COMP6237 Data Mining

Big Data Processing

Jonathon Hare
jsh2@ecs.soton.ac.uk

What is “Big Data”?

"**Big data** is an evolving term that describes any voluminous amount of structured, semi-structured and unstructured **data** that has the potential to be mined for information"

–Margaret Rouse, WhatIs.com

“**Big data** is a buzzword, or catch phrase, used to describe a massive volume of both structured and unstructured **data** that is so **large** that it's difficult to process using traditional database and software techniques.”

–Vangie Beal, webopedia.com

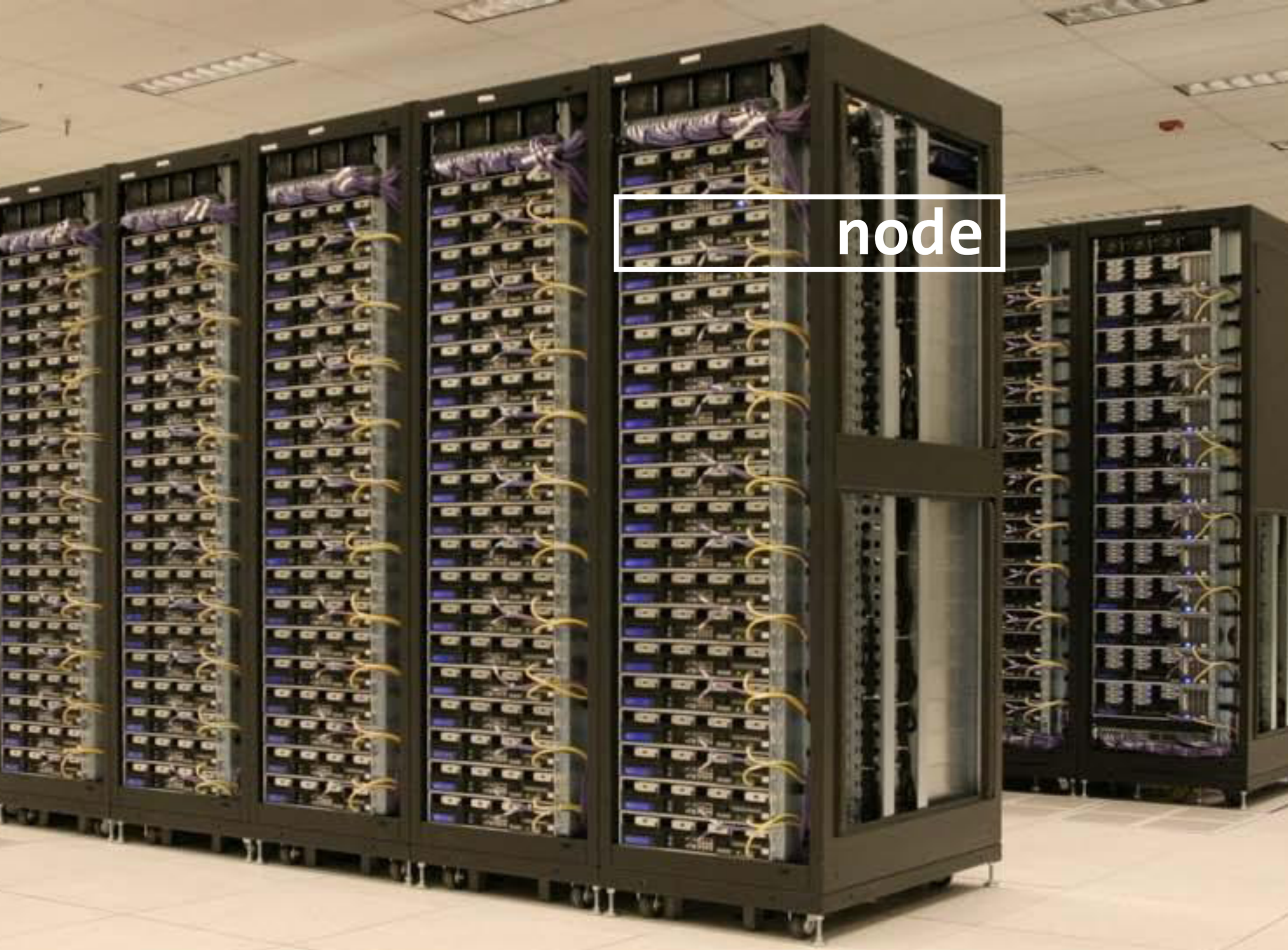
For this lecture, we'll consider big data to be:

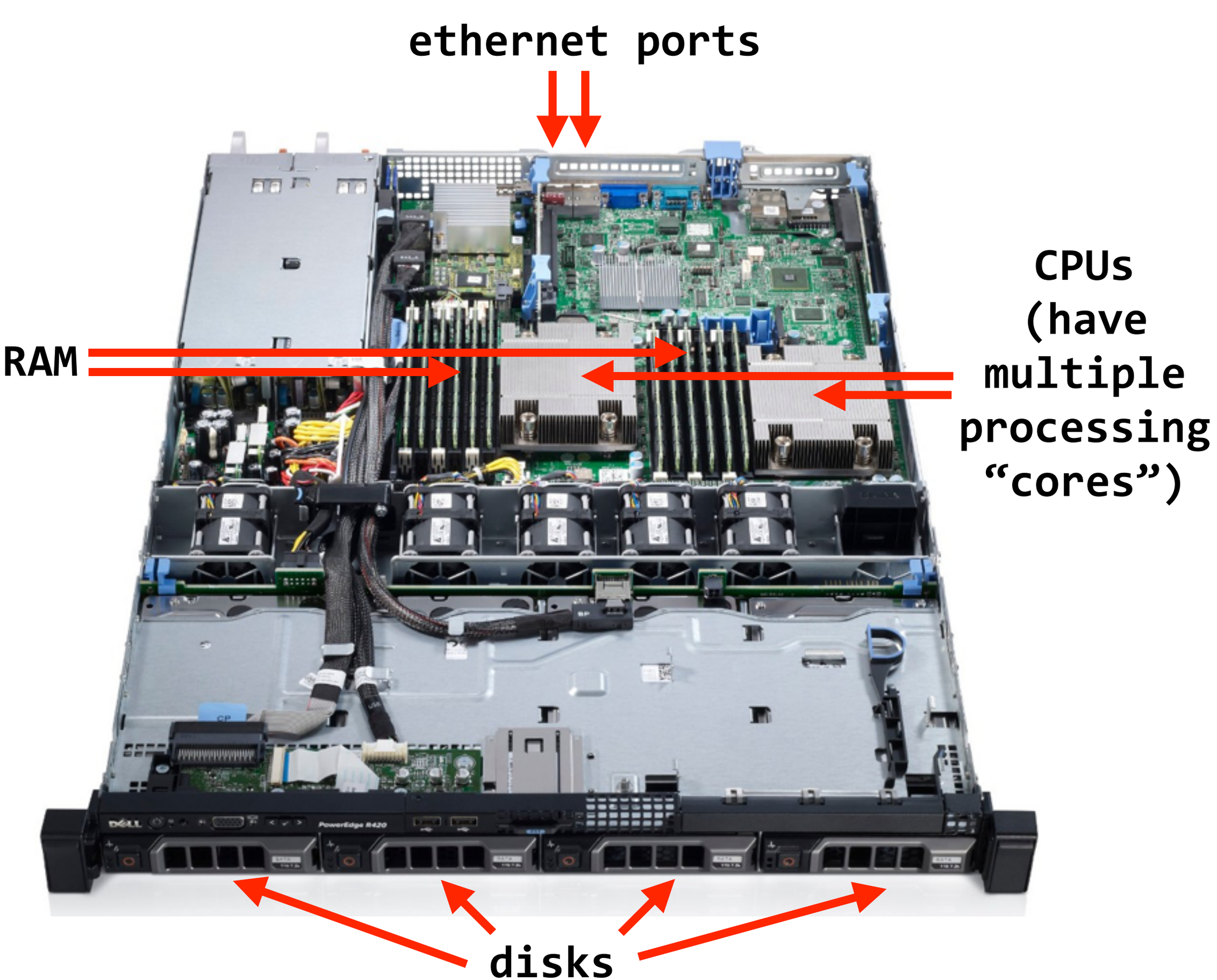
"data that is too large to be effectively stored on a single machine or processed in a time-efficient manner by a single machine."

Distributed Computation

cluster


rack

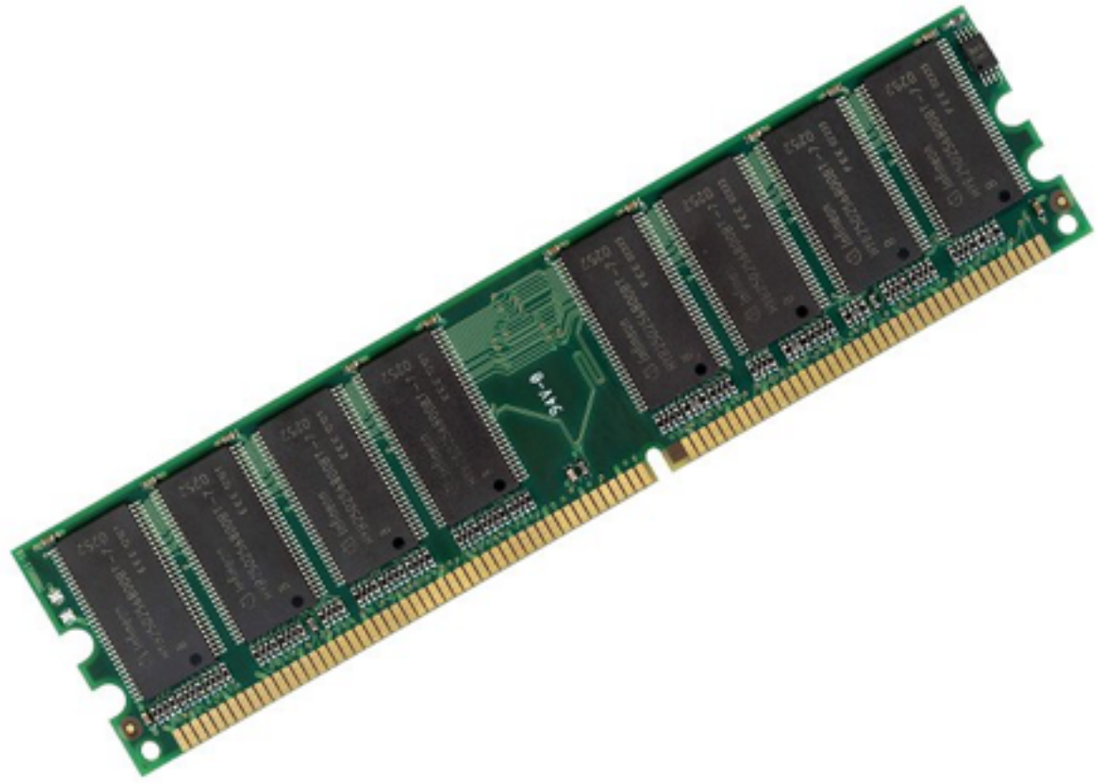

node


ethernet ports
RAM
CPUs (have multiple processing “cores”)
disks

The challenge of dealing with big data

- Mechanical bulk storage is relatively cheap in £££ terms
    - Solid state storage is more expensive and has less capacity

- But… all current bulk storage is **very slow** relative to the rate the CPU can process data from RAM
    - When processing “big” data we need to get the data to the processor quickly



Peak transfer rate: 17066MB/s

Peak transfer rate: 140MB/s
Storage capacity up to 6TB (or even 8TB)

Peak transfer rate: >600MB/s
Storage capacity up to 2TB

Peak transfer rate: 2900MB/s
Storage capacity up to ~4TB
very expensive!!

- Actual disk transfer speeds are much slower than those peak speeds quoted
  - Especially if you are:
    - Reading and writing at the same time
    - Dealing with small files
- To improve performance:
  - Perform bulk reads and writes (avoid small files, use larger blocks)
  - Attach multiple drives to separate busses on the same machine

Gigabit Ethernet

Peak transfer rate: 125MB/s

Infiniband

Peak transfer rate: >600MB/s

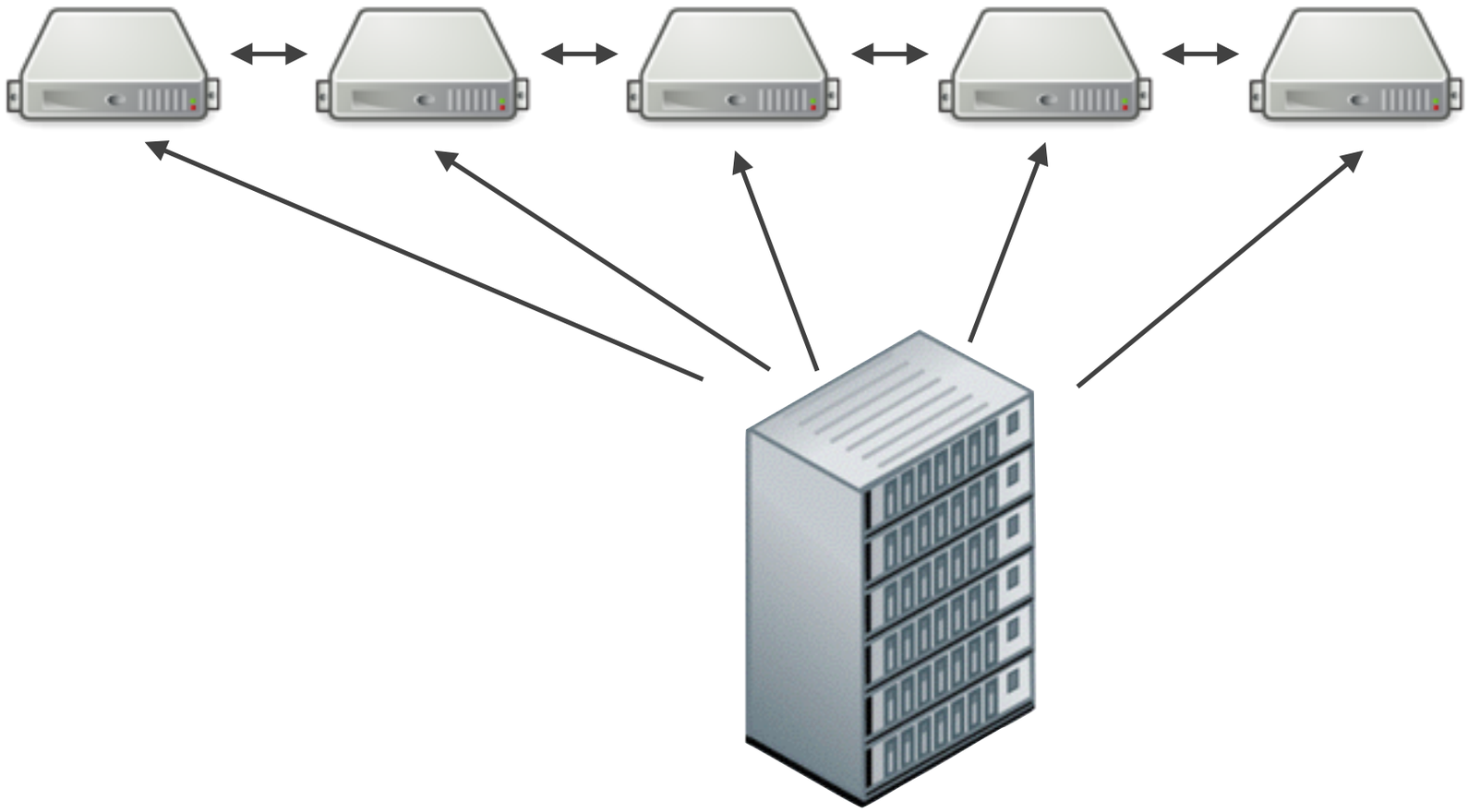
very expensive and limited by the slowest disks

Take-away message:

moving data between machines is bad for processing performance

Distributed Data

Traditional distributed computing **pulls** data to the processors over the network:



limited local storage on each node

Obviously if the data is “big” there is a problem of bandwidth

Distributed Data

Modern approaches distribute the data across the nodes, and **push** computation to the nodes:

Each node has a large amount of local storage

Ideally, data moving between nodes is limited.

Distributed Data

- Data can be distributed in different ways:
  - **Horizontally partition** the data (“sharding”)
    - Disjoint subsets of records of the dataset are held on different machines.
    - Popular in database systems
- **Distributed File Systems…**

Distributed File Systems

- Recap of the basics:
  - A **logical file** on a file system is backed by **physical blocks** on the storage media.
    - The file system is responsible for tracking which blocks belong to a file, together with the order of those blocks.
- On a standard (i.e. desktop) file system, blocks are usually between 512B and 4KB in size.

Distributed File Systems

- In a Distributed File System (DFS) the physical blocks belonging to a single logical file are spread across multiple drives and multiple machines in a cluster.
    - DFS blocks are usually much larger - i.e. 64MB
    - Assumption is that the files being stored are themselves much **larger than the block size**.
        - You explicitly want to avoid small files… merge dataset records into a “container” file.
- Allows more **efficient data transfer** between nodes.
- **Fewer blocks** to keep track of.

Distributed File Systems

- Some DFS implementations ensure redundancy through **block-level replication**.
    - Each block is stored more than once in **different drives and nodes**.
        - If a single drive or node fails, the data is still intact.
- **Rack-awareness** ensures block replicas span different physical racks.
    - If a networking or power fault takes out a rack, the file is still accessible.

MapReduce

Motivation

- **Efficient distributed “push” computing over distributed data**
  - …on commodity hardware
    - Not expensive networking interconnects (ethernet).
  - Cheap (consumer) hard-drives.
  - Deal with failure through redundancy (rack-aware block-level replication)

The MapReduce Programming Model

Datatypes and flow

- The entire programming model is based on the idea of records being representable by a *key-value* pair
    - The key and value can be any kind of object however, and often the key is ignored (“Null” keys)

Map

- The Map function takes in a key-value pair from the data being processed and outputs zero or more key-value pairs of data

  - Output key and value **types** can be different to the input key and value types.

Map(k1, v1) → list(k2, v2)

Shuffle

- The shuffle phase (automatically performed by the framework) collates all the Map function outputs by their key, producing a list of <key, list(values)>, sorted by key:

list(k2, v2) → list(k2, list(v2))

Reduce

- The Reduce function is handed each <key, list(values)> pair and produces zero or more values as a result:

Reduce(k2, list(v2)) → list(v3)

Advantages

- Easily parallelised:
  - Map functions run independently
  - Reduce functions run independently

M/R Programming Model

- The M/R programming model allows for efficient distributed computing

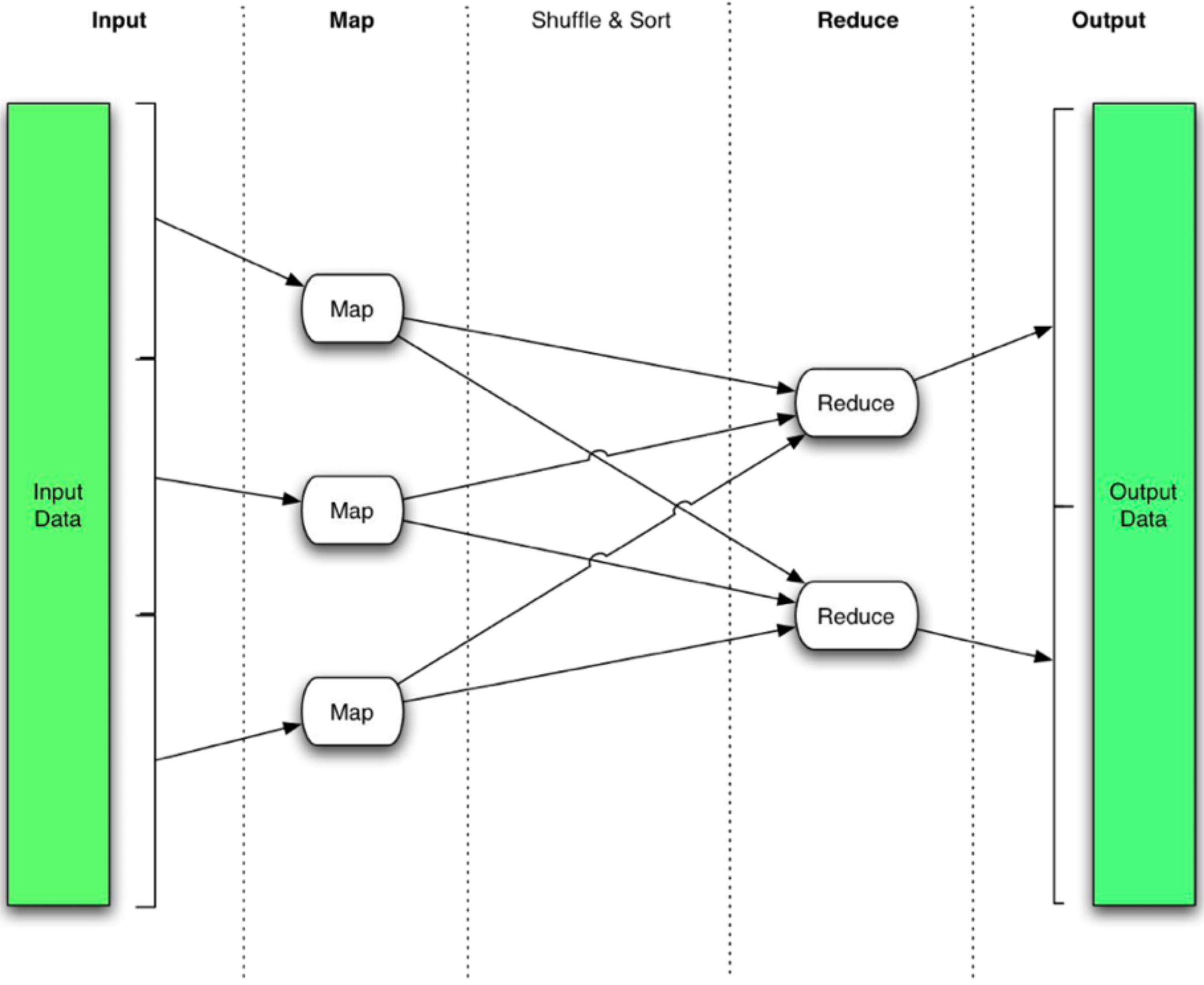
- Works like a unix pipeline:

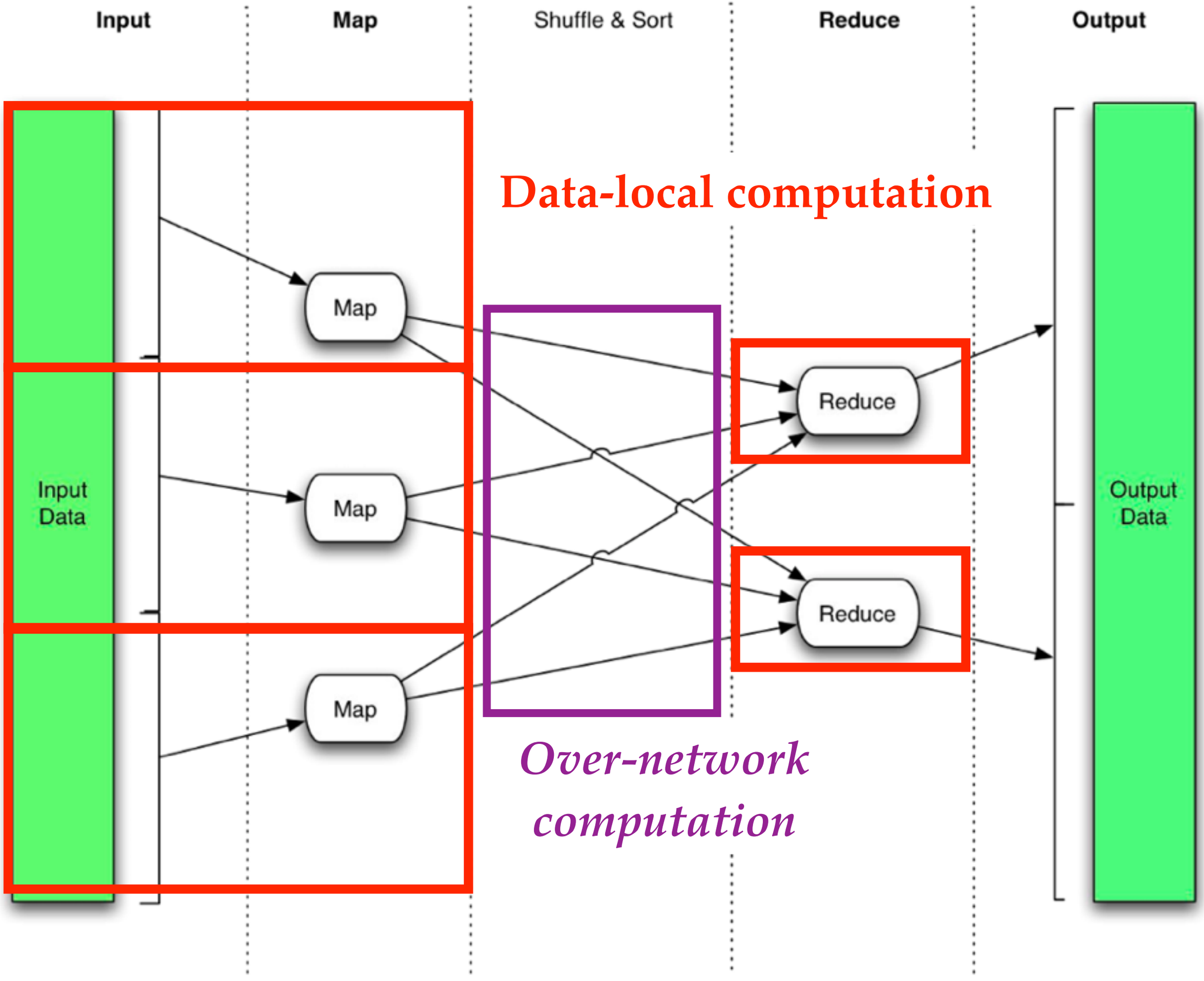
```
cat input | grep | sort              | uniq -c  | cat > output
Input | Map  | shuffle & sort | Reduce   | Output
```

- You provide Map and Reduce functions; the framework does the rest.

  - Real programs sometimes chain together rounds of M-R and sometimes omit the reduce (and shuffle/sort) altogether.

MapReduce and distributed data


Input
Map
Shuffle & Sort
Reduce
Output
Input Data
Map
Map
Map
Reduce
Reduce
Output Data


Input
Map
Shuffle & Sort
Reduce
Output
Data-local computation
Input Data
Map
Map
Map
Reduce
Reduce
Output Data
Over-network computation

MapReduce Word Count

Synopsis: read corpus of text files and counts how often words occur.

Input: collection of text files, read line-by line.

Output: count of number of times each word occurs across corpus.

Map function:

Input: <*Key:* line number, *Value:* line of text>

Outputs: <*Key:* word, *Value:* 1>

Takes a line as input and breaks it into words by splitting on whitespace (or similar). Then, for each word, emit a <key,value> pair with the word and 1.

Reduce function:

Input: <*Key:* Word, *Value:* [list of 1's corresponding to each occurrence]>

Outputs: <*Key:* word, *Value:* count>

Each reducer sums the counts for each word and emits a single <key,value> with the word and sum.

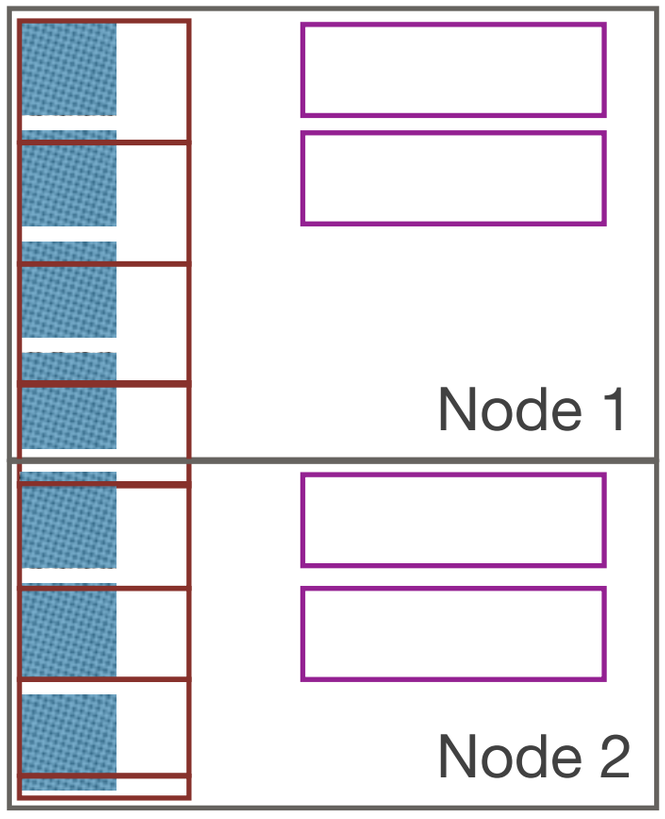

hadoop

- **Hadoop** is an open-source cluster-computing framework with MapReduce and DFS implementations.
  - Initiated by Doug Cutting and Mike Cafarella at Yahoo! as part of the Nutch web search engine, and based on ideas from the Google MapReduce and GFS (Google File System – Google's internal DFS implementation).
  - Now maintained/developed by the Apache Foundation.

- (Mostly) written in **Java**.
- **Hadoop MapReduce** provides the distributed MapReduce framework
  - Most Hadoop MapReduce programs are written in JVM languages (**Java, Scala**, Groovy, etc), but it is possible to use **any language** through “Hadoop Streaming”.
- A number of filesystems can be used, but Hadoop is distributed with **HDFS** (Hadoop DFS).
  - Supports: rack-awareness, block replication, high-availability (no single point of failure), etc.
  - HDFS allows the MapReduce scheduler to be data aware, and thus minimise network traffic.

Hadoop Architecture

InputSplit
(logical partition of a file on a record boundary)



HDFS Block
(physical partition of a file)

Mappers
(process one InputSplit by applying the Map function to each record; Number of concurrent Mappers set by number of available CPU cores and number of InputSplits that need to be processed)

Reducers
(not shown; apply the Reduce function to the sorted and collated Map output; number of reducers is user defined)

Disadvantages of MapReduce

- Standard MapReduce programs follow an acyclic dataflow paradigm – i.e. a stateless Map followed by a stateless Reduce.
    - This doesn't mesh well with **machine learning** algorithms, which are often iterative.
    - You can't for example rely on the result of the previous iteration, if the iteration is defined by a single Map invocation.
        - No computing of Fibonacci!
- It's designed for batch data, rather than streaming or real-time data.

- Frameworks like Hadoop have some features that can help (to an extent):
    - Ability to **save extra data** from the Mapper, that doesn't automatically get sent to the reducer (although can be loaded there).
    - **Shared memory** across Maps from a single InputSplit within a Mapper and Reduces in a Reducer.
        - But not across Mappers or Reducers.
    - Ability to write **flexible chains** (map-reduce-map-reduce…; map-map-map-reduce…), and even skip the reduce all-together.

- But there is still an overhead if you are passing over the same data again and again.

- Other solutions like **Apache Spark** (which is built on Apache Hadoop, but doesn't use the MapReduce component) work around this by having nodes load the data (locally) once, and then cache it in memory for the iterations.

Apache Spark

Spark Architecture

- Spark is built around the concept of distributed datasets (i.e. data stored on HDFS across a Hadoop cluster).
  - Known as **RDD**s (Resilient Distributed Datasets).
    - RDDs can be transparently cached in-memory.
      - Useful if you're re-using the data (perhaps iterating over it multiple times).
- An "RDD partition" corresponds to a subset of records.
  - In the case of an RDD backed by file on HDFS, those are the records in an InputSplit.

Spark Programs

- Spark programs perform **operations** on **RDD**s.
  - You automatically get **data-local computation** where possible.
  - Wide range of operations in two classes:
    - **Transformations:** these take an RDD and return a **new RDD**.
      e.g.: map, filter, groupByKey, reduceByKey, aggregateByKey...
    - **Actions:** these take an RDD and return a **result value**.
      e.g.: reduce, collect, count, first, take, countByKey, foreach…
- Operations can be assembled in complex, unconstrained ways, unlike MapReduce!
  - A Spark program is basically a **Directed Acyclic Graph** of operations applied to RDD(s).

Summary

- When dealing with Big Data, it's the speed of data transfer to the CPU that is the limitation.
    - Distributing the data, and **pushing computation to the data** helps solve this problem.
- The **MapReduce** model is one approach to doing this; **Spark** is another.
    - Next lecture we'll see how they can be applied to data mining.