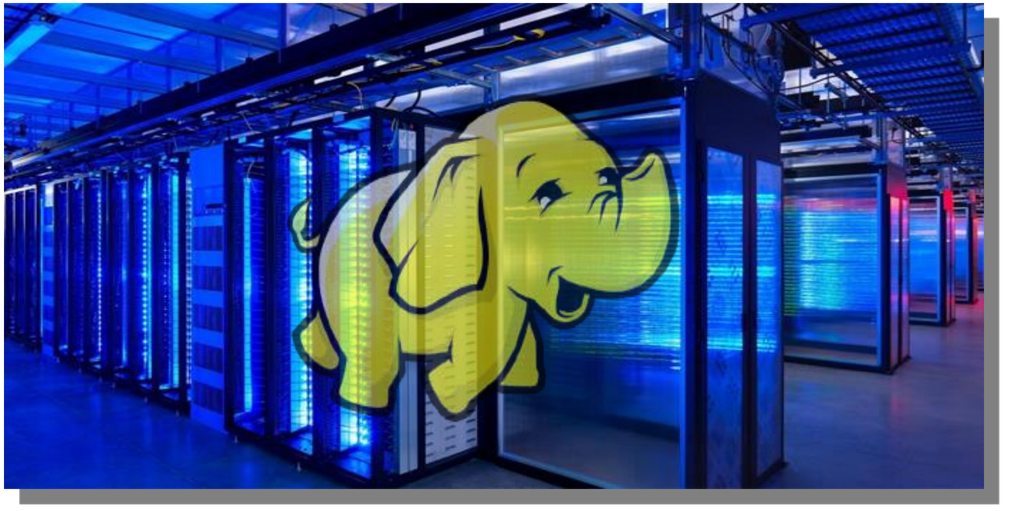
An Introduction to Cluster Computing with Apache Hadoop



Anastasios Gounaris
Apostolos N. Papadopoulos

Assistant Professors
Data Engineering Lab,
Department of Informatics
Aristotle University of Thessaloniki
GREECE

http://delab.csd.auth.gr/courses/c_bigdata/index.html

Outline

- Why one machine is not enough ?

- Parallel architectures

- Important issues in cluster computing

- Hadoop MapReduce

- Theoretical Issues

- Spark

Motivation

We need **more CPUs** because:

To run programs faster

We need **more disks** because:

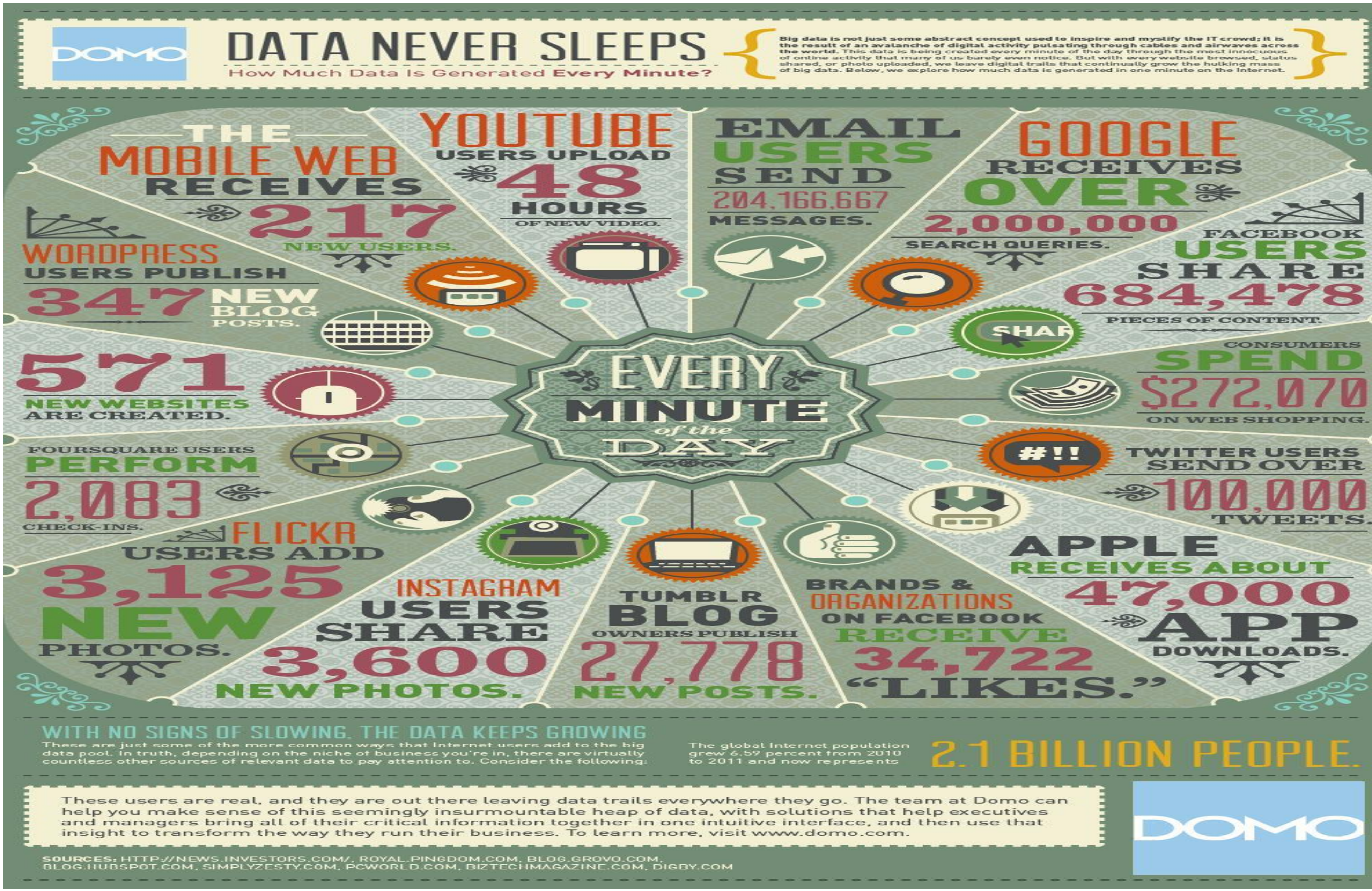
modern applications require huge amounts of data

with many disks we can perform I/O in parallel

Assume that we are able to build a single disk with **500 TB** capacity. This is enough to store **more than 20 billion webpages** (assuming an average size per page of **20KB**).

However, **just to scan** these 500 TB we need **more than 4 months** if the disk can bring **40 MB/sec**. Imagine the time required to process the data !

What is Happening Today



In the Near Future

"***IBM Research*** *and Dutch astronomy agency* ***Astron*** *work on new technology to handle*

one exabyte of raw data per day

that will be gathered by the world largest radio telescope, the ***Square Kilometer Array****, when activated in* ***2024***."

Some Challenges

- Scalability
- Load balancing
- Fault Tolerance
- Efficiency
- Data Stream processing
- Support for complex objects
- Accuracy/Speed tradeoffs (with performance guarantees)

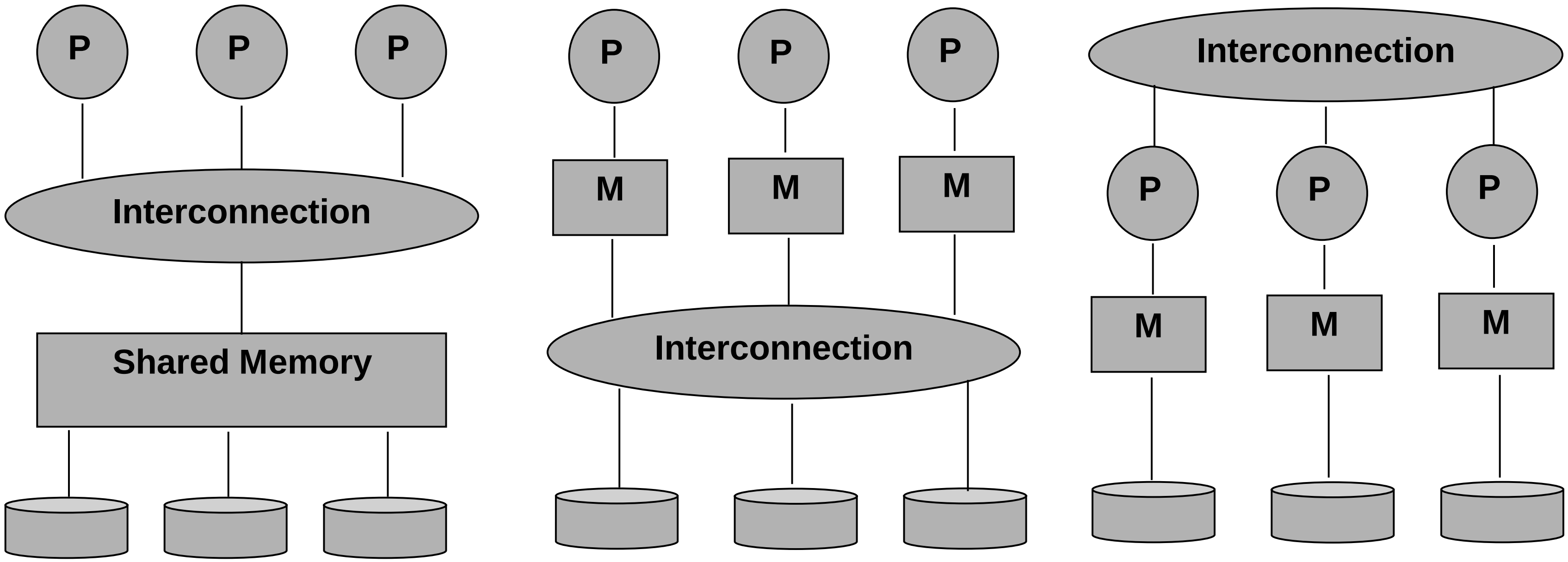
Parallel Architectures

Shared Memory: processors share a common main memory and also share secondary storage (e.g., disks)

Shared Disk: processors share only secondary storage, whereas each processor has its own private memory

Shared Nothing: processors do not share anything, each one has private secondary storage and memory

Parallel Architectures



shared memory　　shared disk　　shared nothing

Scalability

Scale-Up: put more resources into the system to make it bigger and more powerful

Scale-Out: connect a large number of “ordinary” machines and create a cluster

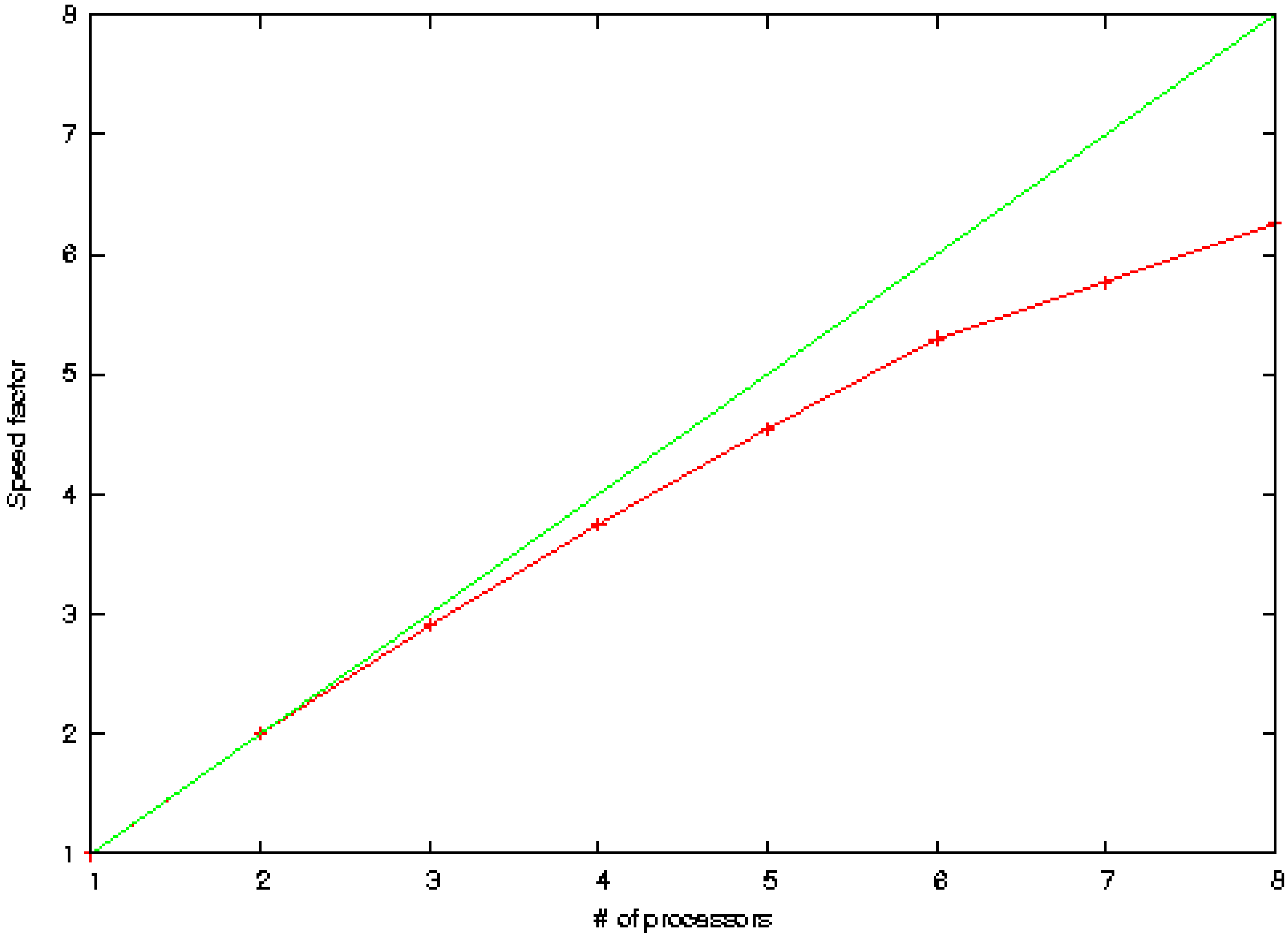
Scale-Out is **more powerful** than Scale-Up, and also **less expensive**

Scalability: measures

Among the three parallel architectures, shared-nothing is the one that **scales best**. This is the main reason for being adopted for building massively parallel systems (thousands of processors)

- **Speedup**: monitor performance by increasing the number of processors
- **Sizeup**: monitor performance by increasing only the dataset size
- **Scaleup**: monitor performance by increase both the number of processors and the dataset size

The Speedup Curve



Real Curves are Non-Linear

Why ?

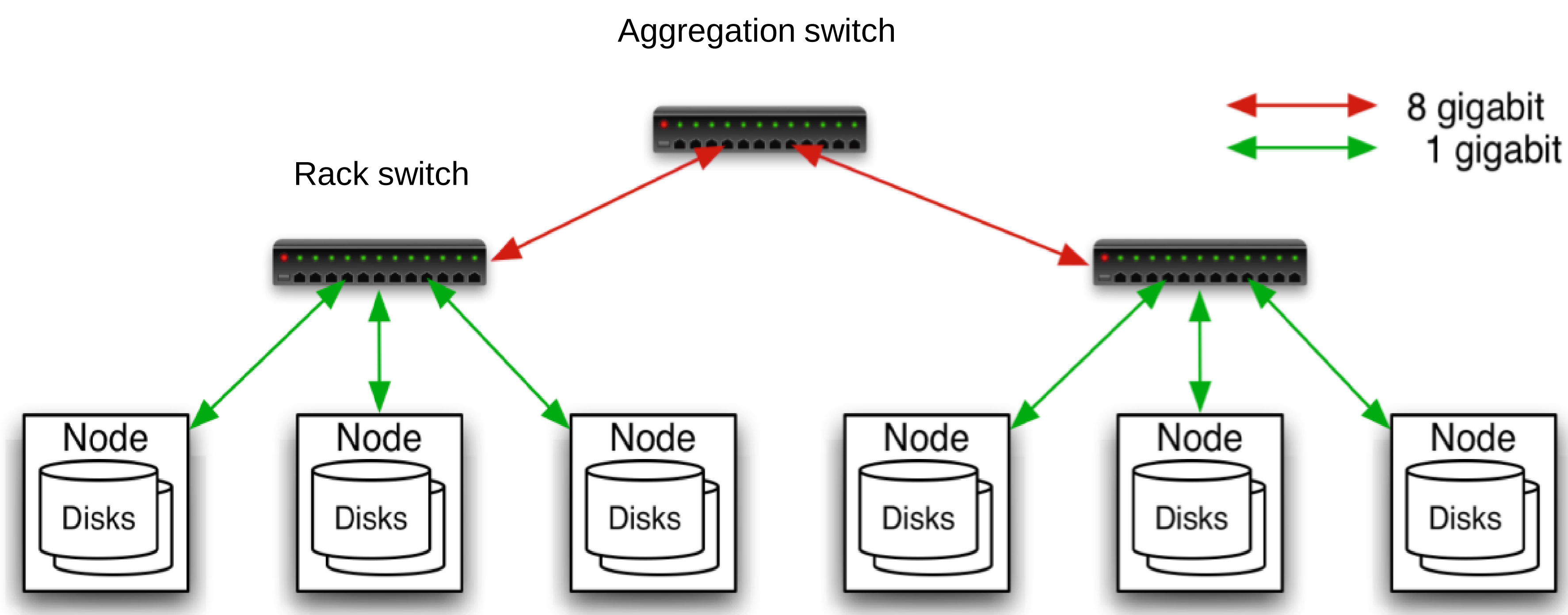
Start-up costs: cost for starting an operation in a processor

Interference: cost for communication among processors and resource congestion

Skew: either in data or tasks → the slowest processor becomes the bottleneck

Result formation: partial results from each processor must be combined to provide the final result.

Cluster Configuration Example



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- 8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Source: Matei Zaharia

Fault Tolerance

Failures are very common in massively parallel systems

Let *P* the probability that a disk will fail in the next month. If we have *D* disks in total, the probability that at least one disk will fail is given by:

Prob {*at least one disk failure*} = 1 - (1- *P*)^*D*

e.g., *D* = 10000, *P* = 0.0001

Prob {*at least one disk failure*} = 0.63

Fault Tolerance

Failures may happen because of:

Hardware not working properly

- Disk failure
- Memory failure (8% of DIMMs have problems)
- Inadequate cooling (CPU overheating)

Resource unavailability

- Due to overload

We must provide fault tolerance in the cluster!

Fault Tolerance

Simplest protocol: if there is a failure, restart the job.

Assume a job that requires 1 week of processing. If there is a failure once per week, the job will never finish!

Fault Tolerance

A better protocol:

Replicate the data and also split the job in parts and replicate them as well. As an alternative, submit a smaller job (task) and if it fails then start another one.

<u>A large job must be decomposed to simpler ones.</u>

Problems with MPI/RPC

Really hard to do at scale:

- How to split problem across nodes?

– Important to consider network and data locality

- How to deal with failures?

– If a typical server fails every 3 years, a 10,000-node cluster sees 10 faults/day!

- Even without failures: stragglers (a node is slow)

Hadoop

A very successful model and platform to run jobs in massively parallel systems (thousands of processors and disks)

It contains two parts:

- the Hadoop MapReduce layer
- the Hadoop Distributed File System (HDFS)

Hadoop is the open-source alternative of MapReduce and Google File System (GFS) invented by Google. It has been used in Google's data centers mainly for:

1) constructing and maintaining the **Inverted Index** and

2) executing the **PageRank** algorithm.

Hadoop Ecosystem - indicative

Ambari
(Provisioning, Management, Monitoring)

ZooKeeper
(Coordination)

Oozie
(Workflow & Scheduling)

Mahout
(Machine Learning)

Pig
(Scripts)

Hive
(SQL queries)

Spark/
Flink

Hbase
(NoSQL)

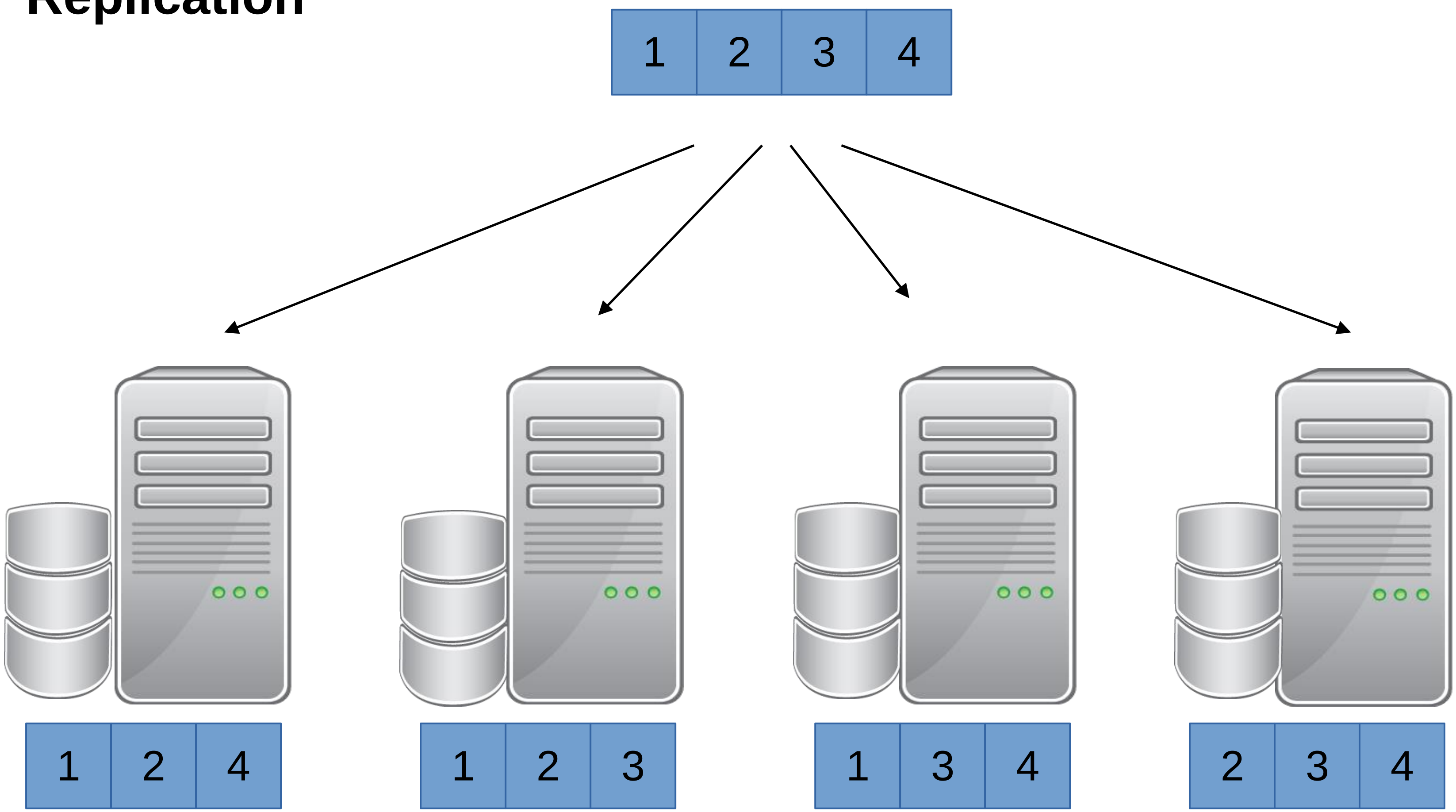
Hadoop MapReduce

Hadoop Distributed File System (HDFS)

Sqoop
(Data Integration)

Hadoop

Replication



The the file is split in chunks. Each is replicated three times in this example.

Processing in Hadoop

Based on key-value pairs

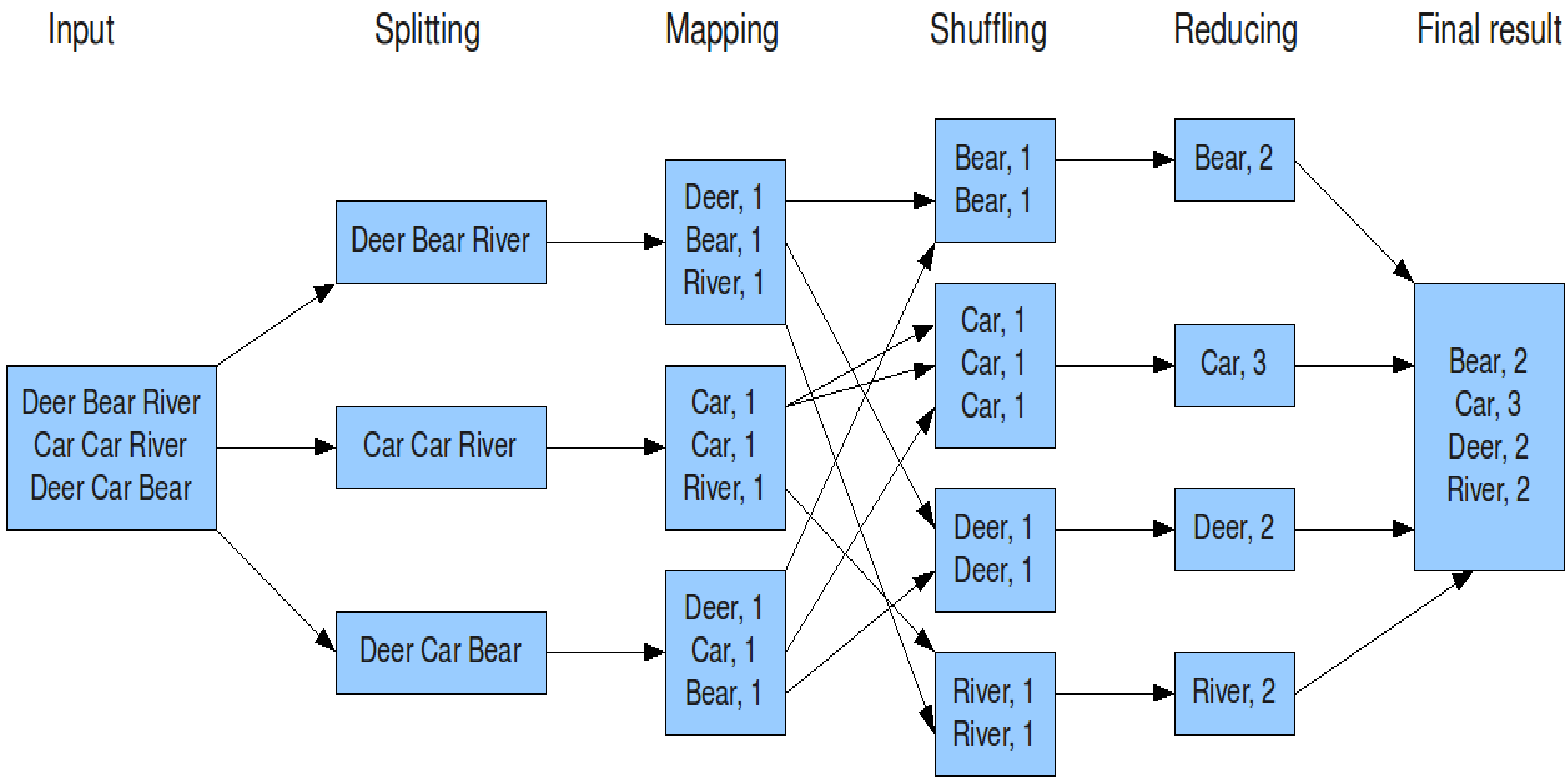
Each job is composed of one or more MR stages

Each MR stage comprises:

- the **map** phase
- the **shuffle-and-sort** phase
- the **reduce** phase

The programmer **focuses on the problem**. Replication, fault tolerance, scheduling, re-scheduling and other low level procedures are handled by Hadoop.

WordCount: the “hello world” of Hadoop



MapReduce API

The programmer must implement the following functions:

map(): accepts a set of key-value pairs and generates another list of key-value pairs.

combine(): performs an aggregation before sending the data to reducers (reduces network traffic).

partition(): uses a hash function to distribute data to reducers (load balancing, avoids hotspots).

reduce(): accepts a key and a list of values for this specific key and performs an aggregation.

Note: combine() and partition() are optional

WordCount in Hadoop

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text,
    Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

   public void map(LongWritable key, Text value, Context
    context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable,
    Text, IntWritable> {

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

 public static void main(String[] args) throws Exception {

   Configuration conf = new Configuration();

   Job job = new Job(conf, "wordcount");

   job.setOutputKeyClass(Text.class);

   job.setOutputValueClass(IntWritable.class);

   job.setMapperClass(Map.class);

   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);

   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.addInputPath(job, new Path(args[0]));

   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);

 }

}
```

WordCount: the **driver** program

```
public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "wordcount");
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      job.setMapperClass(Map.class);
      job.setReducerClass(Reduce.class);
      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TextOutputFormat.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new
   Path(args[1]));
      job.waitForCompletion(true);
   }
```

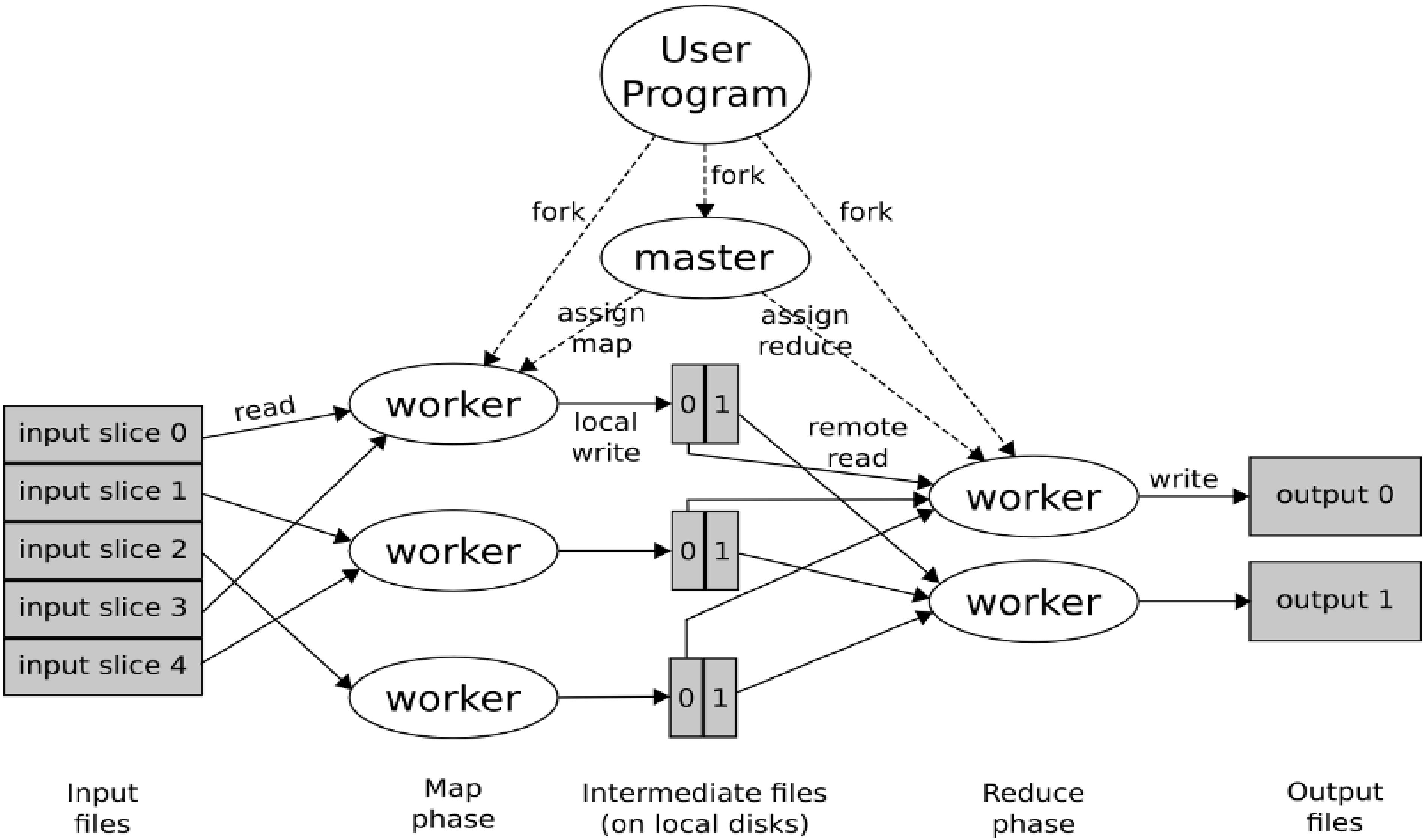
WordCount: the **map()** function

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

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

WordCount: the **reduce()** function

```
public static class Reduce extends Reducer<Text, IntWritable,
  Text, IntWritable> {


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

Workflow in Hadoop



HDFS

HDFS

Hadoop Internals

NameNode: It is the master of HDFS that controls the slave DataNodes to perform low level I/O tasks. The NameNode is the bookkeeper of HDFS and responsible to generate and distribute file splits.

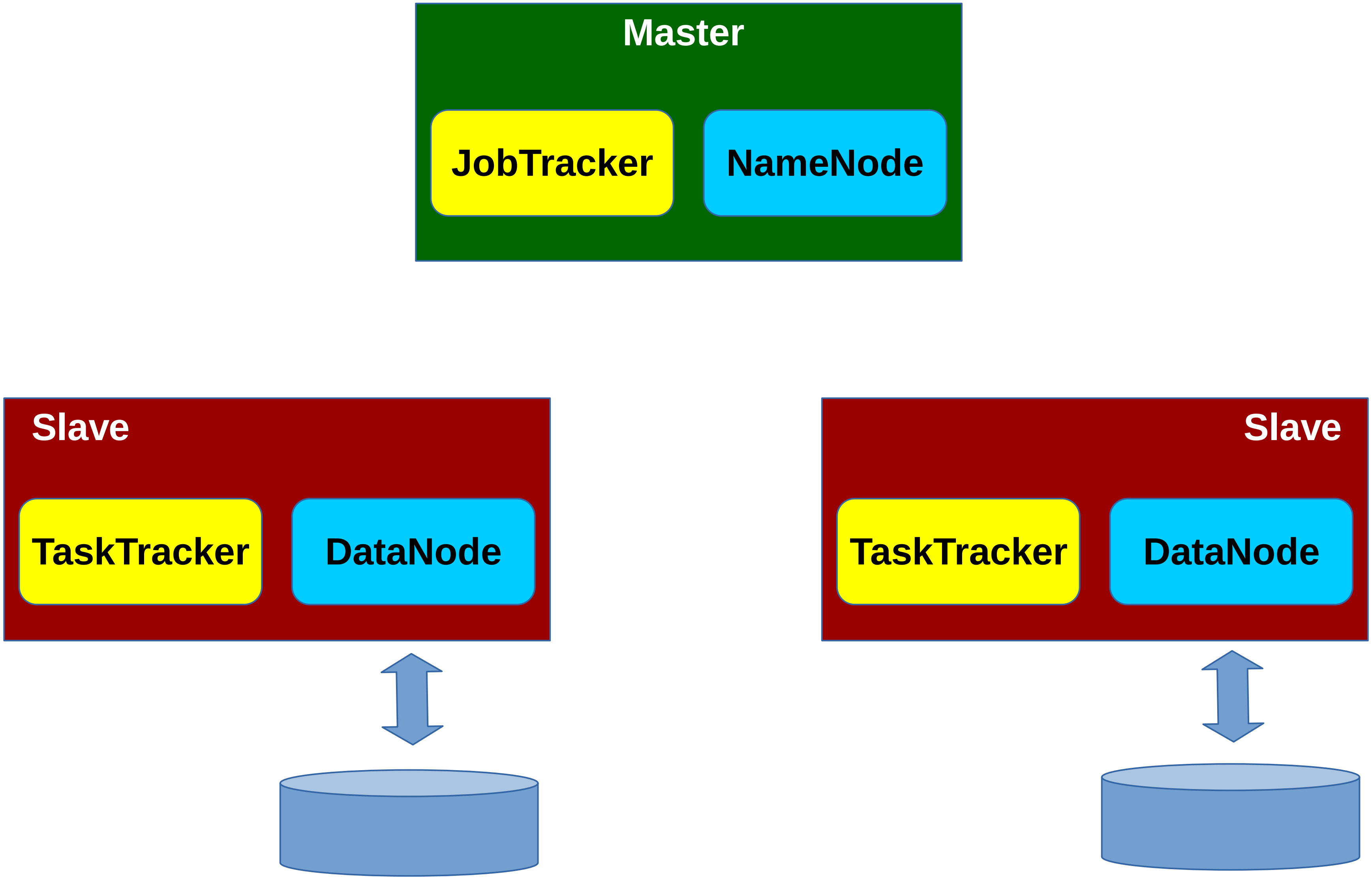
Secondary NameNode: It helps the NameNode to maintain the good shape of HDFS and participates in the recovery process.

DataNode: Each slave machne runs a DataNode daemon. Its main responsibility is to read/write HDFS blocks from/to the local file system. May communicate with other DataNodes for replication.

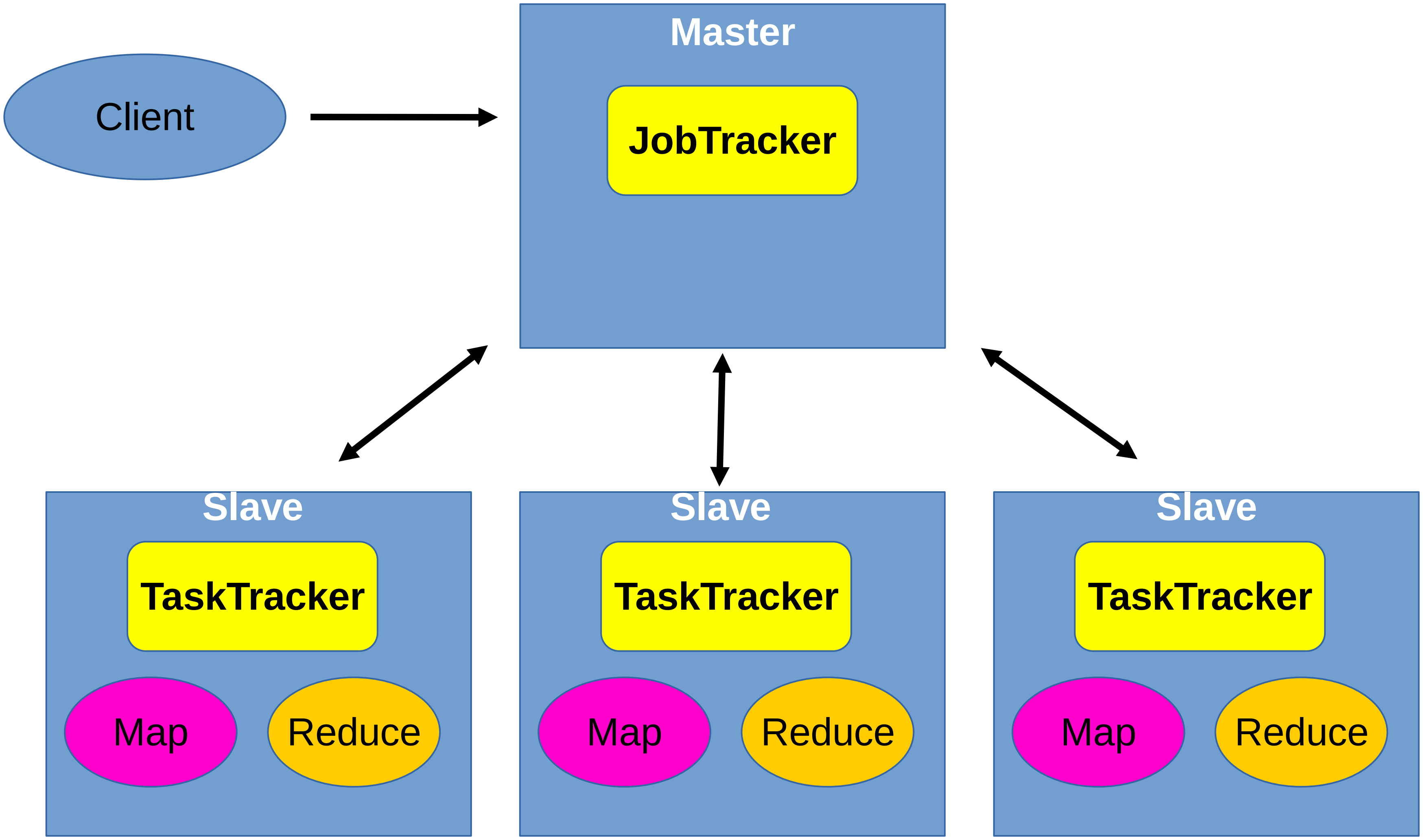
JobTracker: This daemon lies in between the user application and the Hadoop cluster. The main responsibility is to generate an execution plan for the user's job and to create tasks and monitor their progress.

TaskTracker: This is the slave daemon for JobTracker. Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. There is a single TaskTracker per slave node.

Hadoop Internals



Hadoop Internals



Hadoop Internals

YARN (**Y**et **A**nother **R**esource **N**egotiator)

Also known as **MapReduce2**, it was designed to overcome

- scalability problems when we have many thousands of cores in the cluster.
- Tight coupling with the MapReduce programming model

Up to now, the JobTracker takes care of resource allocation, job scheduling (matching tasks with TaskTrackers) and task progress monitoring (keeping track of tasks, restarting failed or slow tasks, and doing task bookkeeping, such as maintaining counter totals).

YARN <u>splits the responsibility of the JobTracker</u> to several components.

MRv1 vs MRv2

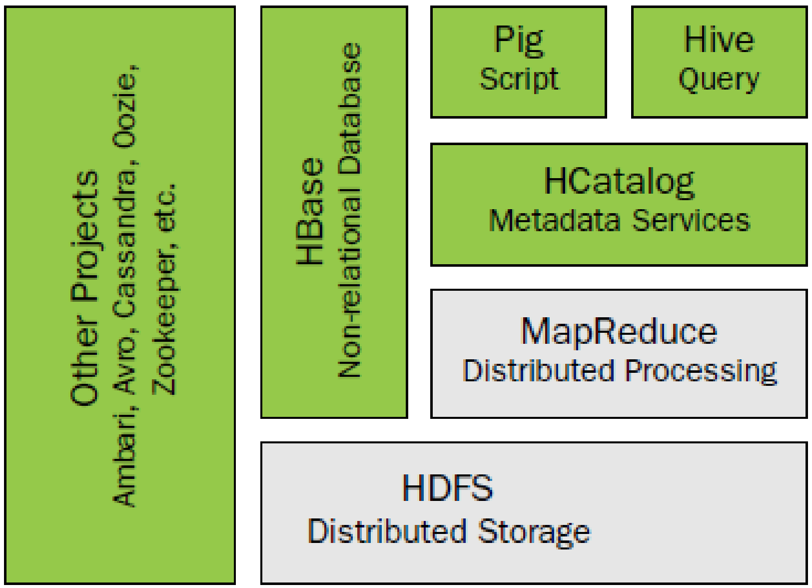


Figure 3.1 The Hadoop 1.0 ecosystem. MapReduce and HDFS are the core components, while other components are built around the core.

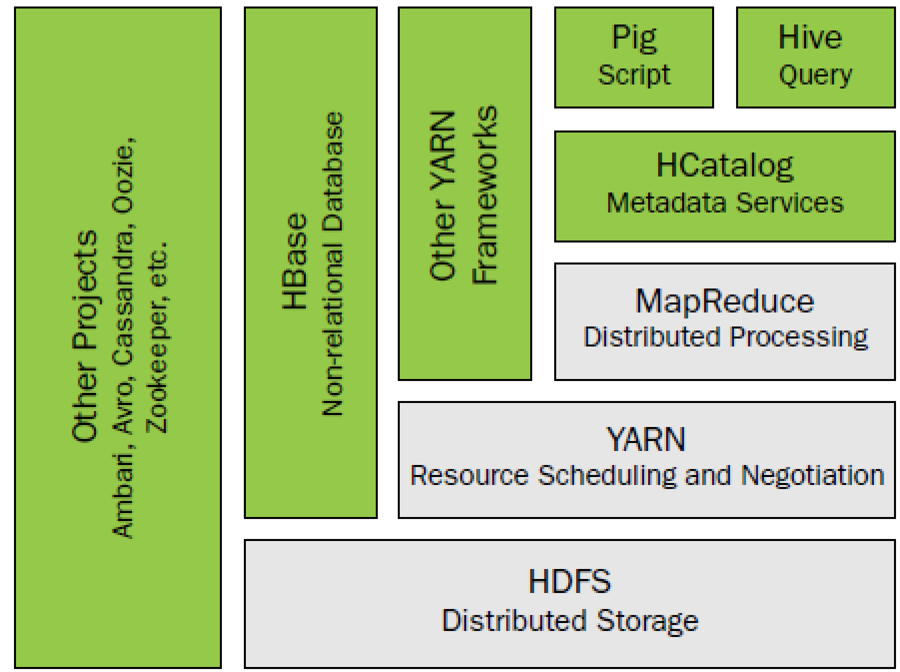


Figure 3.2 YARN adds a more general interface to run non-MapReduce jobs within the Hadoop framework

From “Apache Hadoop™ YARN Moving beyond MapReduce and Batch Processing with Apache Hadoop™ 2”, by Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec,Jeff Markham

Theoretical Issues in MR

Paper titles:

- “Upper and Lower Bounds on the Cost of a Map-Reduce Computation”
- “On the Computational Complexity of MapReduce”
- “A new Computation Model for Cluster Computing”
- “Fast Greedy Algorithms in MapReduce and Streaming”
- “Minimal MapReduce Algorithms”
- “Filtering: A Method for Solving Graph Problems in MapReduce”
- “A Model of Computation for MapReduce

MR Limitations

Difficult to design efficient/optimal algorithms

- everything must be expressed in key-value pairs and
- Manually programmed by users,
- Strictly following a 2-phase (Map→Reduce) programming paradigm

A lot of disk I/Os (mappers reading HDFS and writing local data)

A lot of network traffic (shuffling is expensive)

Difficult to handle data skew (the curse of the last reducer!)

Not very good for iterative processing (requires many MR stages)

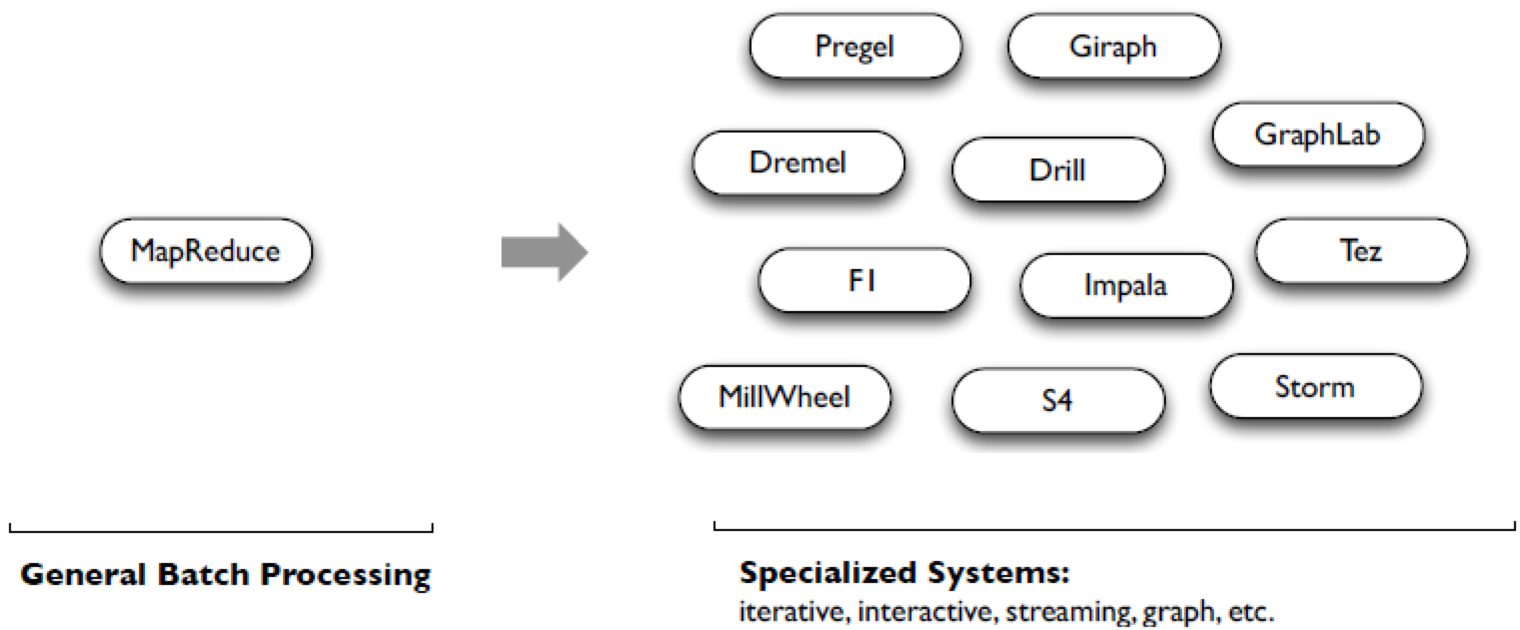
Not very good for **streaming applications**

MR Limitations (w.r.t. Graphs)

MapReduce is not the ideal platform for graph processing and mining graph objects, due to the **iterative** nature of most algorithms.

Each iteration in MapReduce is usually expensive because due to I/O operations in HDFS.

Need for Specialized Systems

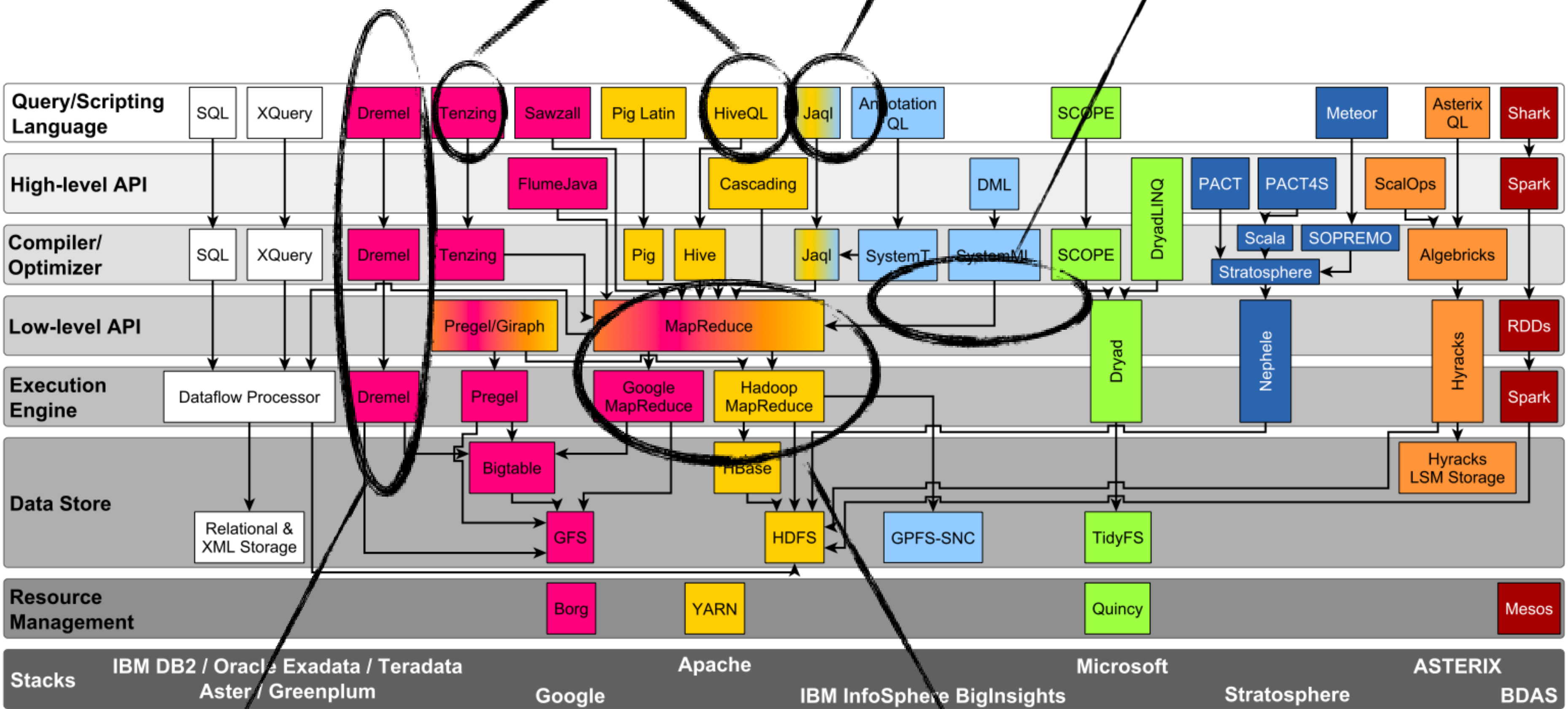


Too many ad-hoc solutions

SQL--

XQuery?

wrong platform?



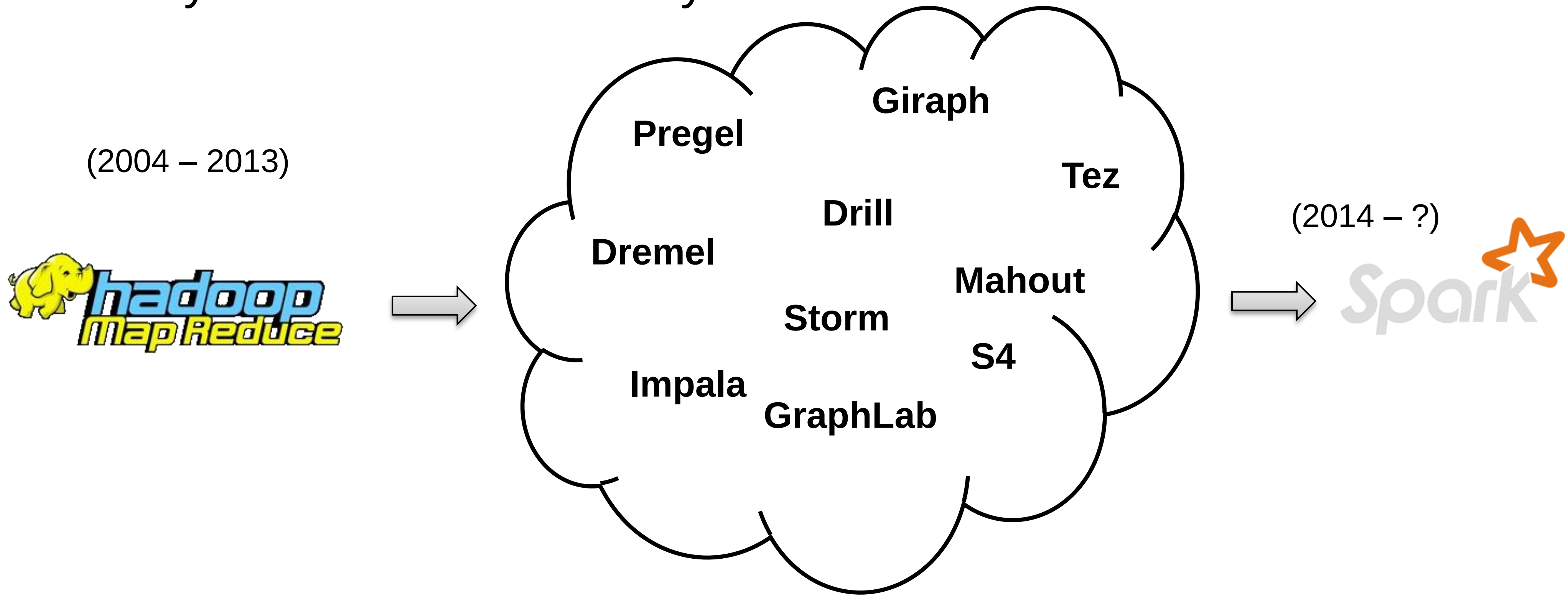
column store++

scalable parallel sort

Slide by Volker Markl

Spark

- Aims to create a unified cluster computing platform.
- Very successful as of today!



Specialized Systems

(iterative, interactive, ML, streaming, graph, SQL, etc)

General Batch Processing