filter or transform
- Write the result

Word Count with MapReduce

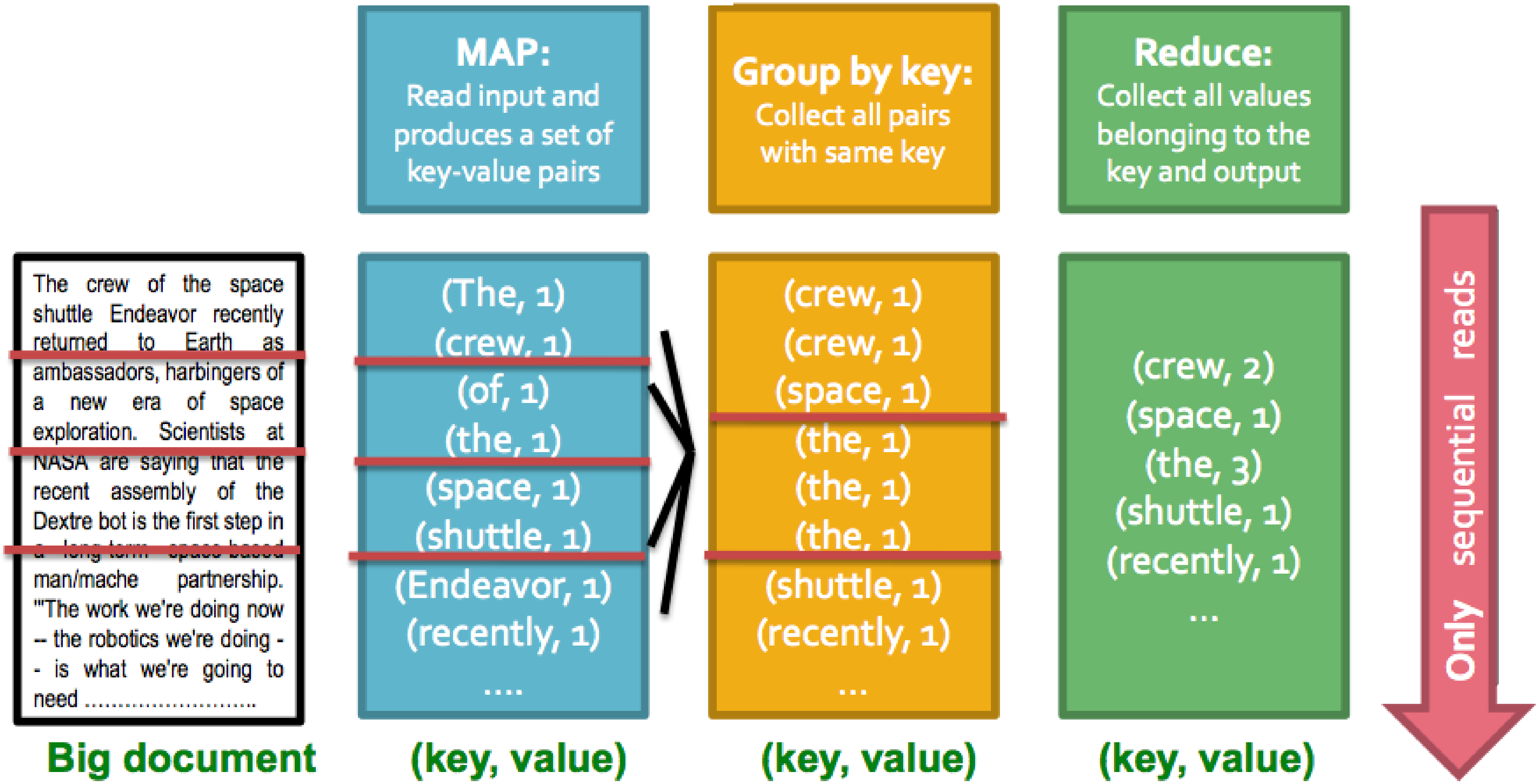
- Input: a set of key-value pairs
- Programmer specifies two methods:
- **Map (k, v) ->** <k’, v’>*
  - Takes a key-value pair and outputs a set of key-value p airs
  - There is one Map call for every (k,v) pair
- **Reduce (k’, <v’>*) ->** <k’, v’’>*
  - All values v’ with same key k’ are reduced together and processed in v’ order
  - There is one Reduce function call per unique key k’

MapReduce Framework

- Divided in two phases
  - Map phase
  - Reduce phase
- Both phases use key-value pairs as input and output
- The implementer provides map and reduce functions
  - MapReduce framework orchestrates splitting, and distributing of Map and Reduce phases

MapReduce Framework

- Most of the pieces can be easily overridden
- Job – execution of map and reduce functions to accomplish a task
- Task – single Mapper or Reducer
  - Performs work on a fragment of data



The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need ....

Big document

MAP:
Read input and produces a set of key-value pairs

(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
....

(key, value)

Group by key:
Collect all pairs with same key

(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
...

(key, value)

Reduce:
Collect all values belonging to the key and output

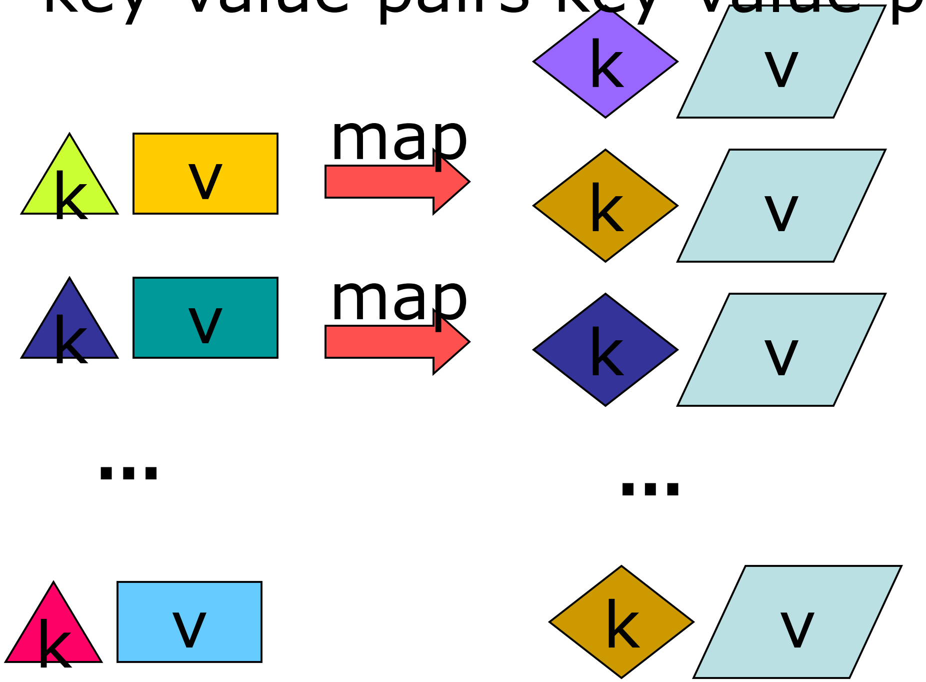
(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
...

(key, value)

MapReduce: The Map Step

Input
key-value pairs

Intermediate
key-value pairs

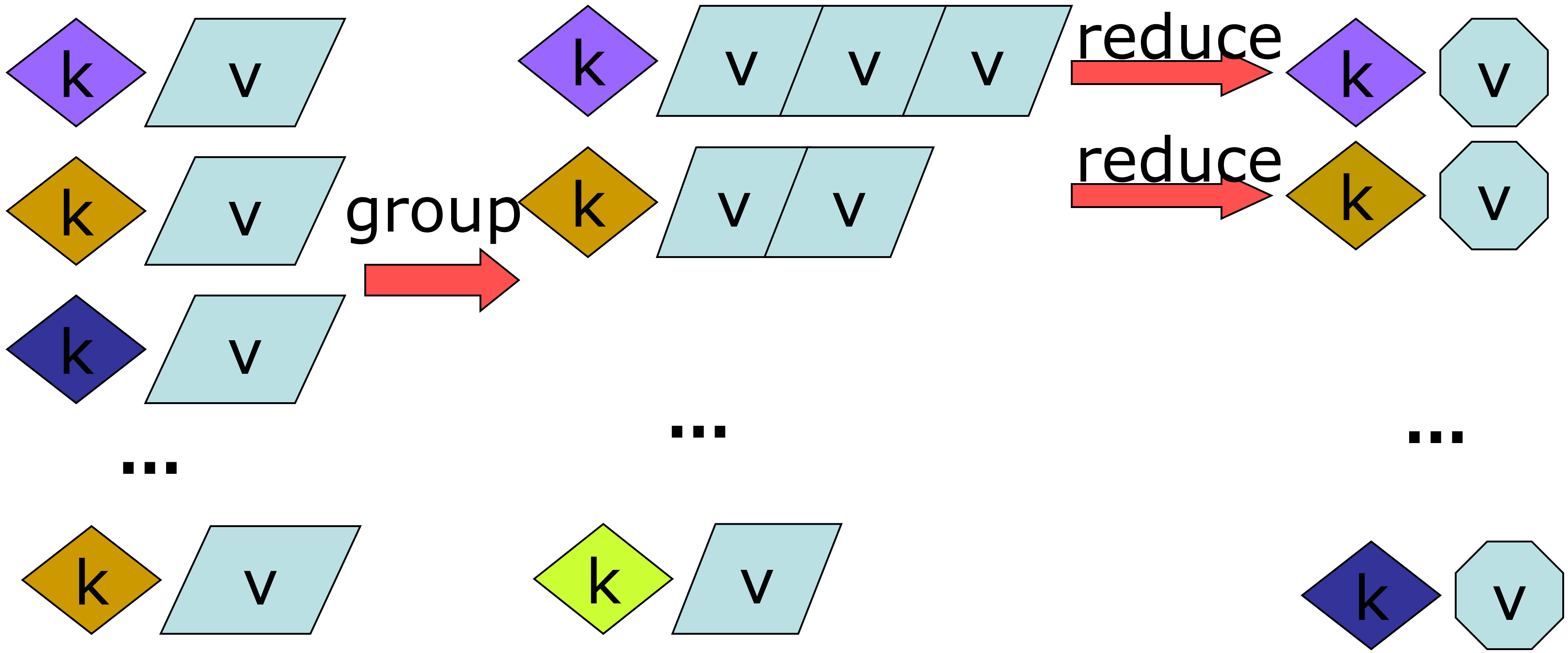


MapReduce: The Reduce Step

Intermediate key-value pairs

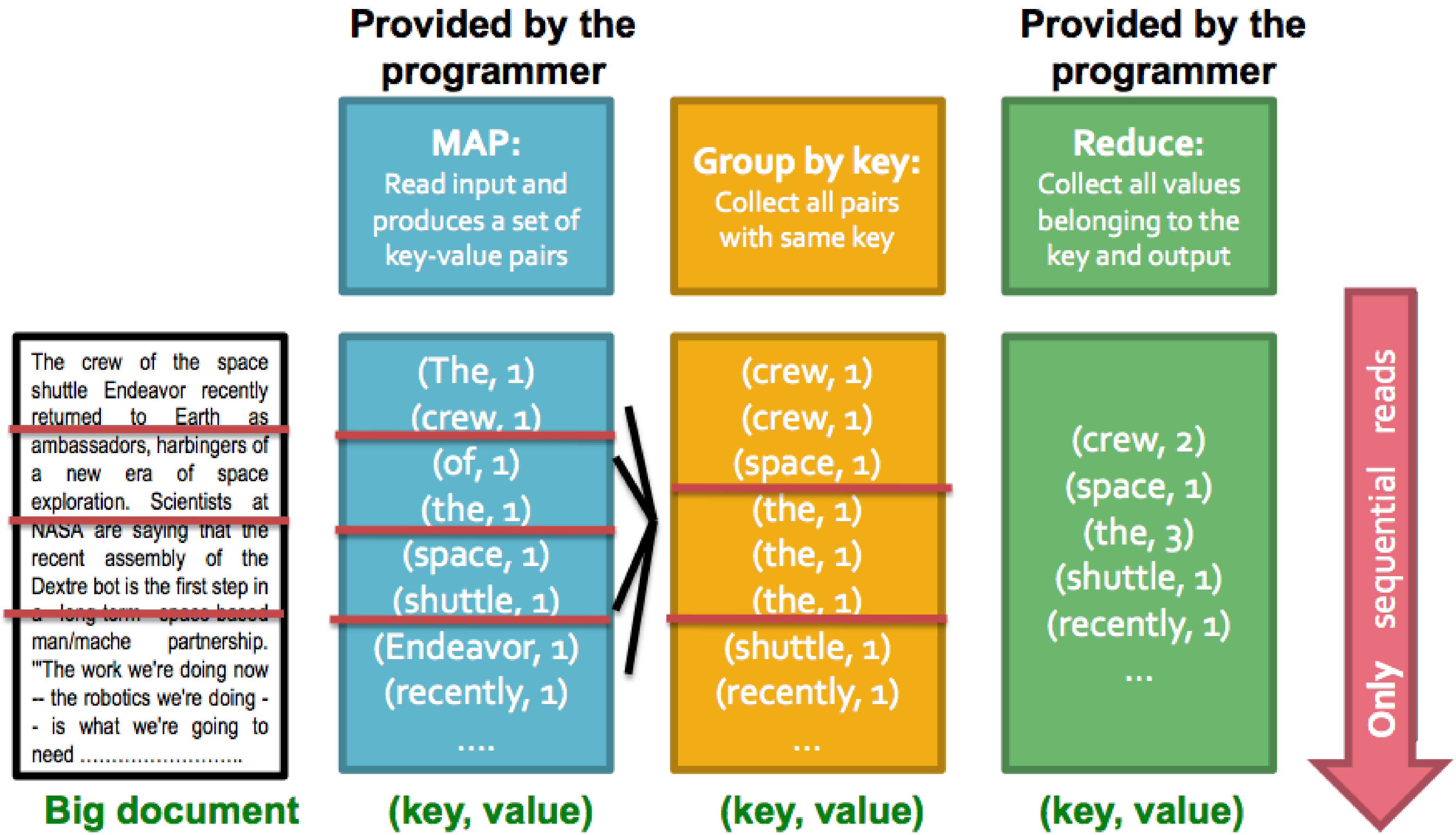
Key-value groups

Output key-value pairs



MapReduce

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs



Provided by the programmer

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key

Provided by the programmer

Reduce:
Collect all values belonging to the key and output

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need ……………………..

(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
….

(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
…

(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
…

Only sequential reads

Big document

(key, value)

(key, value)

(key, value)

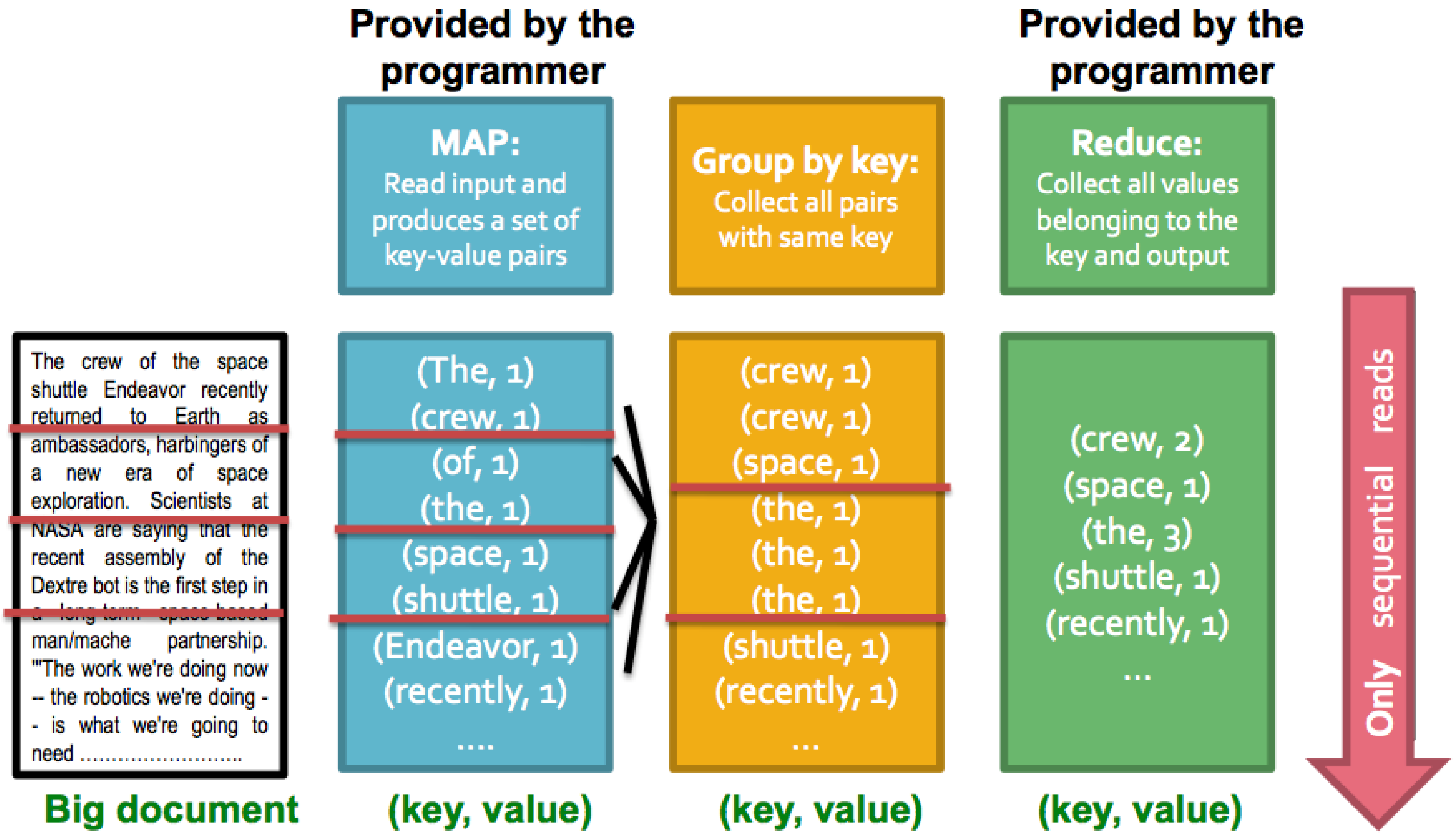
Word Count using MapReduce

- map(key, value):
  - key: document name; value: text of document
  - for each word w in value:

    emit(w, 1)

- reduce(key, values):
  - key: a word; values: an iterator over counts

  ```
  result = 0
  for each count v in values:
          result += v
  emit(key, result)
  ```



Provided by the programmer

Provided by the programmer

MAP: Read input and produces a set of key-value pairs

Group by key: Collect all pairs with same key

Reduce: Collect all values belonging to the key and output

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need ……………………..

(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
….

(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
…

(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
…

Only sequential reads

Big document

(key, value)

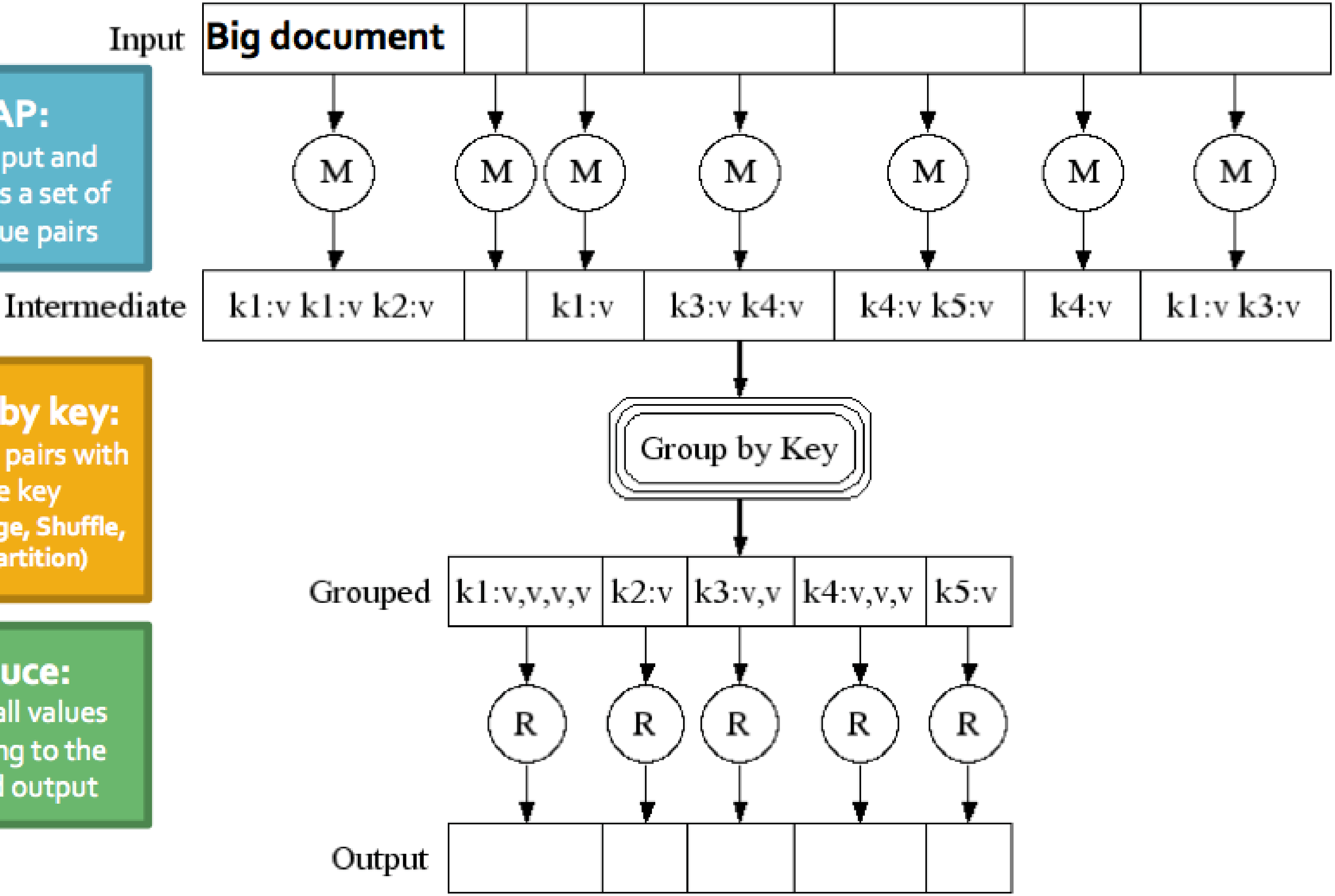
(key, value)

(key, value)

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



Input

Big document

M M M M M M M

Intermediate

k1:v k1:v k2:v | | k1:v | k3:v k4:v | k4:v k5:v | k4:v | k1:v k3:v

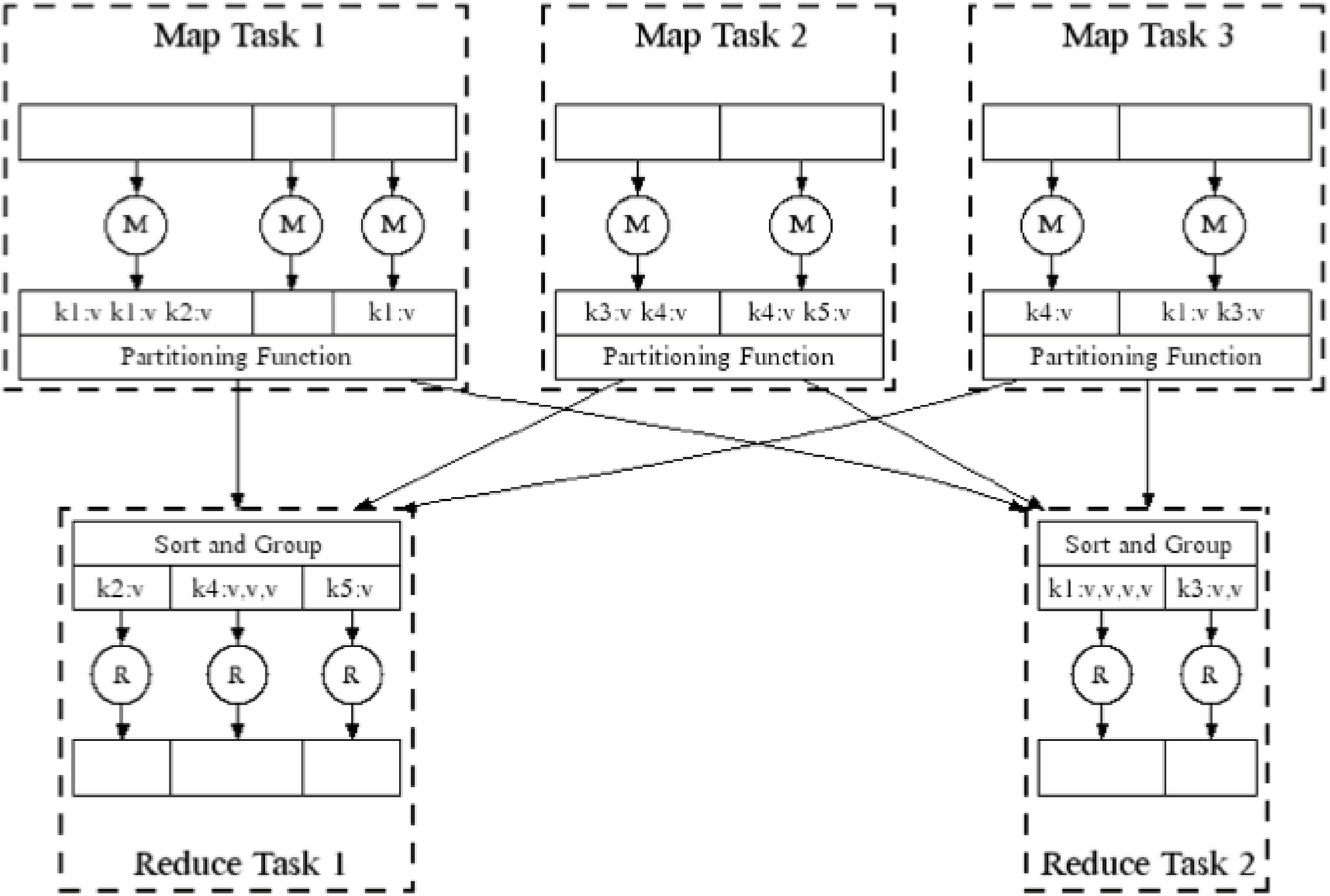
Group by Key

Grouped

k1:v,v,v,v | k2:v | k3:v,v | k4:v,v,v | k5:v

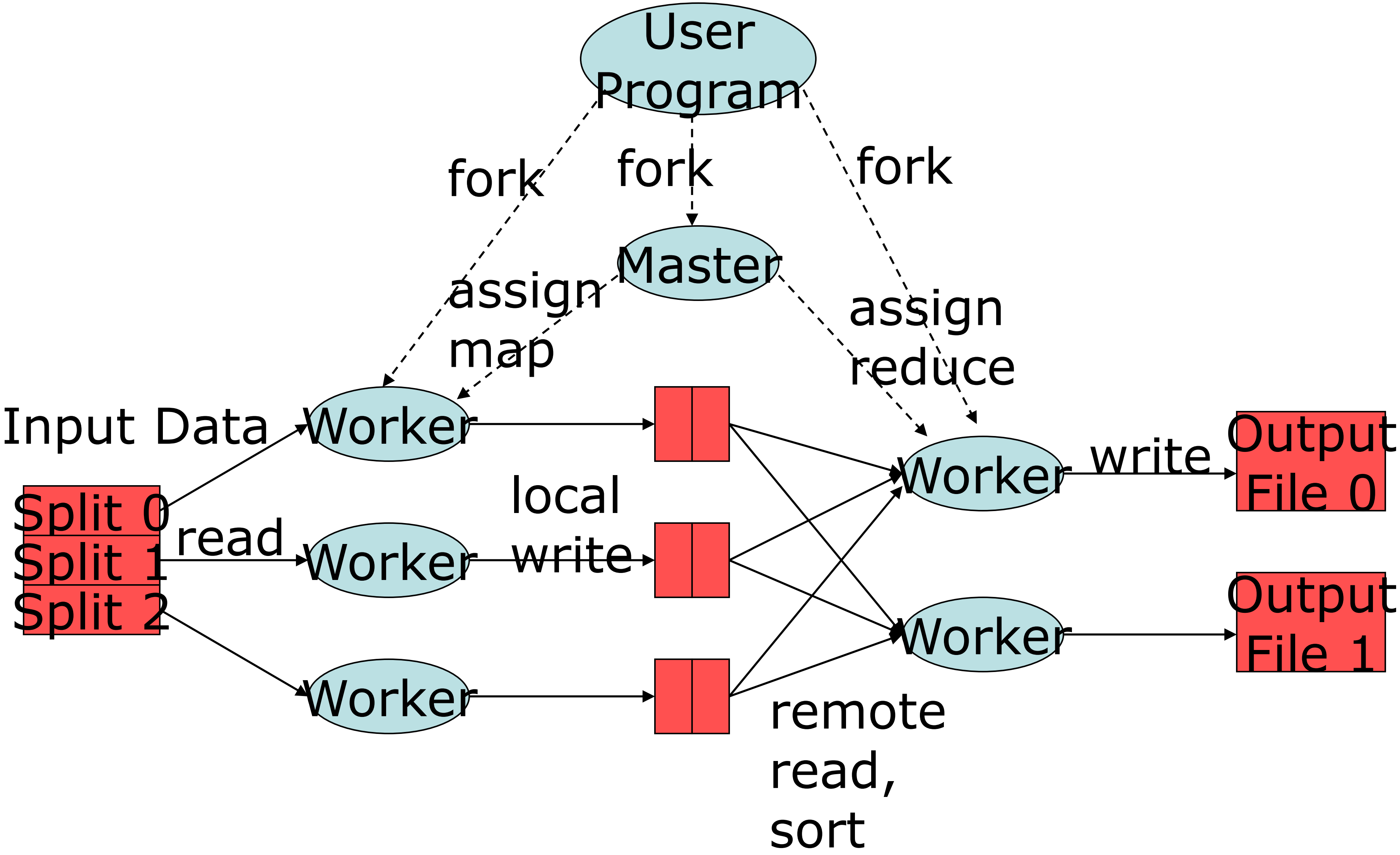
R R R R R

Output



All phases are distributed with many tasks doing the work

Distributed Execution Overview



Data flow

- Input, final output are stored on a distributed file system
  - 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 map reduce task

Coordination

- Master data structures
  - 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

Failures

- Map worker failure
  - Map tasks completed 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
- Master failure
  - MapReduce task is aborted and client is notified

Map Reduce Tasks

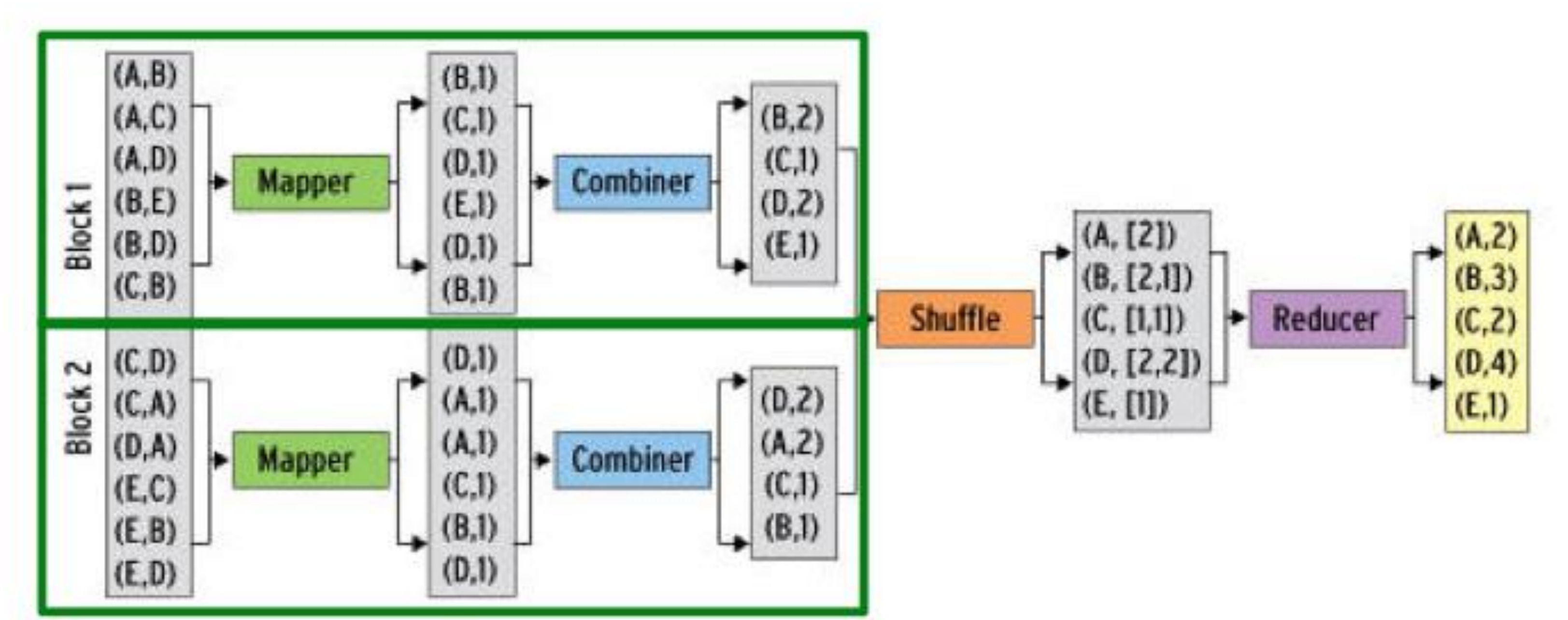
- Map-Reduce environment takes care of
  - Partitioning the input data
  - Scheduling the program
  - execution across a set of machines
  - Performing the group by key step
  - Handling machine failures
  - Managing required intermachine communication

Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

Combiner

- Combiner combines the values of all keys of a single mapper (single machine)



- Much less data needs to be copied and shuffled

Partition Function

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

Implementations

- Google
  - Not available outside Google
- Hadoop
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

Map-Reduce Resources

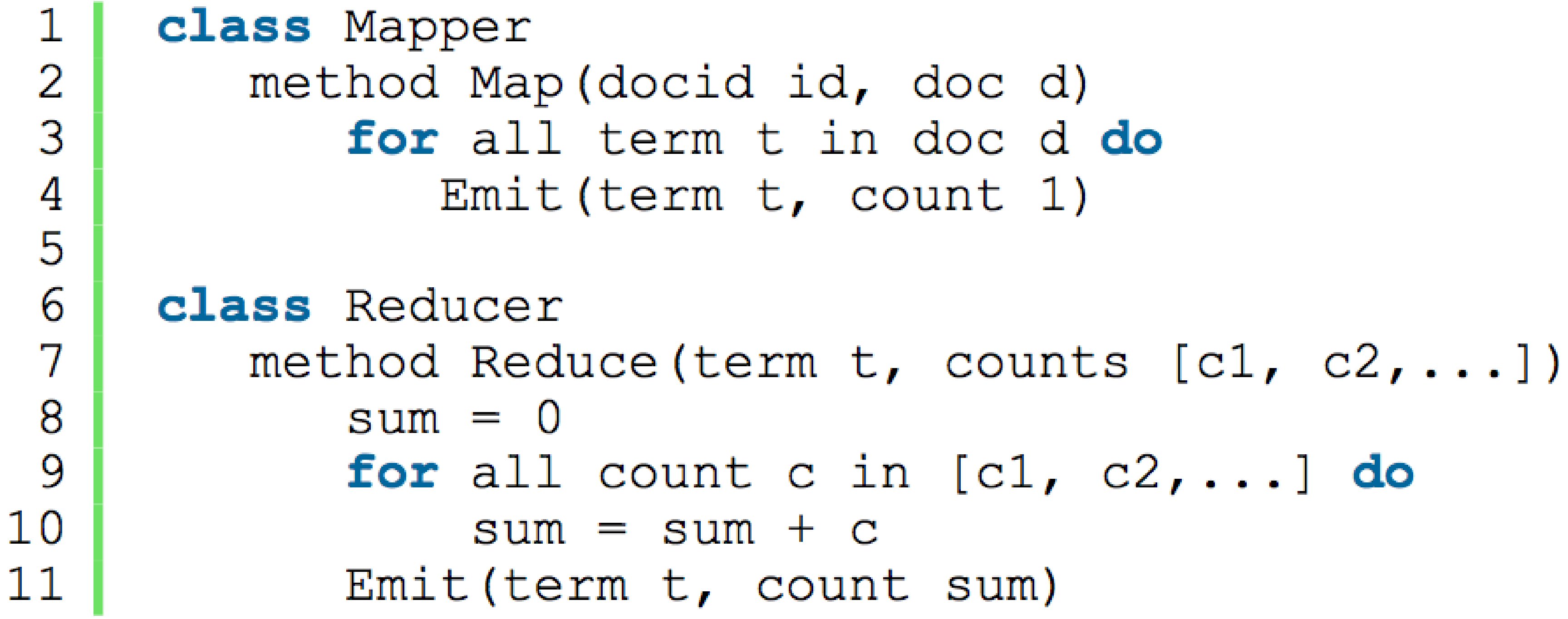
- Jeffrey Dean and Sanjay Ghemawat,
  - MapReduce: Simplified Data Processing on Large Clusters

    http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung,
  - The Google File System

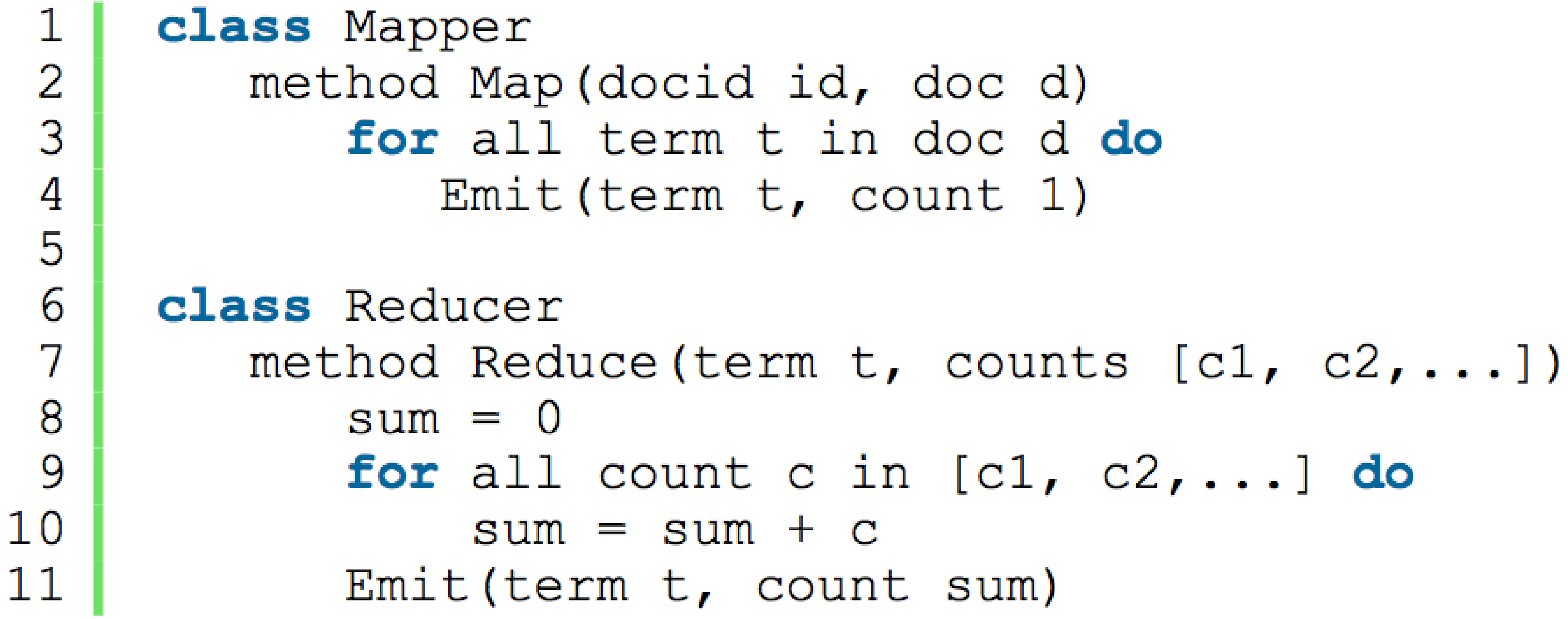
    http://labs.google.com/papers/gfs.html

MapReduce Examples: Word Counting



```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

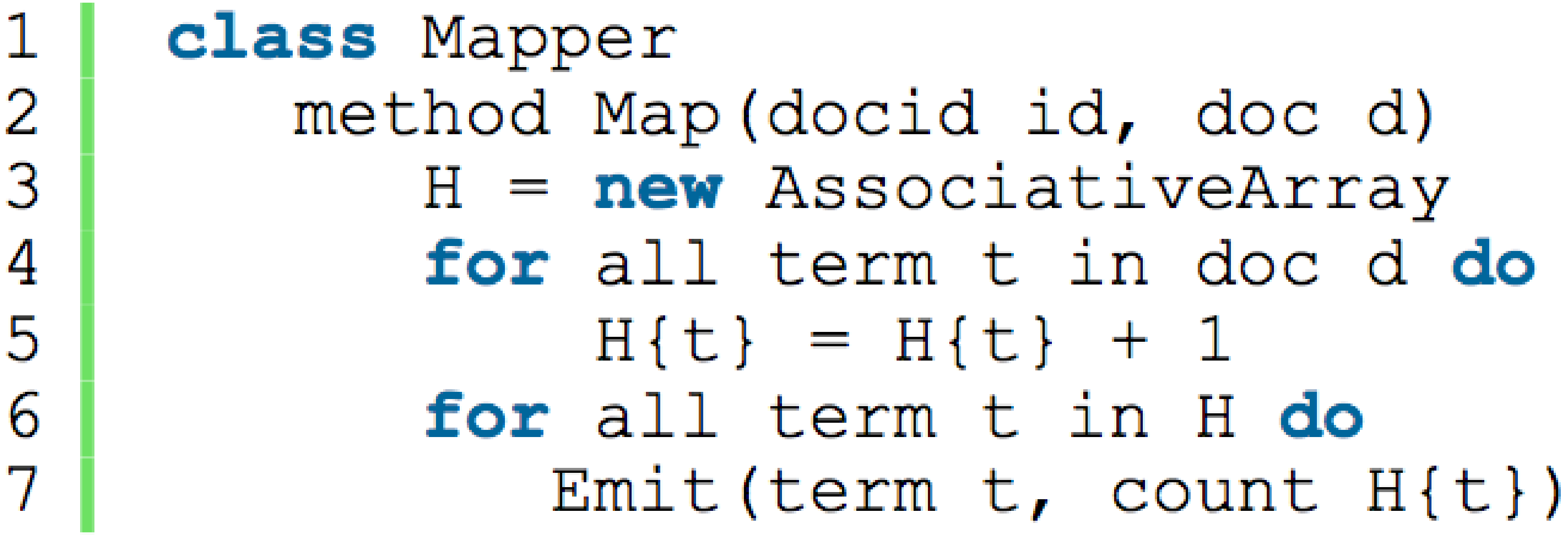
MapReduce Examples: Word Counting



```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

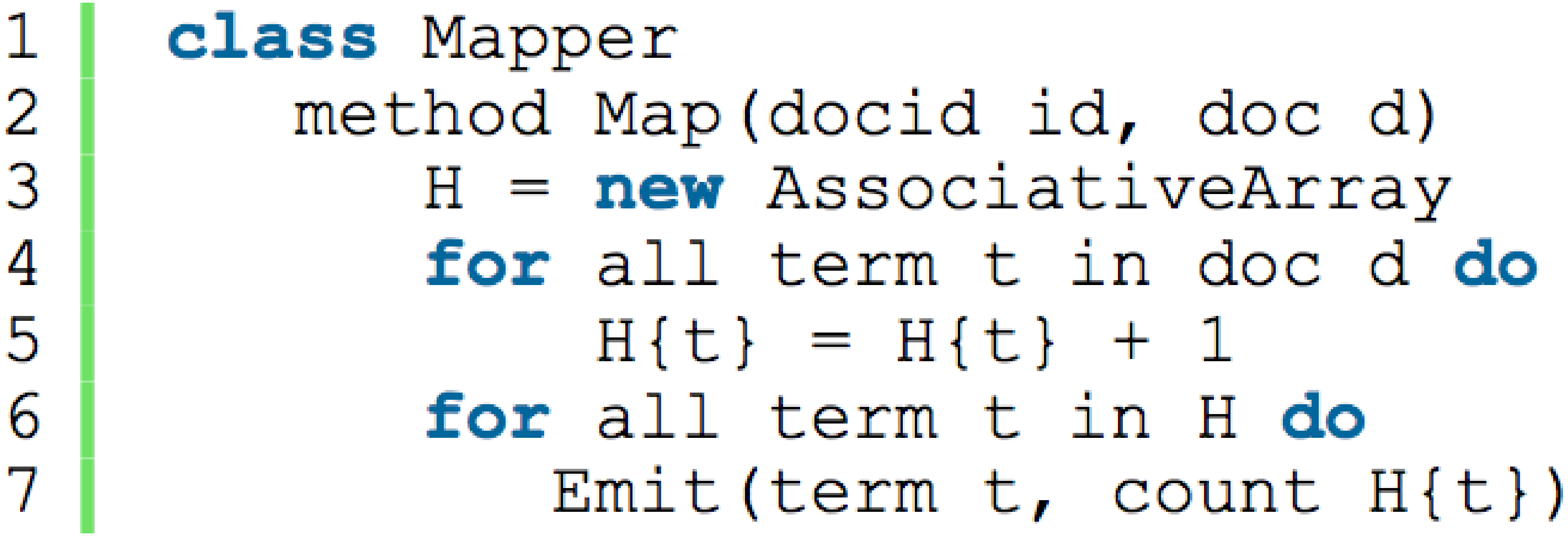
Disadvantage of this approach is a high amount of dummy counters emitted by the Mapper.

MapReduce Examples: Word Counting



```
class Mapper
   method Map(docid id, doc d)
      H = new AssociativeArray
      for all term t in doc d do
          H{t} = H{t} + 1
      for all term t in H do
         Emit(term t, count H{t})
```

MapReduce Examples: Word Counting

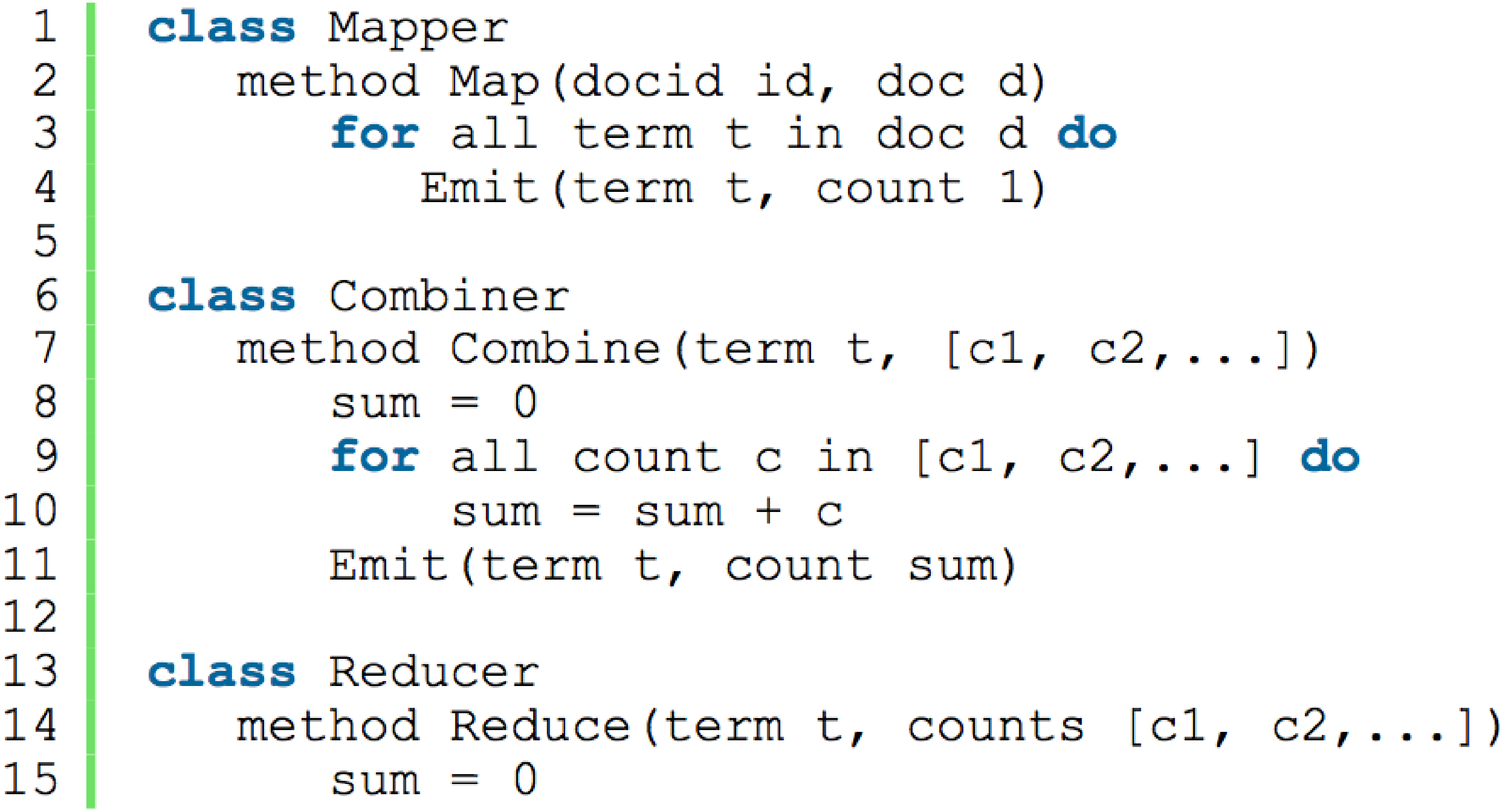


```
class Mapper
   method Map(docid id, doc d)
      H = new AssociativeArray
      for all term t in doc d do
          H{t} = H{t} + 1
      for all term t in H do
         Emit(term t, count H{t})
```

The Mapper can decrease a number of counters via summing counters
for each document

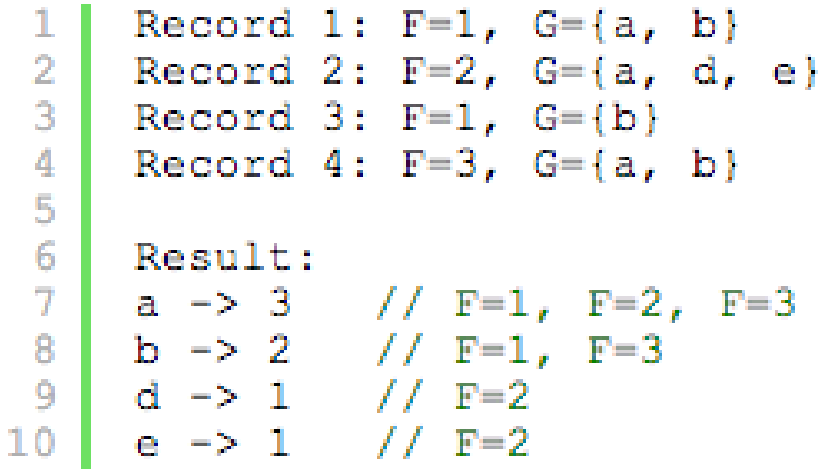
MapReduce Examples: Word Counting

- Accumulate counters not only for one document, but for all documents processed by one Mapper node



```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Combiner
   method Combine(term t, [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
         sum = sum + c
      Emit(term t, count sum)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
```

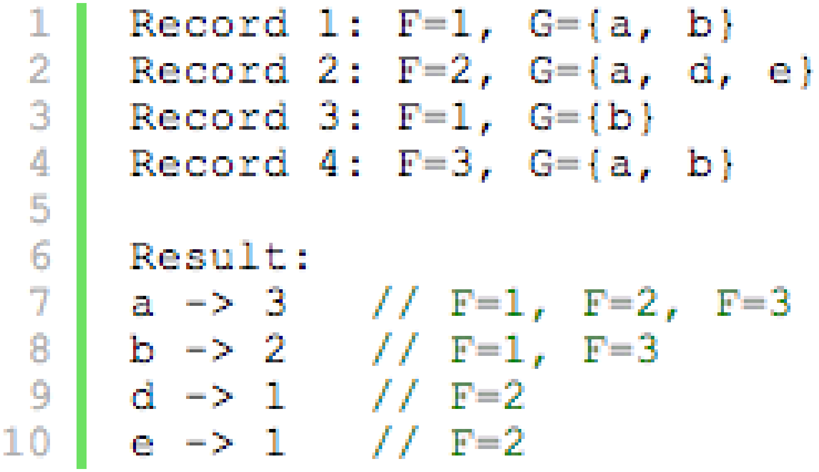
MapReduce Examples: Unique Items Counting



```
Record 1: F=1, G={a, b}
Record 2: F=2, G={a, d, e}
Record 3: F=1, G={b}
Record 4: F=3, G={a, b}

Result:
a -> 3   // F=1, F=2, F=3
b -> 2   // F=1, F=3
d -> 1   // F=2
e -> 1   // F=2
```

- There is a set of records that contain fields F and G. Count the total number of unique values of field F for each subset of records that have the same G (grouped by G).

MapReduce Examples: Unique Items Counting



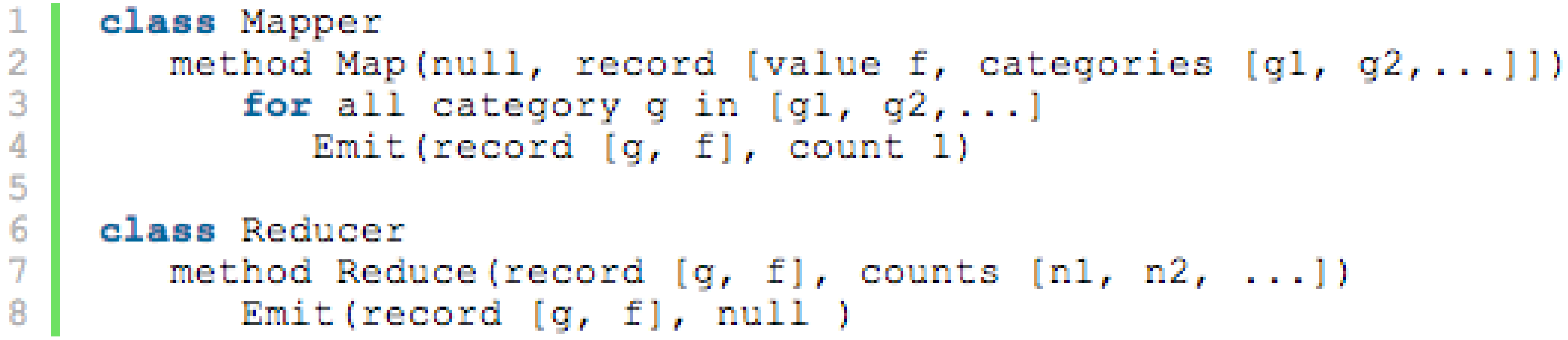
```
Record 1: F=1, G={a, b}
Record 2: F=2, G={a, d, e}
Record 3: F=1, G={b}
Record 4: F=3, G={a, b}

Result:
a -> 3   // F=1, F=2, F=3
b -> 2   // F=1, F=3
d -> 1   // F=2
e -> 1   // F=2
```

- The problem can be a little bit generalized and formulated in terms of faceted search:
  - Problem Statement: There is a set of records. Each record has field F and arbitrary number of category labels G = {G1, G2, …} . Count the total number of unique values of filed F for each subset of records for each value of any label

MapReduce Examples: Unique Items Counting

- Two stages, 2 MapReduce Jobs
  - At stage 1 Mapper emits dummy counters for each pair of F and G; Reducer calculates a total number of occurrences for each such pair. The main goal of this phase is to guarantee uniqueness of F values
  - At stage 2 pairs are grouped by G and the total number of items in each group is calculated

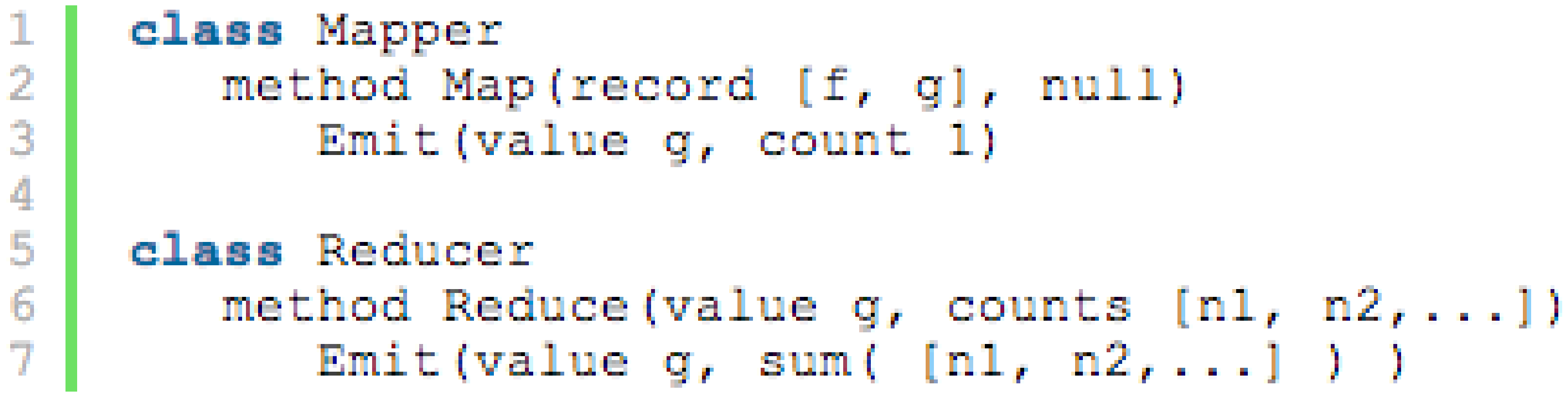
MapReduce Examples: Unique Items Counting



```
class Mapper
   method Map(null, record [value f, categories [g1, g2,...]])
      for all category g in [g1, g2,...]
         Emit(record [g, f], count 1)

class Reducer
   method Reduce(record [g, f], counts [n1, n2, ...])
      Emit(record [g, f], null )
```

- Stage 1: Mapper emits dummy counters for each pair of F and G; Reducer calculates a total number of occurrences for each such pair. The main goal of this phase is to guarantee uniqueness of F values

MapReduce Examples

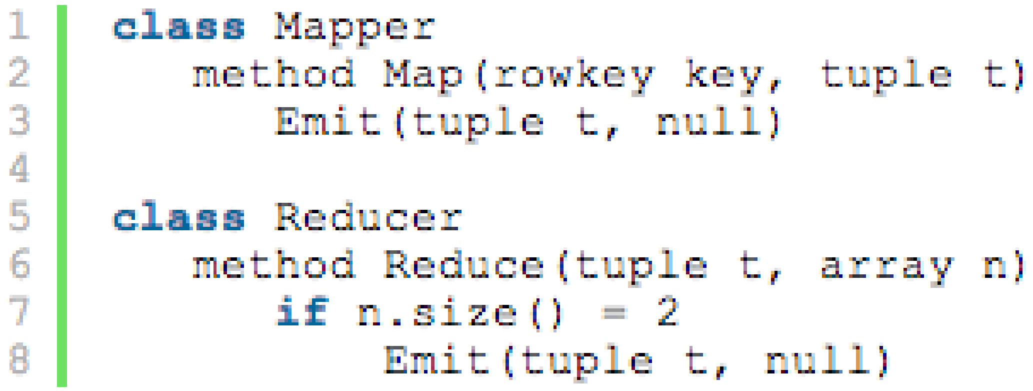


```
class Mapper
    method Map(record [f, g], null)
        Emit(value g, count 1)

class Reducer
    method Reduce(value g, counts [n1, n2,...])
        Emit(value g, sum( [n1, n2,...] ) )
```

- Stage 2: pairs are grouped by G and the total number of items in each group is calculated

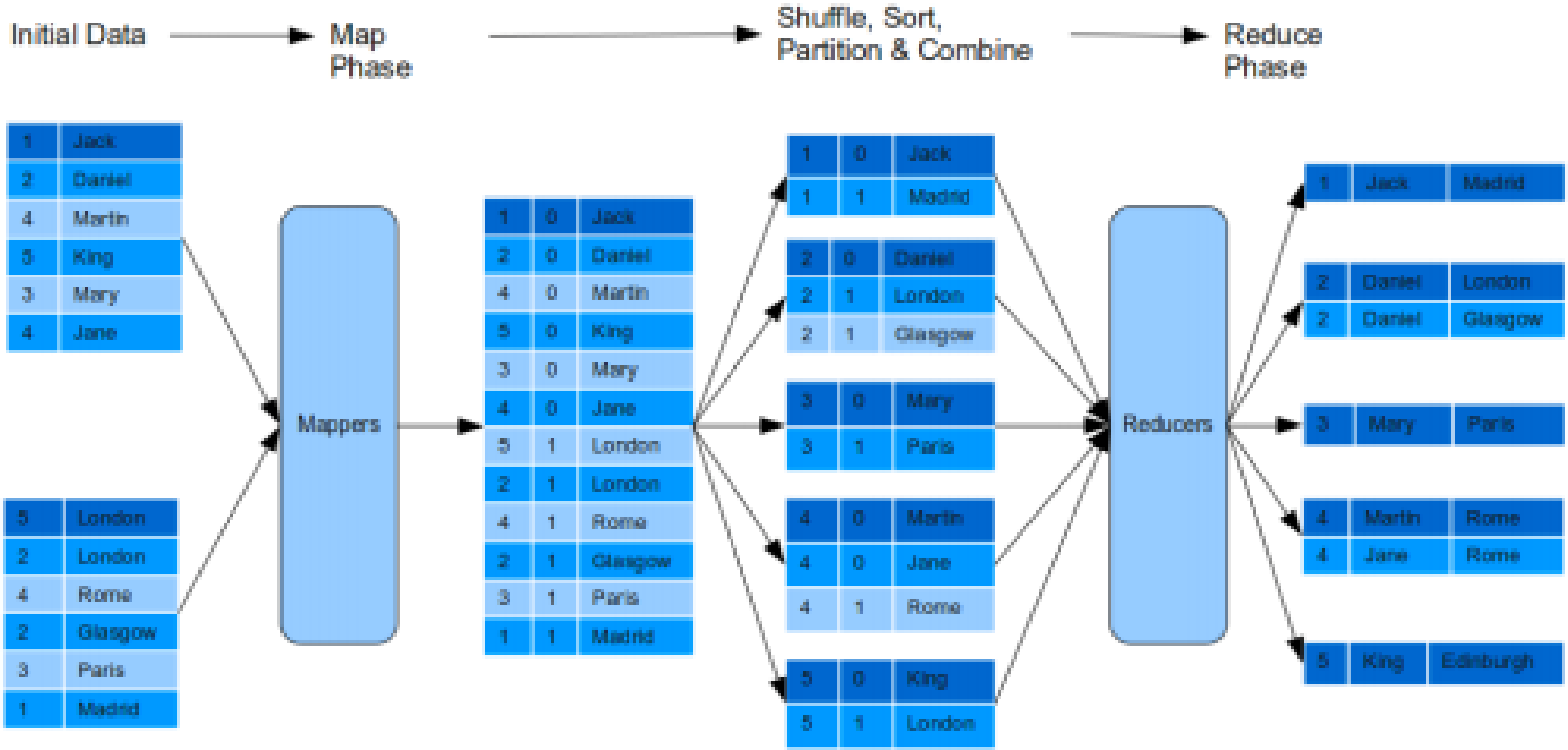
MapReduce Examples: Intersection

- Mappers are fed by all records of two sets to be intersected
  - Reducer emits only records that occurred twice.
  - Works only if both sets contain this record because record includes primary key and can occur in one set only once



```
class Mapper
   method Map(rowkey key, tuple t)
      Emit(tuple t, null)

class Reducer
   method Reduce(tuple t, array n)
      if n.size() = 2
          Emit(tuple t, null)
```

MapReduce Examples: Join



1	Jack
2	Daniel
4	Martin
5	King
3	Mary
4	Jane

5	London
2	London
4	Rome
2	Glasgow
3	Paris
1	Madrid

1	0	Jack
2	0	Daniel
4	0	Martin
5	0	King
3	0	Mary
4	0	Jane
5	1	London
2	1	London
4	1	Rome
2	1	Glasgow
3	1	Paris
1	1	Madrid

1	0	Jack
1	1	Madrid

2	0	Daniel
2	1	London
2	1	Glasgow

3	0	Mary
3	1	Paris

4	0	Martin
4	0	Jane
4	1	Rome

5	0	King
5	1	London

1	Jack	Madrid

2	Daniel	London
2	Daniel	Glasgow

3	Mary	Paris

4	Martin	Rome
4	Jane	Rome

5	King	Edinburgh

- 2 Simple examples
  - Mapper, Reducer pseudocode
  - Implementation example
- Can be solved differently
- An important consideration is efficiency. The more mappers and reducers we use, the more computation is done in parallel and the faster is the overall computation

Efficient Implementation

- We know that each reducer gets all values for the same key. Thus, if we have only one key or only few keys, we cannot use many reducers
- For many problems we will have to have one reducer in the end. But we should try to get as much computation done in parallel as possible before we get to the one reducer
- The following examples will illustrate this point

Solution 1: MapReduce for Max Integer

- MapReduce Job

- Mapper // Compute max per line
  - Input: key = fileID_lineN,
    value = line
  - Output: key = 1,
    value = max per line

```
Input< fileID_lineNumber,line>
        Int max = Integer.NegInfinity;
        String[] tokens = line.split("\n"); // Tokenize the line
        For( String token: tokens){
                Int value = Integer.parseString(token);
                If(value > max){// Compute the max for the line
                        Max = value;
                }}
        1);
```

- Reducer // Compute the overall max
  - Input: key = 1,
    value = array of max values per line for each file
  - Output: key = 1, value = max per file

```
Input<1, Integer[] values>
        Int max = Integer.NegInfinity;
        For(Integer value : values){
        // Compute the max for the file
                If (value > max){
                        Max = value;
                }}
        Emit(1,max);
```

Discussion

- In the example on the previous slide we use one MapReduce job
- The mapper's output key/value pairs are
  - Key = 1, value = max per line
- After all mappers are done, there will be key/value pairs with only one key = 1
- We can use only one reducer. That reducer will get all max values for all lines and compute the overall max
- This is not efficient because the reducer does a large amount of work NOT in parallel

Solution 2: MapReduce for Max Integer

- MapReduce Job #1

- Mapper // Compute max per line
  - Input: key = fileID_lineN,
    value = line
  - Output: key = fileID,
    value = max per line

```
Input< fileID_lineNumber,line>
        Int max = Integer.NegInfinity;
        String[] tokens = line.split("\n"); // Tokenize the line
        For( String token: tokens){
                Int value = Integer.parseString(token);
                If(value > max){// Compute the max for the line
                        Max = value;
                }}
        Emit(fileID,max);
```

- Reducer // Compute the max per file
  - Input: key = fileID,
    value = array of max values per line
  - Output: key = 1, value = max per file

```
Input<fileID, Integer[] values>
        Int max = Integer.NegInfinity;
        For(Integer value : values){
        // Compute the max for the file
                If (value > max){
                        Max = value;
                }}
        Emit(1,max);
```

Solution 2: MapReduce for Max Integer

- MapReduce Job #2

- Mapper // Compute the max overall
  - Input: key = 1, value = max per file
  - Output: key = 1, value = max per file

```
Input< 1,max>

        Emit(1,max);
```

- Reducer // Compute the overall max
  - Input: key = 1, array of max values for each file
  - Output: key = 1, value = max

```
Input<1, Integer[] maxValues>
        Int max = Integer.NegInfinity;
        For(Integer value : maxValues){
        // Compute the overall max
        If (value > max){
                        Max = value;
                }}
        Emit(1,max);
```

Discussion

- Solution 2 uses two MapReduce jobs
- The first MapReduce job will compute the max per file, the second will compute the max overall
- Note that the second mapper just passes though the key/value pairs. The second reducer will do the actual work. It is possible to use one MapReduce job with one Mapper and two Reducers instead
- This solution is more efficient because we at least compute the max per file with multiple reducers
- However, this solution still does not do more computation locally, on the mapper nodes

Solution 3: MapReduce for Max Integer

- MapReduce Job

- Mapper // Compute max per line
  - Input: key = fileID_lineN,
    value = line
  - Output: key = fileID,
    value = max per line

```
Input< fileID_lineNumber,line>
        Int max = Integer.NegInfinity;
        String[] tokens = line.split("\n"); // Tokenize the line
        For( String token: tokens){
                Int value = Integer.parseString(token);
                If(value > max){// Compute the max for the line
                        Max = value;
                }}
        Emit(fileID,max);
```

- Reducer // Compute the max per file
  - Input: key = fileID,
    value = array of max values per line
  - Output: key = 1, value = max per file

```
Input<fileID, Integer[] values>
        Int max = Integer.NegInfinity;
        For(Integer value : values){
        // Compute the max for the file
                If (value > max){
                        Max = value;
                }}
        Emit(1,max);
```

Solution 3: MapReduce for Max Integer

- Combiner// Compute the max overall

  For all files on the mapper node
  - Input: key = 1, value = max per file
  - Output: key = 1, value = max per node

```
Input<1, Integer[] maxValues>
        Int max = Integer.NegInfinity;
        For(Integer value : maxValues){
        // Compute the overall max
        If (value > max){
                        Max = value;
                }}
        Emit(1,max);
```

- Reducer // Compute the overall max
  - Input: key = 1, value = max per file
  - Output: key = 1, value = max

```
Input<1, Integer[] maxValues>
        Int max = Integer.NegInfinity;
        For(Integer value : maxValues){
        // Compute the overall max
        If (value > max){
                        Max = value;
                }}
        Emit(1,max);
```

MapReduce for Average

- Average: MapReduce Job #1

- Mapper // Compute the sum per line
  - Input: key = fileID_lineN, value = line
  - Output: key = fileID,

    value = <sum per line, N>

    N = number of integers per line

```
Input< fileID_lineNumber,line>
    Int sum = 0;
    String[] tokens = line.split("\n"); // Tokenize the line
    For( String token: tokens){
        Int value = Integer.parseString(token);
        sum+=value;
    Emit(fileID,<sum,tokens.length>);
```

- Reducer // Compute the sum per file
  - Input:key = fileID,

    value = <sum per line, N>
  - Output: key = 1, value = <sum per file,M>

    M = number of integers per file

```
Input<fileID, Pair<sum,N>[] values>
    Int sum = 0;
    Int M = 0;
    For(Pair value : values){
        sum += value.sum;
        M+= value.N;
    Emit(1,<sum,M>);
```

MapReduce for Average

- Average: MapReduce Job #2

- Mapper
  - Input: key = 1, value = <sum per file,M>
  - Output: key = 1, value = <sum per file,M>

```
Input< 1,<sum,M>>

        Emit(1,<sum,M>);
```

- Reducer
  - Input: key = 1, value = <sum per file,M>
  - Output: key = 1, value = average

```
Input<1, Pair<sum,M>[] values>
        Int sum = 0;
        Int N = 0;
        For(Pair value : values){
                sum += value.sum;
                N+= value.M;
        Emit(1,sum/N);
```

- Some other examples for usages of Map-Reduce
- Google used to use Map-Reduce for a lot of computations