ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΕΛΟΠΟΝΝΗΣΟΥ

Big Data Management

MapReduce

University of the Peloponnese
Dept. of Informatics and Telecommunications

Outline

- MapReduce basics
  - introduction
  - the MapReduce paradigm
- Anatomy of a MapReduce job
  - execution stages
  - job submission
  - tasks
- MapReduce environment
- Examples using MapReduce

Motivation

- Need to process petabytes of data ...
- ... on your multi-thousand commodity PC cluster ...
- ... and make it easy!

- Idea:
  - bring computation close to the data
  - store files multiple times for reliability

⇒ Enter Map/Reduce

What is MapReduce?

An elegant way to work with big data, aka

1. The programming model
   Functional-like programming → Map/reduce functions

2. The execution framework
   Input splitting, intermediate result sort/shuffling, task scheduling, ...

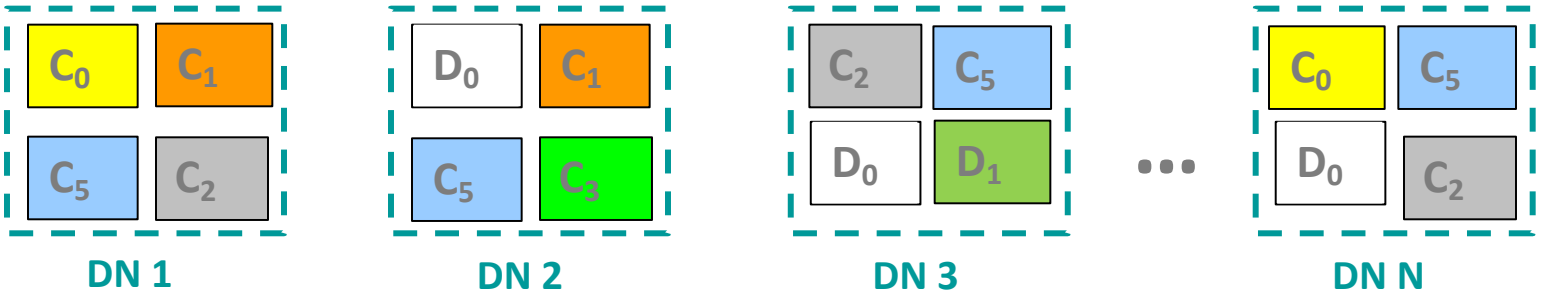
3. The software implementation
   Google MapReduce, Hadoop, Storm/Kafka, AcB/Cell, ...

Remember…

- Data chunks
  - file is split into blocks
  - typically each block is 64 or 128MB
  - each block is replicated (usually 3x)
  - keep replicas in different racks
- Name node
  - stores metadata about where files are stored
  - might be replicated
- HDFS client
  - talks to NN to find data blocks
  - connects directly to DN to access data

Distributed file system

- Reliable distributed file system
- Data kept in chunks spread across machines
- Each chunk is replicated on different machines
  - seamless recovery from disk or machine failure



Bring computation directly to the data!

Data nodes also serve as compute nodes

MapReduce premises

- Scale *out*, not *up*
- Design for fault tolerance (assume failures are the norm)
- Move processing to the data
- Optimise for high throughput batch jobs
  - i.e., sequential reads, no random accesses
- Hide system-related stuff from the app developers
  - ⇒ provide automated distribution, parallelisation, scheduling, . . .
- Provide mechanisms for progress report and monitoring
- ‘‘Always’’ operates on key-value pairs

Warm up task

- *"Million lyrics dataset"*

> Hear the rime of the Ancient Mariner, see his eye as he stops one of three. Mesmerises one of the wedding guests. Stay here and listen to the nightmares of the Sea.
>
> And the music plays on, as the bride passes by. Caught by his spell and the Mariner tells his tale.
>
> Driven south to the land of the snow and ice, to a place where nobody's been. Through the snow fog flies on the albatross. Hailed in God's name, hoping good luck it brings.
>
> And the ship sails on, back to the North. Through the fog and ice and the albatross follows on.
>
> The mariner kills the bird of good omen. His shipmates cry against what he's done, but when the fog clears, they justify him and make themselves a part of the crime. [...]

Q: Compute all unique words in the stream and their number of appearances (*Word Count*)

Task: Word count cont'd

Pfft... We can do that with simple shell scripts...

- Using shell programming, take #1:

```
cat input.txt | tr -d '.,;' | tr '[:upper:]' '[:lower:]' | \
    tr '[:space:]' '\n' | sort | uniq -c | awk '{print $2" "$1}'
```

Now what? → Must sort entire input and go over it twice!

Let's combine these two and go all-out distributed!

- Using shell programming, take #2:

```
cat input.txt | tr -d '.,;' | tr '[:upper:]' '[:lower:]' |
  awk '\
    {for (i = 1; i <= NF; i++) freqs[$i]++;} END \
    {for (word in freqs) print word" "freqs[word];}' | \ sort
```

What's wrong with this?

→ Out of RAM as freqs grows huge! Plus serial execution...

Task: Word Count

- Case 1:
  - file too large for memory, but all <word, count> pairs fit in memory
- Case 2:
  - count and store the occurrences of words
  - this can be naturally parallelisable
  - case 2 captures the essence of MapReduce

MapReduce: Overview

- Sequentially read a lot of data

- Map:
  - extract something you care about
- Group by key: Sort and Shuffle
- Reduce:
  - aggregate, summarise, filter or transform
- Write the result

Outline stays the same,
Map and Reduce change to fit the problem

More Specifically

- Input: a set of key-value pairs

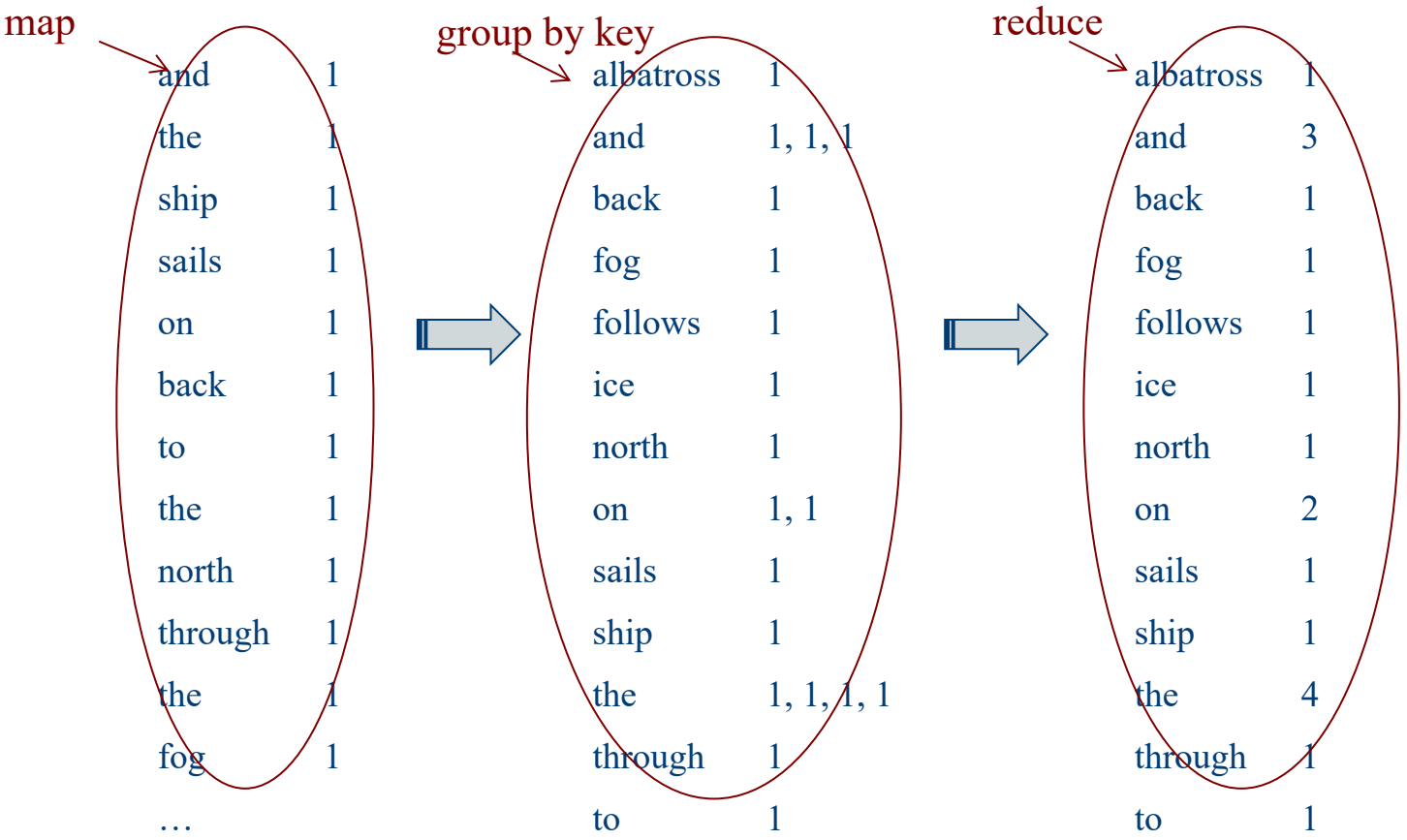
- Programmer specifies two methods:

  - **Map(k_1, v_1)** $\rightarrow$ [(k_2, v_2)]
    - Takes a key-value pair and outputs a set of key-value pairs
      - e.g., key is the filename, value is a single line in the file
    - There is one Map call for every (k_1,v_1) pair

  - **Reduce(k_2, [v_2])** $\rightarrow$ [(k_3, v_3)]
    - All values v_2 with same key k_2 are reduced together
    - There is one Reduce function call per unique key k_2

Word Count Using MapReduce

''And the ship sails on, back to the North, through the fog and ice and the albatross follows on.''



Word Count Using MapReduce cont'd

map arguments (key, value)

```
void Map(int lineNo, String line) {
        foreach (Word w in line)
                emit(tolower(w), 1);
}
```

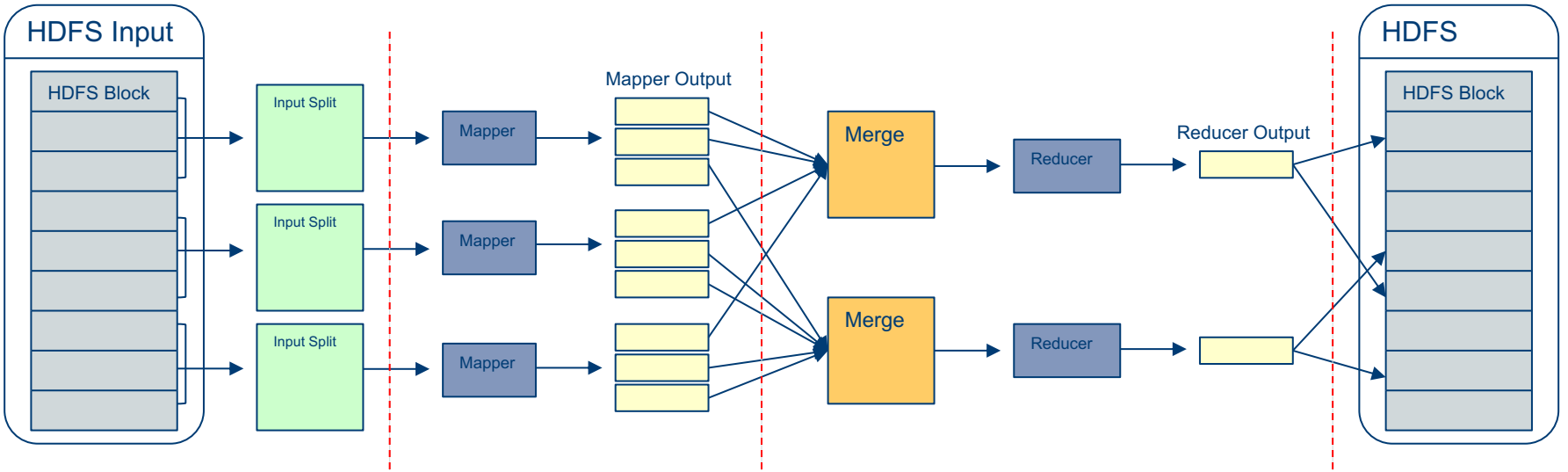
output

reduce arguments (key, value)

```
void Reduce(Word w, int[] counts) {
        int sum = 0;
        foreach (int i in counts)
                sum += i;
        emit(w, sum);
```

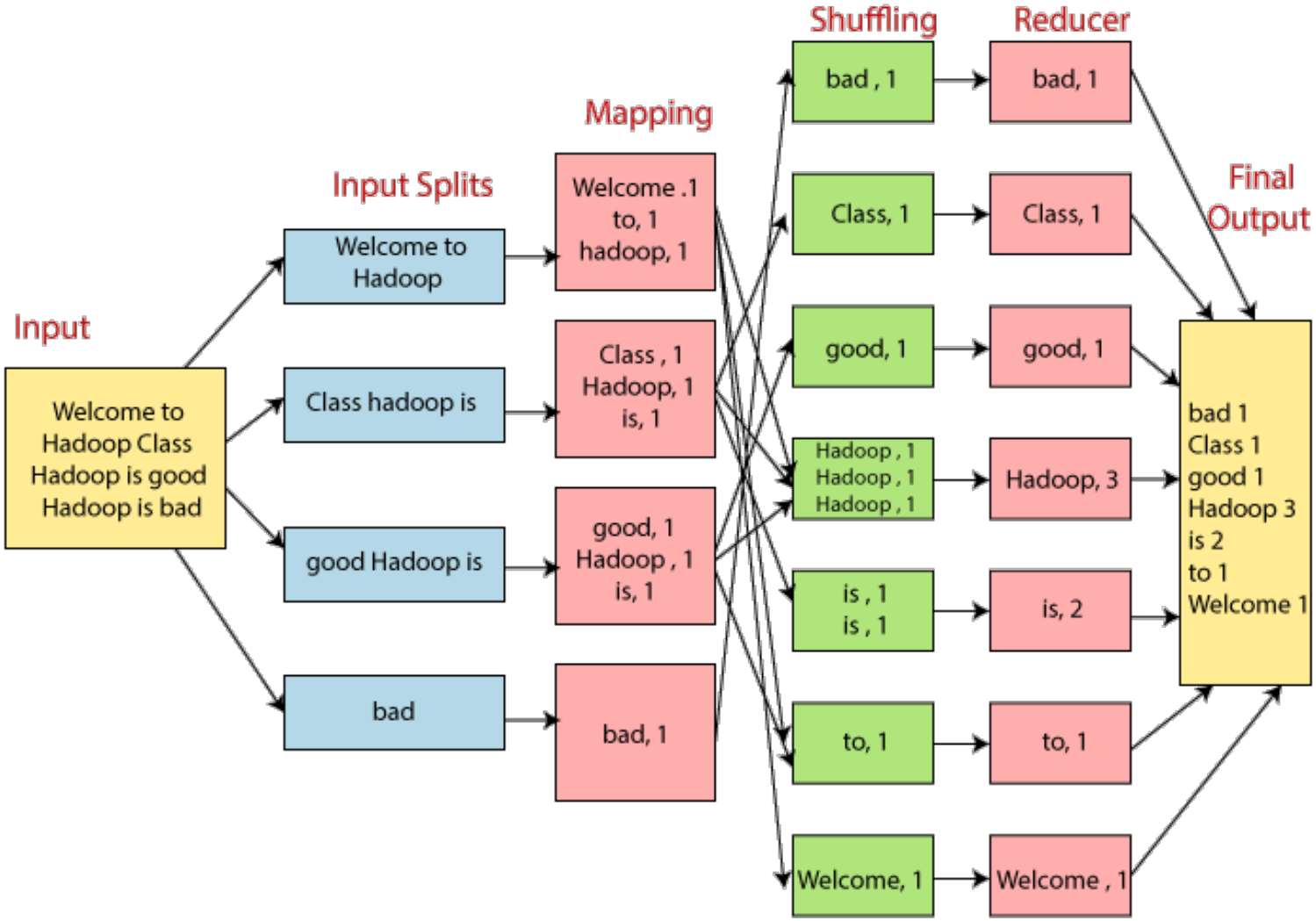
output

MapReduce processing steps

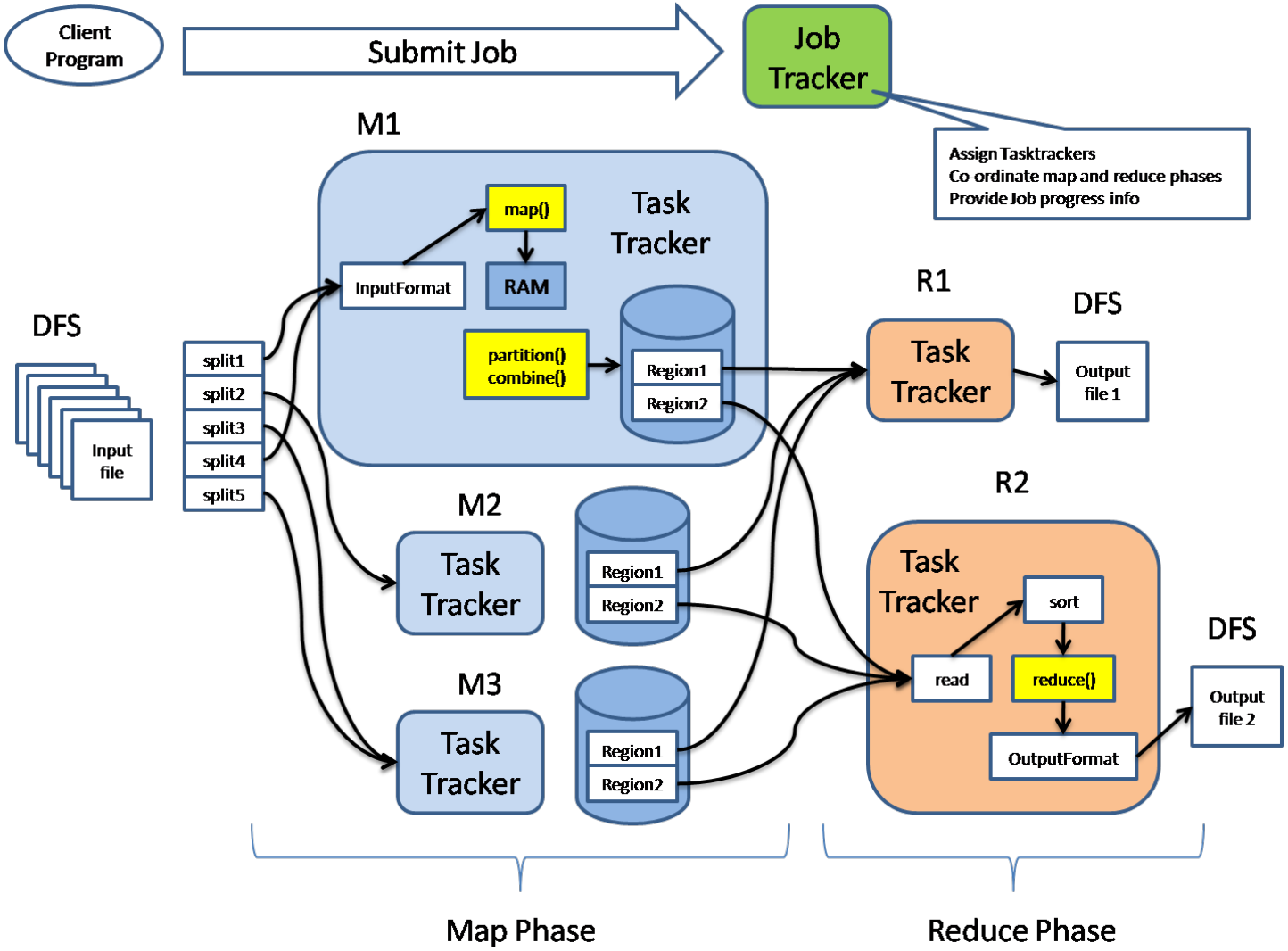


- Input parsing & splitting → *InputFormat*
- Map function execution → *Mapper*, *Combiner*
- Intermediate result partitioning, sorting, and storing → *Partitioner, Merge*
- Intermediate data transfer to reducers → *Shuffle*
- Reduce function execution → *Reducer*
- Final output storage → *OutputFormat*

Hadoop MapReduce phase diagram



Hadoop MapReduce architecture



Map it!

- Input files: elements (e.g., tuple or a document or a document line)
- Chunk: collection of elements (no element is stored across two chunks)
- Map function:
  - written by the user
  - input element as its argument
  - produces zero or more key-value pairs

- Map task can produce several key-value pairs with the same key, even from the same element
- Example:

  Input: a repository of documents

  Map task: reads a line and breaks it into its sequence of words

  $w_1, w_2, \ldots, w_n$

  emits a sequence of key-value pairs where the value is always 1,

  i.e., $(w_1, 1), (w_2, 1), \ldots, (w_n, 1)$

Grouping by key

- The key-value pairs are grouped by key
- The values associated with each key are formed into a list of values
- Job tracker
  - merges the files from each Map task
  - feeds the merged file to the grouping process as a sequence of key-list-of-value pairs

Reduce it!

- Input data: a set of (key, [value]) pairs produced by the merging/grouping process
- Reduce function:
  - written by the user
  - has as its argument a pair consisting of a key and its list of associated values (k, [v_1, v_2, . . . , v_n])
  - produces zero or more key-value pairs

- Example:

  Input: a sequence of word-value pairs

  Reduce task: add up all the occurrences of a word

  emits the word and the sum (per reducer)

  emits a sequence of (w,m) pairs, where w is a word that appears at least once

MapReduce environment

MapReduce environment takes care of:

- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the group by key step
- Handling machine failures
- Managing required inter-machine communication

Data flow

- Input and final output are stored on a distributed file system (FS)
  - 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
- Output is often input to another MapReduce task

Job tracker

- Job tracker takes care of coordination:
  - task status (idle, in-progress, completed)
  - the identity of the worker machine (for non-idle tasks)
  - the locations for each completed map task
  - idle tasks get scheduled as workers become available
  - when a map task completes, it sends the job tracker the location and sizes of its *R* intermediate files, one for each reducer
  - job tracker pushes this info to reducers
- Job tracker pings workers periodically to detect failures

Dealing with failures

- Map worker failure
  - map tasks completed or in-progress at worker are reset to idle
    - even completed map tasks are re-executed because their output is stored on the local disk(s) of the failed machine
  - reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - only in-progress tasks are reset to idle
  - reduce task is restarted
- Job tracker failure
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

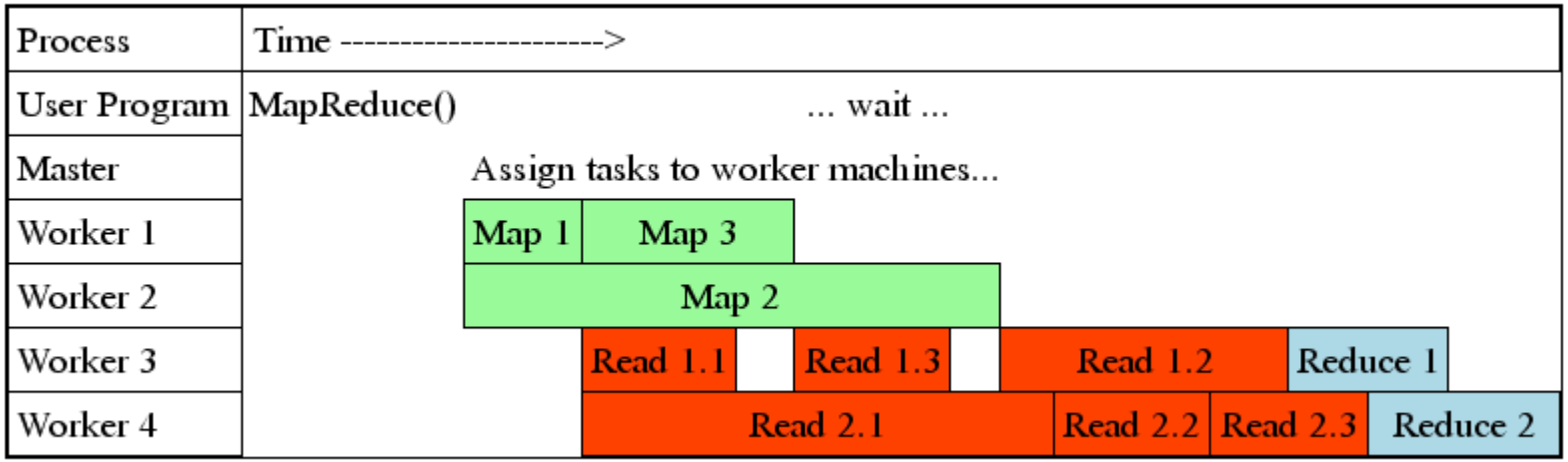
- *M* map tasks, *R* reduce tasks

- Rule of a thumb:
  - make *M* much larger than the number of nodes in the cluster
  - one chunk per map is common
  - improves dynamic load balancing and speeds up recovery from worker failures

- Usually *R* is smaller than *M*
  - because output is spread across *R* files

Task granularity & pipelining

- Fine granularity tasks: many more map tasks than machines
  - minimises time for fault recovery
  - can do pipeline shuffling with map execution
  - better dynamic load balancing
- For example, use 200.000 map and 5.000 reduce tasks with 2.000 machines



Refinements: Backup tasks

- Problem
  - slow workers significantly lengthen the job completion time, because of
    - other jobs on the machine,
    - bad disks,
    - weird things
- Solution
  - near end of phase, spawn backup copies of tasks
    - whichever one finishes first “wins”
- Effect
  - dramatically shortens job completion time

Refinement: Combiners

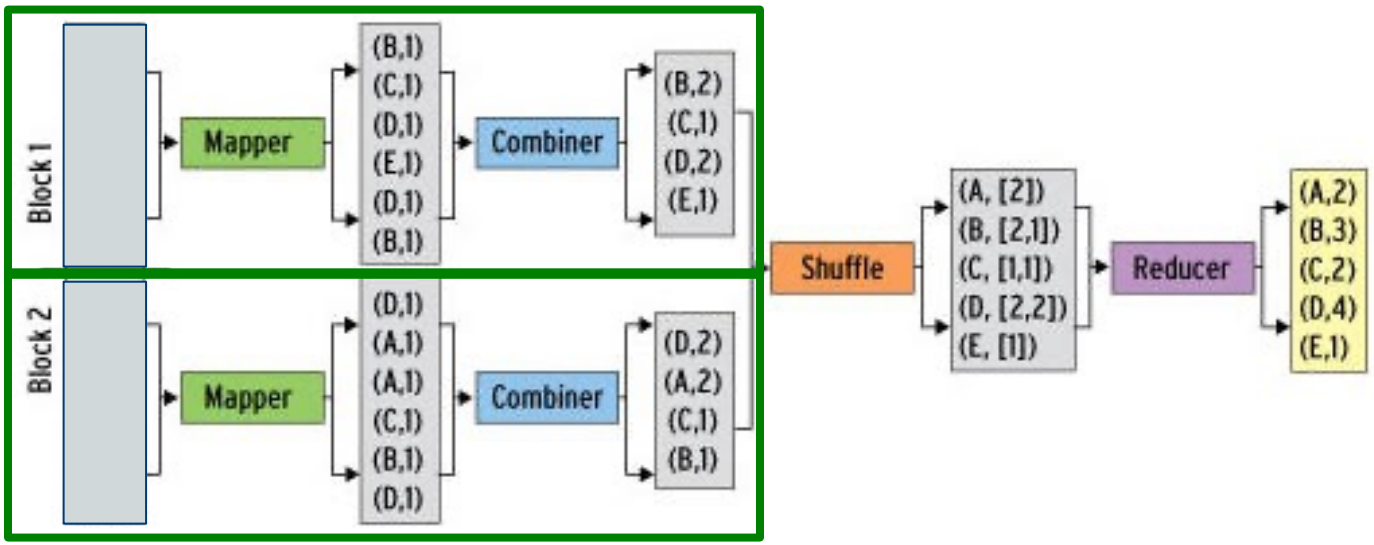
- Often a Map task will produce many pairs of the form (k,v_1), (k,v_2), ... for the same key k
  - e.g., popular words in the word count example

- Can save network time by pre-aggregating values in the mapper:
  - `combine(k, list(`v_1`))` → `(k,` v_2`)`
  - combiner is usually same as the reduce function

- Works only if reduce function is commutative and associative

Refinement: Combiners cont'd

- Back to our word counting example…
  - combiner combines the values of all keys of a single mapper (i.e., single machine)



  - much less data needs to be copied and shuffled!

Refinement: Partition function

- Wants to control how keys get partitioned
  - inputs to map tasks are created by contiguous splits of input file
  - reduce needs to ensure that records with the same intermediate key end up at the same worker

- System uses a default partition function:
  - `hash(key) mod R`

- Sometimes it is useful to override the hash function:
  - e.g., `hash(hostname(URL)) mod R`
  - ensures URLs from a host end up in the same output file

Algorithm design

- Remember!
  - mappers run in isolation
  - you have no idea in what order the mappers run
  - you have no idea on what node the mappers run
  - you have no idea when each mapper finishes

- Consider using tools for synchronisation
  - ability to hold state in reducer across multiple key-value pairs
  - sorting function for keys
  - partitioner
  - cleverly-constructed data structures

MapReduce patterns

- Summarization
  - numerical summarization
  - inverted index
  - counting/summing
- Filtering
- Data organization (sort, merge, etc.)
- Relational-based (join, selection, projection, etc.)
- Iterative messaging passing (graph processing)
- Others
  - simulation of distributed systems
  - cross-correlation
  - …

Counting and summing

- Problem statement
  - there is a number of documents
  - each document is a set of terms
  - calculate total number of occurrences of each term in all documents
  - alternatively, it can be an arbitrary function of the terms
    - e.g., a log file where each record contains a response time and it is required to calculate an average response time

- Applications:
  - log analysis
  - data querying
  - …

Counting and summing: solution

- Map it!

```
class Mapper
    method Map(docid id, doc d)
        for all term t in doc d do
            emit(term t, count 1)
```

- Reduce it!

```
class Reducer
    method Reduce(term t, counts [c1, c2, …])
        sum=0
        for all count c in [c1, c2, …]
            sum = sum + c
        emit(term t, count sum)
```

```
class Combiner
    method Combine (term t, [c1, c2, …])
        sum=0
        for all count c in [c1, c2, …]
            sum = sum + c
        emit(term t, count sum)
```

Collating

- Problem statement
  - there is a set of items and some function of one item
  - it is required to save all items that have the same value of function into one file or perform some other computation that requires all such items to be processed as a group

- Applications
  - inverted indexes
  - ETL (extract-transform-load) process
  - …

Collating: solution

- Map it!
  - mapper computes a given function and emits value of the function as a key and item itself as a value

```
class Mapper
 method Map(docid id, doc d)
        for all term t in doc d do
               emit(f(t), term t)
```

- Reduce it!
  - reducer obtains all items grouped by function value and processes or saves them

```
class Reducer
 method Reduce(f(t), terms [t1, t2, …])
        //process or save
        emit(term t, count sum)
```

Cross-correlation

- Problem statement
  - there is a set of tuples
  - for each possible pair of items calculate a number of tuples where these items co-occur

- Applications
  - text analysis (items are words, tuples are sentences)
  - market analysis (customers who buy *this* tend to also buy *that*)
  - log analysis (users that visit *this* page tend to also visit *that* page)

Cross-correlation: solution

- Map it!

```
class Mapper
    method Map(null, basket [i1, i2, …])
        for all item i in basket
            for all item j in basket
                emit(pair (i, j), count 1)
```

- Reduce it!

```
Class Reducer
    method Reduce(pair (i, j), counts [c1, c2, …])
        s = sum([c1, c2, …])
        emit(pair (i, j), count s)
```

Selection

- Problem statement
  - there is a set of tuples
  - select those tuples that satisfy a given property

- Map it!

```
class Mapper
    method Map(rowkey key, tuple t)
        if tuple t satisfies the predicate
            emit(tuple t, null)
```

- Reduce it!

```
class Reducer
    method Reduce(tuple t, null)
        emit(tuple t, null)
```

Projection

- Problem statement
  - there is a set of tuples
  - extract the required fields from each tuple

- Map it!

```
class Mapper
    method Map(rowkey key, tuple t)
            //extract required fields to tuple g
            tuple g = project(t)
            emit(tuple g, null)
```

Join

- Problem statement
  - compute the join $R(A,B) \bowtie S(B,C)$
  - R and S are each stored in files
  - tuples are pairs (a,b) or (b,c)

R

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

$\bowtie$ S

B	C
b_2	c_1
b_2	c_2
b_3	c_3

=

A	B	C
a_3	b_2	c_1
a_3	b_2	c_2
a_4	b_3	c_3

- Pseudocode sketch…
  - turn each input tuple $R(a,b)$ into key-value pair $(b,(R.a))$
  - turn each input tuple $S(b,c)$ into $(b,(S.c))$
  - after, the MapReduce system groups together the intermediate data by the intermediate key, i.e. the b values
  - match all the pairs $(b,(R.a))$ with all the pairs $(b,(S.c))$ and output (a,b,c)

Join: solution

- Map it!

```
class Mapper
    method Map(join_key k, tuple [k, v1, v2, … ])
        emit(join_key k, tagged_tuple[set_name tag,
        values[v1, v2, …])
```

- Reduce it!

```
class Reducer
    method Reduce(join_key k, tagged_tuples[])
        for each r in tagged_tuples
            for each s in tagged_tuples
                if r.tag = 'R' and s.tag = 'S'
                    emit(null, tuple [k, r, s])
```

Tweet calculations

- Given a list of tweets (username, date, text) determine first and last time and the number of times a user commented
- Map it!

```
class Mapper
 method Map(userid uid, tuple t)
        emit(userid uid, tuple t)
```

- Reduce it!

```
class Reducer
 method Reduce(uid, tuples [t1, t2, …])
        for each t in tuples
          if t.date < minDate result.minDate = t.date
          if t.date > maxDate result.maxDate = t.date
          result.sum = result.sum + 1
        emit(uid, result)
```

Example: Language model

- Statistical machine translation
  - need to count number of times every 5-word sequence occurs in a large corpus of documents
- Very easy with MapReduce
  - Map it!
    - Extract (5-word sequence, count) from document
  - Reduce it!
    - Combine the counts

Example: Host size

- Suppose we have a large web corpus
- Look at the metadata file
  - lines of the form: (URL, size, date, …)
- For each host, find the total number of bytes
  - that is, the sum of the page sizes for all URLs from that particular host

- Other examples:
  - link analysis and graph processing
  - Machine Learning algorithms

Word count: java code

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

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));
    }
```

Min/max calculation: java code

- Given a list of tweets (username, date, text) determine first and last time and the number of times a user commented

```
public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException, ParseException {
        Map<String, String> parsed = MRDPUtils.parse(value.
            toString());
        String strDate = parsed.get(MRDPUtils.CREATION_DATE);
        String userId = parsed.get(MRDPUtils.USER_ID);
        if (strDate == null || userId == null) {
          return;
        }
        Date creationDate = MRDPUtils.frmt.parse(strDate);
        outTuple.setMin(creationDate);
        outTuple.setMax(creationDate);
        outTuple.setCount(1);
        outUserId.set(userId);
        context.write(outUserId, outTuple);
}
```

Min/max calculation: java code cont'd

```
public void reduce(Text key, Iterable<MinMaxCountTuple> values,
        Context context) throws IOException, InterruptedException {
        result.setMin(null);
        result.setMax(null);
        int sum = 0;
        for (MinMaxCountTuple val : values) {
                if (result.getMin() == null
                        || val.getMin().compareTo(result.getMin()) < 0)
                                {
                        result.setMin(val.getMin());
                }
                if (result.getMax() == null
                        || val.getMax().compareTo(result.getMax()) > 0)
                                {
                        result.setMax(val.getMax());
                        }
                        sum += val.getCount();}
        result.setCount(sum);
        context.write(key, result);
}
```

References/Thanks

- Slide resources
  - "Big data: Systems, programming, management" course @ School of Computing Science, University of Glasgow
- Many thanks to
  - Peter Triantafillou
  - Nikos Ntarmos