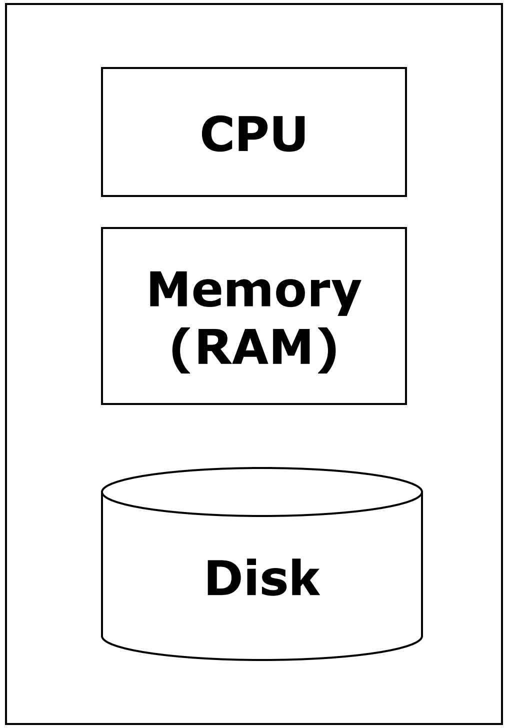
Hadoop: HDFS and Map Reduce

Wojtek Kowalczyk

(Chapter 2 MMDS book)

Based on:
http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt
https://teaching.csse.uwa.edu.au/units/CITS3402/labs/project2-2018/amp_mapreduce.pdf

Single-node architecture



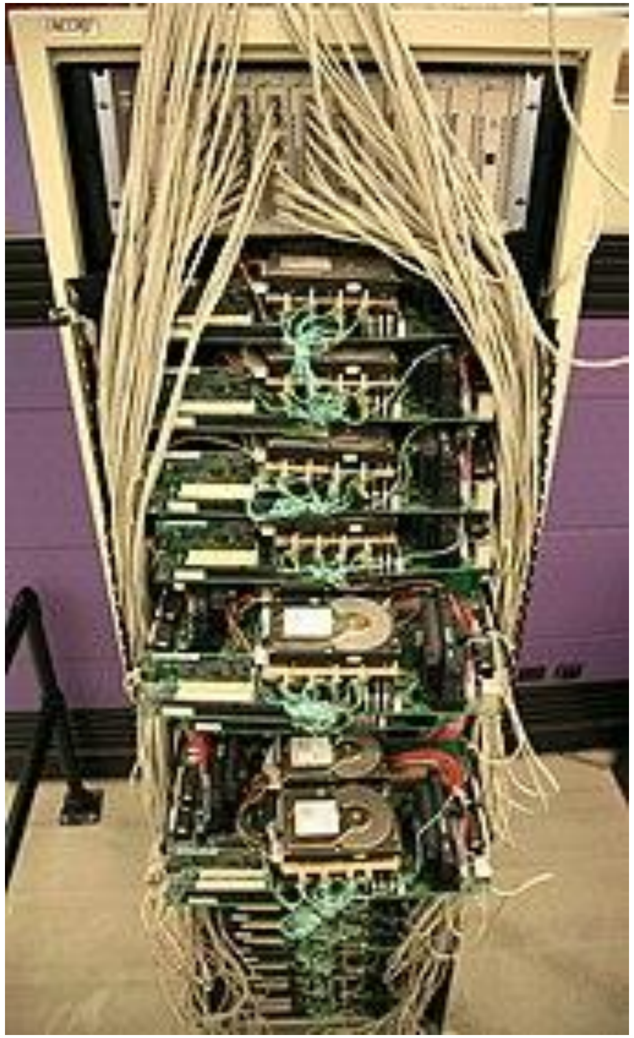
Machine Learning, Statistics

"Classical" Data Mining

Commodity Clusters

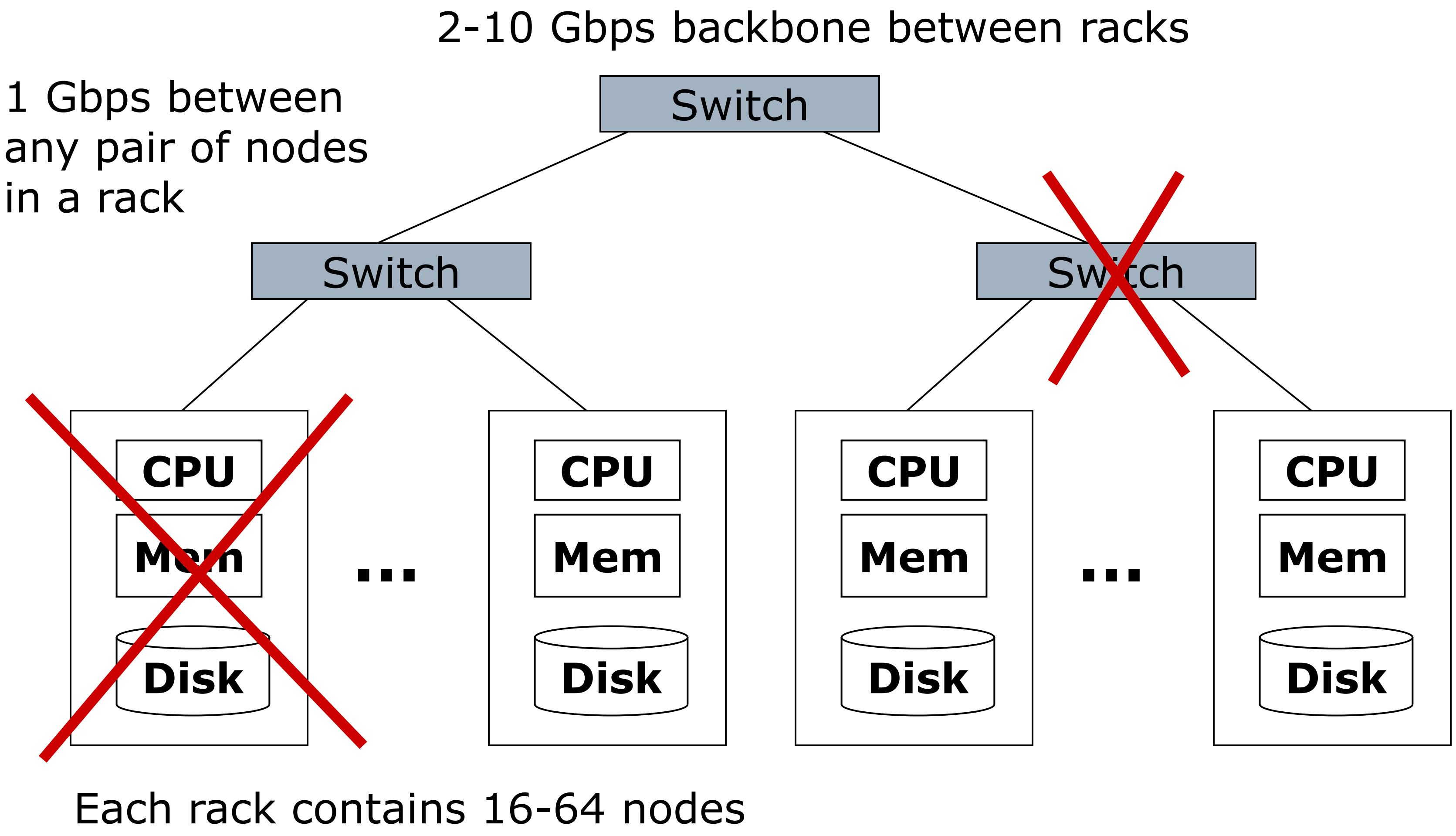
- Web data sets can be very large
  - Tens to hundreds of terabytes or petabytes
- Cannot mine on a single server (why?)
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- How to organize computations?
- How to handle hardware failures?

Google’s dilemma:
Computer cluster vs. Supercomputer ?



Source: wikipedia

Cluster Architecture



Stable storage

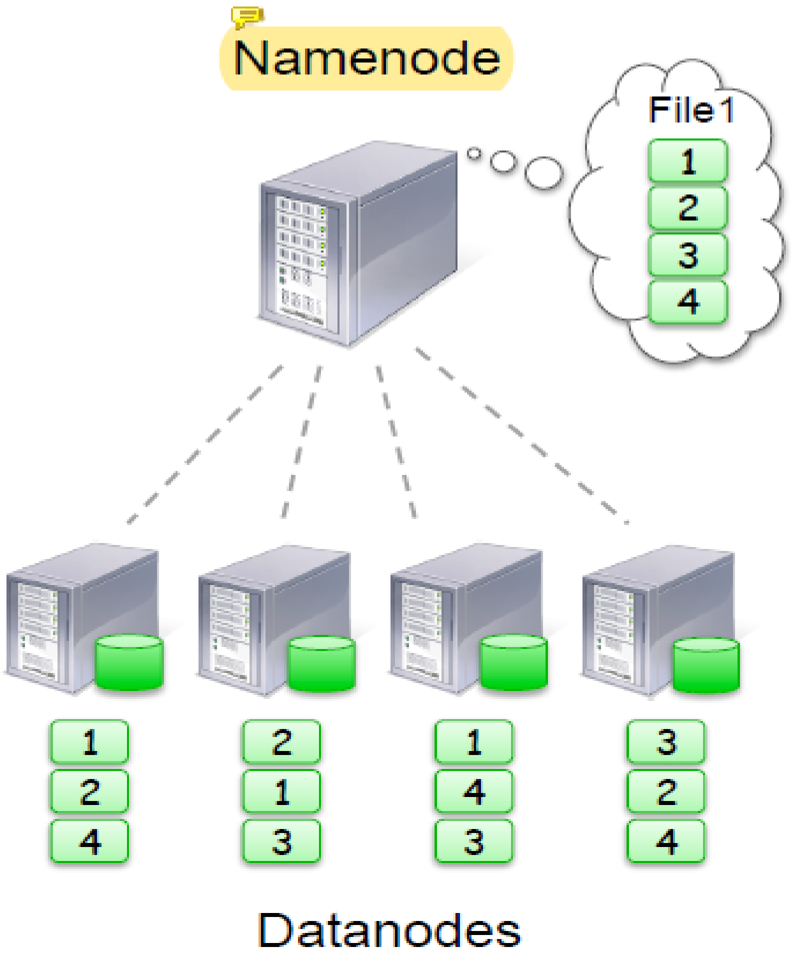
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common
  - No overwrites (!)

- Main problem: nodes can fail -> how can we prevent data loss?

- Answer: Distributed File System
  - Provides global file namespace; a dedicated namenode(s)
  - Chunks, replicas, hashes, self-monitoring/healing mechanism
  - Google GFS; Kosmix KFS**; Hadoop HDFS**

Distributed File System

- Files split into 128MB *blocks*
- Blocks replicated across several *datanodes* (usually 3)
- *Namenode* stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



Warm up: Word Count (1)

- Input:
  - a large file of words, one word per line
- Task:
  - count the number of times each distinct word appears in the file
- Sample application:
  - analyze web server logs to find popular URLs *(what is a webserver log?)*

127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)"

Word Count (2)

- Case 1: Entire file fits in memory (RAM) -> trivial

- Case 2: File too large for RAM but all <word, count> pairs fit in RAM -> trivial

- Case 3: File on disk, too many distinct words to fit in memory
  - `sort datafile | uniq -c`

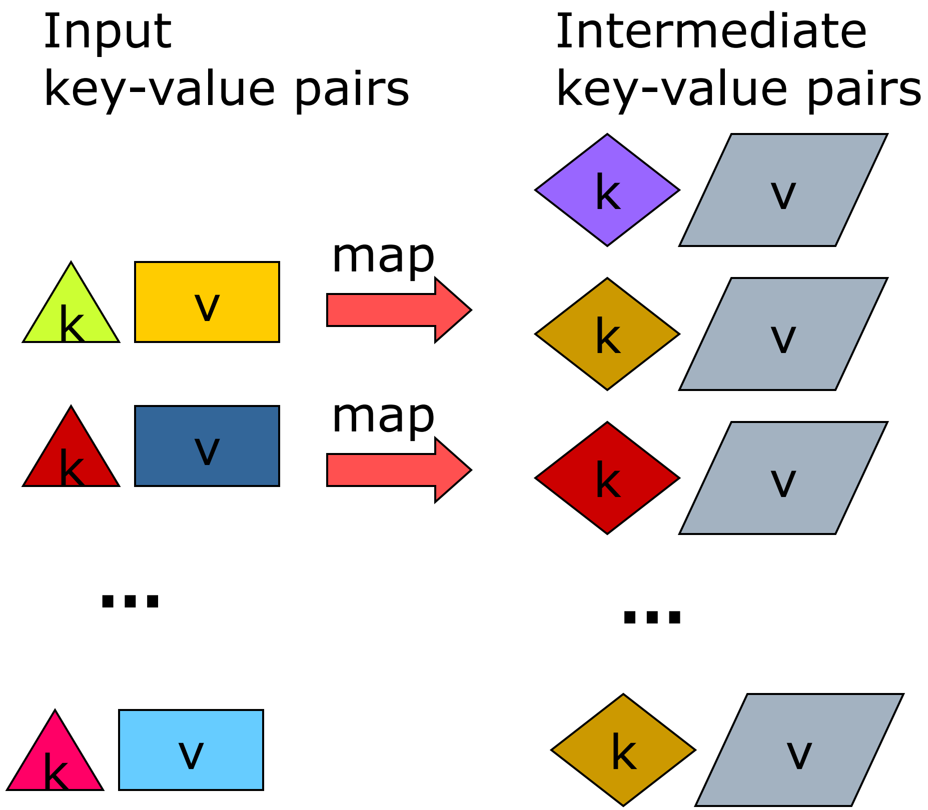
  Sorting on HD is slow but “doable” -> check how?

Word Count (3)

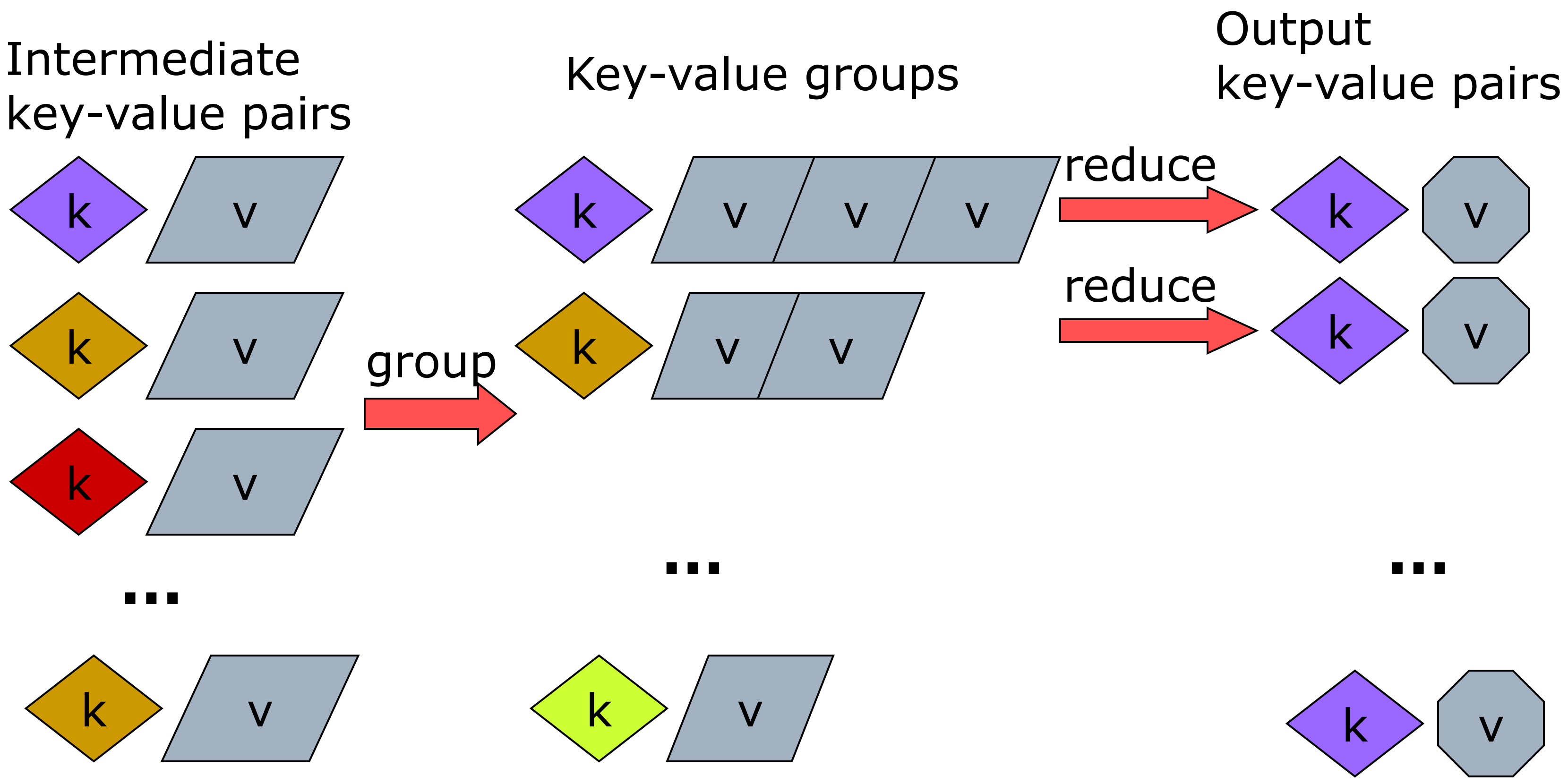
- To make it slightly harder, suppose we have a large corpus of documents
- Count the number of times each distinct word occurs in the corpus
  - `words(docs/*) | sort | uniq -c`
  - where `words` is a function that takes a file and outputs the words in it, one to a line
- The above captures the essence of MapReduce
  - Great thing is it is naturally parallelizable

Will not work when the harddisk is too small!

MapReduce: The Map Step



MapReduce: The Reduce Step



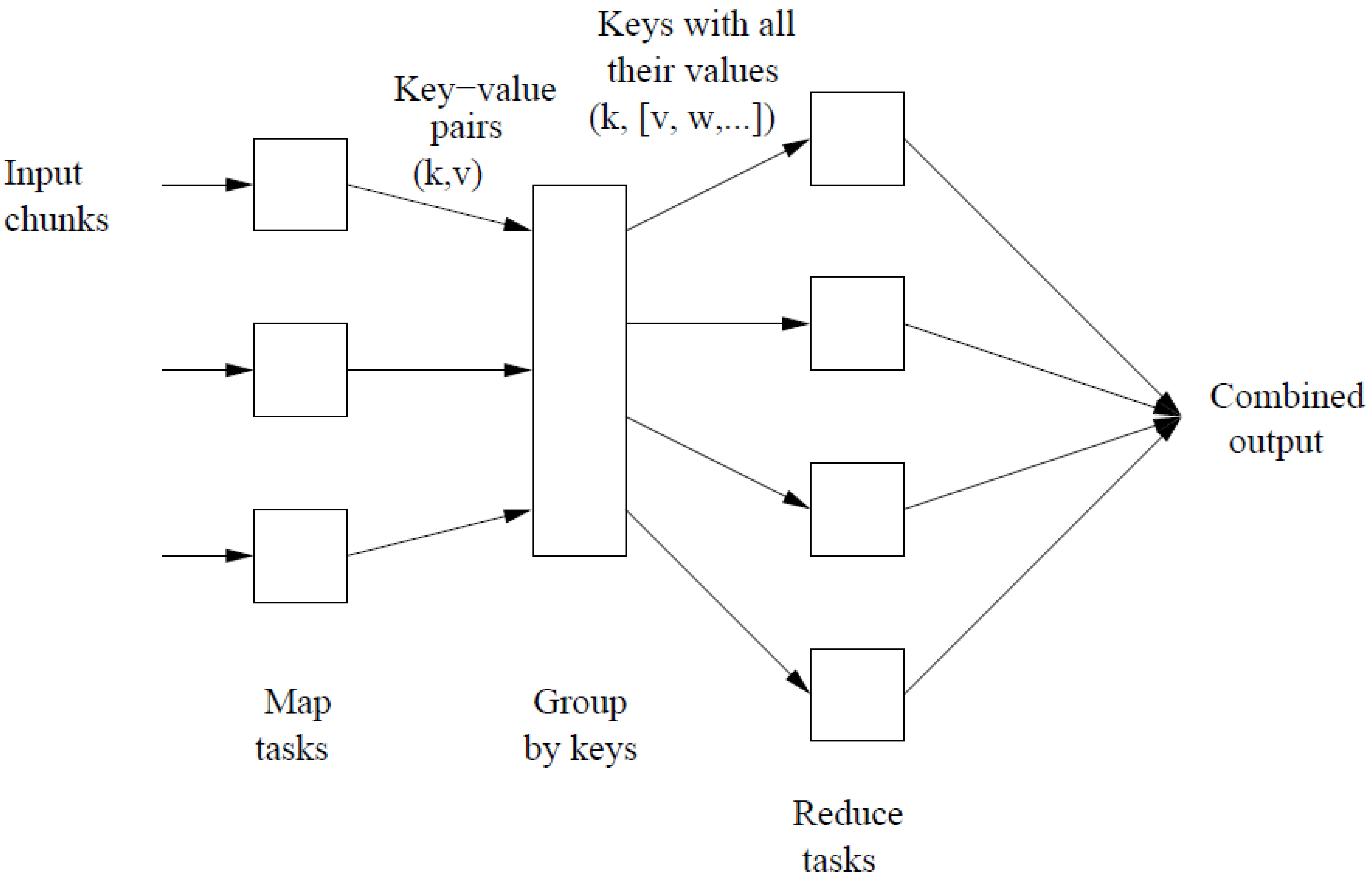


Figure 2.2: Schematic of a MapReduce computation

MapReduce: an abstract model

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

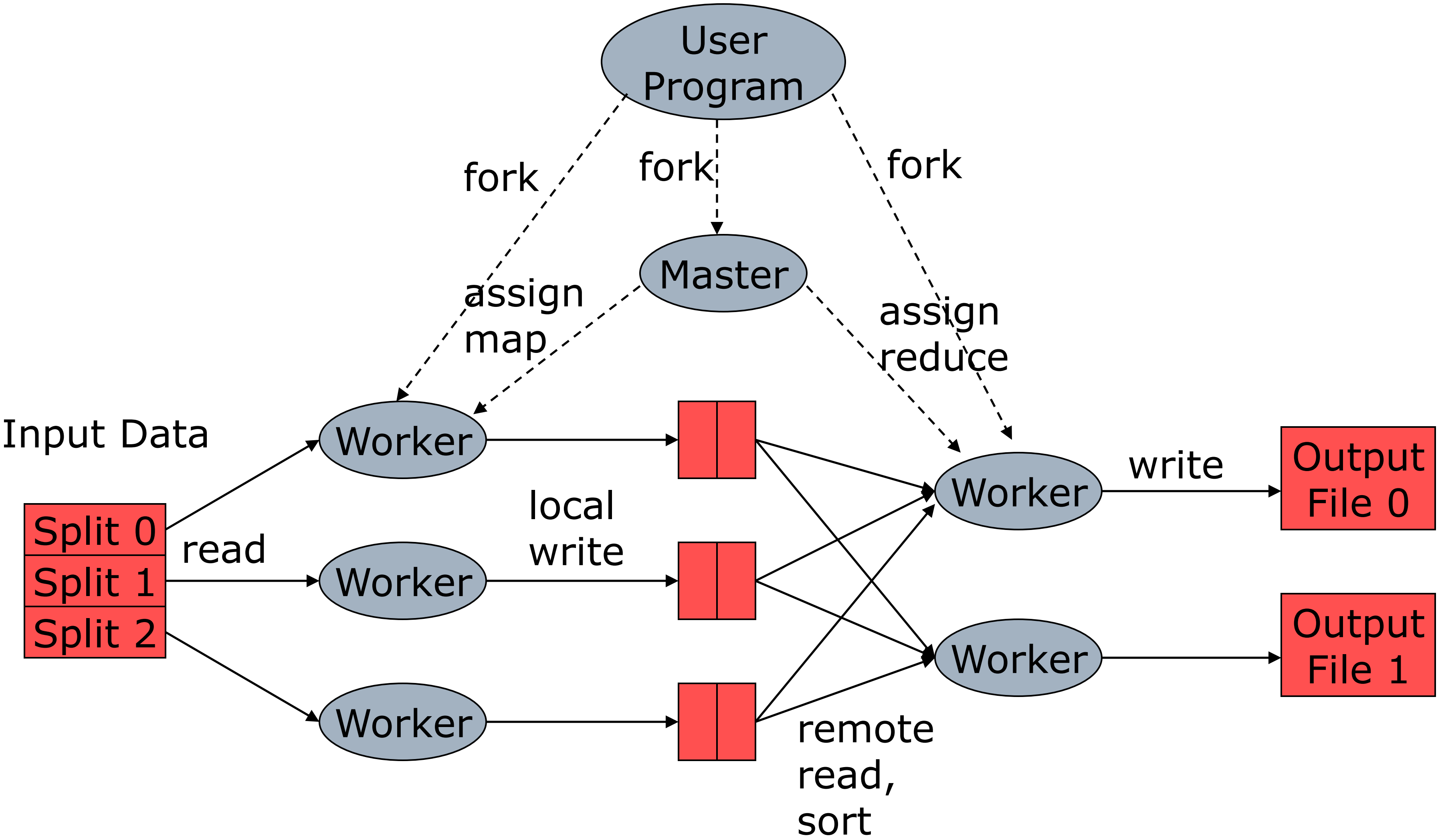
Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
        emit(w, 1) //write to a harddisk (hdfs)
```

"Magic": distribute & aggregate by keys [w, 1,1,1,1,1,…,1]

```
reduce(key, values):
// key: a word; values: an iterator over counts
        result = 0
        for each count v in values:
                result += v
        emit(result) //write to a harddisk (hdfs)
```

Distributed Execution Overview



Data flow

- Input, final output are stored on DFS (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

How many Map and Reduce jobs?

- M map tasks, R reduce tasks
- Rule of thumb:
  - Make M and R much larger than the number of nodes in cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
- Usually R is smaller than M, because output is spread across R files

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, *like addition:* (a+b=b+a) & (a+b)+c=a+(b+c)

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

Exercise 1: Host size

- Suppose we have a large web corpus
- Let’s look at the metadata file:
  - Lines of the form [URL, host, size, date, …]
- For each host, find the total number of bytes
  - i.e., the sum of the page sizes for all URLs from that host

Host size: Solution

- map(k,v) → list(k1,v1)
- reduce(k1, list(v1)) → (k1,v2)

- map(line_id, line_content)->[host, page_size]

- *[$host_1$, s_{11}, s_{12}, …], [$host_2$, s_{21}, s_{22}, s_{32},..],…*

- reduce(host, [sizes])->[host, sum(sizes)]

Exercise 2: Distributed Grep

- Find all occurrences of the given pattern in a very large set of files

Unix "grep" function":

grep searches the named input FILEs (or standard input) for lines containing a match to the given PATTERN (regular expression). By default, grep prints the matching lines.

- Example.:

[kowalczykwj@gold ~]$ history | grep das5

```
  512  ssh wojtek@fs1.das5.liacs.nl
  689  ssh das5
  690  history |grep das5
  837  ssh kowalczykwj@fs1.das5.liacs.nl
 1005  history | grep das5
```

Distributed Grep: solution

- map(document_id, document)
  ->[document_id, ‘grep output’]
  (if non-empty)

- reduce(document_id, [‘grep outputs’])
  ->[document_id, [‘grep outputs’]]

Exercise 3: Graph reversal

- Given a directed graph as an adjacency list:

  src1: dest11, dest12, …

  src2: dest21, dest22, …

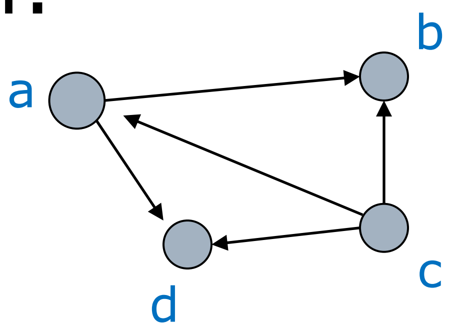
- Construct the graph in which all the links are reversed

Example

- Given a directed graph:

a: b,d

c: a,b,d

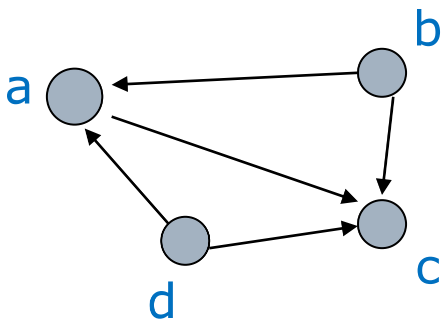


- Construct the graph in which all the links are reversed:

a: c

b: a, c

d: a, d



Graph reversal: solution

- Given a directed graph as an adjacency list:

  src1: dest11, dest12, …

  src2: dest21, dest22, …

  …. : ….

- map(src, [dest1, …, dest_k])
  ->[[dest1, src], [dest2, src], …, [dest_k, src]]

- reduce(dest, [src_1, …, src_n])
  ->[dest, [src_1, …, src_n]]

Exercise 4: Frequent Pairs

- Given a large set of market baskets, find all frequent pairs (i.e., frequency > threshold)

  - Remember definitions from Association Rules lectures:

    given a set of “itemsets” (transactions) T, a set of items is called frequent if it occurs in >s% of transactions (s is a parameter (support))

Frequent Pairs: Solution

- map(line_id, item_set) -> [(i1, i2), 1]
  (all possible pairs from the item_set)

- reduce((i1, i2), [1, 1, 1,…1])->
  if sum([1, …1]>threshold
  emit([(i1,i2), sum(1’s)]);

Exercise 5: Matrix-Vector Multiplication

- Given a large *nxn* matrix *M*, represented by a list of (i, j, m_{ij}) triplets and a vector v of length *n*, calculate *w=Mv*

$$w_i = \sum_j m_{ij} v_j$$

- Scenario 1: *M* huge, v small enough to be kept in RAM
- Scenario 2: both *M* and *v* too big to be kept in RAM
- Motivation: the PageRank algorithm!!!

Matrix-Vector Multiplication: Solution

- Scenario1: v fits in RAM

- map(line_id, (i, j, m_{ij}) -> $[i, m_{ij}v_j]$

- reduce(i, $[m_{i1}v_1, m_{i2}v_2, \ldots]$)
  -> (i,sum($m_{ij}v_j$))
  [the i-th element of Mv]

  M=million by million -> 8TB;
  v=million by 1 -> 8MB

Matrix-Vector Multiplication: Solution

- Scenario2: v does not fit in RAM
  split M into stripes of k columns and v into chunks of length k and apply algorithm from the previous slide, summing up the results:

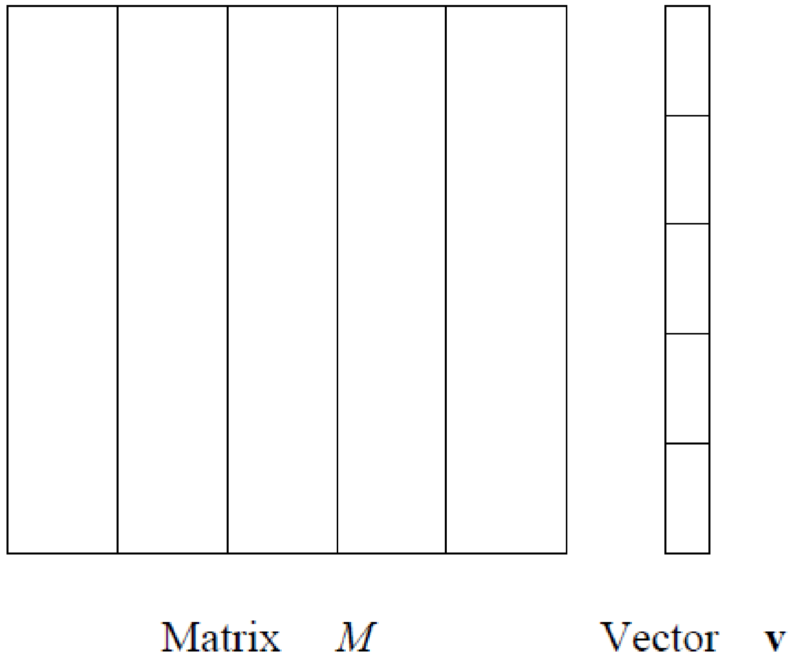


Figure 2.4: Division of a matrix and vector into five stripes

Exercise 6: Matrix-Matrix Multiplication

- Given two large matrices *M*, N calculate P=MN, given by:

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

- *input:*

  $(\text{“}M\text{”}, i, j, m_{ij})$ *or* $(\text{“}N\text{”}, j, k, n_{jk})$

Matrix-Matrix Multiplication: Solution

- *For each m_{ij} or n_{jk}:*
- *map((“M”, i, j, m_{ij})) -> [j, (“M”, i, m_{ij})]*
- *map(“N”, j, k, n_{jk}) -> [j,(“N”, k, n_{jk})]*

- *reduce(j, [(“M”, i, m_{ij}), (“N”, k, n_{jk})])*
  -> emit([(i,k), $m_{ij}n_{jk}$]), for all possible (i,k)

- *To get p_{ik}, run another MapReduce to sum up all the terms with key (i, k)!*

Limitations of MapReduce

- Usually a ***sequence of MapReduce operations*** is needed (complex SQL queries, construction of a decision tree, …)
- Very ***limited/limiting "conceptual basis"*** for programmers: thinking in terms of "Map" and "Reduce"
- We need ***"higher level constructs"*** that can be incorporated into the Hadoop/MapReduce framework …
- Performance: ***make better use of CPU's and RAM's!***
- ***==> Pig, Hive, Hbase, Mahout, Storm, Spark, …***
- www.cloudera.com/products/open-source/apache-hadoop.html

Building on Hadoop Map Reduce

Wojtek Kowalczyk

Hadoop and MapReduce

- Hadoop Distributed File System (hdfs)
  - chunks, replicas, “read-only”, “write/append once”
  - unlimited scalability (million of nodes)
  - very robust, can run on a cluster/grid/WAN

- MapReduce:
  - **Map**: “process chunks of data”
  - *shuffle and sort (implicit, pre-programmed)*
  - **Reduce**: “aggregate partial results”

Example algorithms in MapReduce

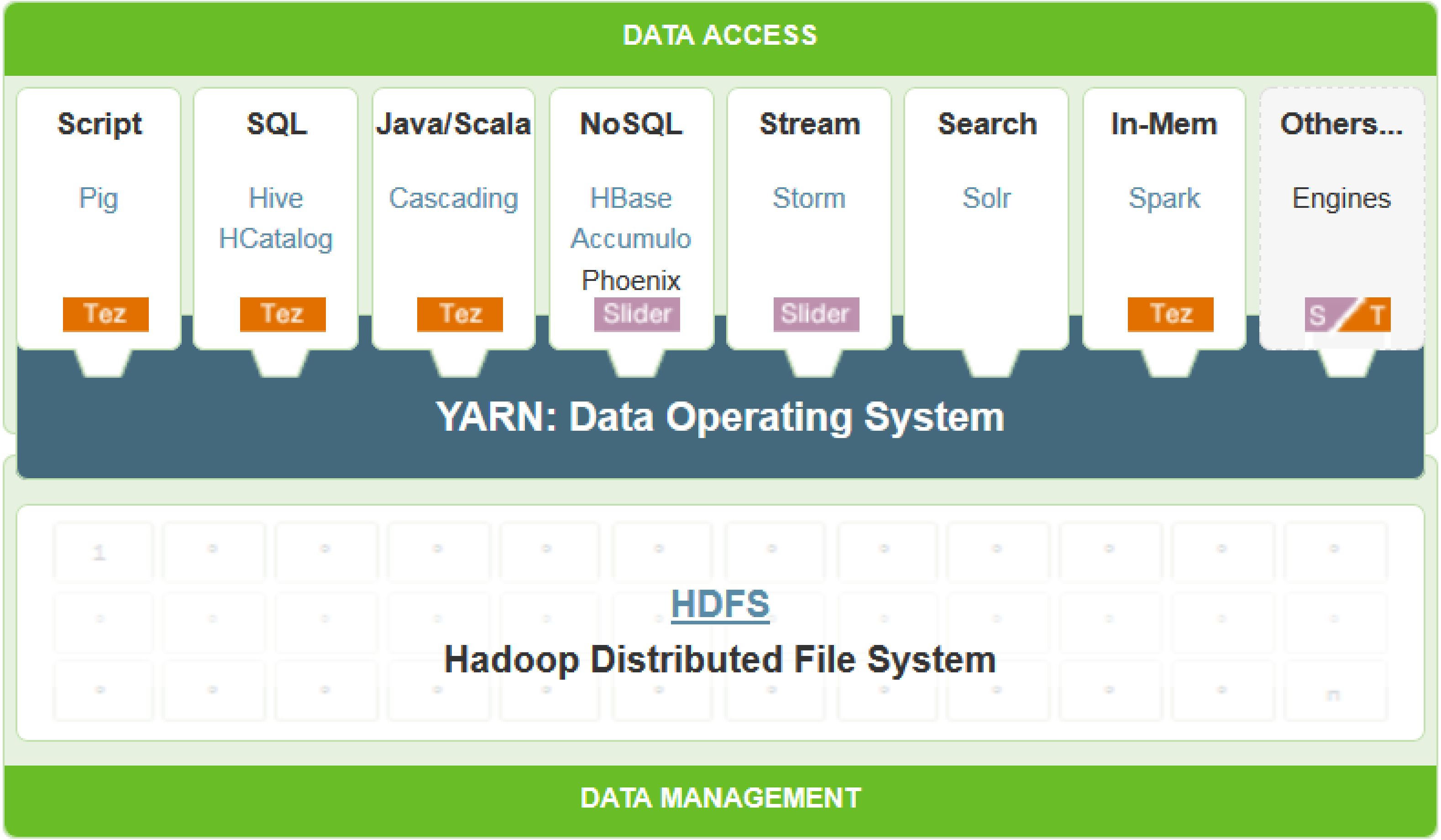
- word count
- distributed grep
- inverted index (documents -> “word index”)
- matrix multiplication
- PageRank
- atomic “database” operations: join, merge, group by, …
- Locality Sensitive Hashing (LSH)
- …
- *check the MMDS book!*

Limitations of MapReduce

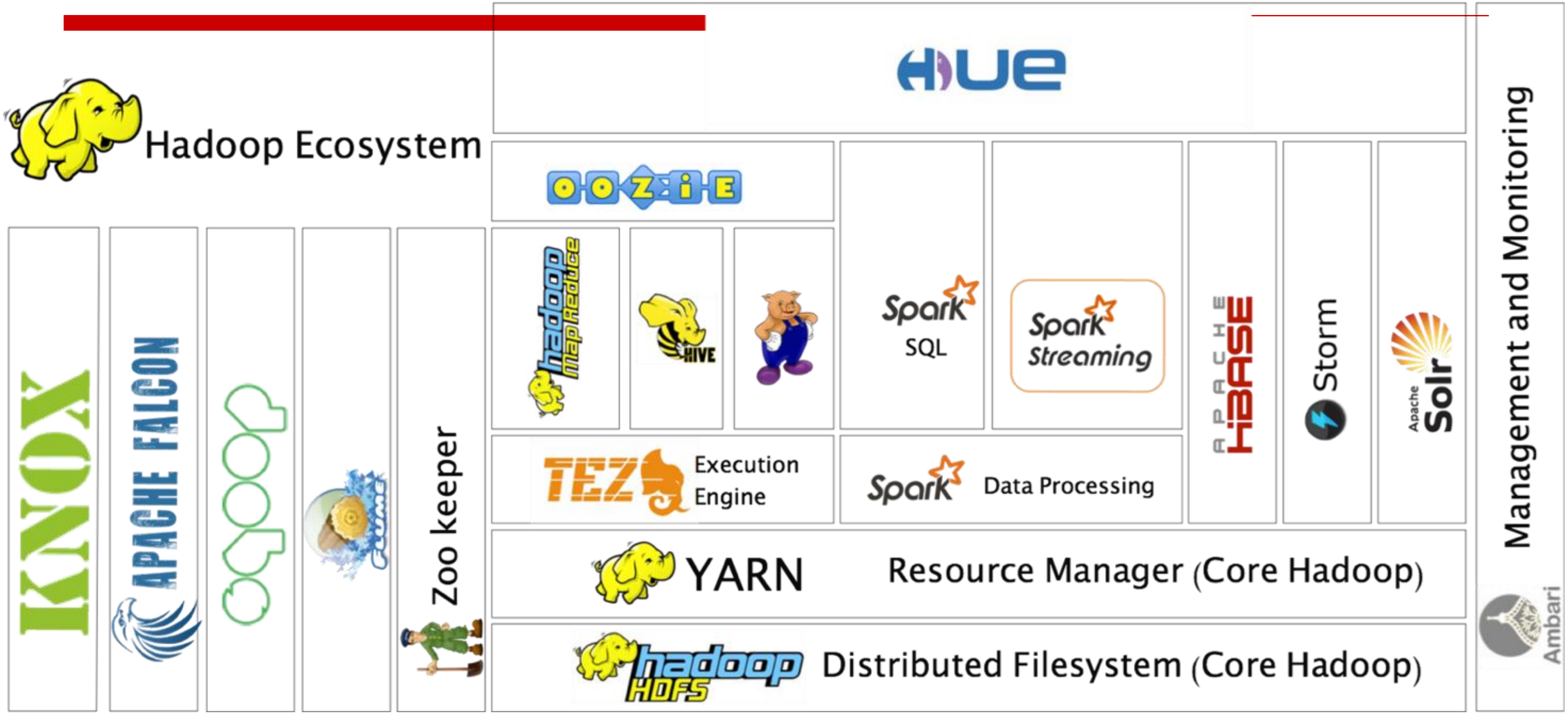
- Usually a ***sequence of MapReduce operations*** is needed (complex SQL queries, construction of a decision tree, …)
- Very ***limited/limiting "conceptual basis"*** for programmers: thinking in terms of "Map" and "Reduce"
- We need ***"higher level constructs"*** that can be incorporated into the Hadoop/MapReduce framework …
- Performance: ***make better use of CPU's and RAM's!***
- ***==> Pig, Hive, Hbase, Mahout, Storm, Spark, …***

Hadoop Data Platform (→ Cloudera.com)

http://hortonworks.com/hdp/



Hadoop Overview



Components

- **Core Hadoop Ecosystem**
  (definitions and use cases)

  - ❑ HDFS
  - ❑ YARN
  - ❑ Mesos
  - ❑ Zookeeper
  - ❑ MapReduce
  - ❑ Spark
  - ❑ Storm
  - ❑ Pig
  - ❑ Hive
  - ❑ Sqoop
  - ❑ Oozie
  - ❑ Kafka**
  - ❑ Flume
  - ❑ Flink
  - ❑ Ambari

- **Query Engines**
  - ❑ Drill
  - ❑ Phoenix
  - ❑ Presto
  - ❑ Hue
  - ❑ Zeppelin
  - ❑ * Impala

- **External Data Storage**
  - ❑ MySQL
  - ❑ HBase
  - ❑ Cassandra
  - ❑ MongoDB

https://spark.apache.org/

APACHE Spark™ *Lightning-fast unified analytics engine*

PySpark