COMP9033 DATA ANALYTICS

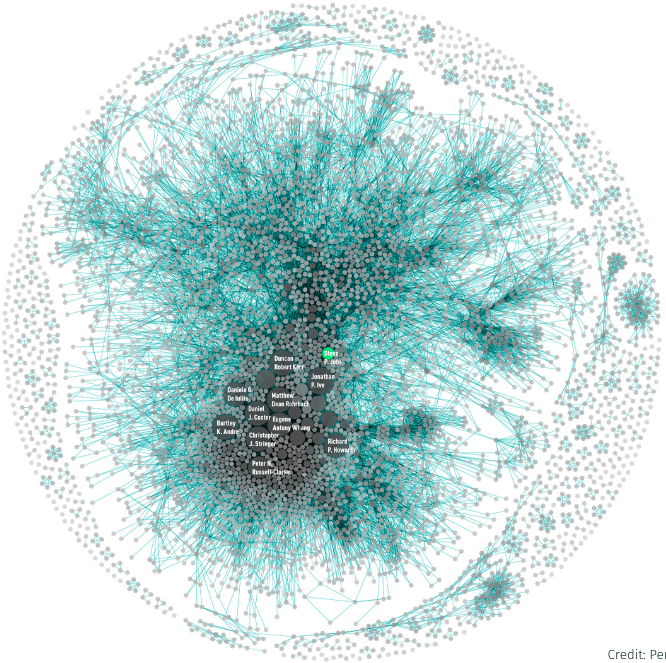
11/12

BATCH DATA ANALYSIS

DR. DONAGH HORGAN

DEPARTMENT OF COMPUTER SCIENCE
CORK INSTITUTE OF TECHNOLOGY

2018.04.24



Credit: Periscopic

Overview

1. Sets and probability:
    - Set notation.
    - Set cardinality.
    - Probability.
    - Conditional probability.
    - Examples.
2. Association rule mining:
    - What it is.
    - Terminology.
    - Support, confidence and lift.
    - Brute force mining.
    - Efficient rule mining.

0.2 / THIS WEEK

1. Big data:
    - The three Vs definition.
    - Why it’s relevant now.
2. High volume data:
    - Why it’s a problem.
    - Cluster computing.
    - Distributed file systems.
3. MapReduce:
    - Cluster architecture.
    - How it works.
    - Examples.
4. Batch data frameworks:
    - Mahout.
    - TensorFlow.
    - Spark.

Big data

1.1 / BIG DATA

- Over the past few weeks, we have covered a variety of topics in the area of data analysis, but we have yet to consider *big data analysis*. Why?
- Big data is specialised:
    - Data analysis is a set of *general* tools for solving *general* problems.
    - Big data analysis is a set of *specialised* tools for solving *specific* problems.
    - Big data analysis problems are a subset of general data analysis problems — we need to learn to walk before we can run!
- Big data is hard:
    - Big data is a complex topic and comes with an additional layer of problems that must be solved.
    - Many “small data” solutions don’t translate (*e.g*. due to performance).
    - Generally, we have to adjust, adapt or re-architect our approach.
    - In some cases, we *must* resort to heuristic (*i.e*. suboptimal) approaches.

1.2 / WHAT IS BIG DATA?

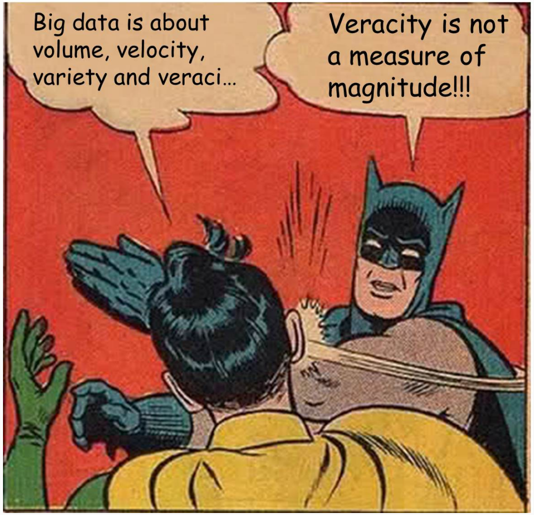
- Big data problems are encountered when analysing data at large scales.
- It is broadly (though not universally) agreed that there are three varieties:
  1. Volume.
  2. Velocity.
  3. Variety.
- *High volume* describes situations where the quantity of data to be analysed is so large that conventional storage and/or processing techniques cannot be used, *e.g*. due to hard disk, memory or CPU limitations.
- There is no specific threshold for the amount of data that constitutes a high volume problem, *e.g*. what we consider high volume today may not be in ten years due to Moore's Law.

1.3 / WHAT IS BIG DATA?

- *High velocity* describes situations where the rate at which data is produced is so fast that conventional processing systems cannot analyse the data in a reasonable amount of time.
- Increasingly, data is produced at faster rates and from more sources (*e.g.* stock markets, mobile devices, IoT), but we still require results quickly.
- *High variety* describes situations where there are a vast number of forms of data to be analysed: structured (*e.g.* SQL), semi-structured (*e.g.* CSV, JSON, XML) and, increasingly, unstructured (*e.g.* tweets, images, audio).
- In order to cope, we must build more robust systems, capable of extracting information from a plethora of data sources.

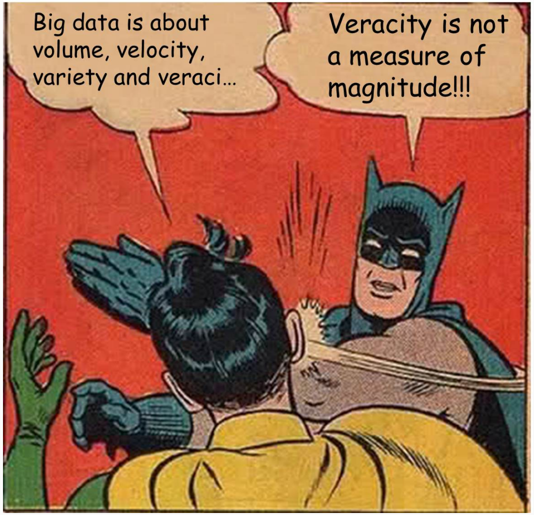
1.4 / WHAT IS BIG DATA?

- Volume, velocity and variety are known as *the three Vs*, and are the most commonly accepted definitions of big data.
- Other properties have also been proposed, but are controversial and not widely accepted.
- Generally, this is because the properties apply to *both* small data and big data problems, *i.e.* they do not become problems due to data scale alone.


Big data is about volume, velocity, variety and veraci...
Veracity is not a measure of magnitude!!!

Credit: Doug Laney

- One example is *veracity*, a property that defines the uncertainty or ambiguity of data.
- However, imprecise data is a problem at *all* scales of data magnitude, *e.g.*
  - Typos during manual data entry.
  - Faulty sensors.
  - Intentional misreporting.


Big data is about volume, velocity, variety and veraci...
Veracity is not a measure of magnitude!!!

Credit: Doug Laney

1.6 / WHY IS BIG DATA A PROBLEM NOW?

- The rate at which data is being produced is accelerating:
  - Data producing devices (*e.g*. mobile, wearable, IoT) are growing in number.
  - Monitoring tools and storage are becoming increasingly more affordable.
  - Data analysis tools are becoming more mature, incentivising the capture of more metrics and key performance indicators.
- It's *estimated* that:
  - 2.5 exabytes of data are produced every day (1 exabyte = 1 billion gigabytes!).
  - About 75% of this data is unstructured.
  - The current growth in data *already* exceeds Moore's Law, *i.e*. conventional technology will become less effective at dealing with big data in the future.
  - The Internet of Things (IoT) will bring somewhere between thirty and two hundred billion devices online in the coming decade.
- Big data *is* a problem now and *will* be a problem in future!

1.7 / HIGH VOLUME DATA

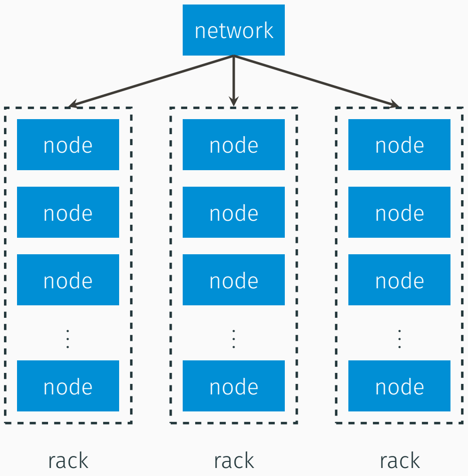
- High volume data problems occur when the amount of data becomes so large that conventional storage and processing systems begin to fail:
    - Data becomes too large to hold in memory.
    - Data becomes too large to store on disk.
    - The CPU time required to compute even trivial operations becomes so large that it is not feasible to execute tasks in a reasonable amount of time.
- There are two solutions to this problem:
    1. Scale vertically: build more powerful computers, which are capable of handling the additional demand.
    2. Scale horizontally: use cluster computing to distribute the workload over many standard/commodity computers.

1.8 / HIGH VOLUME DATA

- Until very recently, building bigger computers was a popular solution:
    - Relatively few problems required massive amounts of resources.
    - The high cost of purchasing a supercomputer was generally offset by the benefits of solving the problem.
- However, the rate at which data is currently being produced *exceeds* Moore's Law, which means that
    - An increasing number of problems require massive compute power to solve.
    - The cost of sufficiently powerful supercomputers is increasing.
    - The useful lifespan of supercomputers is decreasing.
- Clusters of compute nodes built from commodity hardware are generally less expensive than supercomputers and can be expanded in an ad hoc manner.
- However, conventional solutions (*e.g*. data storage, algorithms) must be adapted before they can operate on a clustered architecture.

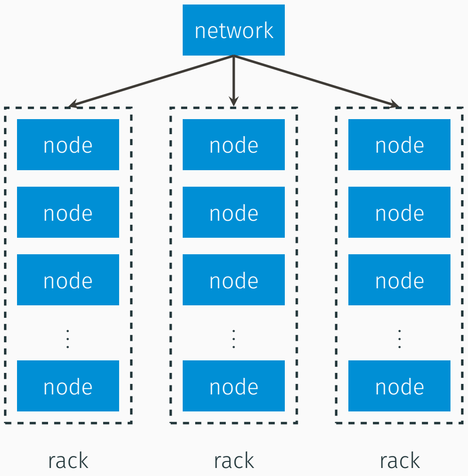
1.9 / CLUSTER COMPUTING

- A cluster computing environment usually consists of three distinct components:
    1. Nodes.
    2. Racks.
    3. Networks.
- Nodes are simply commodity computers, with standard compute, memory and storage capabilities.



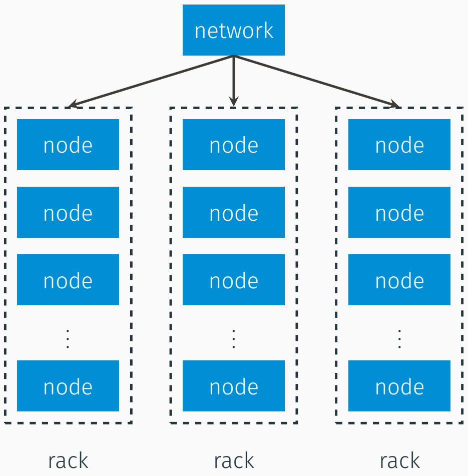
1.10 / CLUSTER COMPUTING

- Racks are groups of nodes connected together by a local network.
- Groups of racks are then connected together by a wider area network, which may in turn connect to one or more other wider area networks.
- Typically, the network bandwidth across racks is smaller than the network bandwidth inside racks.

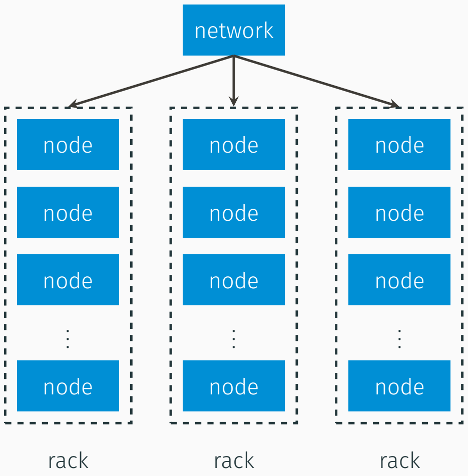


1.11 / CLUSTER COMPUTING

- Failure can be catastrophic in a clustered environment:
    - If a node fails, then its data is lost and any computation it was involved with may have to be aborted.
    - If a rack fails, then effectively all its nodes fail.
    - If a network fails, then effectively all its racks fail.



- In practice, nodes, racks and networks can (and do) fail, so we must design with failure in mind.
- Generally, the problem is tackled in a two stage manner:
    1. Distributed file systems are used to mitigate against data loss.
    2. Distributed compute frameworks are used to mitigate against computation disruption.



1.13 / DISTRIBUTED FILE SYSTEMS

- To mitigate against data loss due to node, rack or network failure, data is typically stored on a *distributed file system* (DFS).
- Many varieties of distributed file system have been proposed and/or implemented, *e.g.*
    - Amazon S3.
    - Ceph.
    - GlusterFS.
    - Google File System (GFS).
    - Hadoop distributed file system (HDFS).
    - Microsoft DFS.
- In general, different implementations have different design goal priorities, *e.g.* fault tolerance, data parallelism, read/write throughput, scalability.

1.14 / THE HADOOP DISTRIBUTED FILE SYSTEM

- The *Hadoop distributed file system* (HDFS) is often used in batch data analysis applications.
- This is because the design of HDFS makes a number of assumptions about the kind of data being stored in the cluster, *i.e*.
    1. Files are so large that it is not feasible to store them on a single node.
    2. Files are written once and read many (WORM) times.
    3. When files are read, they are read in whole.
- These assumptions map well to many high volume data problems, *i.e*.
    - There is a massive quantity of data to be analysed, usually more than can be stored on a single computer.
    - Data is typically accumulated over time, and so is often stored in an append-only fashion.
    - Usually, the entire data history is analysed, *i.e*. data is processed in batch.

MapReduce

2.1 / MAPREDUCE

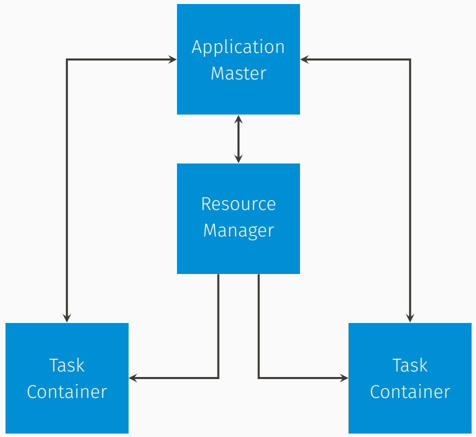
- MapReduce is a distributed computation paradigm for processing large, infrequently updated quantities of data on clustered computers:
    - Large computations are broken down into small, isolated tasks.
    - These tasks are assigned to available worker nodes.
    - The outputs of the individual tasks are merged to form the final result.
- Google are credited with inventing MapReduce, and have been granted a patent on it, but several implementations are available.
- The most well known freely available implementation is Hadoop MapReduce, which operates on HDFS.
- While Hadoop MapReduce is written in Java, MapReduce as a concept is not language specific.

- MapReduce is used to solve a variety of high volume data problems in many different industries[1], *e.g.*
    - Amazon Elastic MapReduce.
    - Comcast (network management).
    - EBay (search optimisation).
    - Facebook, LinkedIn, Spotify (recommendations/analytics).
    - Google PageRank.
    - Hulu, Etsy, Rackspace (server log analysis).
    - Salesforce (customer behaviour prediction).
    - Telefonica (user modelling).
    - Twitter (processing tweets and logs).
    - Yahoo (research).

[1]A more exhaustive list of Hadoop MapReduce uses is available at bit.ly/2opdJWp.

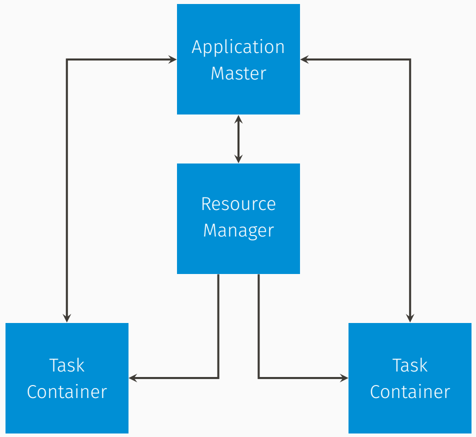
2.3 / HADOOP MAPREDUCE ARCHITECTURE

- In Hadoop 2, MapReduce jobs are coordinated by a *Resource Manager*:
    - Schedules jobs according to available CPU, memory and disk space on worker nodes.
    - Knows where nodes are (*i.e.* rack aware).
- Typically, the Resource Manager is colocated with the cluster's HDFS NameNode on the cluster master.



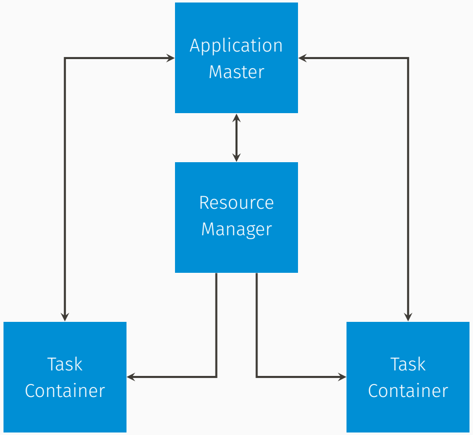
2.4 / HADOOP MAPREDUCE ARCHITECTURE

- When a MapReduce job is run, the Resource Manager spawns an *Application Master*, which manages the state of the MapReduce job throughout its lifespan.
- The Application Master is responsible for negotiating compute resources from the Resource Manager that are “local” to the data required for the job.
- Usually the Application Master is located on a worker node, similar to the HDFS DataNode.



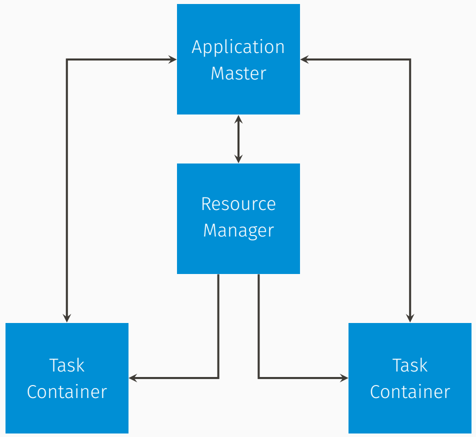
2.5 / HADOOP MAPREDUCE ARCHITECTURE

- When the Application Master requests access to compute resources from the Resource Manager, the Resource Manager allocates a number of *task containers*, *i.e.* bounded compute resources on one or more worker nodes.
- The Application Master then uses the task containers to execute the map (and shuffle) and reduce tasks.



2.6 / HADOOP MAPREDUCE ARCHITECTURE

- If a task fails, the Application Master will attempt to rerun it on another node.
- If the Application Master fails, the Resource Manager will restart the MapReduce job.
- Depending on the cluster configuration, it may be possible to recover the outputs of previously completed tasks, avoiding the re-execution of all tasks.
- If the Resource Manager fails, then the MapReduce cluster will fail (though high availability configurations are possible).



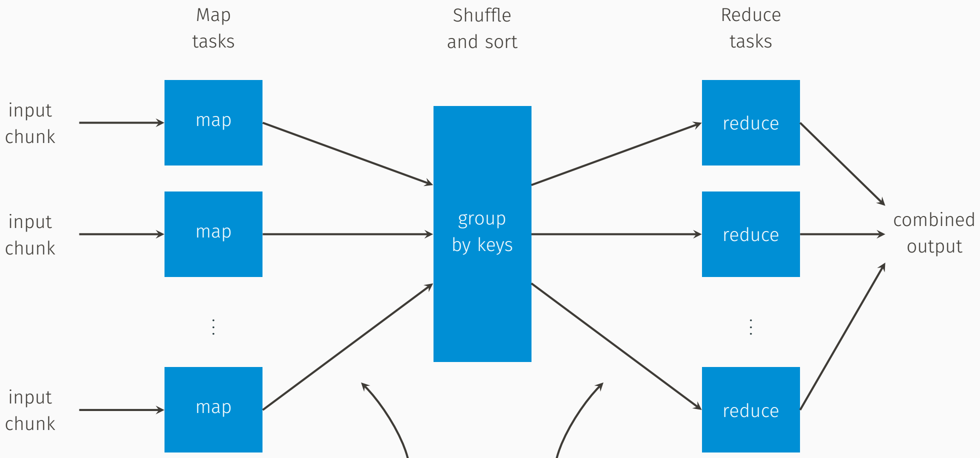
2.7 / HOW MAPREDUCE WORKS

- MapReduce works by decomposing computations into two distinct phases:
  - A mapping phase, where chunks of the input data are processed independently of one another (*i.e.* in parallel).
  - A reduction phase, where the results of the mapping phase are combined to formulate the final result (can also be run in parallel).
- The operations performed in the mapping and reduction phases are known as *map* and *reduce* tasks, respectively.
- Both map and reduce tasks *must* be stateless functions, so that if a task fails on one compute node, it can readily be reassigned to another without aborting the entire computation.
- Decomposing computations into map and reduce tasks can be a complicated procedure — MapReduce algorithms often don't resemble their "small" data equivalents.

2.8 / HOW MAPREDUCE WORKS

- A typical MapReduce job works as follows:
    1. A batch data processing job is decomposed into an operation consisting of map and reduce tasks *only*.
    2. The locations of the data chunks to be processed are identified (usually the data is stored on a DFS).
    3. Map tasks are assigned for each data chunk, and are then scheduled to be executed by a number of compute nodes “close” to those chunks (to reduce network I/O).
    4. The mappers process the assigned chunks of data and convert them into lists of key-value pairs, according to some user-defined function.
    5. The key-value pairs resulting from all of the map tasks are grouped by their keys.
    6. One or more reduce tasks are assigned to process the grouped key-values lists.
    7. Reduce tasks are assigned to a number of compute nodes, each one processing all of the values associated with a given key, according to some user-defined operation.
    8. The outputs of all of the reduce tasks are combined to produce the final result.

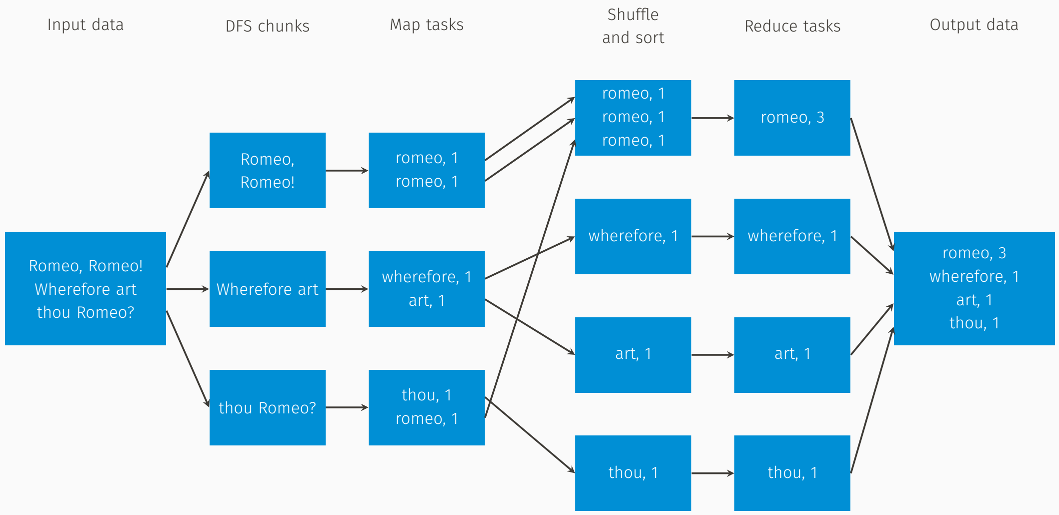
2.9 / HOW MAPREDUCE WORKS



Map task outputs are lists of key-value pairs, *e.g.* $[(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)]$

Reduce task inputs are key-values lists, *e.g.* $(k_1, [v_1, v_2, \ldots, v_m])$

2.10 / EXAMPLE: WORD COUNTING



2.11 / CLUSTER SIZE

- For maximum parallelism, we could build a cluster with enough nodes, so that all of the reduce tasks run simultaneously.
- However, this is rarely done in practice.
- The number of unique keys produced by a set of mapping operations (and, therefore, the number of reduce tasks required) is typically very large, and so it would be costly to build a cluster with enough capacity to run them all in parallel.
- Also, the time required to complete a reduce operation can depend on its inputs, and so some reduce tasks will finish faster than others — it would be an inefficient use of resources to have a large number of inactive nodes.

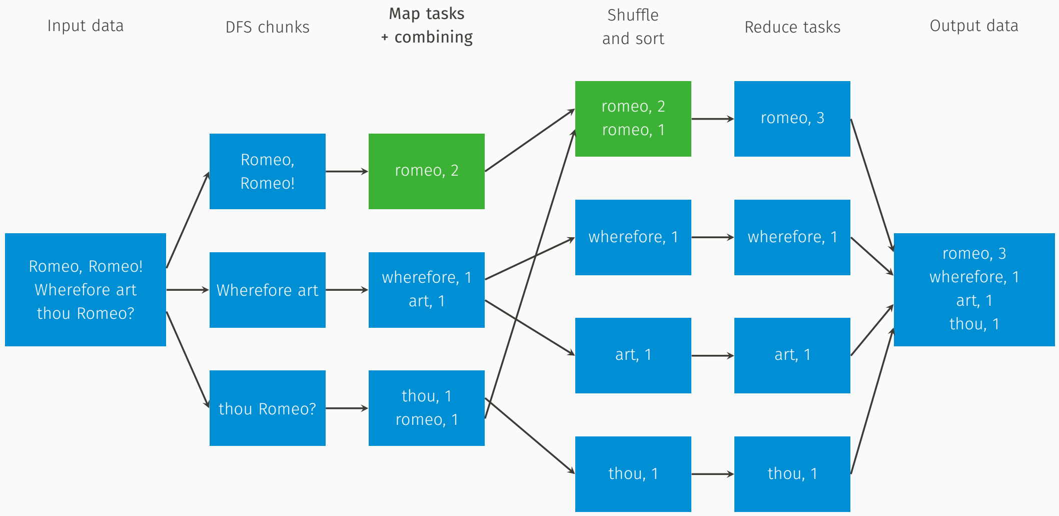
2.12 / CLUSTER PERFORMANCE

- Generally, map and reduce tasks are designed so that they do not consume large amounts of resources:
    - Usually, nodes are commodity hardware and so are not designed to execute complex tasks quickly.
    - Instead, data and task parallelism take care of computational complexity.
    - If individual tasks to take a long time to complete, node failures may significantly increase the overall compute time.
- In practice, the biggest bottleneck in the system is the limited bandwidth between compute nodes:
    - If map tasks produce large numbers of key-value pairs, we are forced to transfer large amounts of data across the DFS in order for the reduce tasks to produce the result.
    - The time required to transfer the data can delay the computation significantly.

2.13 / COMBINING

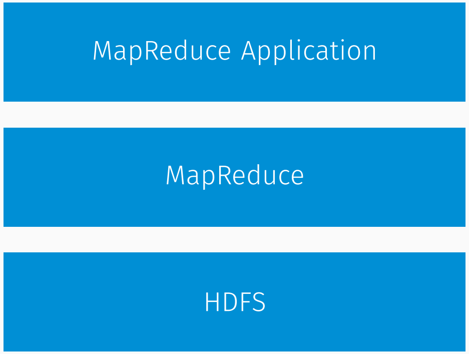
- One technique that mitigate against this bottleneck is *combining*:
    - Move some reduction logic to the mapping phase to reduce the amount of data produced by each map task.
    - If each map task produces less data, then we don't have to transfer as many key-value pairs across the DFS.
- However, we can't combine map and reduce tasks in every situation:
    - Combination only works where the reduce operation is both *associative* and *commutative*, *i.e.* values can be processed in any order and the result will always be the same.
    - Associativity and commutativity are not general properties of computations, and so we can only use combination to decrease execution time in a limited subset of problems.

2.14 / EXAMPLE: WORD COUNTING

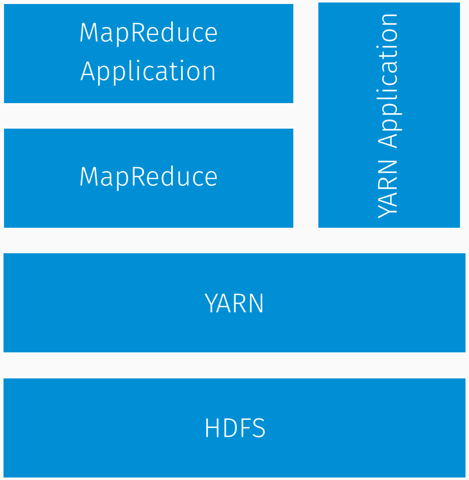


Batch data frameworks

- In the first release of Hadoop, the MapReduce layer operated directly on HDFS (see opposite).
- This architecture imposed a number of constraints on the kinds of applications that could be developed:
  - As Hadoop MapReduce was a Java application, all child applications had to be written in Java.
  - The Hadoop MapReduce API was complex, which increased both code complexity and application development time.



- This architecture was significantly redesigned in Hadoop 2.
- The *Yet Another Resource Negotiator* (YARN) resource manager was added as an abstraction layer between HDFS and HDFS applications.
- YARN enables applications to be built directly on top of HDFS, while also allowing applications to be built on top of MapReduce.



3.3 / BATCH DATA FRAMEWORKS

- In recent years, there has been a trend towards using frameworks which further abstract these layers, making it simpler and faster to develop new applications, *e.g.*
  - Apache Flink.
  - Apache Giraph.
  - Apache Mahout.
  - Apache PredictionIO.
  - Apache Spark.
  - Apache SystemML.
  - H20.
  - TensorFlow.

3.4 / APACHE MAHOUT

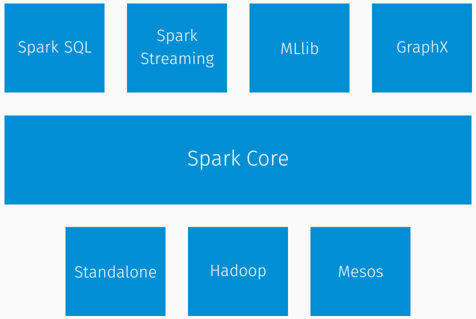
- Apache Mahout is an open source library of scalable, distributed machine learning algorithms.
- It supports a number of different techniques, *e.g*.
    - Classification: Naive Bayes.
    - Clustering: K means.
    - Collaborative filtering: user-based and item-based.
    - Dimensionality reduction: PCA.
- Initially, Mahout implemented its algorithms directly on top of MapReduce.
- However, in 2015, there was a shift in focus towards providing scalable algorithms in general, not just for MapReduce.
- Mahout has since deprecated most of its MapReduce algorithms in favour of Flink/H20/Spark implementations.

3.5 / TENSORFLOW

- TensorFlow is an open source library of machine learning algorithms, originally developed by Google.
- TensorFlow supports many machine learning algorithms:
    - Regression: linear regression, logistic regression, neural networks.
    - Classification: neural networks.
    - Clustering: K means.
- Like Spark, TensorFlow supports a number of different languages, including C++, Go, Java and Python.
- It also supports distributed computation using Hadoop.

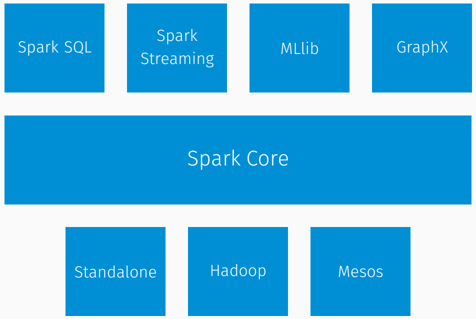
3.6 / APACHE SPARK

- Apache Spark is an open source software framework for processing both batch *and* streaming data in a compute cluster.
- Spark can be run in a number of different cluster configurations:
    1. Standalone, using Spark's built-in cluster manager.
    2. Hadoop, via YARN.
    3. Apache Mesos.



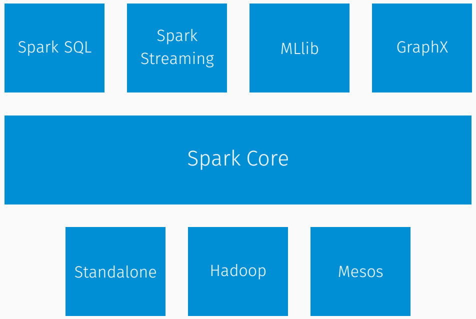
3.7 / APACHE SPARK

- Spark's functionality is provided by four libraries:
    1. Spark SQL: Query data using SQL in Spark applications.
    2. Spark Streaming: Data stream processing support.
    3. MLlib: Distributed machine learning algorithms for batch and streamed data.
    4. GraphX: Graph data analysis.



3.8 / APACHE SPARK

- The Spark API is available in several languages:
  - Java.
  - Python.
  - R.
  - Scala.
- Spark applications can be developed with code from one or more of these languages, enabling rapid development of experimental and enterprise software, side by side.



3.9 / APACHE SPARK

- Because the Spark API is available in many languages, we aren’t limited to writing MapReduce jobs in Java (like in Hadoop).
- For instance, the code to the right implements a MapReduce job for counting words in a file on a HDFS in Python.
- For context, the equivalent code in Java is 60 lines without line breaks and comments!

```python
import pyspark

# Create a Spark context object
sc = pyspark.SparkContext(master='local[*]')

# Load a text file from HDFS
text_file = sc.textFile('hdfs://...')

# Execute a MapReduce job
text_file.flatMap(lambda line: line.split()) \
         .map(lambda word: (word, 1)) \
         .reduceByKey(lambda x, y: x + y)
```

- Spark offers significant performance benefits over using MapReduce directly:
  - Where possible, Spark holds compute task results in-memory to speed up the next stage of calculation.
  - In contrast, MapReduce always reads from HDFS before running a task and writes to HDFS at the completion of a task.
- If no memory is available to store the results of a particular task, Spark defaults to writing them to disk, and so in a worst case scenario it is still roughly comparable to MapReduce.
- In 2014, Spark won the Daytona GraySort Benchmark Contest, beating MapReduce’s previous winning score:
  - Hadoop sorted 102.5 TB of data in 72 minutes, using 2100 nodes.
  - Spark sorted 100 TB of data in 23 minutes, using just 206 nodes.

Summary

X.1 / SUMMARY

- Big data is a big problem, but only in some cases:
    - We still need small data analytics for lots of problems.
    - Big data is much harder, need to learn the basics then adapt.
- Using DFS + distributed compute cluster is a good solution for high volume data processing:
    - MapReduce is useful, but there can be a steep learning curve.
    - However, newer frameworks are making it easier to build complex distributed applications.
    - Need to pick the right tool for the job!
- Lab work:
    - Create a standalone Spark cluster.
    - Run a MapReduce job in Spark to analyse some data.
- Next week: streaming data analysis and data ethics!

1. Ullman et al. *Mining of Massive Data Sets.* Cambridge University Press, 2014. (stanford.io/1qtgAYh)
2. Landset et al. *A survey of open source tools for machine learning with big data in the Hadoop ecosystem.* Journal of Big Data, 2(1), p.24. (bit.ly/2oHv3RG)