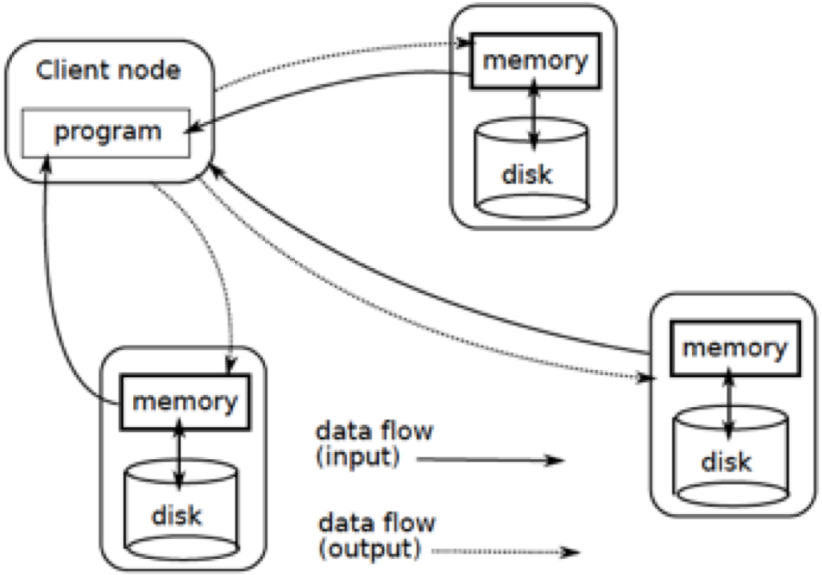
Frameworks for large scale data processing

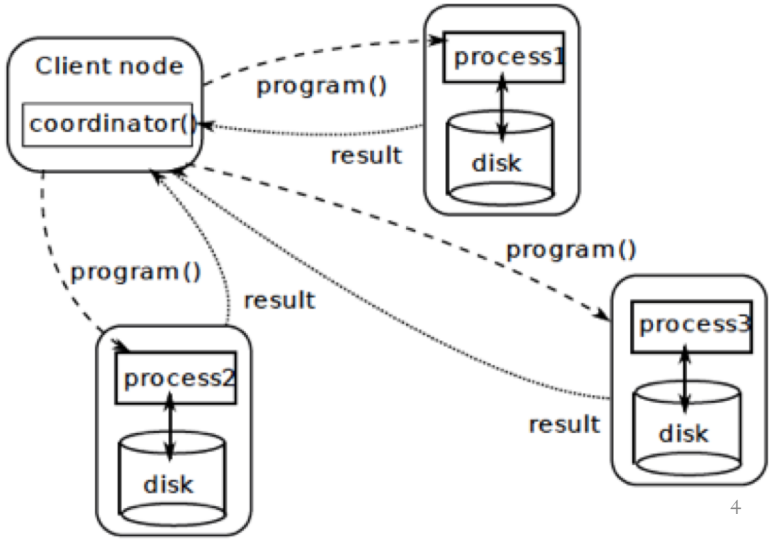
Introduction

Large-scale distributed (cluster) computing

- Provide a framework that
  - hides the complexity of distribution to developers (load balancing, fault tolerance)
  - provides data storage capabilities
  - focuses on the fundamental nuggets of the computation, supporting new computation models, for processing data in parallel
- Principles
  - Scale “out", not “up"
  - Move processing to data (data locality)
  - No focus on random access but on batch access
  - Tailored to analytical processing

Data locality





Large-scale data processing frameworks

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc…
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, R, and Python

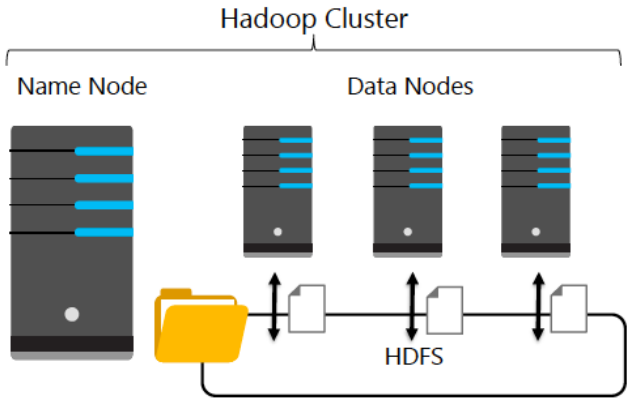
- MapReduce: A programming model (inspired by standard functional programming operators) to facilitate the development and execution of distributed tasks

Hadoop

MapReduce implementations

- Google MapReduce
  - Proprietary system
- Hadoop
  - Open-source MapReduce framework (Apache project)
  - Originally developed by Yahoo (but originates from Google MapReduce)

The core of Hadoop



- Master-slave architecture
- HDFS (Hadoop Distributed File System)
  - A distributed file system
  - Fault-tolerant
  - Data is replicated with redundancy across the cluster
  - CP: consistent and partition tolerant
- Large-scale data processing infrastructure based on the MapReduce programming paradigm
  - Provides a high level abstraction view (programmers do not need to take care of task scheduling and synchronization)
  - Fault-tolerant (node and task failures are automatically managed by the Hadoop system)

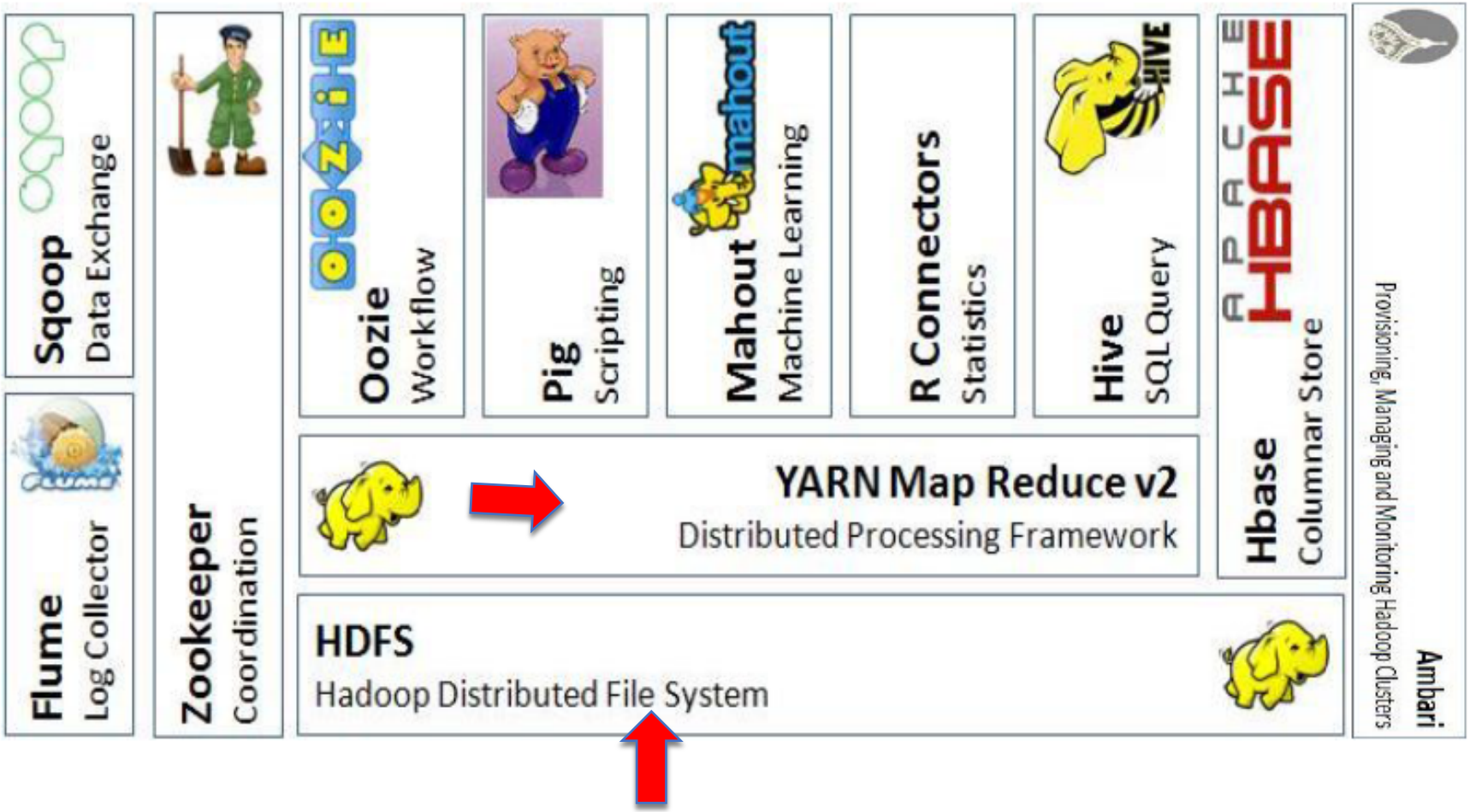
Large-scale data processing infrastructure in Hadoop

- Separates the what from the how

- MapReduce abstracts away the "distributed" part of the problem (scheduling, synchronization,etc.)
  - *Programmers focus on what*

- The distributed part (scheduling, synchronization,etc.) of the problem is handled by the framework
  - *The Hadoop infrastructure focuses on how*

Hadoop ecosystem



Introduction to HDFS

HDFS

- Large-scale distributed file system
- Simplest approach for large-scale distributed data storage

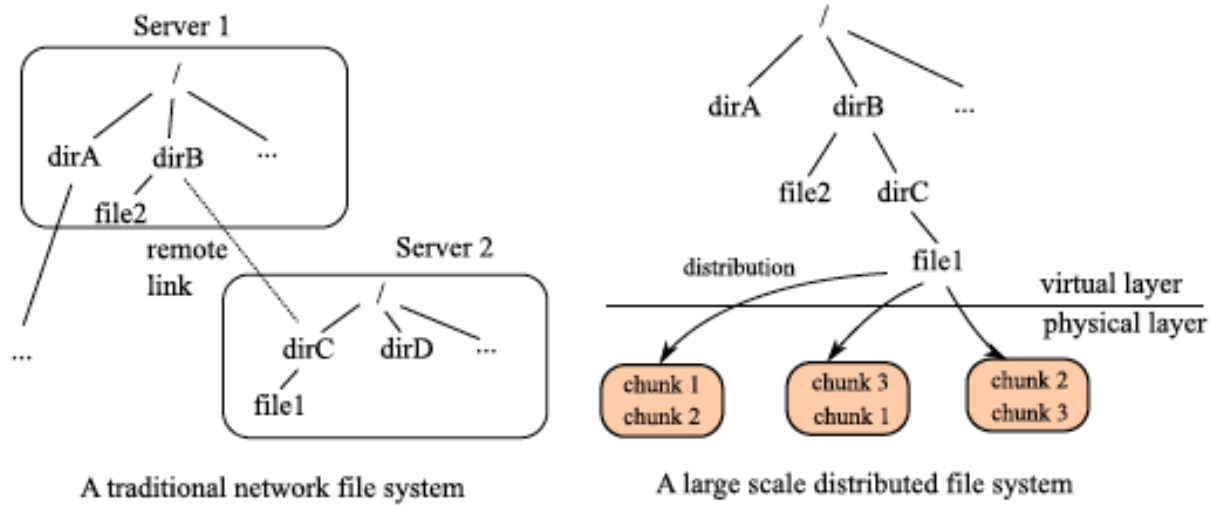
Reference scenario

- Batch processing
  - large amount of input data: modest number of huge (multi-gigabytes) files
  - runs a (possibly long lasting) job to process it
  - produces some output data

- Read-intensive scenarios
  - files are written-once, mostly appended to (perhaps concurrently)
  - high number of read requests of the whole file (limited random accesses)

The problem

- **Standard Network File System** (left part) does not meet scalability requirements (what if file1 gets really big?)



- **Distributed File System** storage, based on (i) a virtual file namespace, and (ii) partitioning of files in (possibly replicated) "chunks"

Data distribution

- Partitioning: the architecture works best for very large files (e.g., several Gigabytes), partitioned in large (64-128 MBs) chunks
  - this limits the metadata information to be managed
  - compare with 4-32 KB DBMS pages
  - *block-based approach*, chuncks defined based on the record position in the file
- Replication: usually, each partition is replicated 3 times
  - nodes holding copies of one chunk are located on different racks
  - *synchronous multi-leader replication protocol*
- Chunk size and the degree of replication can be set by the user

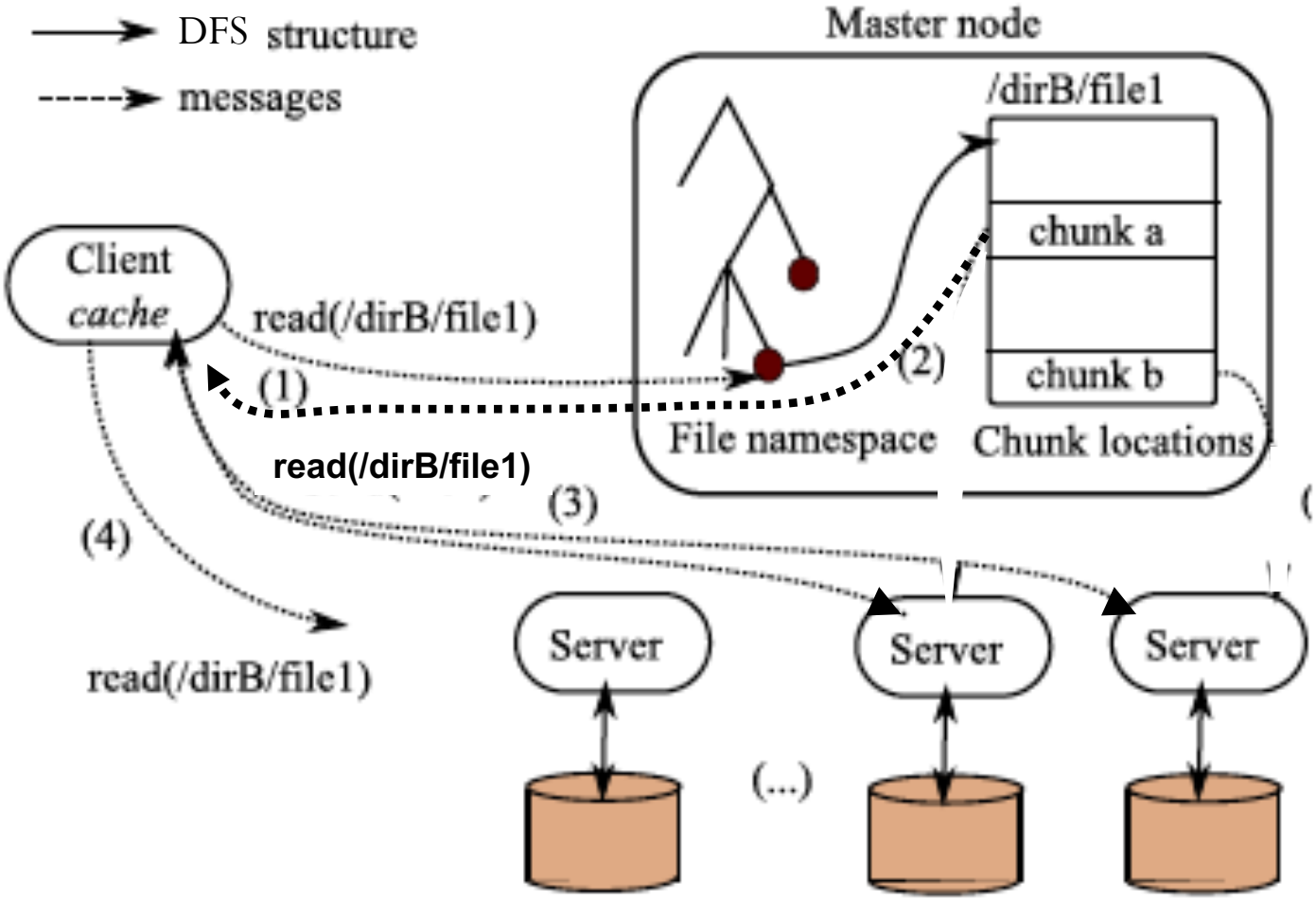
System architecture

- Master/slave architecture
- One Master node (NameNode)
  - performs administrative tasks: replication and balancing; garbage collection
  - manages the file system namespace
  - regulates access to files by clients for reads and writes
  - communications with the Master only involve transfer of metadata (limited data transfer)
  - might be replicated for fault tolerance
- Multiple Slave servers (DataNodes)
  - store chunks/partitions
  - store and retrieve the blocks when they are told to (by clients or the Master)
- Blocks are themselves stored on standard single-machine file systems
  - DFS lies on top of the standard Operating System stack

Client applications

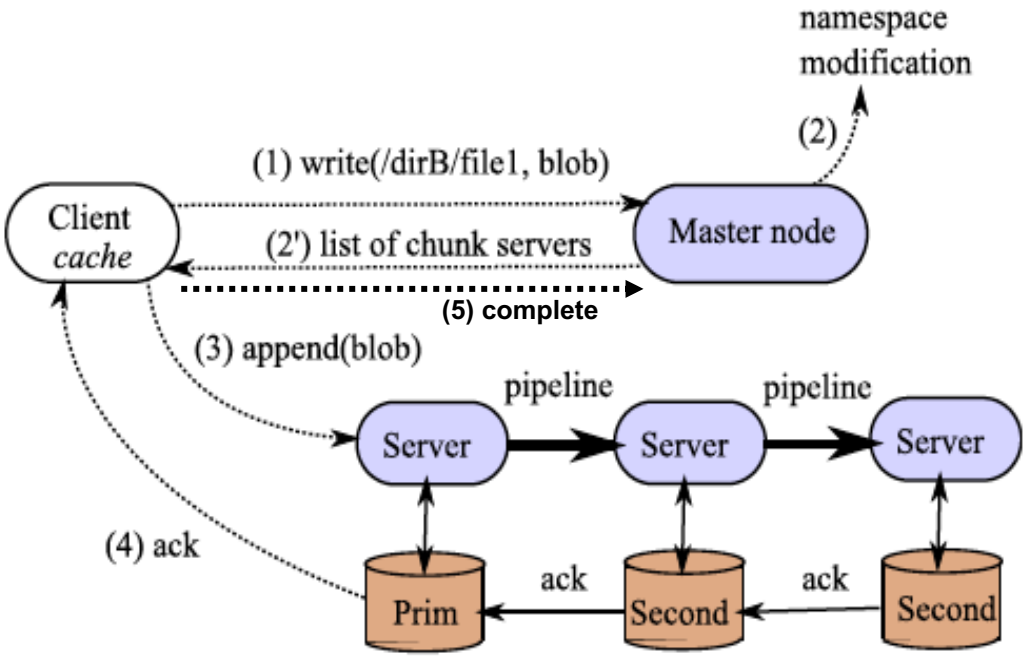
1. File access through specific APIs
2. Talk to the master node to find data/chunk servers associated with the file of interest
3. Connect to the selected chunk servers to access data
4. The Client keeps in its *cache* the addresses of the nodes accessed in the past, this knowledge can be used for subsequent accesses
   - improves scalability

File read


DFS structure
messages
Master node
/dirB/file1
chunk a
chunk b
File namespace
Chunk locations
Client
cache
read(/dirB/file1)
(1)
(2)
read(/dirB/file1)
(3)
(4)
read(/dirB/file1)
Server
Server
Server
(...)

File write (append)

- non-concurrent append() operation



- Multi leader, synchronous (one write on a given chunk at the time)
- The master returns the servers addresses ordered with respect to their distance from the client

Recovery management

- Logging at both Master and Server sites
- The Master sends heartbeat messages to servers, and initiates a replacement when a failure occurs (for *availability*)
- The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace
- If the NameNode machine fails, manual intervention is usually required

Summary

Feature	In HDFS Data Stores
Reference scenarios	Analytical
Architecture	Master-slave
Partitioning	Block-based
Replication	Multi-leader, synchronous
Consistency	Strong
Availability	Limited
Fault tolerance	Master-slave architecture, replication of the master node
CAP theorem	CP

Introduction to MapReduce

Typical Large-scale Problem

MAP

- Iterate over a large number of records in parallel
- Extract something of interest from each iteration
- Shuffle and sort intermediate results of different concurrent iterations
- Aggregate intermediate results
- Generate final output

REDUCE

Key idea: provide a functional abstraction for these two operations

MapReduce framework

An example: Word Count

- Input
  - A large textual file of words
- Problem
  - Count the number of times each distinct word appears in the file
- Output
  - A list of pairs <word, number of occurrences in the input file>

An example: Word Count

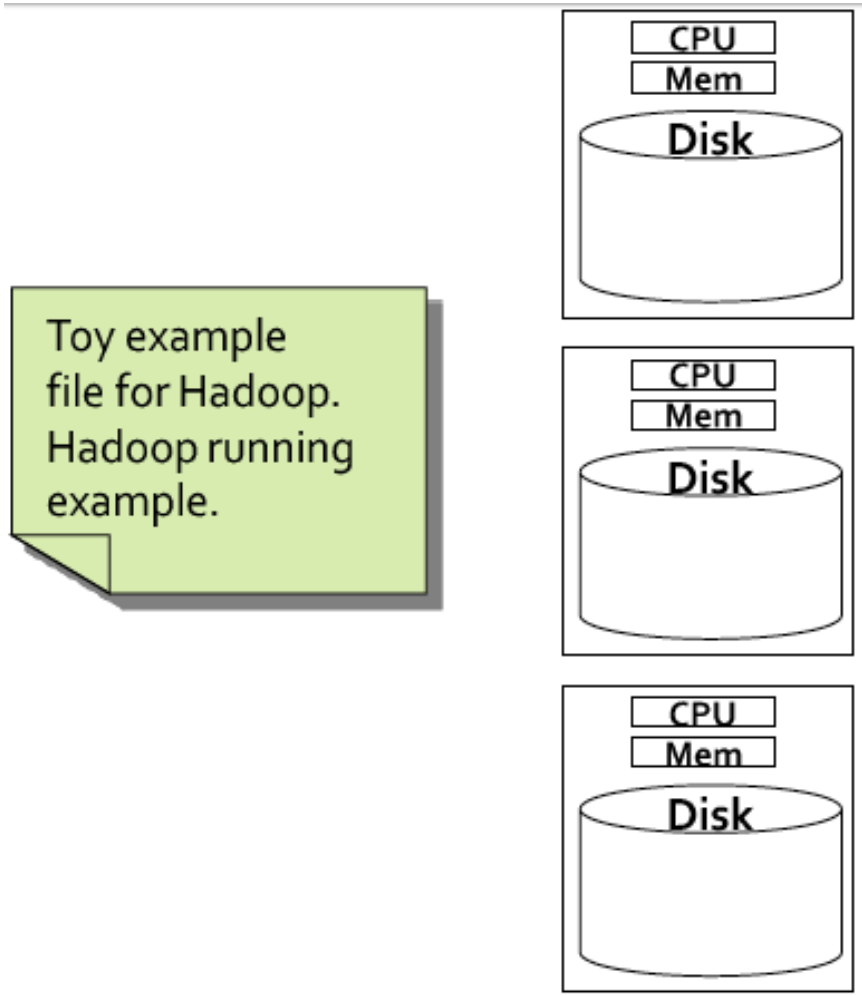
- Case 1: Entire file fits in main memory
  - A traditional single node approach is probably the most efficient solution in this case
  - The complexity and overheads of a distributed system impact negatively on the performance when files of few GBs are analyzed

An example: Word Count

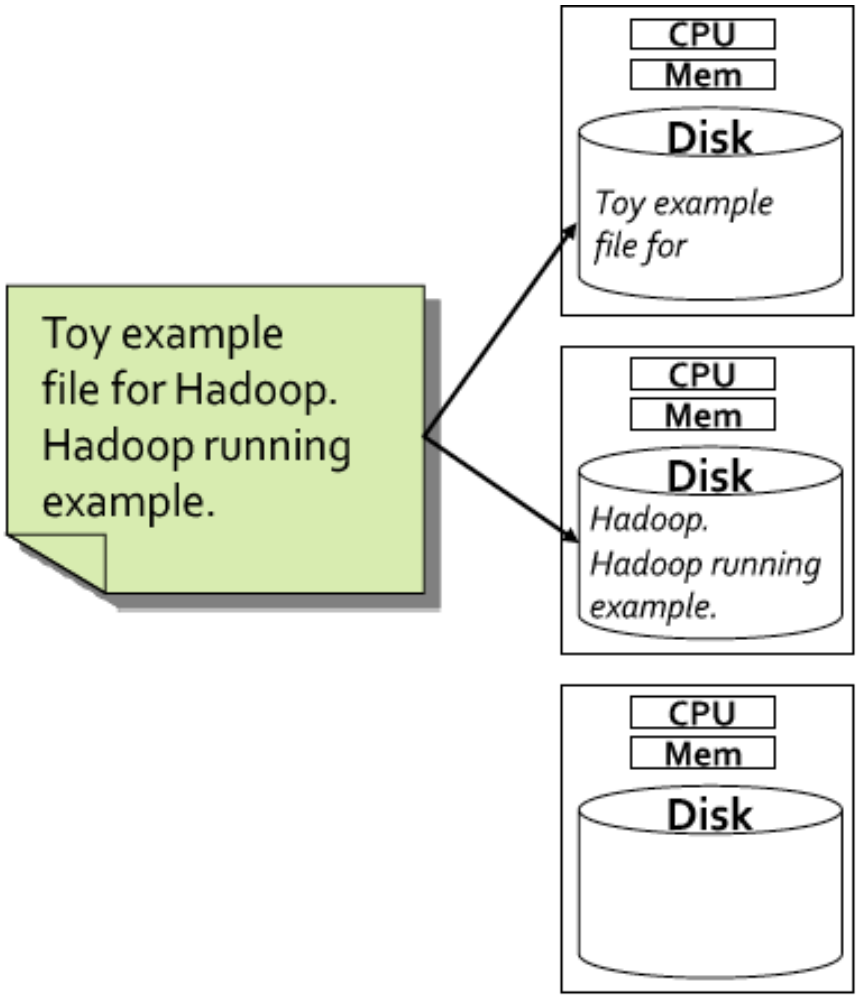
- Case 2: File too large to fit in main memory
  - Rely on a cluster
  - Distributed file system (e.g., HDFS) useful in this case: file partitioning and replication
  - How can we split this problem in a set of (almost) independent sub-tasks and execute it on the cluster?

- Suppose that
  - The cluster has 3 nodes
  - The content of the input file is
    - "Toy example file for Hadoop. Hadoop running example."
  - The input file is split in two chunks (number of replicas =1)

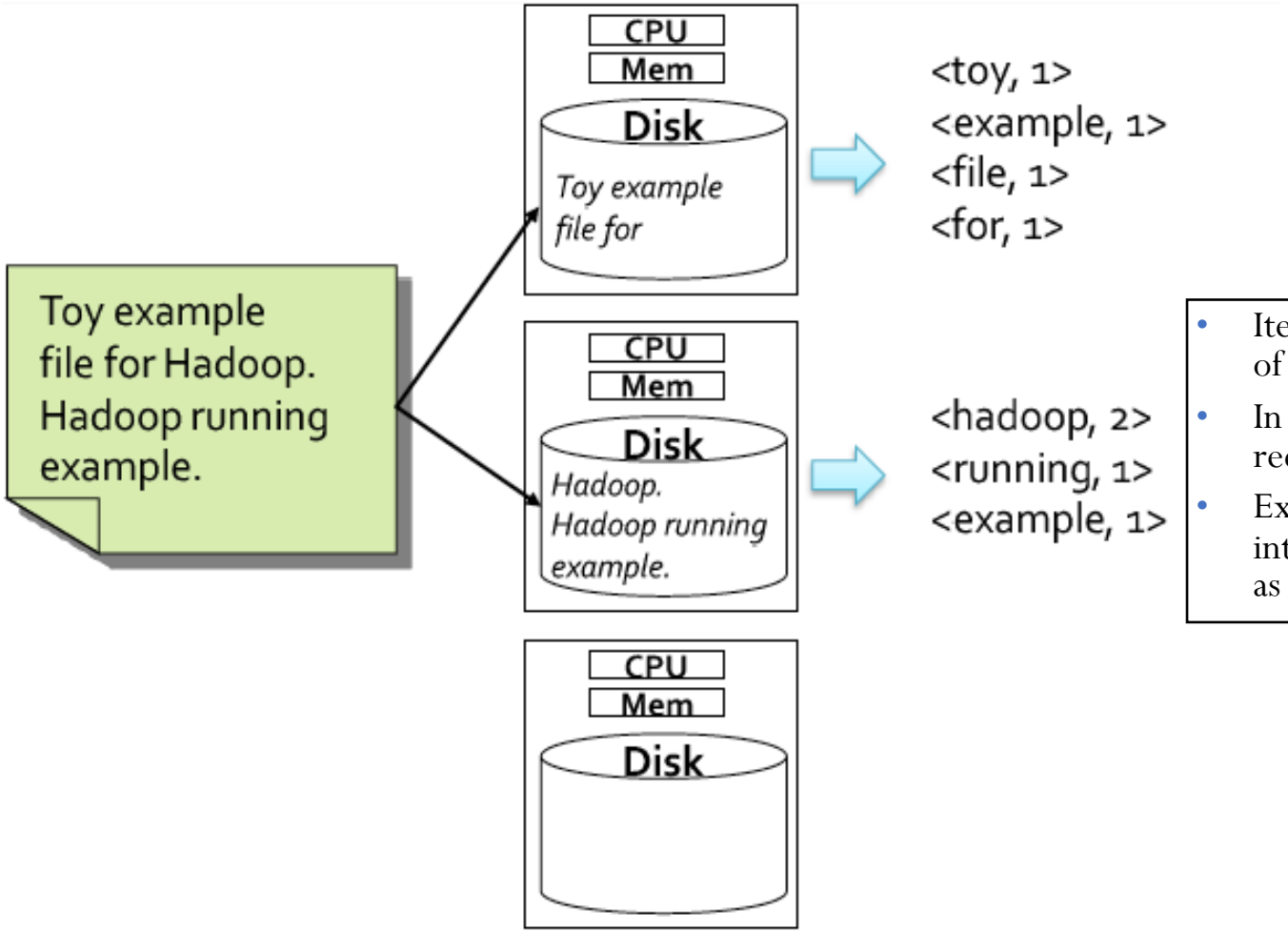
An example: Word Count – case 2



An example: Word Count – case 2



An example: Word Count – case 2



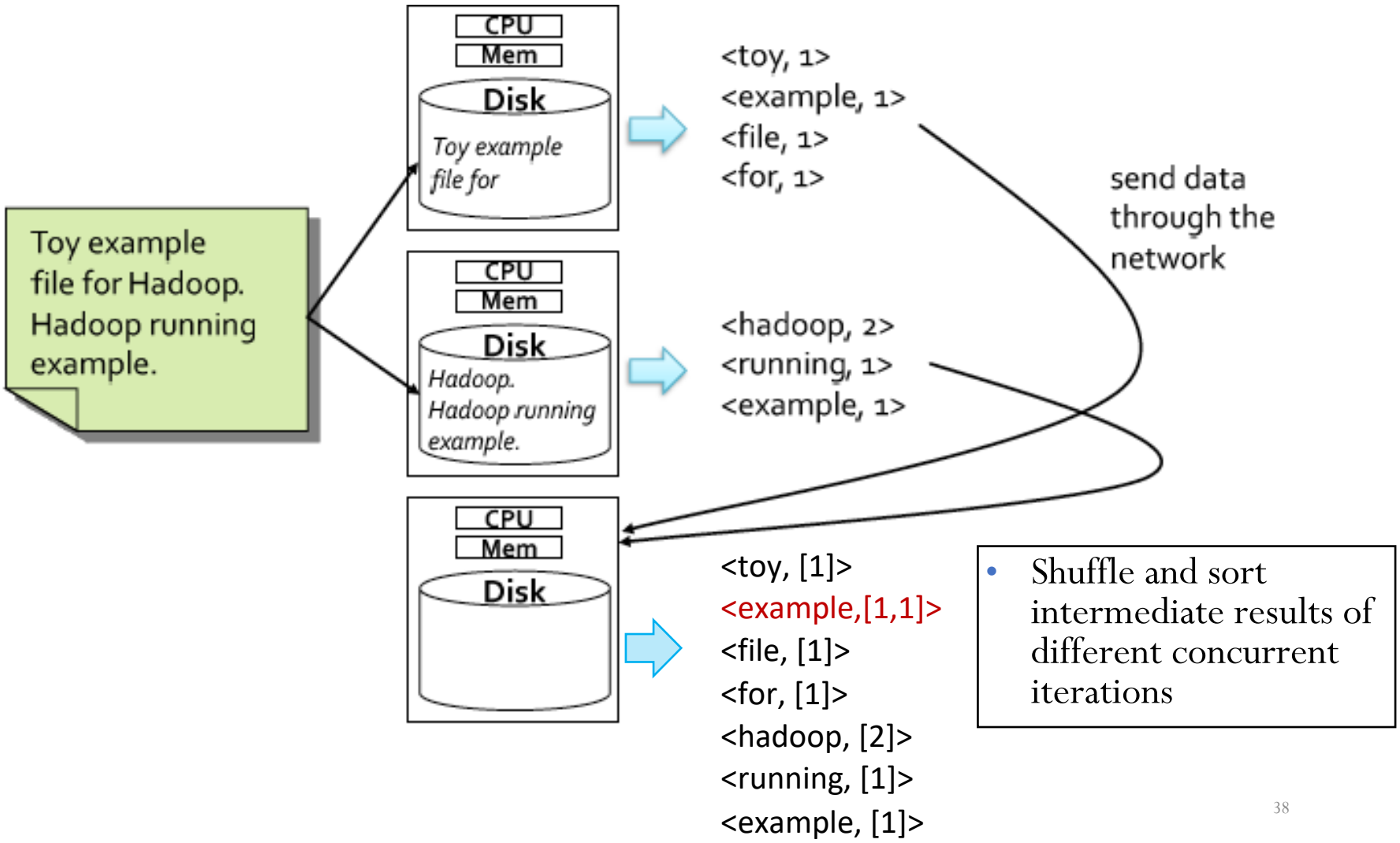
MAP

- Iterate over a large number of records in parallel
- In this example: each record, one text
- Extract something of interest from each iteration, as a set of pairs

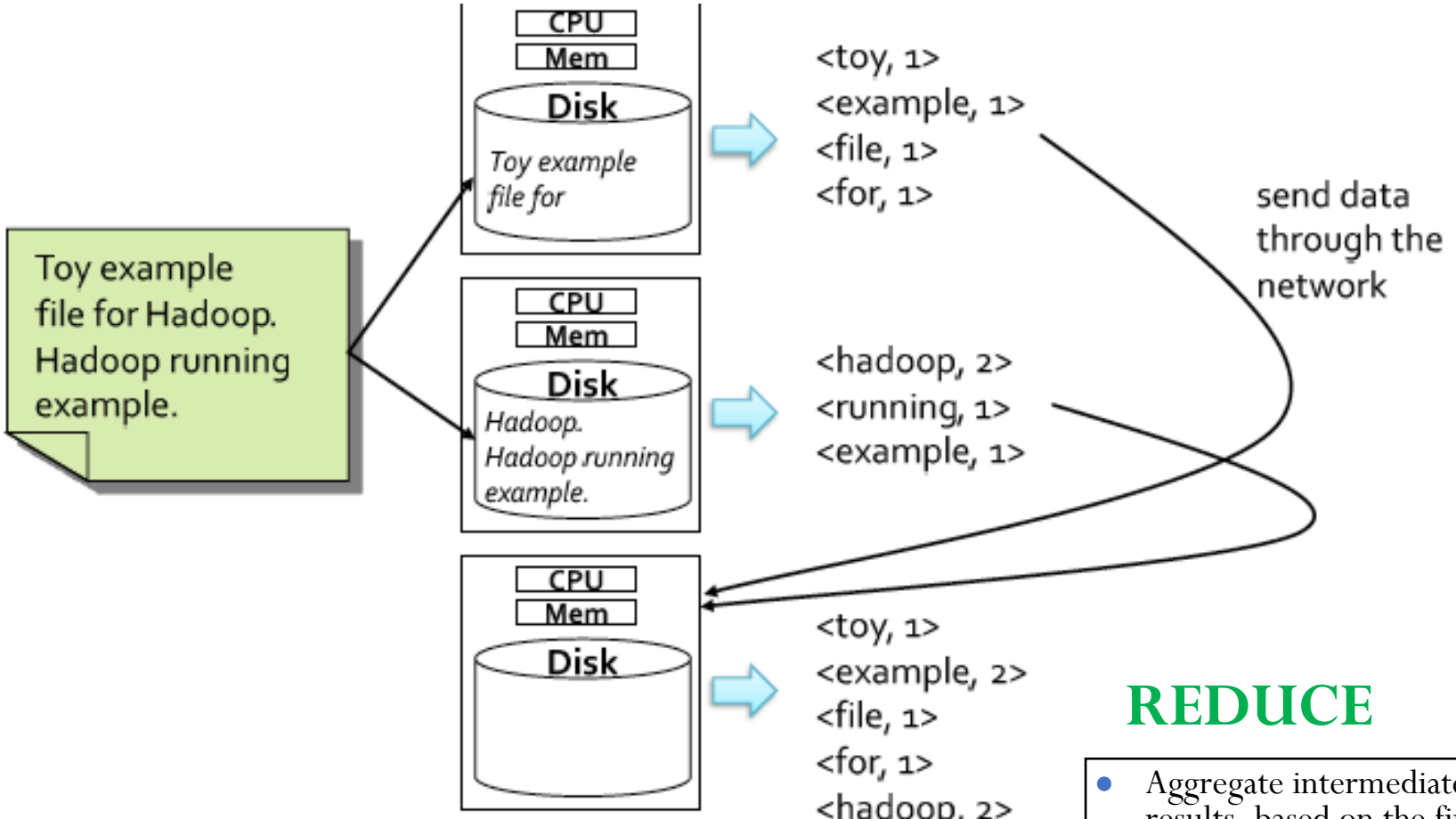
An example: Word Count – case 2

- The problem can be easily parallelized
  - Each server processes its chunk of data and counts the number of times each word appears in its chunk
  - Each server can perform it independently
- Synchronization is not needed in this phase

An example: Word Count – case 2



An example: Word Count – case 2



REDUCE

- Aggregate intermediate results, based on the first pair element
- Generate final output, as a set of pairs

An example: Word Count – case 2

- Each server sends its local (partial) list of pairs <word, number of occurrences in its chunk> to a server that is in charge of aggregating local results and computing the global list/global result
- The server in charge of computing the global result needs to receive all the local (partial) results to compute and emit the final list

- A simple synchronization operation is needed in this phase

An example: Word Count – a more realistic case - complexity

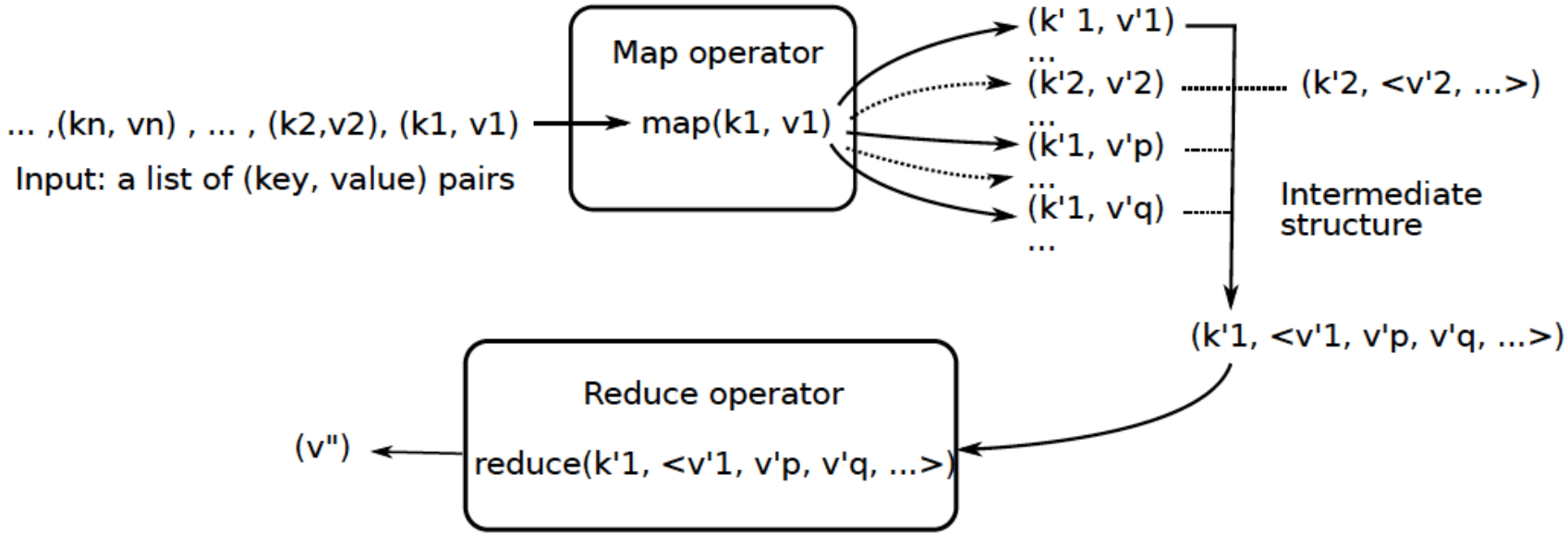
- Now suppose that
  - The file size is 100 GB and the number of distinct words occurring in it is at most 1.000
  - The cluster has 101 nodes
  - The file is optimally spread across 100 nodes and each of these servers contains one or many chunks (partitions) of the input file (1 GB, corresponding to 1/100 of the original file), stored in a distributed file system
- Each server reads 1GB of data from its local hard drive
  - Few seconds
- Each local list is composed of at most 1.000 pairs (because the number of distinct words is 1.000)
  - Few MBs
- The maximum amount of data sent on the network is 100 x size of local list (number of servers x local list size)
  - Some MBs

MapReduce programming paradigm in brief

- The MapReduce programming paradigm is based on the basic concepts of functional programming
- The programmer defines the program logic as two functions:
  - Map: do something to everything in a list
  - Reduce: combine/aggregate results of a list in some way
- The MapReduce environment takes in charge distribution aspects
- A complex program can be decomposed as a chain of Map and Reduce tasks

MapReduce programming paradigm in brief

Important: each pair, at each phase, is processed independently from the other pairs.



Network and distribution are transparently managed by the MapReduce environment.

MapReduce programming paradigm in brief

- The Map phase can be viewed as a transformation over each element of a data set
  - This transformation is a function m defined by the designer
  - Each application of m happens in isolation, so it can be parallelized
- The Reduce phase can be viewed as an aggregate operation
  - The aggregate function is a function r defined by the designer
  - Also the reduce phase can be performed in parallel since each group of key-value pairs with the same key can be processed by a distinct node
- The shuffle and sort phase is always the same
  - i.e., group the output of the map phase by key
  - it does not need to be defined by the designer

Back to the example – word count

- Input
  - A textual file (i.e., a list of words)
- Problem
  - Count the number of times each distinct word appears in the file
- Output
  - A list of pairs
  - <word, number of occurrences in the input file>

- *Preliminary issue:*
  - The content of each partition has to be interpreted as a set of (key-value) pairs

Back to the example – word count

Each partition = one (key, value) pair

Value: a list of words

Key: identifier of the list of words (e.g., the position of the text in the file)

```
map(key, value)
      //key: identifier, value: a list of words
      for each distinct word w in value
            emit(w,count(w,value))
```

```
reduce(key, values):
   // key: a word; value: a list of integers
   occurrences = 0
   for each c in values:
         occurrences = occurrences + c

   emit(key, occurrences)
```

Back to the example – word count

Each partition = one (key, value) pair

Value: a list of words

Key: identifier of the list of words (e.g., the position of the text in the file)

```
map(key, value)
      //key: identifier, value: a list of words
      for each word w in value
            emit(w,1)
```

```
reduce(key, values):
    // key: a word; value: a list of integers
    occurrences = 0
    for each c in values:
          occurrences = occurrences + c

    emit(key, occurrences)
```

LESS EFFICIENT

…WHY?

Back to the example – word count

Each partition = many (key, value) pairs

Value: one word

Key: identifier of the word (e.g., the position of the word in the partition)

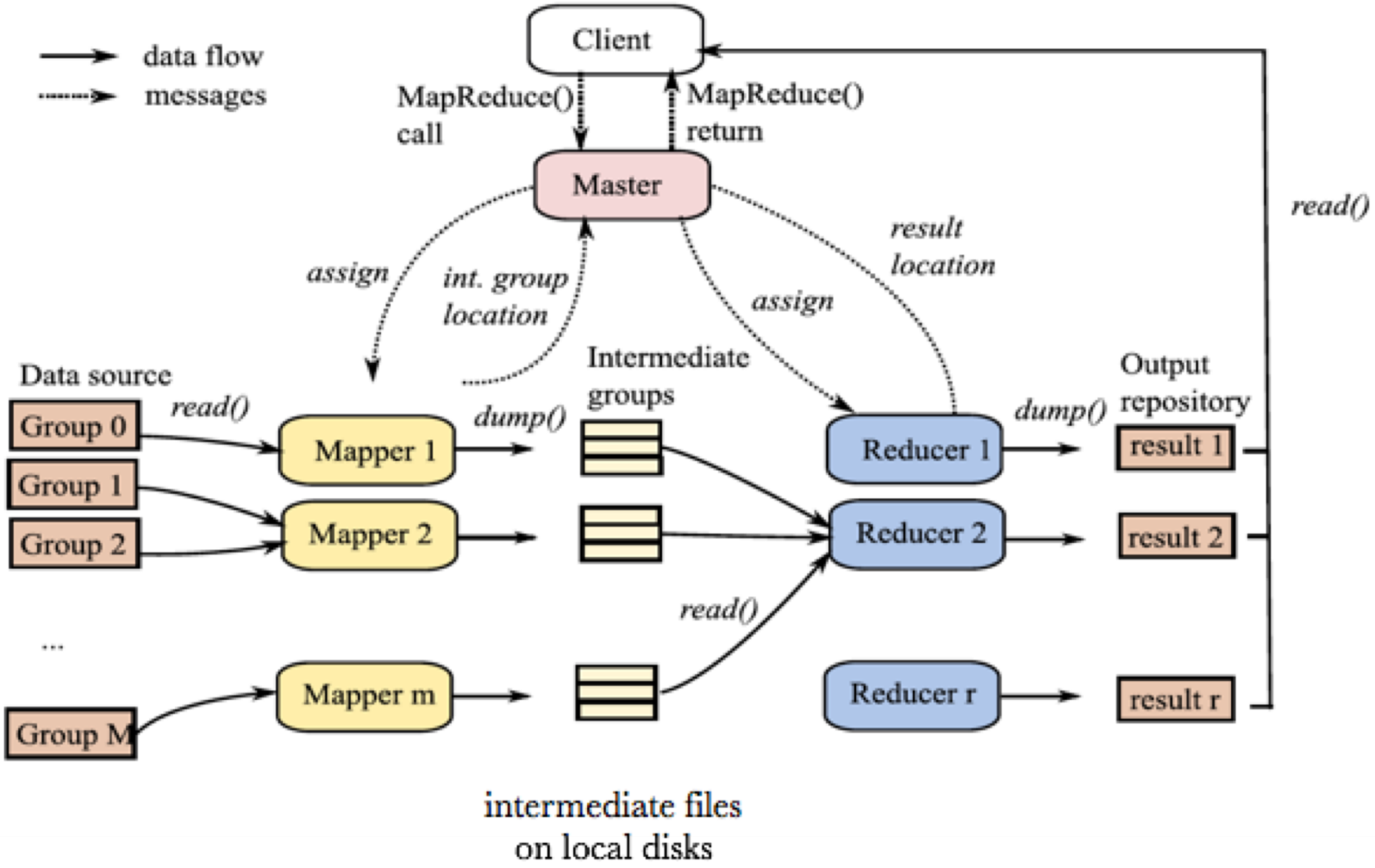
```
map(key, value)
      //key: identifier, value: word
            emit(value,1)
```

```
reduce(key, values):
    // key: a word; value: a list of integers
    occurrences = 0
    for each c in values:
          occurrences = occurrences + c

    emit(key, occurrences)
```

MapReduce data structures

- Key-value pair is the basic data structure in MapReduce
  - keys and values can be: integers, float, strings, ...
  - they can also be (almost) arbitrary data structures defined by the designer
- Both input and output of a MapReduce program are lists of key-value pairs
- *The design of a MapReduce program requires to impose the key-value structure on the input and output data sets*
  - for records in a file, input keys may correspond to their position in the file

Processing map reduce job



Spark

MapReduce limitations: programming overhead

- MapReduce is great for large-data processing
- MapReduce is a low level programming approach
- Very powerful but not easy from a programming point of view
  - writing Java programs (as in Hadoop) for everything is complex, verbose and slow
  - not everyone wants to (or can) write Java code

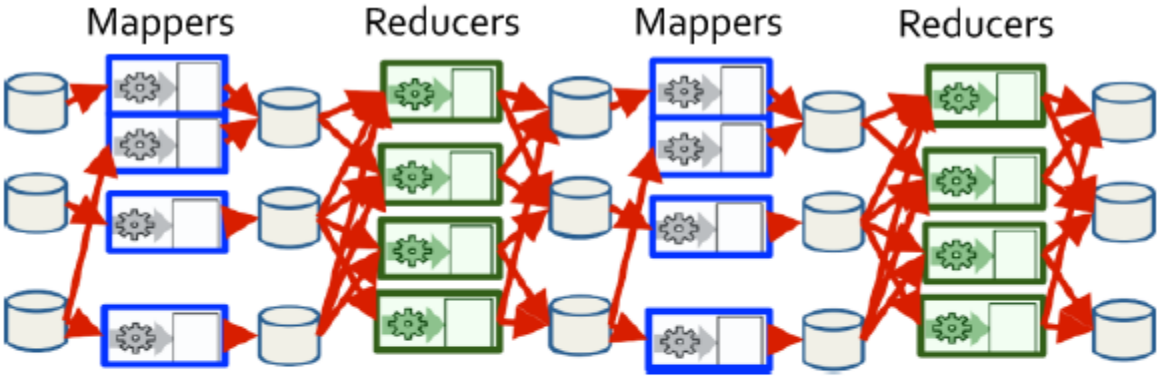
Example: MapReduce code

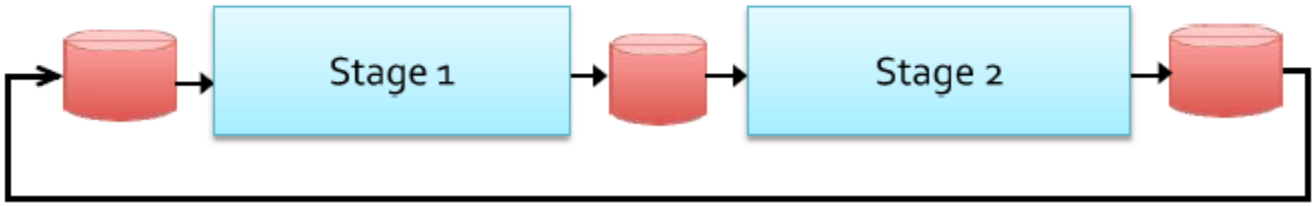
```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));

                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for use
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

Pig Slides adapted from Olston et al.

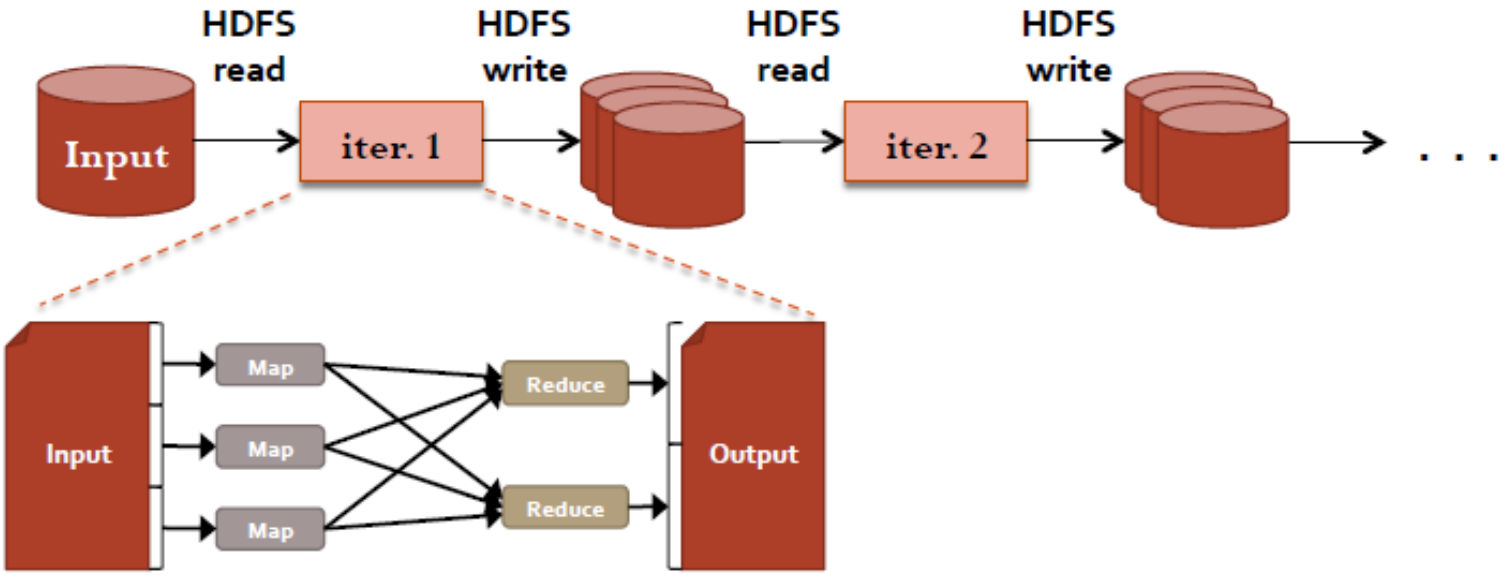
MapReduce limitations: iterative jobs

- Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage



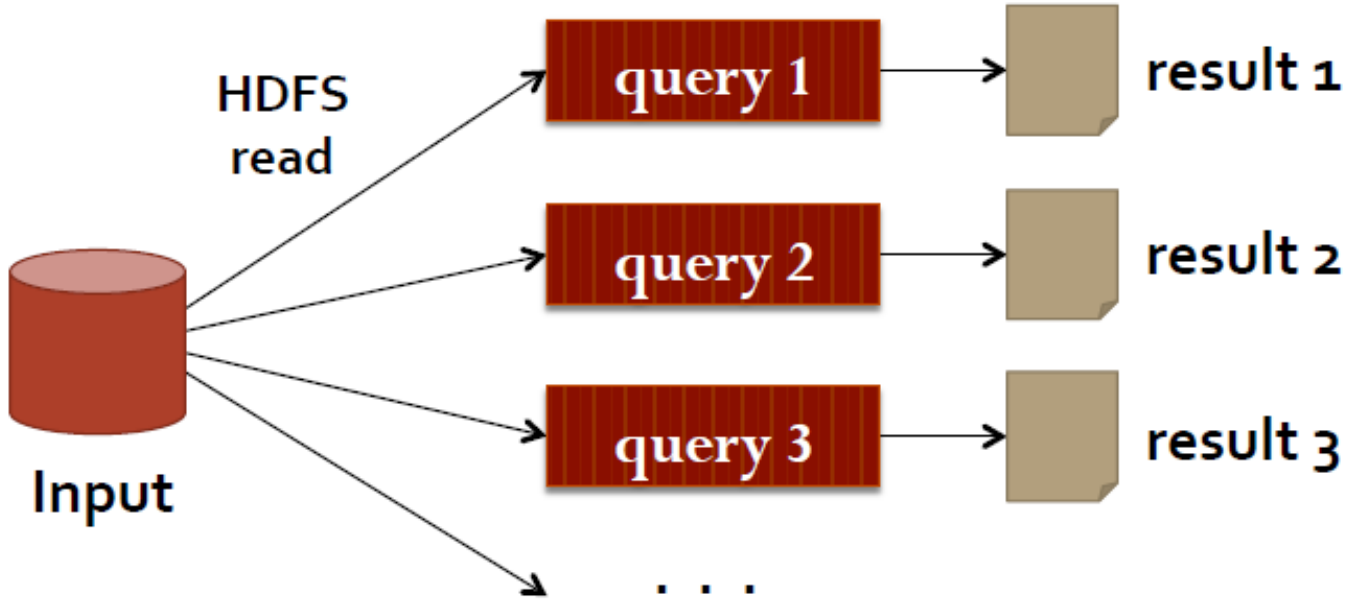


MapReduce limitations: iterative jobs



Iterative jobs are slow due to disk I/O, high communication, and serialization

MapReduce limitations: multiple reads of the same dataset

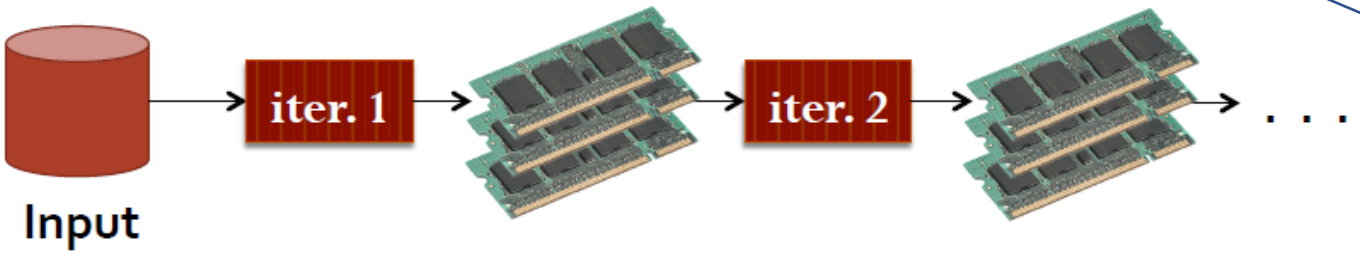


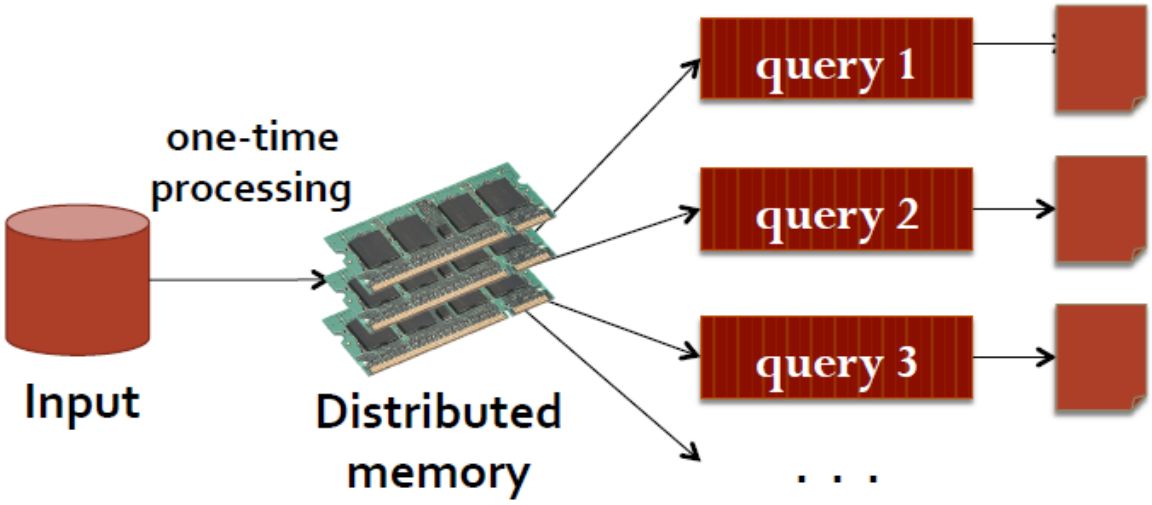
Why Spark?

- Motivation
  - Using MapReduce for complex iterative jobs or multiple jobs on the same data involves lots pf disk I/O
- Opportunity
  - The cost of main memory decreased
  - Hence, large main memories are available in each server
- Solution
  - Keep more data in main memory
  - Enabling data sharing in main memory as a resource of the cluster

Data sharing in Spark

Data is shared between the iterations by using the main memory
(or at least part of it)

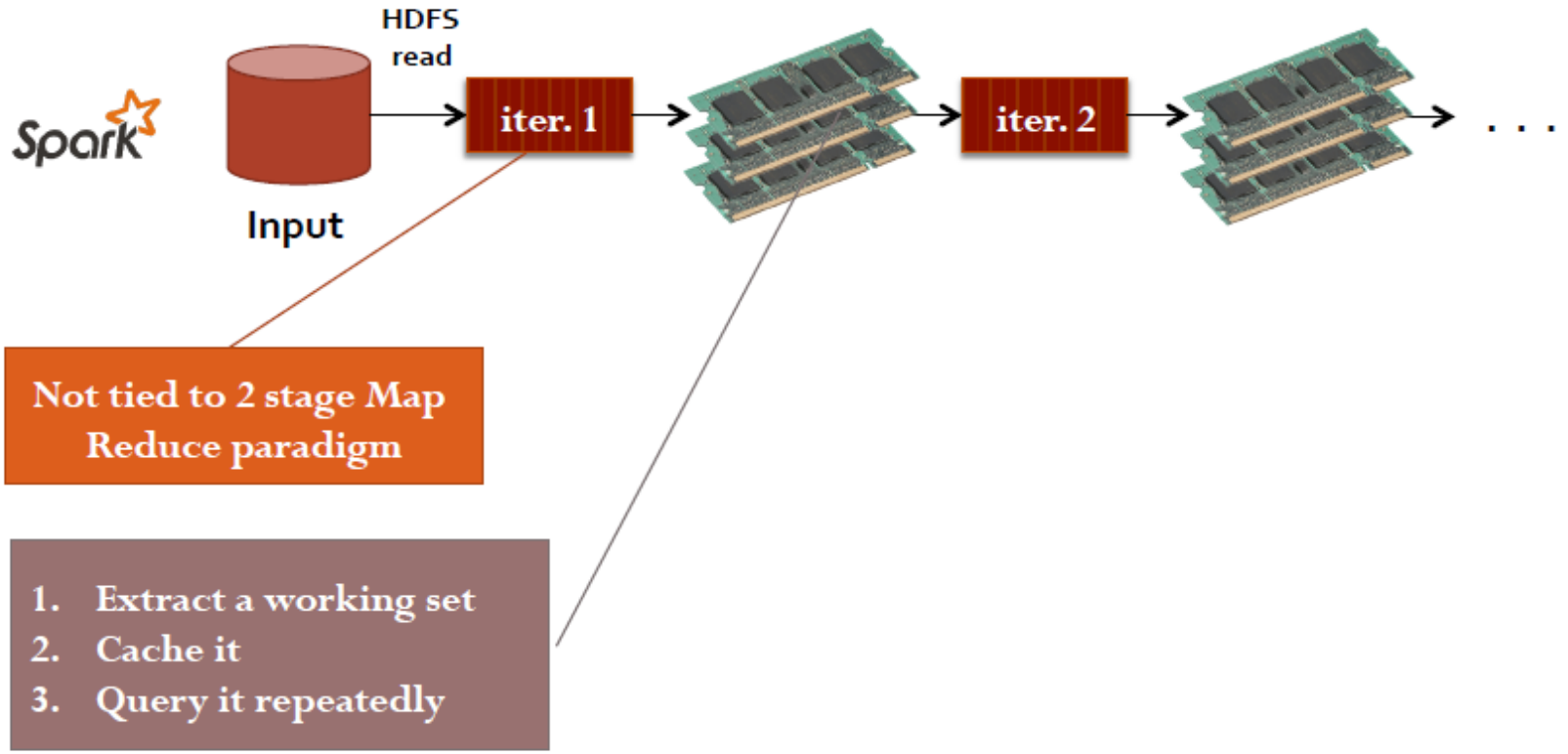




Data is read only once from HDFS and stored in main memory
Split of the data across the main memory of each server

10-100x faster than network and disk

Spark data flow



Spark data flow

- Everything you can do in Hadoop, you can also do in Spark
- Spark's computation paradigm is not "just" a single MapReduce job, but a multi-stage, in-memory dataflow graph based on Resilient Distributed Datasets (RDDs)
- Data are represented as RDDs
  - Partitioned/distributed collections of objects spread across the nodes of a cluster
  - Stored in main memory (when it is possible) or on local disk
- Spark programs are written in terms of operations on RDDs

Spark computing framework

- Provides a programming abstraction (based on RDDs) and transparent mechanisms to execute code in parallel on RDDs

- Manages job scheduling and synchronization
- Manages the split of RDDs in partition and allocates RDDs' partitions in the nodes of the cluster

- Hides complexities of fault-tolerance and slow machines
  - RDDs are automatically rebuilt in case of machine failure (lazy evaluation)