HPC & Big Data

Adam S.Z Belloum
Software and Network engineering group
University of Amsterdam

Introduction to MapReduce programing model

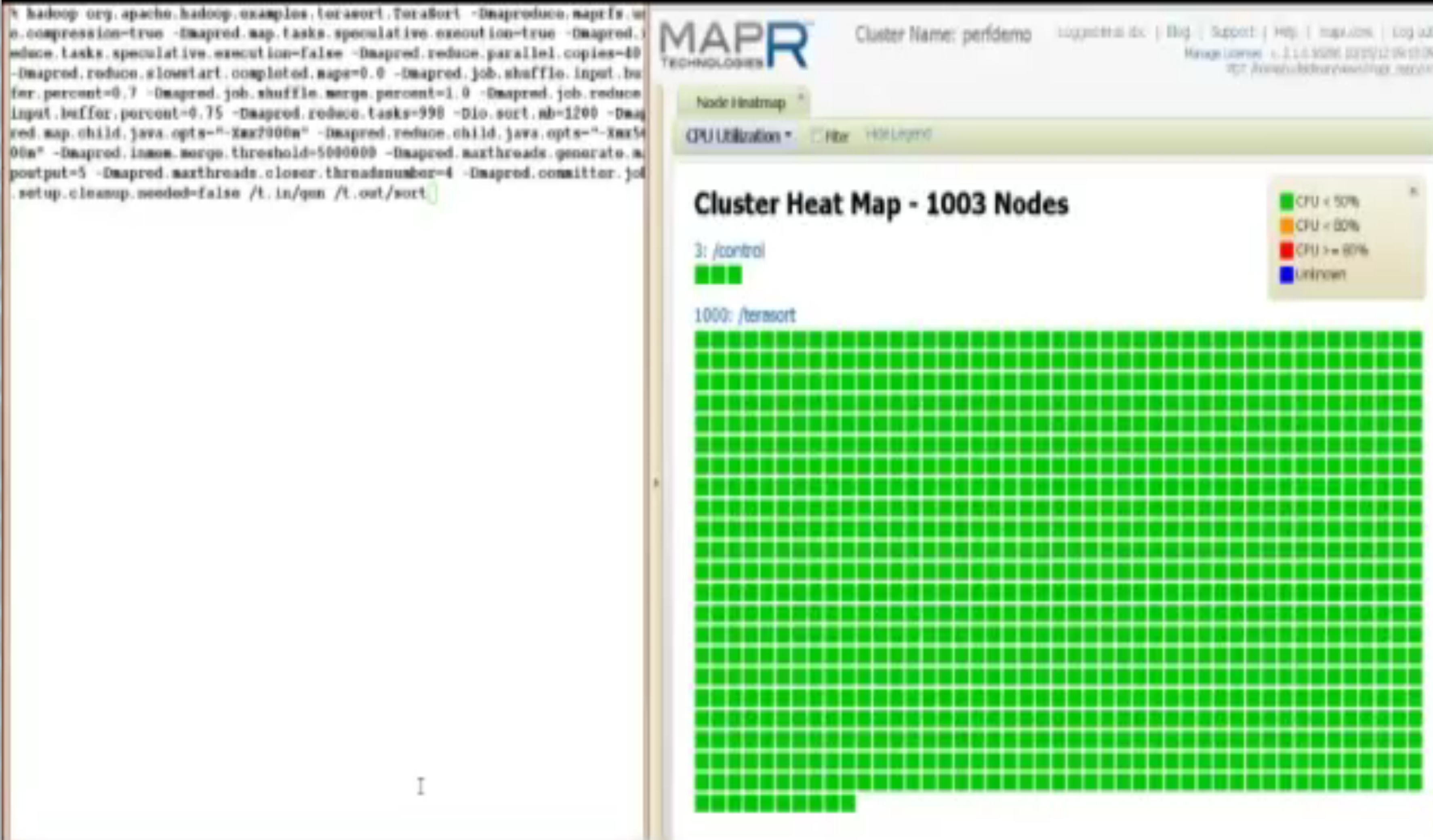
Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

Problem: lots of data

- Back-of-the-envelop estimate:
  - 20+ billion web pages x 20KB = 400+ terabytes
- One computer can read 30-35 MB/sec from disk
  - ~four months to **read** the web
- ~1,000 hard drives just to store the web
- **Even more to do something with the data**

http://ccgrid07.lncc.br/docs/industry_track/Google.pdf

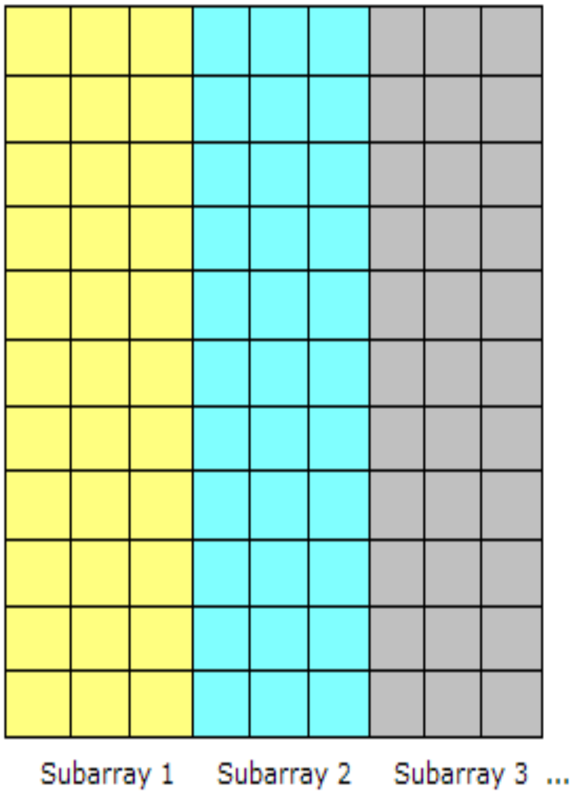


Cluster Name: perfdemo
Node Heatmap
CPU Utilization
Filter
Cluster Heat Map - 1003 Nodes
3: /control
1000: /terasort
CPU < 50%
CPU < 80%
CPU >= 80%
Unknown

same computations different dataset

A common situation

- a large amount of consistent data
- If the data can be **decomposed** into equal-size partitions,
- we can devise a parallel solution.
- If for each array element**,**
  - **no dependencies** in the computations,
  - and **no communication** are required between tasks



http://code.google.com/edu/parallel/mapreduce-tutorial.html#MapReduce

What is MapReduce?

- A programming model
  - Origin in functional programming like Lisp
  - (& its associated implementation)
- For processing **large data set**
- Exploits **large set** of **commodity computers**
- Executes process in distributed manner
- Offers high degree of transparencies

Example – Programming model

employees.txt

# LAST	FIRST	SALARY
Smith	John	$90,000
Brown	David	$70,000
Johnson	George	$95,000
Yates	John	$80,000
Miller	Bill	$65,000
Moore	Jack	$85,000
Taylor	Fred	$75,000
Smith	David	$80,000
Harris	John	$90,000
...	...	...
...	...	...

mapper

```
def getName (line):
   return  line.split(‘\t’)[1]
```

reducer

```
def addCounts (hist,  name):
   hist[name] = \
   hist.get(name,default=0) + 1
   return hist

input = open(‘employees.txt’, ‘r’)

intermediate = map(getName, input)

result = reduce(addCounts, \
            intermediate, {})
```

Q: “What is the frequency of each first name?”

Example – Programming model

employees.txt

# LAST	FIRST	SALARY
Smith	John	$90,000
Brown	David	$70,000
Johnson	George	$95,000
Yates	John	$80,000
Miller	Bill	$65,000
Moore	Jack	$85,000
Taylor	Fred	$75,000
Smith	David	$80,000
Harris	John	$90,000
...	...	...
...	...	...

Q: "What is the frequency of each first name?"

mapper

```
def getName (line):
    return (line.split('\t')[1], 1)
```

reducer

```
def addCounts (hist, (name, c)):
    hist[name] = \
    hist.get(name,default=0) + c
    return hist
```

```
input = open('employees.txt', 'r')

intermediate = map(getName, input)

result = reduce(addCounts, \
                intermediate, {})
```

Key-value iterators

Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

The Master/Worker approach

- The MASTER:
  - **initializes** the array and **splits** it up according to the number of available WORKERS
  - **sends** each WORKER its subarray
  - **receives** the results from each WORKER

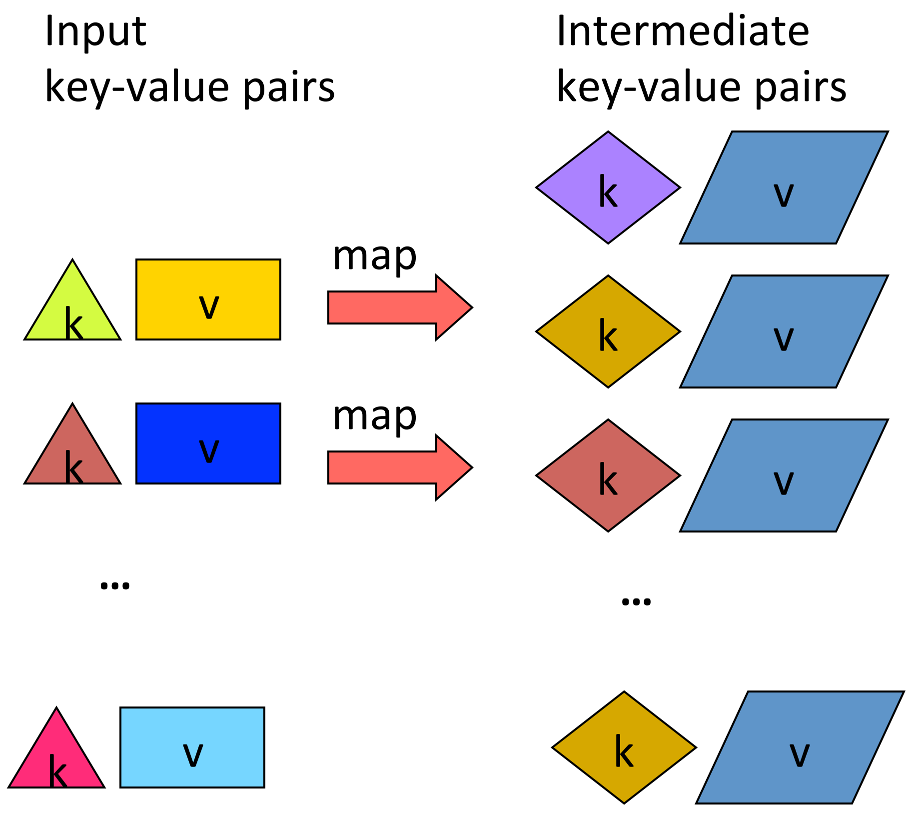
- The WORKER:
  - **receives** the subarray from the MASTER
  - **performs** processing on the subarray
  - **returns** results to MASTER

http://code.google.com/edu/parallel/mapreduce-tutorial.html#MapReduce

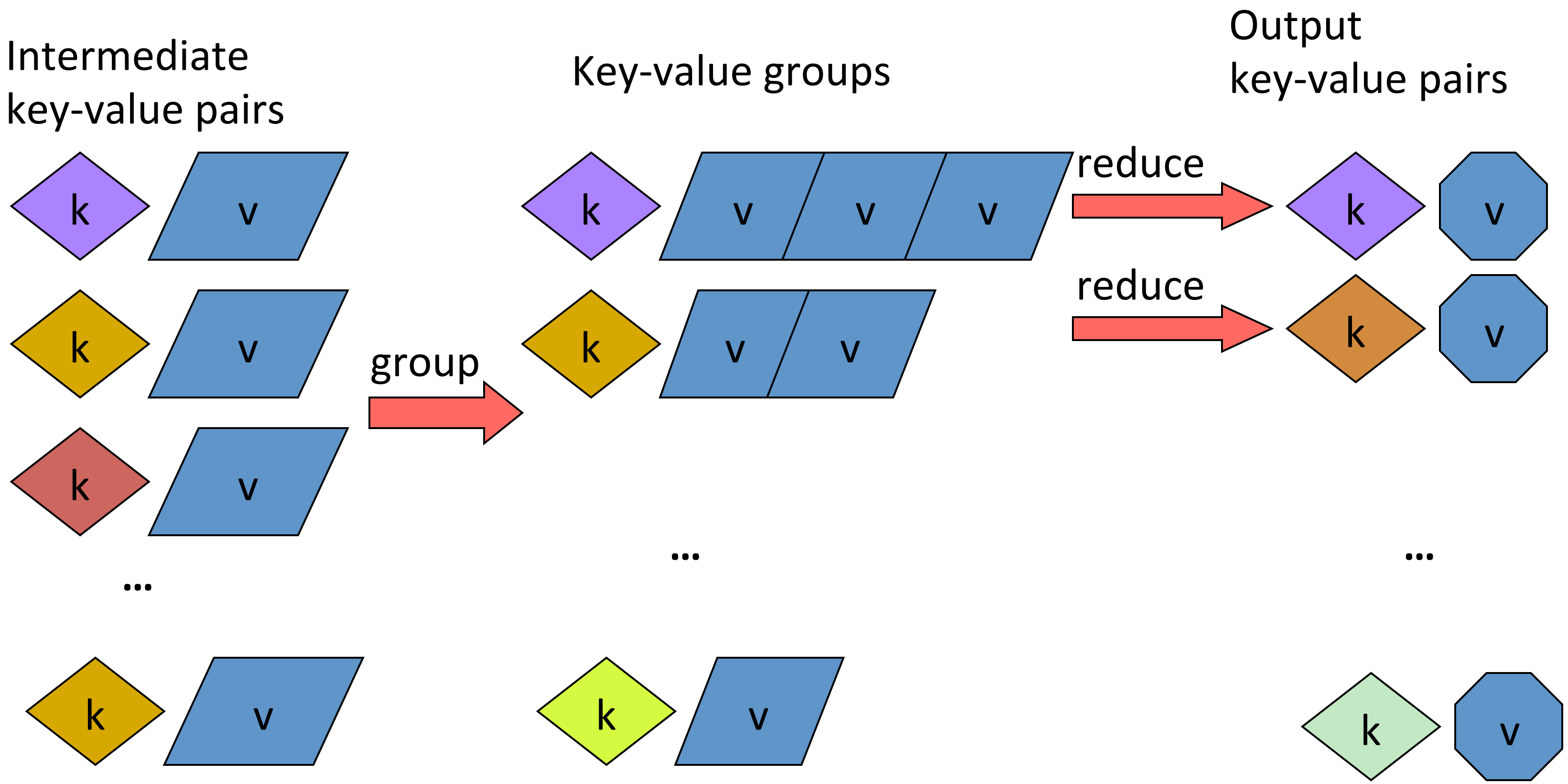
MapReduce

- Workers are assigned
  - A map function
  - A reduce function
- 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

MapReduce: The Map Step



MapReduce: The **Reduce Step**



Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

MapReduce Examples

- **Distributed `grep`**
- **Count of URL Access Frequency**
- **Reverse Web-Link Graph**
- **Term-Vector per Host**
- **Inverted Index**

Inverted Index:

- map function parses each document, and emits a sequence of <word, document ID>
- reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> . The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions

MapReduce Examples in Science

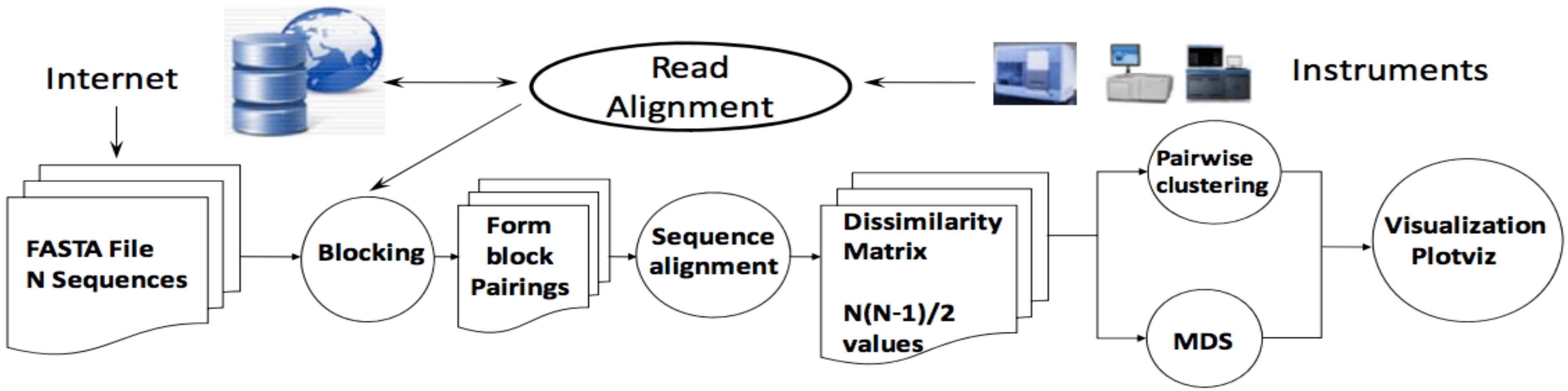


Figure 1 Pipeline for analysis of metagenomics Data

Clouds and MapReduce for Scientific Application

http://grids.ucs.indiana.edu/ptliupages/publications/CloudsandMR.pdf

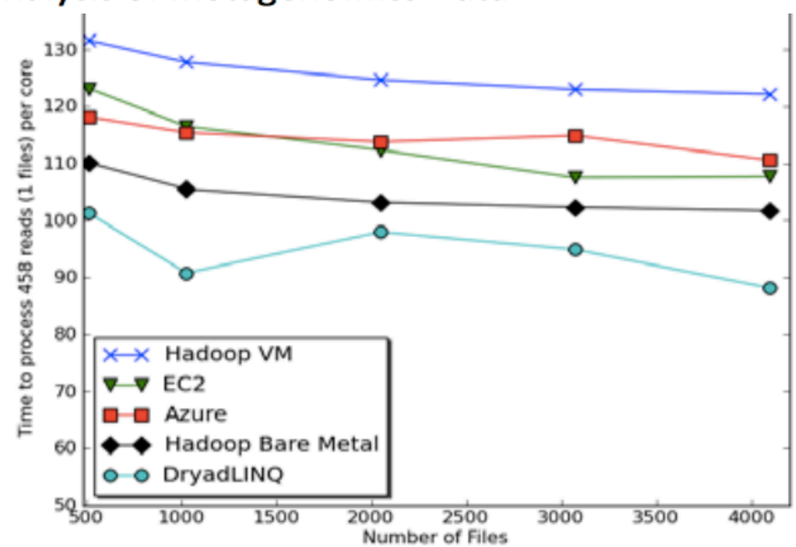


Figure 3: Time to process a single biology sequence file (458 reads) per core with different frameworks[18]

Example: Word Count (1)

- We have a large file of words, one word to a line
- **Count** the number of times each **distinct** word appears in the file
- Sample application:
  - analyze web server logs to find popular URLs

Example: Word Count (2)

- Case 1: Entire file fits in **memory**
- Case 2: File **too large** for mem, but all

  <word, count> pairs fit in mem
- Case 3: File on disk, **too many** distinct words to fit in memory

```
$ sort datafile | uniq –c
```

Example: 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

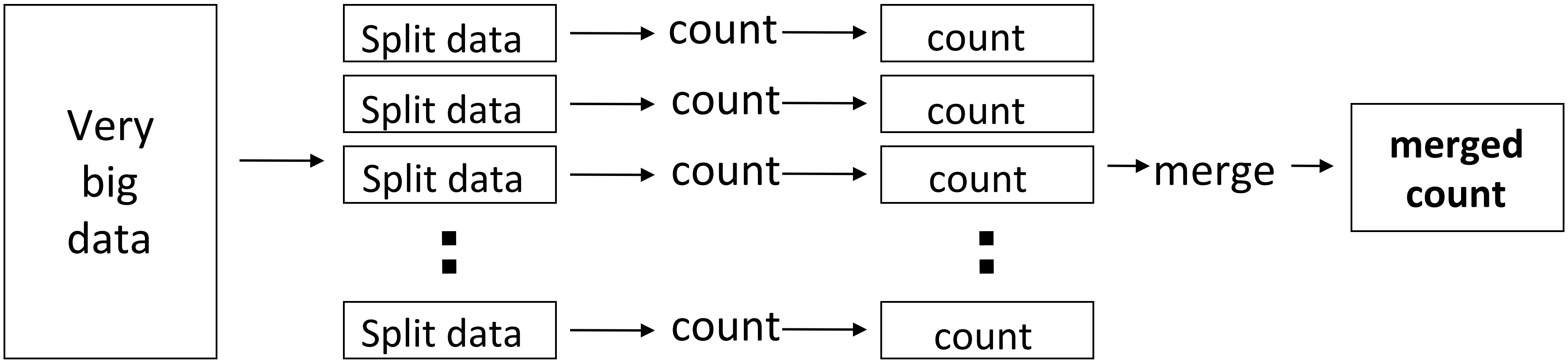
```
$ words(docs/*) | sort | uniq -c
```

  – where **`words`** 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

Distributed Word Count



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;                value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(result)
```

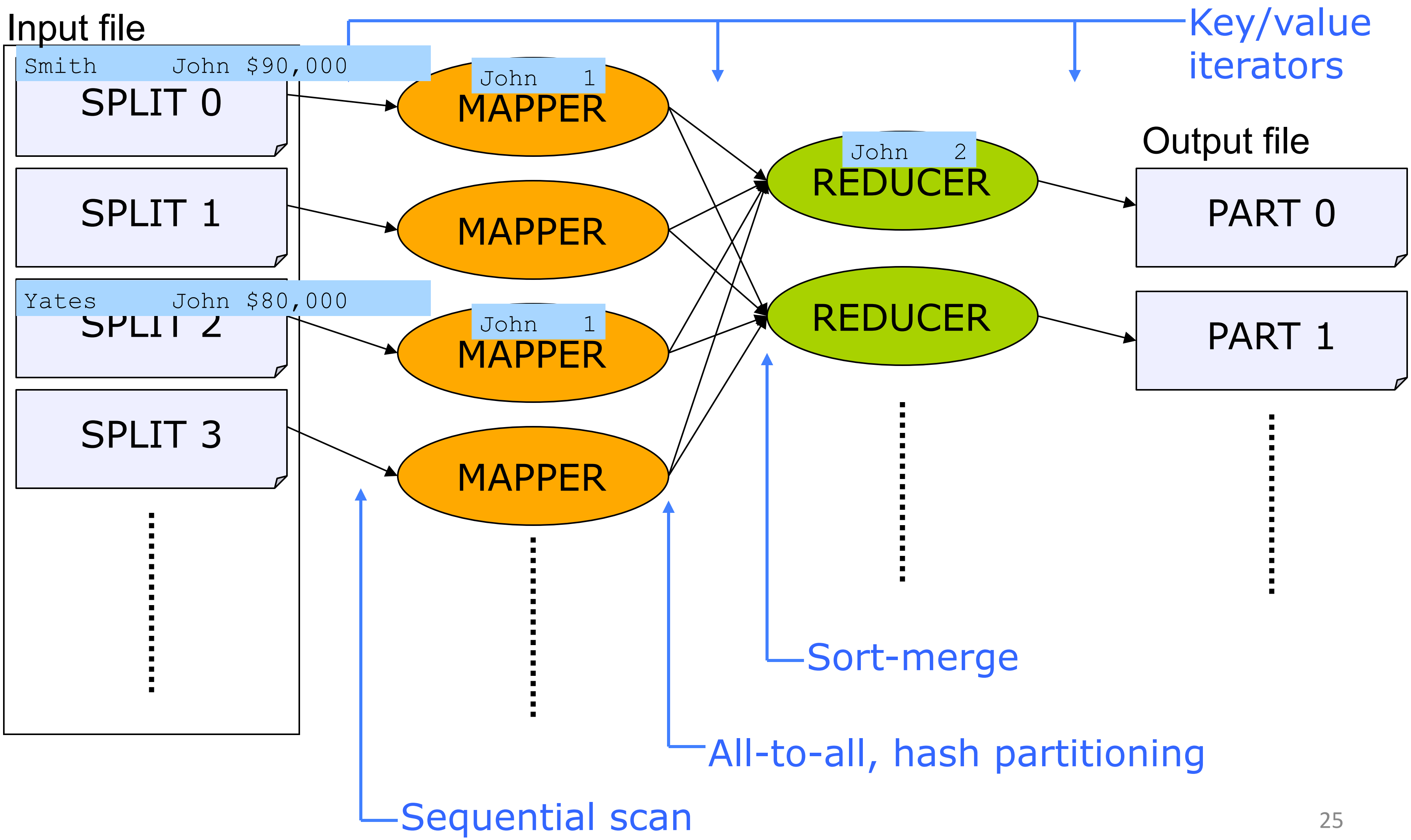
Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

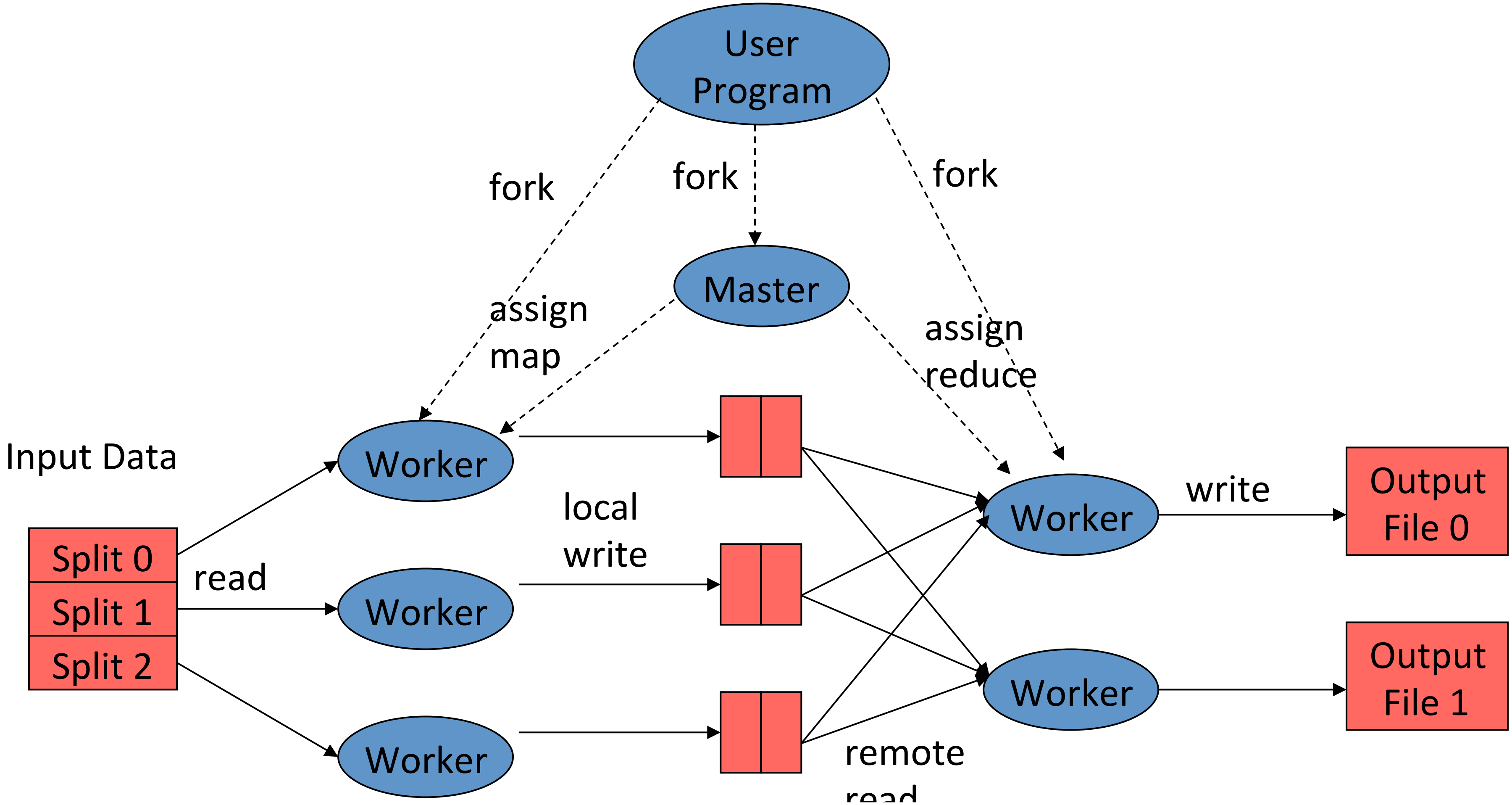
Data flow

- **Input**, **final output**
  - are stored on **a distributed file system**
    - Haadoop distributed Filesystem (HDFS) → (googleFileSystem)
  - 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
    - Haadoop local Filesystem (HFS or FS)
- Output is often input to another MapReduce task

Execution model: Flow



Distributed Execution Overview



7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

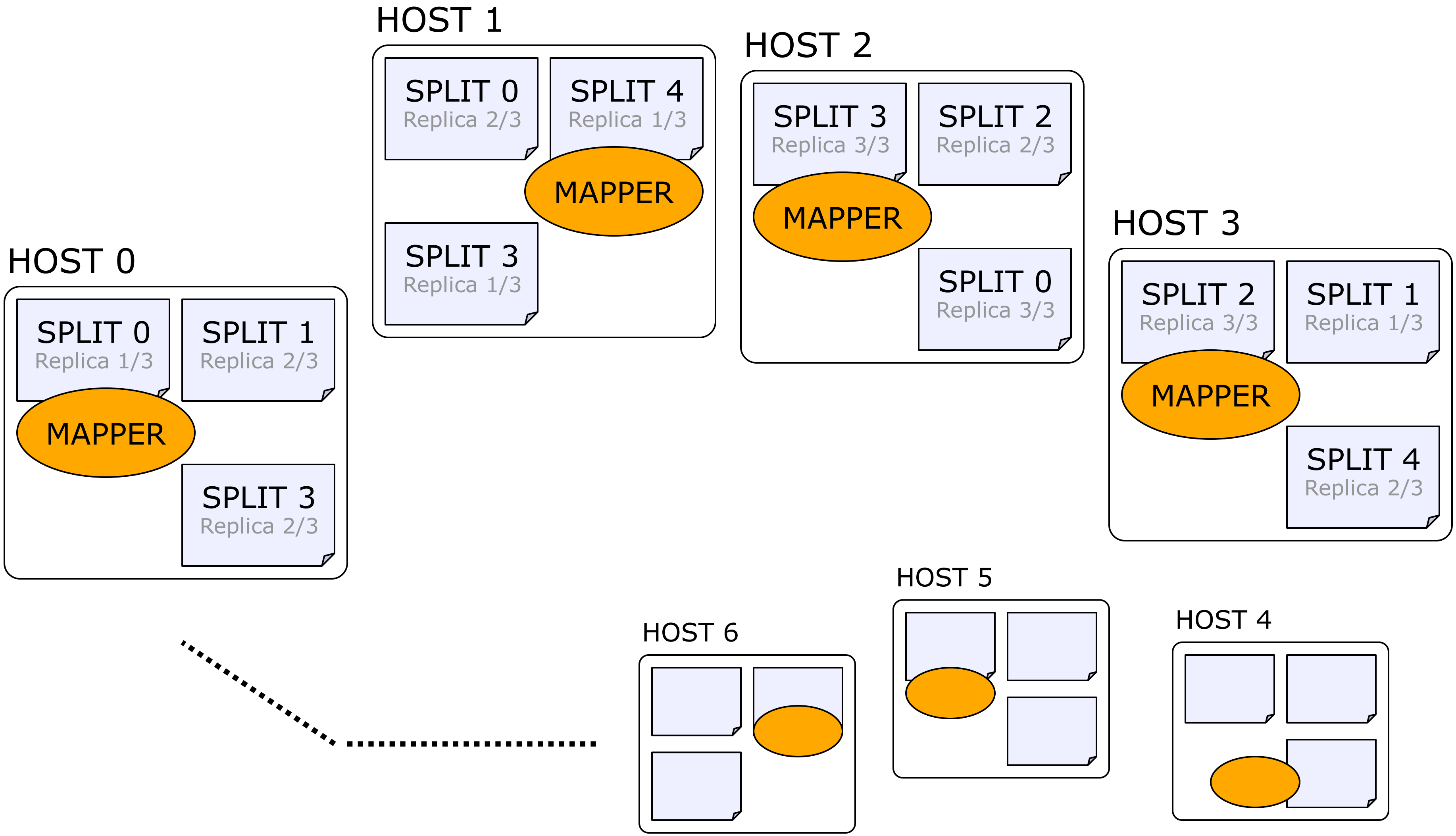
How many Map and Reduce jobs?

- 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

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

Execution model: Placement

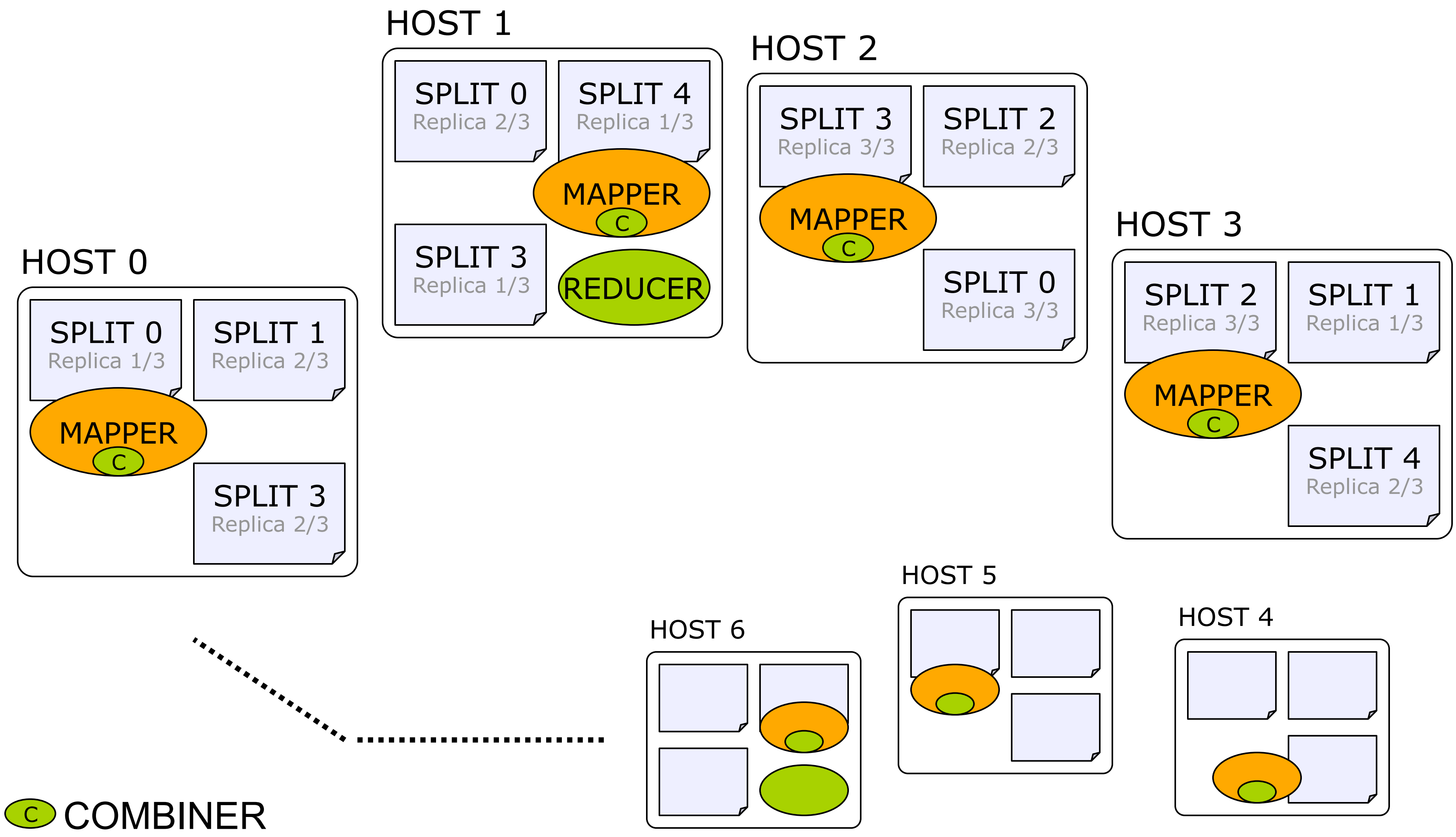


Computation *co-located* with data (as much as possible)

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

Execution model: Placement



Rack/network-aware

Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

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

Content

- Introduction
- Master/Worker approach
- MapReduce Examples
- Distributed Execution Overview
- Data flow
- Coordination
- Failure
- Partitioning function
- Hadoop (example of implementation)

Partition Function

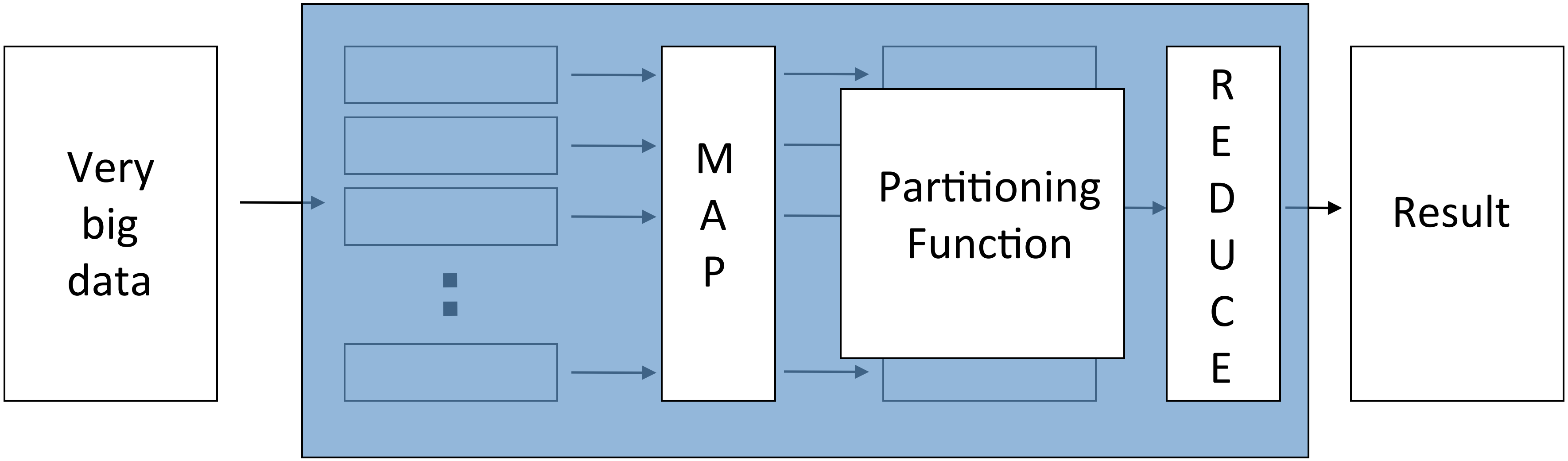
- **Inputs** to **map tasks**
  - are created by contiguous splits of input file

- 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`

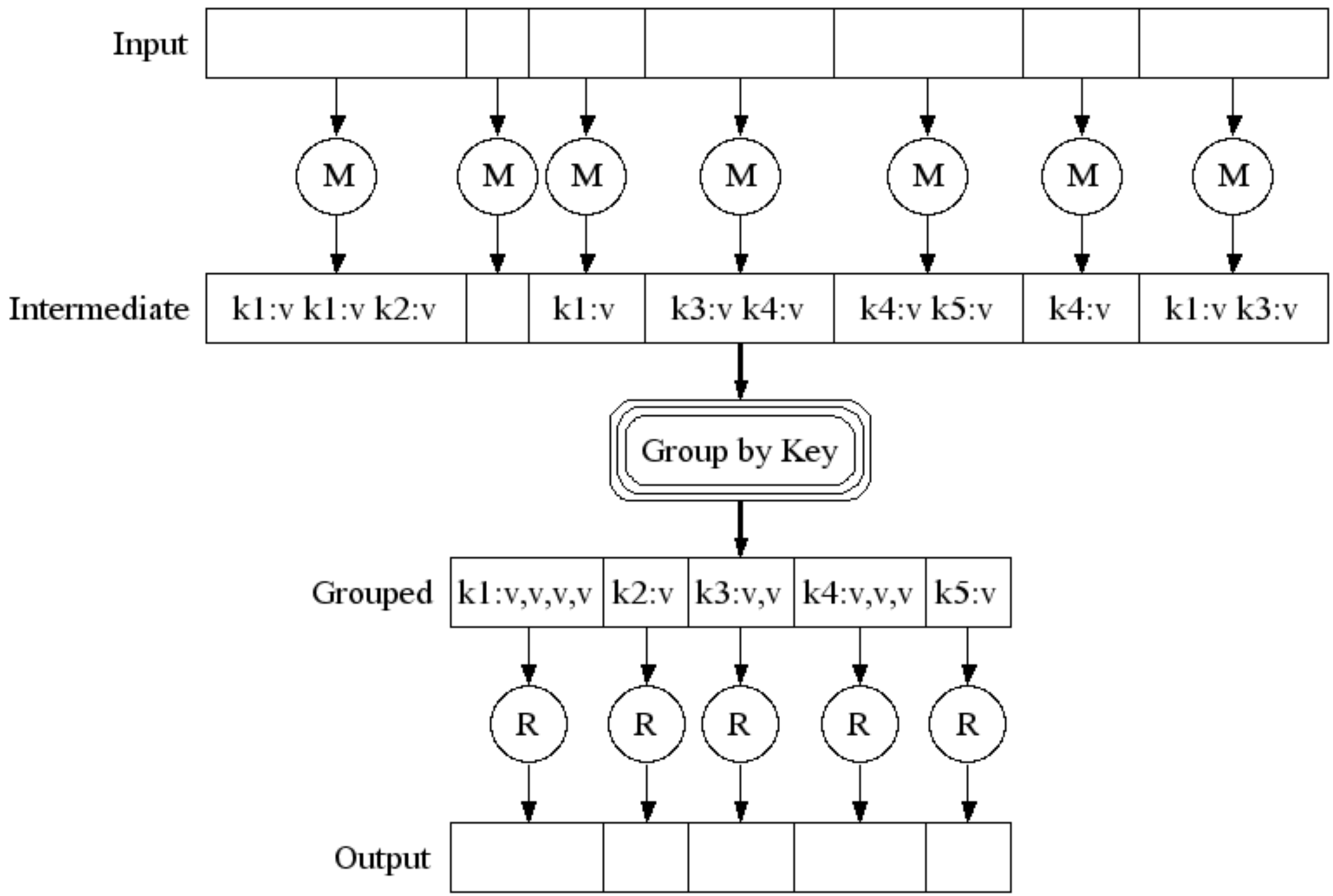
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

Map Reduce



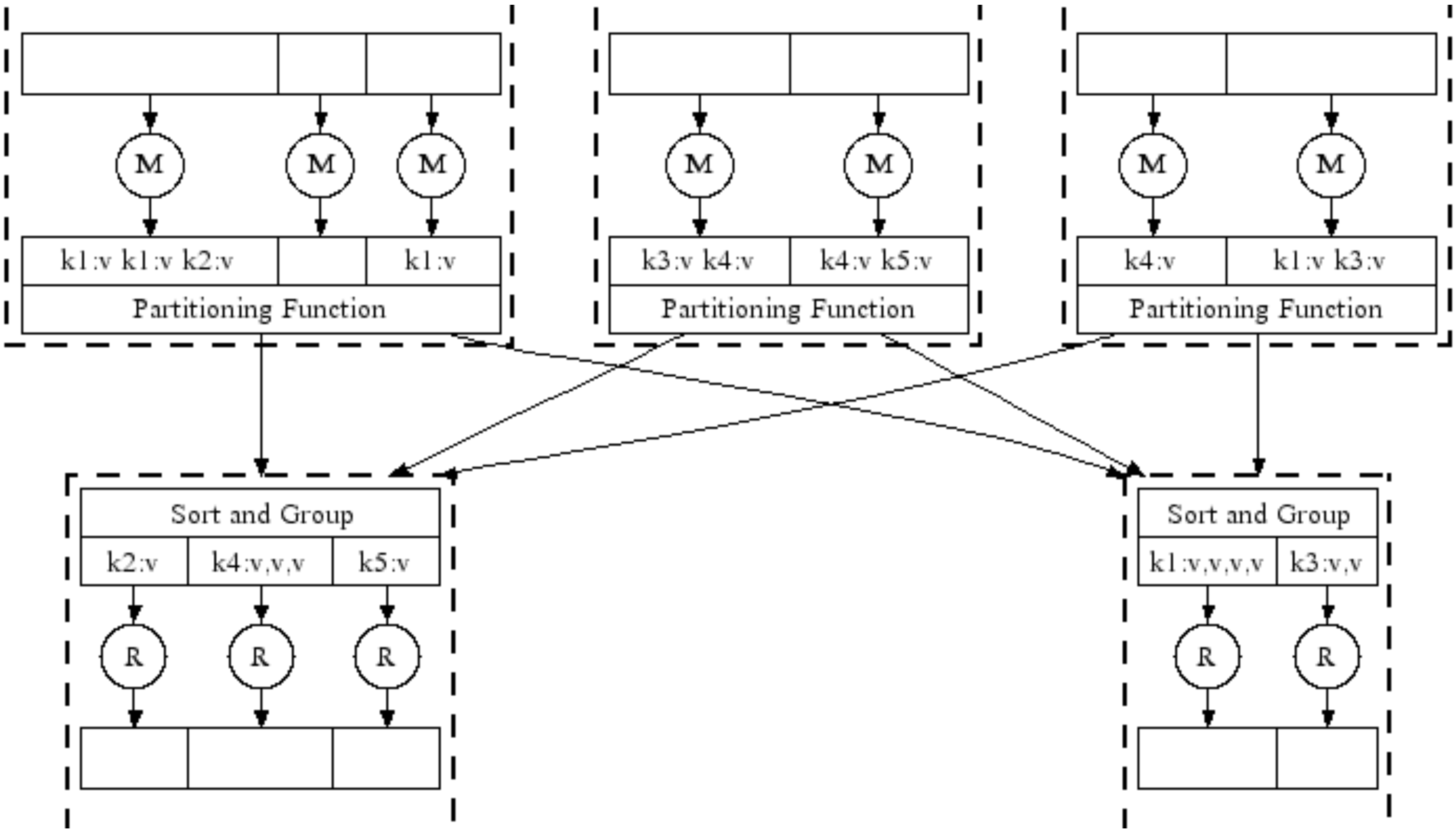
- **Map**:
  - Accepts
    - *input* key/value pair
  - Emits
    - *intermediate* key/value pair
- **Reduce** :
  - Accepts
    - *intermediate* key/value* pair
  - Emits
    - *output* key/value pair

Partitioning Function



http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0007.html

Partitioning Function



http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0008.html

Partitioning Function (2)

- Default : `hash(key) mod R`
- Guarantee:
  - Relatively well-balanced partitions
  - Ordering guarantee within partition
- Distributed Sort
  - Map:

    ```
    emit(key,value)
    ```
  - Reduce:

    ```
    emit(key,value)
    ```

MapReduce

- Distributed Grep
  - Map:

```
if match(value,pattern) emit(value,1)
```

  - Reduce:

```
emit(key,sum(value*))
```

- Distributed Word Count
  - Map:

```
for all w in value do emit(w,1)
```

  - Reduce:

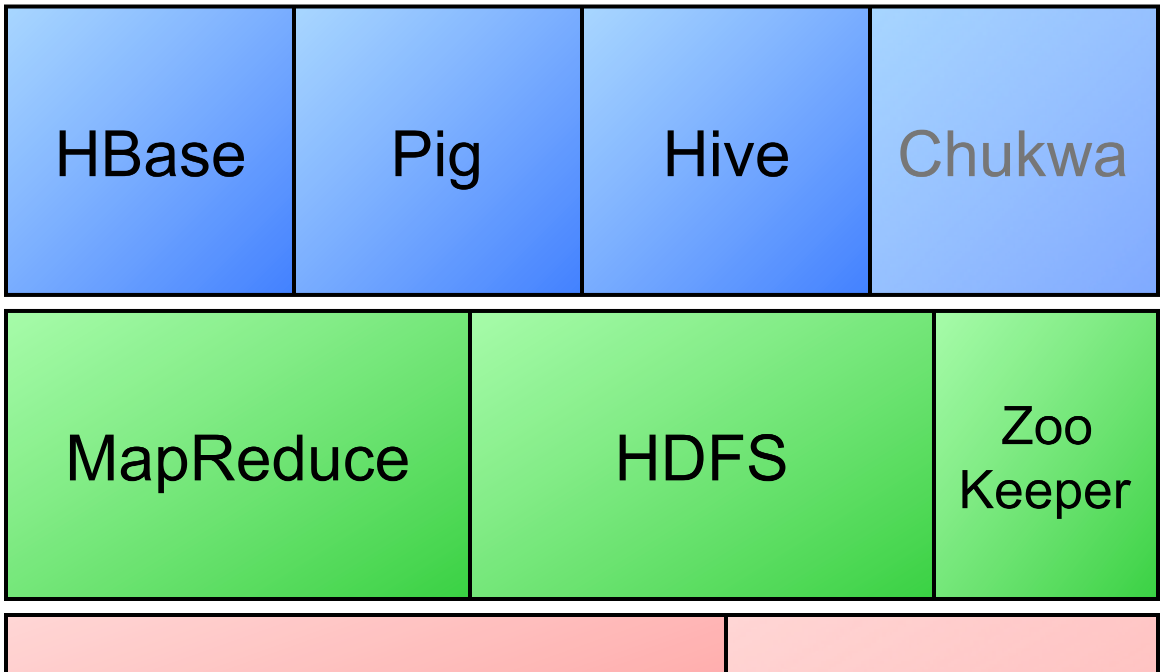
```
emit(key,sum(value*))
```

MapReduce outside Google

- Hadoop (Java)
  - Emulates MapReduce and GFS
  - The architecture of Hadoop MapReduce and DFS is master/slave

	Master	Slave
MapReduce	jobtracker	tasktracker
DFS	namenode	datanode

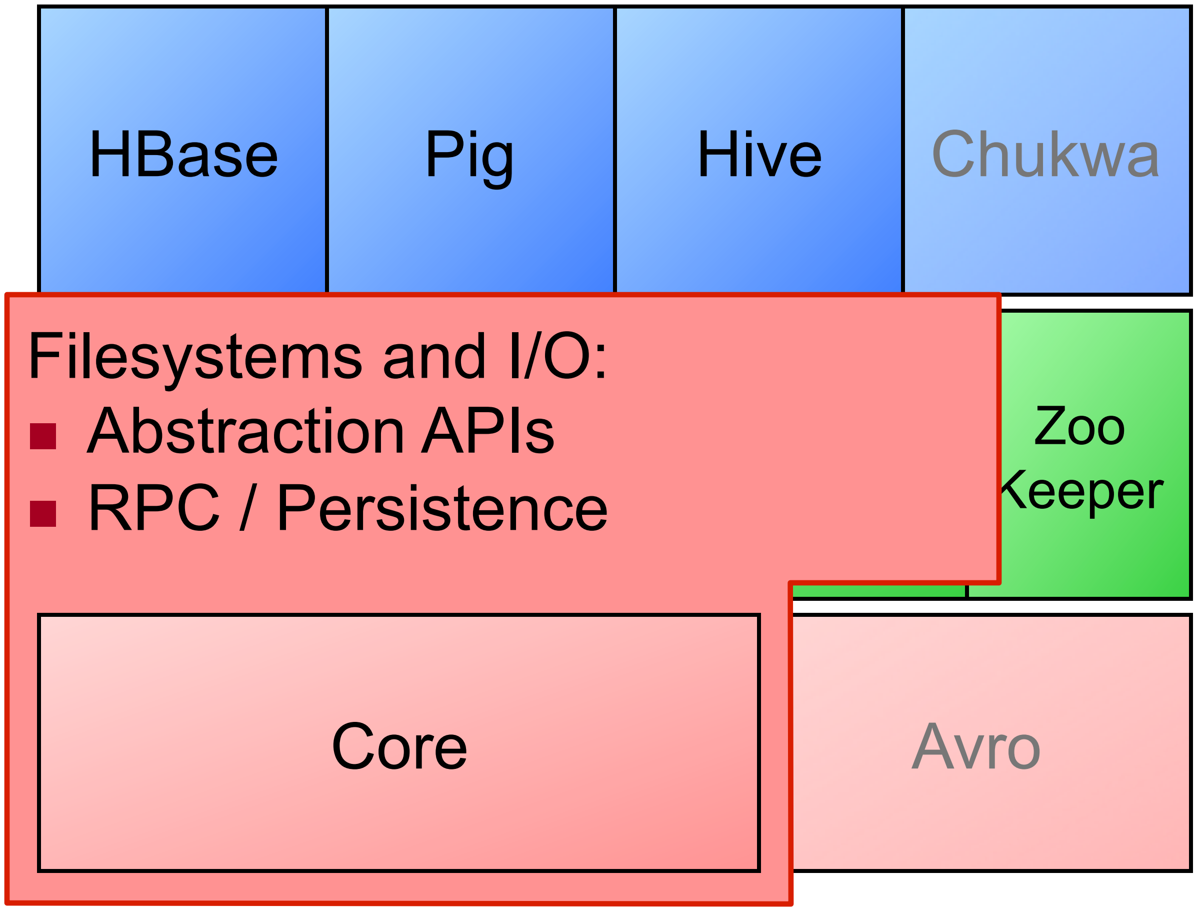
Hadoop



Hadoop's stated mission (Doug Cutting interview):

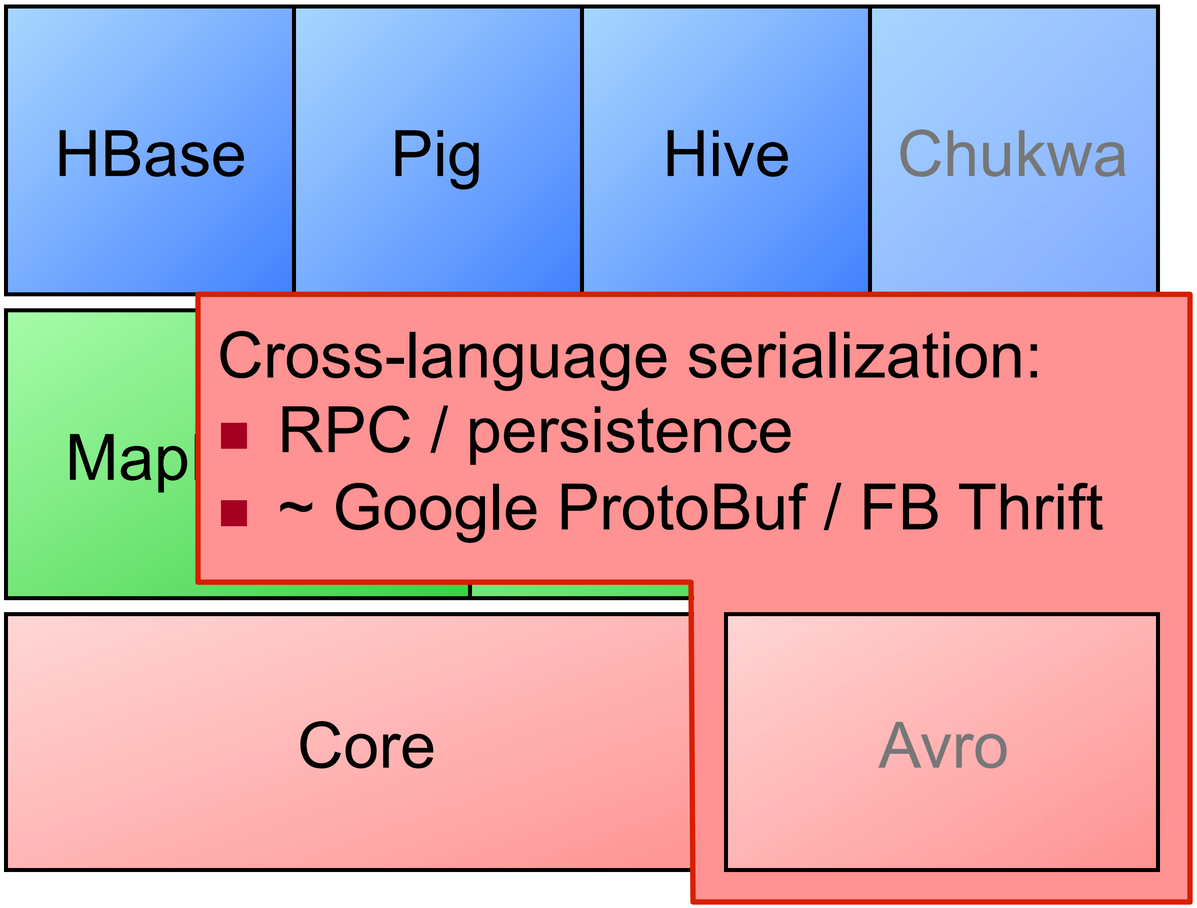
Commoditize infrastructure for web-scale, data-intensive applications

Hadoop

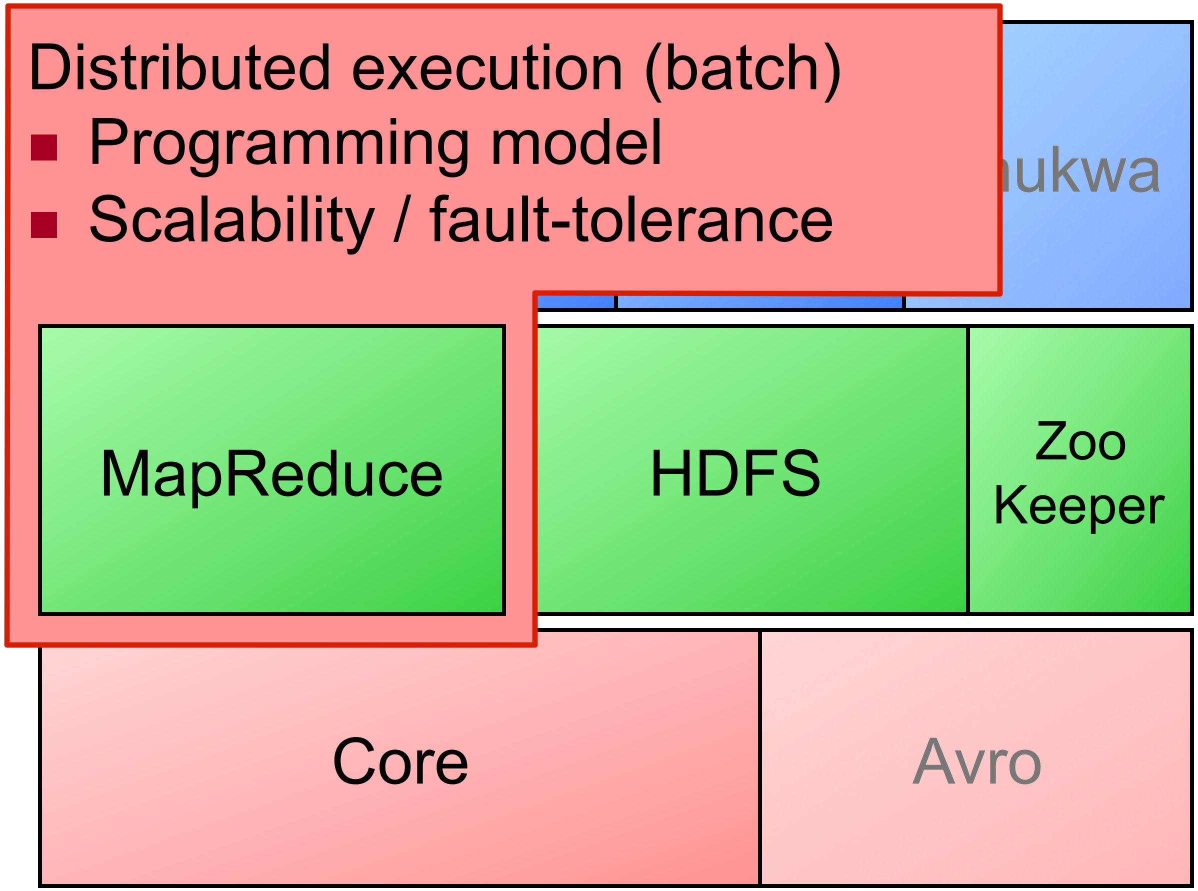


HBase
Pig
Hive
Chukwa
Filesystems and I/O:
Abstraction APIs
RPC / Persistence
Zoo
Keeper
Core
Avro

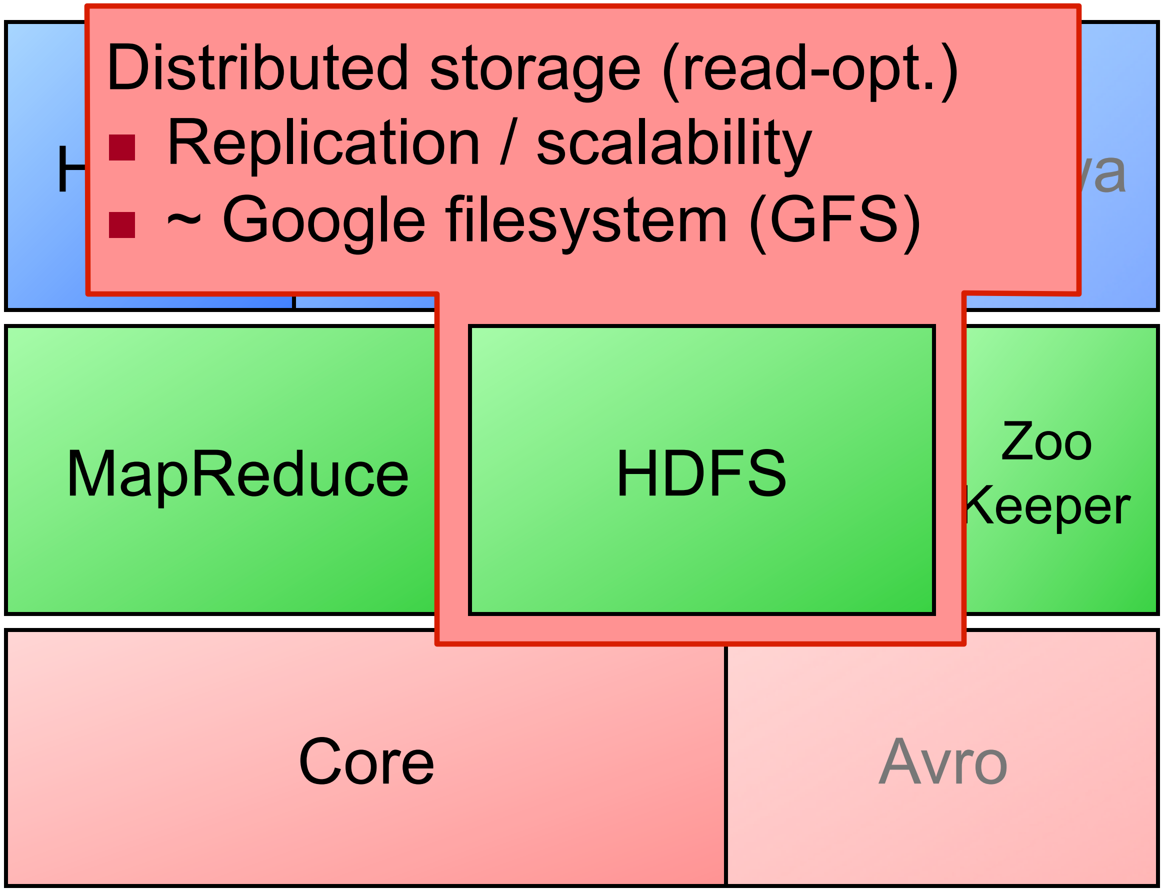
Hadoop


HBase
Pig
Hive
Chukwa
Map
Cross-language serialization:
RPC / persistence
~ Google ProtoBuf / FB Thrift
Core
Avro

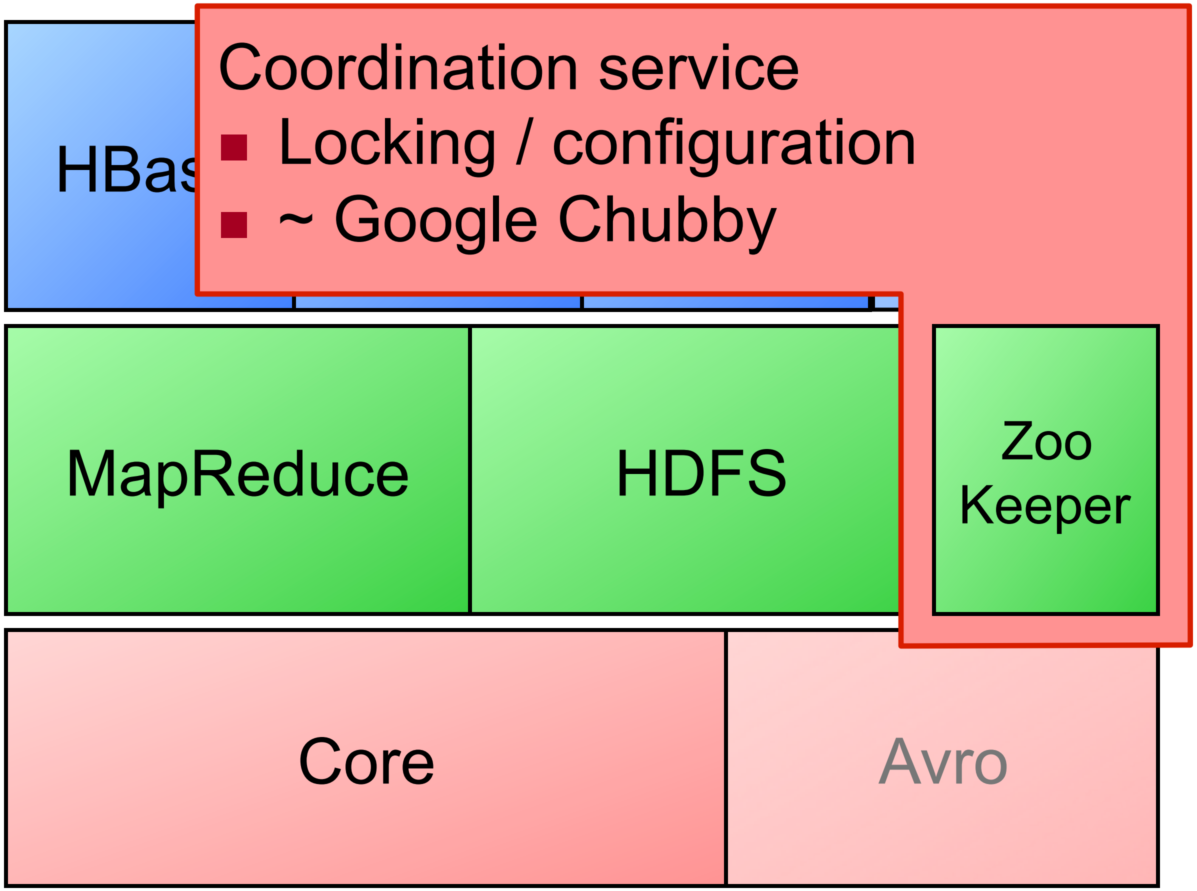
Hadoop


Distributed execution (batch)
Programming model
Scalability / fault-tolerance
ukwa
MapReduce
HDFS
Zoo Keeper
Core
Avro

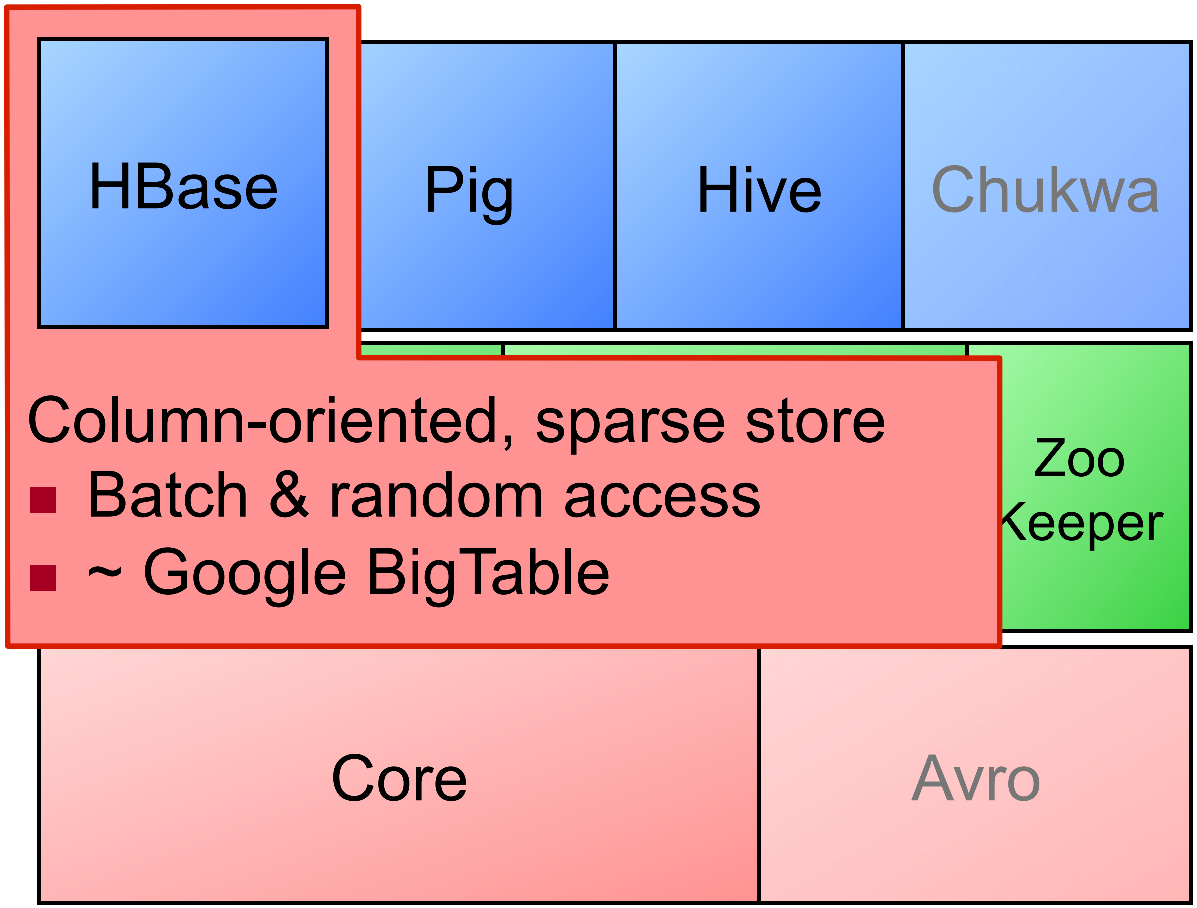
Hadoop


Distributed storage (read-opt.)
Replication / scalability
~ Google filesystem (GFS)
H
ya
MapReduce
HDFS
Zoo
Keeper
Core
Avro

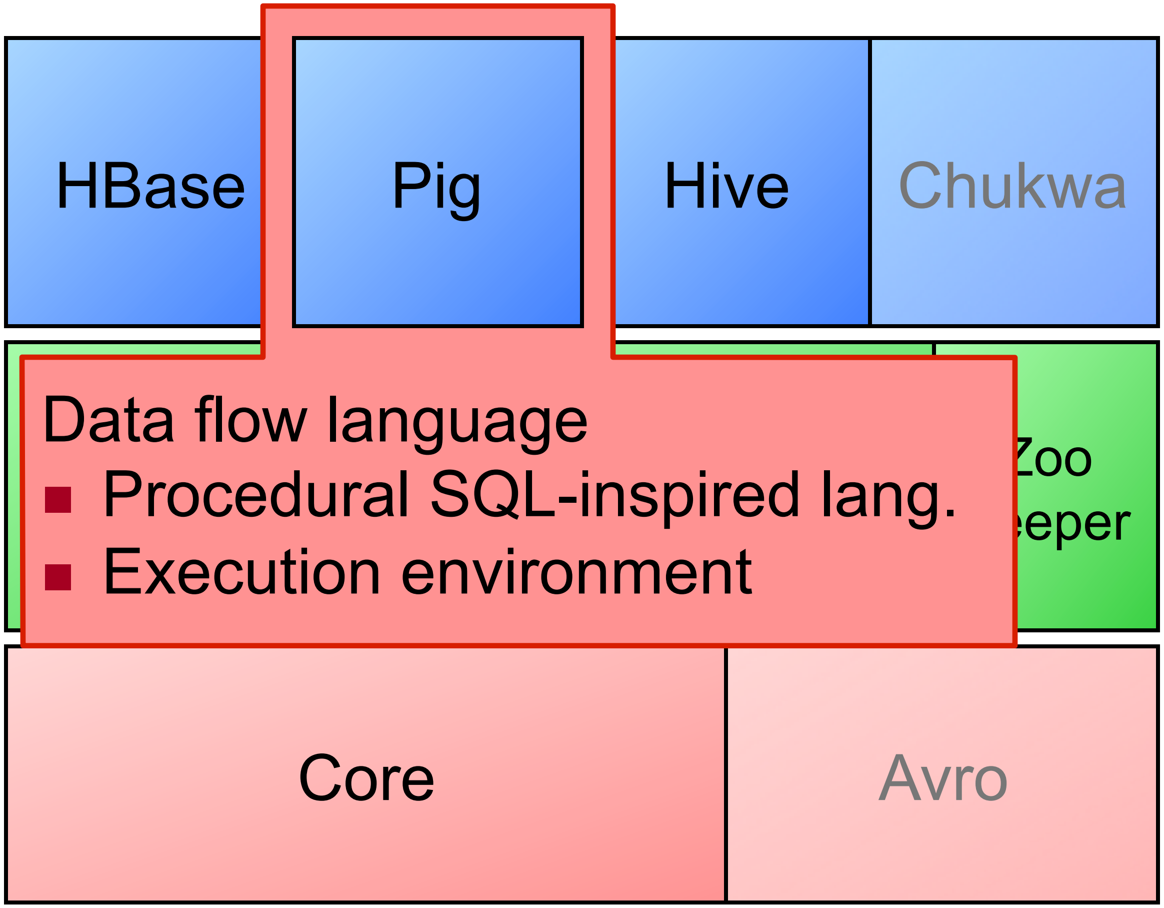
Hadoop


HBas
Coordination service
Locking / configuration
~ Google Chubby
MapReduce
HDFS
Zoo
Keeper
Core
Avro

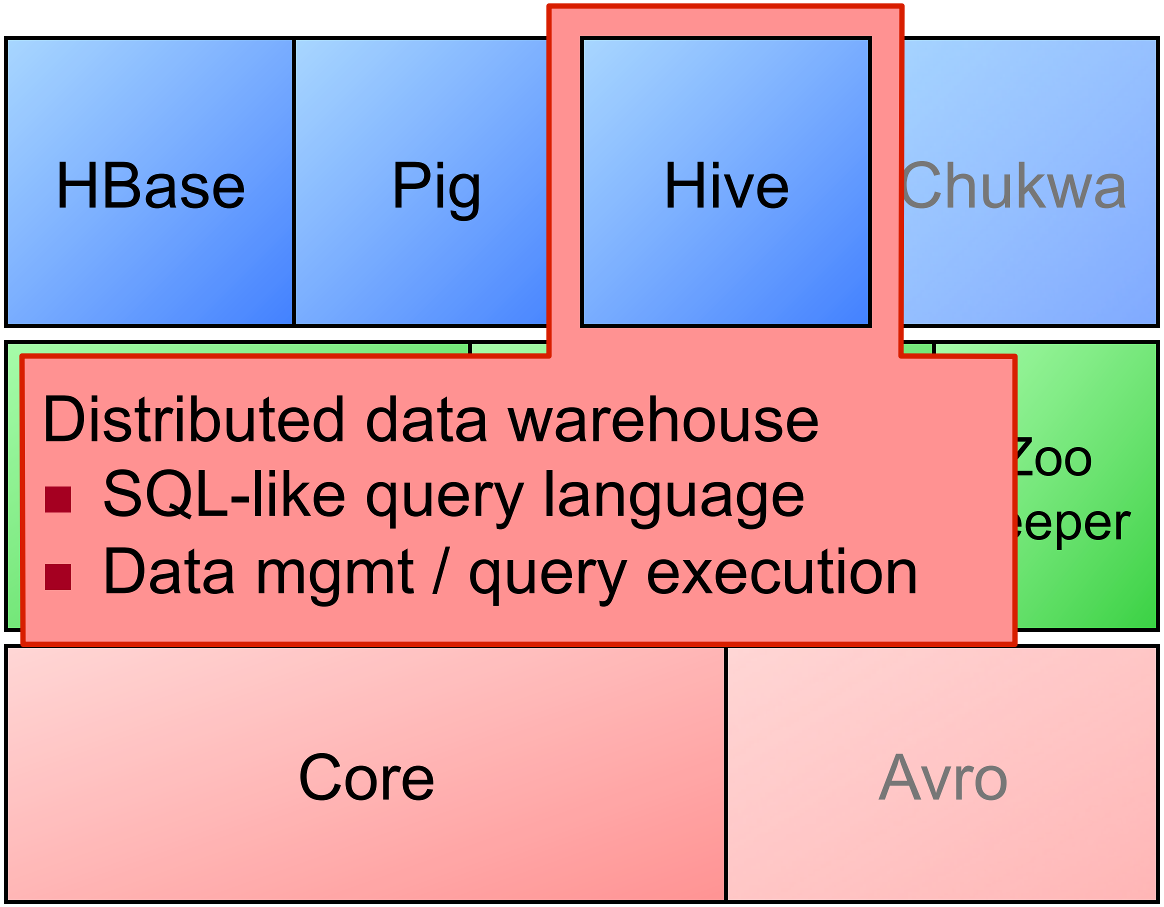
Hadoop


HBase
Pig
Hive
Chukwa
Column-oriented, sparse store
Batch & random access
~ Google BigTable
Zoo
Keeper
Core
Avro

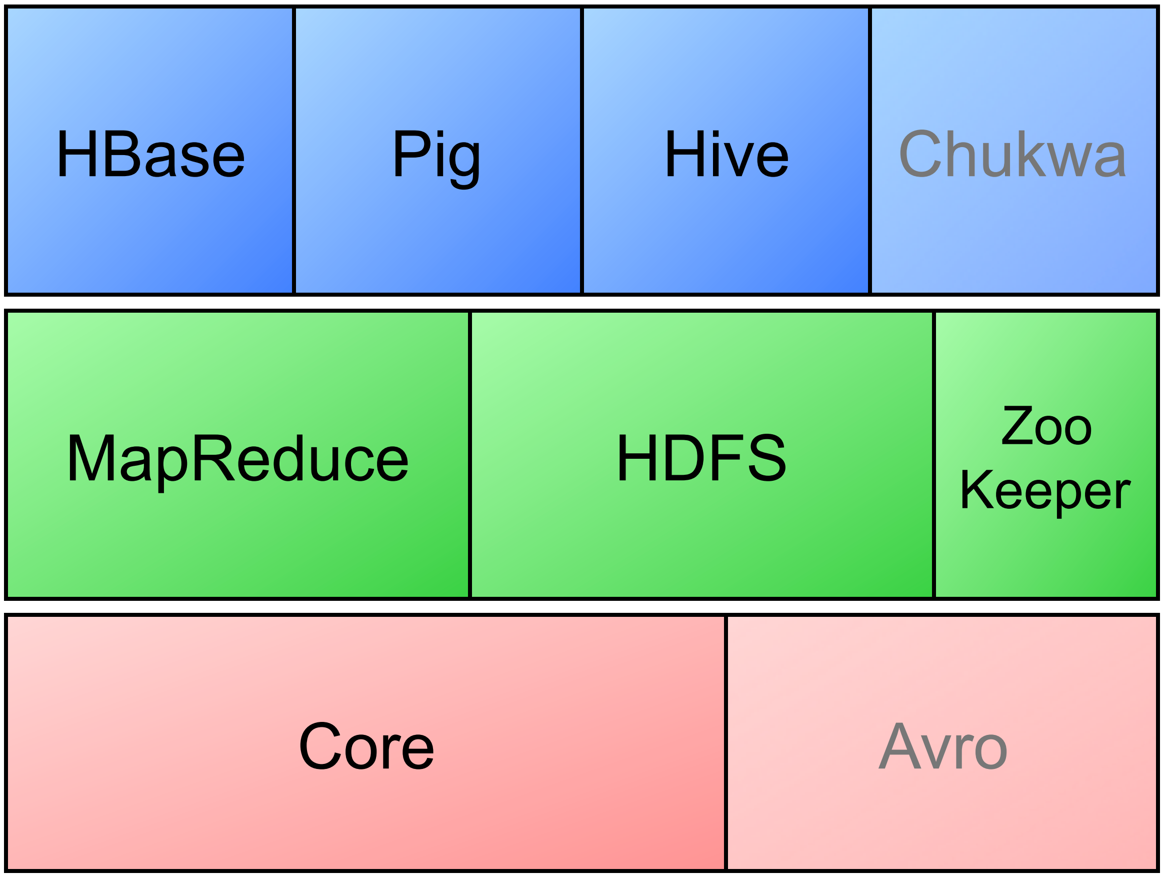
Hadoop


HBase
Pig
Hive
Chukwa
Data flow language
Procedural SQL-inspired lang.
Execution environment
Zoo
eeper
Core
Avro

Hadoop


HBase
Pig
Hive
Chukwa
Distributed data warehouse
SQL-like query language
Data mgmt / query execution
Zoo
eeper
Core
Avro

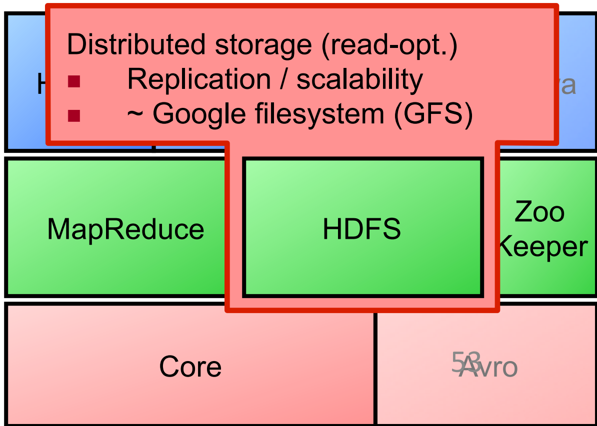
Hadoop



…… …… more

Hadoop: HDFS

- Hadoop's Distributed File System is designed to **reliably** store very large files across machines in a large cluster.
- Hadoop DFS stores each file as a **sequence of blocks**, all blocks in a file except the last block the same size.
  - Blocks belonging to a file are **replicated** for fault tolerance.
  - Block size & replication factor are configurable per file.
  - Files in HDFS are "**write once**" and have strictly one writer at any time.



Hadoop: HDFS

- An HDFS installation consists of a single **Namenode** a (**master server)**that
  - **manages** the file system namespace
  - **regulates** access to files by clients.

- And a number of **Datanodes**, one **per node** in the cluster, which
  - **manage** storage attached to the nodes that they run on

Hadoop: HDFS

- **Namenode**
  - **Makes** filesystem namespace operations like opening, closing, renaming etc. of files and directories **available via** an **RPC interface**.
  - determines the mapping of blocks to Datanodes.
- **Datanodes** are **responsible**
  - for serving read & write requests from filesystem clients
  - perform block creation, deletion, and replication upon instruction from the Namenode.

Summary

- A simple programming model for processing large dataset on large set of computer cluster
- Fun to use, focus on problem, and let the library deal with the messy detail

References

- Original paper (http://labs.google.com/papers/mapreduce.html)
- On wikipedia (http://en.wikipedia.org/wiki/MapReduce)
- Hadoop – MapReduce in Java (http://lucene.apache.org/hadoop/)
- Starfish - MapReduce in Ruby (http://rufy.com/starfish/)