

MAP/REDUCE PROGRAMMING

BIG DATA

Félix Cuadrado

felix.cuadrado@qmul.ac.uk

Workshop on Scientific Applications for IoT

ICTP, Trieste, Italy. 23rd March 2015

Contents

- **Parallel Computing**
- The Map/Reduce Programming Model
- Apache Hadoop
- Map/Reduce Programming
- Dataflow languages

Parallel Computing

- The use of a number of processors, working together, to perform a calculation or solve a problem.
- The calculation will be divided into tasks, sent to different processors.
 - **Processor coordination** will be required
- Processors can be different cores in the same machine, and/or different machines linked by a network

Parallel computing is hard

- Parallel Computing is generally very hard, because:
 - Many algorithms are hard to divided into subtasks (or cannot be divided at all)
 - The subtasks might use results from each other, so coordinating the different tasks might be difficult
- Some problem areas are much easier than others to parallelize

Why Parallelize?

- There are many problems we cannot solve by running them on a simple processor/machine. They are:
 - Too large (do not fit in one machine)
 - Take too long

Parallel computing can provide faster results, and it can even be cheaper

Sequential Program Execution

- Basic model based on Von Neumann architecture in the 40s
- One instruction is fetched, decoded and executed at a time
- As processor speed increases, more instructions can be executed in the same time

Single processor limitations

- We have roughly reached the practical limitations in the amount of computing power a single processor can have
 - We cannot make processors much faster, or much bigger
- According to Moore's law the number of transistors per chip will continue to increase
- But this means now we have chips containing an increasing amount of processors
 - Multicore chips

Our first parallel program

- Task: count the number of occurrences of each word in one document
- Input: text document
- Output: sequence of: word, count
 - The 56
 - School 23
 - Queen 10
 - ...

Program Input

Queen Mary University of London (QMUL) has been ranked among the top 100 universities in the world in the latest edition of the respected QS World University Rankings, released today (Tuesday 16 September). QMUL has risen almost 50 places in the last two years, and is now listed at 98th position globally.

Principal and President of Queen Mary, Professor Simon Gaskell, comments: “The improvement in our ranking over the last two years is a tremendous achievement that stems from the hard work and achievements of all our staff.

“It is also evidence of Queen Mary’s increasingly prominent role in global academia, and a sign of our ongoing reputation as a destination for the very best students, inspiring teachers, and leading researchers from across the world.” QMUL is rated particularly highly for the number of international students on campus. With students and staff from more than 150 countries, it is ranked as the world’s 25th ‘most international’ university for students.

The university is ranked 19th amongst UK institutions and 10th in the UK for both research impact and staff to student ratio.

2014 sees the tenth anniversary of the QS World University Rankings, which are based on 90,000 survey responses. More than 3,000 universities were considered for ranking. QS is the only global ranking to have been independently scrutinised and IREG Approved.

How to solve the problem on a single processor?

```
List<String> words= text.split();
Hashtable<String,Integer> count = new Hashtable();
for (String word: words){
    if(count.containsKey(word){
        count.put(word, count.get(word)+1);
    }
    else{
        count.put(word,1)
    }
}
```

Parallelizing the problem

- Splitting the load on subtasks:
 - Split sentences/lines into words
 - Count all the occurrences of each word
- ...What do we do with the intermediate results?
 - Merge into single collection
 - Possibly requires parallelism too

Contents

- Parallel Computing
- **The Map/Reduce Programming Model**
- Apache Hadoop
- Map/Reduce Programming
- Dataflow languages

MapReduce

- *“A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.”*

Dean and Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Google Inc.

- More simply, MapReduce is:
 - A parallel programming model and associated implementation.

MapReduce Programming Model

- Process data using special **map()** and **reduce()** functions
 - The map() function is called on every item in the input and emits a series of intermediate key/value pairs
 - All values associated with a given key are grouped together
 - The reduce() function is called on every unique key, and its value list, and emits a value that is added to the output
- More formally,
 - $\text{Map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
 - $\text{Reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(k_2, v_2)$

Example: word count

```
public void Map (String filename,  
                String text) {  
  
    List<String> words= text.split();  
    for (String word: words) {  
        emit(word, 1)  
    }  
  
}
```

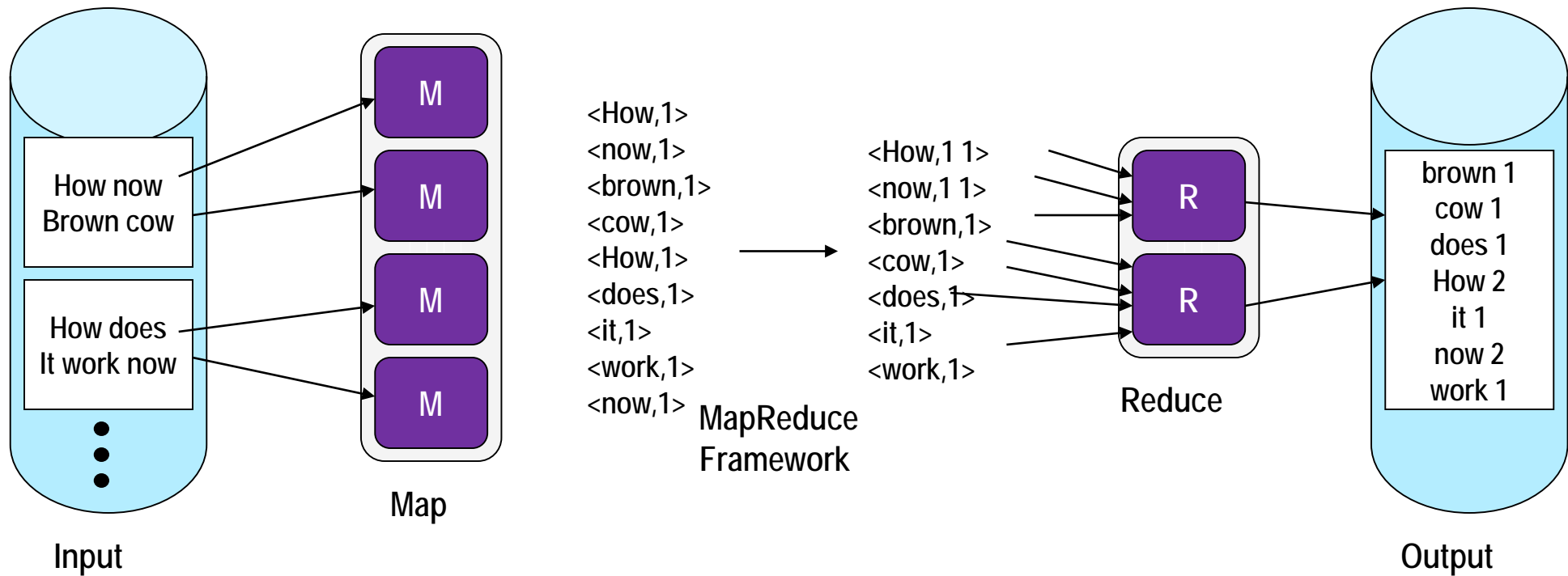
Example: word count

```
public void Reduce (String key,  
                    List<Integer> values) {  
    int sum = 0;  
    for (Integer count: values) {  
        sum+=count;  
    }  
    emit(key, sum);  
}
```


How MapReduce parallelizes

- Input data is partitioned into processable chunks
- One Map job is executed per chunk
 - All can be parallelized (depends on number of nodes)
- One Reduce Job is executed for each distinct key emitted by the Mappers
 - All can be parallelized (partitioned 'evenly' among nodes)
- Computing nodes first work on Map jobs. After all have completed, a synchronization step occurs, and they start running Reduce jobs

Word Count Example



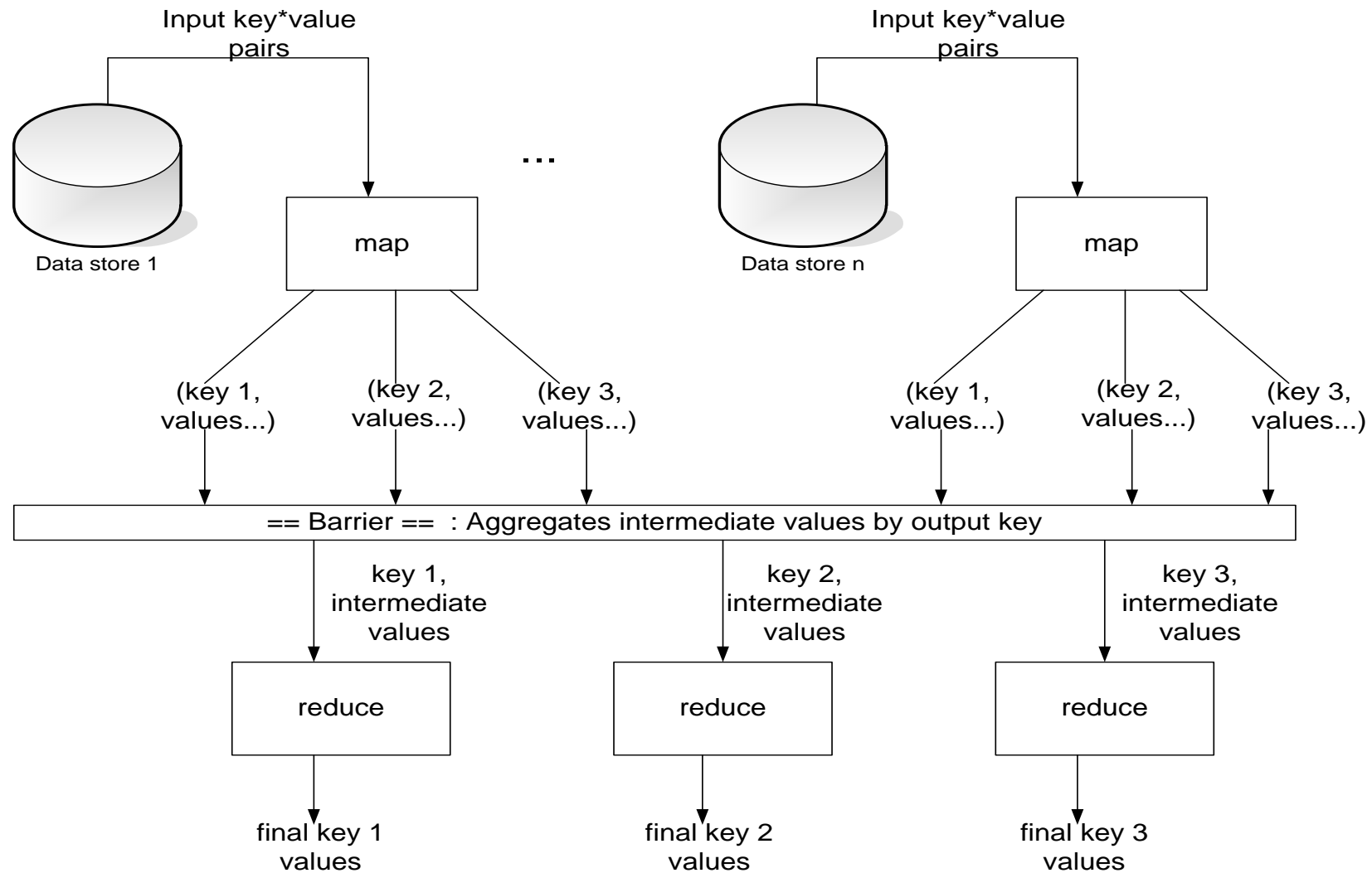
A Brief History

- Inspiration from functional programming (e.g., Lisp)
 - map() function
 - Applies a function to each **individual** value of a sequence to create a **new list** of values
 - Example: square $x = x * x$
map square [1,2,3,4,5] returns [1,4,9,16,25]
 - reduce() function
 - Combines all elements of a sequence using a binary operator
 - Example: sum = (each elem in arr, total +=)
reduce [1,2,3,4,5] returns 15 (the sum of the elements)

MapReduce Benefits

- High level parallel programming abstraction
- Framework implementations provide good performance results
- Scalability close to linear with increase in cluster size
- Greatly reduces parallel programming complexity
- However, it is not suitable for every parallel programming algorithm!

Synchronization and message passing



Shuffle and Sort steps

- Every key-value item generated by the mappers is collected
 - items are transferred over the network
- Same key items are grouped into a list of values
- Data is partitioned among the number of Reducers
- Data is copied over the network to each Reducer
- The data provided to each Reducer is sorted according to the keys


MapReduce Runtime System

- Partitions input data
- Schedules execution across a set of machines
- Handles load balancing
- Shuffles, partitions and sorts data between Map and Reduce steps
- Handles machine failure transparently
- Manages inter process communication

Contents

- Parallel Computing
- The Map/Reduce Programming Model
- **Apache Hadoop**
- Map/Reduce Programming
- Dataflow languages

The Apache Hadoop project

- The brainchild of Doug Cutting (Yahoo) 
- Open source project hosted at Apache
- Started in 2007 when code was spun out of Nutch
- Has grown into a large top-level project at Apache with significant ecosystem
 - V2 (YARN, 2013) structures it as a generic platform
 - Even third-party distros *a la* Linux (Cloudera, Hortonworks)

Hadoop Physical Requirements

- Designed to run in clusters of commodity PCs
 - Leverages heterogeneous capabilities
- Scales up to thousands of connected machines
- Suitable for Local Networks / DataCenters
 - Rack servers connected over a LAN
 - Clusters distributed over the Internet are not feasible
 - Network would become an enormous bottleneck

Hadoop Architecture

- Hadoop executes on a cluster of networked PCs
- Each node runs a set of daemons

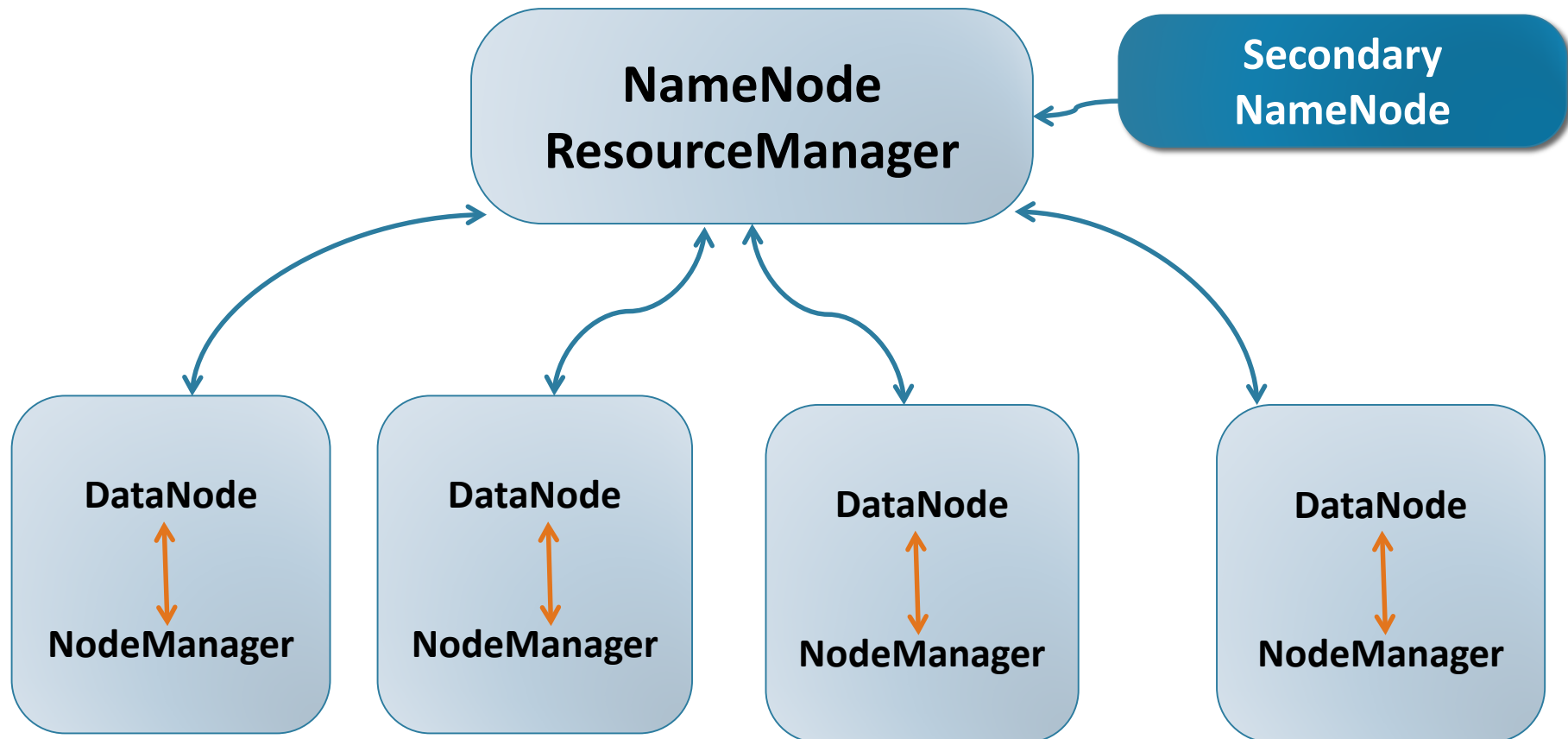
- ResourceManager
- NodeManager

Computing

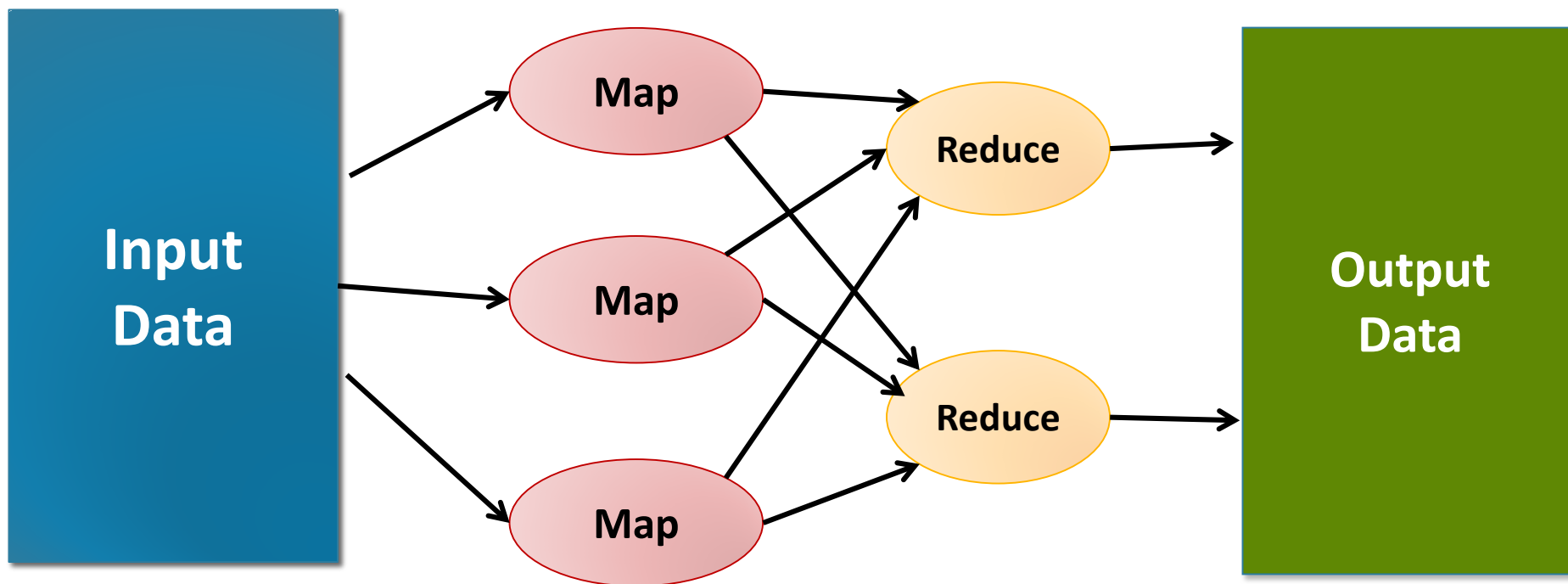
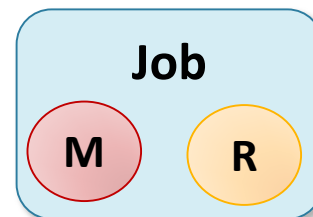
- NameNode
- SecondaryNameNode
- DataNode

Storage

Hadoop Master-Slave architecture



Map/Reduce job



Hadoop job

- A Hadoop Job is packaged as a Jar file containing all the code for Mapper and Reducer functions
- The job is assigned a cluster-unique id
 - A set number of reattempts is managed for job tasks
- The file is replicated over the Hadoop nodes
 - Move computation to the data

Shuffling: @Mapper

1. All key value pairs are **collected**
in-memory buffer (100MB default size), spills to HD
2. Pairs are **partitioned** depending on target reducer
each partition is sorted by key
3. **Combiner** runs on each partition
4. Output is available to the Reducers through HTTP server threads

Sort: @Reducer

1. The reducer **copies** output from mappers
 - asks ApplicationManager for map output locations
2. Downloaded output is **merged** and **sorted** into the full input for the Reducer
 - List of $\langle k_2, \text{list}\langle v_2 \rangle \rangle$, sorted by k_2

Contents

- The Apache Hadoop project
- Hadoop job execution: YARN
- **Hadoop storage: HDFS**
- The Combiner

HDFS

- HaDooop Distributed Filesystem
 - Shared storage among the nodes of the Hadoop cluster
- Storage for Input and output of MapReduce jobs
- HDFS is Tailored for MapReduce jobs
 - Large block size (64MB default)
 - But not too large, blocks define the minimum parallelization unit
- HDFS is not a POSIX compliant Filesystem
 - Tradeoffs for improving data processing throughput

HDFS Data distribution

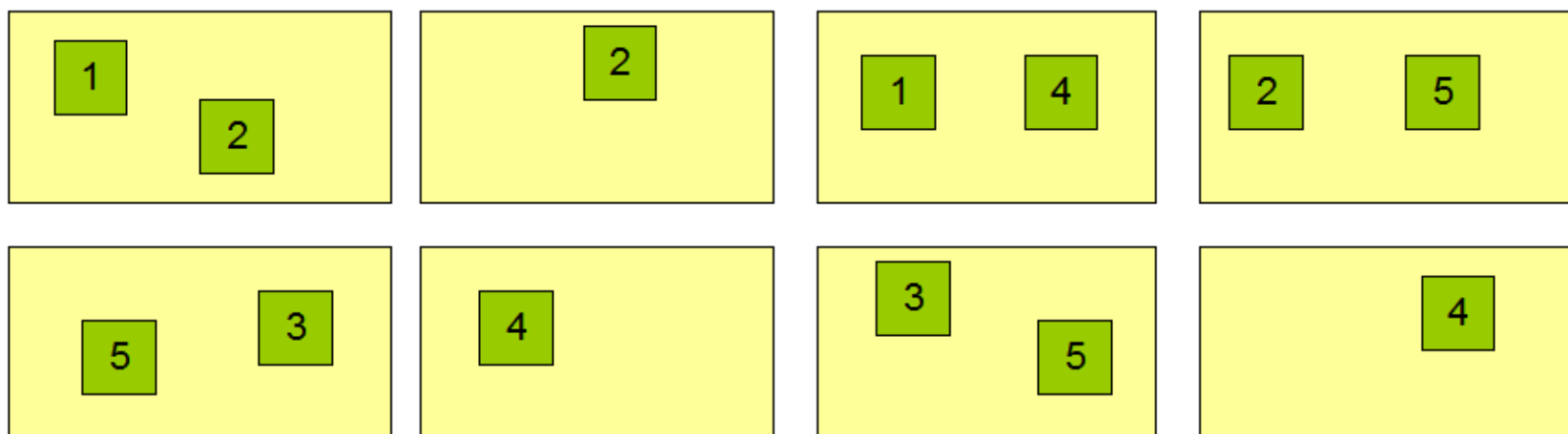
- Data distribution is a key element of the MapReduce model and architecture
- “Move computation to data” principle
- Blocks are replicated over the cluster
 - Default ratio is three times
 - spread replicas among different physical locations
 - Improves reliability

Data replication

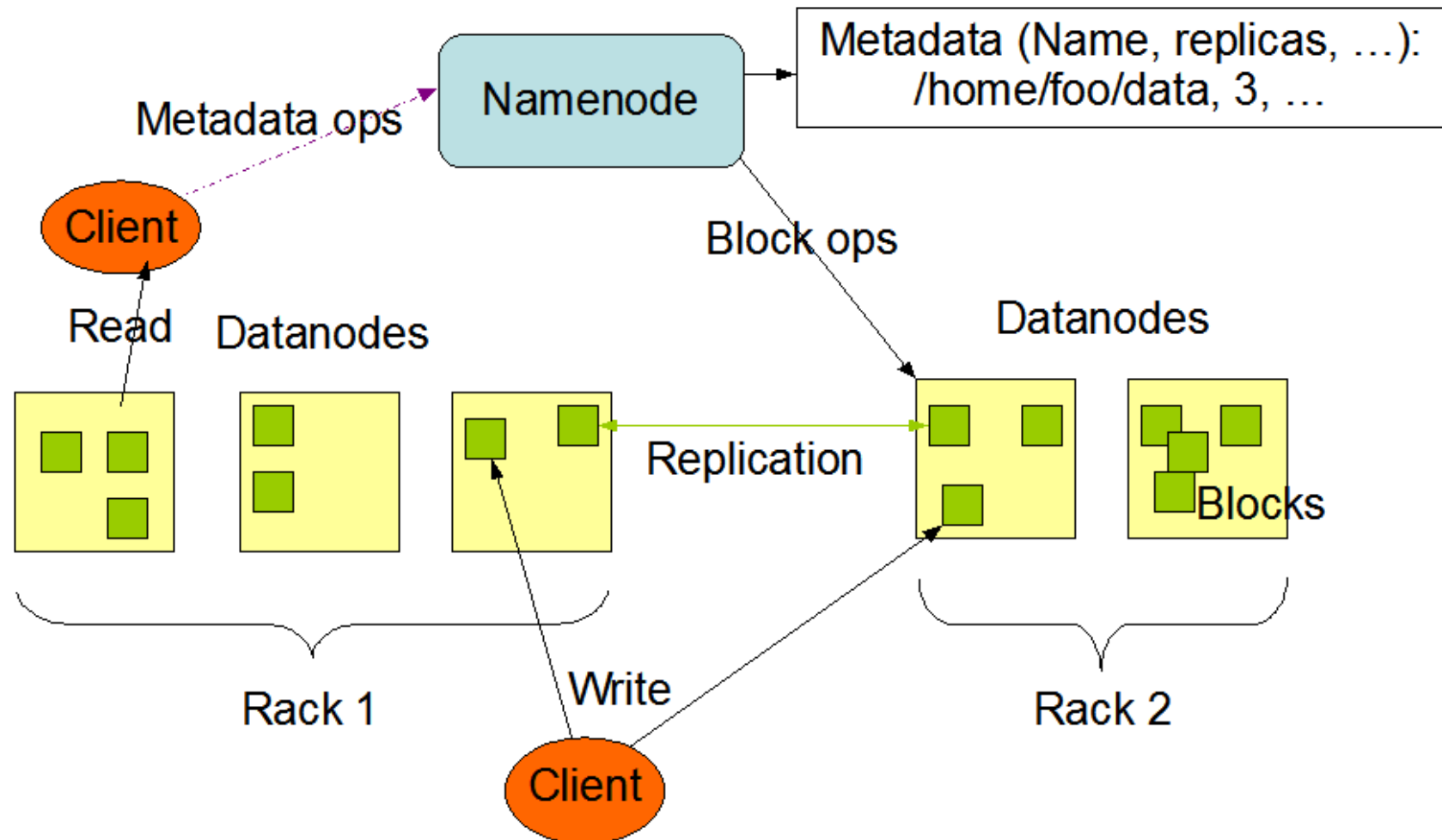
Block Replication

Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...

Datanodes



HDFS Usage



MR Job input and output data

1. Input data -> Mappers

- Mappers are assigned input splits from HDFS input path
 - (default 64MB)
- Data locality optimization: ApplicationManager attempts to assign Mappers where data block is stored

2. Reducers -> Output data

- Reducer output copied to HDFS
 - One file per Reducer
- For reliability concerns, HDFS replication

The cost of communications

- Parallelizing Map and Reduce jobs allow algorithms to scale close to linearly
- One potential bottleneck for MapReduce programs is the cost of Shuffle and Sort operations
 - Data has to be copied over network communications
 - All the keys emitted by the mappers
 - Sorting large amounts of elements can be costly
- Combiner is an additional optional step that is executed before these steps

Contents

- Parallel Computing
- The Map/Reduce Programming Model
- Apache Hadoop
- **Map/Reduce Programming**
- Dataflow languages

Writables (Hadoop Java Data types)

- Hadoop uses its own hierarchy of datatypes
 - They implement Writable interface
- Equivalent to Java types, designed for **serialization**
- Should also implement **Comparable**
 - Enabling key sort operations
- Most Hadoop types are wrappers to Java types
 - get, set methods for accessing its value.

Default Writable Types

Class	Description
BooleanWritable	Wrapper for a Boolean variable
ByteWritable	Wrapper for a single byte
DoubleWritable	Wrapper for a Double
FloatWritable	Wrapper for a Float
IntWritable	Wrapper for a Integer
LongWritable	Wrapper for a Long
Text	Wrapper to store text using the UTF8 format
NullWritable	Placeholder when the key or value is not needed

Sample custom Writable

```
public class Edge implements WritableComparable<Edge>{
    private String departureNode;
    private String arrivalNode;
    // getters and setters
    public void readFields(DataInput in) throws IOException{
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }
    public void write(DataOutput out) throws IOException {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }
    public int compareTo(Edge o) {
        return (departureNode.compareTo(o.departureNode) != 0) ?
            departureNode.compareTo(o.departureNode) :
            arrivalNode.compareTo(o.arrivalNode);
    }
}
```

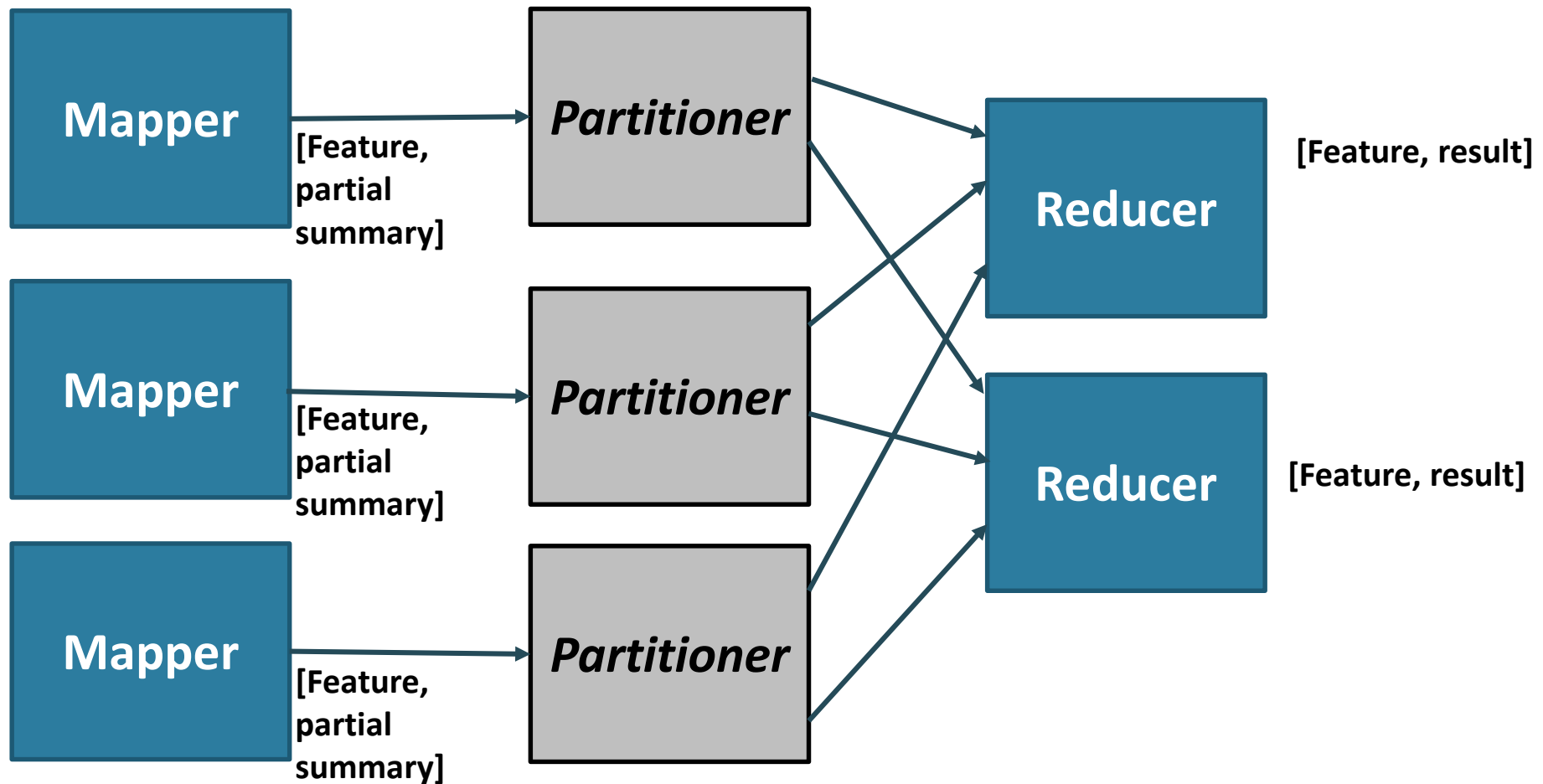
MapReduce job main class

- The Job class assembles a complete configuration for a map/reduce job
 - setMapperClass
 - setReducerClass
 - setCombinerClass
 - setMapOutputKeyClass, setMapOutputValueClass
 - FileInputFormat, FileOutputFormat
 - setNumReduceTasks

Numerical Summarization

- Goal: Calculate aggregate statistical values over a dataset
- Extract features from the dataset elements, compute the same function for each feature
- Examples: count, maximum/ minimum values, average/median/std deviation

Numerical Summarization Structure



Writing Map and Reduce functions

- Mapper
 - Find features in Input
 - Set partial aggregate value for the features in that iteration
- Reducer
 - Compute final aggregate result from all the intermediate values

Computing averages

Input: Row with student information, grade

Goal: Compute module average

ec03847293847	100
ec29347298347	100
ec23894283472	100
ec23489209348	100
ec23492834343	100
ec34948758493	0
ec56456456545	100
ec73453435434	100

Mapper

Mapper

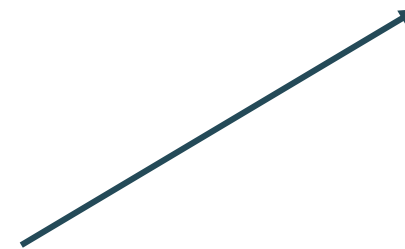
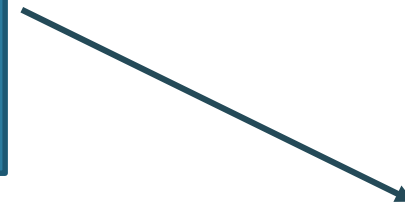
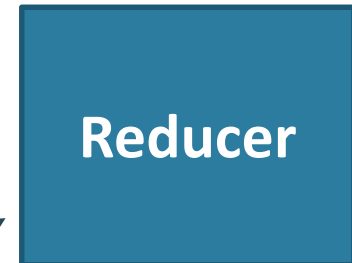
Reducer

Computing averages

Input: Row with student information, grade

Goal: Compute module average

ec03847293847	100
ec29347298347	100
ec23894283472	100
ec23489209348	100
ec23492834343	100
ec34948758493	0
ec56456456545	100
ec73453435434	100



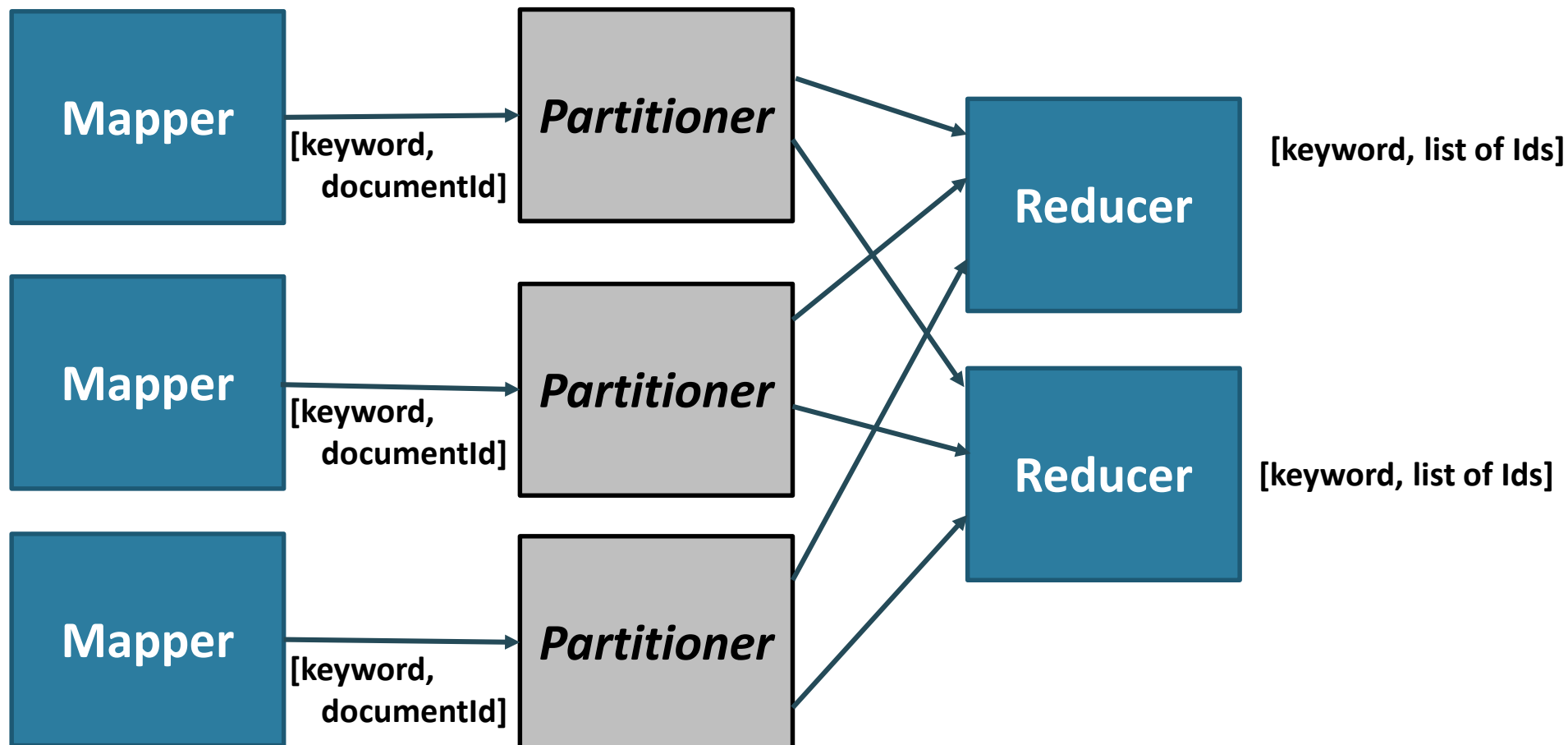
Combining Averages

- Average is NOT an associative operation
 - Cannot be executed partially with the Combiners
- Solution: Change Mapper results
 - Emit aggregated quantities, and number of elements
 - Mapper. For input entries (100,100,20),
 - Emit (100,1),(100,1), (20,1)
 - Combiner: adds aggregates and number of elements
 - Emits (220,3)
 - Reducer
 - Adds aggregates and computes average

Inverted index

- Goal: Generate index from a dataset to allow faster searches for specific features
- Examples: building index from a textbook. Finding all websites that match a search term

Inverted Index Structure



Writing Map and Reduce functions

- Mapper
 - Find features in Input
 - Emit [feature, document identifier]
- Reducer
 - Identity function (emits the list of results provided in shuffle and sort)

Contents

- Parallel Computing
- The Map/Reduce Programming Model
- Apache Hadoop
- Map/Reduce Programming
- **Dataflow languages**

Dataflow languages

- Map/Reduce can express many computations...
- ... but it exposes a relatively low-level programming interface
- Very Rigid: 1 Map and 1 Reduce
 - Possible to chain jobs, but done programmatically
- Approach: Define higher-level languages that can be automatically translated into multiple Map/Reduce jobs
 - Hadoop Pig / Hive

Hive and Pig

- Hive: data warehousing application in Hadoop
 - Query language is HQL, variant of SQL
 - Tables stored on HDFS as flat files
 - Developed by Facebook, now open source
- Pig: large-scale data processing system
 - Scripts are written in Pig Latin, a dataflow language
 - Developed by Yahoo!, now open source
 - Roughly 1/3 of all Yahoo! internal jobs
- Common idea:
 - Provide higher-level language to facilitate large-data processing
 - Higher-level language “compiles down” to Hadoop jobs



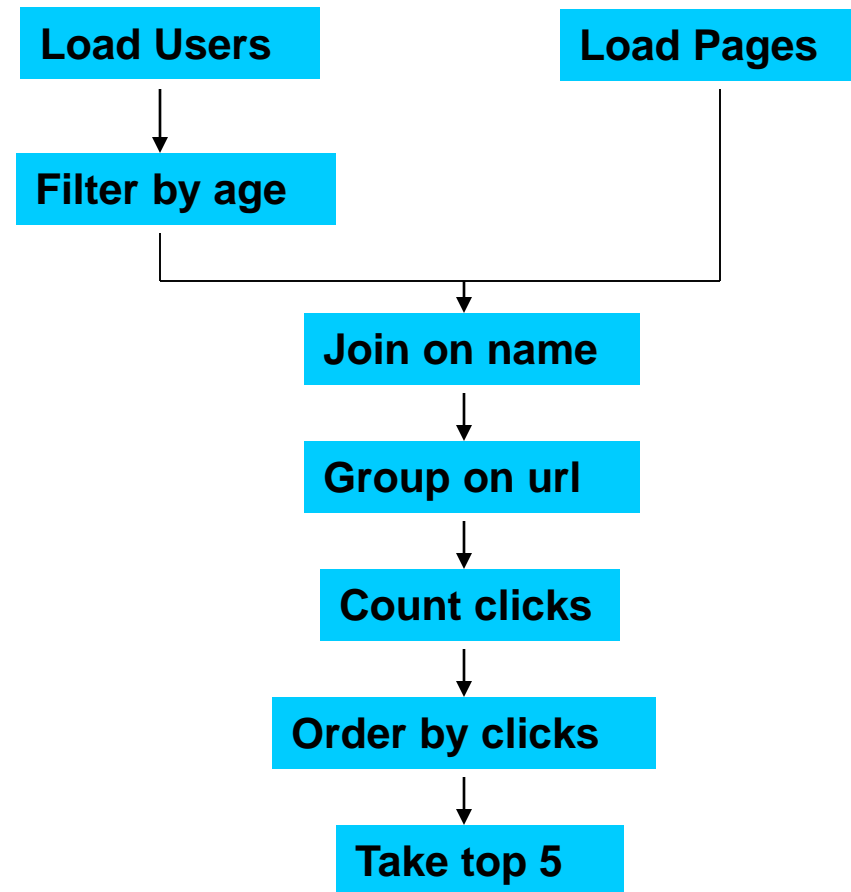
Pig

- Started at Yahoo! Research
- Now runs about 30% of Yahoo!'s jobs
- Features:
 - Expresses sequences of MapReduce jobs
 - Data model: nested “bags” of items
 - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
 - Easy to plug in User-Defined Java functions



An Example Problem

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited pages by users aged 18 - 25.



In MapReduce

```

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.mapred.FileInputFormat;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapperCombiner;
import org.apache.hadoop.mapred.MapperReducer;
import org.apache.hadoop.mapred.Partitioner;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.ReducerCombiner;
import org.apache.hadoop.mapred.ReducerGrouping;
import org.apache.hadoop.mapred.ReducerRunner;
import org.apache.hadoop.mapred.ReducerWrapper;
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;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.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(" " + 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(0, 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
            // 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 Reducer<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.setOutputFormat(SequenceFileOutputFormat.class);
    lp.setMapperClass(LoadPages.class);
    lp.setCombinerClass(LoadPages.class);
    lp.setReducerClass(LoadPages.class);
    lp.setNumReduceTasks(1);
    Job loadPages = new Job(lp);

    JobConf ifu = new JobConf(MRExample.class);
    ifu.setJobName("Load and Filter Users");
    ifu.setInputFormat(SequenceFileInputFormat.class);
    ifu.setOutputFormat(SequenceFileOutputFormat.class);
    ifu.setMapperClass(LoadAndFilterUsers.class);
    ifu.setCombinerClass(LoadAndFilterUsers.class);
    ifu.setReducerClass(LoadAndFilterUsers.class);
    ifu.setNumReduceTasks(1);
    Job loadUsers = new Job(ifu);

    JobConf join = new JobConf(MRExample.class);
    join.setJobName("Join Users and Pages");
    join.setInputFormat(KeyValueTextInputFormat.class);
    join.setOutputFormat(SequenceFileOutputFormat.class);
    join.setMapperClass(LoadJoined.class);
    join.setCombinerClass(LoadJoined.class);
    join.setReducerClass(ReduceUrls.class);
    join.setNumReduceTasks(1);
    Job joinJob = new Job(join);

    JobConf group = new JobConf(MRExample.class);
    group.setJobName("Group URLs");
    group.setInputFormat(KeyValueTextInputFormat.class);
    group.setOutputFormat(SequenceFileOutputFormat.class);
    group.setMapperClass(LoadAndFilterUsers.class);
    group.setCombinerClass(ReduceUrls.class);
    group.setReducerClass(ReduceUrls.class);
    group.setNumReduceTasks(1);
    Job groupJob = new Job(group);

    JobConf top100 = new JobConf(MRExample.class);
    top100.setJobName("Top 100 sites");
    top100.setInputFormat(SequenceFileInputFormat.class);
    top100.setOutputFormat(SequenceFileOutputFormat.class);
    top100.setMapperClass(LoadClicks.class);
    top100.setCombinerClass(LimitClicks.class);
    top100.setReducerClass(LimitClicks.class);
    top100.setNumReduceTasks(1);
    Job top100Job = new Job(top100);

    JobConf jc = new JobConf(MRExample.class);
    jc.setJobName("Find top 100 sites for users");
    jc.setInputFormat(SequenceFileInputFormat.class);
    jc.setOutputFormat(SequenceFileOutputFormat.class);
    jc.setMapperClass(LoadAndFilterUsers.class);
    jc.setCombinerClass(LoadAndFilterUsers.class);
    jc.setReducerClass(LoadAndFilterUsers.class);
    jc.setNumReduceTasks(1);
    Job joinJob2 = new Job(jc);
}

```

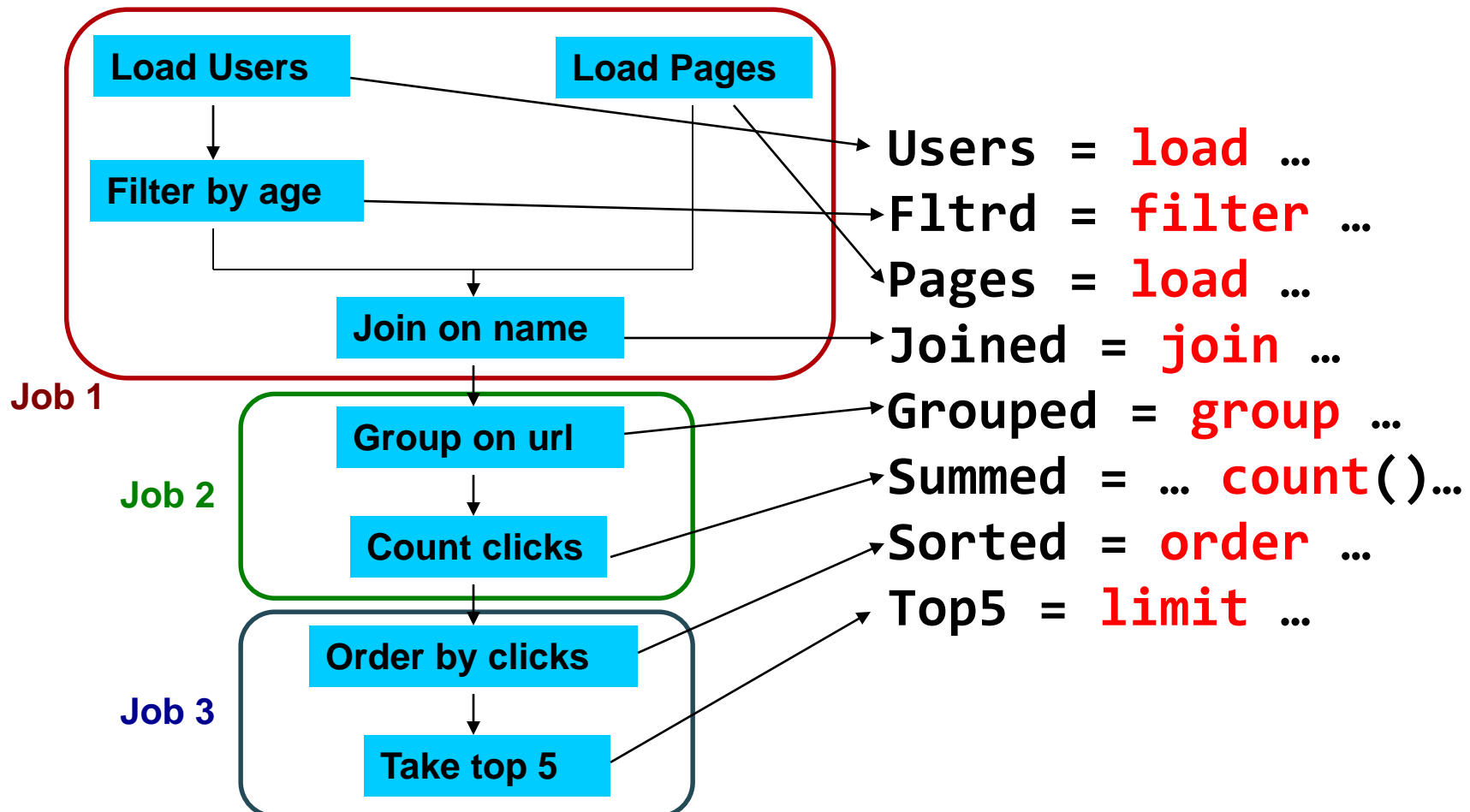
In Pig Latin

```
Users      = load 'users' as (name, age);
Filtered   = filter Users by
              age >= 18 and age <= 25;
Pages      = load 'pages' as (user, url);
Joined     = join Filtered by name, Pages by user;
Grouped    = group Joined by url;
Summed     = foreach Grouped generate group,
              count(Joined) as clicks;
Sorted    = order Summed by clicks desc;
Top5      = limit Sorted 5;

store Top5 into 'top5sites';
```

Ease of Translation

Notice how naturally the components of the job translate into Pig Latin.



Critique of dataflow languages

- They speedup development of **some** analysis tasks
- ...but they do not help with performance
- All the limitations of MapReduce are still present
- Plus potentially inefficiencies from the automated translations (maybe more jobs than required, although implementations are reasonable.
 - According to PigMix2, 16% less efficient than well written MR code
 - <https://cwiki.apache.org/PIG/pigmix.html>