Data Analytics with HPC

Hadoop 1: Map reduce

Reusing this material

What is MapReduce

- MapReduce is a parallelisation pattern suitable for distributed systems
  - A programming paradigm/ a way of thinking
- Typically, the programmer supplies **map** and **reduce** functions and some kind of framework implements all the scheduling and data movement required to run the program in parallel
- First publicized by Google to scale their data processing needs (“index the web”)

Map

- A function is “mapped” over all input data
  - The same function is applied to each piece of data:
    - Function f, defined by f(x) =x*x
    - Then map(f,[1,2,3,4,5]) = [1,4,9,16,25]
- The map function used for MapReduce must always return a list of (key,value) pairs
  - Both key and value are derived in some way from the input data
- When you run MapReduce at scale, the Map function is run on every node where the input data resides

Reduce

- Reduce combines data back together
  - Summary operation
- In its simplest case, a single function is applied with multiple arguments:
  - Reduce(+,[1,2,3]) = 6
- In MapReduce the reduce function does not reduce to a single number, but to a set of (key,value) pairs where you end up with a single value for each *input* key

Map Reduce pattern

- Must provide *stateless* Map and Reduce functions:

	Input	Output
Map	<Key1 : Value1>	List(<Key2 : Value2>)
Reduce	<Key2 : List(Value2) >	List(<Key3 : Value3>)

- Framework groups by Key2 before calling reducers
  - Only one reduce call for each unique Key2 key
  - To count words:

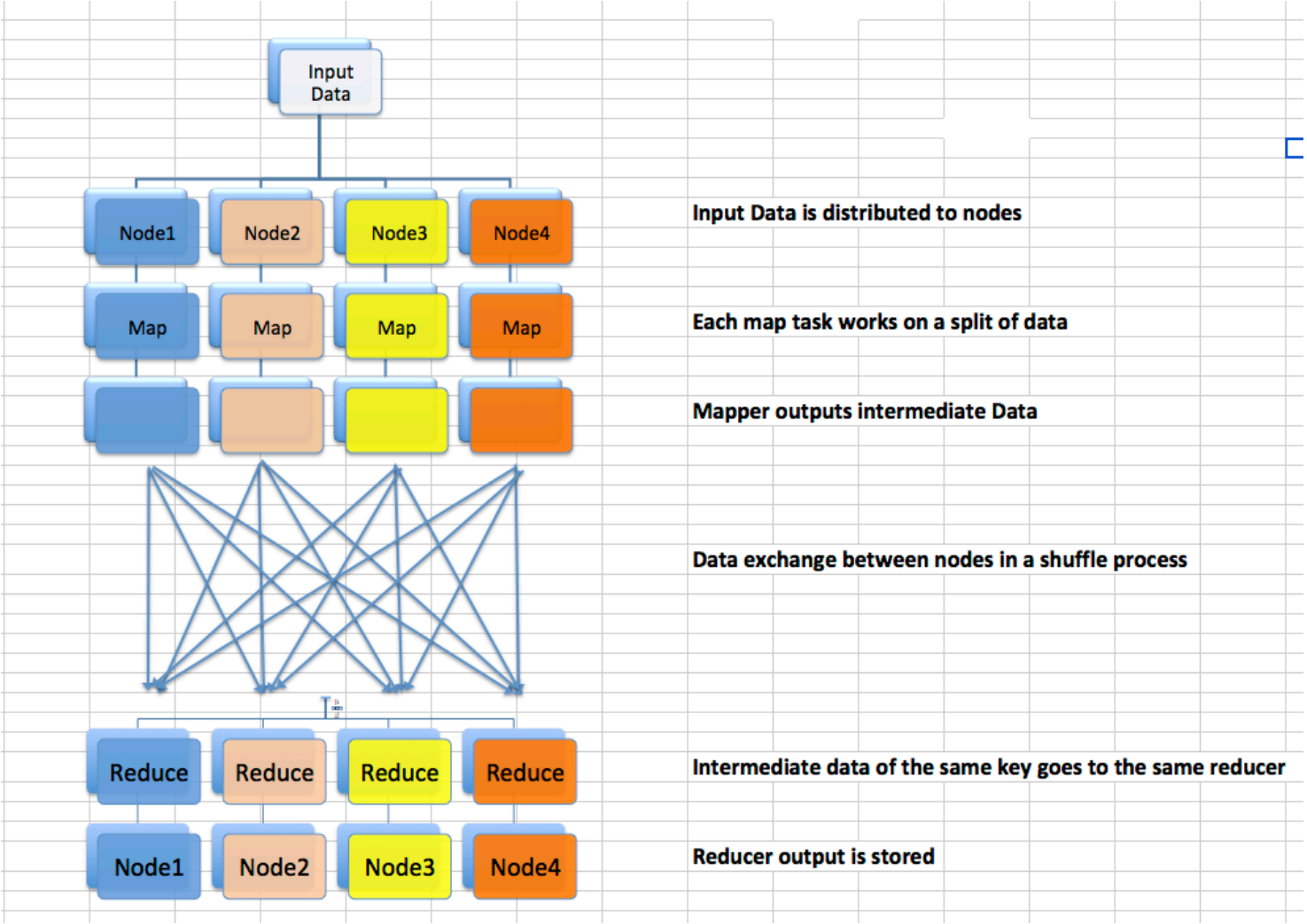
	Input	Output
Map	<223, “shop at my shop”>	[<shop,1>, <at,1>, <my,1>, <shop,1>]
Reduce	<shop, [1,1]>	[<shop, 2>]

	Input	Output
Map	<Integer : Text>	List(<Word : Integer>)
Reduce	<Word : List(Integer) >	List(<Word : Integer>)

epcc

THE UNIVERSITY OF EDINBURGH

Word Count – paper MapReduce exercise



Counting words with Map Reduce

Map Input	Map Output
<0 : “A boy drove a car”>	[<a,1>, <boy,1>, <drove,1>, <a,1>, <car,1>]
<1 : “A car drove at a bus”>	[<a,1>, <car,1>, <drove,1>, <at,1>, <a,1>, <bus,1>]
<2 : “Can a boy drive a car?”>	[<can,1>, <a,1>, <boy,1>, <drive,1>, <a,1>, <car,1>]
<3 : “A danger – a banana!”>	[<a,1>, <danger,1>, <a,1>, <banana,1>]

Reduce Input	Reduce output
<a,[1,1,1,1,1,1,1,1]>	<a,8>
<at, [1]>	<at,1>
<banana,[1]>	<banana,1>
<boy, [1,1]>	<boy,2>
<bus,[1]>	<bus,1>
<can,[1]>	<can, 1>
<car,[1,1,1]>	<car, 3>
<danger,[1]>	<danger,1>
<drive,[1]>	<drive,1>
<drove,[1,1]>	<drove,2>

epcc

THE UNIVERSITY OF EDINBURGH

Map Reduce exercise 1

- From US National Bureau of Economic Research
  - http://www.nber.org/patents/ (Cite75_99.txt)
  - Lists patent IDs and the other patents they cite
    - "CITING", "CITED"
    - 3858241, 956203
    - 3858241, 1324234
    - 3858242, 1515701
    - 3858244, 956203
- **Count the number of times each patent is cited**
  - Tip: Do not need output for patents that are never cited
  - Tip: Reader is easily told to ignore the header row
  - Desired output:
    - 956203, 2
    - 1515701, 1
    - 1324234, 1

	Input	Output
Map	<Key1 : Value1>	List(<Key2 : Value2>)
Reduce	<Key2 : List(Value2) >	List(<Key3 : Value3>)

Map Reduce exercise 1 answer

- Reader: key/value pair both of type integer
- Map: <Integer,Integer> → List(<Integer,Integer>)
  - Extracts the cited patent id and outputs it as key with value 1

Map Input	Map Output
<3858241, 956203>	[<956203,1>]
<3858241, 1324234>	[<1324234,1>]

- Reduce <Integer,List(Integer)> → List(<Integer,Integer>)
  - Simply sums the values as outputs along with the input key

Reduce Input	Reduce output
<956203, [1, 1, 1, 1] >	<956203, 4>
<13242434, [1, 1]>	<13242434,2>

Map Reduce exercise 2

- Same citation data set

```
"CITING", "CITED"
3858241, 956203
3858241, 1324234
3858242, 1515701
3858244, 956203
```

- Different problem: **What patents cite a certain patent**
- Map Reduce Task:
  - Invert citation data set to get for each patent the list of patents that cite it
  - Desired output:

```
956203,  3858241, 3858244
1515701, 3858242
1324234, 3858241
```

Map Reduce exercise 2 answer

- Reader: key/value pair both of type integer
- Map: <Integer,Integer> → List(<Integer,Integer>)
  - Extracts the cited patent id and outputs it as key with citing as va

Map Input	Map Output
<3858241, 956203>	[<956203,3858241>]
<3858241, 1324234>	[<1324234,3858241>]

- Reduce <Integer,List(Integer)> → List(<Integer,String>)
  - C key

Reduce Input	Reduce output
<956203, [3858241, 3858244] >	[<956203, “3858241, 3858244”>]
<13242434, [3858241]>	[<13242434, “3858241” >]

Finding similar patents −exercise3

- Patent data:

Shock absorbent collar for armor plate

US 3858241 A

ABSTRACT

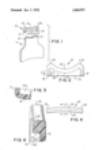
A shock absorbent collar for a protective torso armor plate for human beings made of expanded plastic material. The expanded plastic is crushable and, therefore, impact absorbing. The collar protects the neck, chin, and face or other portions of the head of the wearer of the armor plate in case of sudden deceleration of the body of the wearer of the armor plate, which would shift upwardly in such event and in the absence of the collar would strike the neck or chin or other parts of the head of the wearer with damaging force.

Publication number	US3858241 A
Publication type	Grant
Publication date	Jan 7, 1975
Filing date	Mar 26, 1974
Priority date ⓘ	Mar 26, 1974
Inventors	Durand Philip E, Norris Lonnie H
Original Assignee	Us Army
Export Citation	BiBTeX, EndNote, RefMan

Patent Citations (5), Referenced by (5), Classifications (5)

External Links: USPTO, USPTO Assignment, Espacenet

IMAGES (1)



DESCRIPTION (OCR text may contain errors)

United States Patent Durand et al. 1 Jan. 7, 1975 [5 SHOCK ABSORBENT COLLAR FOR 3,398,406 8/1968 Waterbury 2/2.5 ARMOR PLATE 3,557,384 1/1971 Barron et al 2/2.5 3,634,889 1/1972 Rolsten 2/2.5 [75 Inventors: Philip E. Durand, Hudson;

Lonnie Norris Mllford Primary ExaminerAlfred R. Guest both of Mass- Attorney, Agent, or Firr nNathan Edelberg; Robert T. [73] Assignee: United States of America as Glbson; Charles Raine)! represented by the Secretary of the Army, Washington, DC. ABSTRACT [22] Filed. 26 1974 A shock absorbent collar for a protective torso armor plate for human beings made of

- Patent citation records:

```
"CITING", "CITED"
3858241, 956203
3858241, 1324234
3858242, 151570
3858244, 956203
```

- **How could you identify similar patents?**

Finding similar patents with Map Reduce

Using 'patents frequently cited together' strategy

"CITING"	"CITED"
1111	9999
1111	2222
1111	7777

Using 'patents frequently citing same patents' strategy

"CITING"	"CITED"
1111	9999
3333	9999
8888	9999

Finding similar patents with Map Reduce

- **Using 'patents frequently cited together' strategy**
- First gather all citations made by each patent:

Map Input	Map Output
<"1111", "9999">	[<"1111", "9999">]

Reduce Input	Reduce Output
<"1111", ["9999", "2222", "7777"] >	[<"1111", "9999, 2222, 7777">]

- Next count all pairs that are cited together

Map Input	Map Output
<"1111", "9999, 2222, 7777">	[<"2222+9999", 1>, <"2222+7777", 1> , <"7777+9999", 1>]

Reduce Input	Reduce Output
<"2222+9999", [1, 1, 1, 1] >	[<"2222+9999", 4>]

epcc

THE UNIVERSITY OF EDINBURGH

Finding similar patents with Map Reduce

- **Using 'patents frequently citing same patents' strategy**
- First gather all citations for each patent:

Map Input	Map Output
<"1111", "9999">	[<"9999", "1111">]

Reduce Input	Reduce Output
<"9999", ["1111", "3333", "8888"] >	[<"9999", "1111, 3333, 8888">]

- Next count all pairs that are cited together

Map Input	Map Output
<"9999", "1111, 3333, 8888">	[<"1111+3333", 1>, <"1111+8888", 1> , <"3333+8888", 1>]

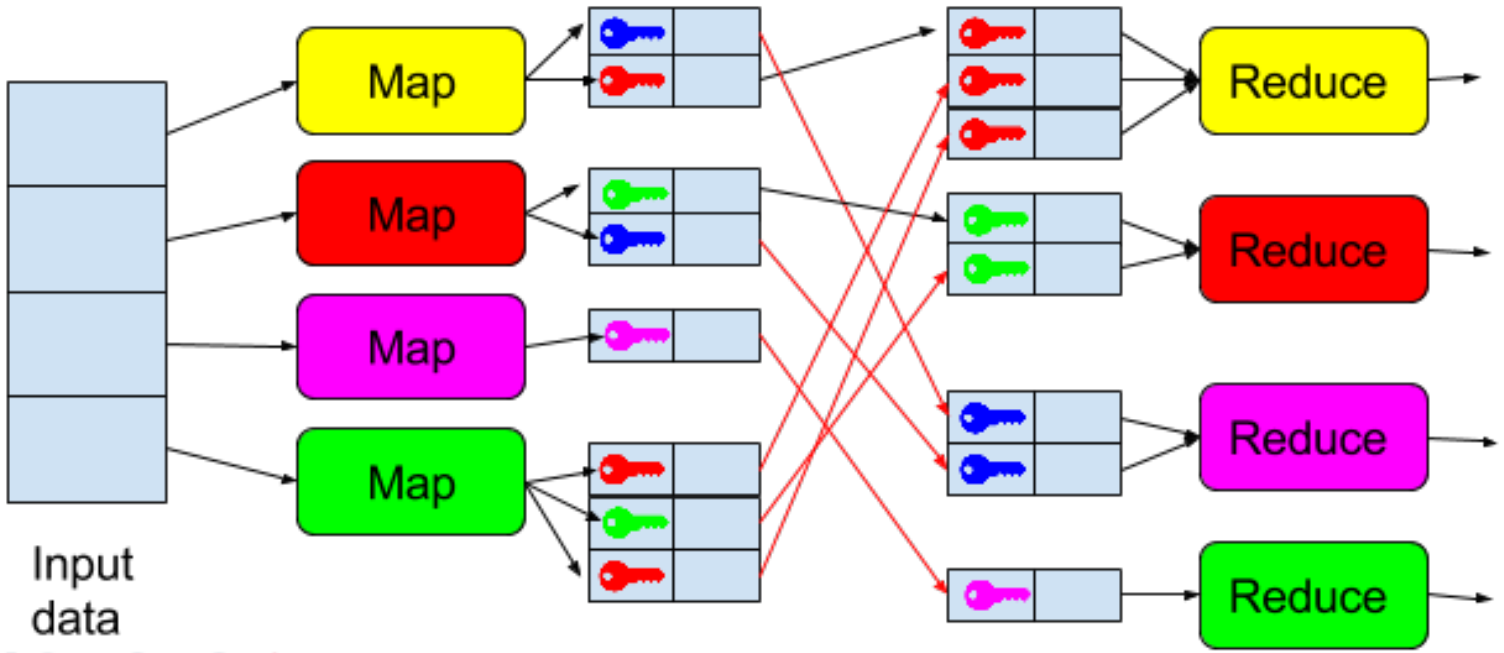
Reduce Input	Reduce Output
<"1111+3333", [1, 1, 1, 1] >	[<"1111+3333", 4>

epcc

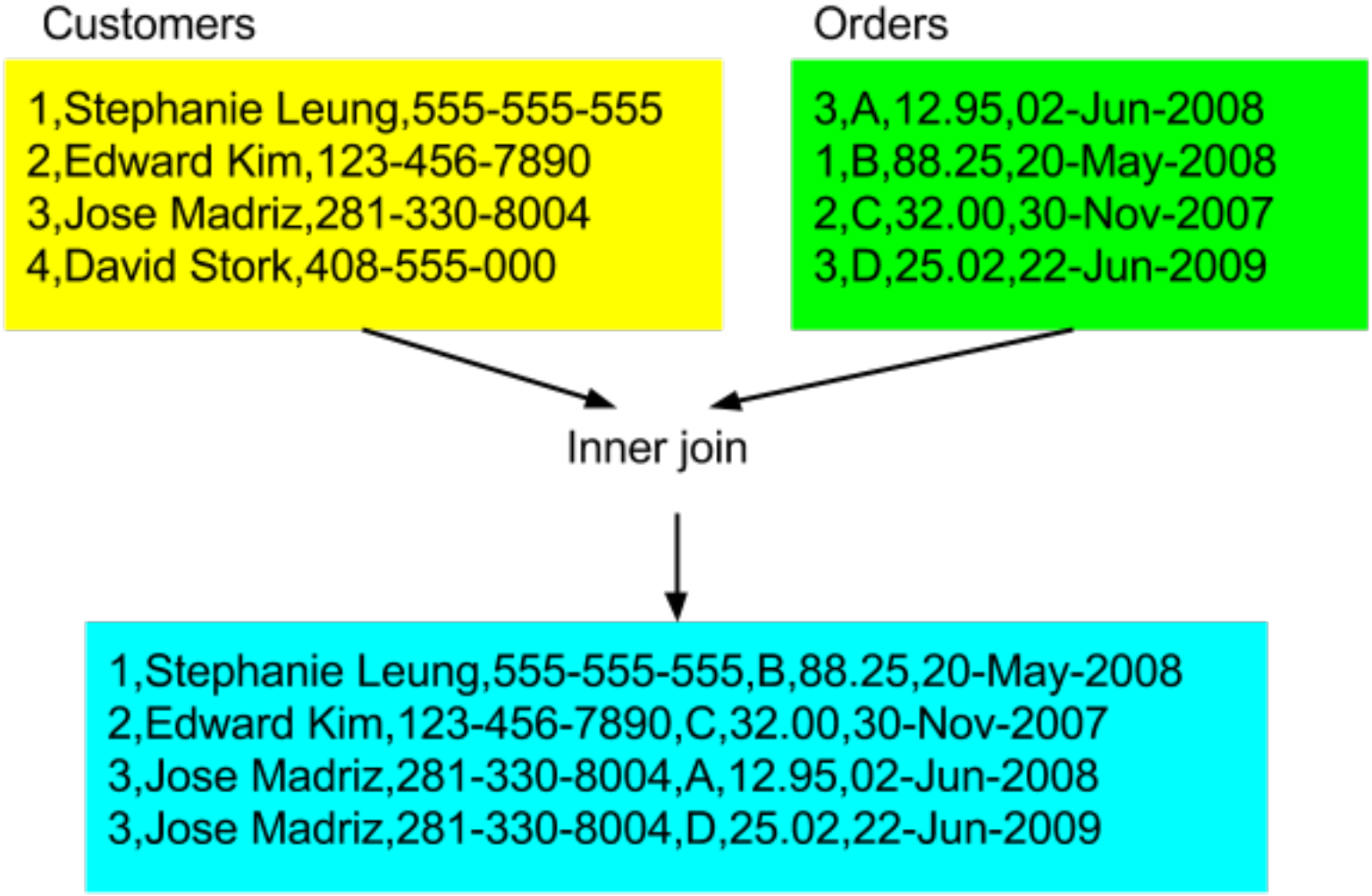
THE UNIVERSITY OF EDINBURGH

Map Reduce at scale

- Stateless map and reduce functions allows massive parallelisation
- Between the Map and Reduce stages the grouping and moving data stage can be expensive

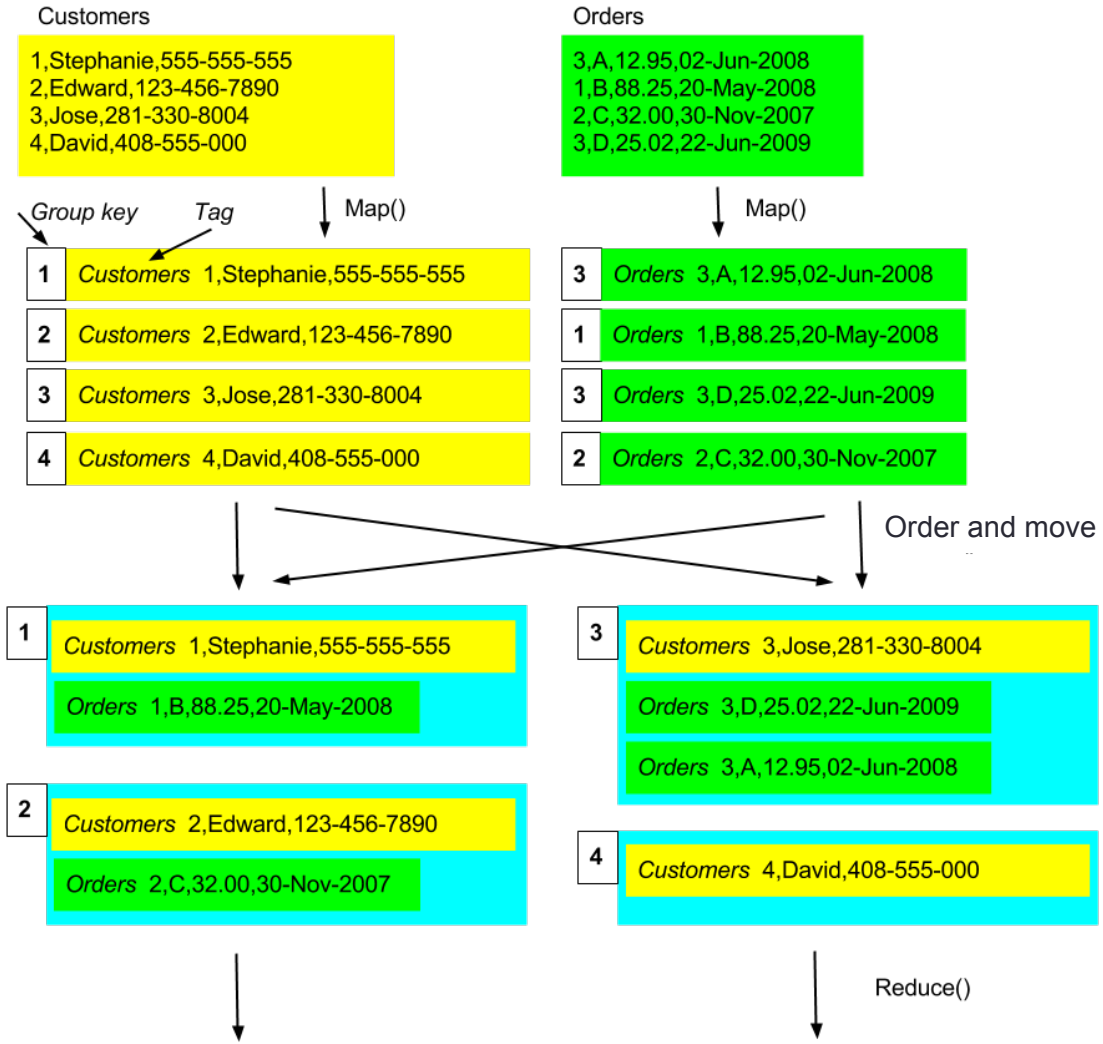


Joining multiple data sets: Inner Join



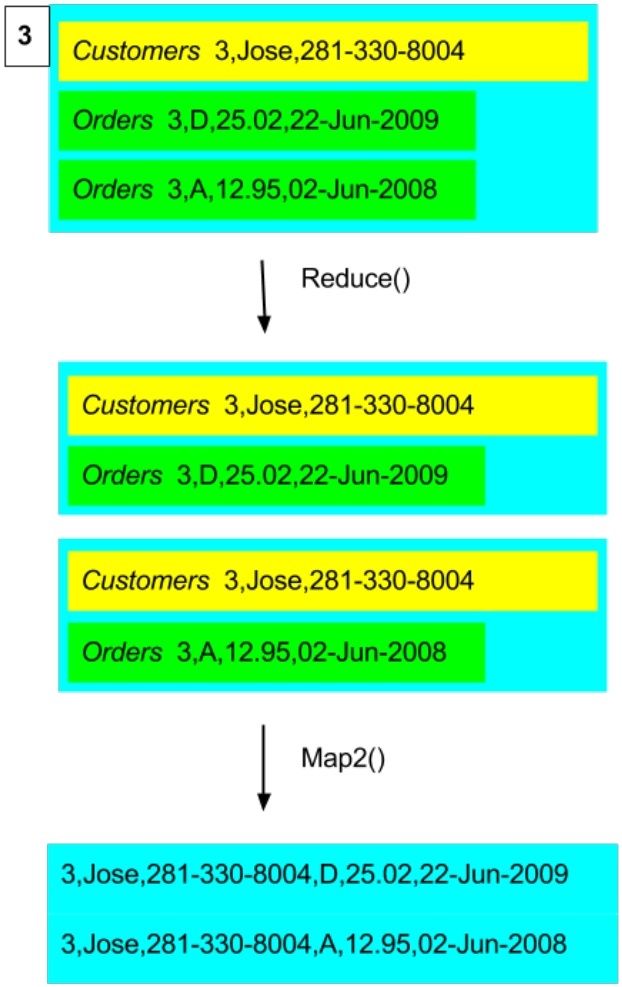
Example from: Hadoop in Action, Chuck Lam

Reduce side join: repartitioned join 1



- Add a tag to store data source filename along with each record
  - To preserve stateless system
  - For state info (metadata) to persist
  - Map function has to be the same on all data
- Group key is the joining attribute
  - Reducer is called on set of records with same group key

Reduce side join: repartitioned join 2



- Reduce produces cross-product of records with a single instance of each tag in each output
- Second Mapper implements join style (inner, outer etc):
  - Reorders
  - Strips out tags
  - Skips customer without orders.
- Hadoop has classes that support such join patterns.

Map side join: replicated joins

- Reduce-side joins (most of processing done on reduce side) require lots of expensive data transfer in shuffle phase.
- If joining one large dataset and one small dataset it may be more efficient to move small dataset to all nodes and then execute the join at the Map stage (and eliminate the Shuffle and Reduce stages).
- Hadoop provides a Distributed Cache to distribute files to all nodes in the cluster.

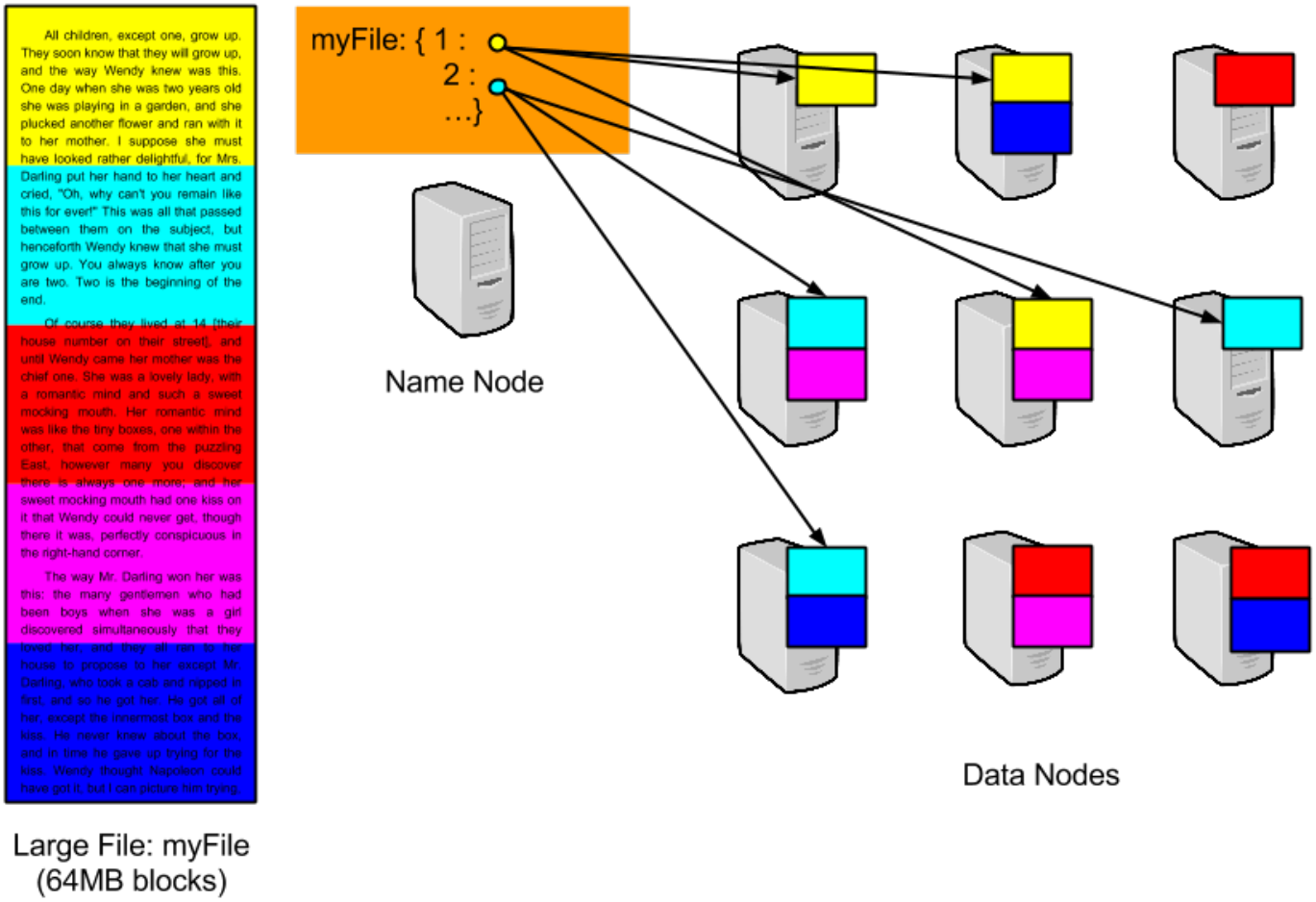
Alternatives to replication join

- Sometimes data sets are just too big for replication join
- Reduce data transfer by map-side filtering
  - Reduce amount of data transfer by filtering to only those records of interest, e.g. only those customers who live in Scotland.
    - Note: applying such a filter may make the data set small enough to use the replicated join strategy.
  - Replicate only the join keys rather than the whole records
    - Thus only data which will actually be joined is transferred
  - If join keys are still too large consider a smaller data structure that gives an approximate answer, e.g. Bloom filter
    - BloomFilter.contains(x) – returns true if x is in filter
    - BloomFilter.contains(x) – returns either true or false if x is not in filter.
    - Level of false positives related to the size of the filter.

Hadoop

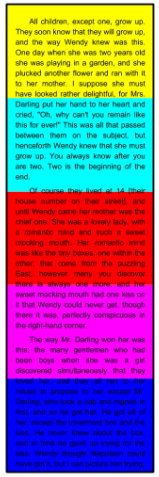
- Hadoop with MapReduce
  - Map
  - Reduce
  - Sort
  - Shuffle
- Hadoop Distributed File System (HDFS) – distributed storage

Hadoop Distributed File System

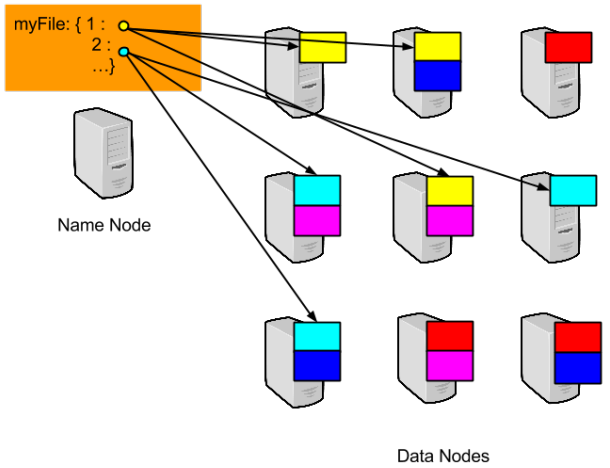


Hadoop Distributed File System

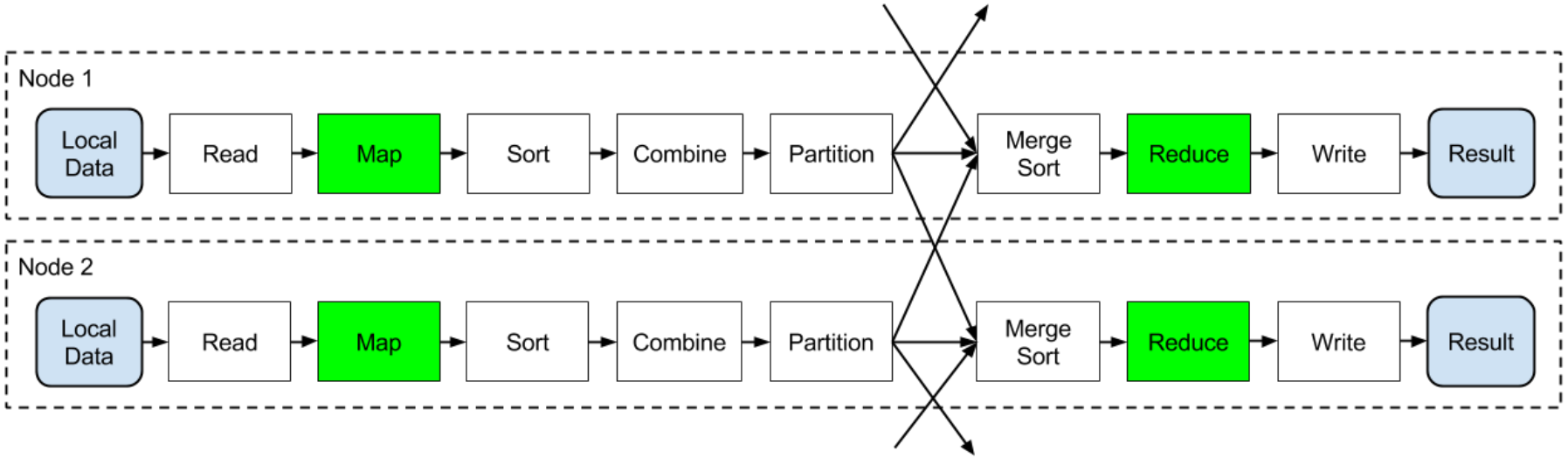
- Typical use: write once, read many
  - Computation runs on Data Nodes
- Distributed
- Data redundancy
- Cluster of commodity nodes
- Designed to withstand failure
  - But Name Node is a single point of failure (see secondary name node)
- Optimised for the tasks in hand
  - Not a POSIX file system
- Placement strategies can be aware of data centre configuration



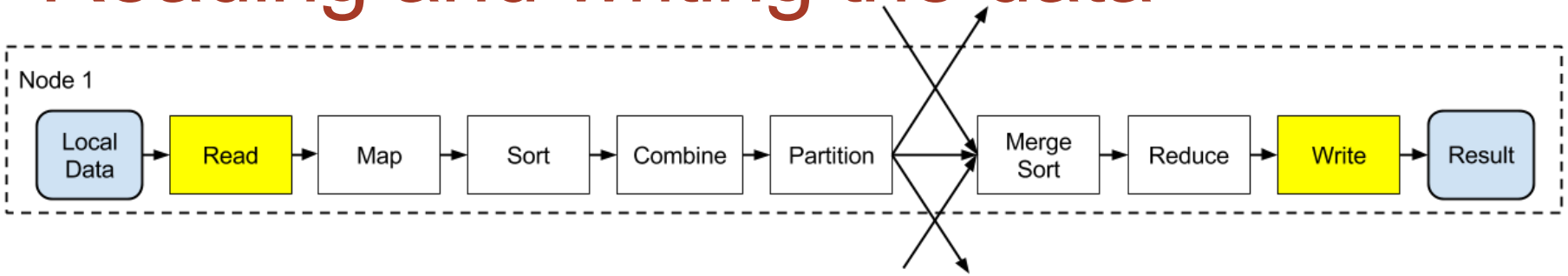
Large File: myFile
(64MB blocks)



Hadoop Framework

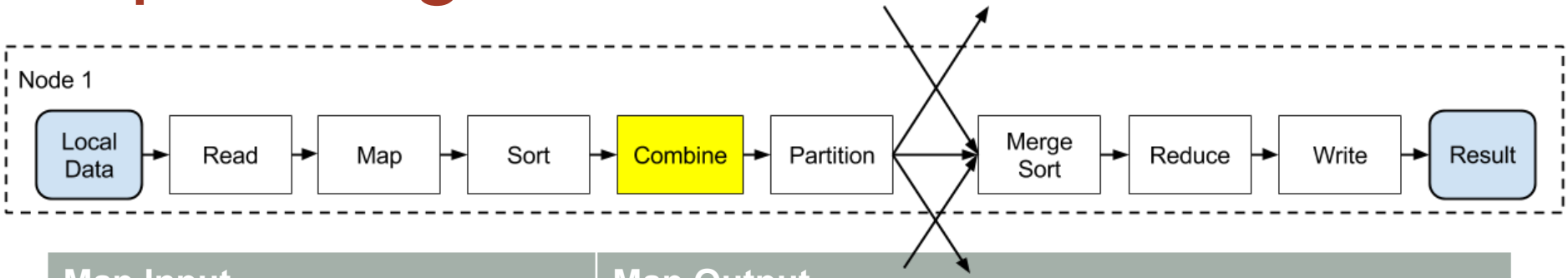


Reading and writing the data



- InputFormat interface
  - TextInputFormat (key: byte offset of line, value: line text)
  - KeyValueTextInputFormat (each line has key/separator/value)
  - SequenceFileInputFormat (Hadoop's compressed binary format)
  - NLineInputFormat (like TextInputFormat but multi-line)
- OutputFormat interface
  - TextOutputFormat (one record per line, key/separator/value)
  - SequenceFileOutputFormat (compressed binary)
  - Filename is "part-*xxxx*" where *xxxx* is the partition ID

Optimising with a combiner

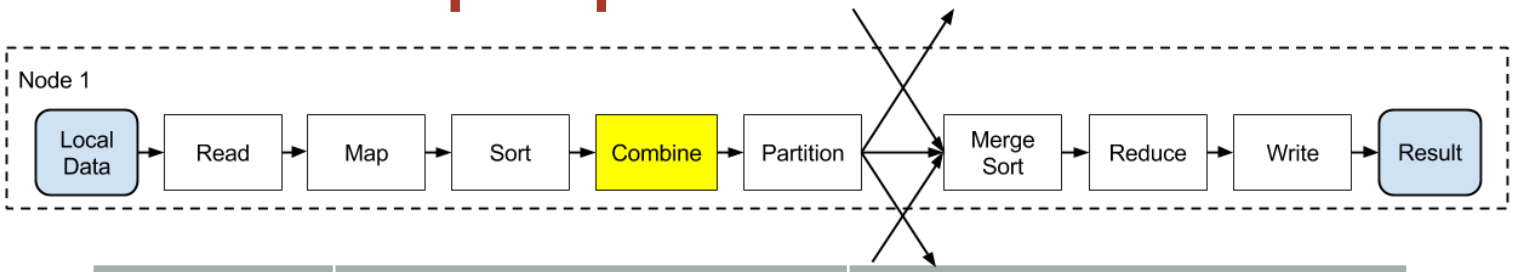


Map Input	Map Output
<0 : “A boy drove a car”>	[<a,1>, <boy,1>, <drove,1>, <a,1>, <car,1>]
<1 : “A car drove at a bus”>	[<a,1>, <car,1>, <drove,1>, <at,1>, <a,1>, <bus,1>]
<2 : “Can a boy drive a car?”>	[<can,1>, <a,1>, <boy,1>, <drive,1>, <a,1>, <car,1>]
<3 : “A danger – a banana!”>	[<a,1>, <danger,1>, <a,1>, <banana,1>]

Local reduction operation = reduce communications necessary

Combiner Input	Combiner output
<a,[1,1]>	<a, [2]>

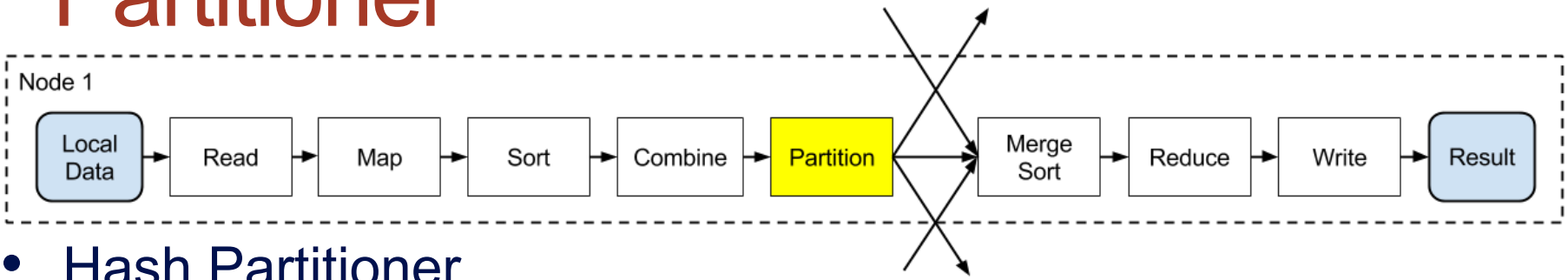
Combiner properties



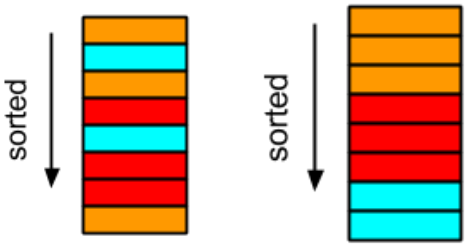
	Input	Output
Map	<Key1 : Value1>	List(<Key2 : Value2>)
Combine	<Key2 : List(Value2) >	<Key2 : List(Value2) >
Reduce	<Key2 : List(Value2) >	List(<Key3 : Value3>)

- Optimisation only
  - Framework may execute zero, one or more times
  - Must not alter the final result
  - A helper to the reducer
- Keys *must* not be altered
  - Hadoop does not re-sort after the Combine stage

Partitioner



- Hash Partitioner
  - Default
  - Everything with same key will be on same node
  - Diff. keys can end up on same node, too

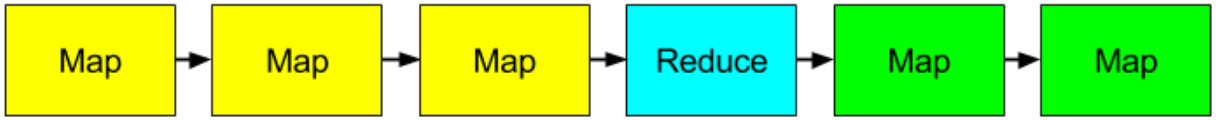


- Total Order Partitioner
  - Maintains order
  - Configure to partition evenly

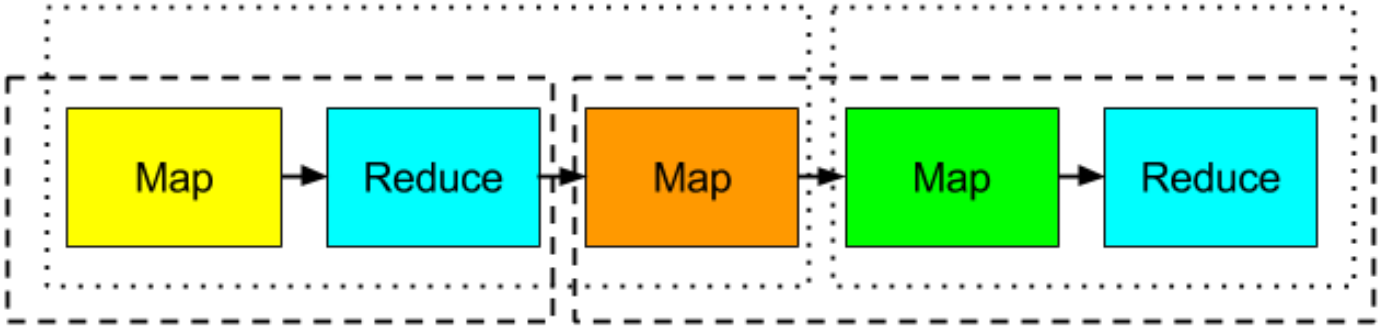
- Bespoke
  - For highly skewed data hash partitioner may not partition work evenly
  - Maybe some keys require more processing by Reducer

Chaining Map Reduce Jobs

- A single map reduce job has
  - One REDUCE stage
  - One or more MAP stages before the reduce
  - Zero or more MAP stages after the reduce

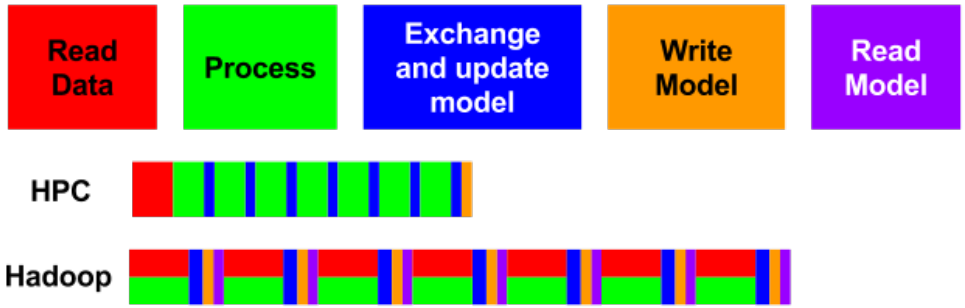


- Need to chain multiple map reduce jobs when:
  - There is more than one REDUCE stage (grouping of data by key)
  - MAP stages between REDUCE jobs could be part of either job



Chain, but don't iterate

- Each Hadoop job reads data from the HDFS and writes output to the HDFS
  - No data is maintained in memory between jobs
- Fine for short chains of processing
- Very inefficient for iterative algorithms
  - Data (even static data) must be read from disk at each iteration



- Spark – supports caching data
- Twister – iterative map reduce

Programming Hadoop

- Hadoop framework is written in Java
- Two models for writing Map, Reduce and Combine functions
  - Java classes
  - Hadoop **streaming**
    - Functions are scripts that read from standard input and write to standard output
- If writing your own partitioners or getting into the internals of Hadoop you will need to use Java
  - But for most problems you do not need to do this.

Map class in Java

Mapper<
InputKeyType, InputValueType,
OutputKeyType, OutputValueType >

Must implement function:
void map(InputKeyType, InputValueType, Context)

```
public static class MapClass
  extends Mapper<Text, Text, Text, Text>
{
    public void map(Text key, Text value, Context context)
    {
       context.write(value, key);
    }
}
```

This mapper simply swaps the key and value

write output data using context.write(outputKey, outputValue)

Can call multiple times and hence output List(<OutputKeyType, OutputValueType>)

Reduce class in Java

Reducer<
InputKeyType, InputValueType,
OutputKeyType, OutputValueType >

```
public static class Reduce
    extends Reducer<Text, Text, Text, Text>
{
  public void reduce( Text key,
                      Iterable<Text> values,
                      Context context)
  {
    String csv = "";
    for (Text val:values)
    {
      if (csv.length() > 0) csv += ",";
      csv += val.toString();
    }
    context.write(key, new Text(csv));
  }
}
```

Uses iterator to get list of values – can thus support large lists with low memory footprint. So long as the rest of the method is similarly low memory. This example is not!

write output data using
context.write(outputKey, outputValue)
Can call multiple times if desired

epcc

THE UNIVERSITY OF EDINBURGH

Streaming Mapper (see demo)

- Input: rows of key/value pairs separated by TAB character
- Output: rows of key/value pairs separated by TAB character
- **Stateless**
  - Process one line at a time with no state maintained between lines.

```
1<TAB>A long time ago
2<TAB>in a galaxy far
3<TAB>far away
```

```
a<TAB>1
long<TAB>1
time<TAB>1
ago<TAB>1
in<TAB>1
a<TAB>1
galaxy<TAB>1
...
```

Streaming Reducer

- Input is rows of key/value pairs separated by TAB character
- Input guarantees that all the key/value pairs associated with a specific key will be contiguous in the input stream
  - When key changes you know you have seen all the values associated with that key
- Output rows of key/value pairs separated by TAB character
- Stateless
  - Can maintain state while processing rows with the same key.
  - Must not maintain state across rows with different keys

Streaming Reducer

```
a<TAB>1
a<TAB>1
a<TAB>1
far<TAB>1
far<TAB>1
time<TAB>1
```

```
a<TAB>3
far<TAB>2
time<TAB>1
```

Hadoop vs MPI/HPC

- Fault tolerance
  - Hadoop is designed specifically with fault tolerance in mind
  - MPI provides little support for fault tolerance and most MPI programs assume the system hardware will not fail
- Specific vs general
  - Hadoop is a framework for a specific data processing pattern
  - MPI allows you to code any algorithm you wish
- Iterative algorithms
  - Hadoop very poor at multiple iterations over the data
  - Very easy to write such programs in MPI
- Speed
  - If you have a reliable HPC system an optimised MPI implementation should perform considerably better than Hadoop

Hadoop vs MPI/HPC cont.

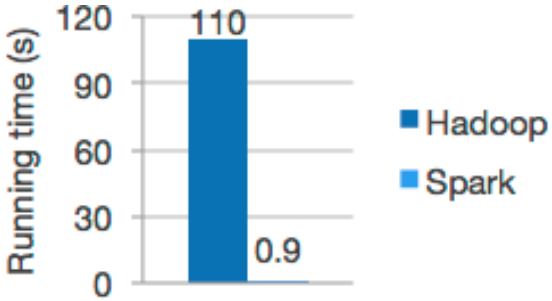
- Cost
  - Hadoop simple to write and can run reliably on commodity hardware.
  - MPI typically run on expensive HPC systems
    - MPI can run on clouds but have to build your own fault tolerance.
- Dynamic nature of data
  - Hadoop is good for processing massive amounts of data that is written once and processed often
  - HPC systems may not scale well to such massive datasets being uploaded.

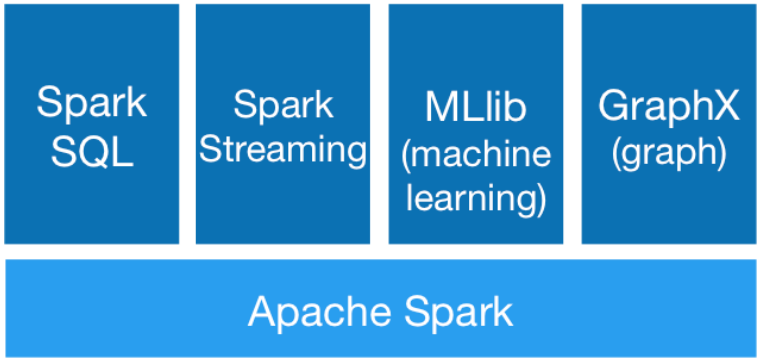
Hadoop Ecosystem

- HBASE
  - Distributed, scalable big data store
  - Columnar database
- PIG
  - Higher level data flow language for programming Hadoop
- Mahout
  - Scalable machine learning and data mining over Hadoop
- Spark
  - Machine learning algorithms

A little more on Spark

- Explicitly supports caching data
  - Speeds up iterative algorithms
- Can use HDFS as the data source
- More that just map/reduce
  - Transformations:
    - map, filter, union, Cartesian, join, sample…
  - Actions:
    - reduce, collect, count, first, countBy, foreach…





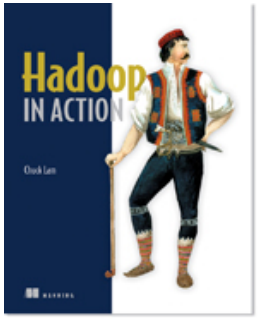
epcc

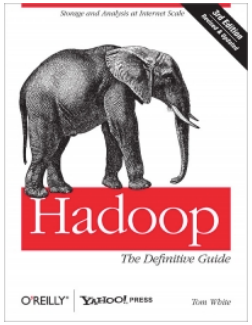
THE UNIVERSITY OF EDINBURGH

Additional reading

- Google File System
  - http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf
- Map Reduce
  - http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

- Examples taken from *Hadoop in Action*
  - http://www.manning.com/lam/
- For Hadoop 3, O'Reilly's *Hadoop, The Definitive Guide* is good.
- Plenty online





Thanks

- Ally Hume, EPCC
- Adam Carter, EPCC
- Eilidh Troup, EPCC