MapReduce

COMPUTAÇÃO DE ALTO DESEMPENHO 2018/2019

HERVÉ PAULINO

SLIDES ADAPTED FROM “DATA-INTENSIVE TEXT PROCESSING WITH MAPREDUCE” BY JIMMY LIN

Bibliography

Chapters 1 and 2 of **Data-Intensive Text Processing with MapReduce,** Jimmy Lin and Chris Dyer, Morgan and Claypool Publishers
https://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf

Typical Large-Data Problem

Map

Iterate over a large number of records

Extract something of interest from each

Shuffle and sort intermediate results

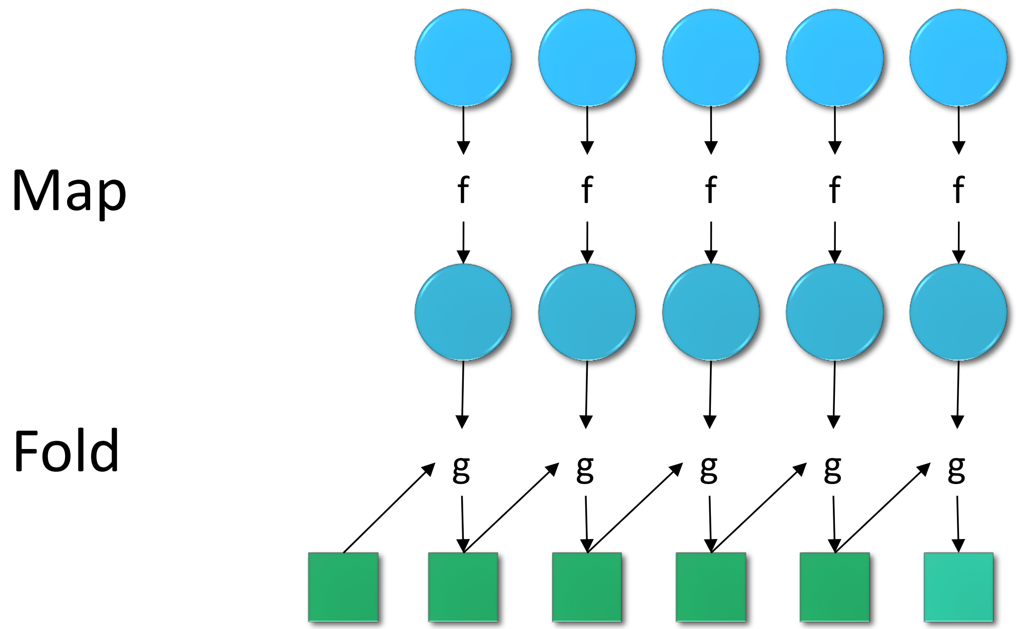
Aggregate intermediate results Reduce

Generate final output

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

MapReduce ~ Map + Fold from functional programming!



MapReduce

Programmers specify two functions:

- map $(k_1, v_1) \rightarrow \langle k_2, v_2 \rangle^*$
- reduce $(k_2, v_2^*) \rightarrow \langle k_3, v_3 \rangle^*$
  - All values with the same key are reduced together

The runtime handles everything else…

MapReduce Key Concepts

Data distributed at load time

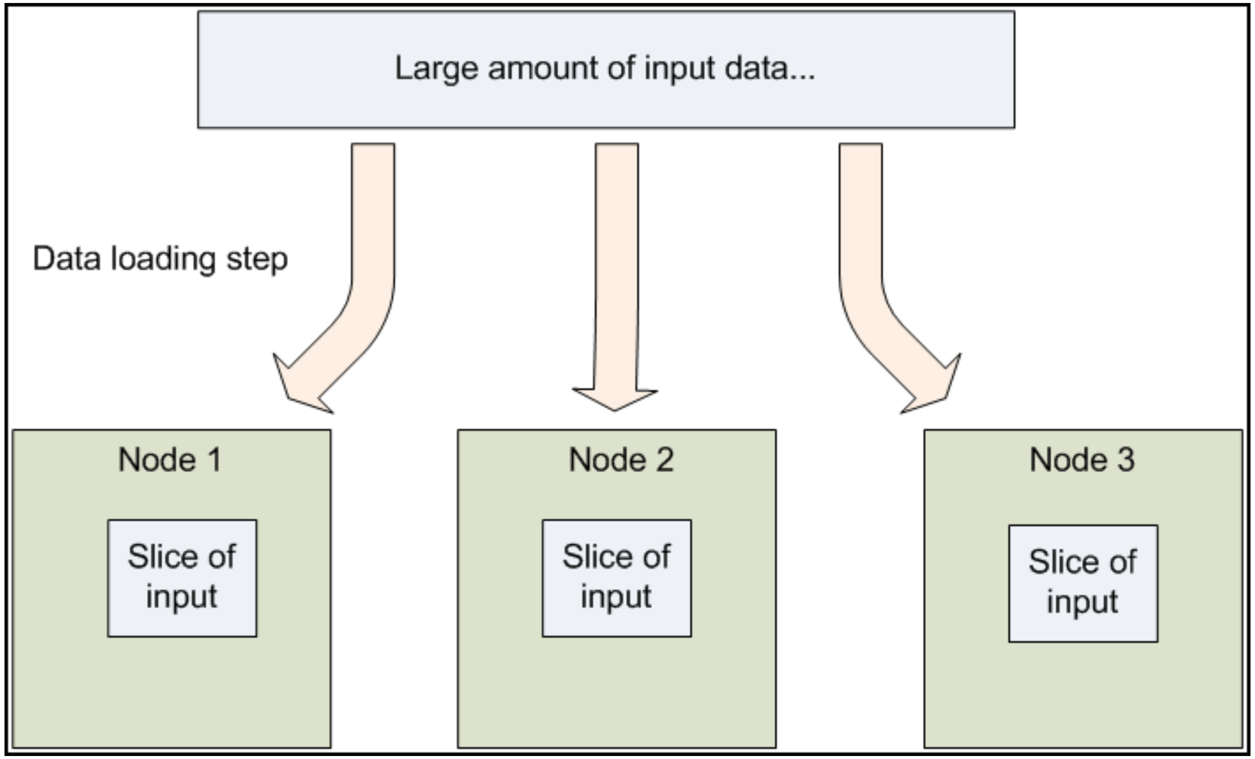
Records are processed in isolation

- Benefit: reduced communication

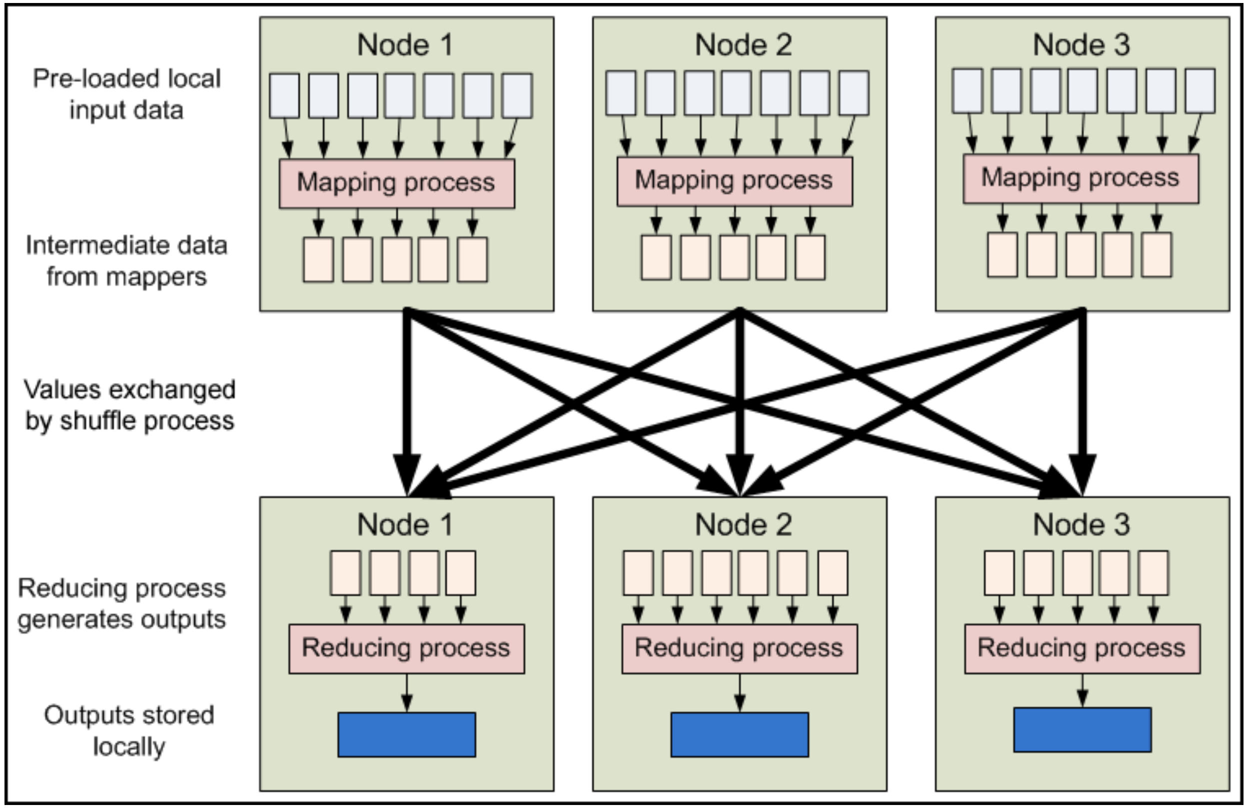
Tasks:

- Mapper – task that processes records
- Reducer – task that aggregates results from mappers

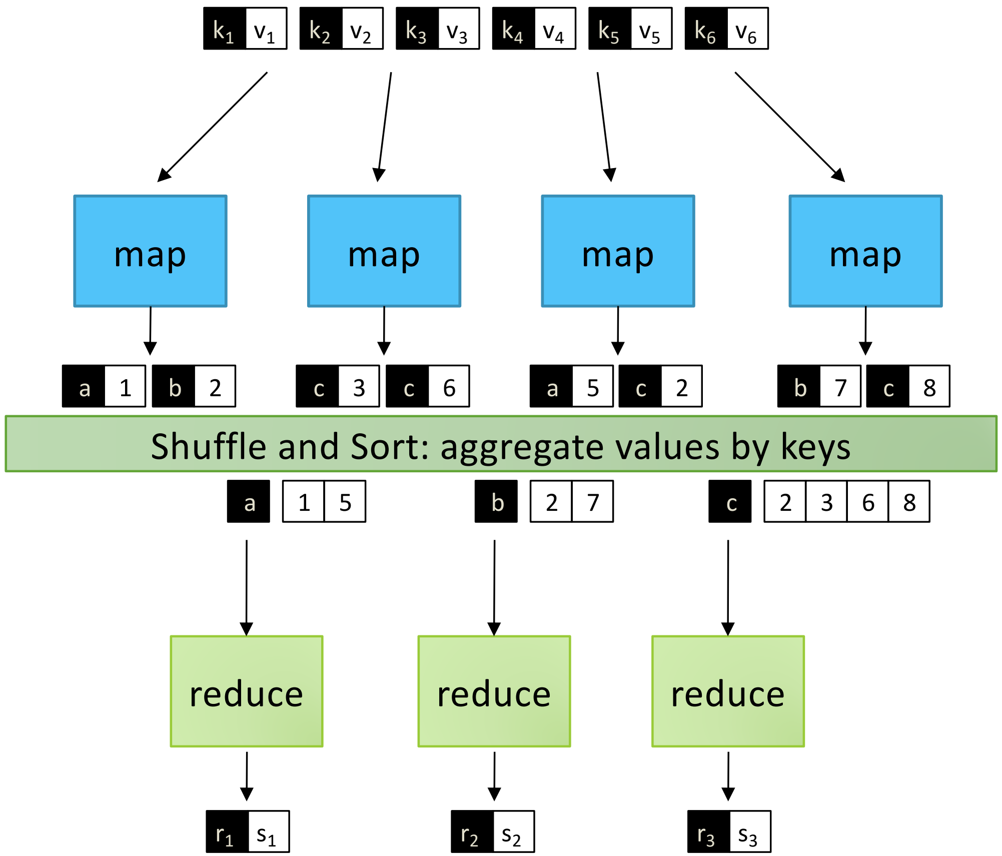
Distribute data at load time



MapReduce



Node 1
Node 2
Node 3
Pre-loaded local input data
Mapping process
Mapping process
Mapping process
Intermediate data from mappers
Values exchanged by shuffle process
Node 1
Node 2
Node 3
Reducing process generates outputs
Reducing process
Reducing process
Reducing process
Outputs stored locally



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
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

MapReduce versus Grid/Cluster Computing

How is the previous picture different from normal grid/cluster computing?

Grid/cluster:

Programmer manages communication via MPI

vs

Hadoop:

communication is implicit

Hadoop manages data transfer and cluster topology issues

Scalability

Hadoop overhead

- MPI does better for small numbers of nodes

Hadoop – flat scalabity ➔ pays off with large data

- Little extra work to go from few to many nodes

MPI – requires explicit refactoring from small to larger number of nodes

MapReduce

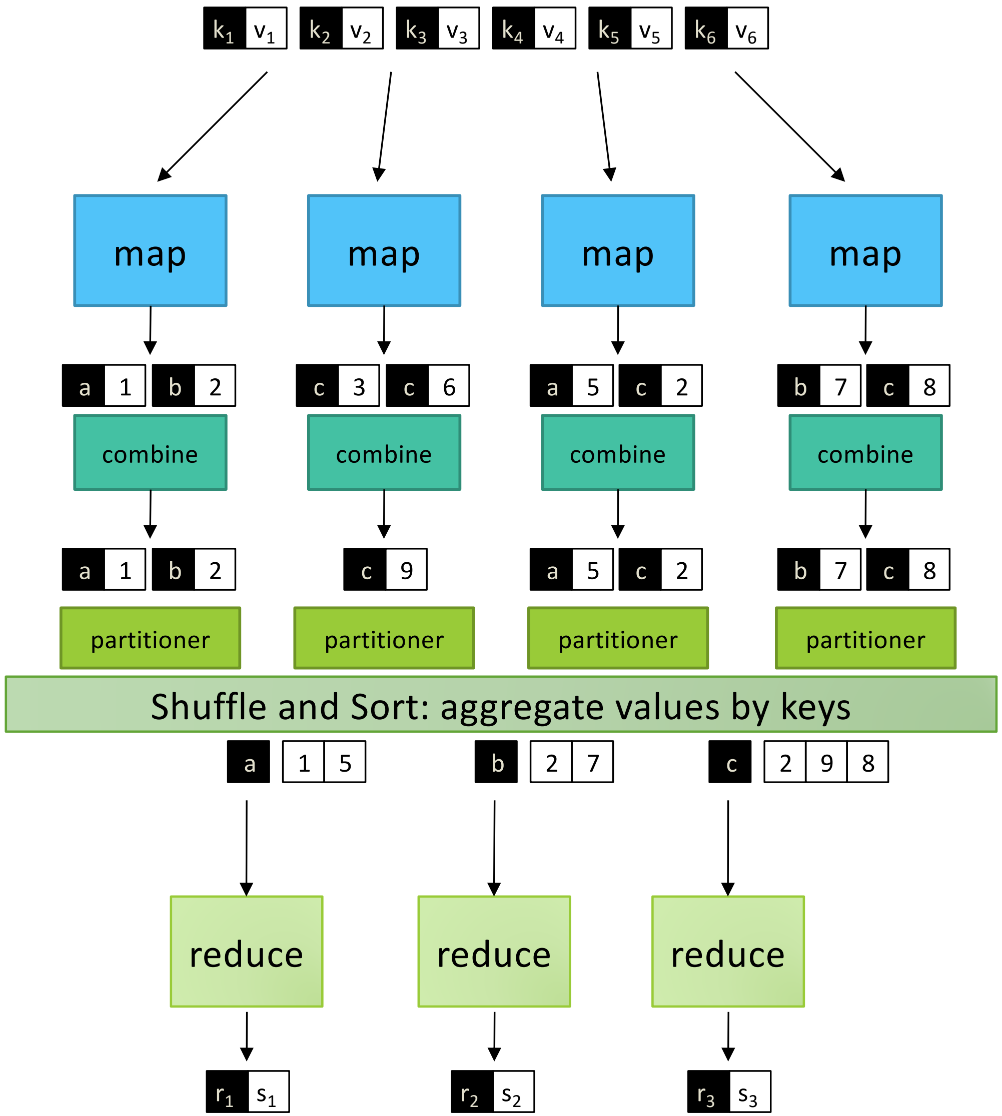
Programmers specify two functions:

- map (k, v) → <k’, v’>*
- reduce (k’, v’) → <k’, v’>*
- All values with the same key are reduced together

The runtime handles everything else…

Not quite…usually, programmers also specify:

- partition (k’, number of partitions) → partition for k’
- Often a simple hash of the key, e.g., hash(k’) mod n
- Divide up key space for parallel reduce operations
- combine (k’, v’) → <k’, v’>*
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce 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
partitioner
partitioner
partitioner
partitioner
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

MapReduce Runtime

Handles scheduling
- Assigns workers to map and reduce tasks

Handles “data distribution”
- Moves processes to data

Handles synchronization
- Gathers, sorts, and shuffles intermediate data

Handles faults
- Detects worker failures and restarts

Everything happens on top of a distributed FS (later)

“Hello World”: Word Count

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


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

In Practice - Mapper Code

Input of type <LongWritable, Text>

- The key (the LongWritable) can be assumed to be the position in the document our input is in. This doesn't matter for this example.

Output of type <Text, LongWritable>.

- The key is the token, and the value is the count. This is always 1.

```
public static class TokenizerMapper extends Mapper<Object, Text,
Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);

    public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        Text word = new Text(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

In Practice - Reducer Code

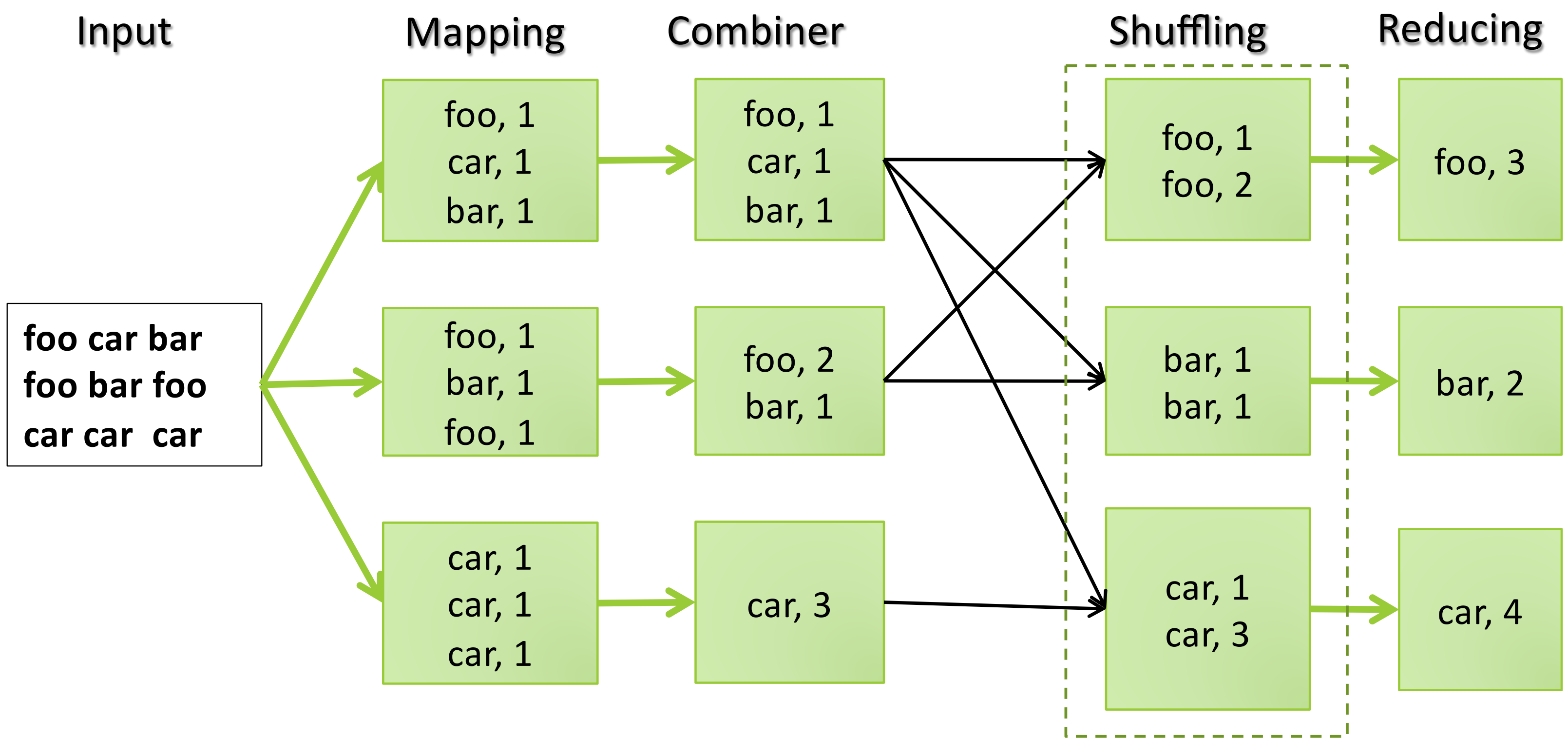
Input is the Mapper's output, of type <Text, LongWritable>

Output is still a <Text,LongWritable>

- but it reduces N inputs for token T, into one output <T, N>

```
public static class IntSumReducer extends
                Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
 Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values)
            sum += val.get();
         result.set(sum);
        context.write(key, result);
    }
  }
```

Word Count with Combiner



In Practice- Combiner Code

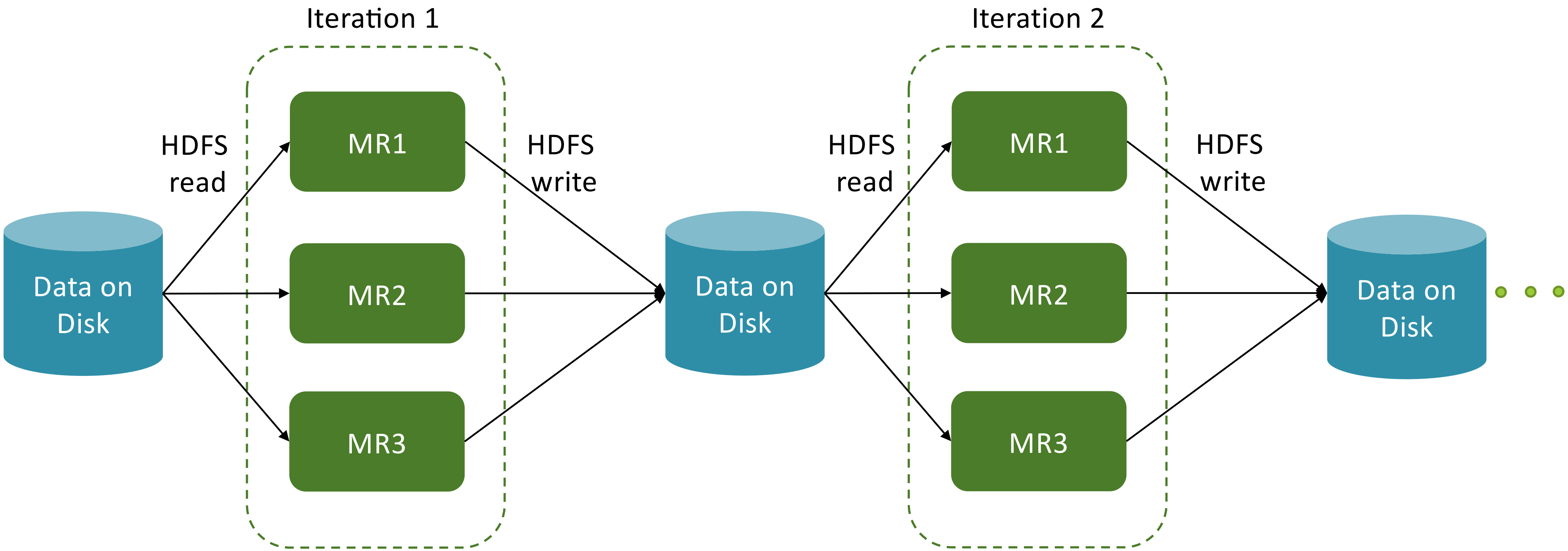
All reductions that are associative an commutative may be used as combiners

So it is the same

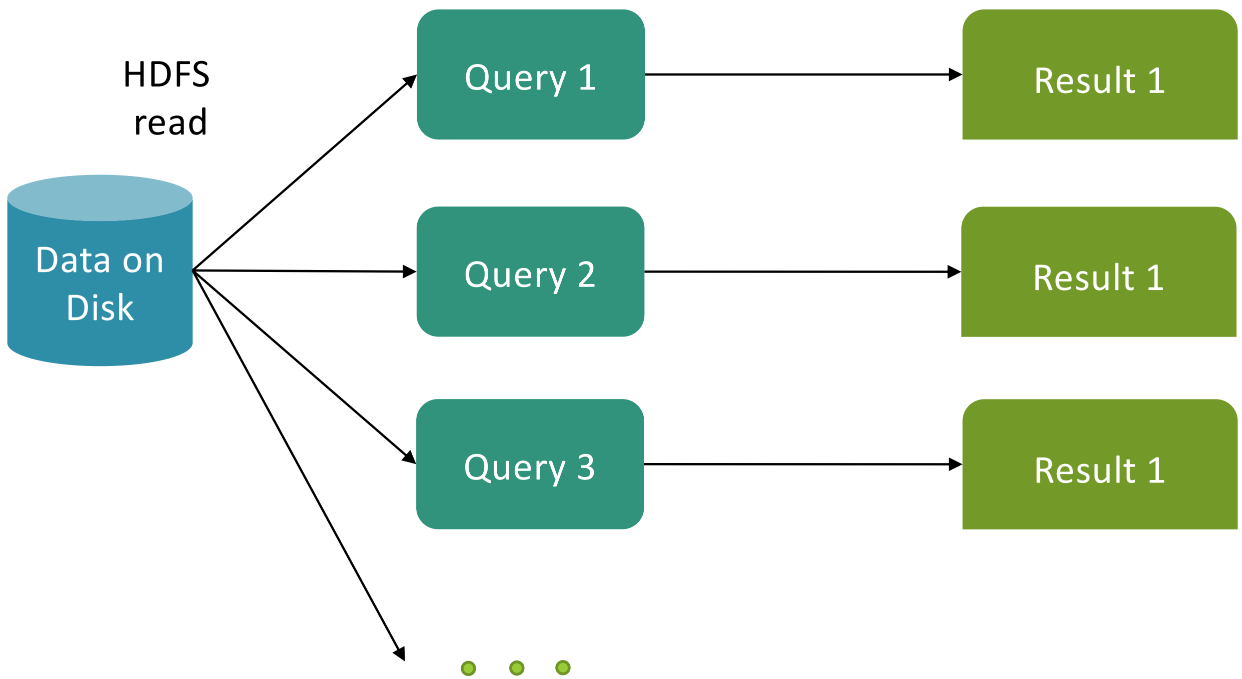
Putting It All Together

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");

job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, …);
FileOutputFormat.setOutputPath(job, ….);
```

Iterative Procedure in MapReduce



Interactive Procedure in MapReduce



MapReduce Implementations

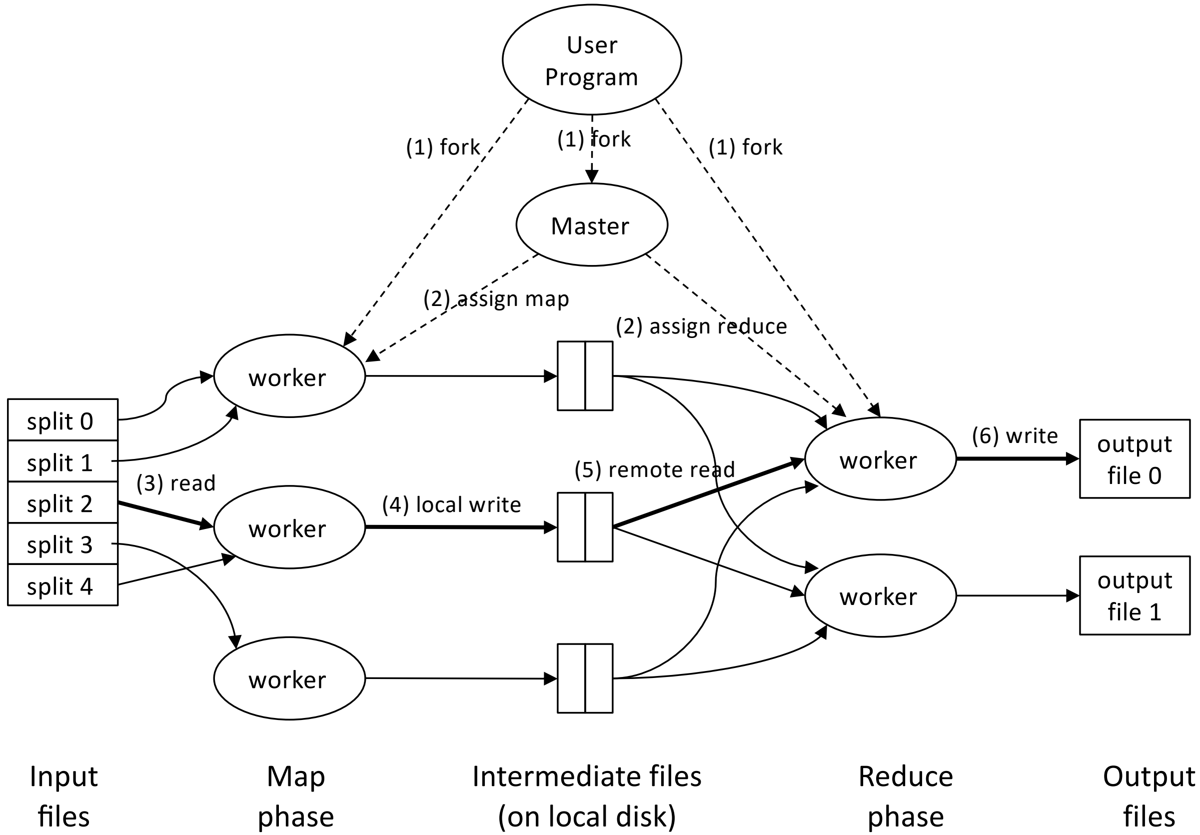
MapReduce is a programming model

Google has a proprietary implementation in C++

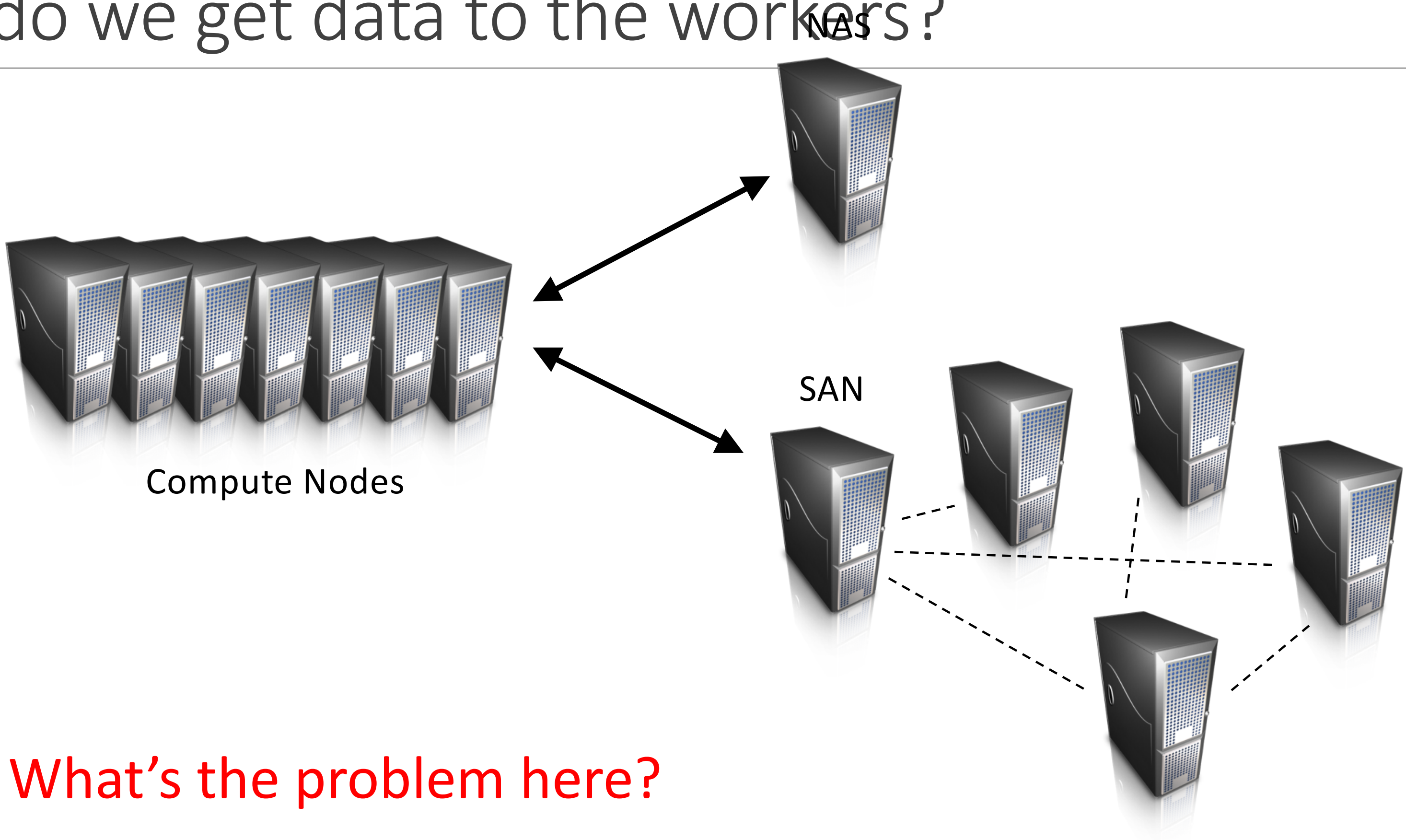
- Bindings in Java, Python

Hadoop is an open-source implementation in Java

- Project led by Yahoo, used in production
- Rapidly expanding software ecosystem


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disk)
Reduce phase
Output files

How do we get data to the workers?



What's the problem here?

Distributed File System

Don't move data to workers… move workers to the data!

- Store data on the local disks of nodes in the cluster
- Start up the workers on the node that has the data local

Why?

- Not enough RAM to hold all the data in memory
- Disk access is slow, but disk throughput is reasonable

A distributed file system is the answer

- GFS (Google File System)
- HDFS for Hadoop (= GFS clone)

Distributed File System: Assumptions

Commodity hardware over "exotic" hardware
- Scale out, not up

High component failure rates
- Inexpensive commodity components fail all the time

"Modest" number of HUGE files

Files are write-once, mostly appended to
- Perhaps concurrently

Large streaming reads over random access

High sustained throughput over low latency

GFS slides adapted from material by (Ghemawat et al., SOSP 2003)

Distributed File System: Design Decisions

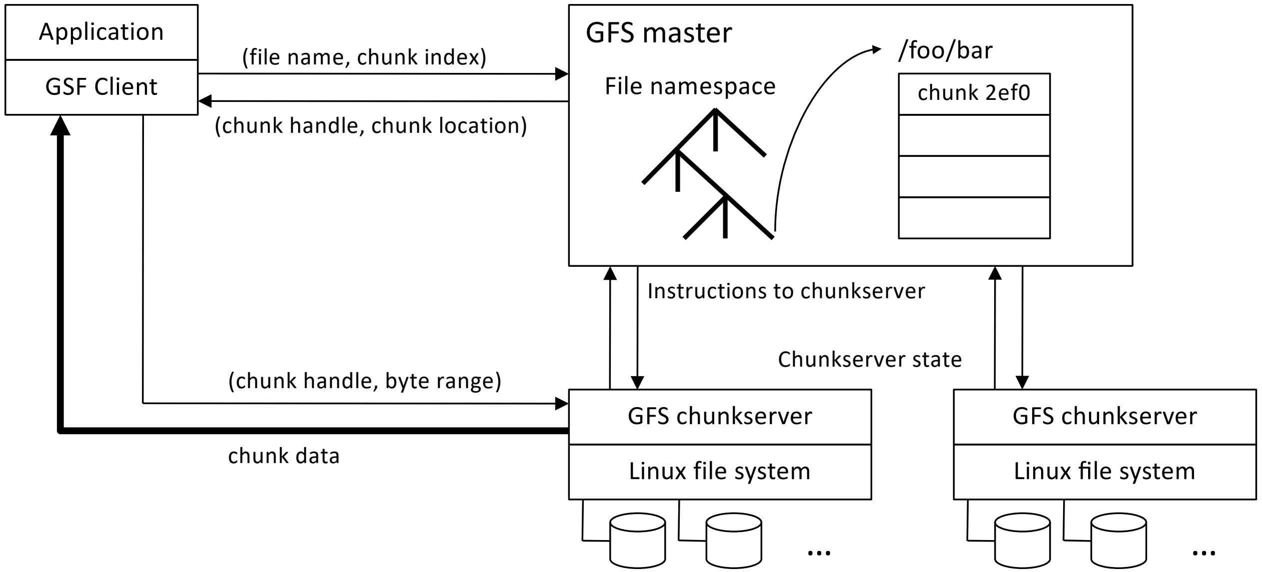
Files stored as chunks
- Fixed size (64MB)

Reliability through replication
- Each chunk replicated across 3+ chunkservers

Single master to coordinate access, keep metadata
- Simple centralized management

No data caching
- Little benefit due to large datasets, streaming reads

Simplify the API
- Push some of the issues onto the client



Redrawn from (Ghemawat et al., SOSP 2003)

Master's Responsibilities

Metadata storage

Namespace management/locking

Periodic communication with chunkservers

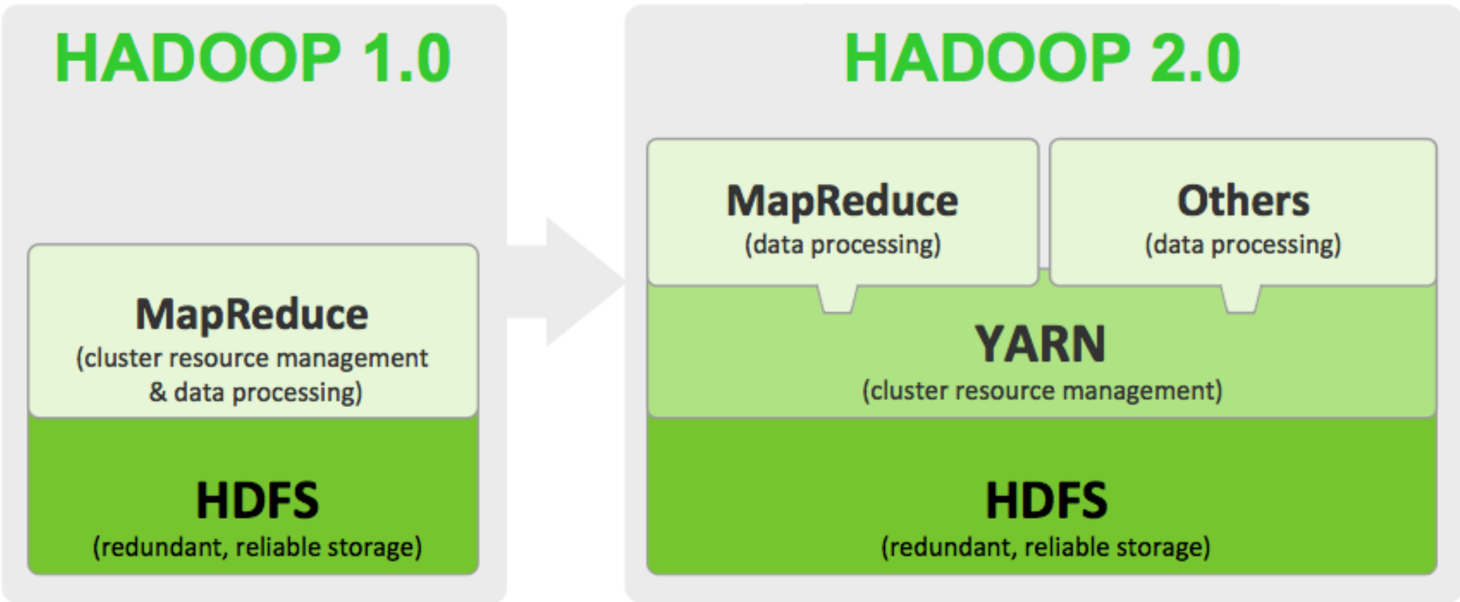
Chunk creation, re-replication, rebalancing

Garbage collection

YARN - Yet Another Resource Negotiator

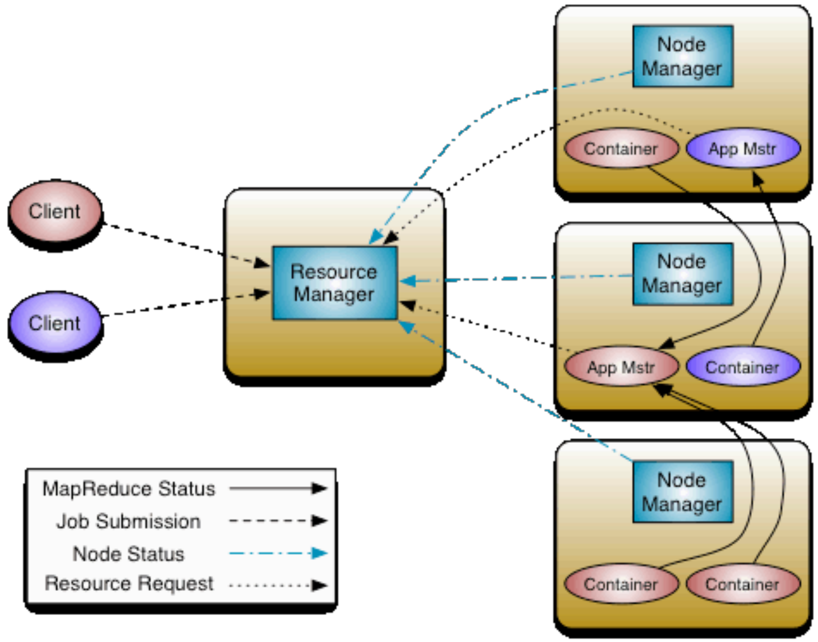
Next version of MapReduce or MapReduce 2.0 (MRv2)

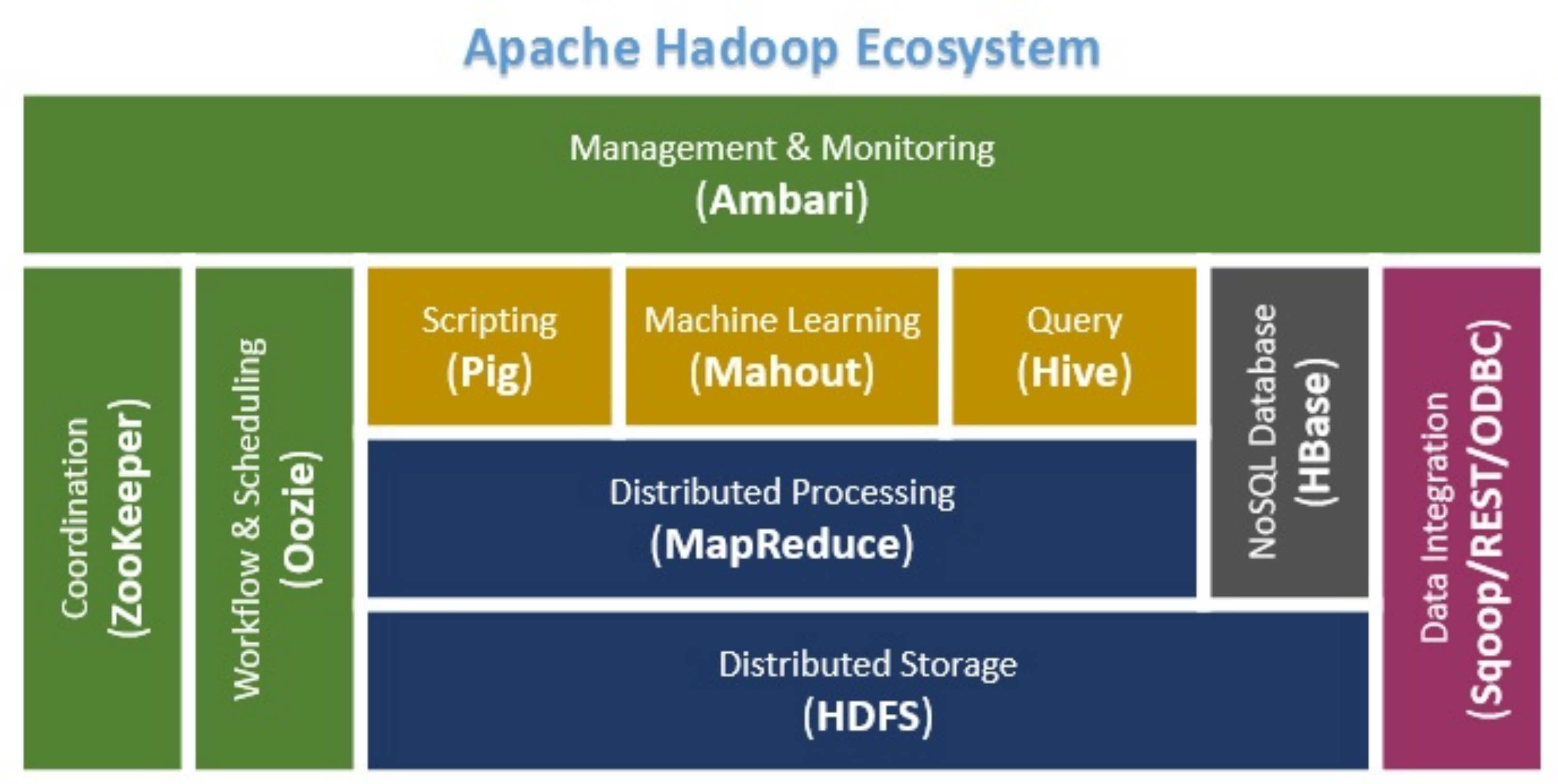
In 2010 group at Yahoo! Began to design the next generation of MR



YARN architecture

- **Resource Manager**
  - Central Agent – Manages and allocates cluster resources
- **Node Manager**
  - Per-node agent – Manages and enforces node resource allocations
- **Application Master**
  - Per Application
  - Manages application life cycle and task scheduling




Apache Hadoop Ecosystem
Management & Monitoring
(Ambari)
Coordination
(ZooKeeper)
Workflow & Scheduling
(Oozie)
Scripting
(Pig)
Machine Learning
(Mahout)
Query
(Hive)
NoSQL Database
(HBase)
Data Integration
(Sqoop/REST/ODBC)
Distributed Processing
(MapReduce)
Distributed Storage
(HDFS)