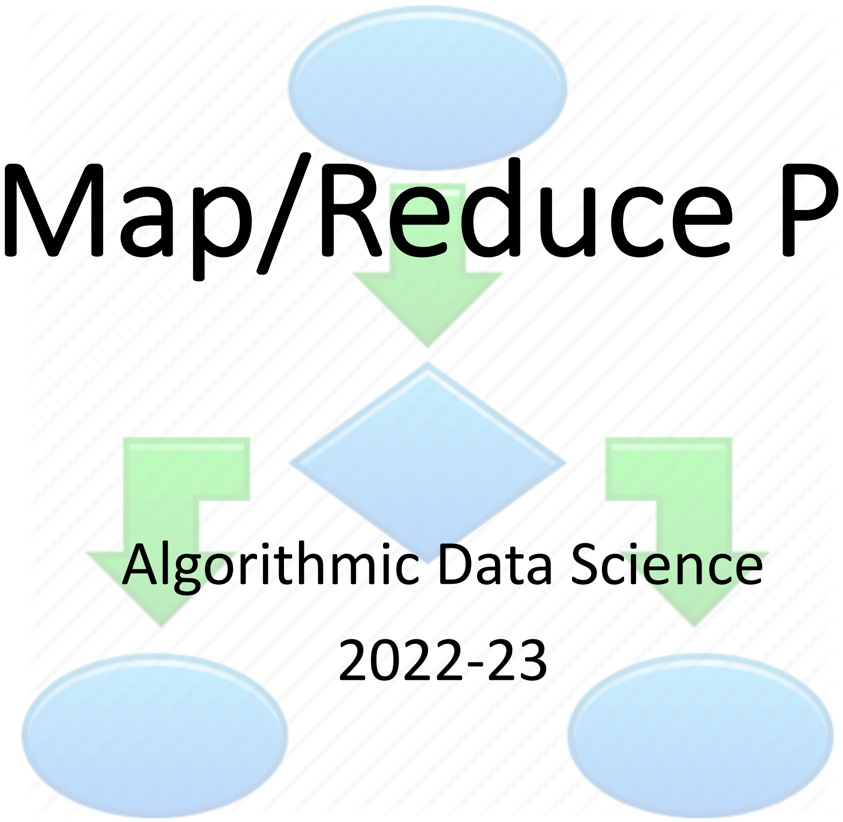
Week 7: Map/Reduce Paradigm



Algorithmic Data Science

2022-23

UNIVERSITY OF SUSSEX

Dr Adam Barrett

Warm-up revision exercise

- What does it mean for two matrices to be compatible for multiplication?
- Write down an algorithm for matrix multiplication of 2 general matrices that are compatible for multiplication, in Python or pseudo code (not just in 1 line with np.mult(A, B)!)

```
def mult(A, B):

    (m, n) = np.shape(A)
    (n, p) = np.shape(B)

    C = np.zeros(m, p)

    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i,j] += A[i,k] * B[k, j]

    return C
```

Week	Who	Topic
1	Barrett	Data structures and data formats
2	Barrett	Algorithmic complexity. Sorting.
3	Barrett	Matrices: Manipulation and computation
4	Barrett	Similarity analysis
5	Rosas	Processes and concurrency
6	Rosas	Distributed computation
7	**Barrett**	**Map/reduce**
8	Barrett	Clustering, graphs/networks
9	Barrett	Graphs/networks, PageRank algorithm
10	Barrett	Databases
11		*independent study*

Padlet for questions

https://uofsussex.padlet.org/abb22/6u9itp7fhshvdg0u

Today

- Space and communication complexity
- MapReduce

Space Complexity

- The space complexity of an algorithm A:X → Y for input *x* is the number of memory cells it takes to execute A(x)
- For a fixed notion of size, then a space bound of A:X→ Y is a function $space$ that for all $x \in X$,

the memory consumption of A(x) $\leq space(size(x))$

We can define complexity classes for space in the same way as for time. We can refer to **linear-space** algorithms and **polynomial-space** algorithms

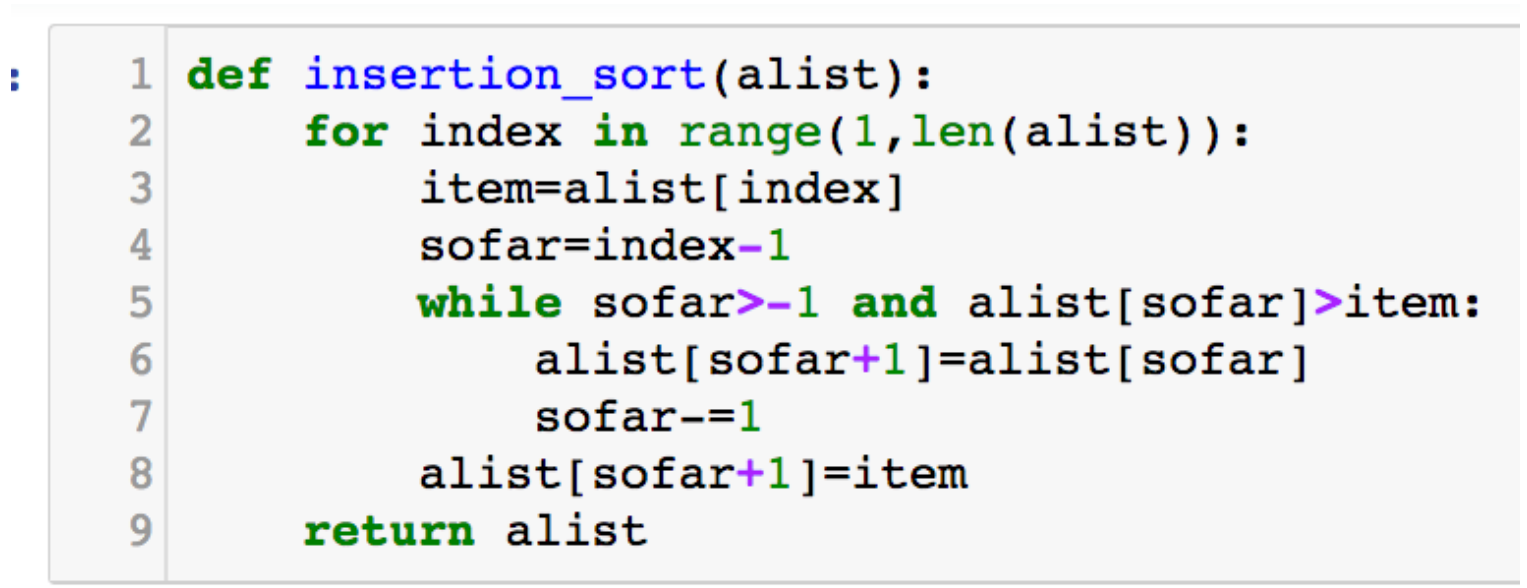
Relationship between Time and Space

- You cannot use more space than time (think about why).

- Therefore, for all A:X→Y the **set of all possible time bounds** is a **subset** of the **set of all possible space bounds**.

- Your *O* notation complexity for space will be something that grows no quicker than your *O* notation complexity for time.

Space complexity / Memory requirements



```python
def insertion_sort(alist):
    for index in range(1,len(alist)):
        item=alist[index]
        sofar=index-1
        while sofar>-1 and alist[sofar]>item:
            alist[sofar+1]=alist[sofar]
            sofar-=1
        alist[sofar+1]=item
    return alist
```

For each element in the list, find its correct place in the already sorted list to the left. Insert it (by shifting everything up) and proceed to next element.

No new lists created, so (asymptotic) memory requirement is just the size of the inputted list:

a*n

where a is memory per list item.
(Can neglect variables of size O(1).)

Naïve matrix multiplication

```
Matrix-Multiply (A,B):
  # Assume A and B are nxn matrices:
    let C be an nxn matrix
    for i in range(1,n):
      for j in range(1,n):
        $c_{ij}$= 0
        for k in range(1,n):
          $c_{ij}$ += $a_{ik}$ * $b_{kj}$
  return C
```

Whole new matrix created, so (asymptotic) memory requirement is the size of the inputted matrices plus the size of the output

$3*a*n^2$

where a is memory per matrix component. (Can neglect variables of size O(1) and O(n).)

Time-space-tradeoff

- Ideally, we want to minimise time complexity and space complexity.
- Sorting can be performed in place so lower bound on space complexity for sorting = O(n)
- In practice, there is usually a tradeoff between time and space complexity. We may use more space to get a faster run-time.
- Insertion-sort sorts in place so space complexity = $O(n)$ and time complexity = $O(n^2)$
- Merge-sort creates new lists at each level of list division so a naïve implementation has space complexity = $O(n \log n)$ and time complexity = $O(n \log n)$

Communication Complexity

- How many packets of data need to be exchanged if processors jointly solve a problem in parallel?
- How does this depend on the size of the problem instance?

Alice and Bob

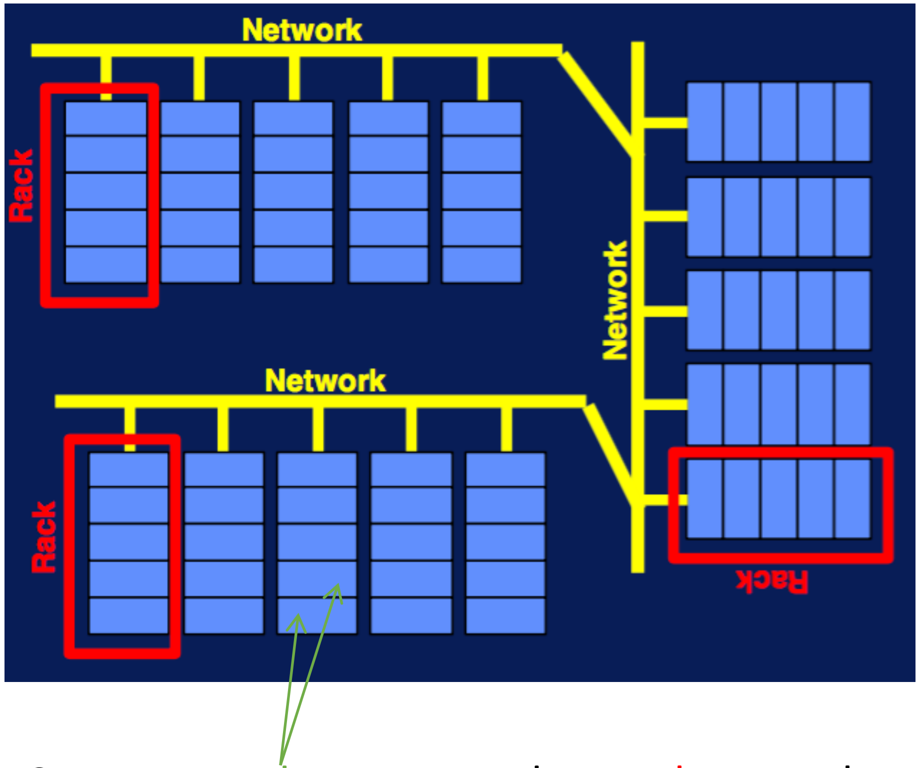
- Alice and Bob are two separated parties.
- Alice receives an n-bit string x and Bob receives an n-bit string y.
- The goal is for one of them to compute a certain function f(x,y).
- How many bits of data need to be communicated between them?
- Obviously, they will succeed if Alice sends Bob n bits (or Bob sends Alice n bits) who then computes f(x,y)
- Is it possible to calculate f with fewer than n bits of communication?
- Sometimes yes. . . Let’s see.

Map/Reduce - Overview

- Recap: distributed computing
- MapReduce and the Hadoop Ecosystem
- Word counting using MapReduce
- Execution of MapReduce algorithm
- Coping with node failures
- When not to use MapReduce
- Matrix-vector multiplication with MapReduce
- Matrix multiplication with MapReduce
- Extending MapReduce
- Communication-cost model and wall-clock time

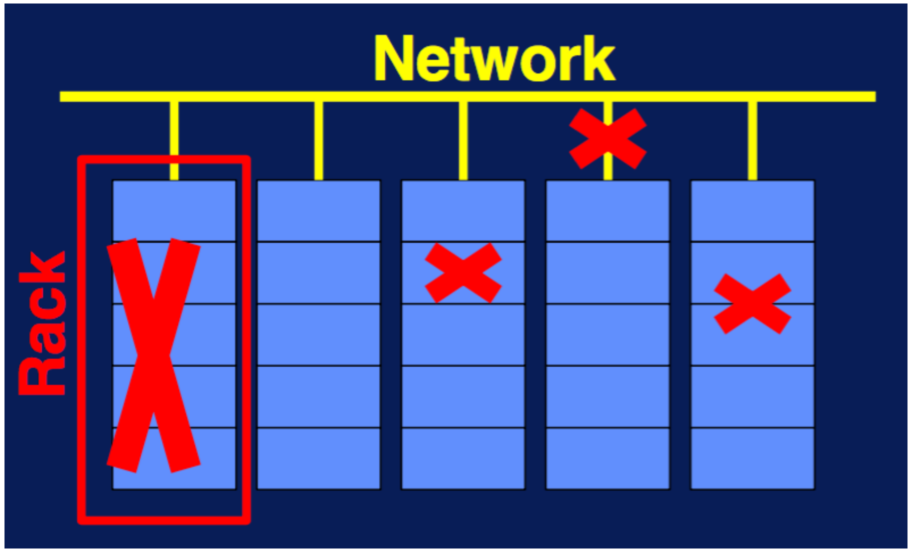
Recall: Data centres

- Big Data analysis can be greatly speeded up through parallel and distributed computation.
- Data centres provide parallelism through **computing clusters** – large collections of commodity hardware, including conventional processors (**"compute nodes"**) connected by ethernet cables



Compute nodes are stored on racks – maybe 8-64 on a rack (5 per rack shown in the diagram).

Recall: Fault tolerance



Components (including individual nodes, entire racks and network connections) can fail. The more components a system has, the more likely it is for one to fail.

To avoid having to abort and restart an entire computation every time a component fails, systems should be fault tolerant. This involves:

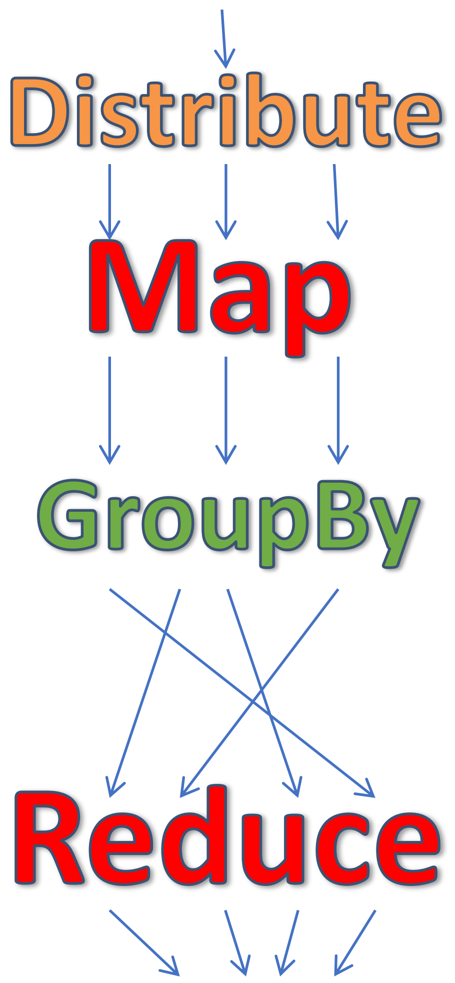
- file replication at different nodes (e.g., x3 in HDFS)
- division of computation into tasks, such that if one fails to complete it can be restarted without affecting other tasks (e.g., MapReduce)

Recall: Distributed file systems

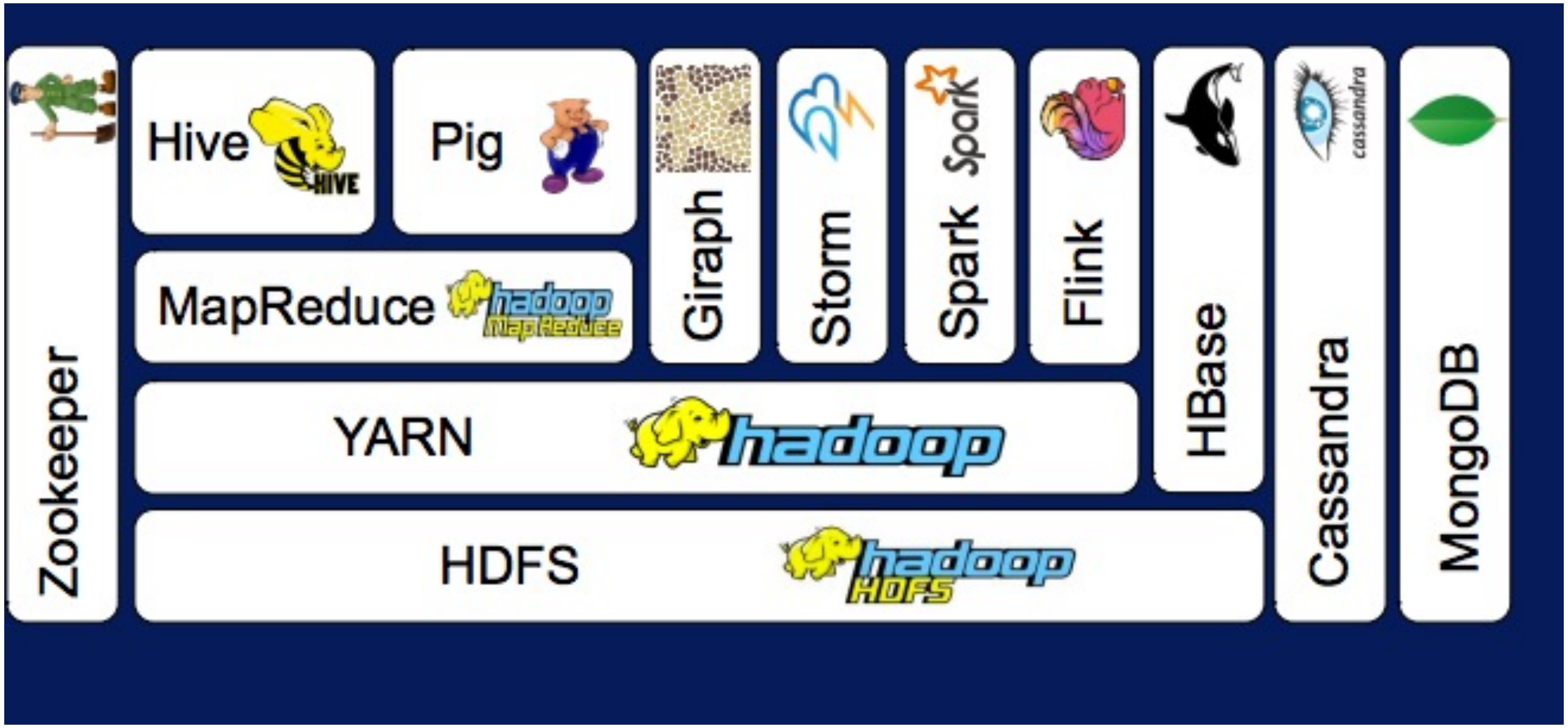
- A DFS may be suitable when
  - Files are enormous (maybe a TB or more)
  - Files are rarely updated. They are appended or read sequentially (read or write random access not required)
- Files are divided into **chunks.**
- Typically (e.g., HDFS), chunks are 64MB and are replicated 3 times at 3 different compute nodes (on 3 different racks).
- The **name node** (or master node) for a file stores where the chunks are to be found.
- The name node is replicated and the directory for the file system knows where to find the copies
- The directory itself can be replicated and all users know where the directory copies are.

MapReduce

- a computing paradigm
- developed by Google for indexing web pages and published in 2004
- open-source implementation Hadoop first made available by Yahoo in 2005
- All you write are two functions ***Map()*** and ***Reduce()***
- The system manages the parallel execution and coordination of the tasks that execute Map and Reduce, as well as dealing with the possibility that one of these tasks might fail.



Hadoop Ecosystem



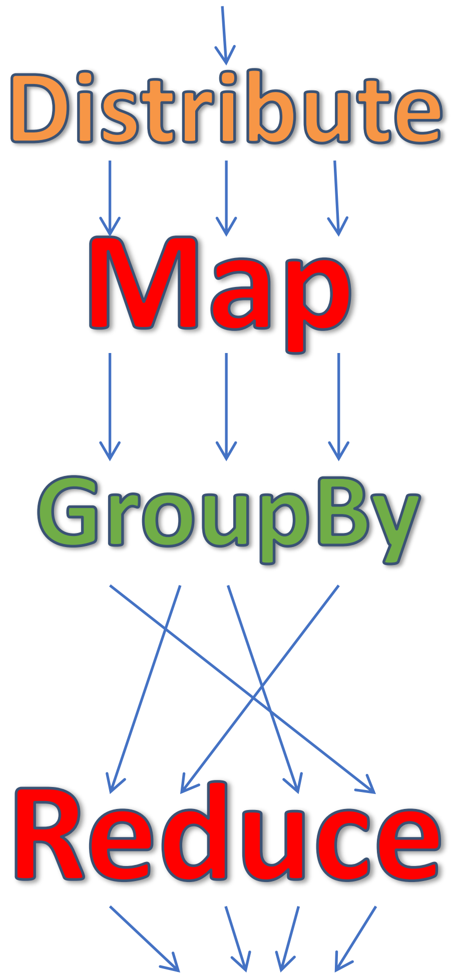
A first MapReduce algorithm

I want to know the frequency of occurrence of all of the words in a number of texts.

How to go about computing that distribution efficiently?

Counting words with MapReduce

- Chunks of the input (individual texts) are distributed to different nodes.
- A worker at each node works out the frequency of each word in its chunk
- Key-value pairs are stored in an intermediate file according to some hash function (e.g., the first letter of the word)
- Intermediate files transferred to the node responsible for those key-values pairs
- A worker at that node adds up the associated key-value pairs and produces a single file with the output.



Map Tasks

- turn a chunk into a sequence of key-value pairs
- Keys do **not** have to be unique

```
def map_for_word_count(document):
  for line in document:
    for token in line:
      yield(token,1)
```

This simply iterates over every token in every line of the open file and yields or outputs a (token,1) pair for every token

- If a token *w* appears *m* times among all the documents assigned to a given Map task, then there will be *m* key-value pairs (*w,1*) among its output.
- If it makes sense as a step, then Map tasks may also **combine** the *m* pairs into a single pair (*w,m*). This is not necessary but is sometimes done to reduce the size of the intermediate files.

Grouping By Key

- Key-value pairs are grouped by key and values associated with each key formed into a list of values.
- Carried out by the system, regardless of what the Map and Reduce functions are.
- The user sets the number of Reduce tasks *r.*
- The master controller process picks a hash function that applies to keys and produces a bucket number from 0 to *r-1*.
- Each key that is output by a Map task is hashed and its key-value pair put in one of *r* local files.
- The master controller typically merges the files from each Map task that are destined for a single Reduce task into a sequence of key-list-of-value pairs

Reduce Tasks

- The Reduce function's input is a pair consisting of a key and a list of values. The output is a sequence of 0 or more key-value pairs.

```
def reduce_for_word_count(key,valuelist):

  total=0
  for value in valuelist:
    total+=value
  yield (key,total)
```

Here we add up all of the values in the valuelist and output a single key,value pair.

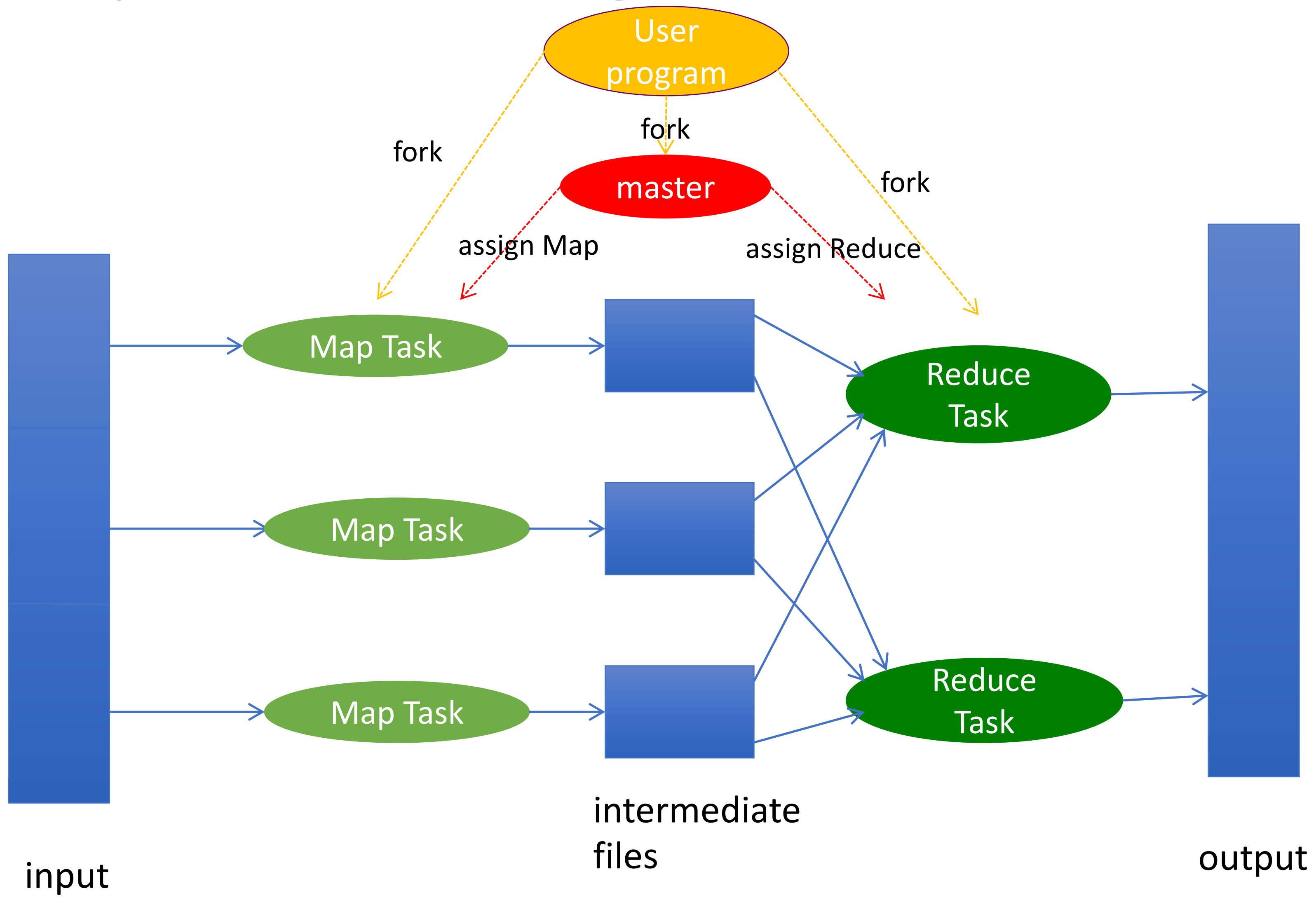
Number of Reduce Tasks

Usually, a single Reduce task will execute multiple reducers (i.e., for multiple keys) since:

- usually more keys than available compute nodes
- significant overhead associated with each reduce task we create e.g., number of intermediate files

Significant variation in lengths of the value lists for different keys can lead to inefficient outcome with nodes idle for some time – usually not too bad if each reduce node does many keys, but something to consider.

Overview of the Execution of a MapReduce Program

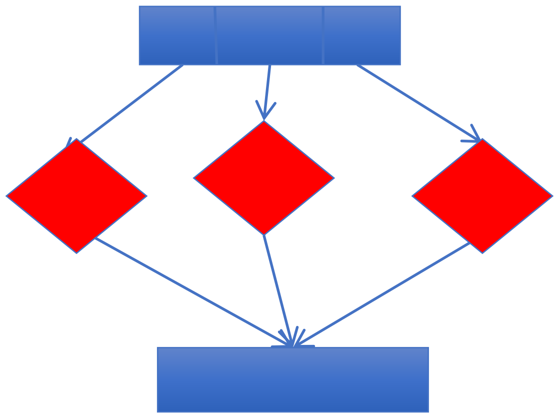
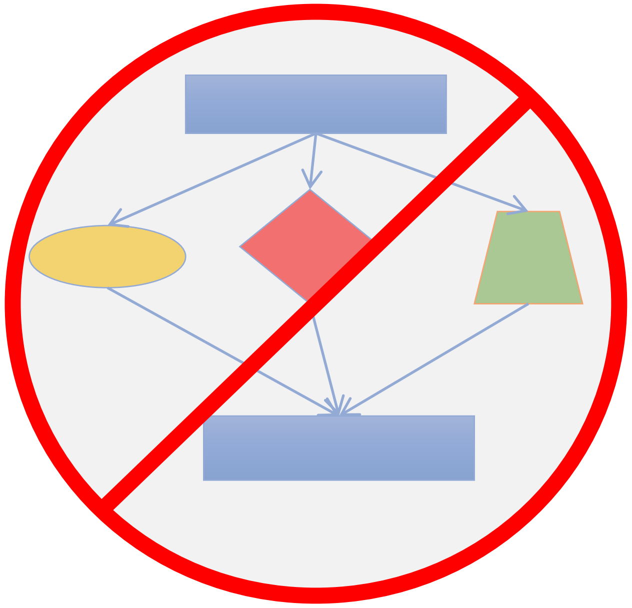


Coping with Node Failures

- If the compute node at which the Master is executing fails, the entire MapReduce job must be restarted.
- Other failures managed by the Master.
- Master periodically *pings* Worker processes to check they are still alive.
- If a Map worker fails, all of the Map tasks assigned to this worker must be redone. The Master reschedules them for a new worker and informs the Reduce tasks where to look for the location of its input
- If a Reduce worker fails, the Master simply reschedules its currently executing Reduce tasks on another reduce worker.

When NOT to use MapReduce

- Don’t use MapReduce for processes which involve relatively little calculation and/or which change the database e.g., interacting with a product data-base
- MapReduce is good for *data-parallelism* i.e., when the same task needs to be done repeatedly to lots of data
- It is not good for *task-parallelism* i.e., when lots of different tasks need to be done to the same data

Matrix-vector multiplication by MapReduce

- Suppose we have an *n x n* matrix **M**, whose element in row *i* and column *j* is denoted m_{ij}.
- Suppose we also have a vector **v** of length *n*, whose *j*th element is v_j
- Then the matrix-vector product is the vector **x** of length *n*, whose *i*th element x_i is given by:

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

How can we carry out these calculations using the MapReduce paradigm?

Matrix-vector multiplication via MapReduce

- Map function:
  - applied to each individual matrix element m_{ij}
  - yield the key-value pair $(i, m_{ij}v_j)$

```
def matrixvector_mapper(i,j,m,v):
  yield(i,m[i][j]*v[j])
```

- Reduce function:
  - sums all of the values associated with a given key
  - yields a pair (i, x_i)

```
def matrixvector_reducer(key,valuelist):
     yield(key,sum(valuelist))
```

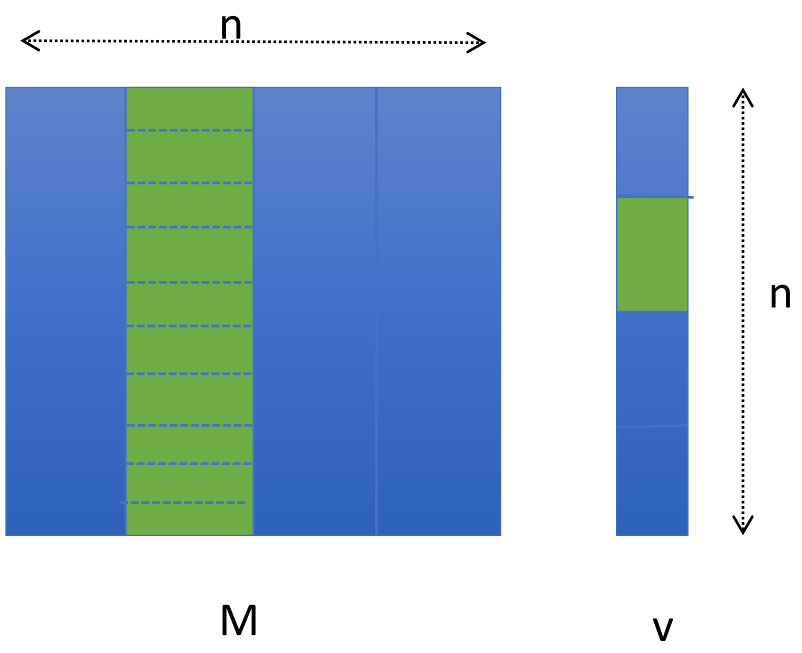
Matrix-vector Multiplication via MapReduce

- In practice, a map worker will be carrying out a number of map function applications
- Each map worker will operate on a chunk of the matrix M (so M in its entirety does not have to be accessed by any node)
- Assuming the vector v can be read into main memory, it will be read in by the map worker and then made available to all applications of the map function.

Really large vectors

If v does not fit into main memory, divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height

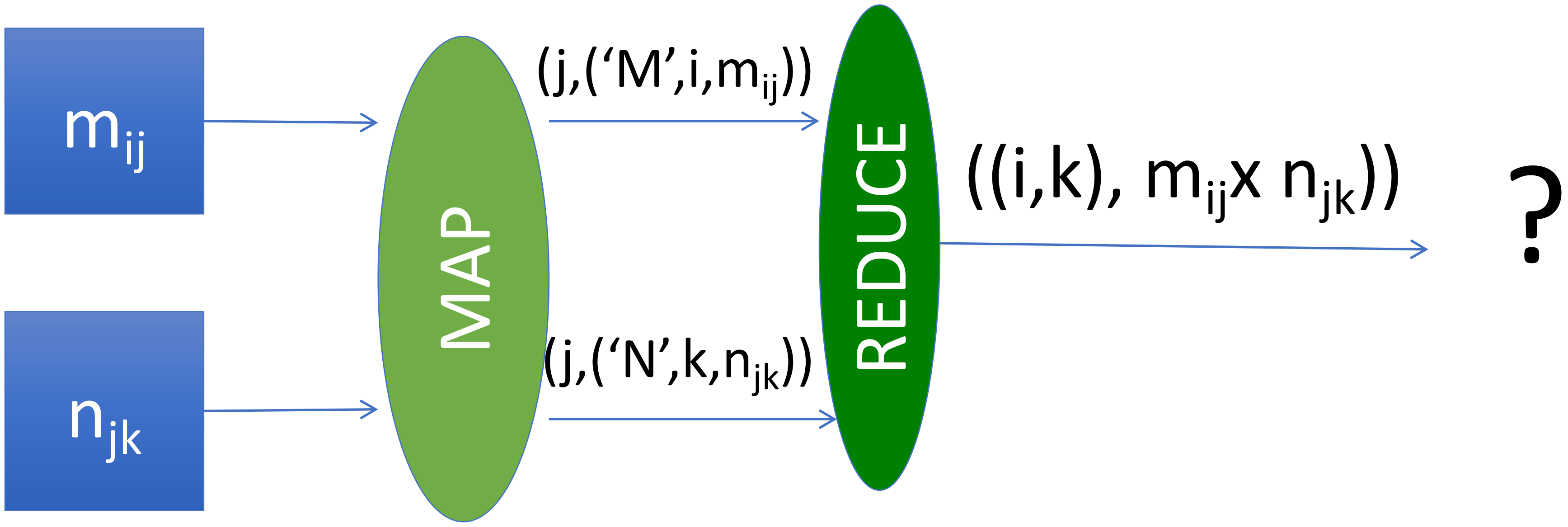
Each map task is assigned a chunk from one of the stripes of the matrix and gets the corresponding stripe from the vector



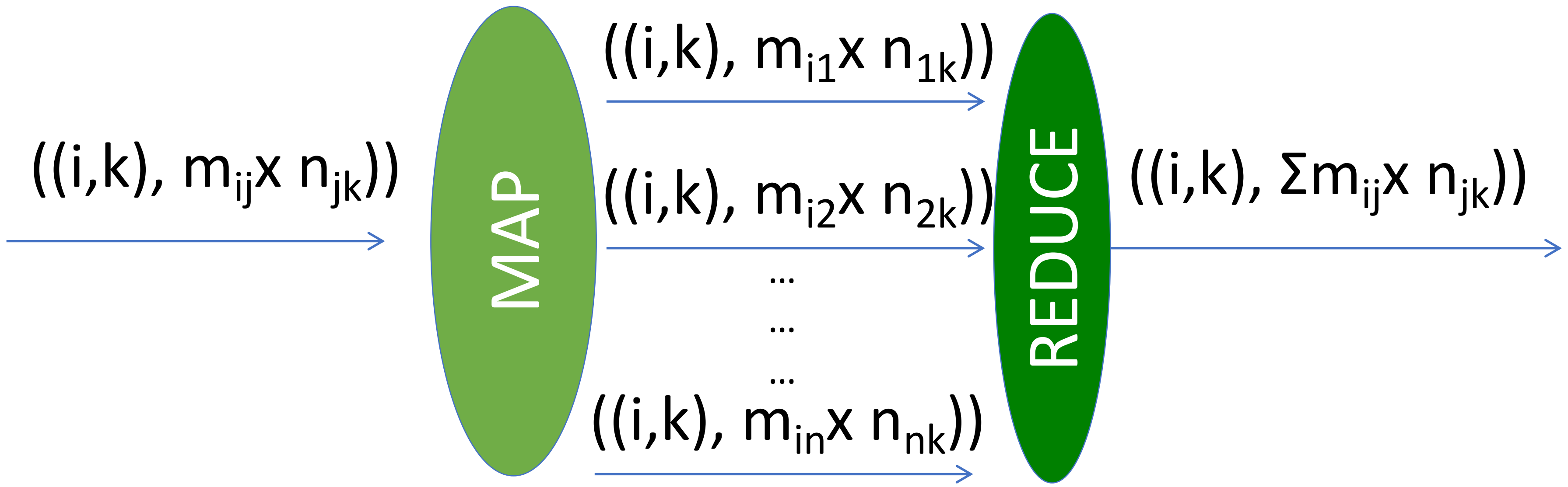
Matrix multiplication using MapReduce

- P=MN =>

$$p_{ik} = \sum_j m_{ij} n_{jk}$$



MapReduce Again!



$((i,k), m_{ij} x\ n_{jk}))$
MAP
$((i,k), m_{i1} x\ n_{1k}))$
$((i,k), m_{i2} x\ n_{2k}))$
…
…
…
$((i,k), m_{in} x\ n_{nk}))$
REDUCE
$((i,k), \Sigma m_{ij} x\ n_{jk}))$

A second round of MapReduce!

- map is the identity function
- reduce sums the list of values associated with a key

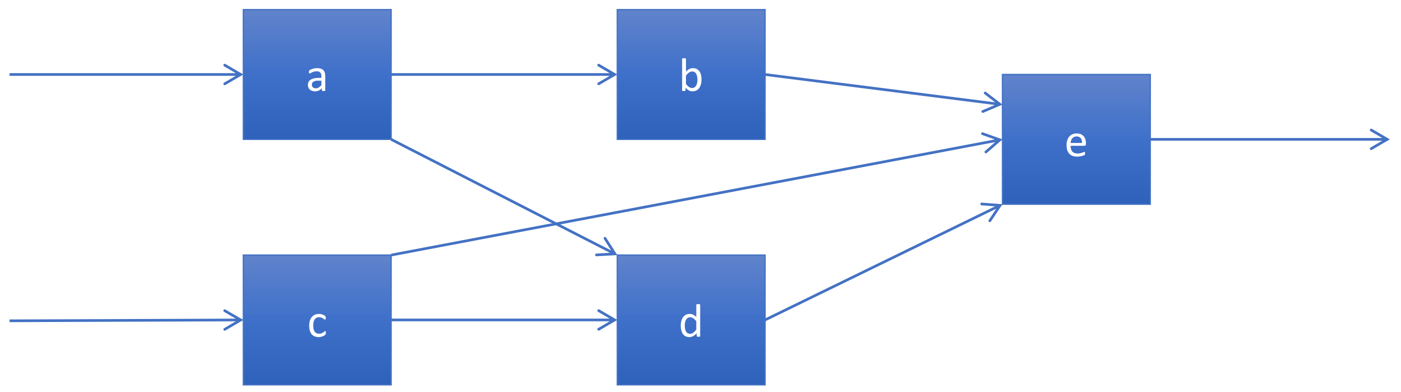
In essence: **First map gets data into convenient structure**
First reduce does products
(Second map does nothing)
Second reduce does sums.

Matrix Multiplication in One Pass

- More work could be done in each function
- The Map function:-
    - For each element of **M**, produce all of the key-value pairs ((i,k),('M',j,m_{ij})) for k=1,2,....
    - For each element of **N**, produce all of the key-value pairs ((i,k),('N',j,n_{jk})) for i = 1,2, ...
- The Reduce function:-
    - Each key (i,k) has a list for each of M and N and for all j
    - Sort by j (for each of M and N)
    - For corresponding items on each list, multiply m_{ij} and n_{jk} and then sum results

Extending MapReduce

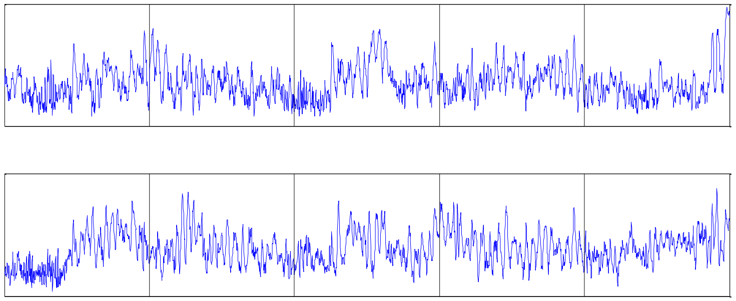
- Workflow systems (e.g., Clustera and Hyracks)



- a collection of functions with an **acyclic** flow graph
- each function can be executed by many tasks
- output is only delivered on completion so it is easy to restart failed tasks

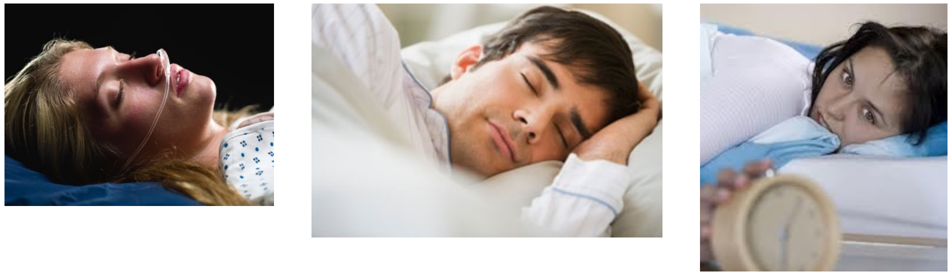
Basic parallelism

E.g., In an experiment, what if you have loads of data samples, from loads of participants, each in a few different scenarios?



Neuroscience experiment:
EEG data from 7 participants
4 states of consciousness during anaesthesia
~500 2s segments for each participant/state

Task: compute some stuff on each 2s segment



Utilise all your cores/compute nodes!
Send each one a bunch of 2s segments!

Communication Cost Model

- Imagine that an algorithm is implemented by an acyclic network of tasks e.g., Map tasks feeding Reduce tasks
- The *communication cost* of a task is the size of the input to that task
- The *communication cost of an algorithm* is the sum of the communication costs of all of the tasks
- Communication cost tends to dominate the perceived efficiency of an algorithm

Example: It turns out that 2-pass matrix multiplication usually has smaller communication cost than 1-pass matrix multiplication (advanced further reading - Leskovec Chapter 2).

Wall-clock time

- this is the time it takes for a parallel algorithm to complete
- whilst communication cost is important in deciding the best algorithm, do not forget wall-clock time
- minimising communication cost would mean assigning all work to one task
- this would not result in a good wall-clock time!