COMP38120 Services on the Web: Workshop 3

Norman Paton, Sandra Sampaio

MANCHESTER
1824
The University of Manchester

Workshop Outline

- Today's workshop seeks to give you a flavour of web scale data processing, and covers:
  - Cloud services and big data.
  - The Map Reduce approach to big data processing:
    - The standard WordCount Example (in more detail).
    - Design patterns.
    - Writing of map reduce programs.

The University of Manchester

Position in Workshop

- Big Data and the Cloud.
- MapReduce concepts.
- MapReduce design patterns.
- How MapReduce surfaces in Hadoop.

For reference – we won't get to this in the workshop.

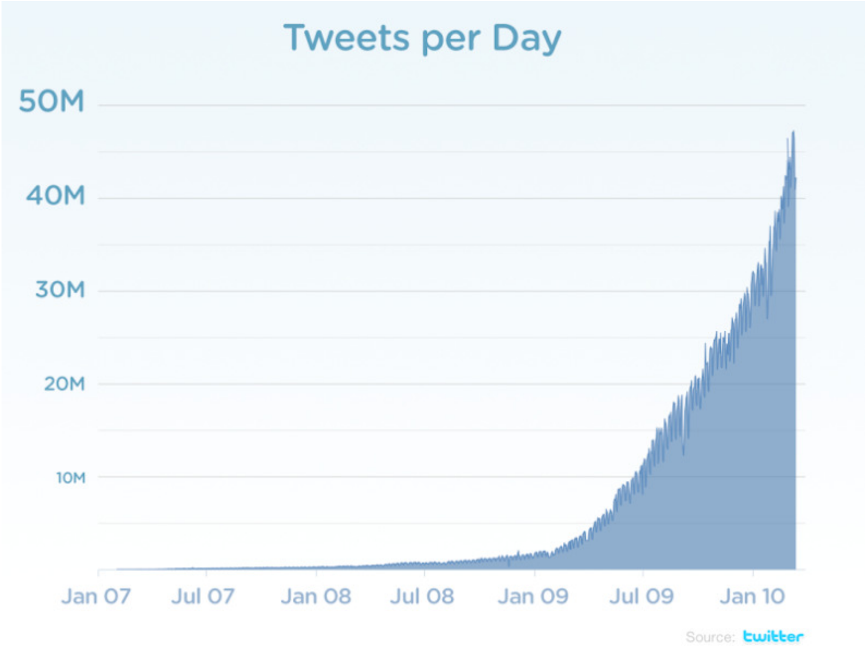
MANCHESTER 1824
The University of Manchester

Web-Scale Data-Intensive Applications

- An increasing number of applications are extremely data intensive; data intensive applications include:
  - web indexing: there are estimated to be over 45 billion pages (www.worldwidewebsize.com/).
  - social media: there are in the region of 500 million tweets per day.
  - instruments: CERN "sifts" around 30 petabytes a year; an Airbus A350 has 6000 sensors that produce 2.5Tb per day.
- So, the trends are that ***the ability to capture and store data is overwhelming the ability to process what is stored***.

Size not the only challenge

- Change and unpredictability are also challenging (https://blog.twitter.com/official/en_us/a/2010/measuring-tweets.htm).
- However, note that twitter's growth rate has dropped a lot:
  - 2009: 900%
  - 2013: 32%
  - Now: past the peak?



Cloud Services for Web-Scale Data-Intensive Applications

- Data-intensive processing is beyond the capability of any individual machine and requires clusters.
- Volatile demand means that it is not surprising that a significant number of companies are opting to "rent" cloud resources, rather than investing in the running of giant data centres, which not everyone has the expertise to manage.
- There is no single model of big data processing:
  – Long running tasks: Batch (e.g. map reduce), Streaming (e.g. Storm).
  – Interactive tasks: Lookup (e.g. NoSQL Databases), Search (household names).

Big Data and Parallelism

- It is not practical to address big data problems in serial; just scanning the data from disc takes much too long.
- Relevant disc trends:
  - Disc capacity has been following Moore's law:http://en.wikipedia.org/wiki/Moore%27s_law.
  - Disc speed has not kept up, especially for seeks (which need physical head movements).
- Thus it has become viable to store more and more data (on more and more discs), but accessing the data in a way that reflects these trends means scanning in parallel. Map Reduce emerged in this context.

The University of Manchester

Big Data and the Web

- The archetypal big data problem in the web involves search. The challenges include:
  - Crawling the web to obtain the raw material over which the indexes are constructed (highly distributed, offline).
  - Constructing the indexes from the crawled data so that the data can be searched efficiently (batch processing, in the data centre).
  - Searching the indexes (interactive response times, in the data centre).
- So, there is not a single processing model for big data, but we will focus on a popular one that applies to many applications, namely map reduce (batch processing, in the data centre).

Position in Workshop

- Cloud Services.
- MapReduce concepts.
- MapReduce design patterns.
- HowMap Reduce surfaces in Hadoop.

Map Reduce

- *Map Reduce* is a scalable programming model, originally developed by Google for tasks such as index building.
- In Map Reduce:
  - applications are developed using two simple, functional operations (*map* and *reduce*) … and a few other supporting players;
  - the infrastructure supports the running of Map Reduce applications in parallel on potentially huge data sets on potentially numerous commodity machines.
- *Hadoop* is a widely used open source implementation of map reduce (hadoop.apache.org).

Map Reduce Functions

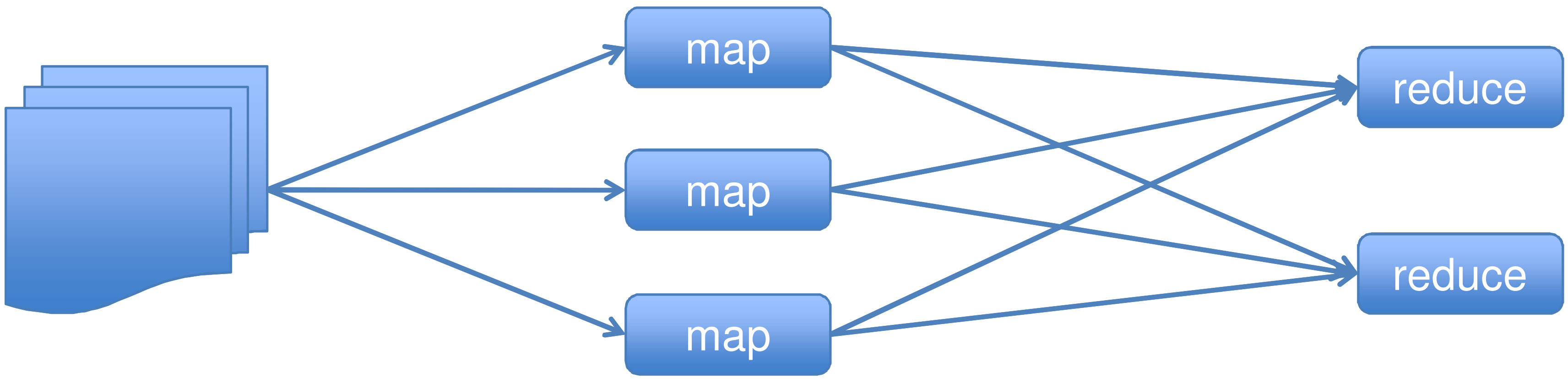
- The map and reduce functions are defined as:
  - map(key_1, $value_1$) -> [(key_2,$value_2$)]
  - reduce(key_2,list of values $value_2$) -> [(key_3, $value_3$)]
- where:
  - map, given a key key_1 and a value $value_1$, generates a collection of key-value pairs (key_2,$value_2$).
  - reduce, given a key key_2 output by map, and a collection of *all* the values $value_2$ associated with that key, returns a new collection of key-value pairs.

The University of Manchester

Word Count Example

- The standard map reduce example program counts the number of occurrences of each word in a document (although this is in some ways a toy task, it is relevant to web indexing).

Word Count at Runtime



the reduce
operation,
given the id
of a
document
and the
document,
emits a
collection of
key-value
pairs
…

the reduce
operation,
given the id
of

a document
and the
document,

…

(the, 1)
(reduce, 1)
(the, 1)
…

(the, 1)
(document, 1)
…

…

(the, [1,1,1])
(reduce,[1])
…

(of, [1,1])
(pairs,[1])
…

(the, 3)
(reduce, 1)
…

(of, 2)
(pairs, 1)
…

The University of Manchester

Word Count Map

- Recall the description of map:
  - map(key_1, $value_1$) -> [(key_2,$value_2$)]
  - map, given a key *key_1* and a value *$value_1$*, generates a collection of key-value pairs (*key_2,$value_2$*).
- In WordCount:
  - key_1 is the identifier of the document (not used).
  - $value_1$ is the document (or part of the document).
  - key_2 is a word from the document.
  - $value_2$ is an occurrence count for key_2.

Map Pseudo-Code

- The map operation, given the *id* of a document and the *document* (or part of the document), emits a collection of key-value pairs, where the key is a term in the document and the value is a (partial) count of the number of occurrences of the word in the document.

```
map(documentId key₁, document value₁) {
   for each term t in value₁ do
      emit(term t, count 1)
}
```

Map Example Inputs/Outputs

Input to map

the reduce operation, given the id of

Output from map

- <the, 1>
- <reduce, 1>
- <operation, 1>
- <given, 1>
- <the, 1>
- <id, 1>
- <of, 1>

Word Count Reduce

- Recall the description of reduce:
  - reduce(key_2, list of values $value_2$) -> [(key_3, $value_3$)]
  - reduce, given a key key_2 output by map, and a collection of *all* the values $value_2$ associated with that key, returns a new collection of key-value pairs.
- In WordCount:
  - key_2 is a term from the document processed by map.
  - $value_2$ is a list of (partial) counts of occurrences of key_2 from map.
  - key_3 is the same as key_2.
  - $value_3$ is the total occurrence count for key_3.

MANCHESTER 1824
The University of Manchester

Reduce Pseudo-Code

- The reduce operation, given a term and a list of partial counts of the term from map, emits a collection of key-value pairs, where the key is the term and the value is the sum of the partial counts.

```
reduce(term key₂, list of count value₂) {
    sum = 0
    for each count in value₂ do
        sum = sum + count;
    emit(term key₂, count sum)
}
```

The University of Manchester

Map Example Inputs/Outputs

Input to reduce

- <the, [1, 1, 1, 1]>

Output from reduce

- <the, 4>

MANCHESTER 1824
The University of Manchester

What Actually Happens?

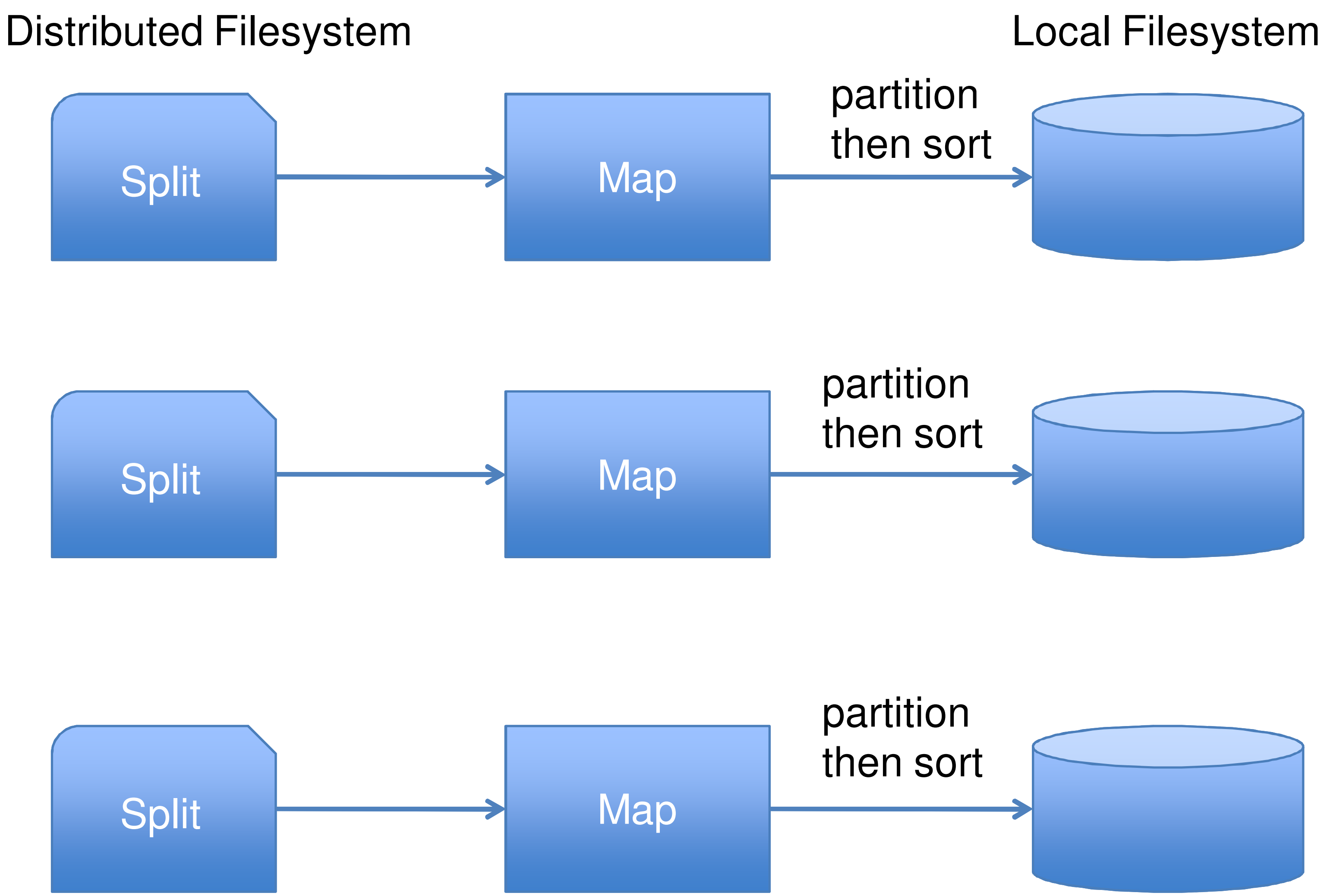
- The whole point of MapReduce is that it scales out (to more nodes). First some terminology:
  - A MapReduce *job* is the unit of work to be performed (the data and the program).
  - A MapReduce job consists of several map and reduce *tasks.*
  - A *task tracker* tracks the progress of each of the map or reduce tasks on a node, and keeps the job tracker informed of progress.
  - A *job tracker* coordinates the different tasks.

The University of Manchester

Map Side

- In terms of map:
  - A map *task* is created for each *split* (i.e. part – often a 64Mb filesystem block) of the input.
  - Wherever possible, the map task will be run on the node where the input data is stored.
  - The map task runs on the split, generating key-value pairs.
  - The output of the map is partitioned into groups that reflect their reduce node, normally by hashing.
  - The partitions are sorted by key.
  - When the output has been written, the task tracker informs the job tracker.

Map side

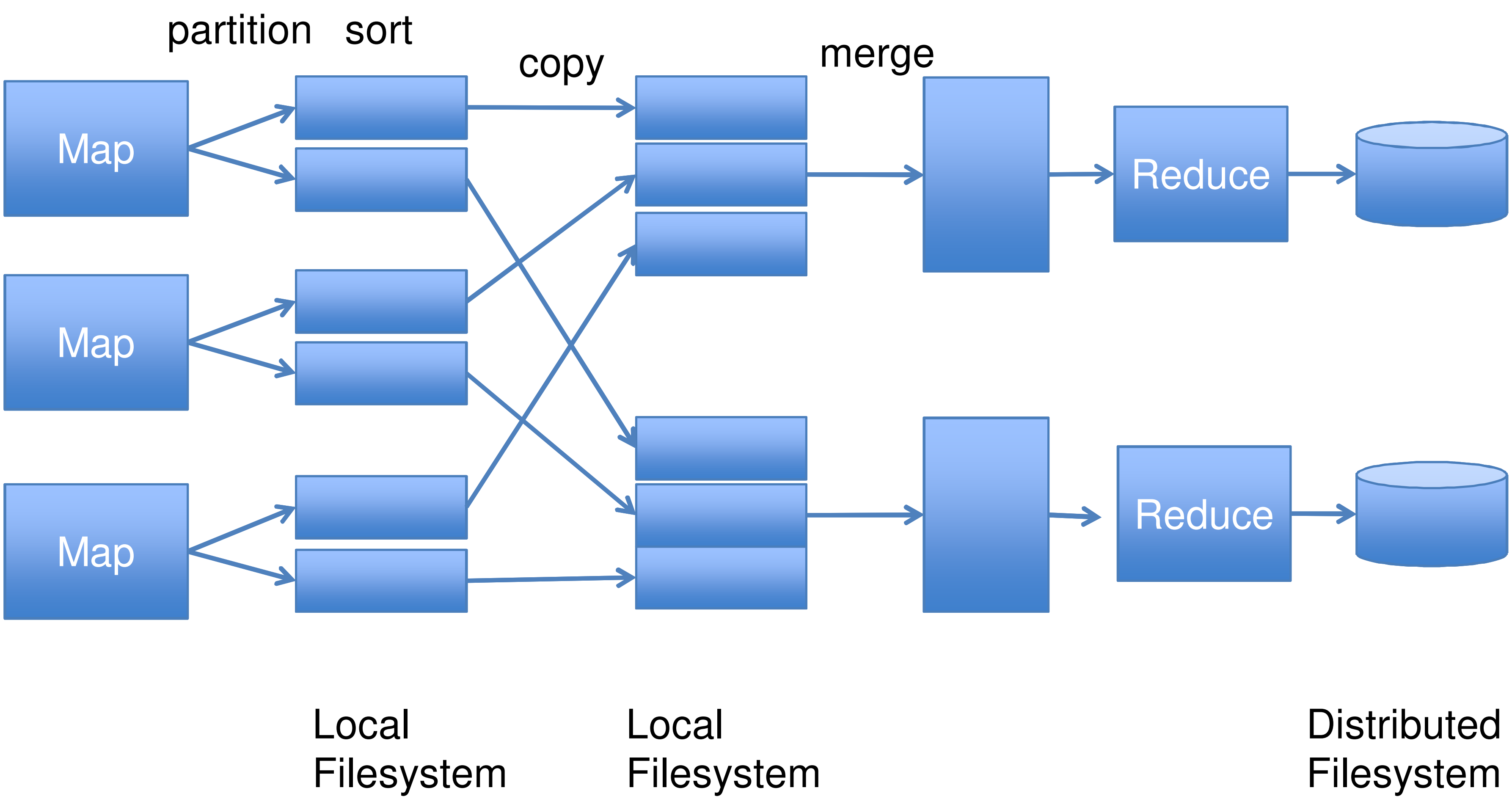

Distributed Filesystem
Local Filesystem
Split
Map
partition
then sort
Split
Map
partition
then sort
Split
Map
partition
then sort

The University of Manchester

Reduce Side

- In terms of reduce:
  - Reduce tasks are created, the task trackers of which poll the job tracker to find out which of the relevant map outputs are complete.
  - The relevant data is pulled over the network.
  - When all the map outputs have been copied the data is merged to produce the inputs for the reduce operation.
  - The reduce operation is run, writing its outputs to the distributed filesystem.

Reduce Side


partition
sort
copy
merge
Map
Map
Map
Reduce
Reduce
Local Filesystem
Local Filesystem
Distributed Filesystem

Activity - 1

- Now you will run a MapReduce job in the workshop.
- Your table will either be:
  - A node that runs a map (3).
  - A node that runs a reduce (2).
  - The job tracker (1).
- Mapper:
  - Someone acts as the task tracker.
  - Someone executes map().
  - Someone executes partition().
  - Someone sorts each partitions.
- Reducer:
  - Someone acts as the task tracker.
  - Someone merges incoming partitions.
  - Someone executes reduce().

Position in Workshop

- Cloud Services.
- MapReduce concepts.
- MapReduce design patterns.
- How MapReduce surfaces in Hadoop.

Basket Analysis in MapReduce

- Another web-scale, data-intensive application is Basket Analysis, where a customer accesses a Web application to shop for products. Each individual interaction by a customer is recorded with information about the value (price) of each basket of products that the customer purchases, as shown in the table on the next slide.
- A simple basket analysis involves calculating the average spent by each customer, considering all the recorded interactions by the customer.

MANCHESTER 1824
The University of Manchester

Basket Analysis

- Note that each customer is associated with a number of interactions, and for each interaction there is a basket value.

CustomerID	BasketValue (in £)
CID_001	26.00
CID_002	30.00
CID_001	40.00
CID_001	20.00
CID_002	35.00
CID_002	25.00
CID_001	10.00
CID_003	25.00
CID_002	35.00

Basket analysis results:
CID_001: 24.00
CID_002: 31.25
CID_003: 25.00

Activity - 2

- Write the basket analysis application as a MapReduce program using pseudo-code.
- A solution to this activity will be made available in Blackboard.

Design Patterns

- Design patterns are a means by which experience and good practice can be passed on.
- In MapReduce, design patterns capture features of implementations that either:
  - enable specific functionalities to be captured within the restrictions of MapReduce, or
  - enable more efficient processing than more naïve implementations.
- Here we discuss one example; for more see: Miner, Donald and Adam Shook, MapReduce design patterns: building effective algorithms and analytics for Hadoop and other systems, O'Reilly, 2012.

MANCHESTER
1824
The University of Manchester

Summarisation

- In summarisation, additional work is carried out within mapper that seeks to reduce the amount of data written to disk and sent over the network to reduce.
- Word count example:
  - Instead of: <the, 1>, <the, 1>, <the, 1>.
  - Summarise as: <the, 3>

Combiner Summarisation

- In Hadoop, as well as *map* and *reduce*, there is an optional *combiner.*
- The infrastructure may choose (or not) to call the combiner, so a developer cannot be sure combiner summarisations will run.
- For WordCount, the combiner can be the reduce operation, which given a collection of values with the same key, aggregates their value.
- Caution is required – only certain operations can perform summarisation without loss of information (they need to be *commutative* and *associative*). Also note that the result type of the reduce is the same as that of the map!

In Mapper Summarisation

- Alternatively, summarisation can take place inside map.

```
map(documentId key1, document
value1) {
   localCache = Hashmap(String ->
Integer)
   for each term t in value1 do {
      count = 1;
      if (localCache.containsKey(t))
         count = localCache.get(t) + 1;
      localCache.put(t,count);
   }
   for each term t in localCache do
      emit(term t, localCache.get(t))
}
```

MANCHESTER 1824
The University of Manchester

Activity - 3

- Write the basket analysis application as a MapReduce program using pseudo-code, in which there is in-mapper summarisation.
- A solution to this activity will be made available in Blackboard.

The University of Manchester

Position in Workshop

- Cloud Services.
- MapReduce concepts.
- MapReduce design patterns.
- How MapReduce surfaces in Hadoop.

MapReduce in Hadoop

- In the labs, you will be writing MapReduce programs, specifically:
  - Making a small functionality change to a given WordCount implementation.
  - Developing a program to build a basic inverted index.
  - Augmenting the basic inverted index with additional functionality from *Documents on the Web*.

The University of Manchester

Where will it run

- The principal focus will be on design and functionality, so you will develop using a *local job runner*.

What does a program look like?

- You will compile a java class definition into a JAR, which hadoop can run.
- The framework knows about your code because you will extend or implement classes and interfaces that are provided by hadoop.

MANCHESTER
1824
The University of Manchester

WordCount Example: map/reduce

```
public class WordCount extends Configured implements Tool
{
    private static class MyMapper extends
            Mapper<LongWritable, Text, Text, IntWritable>
    {
        public void map(LongWritable key, Text value, Context context)
        { …. }
    }
    private static class MyReducer extends
            Reducer<Text, IntWritable, Text, IntWritable>
    {
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context)
        { ….}
    }
}
```

The University of Manchester

WordCount Example: run

```
public int run(String[] args) throws Exception
{
        job.setJobName(WordCount.class.getSimpleName());
        job.setJarByClass(WordCount.class);

        // Set the mapper and reducer classes
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);

        // Set the output classes
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // Set the input and output file paths
        FileInputFormat.setInputPaths(job, new Path(inputPath));
        FileOutputFormat.setOutputPath(job, new Path(outputPath));
       …
}
```

In the labs …

- You can largely ignore most of the material supplied in the word count case, and concentrate on the map, reduce and associated operations.
- You will, however, be asked to consider both performance and functionality aspects of your design, even if you run at small scale.
- There are lots of subtleties and complications we are not introducing or assessing, but hopefully you can get the big picture!

References

- Tom White, Hadoop: The Definitive Guide, Fourth Edition, O'Reilly, 2015.

MANCHESTER
1824
The University of Manchester

Reading for this Week

- Chapters 1 and 2 (you won't need to follow the details of all the examples in Chapter 2) from:
  – Adam Shook, MapReduce design patterns: building effective algorithms and analytics for Hadoop and other systems, O'Reilly, 2012.