Big Data Analytics

COMP336 / COMP529

Dom Richards

University of Liverpool

October 16, 2024

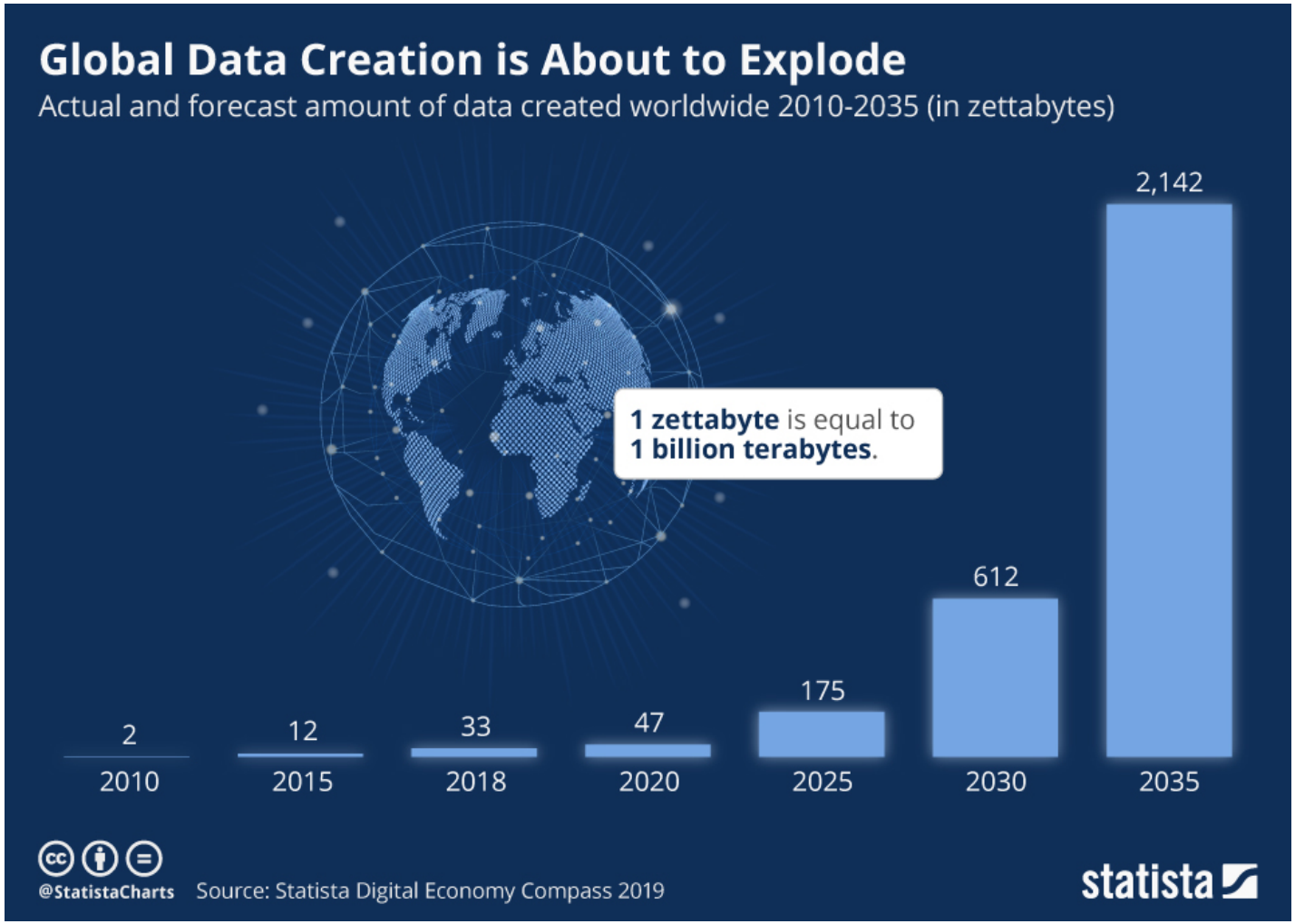
Table of Contents

Introduction

Introduction

- Welcome to Big Data Analytics!
- Over the next 12 weeks, we'll explore the field of big data
- It's concerned with the storage and analysis of datasets that are too large to fit on a computer, and hence typically require a *data center* to store and process
- 'Big' datasets (also referred to as big data) are often very valuable, and they crops up everywhere: written text; social media posts; financial transaction records; industrial sensors; scientific research data
- By the end of this course, you'll have a solid understanding of big data technologies and how to apply them to real-world problems
- We'll introduce a range of techniques to extract value from big datasets including: supervised and unsupervised machine learning; statistical analysis; natural language processing; network science; signal processing

How Common is Big Data?



Source Statista report on global data creation forecasts.

Example Datasets

- Search Engine Data: googlebot
- Social Media Data: Posts, comments, and reactions on platforms like Facebook and Twitter
- E-commerce Data: Online shopping behaviours, product views, and purchases
- Healthcare Data: Patient records, genetic sequences, and medical device data
- IoT Data: Sensor data from connected devices like smart homes and wearables

Understanding Big Datasets

- In the next two sections, we'll explore the characteristics of big datasets and the challenges they present
- In this section, we'll discuss the 6 Vs of big data: volume; velocity; variety; veracity; value; and variability
- In the next section, we'll take a look at how big data is represented for the purpose of analysis

Characteristics of Big Data

- Big data comes in many shapes and sizes, but it can be characterised in terms of “6 Vs”:
  - Volume: the amount of data in a given dataset
  - Velocity: Data can be generated at a range of possible speeds
  - Variety: Data comes in different formats, including structured, semi-structured, and unstructured
  - Veracity: Data quality and accuracy can vary
  - Value: which derives from the ability to transform big data into actionable insights
  - Variability: Data can be inconsistent or change rapidly

Volume

- Volume refers to the quantity of data
- Ranges from GBs upwards

Velocity

- Velocity refers to the speed at which data is being generated, processed, and analysed
- For some big data applications, data arrives continuously from multiple sources in real time
- Example: Credit card transactions need to be validated quickly to prevent fraud
- Rapid decision-making is crucial in fields like:
  - Stock trading platforms, where delays can result in losses
  - Autonomous vehicles, which rely on sensor data being processed in real-time

Variety

- Variety refers to the wide range of data types and formats
- Traditional systems only handled structured data, typically found in relational databases (e.g., SQL)
- Today, data comes in a variety of formats:
  - Structured data: Databases, spreadsheets, CSV, Parquet
  - Unstructured data: Emails, videos, social media posts, images

Veracity

- Veracity refers to the accuracy, quality, and trustworthiness of data
- Big data often includes noisy, incomplete, or inconsistent data
- Example challenges:
  - Social media data may include misleading or biased information
  - Sensor data can contain errors due to faulty devices
- Techniques to improve veracity:
  - Data cleaning: Removing inaccuracies or noise
  - Data validation: Ensuring data is correct before analysis

Value

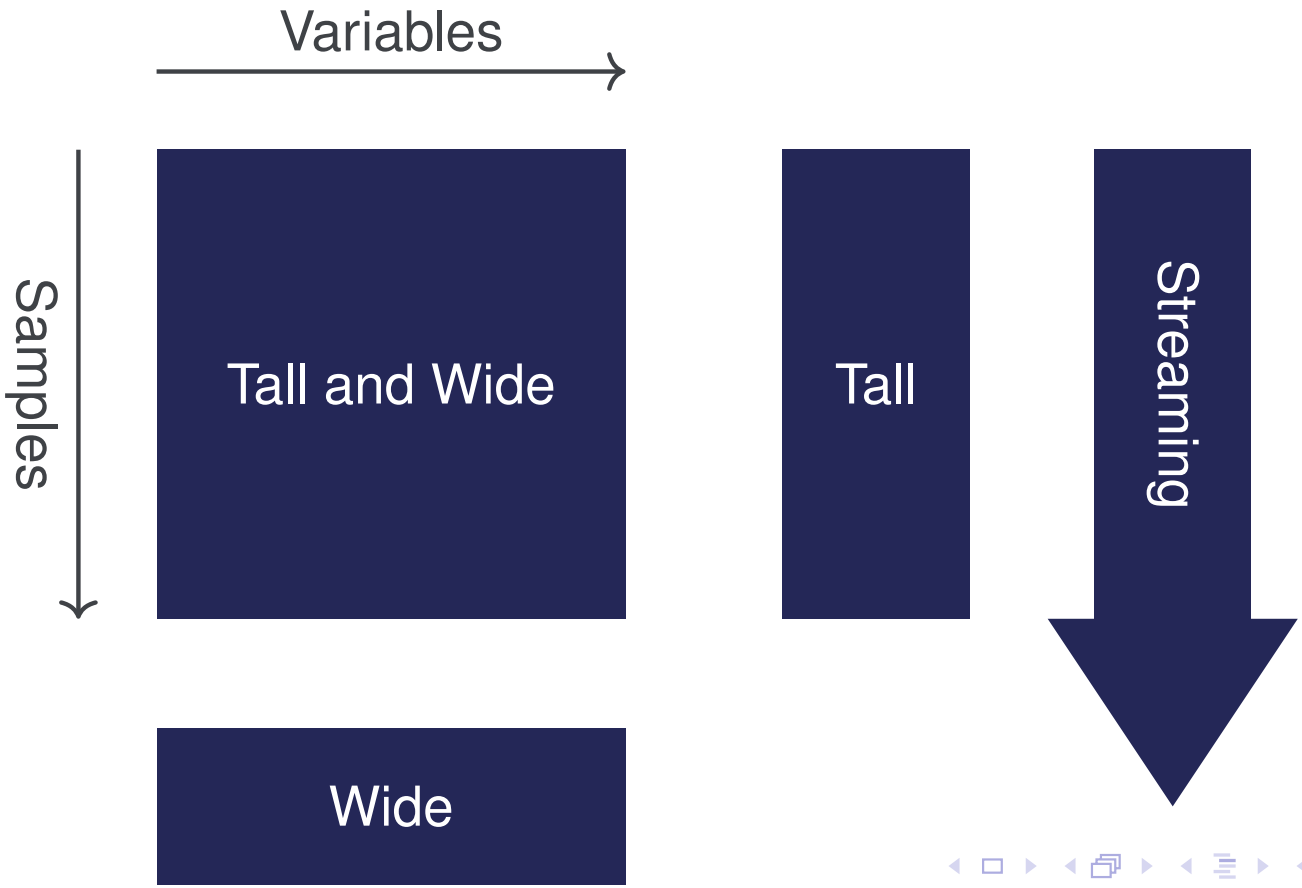
- Value refers to the insights and actionable information extracted from big data
- Big data is often valuable if it can be transformed into meaningful insights
- In this course, we'll see a range of techniques to extract value from big data, including: machine learning; statistical analysis; natural language processing; network science; signal processing.

Variability

- Variability refers to the changing nature of data and the inconsistency in its flow
- Data can be generated at unpredictable rates and can vary in its meaning and relevance
- E.g. regime switching:
  - Social media sentiment analysis can shift rapidly based on trending topics or events
  - IoT sensor data might fluctuate based on environmental changes or technical issues
- Handling variability requires adaptive algorithms and real-time monitoring to respond quickly to data changes

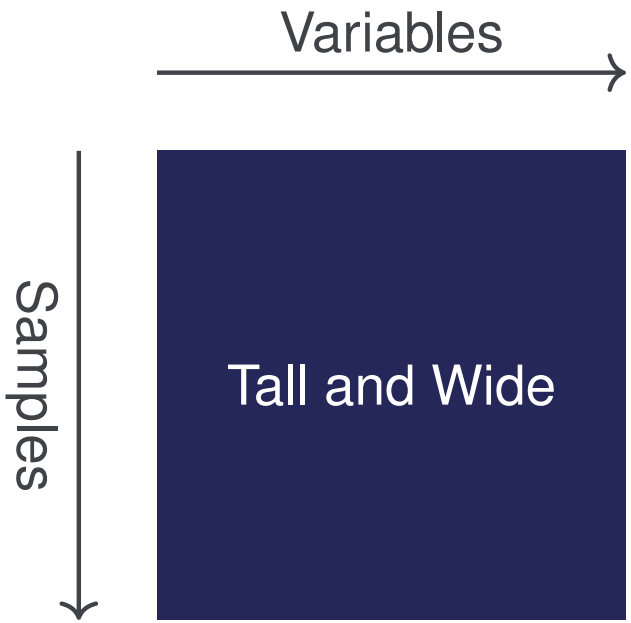
Dataset Types

Big datasets come in various shapes and sizes. We tend to think about matrices of variables and samples, even when the data isn't originally presented to us like this (e.g. text documents or images)



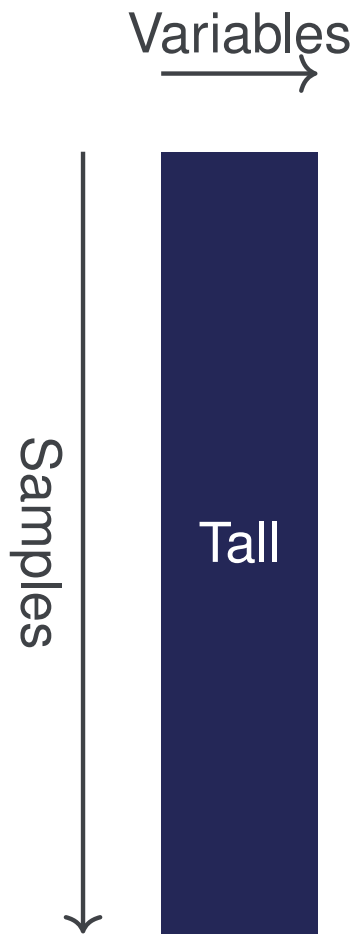
Tall and Wide Datasets

- These datasets can be very large:
  - Millions to billions of variables and samples
- Typical example is networks:
  - Social networks, contagion networks, etc.



Tall Datasets

- Hundreds to thousands of variables
- Thousands to millions of samples
- Often suitable for machine learning
- Examples:
  - NLP (Natural Language Processing)
  - Image databases
  - Scientific computing (e.g., solutions of fluid dynamics equations)
  - Reinforcement learning

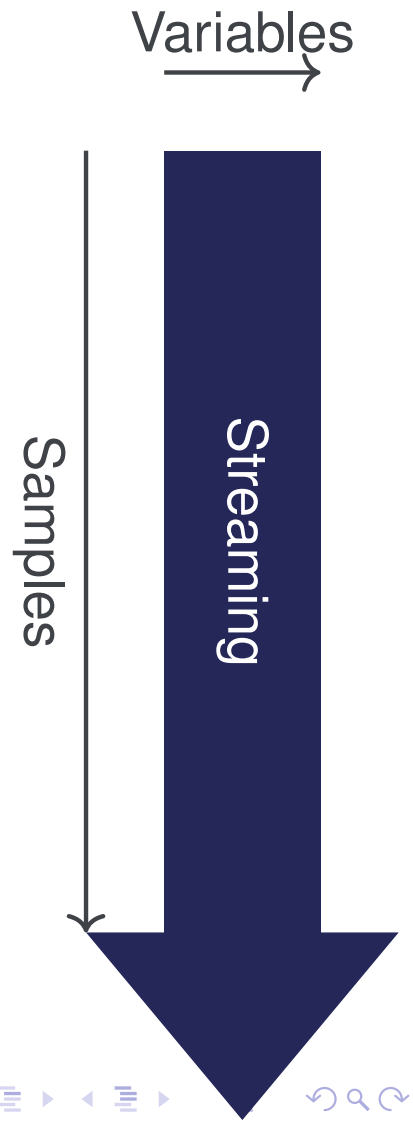


Wide Datasets

- Thousands to millions of variables
- Hundreds to thousands of samples
- Variables $\gg$ samples
- Examples:
  - Genomics datasets
  - Document modelling

Streaming Datasets

- Samples continually created in real time
- Examples:
  - Sensor data
  - Social media feeds
  - Financial transactions



Dense vs Sparse Matrices

- Data matrices can be dense or sparse
- Dense matrix:
  - Most matrix elements are non-zero
  - All elements are stored, including the zeroes
  - Memory usage is proportional to the total number of elements ($variables \times samples$)
- Sparse matrix:
  - Majority of elements are zero
  - Only non-zero elements are stored, often as a coordinate list, which stores triplets (row, column, value)
  - Memory usage is proportional to the number of non-zero elements

Course Overview

2. Course Overview
   - Big Data Middleware
   - Network Science
   - Unsupervised Learning
   - Supervised Learning
   - Natural Language Processing
   - Bayesian Statistics and Filters

Working with Big Data

- In weeks 2, 3 and 11, you'll learn about big data middleware
- Datasets in big data are typically too large to be processed on a single computer, so software has been developed to process such datasets across multiple computers in a data centre
- This software is called *big data middleware*
- It is installed onto a cluster of compute nodes in a data centre, and allows users to interact with the entire cluster as if it were a single computer
  - Storage is distributed across the cluster, and the user is presented with a single file system
  - Processing is distributed across the cluster, allowing the user to operate on all of the data by writing a single program which is executed on all the cluster nodes in parallel
- You'll learn about the two most popular open source big data middleware frameworks: Hadoop and Spark

Hadoop

- Next week we'll see Hadoop.
- Hadoop was the first major open-source big data middleware framework
- It was based on Google's MapReduce and Google File System papers (which are required reading for next week)
- The core components of Hadoop:
  - HDFS (Hadoop Distributed File System): A scalable and fault-tolerant file storage system
  - MapReduce: A programming model for processing large datasets in parallel. Performs fairly simple operations (map and reduce) at huge scales

Storing Data with HDFS

- The Hadoop Distributed File System (HDFS) provides scalable, fault-tolerant data storage
- HDFS splits large files into blocks and distributes them across multiple machines for parallel storage and retrieval

Spark

- In weeks 3 and 11, we'll look at Spark
- Spark was developed after Hadoop to address some of its perceived limitations
- It's often faster than Hadoop (due to a more sophisticated use of RAM on cluster nodes), and can perform a broader range of tasks
- You'll learn about Spark in week 3
- In week 11, you'll see how Spark can perform advanced data analysis, including machine learning and graph processing:
  - Spark's MLLib library supports a wide range of machine learning algorithms
  - Spark's GraphX library enables complex graph computations

Network Science

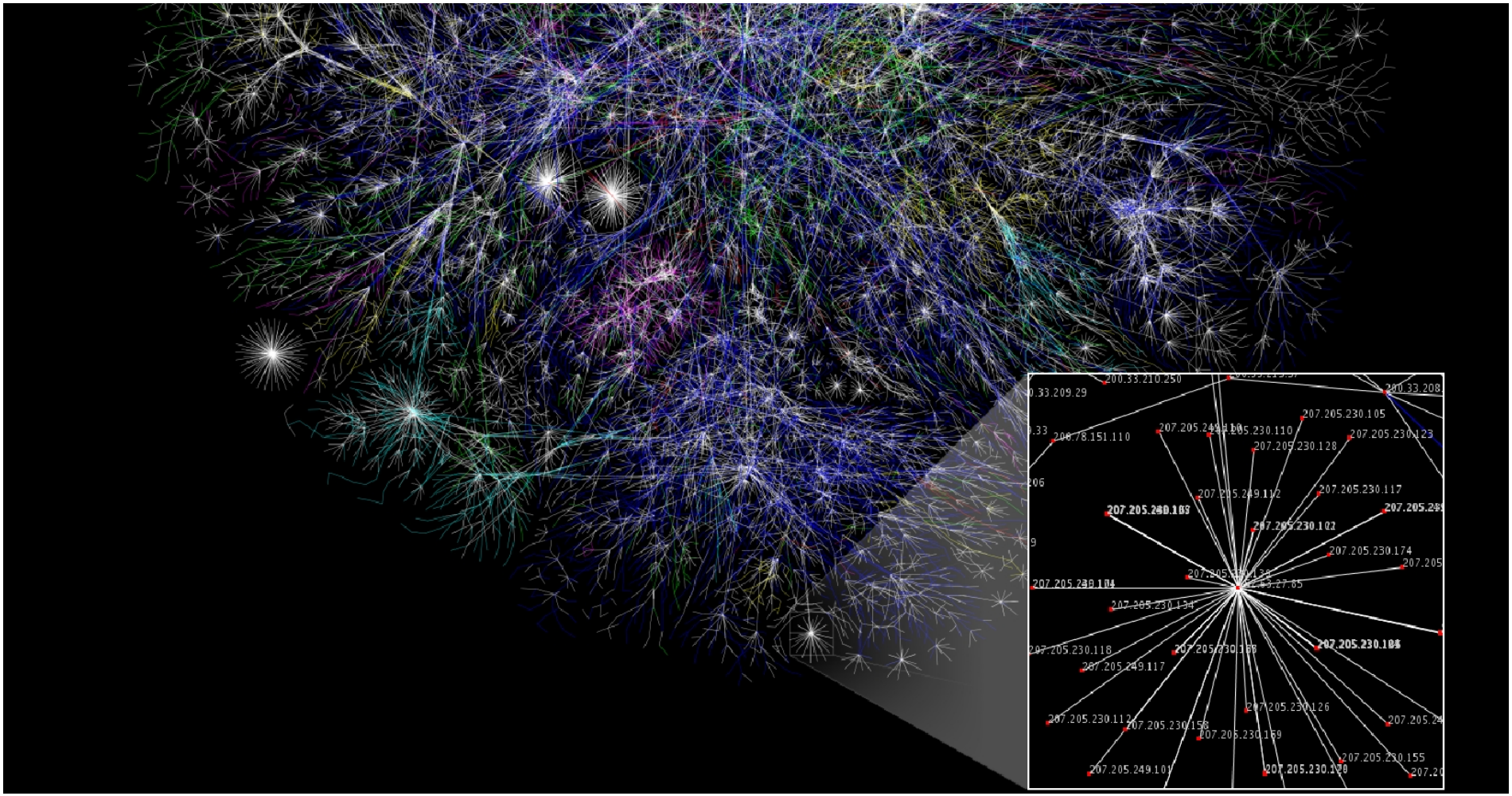
- In weeks 4, you'll move on to the task of extracting value from big datasets
- You'll learn about Network Science, and it's use in processing network (AKA graph) based big datasets
- A network consists of nodes (or vertices) connected by edges (or links)
- Examples of networks in big data include social networks, biological networks, and transportation systems
- Network Science studies the structure, dynamics, and behaviour of networks

Facebook Connections

Facebook Connections Map the World

- The image shows a map of Facebook connections as of 2010
- The lines on the map don't represent physical features like coasts or borders, but instead reflect real human relationships and social networks
- The map also highlights areas with minimal Facebook presence c. 2010, such as China and central Africa, where few connections exist

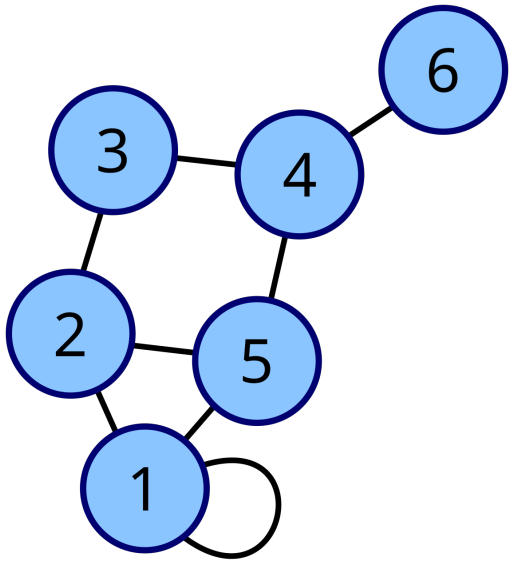
Internet Map



Internet Map Visualisation

- This image shows a map of the Internet, visualising the global structure of interconnected networks
- Each node represents a network, and the edges represent the connections between these networks
- Such visualisations help understand the topology and connectivity of the Internet infrastructure

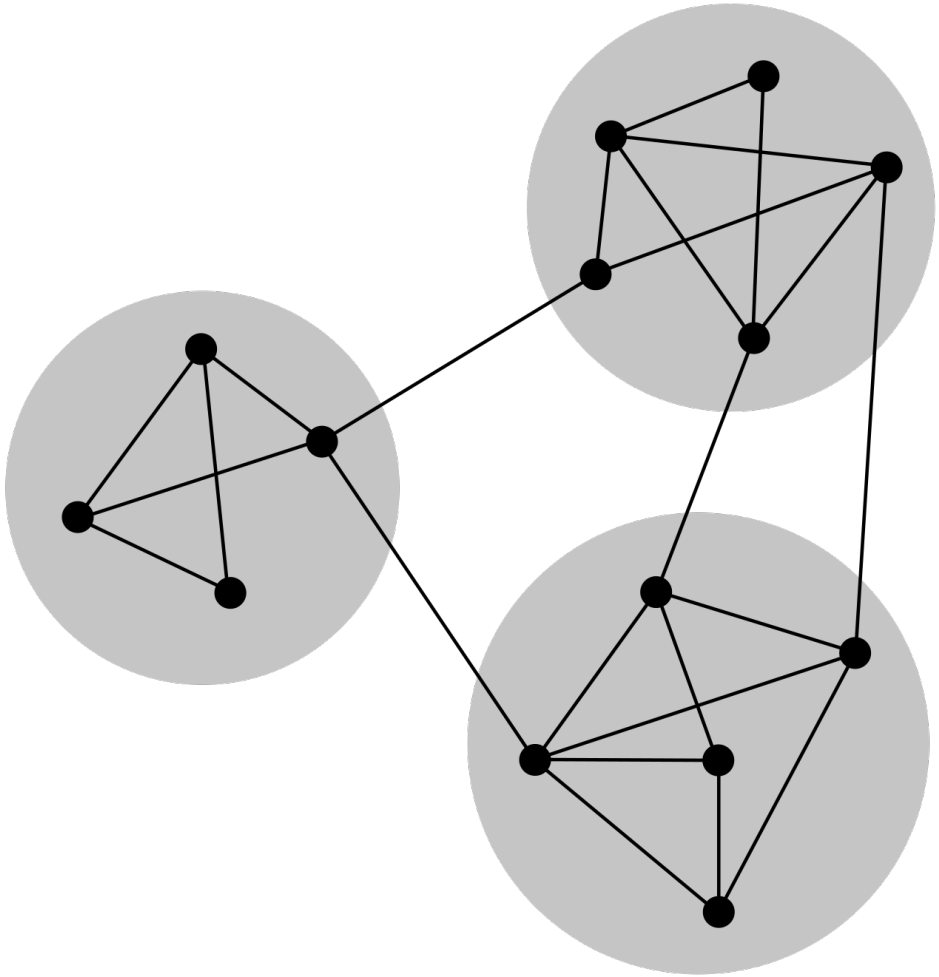
Network Representations



$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

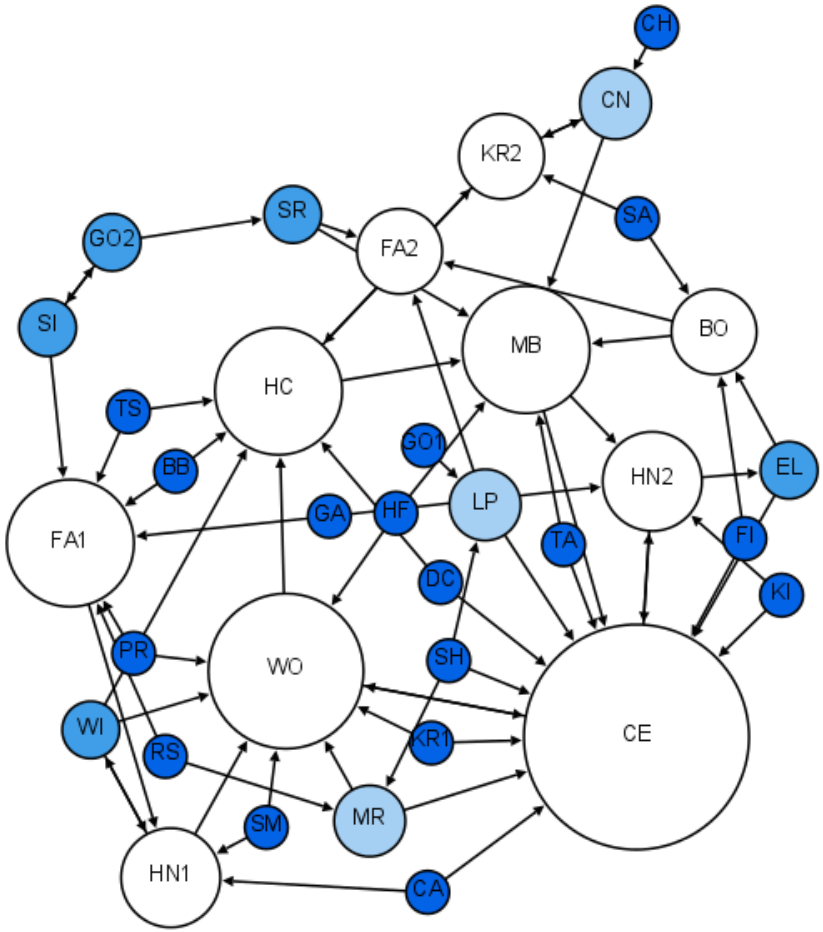
Figure: Comparison between graph and its adjacency matrix.

Network Community Structures



Preferential Attachment in Networks

The Moreno sociogram of a 1st grade class:



Scale-Free Networks

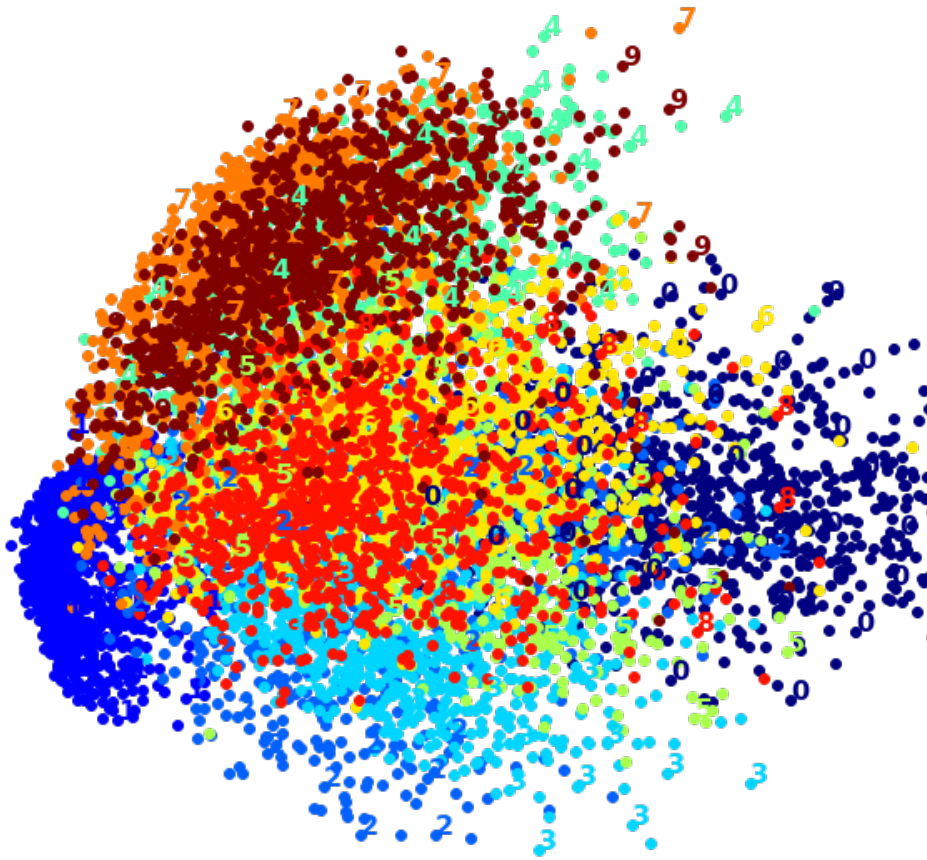
- **Scale-free networks** are a type of network characterised by a power-law degree distribution
- In these networks, most nodes have few connections, while a few nodes (called "hubs") have many connections
- They have a **Power-law distribution**, as you'll see in week 4
- Examples of scale-free networks include the internet, social networks, and biological systems like protein interactions

Unsupervised Learning: PCA and Clustering

- In weeks 5 and 6, you'll learn about unsupervised learning
- Unsupervised learning deals with unlabelled data, where the goal is to find hidden patterns or structure in the data
- You'll see two of the most common techniques in unsupervised learning:
  - **Principal Component Analysis (PCA)** (week 5):
    - A dimensionality reduction technique that projects data onto the principal components—directions of maximum variance
    - PCA helps simplify datasets by reducing features while retaining most of the important information
  - **Clustering** (week 6):
    - Clustering algorithms, such as K-means and hierarchical clustering, group similar data points into clusters based on their features
    - The goal is to partition data into distinct groups where points within a group are more similar to each other than to those in other groups

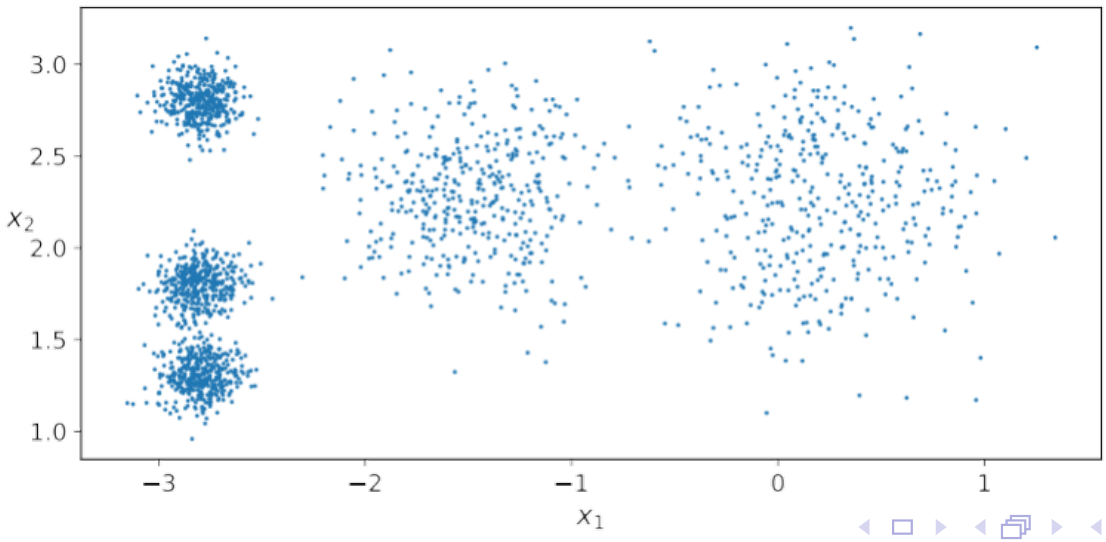
PCA for Exploratory Data Analysis

- We can use PCA to reduce the dimensionality of data all the way down to 2 or 3 dimensions
- At that point, the data can be visualised
- E.g., the plot shows the first two principal components for the handwritten digits dataset
- This often yields insights that can be exploited with further application of machine learning



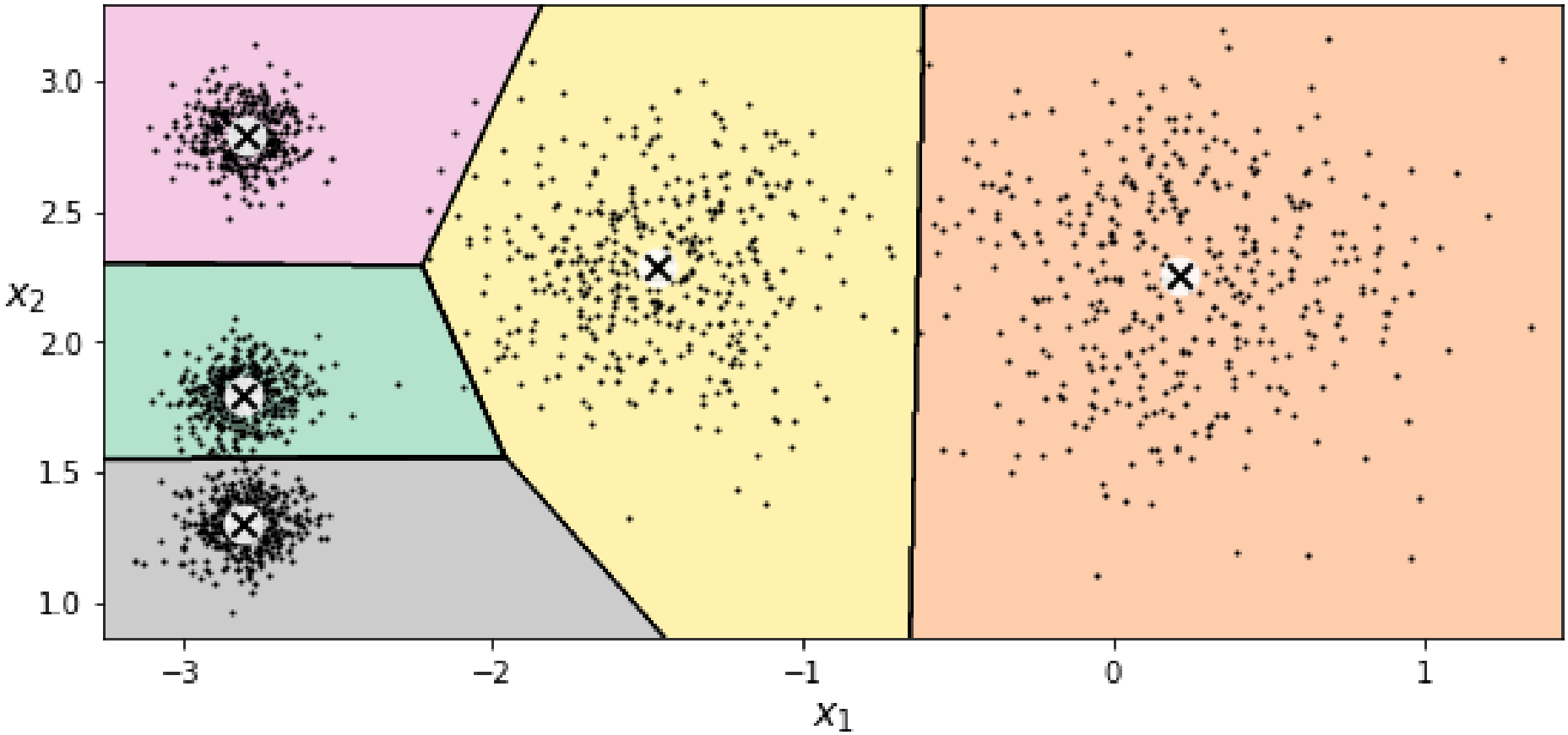
Clustering

- Clustering is an unsupervised learning technique that groups similar data points into distinct clusters
- Clustering allows us to group data points based on their similarity or distance from each other
- It's useful for exploratory data analysis, pattern recognition, and anomaly detection
- Popular algorithms include K-Means, DBSCAN, and Hierarchical Clustering, all of which you'll see in week 6



Clustering with K-Means

Here's the result of applying the K-Means clustering algorithm to the above dataset:



Clustering Faces

Cluster 30

36 36 36

Cluster 31

15 15

Cluster 32

23 23 23

Cluster 33

21 21 21

Cluster 34

26 26 26 26 26

Cluster 35

9 34 34

Cluster 36

31 31

Cluster 37

33 33 33

Cluster 38

6 6

Cluster 39

15 15

Cluster 40

6 6 6

Cluster 41

36 36 36 36

Cluster 42

16 16 16

Cluster 43

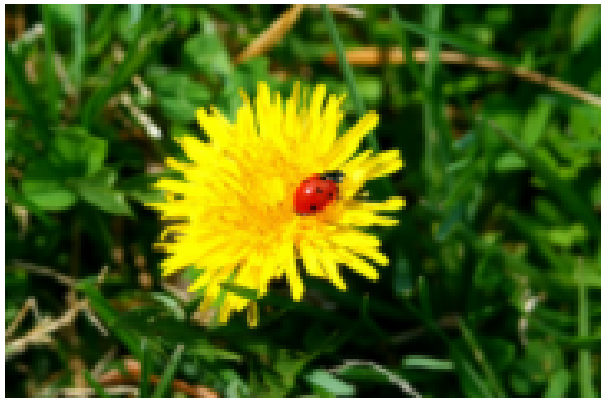
16 16 16

Cluster 44

35 35 35

Clustering-Based Image Compression

Original image



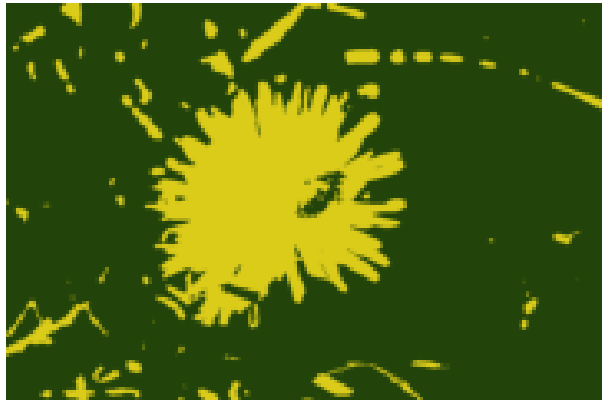
10 colors

8 colors

6 colors

4 colors

2 colors

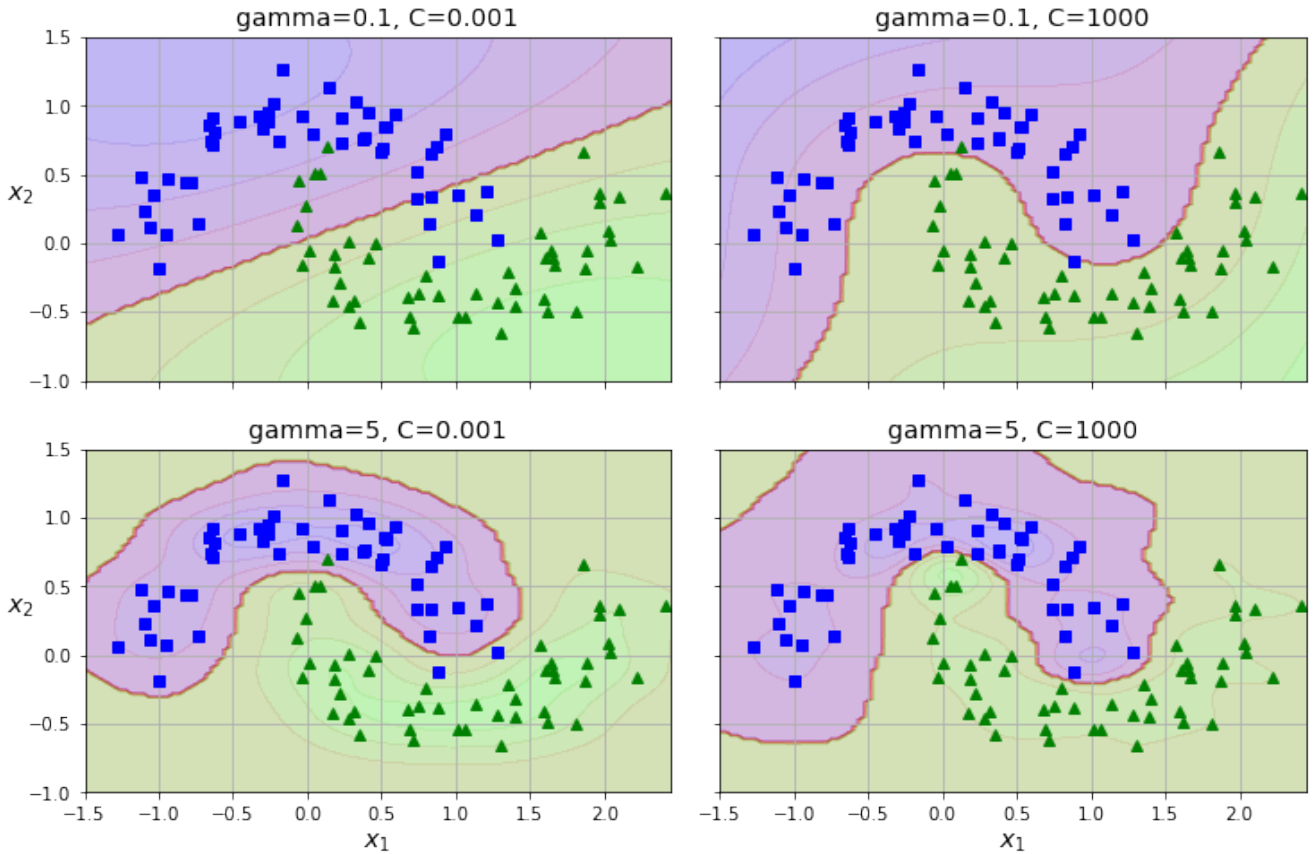


Supervised Learning

- In week 8, we'll look at supervised learning
- Supervised learning is a type of machine learning where the model is trained on labelled data
- The model learns to map input data to output data based on the input-output pairs provided during training
- We'll look at several supervised learning algorithms, including k-Nearest Neighbours and Support Vector Machines

Half Moon Classification

The plot illustrates the use of Support Vector Machines (SVM) on a dataset

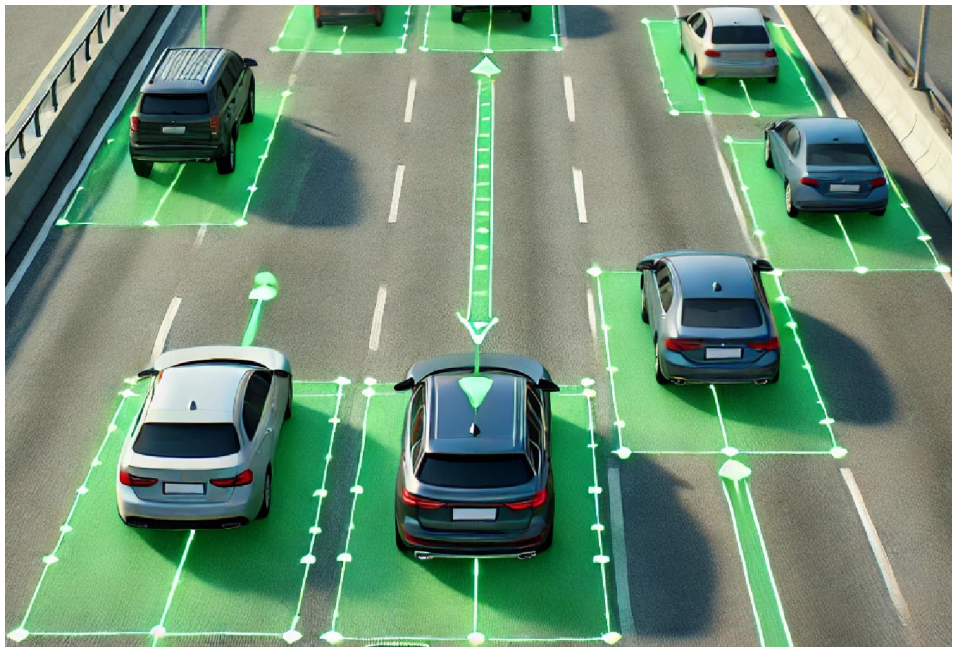


Natural Language Processing

- In week 8, we'll also look at Natural Language Processing (NLP), which enables computers to understand, interpret, and respond to human language
- We'll focus on the task of document search, where the goal is to retrieve relevant documents from a corpus, usually based on a user's query
- We'll look at "bag of words" style document reprentations, culminating in an approach called Term Frequency-Inverse Document Frequency (TF-IDF):

Bayesian Statistics and Filters

- In weeks 7, 9 and 10, you'll learn about Bayesian statistics and filtering
- This will culminate with an introduction to the Kalman filter - a very widely-used algorithm that uses a series of measurements observed over time to estimate the current state of a system and predict it's future evolution
- Applications include vehicle tracking, robotics and navigation



Course Outline

1. Introduction and Course Overview
2. Hadoop
3. Spark
4. Network Science
5. Linear Algebra Approaches
6. Clustering
7. Probabilistic Modelling
8. Classification and Real World Applications
9. Parallel Computing, Bayesian Modelling and Real World Applications
10. Parallel Computing, Bayesian Modelling and Real World Applications
11. Advanced Spark
12. Advanced Approaches

Prep for Next Week

Please read through Google's original white papers on Google File System (GFS) and MapReduce:

- **The Google File System**, 2003
- **MapReduce: Simplified Data Processing on Large Clusters**, 2004

They're very accessible, and they laid the groundwork for Hadoop and HDFS

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that

Distributed File Systems and MapReduce

Big Data Middleware

- Today and in weeks 3 and 11, we'll focus on middleware for big data analysis
- Many datasets (or collections of datasets) are too big to fit onto a single computer
  - We'll see a few examples today
- These datasets are stored in a distributed way across different servers in a data centre
- Big data middleware sits on top of data centre hardware. It:
  - Handles the distributed storage of data
  - Provides a computational abstraction (AKA programming model) to make the storage and subsequent processing of these datasets more tractable

Big Data Middleware

- We'll start with the original big data middleware: Google File System (GFS) and Google MapReduce, which were the first big data middleware solutions
  - Google published papers on them in 2003 and 2004 respectively (and implemented them internally prior to this)
  - We'll introduce them with the original motivating examples from Google
  - We'll see that "map reduce" is a very simple computational paradigm, but you can do a lot of useful big processing with it
  - Google unfortunately hasn't released the source code for GFS and MapReduce, so you can't run it

Big Data Middleware

- We'll then look at the first open source big data middleware solution, which was Hadoop MapReduce and the Hadoop Distributed File System (HDFS)
- All of this will lay the groundwork for introducing Apache Spark next week, which is a fast, cutting edge generalisation of the MapReduce concept

Required Reading for Today's Lecture

- **The Google File System**, 2003
- **MapReduce: Simplified Data Processing on Large Clusters**, 2004

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds

In this pap
designed to s
aspects of our
micro-benchm

Categories

D [4]: 3—Dis

General Te

Design, reliab

Keywords

Fault toleranc

*The authors
{sanjay,hgobi

Permission to m
personal or class
not made or distr
bear this notice a
republish, to post
permission and/o
SOSP'03, Octobe
Copyright 2003 A

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that

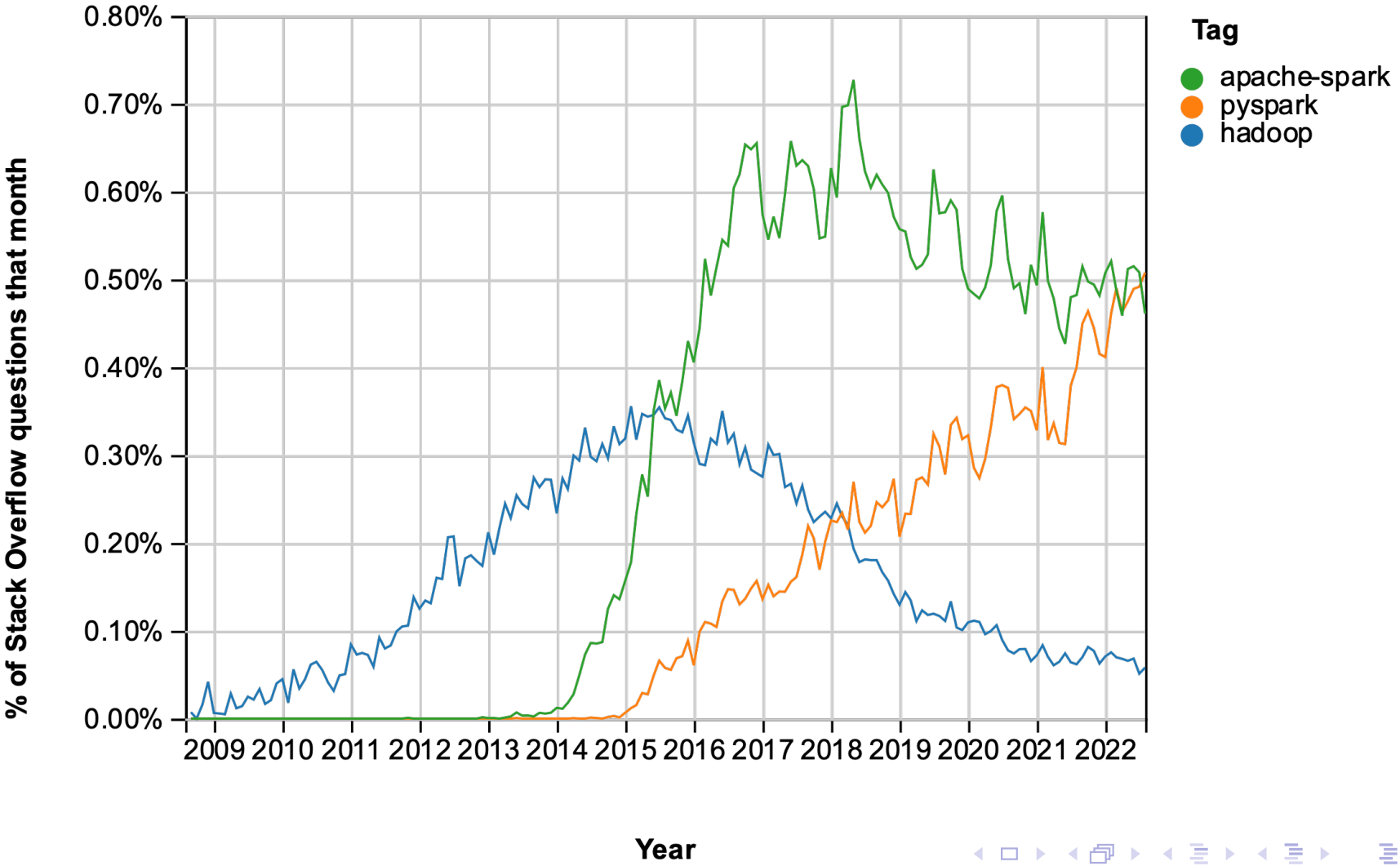
Big Data Middleware at Google

- The original example of big data middleware was Google's proprietary MapReduce and Google File Store:
  - Described by Google in 2003 and 2004 research papers, and implemented at Google some years before that
  - It provided a computational abstraction: simple distributed processing + result aggregation, as we'll see
  - It was designed for use by Google's search engine
- Google's MapReduce software is proprietary and has never been released to the public, but shortly after the 2003 and 2004 publications, Hadoop and HDFS emerged as a free open source equivalent, which use the same MapReduce computational abstraction

Hadoop and Spark

- We'll focus on Spark in week 3, and we'll see advanced Spark algorithms in week 11
- Spark is a newer alternative to Hadoop
  - It can perform the same MapReduce tasks as Hadoop, often with a speed-up of 10x to 100x
  - It also performs a more general set of computations which generalise MapReduce, whilst maintaining the same high performance

Hadoop and Spark Popularity Over Time



Google's Datasets

- Google builds its search engine by using a piece of software known as a "web crawler" to explore the entire web continually in order to find pages, which are then added to a giant index
  - As you might imagine, this generates a lot of data
- The web crawler is called Googlebot
- For more information, see:
  - https://developers.google.com/search/docs/fundamentals/how-search-works
  - https://developers.google.com/search/docs/crawling-indexing/googlebot

Datasets and Data Centre Node Specs

- From Google's MapReduce paper, as of 2004:
  - Google had multiple data centres
  - Each data centre had hundreds or thousands of servers
  - Hard disk capacity per server: 320GB
  - Memory per server: 4GB
  - Network bandwidth per server: 100-200Gbps
  - Average file size: 113GB
- Data was largely generated by the Googlebot web crawler, as we'll see
- Note that individual files were usually small enough to fit onto a single server (113GB vs 320GB).
  - However, the entire file store was far too large to fit onto a single server

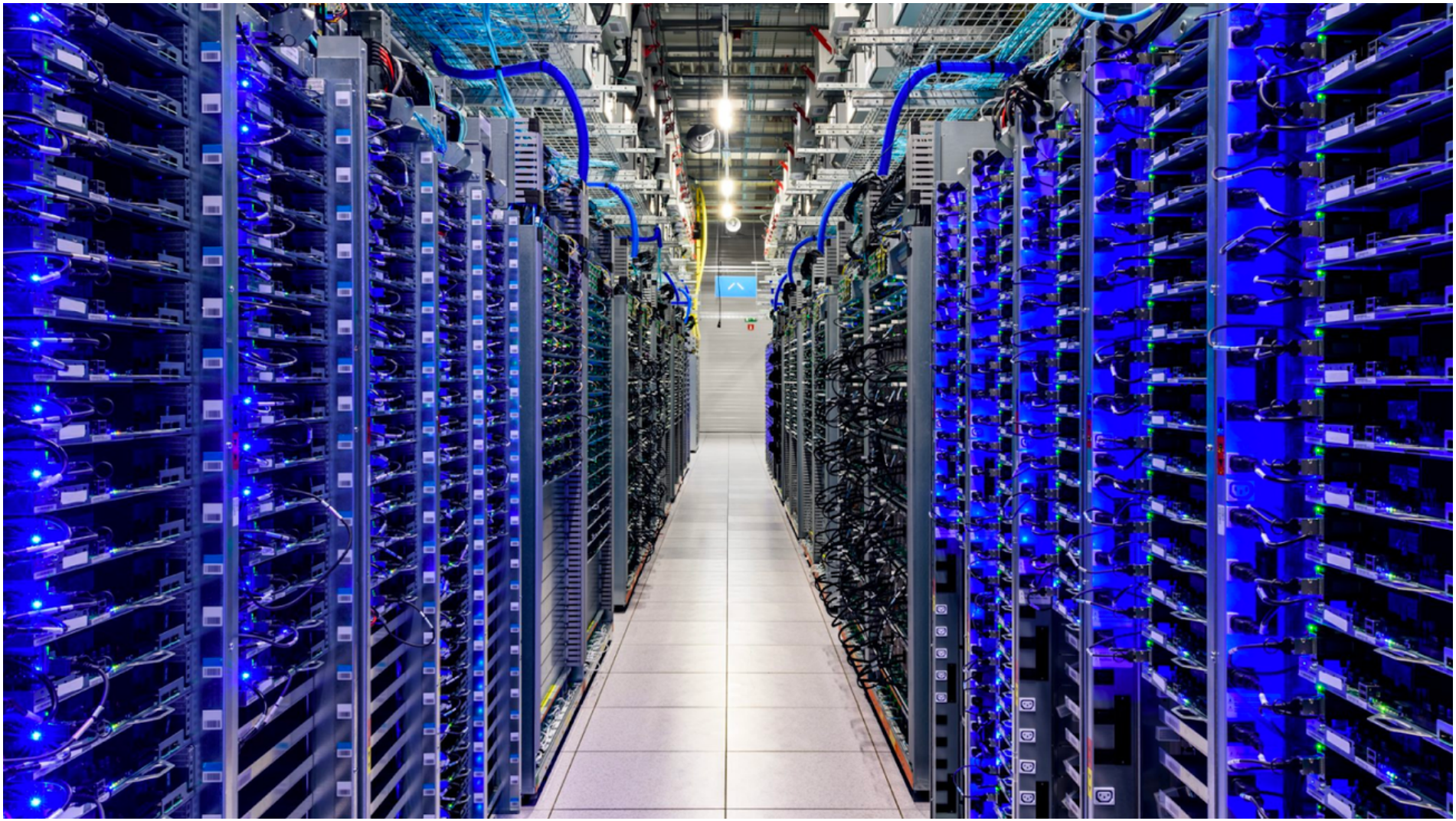
Datasets and Data Centre Node Specs

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

AWS Data Centre External View

Google Data Centre Inside View

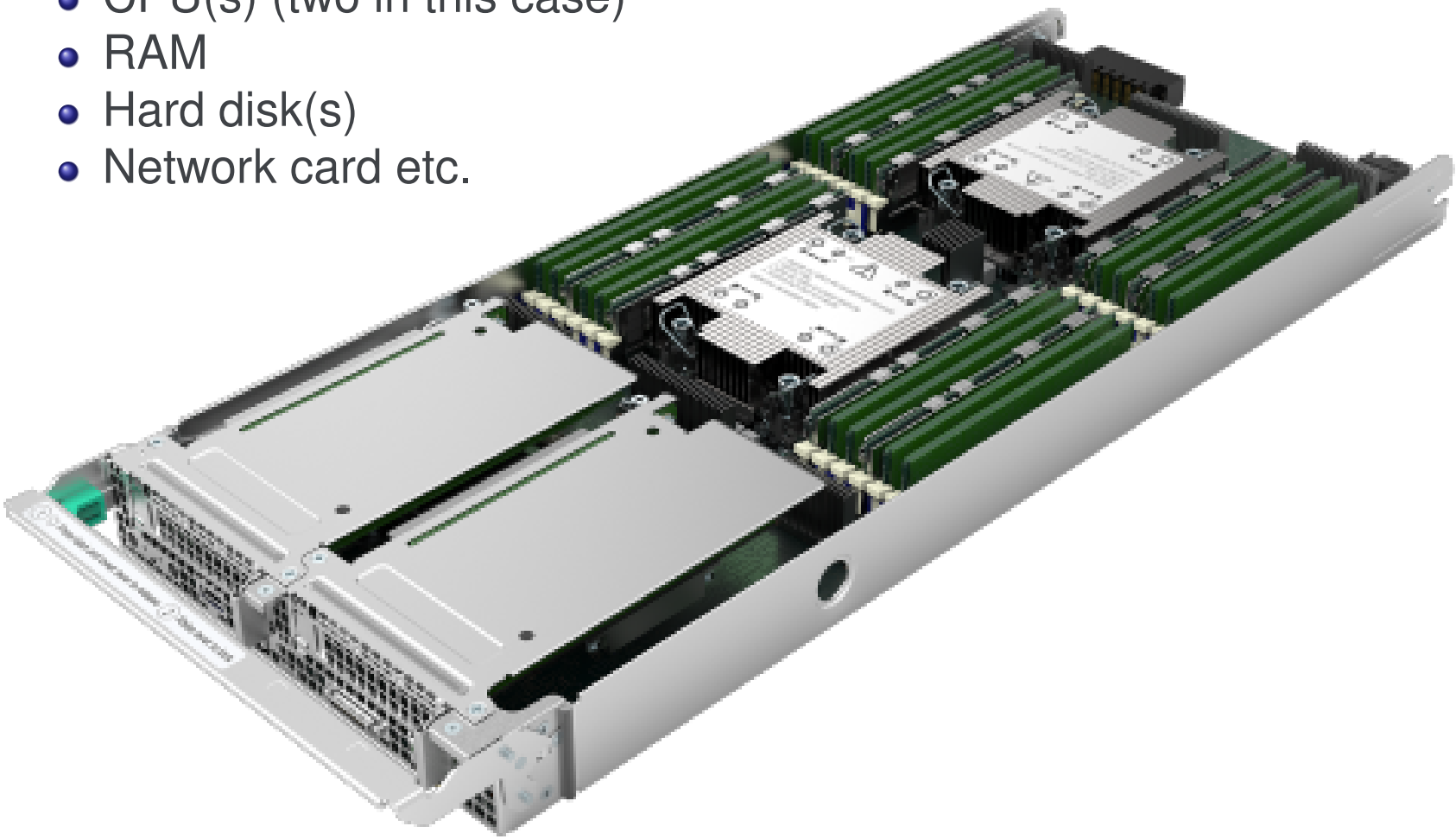


Server Rack

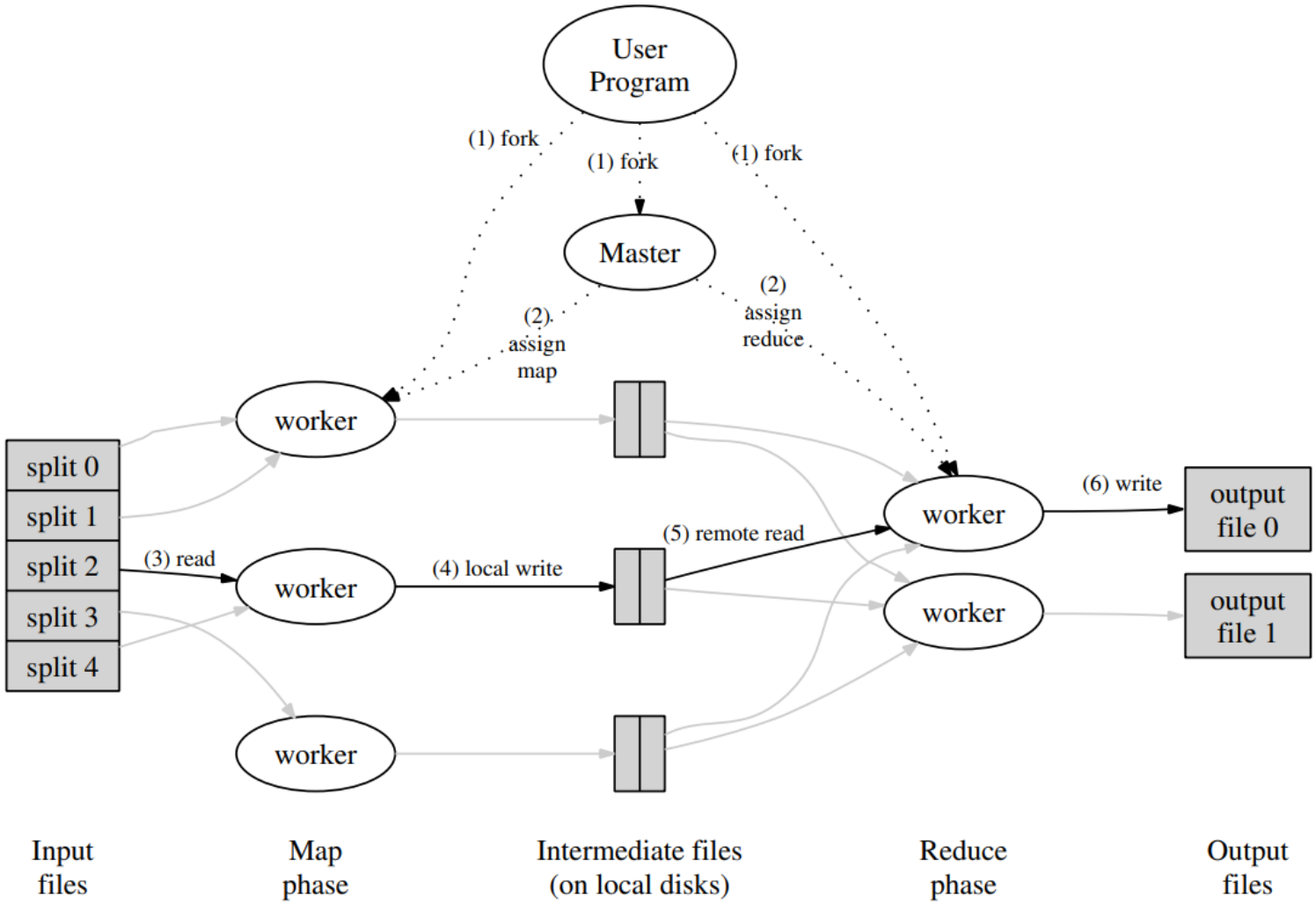
- Server racks hold servers, storage units, and switches

Rack-Mounted Server

- Contains inexpensive "commodity" hardware, similar to a desktop PC:
  - CPU(s) (two in this case)
  - RAM
  - Hard disk(s)
  - Network card etc.



The MapReduce Programming Model



Google File System

- Google's MapReduce is supported by a distributed file system called Google File System (GFS)
- When data is loaded onto the system it is divided into blocks
- Each block is stored on multiple different servers (typically 3)
- Files in GFS are write-once only, although data can be appended
- GFS is optimised for streaming reads of large files, rather than random access
- MapReduce tasks are divided into two phases:
  - Map tasks which are done concurrently on each block of data in a file, with each block being mapped over using the CPU of one of the servers on which it's stored
  - Reduce tasks then combine data to produce the final output

Advantages of Google File System

Advantages of Google's storage approach:

- It's resilient to server failure, because each block is stored on multiple servers
  - This is important, because failures will be frequent with thousands of servers in a cluster
- It allows parallel processing of the data, because the blocks of a single file are stored on different servers, each with its own CPU(s)
- It has huge capacity, because the file system uses hundreds or thousands of servers, each with hundreds of TBs of hard disk space

Advantages of Google File System

- It allows data sharing, because any user with access to GFS has access to the data, and can analyse it with MapReduce
- It's scalable – it scales linearly to handle larger data by adding more servers to the cluster
- Provides a clean abstraction for parallel programming:
  - MapReduce isn't essential for operating on big data, but it makes it easier
  - Google developed MapReduce to simplify big data tasks that engineers were previously implementing in more manual, *ad hoc*, ways.

Using MapReduce for Smaller Datasets

- In Google's original white paper, individual files were usually small enough to fit onto a single server (113GB vs 320GB)
  - However, the entire file store was far too large to fit onto a single server
- Big data middleware is useful for large files that can still technically fit onto a single server.
  - You still get the advantages of resilience, parallel execution of MapReduce, data sharing, etc.
- However, it's less well-suited for very small files:
  - GFS block size is 64MB, so files smaller than this will be stored as a single block, meaning that you'll get no parallel processing speed-up when using MapReduce

Motivating Examples

- In this part, we'll get an idea of just how useful MapReduce is for applying data processing to big data.

Word Counting

- Consider the problem of counting the number of occurrences of each word in a large collection of documents
  - This is an important step for the natural language processing that we'll do in week 9
  - We'll use these word counts to create document representations, and these will form the basis of our search engine designs

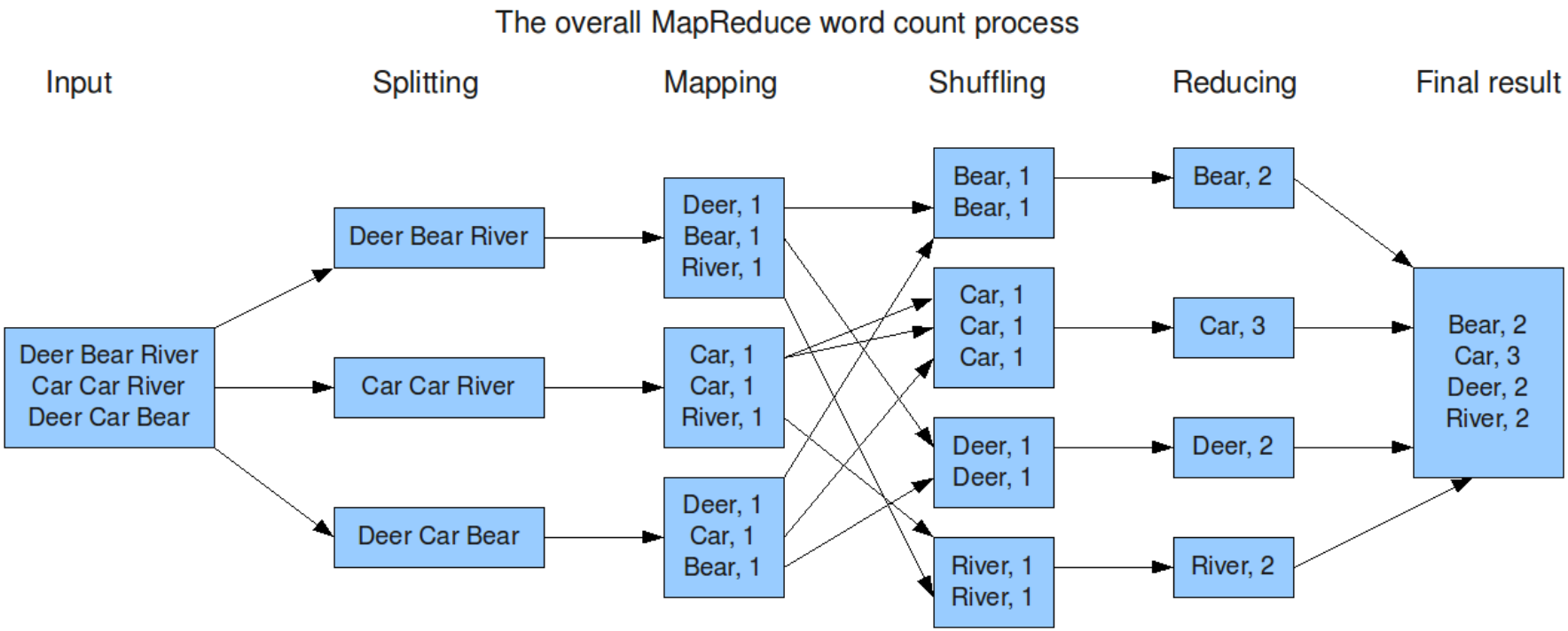
Word Counting

- MapReduce pseudo-code for this problem:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce: Word Count



Count of URL Access Frequency

- Dataset: a log of web page requests
- Output: a count of the number of requests for each web page
- MapReduce implementation:
  - The map function processes logs of web page requests and outputs (URL, 1) for each web request
  - The reduce function adds together all values for the same URL and emits a (URL, total count) pair

Reverse Web-Link Graph

- Dataset: a file of web pages and the URLs that each one links to
- Output: a mapping from each web page to the list of the web pages that link to it
- MapReduce implementation:
  - The map function outputs (target, source) pairs for each link to a target URL found in a page named source.
  - The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: (target, list[source])

Term-Vector per Host

- Dataset: A collection of text files
- Output: A term vector summarises the most important words that occur in a document or a set of documents as a list of `<word, frequency>` pairs.
- MapReduce implementation:
  - The map function emits a `<hostname, term vector>` pair for each input document (where the `hostname` is extracted from the URL of the document).
  - The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final `<hostname, term vector>` pair.

Inverted Index

- Dataset: A collection of text files
- Output: A mapping from each word to the list of documents that contain the word
- MapReduce implementation:
  - The map function parses each document, and emits a sequence of (word, document ID) pairs.
  - The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a (word, list(document ID)) pair.

Distributed Grep

- Dataset: a collection of text files
- Output: for a given regular expression, the set of lines that match it
- MapReduce implementation:
  - The map function emits a line if it matches a supplied pattern.
  - The reduce function is an identity function that just copies the supplied intermediate data to the output.

Hadoop

- Open source software framework, providing the functionality of Google's MapReduce and GFS
- Created by Doug Cutting and Mike Carafella in 2005, based on Google's 2003 and 2004 white papers
- Cutting named the program after his son's toy elephant.

Who Uses Hadoop?

HDFS

- HDFS is Hadoop's implementation of the file system concept outlined by Google in their 2003 GFS white paper
- As with GFS:
  - HDFS works best with large files
    - Typically GBs per file
  - Files in HDFS are write-once only, but can be appended to
  - HDFS is optimised for streaming reads of large files and not random reads
- Files are split into 64MB blocks (for Hadoop v1) or 128MB blocks (for Hadoop v2)
- Each block is replicated across several servers (default 3)

HDFS Implementation Details

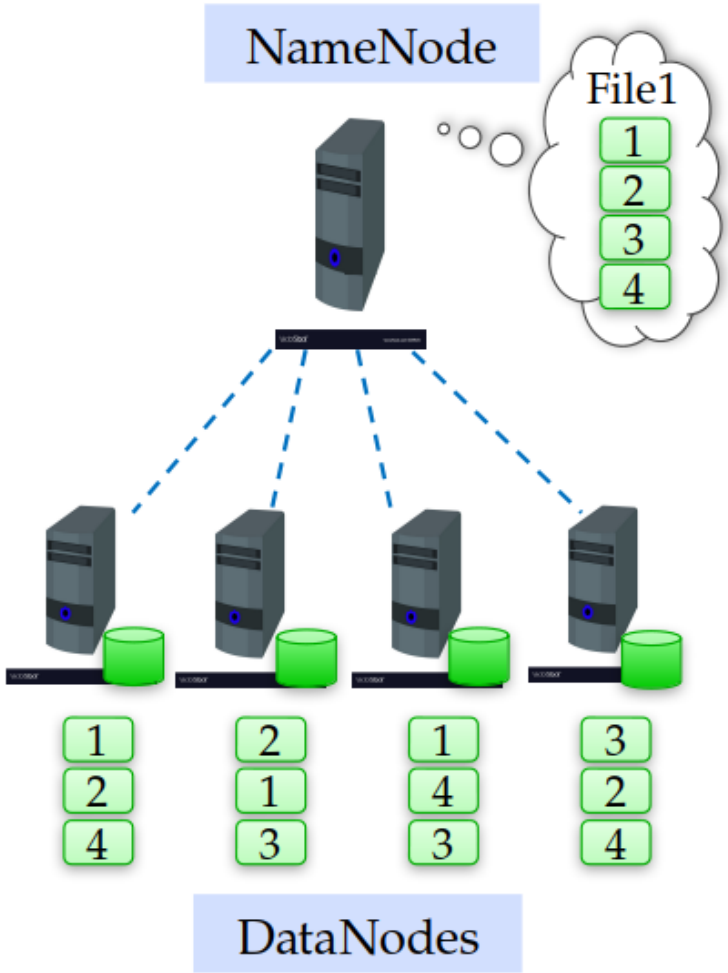
Hadoop clusters include the following daemons (i.e., programs) which work together to maintain HDFS:

- **NameNode**:
  - Runs on the (unique) master server.
  - Acts as the bookkeeper for HDFS.
  - Keeps track of where blocks constituting copies of each file are stored.
  - Monitors the health of the distributed file system.
  - Is a single point of failure for the cluster (so it is commonly on a dedicated machine).

HDFS Implementation Details

- **DataNodes**:
  - Runs on each worker node.
  - Stores the actual blocks (the client can upload files to DataNodes).
  - Informs the NameNode of any local changes to files.
  - Polls the NameNode for any required changes to blocks.
- **Secondary NameNode**:
  - Takes snapshots of the NameNode's metadata.
  - Purpose is to minimise the impact of NameNode failure.

HDFS Implementation Details



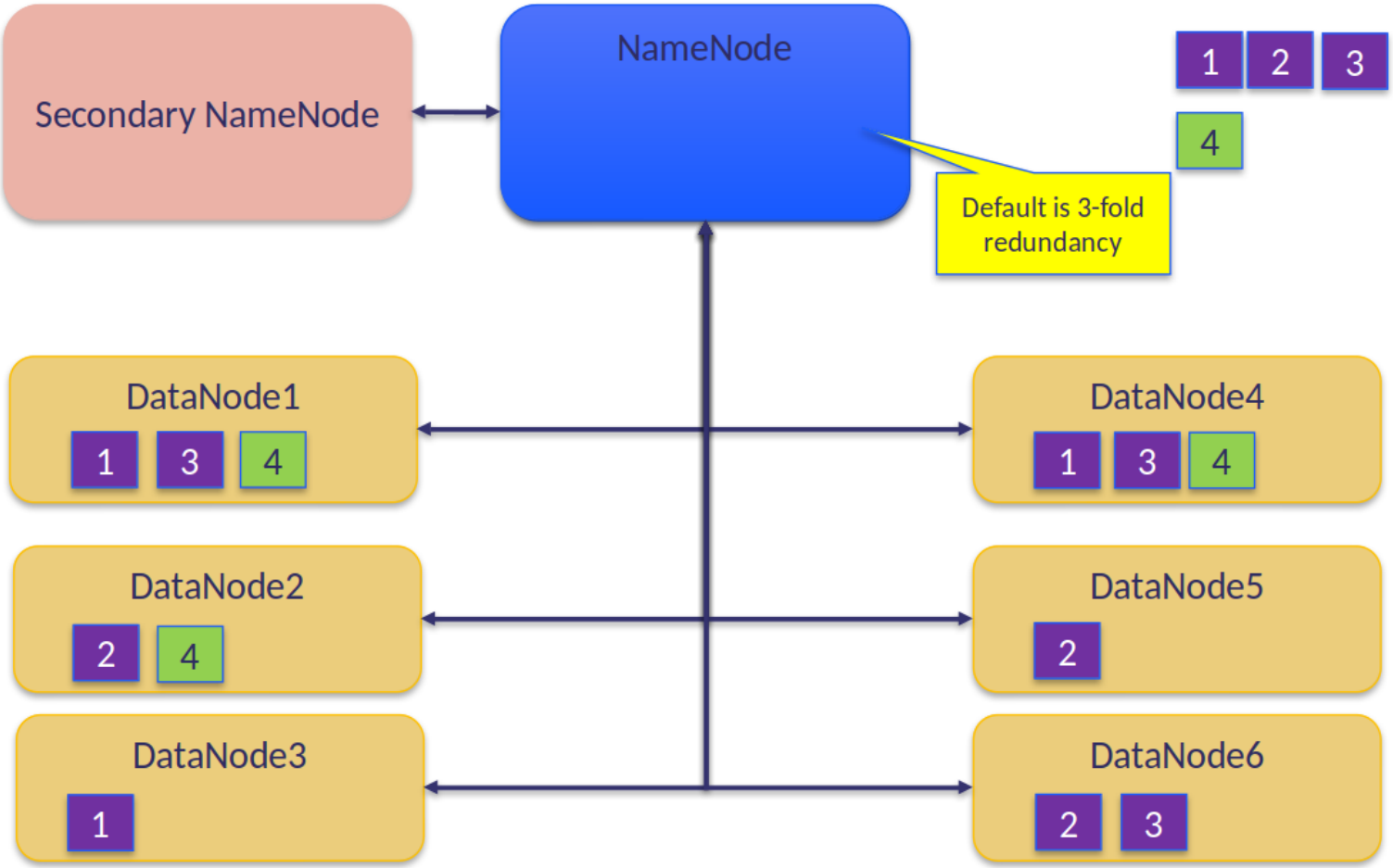
The NameNode

- Stores the HDFS file system information in the FsImage.
  - FsImage is a file stored on the OS filesystem that contains the complete directory structure (namespace) of the HDFS, with details about the location of the data on the Data Blocks and which blocks are stored on which node.
  - This file is used by the NameNode when it is started.
- Updates to the file system (add/remove blocks) do not change the FsImage.
  - They are instead written to a log file.
- When starting, the NameNode loads the FsImage file and then applies the changes in the log file.

The Secondary NameNode

- Not a backup for the NameNode
- Periodically reads the log file and applies the changes to the FsImage file, bringing it up to date
- Allows the NameNode to restart faster when required

HDFS Storage Example



Data Retrieval

- When a client wants to retrieve data:
  - Communicates with the **NameNode** to determine which blocks make up a file and on which data nodes those blocks are stored
  - Then communicates directly with the data nodes to read the data
  - So, no data passes through the master server

Hadoop MapReduce

- Written in Java
- Performs the MapReduce operation on HDFS data

The Mapper

- Reads data as key/value pairs
  - The key is often discarded
- Outputs zero or more key/value pairs
- E.g., for word count:

```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

Shuffle and Sort

- Shuffle: copies the output from each Mapper using HTTP across the network.
- Sort: merge sorts Reducer inputs by key(s) (since different Mappers may have output the same key).
- As a result:
  - Output from the Mapper is sorted by key
  - All values with the same key go to the same worker server

The Reducer

- Called once for each unique key
- Gets a list of all values associated with a key as input
- The reducer outputs zero or more final key/value pairs
  - Usually just one output per input key
- E.g., for word count:

```
public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context) throws IOException, InterruptedExceptio
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Orchestration of Hadoop MapReduce

- "Orchestration" refers to the process of implementing MapReduce computations on a Hadoop cluster
- Hadoop refers to each MapReduce call as a "job":
  - This "job" is broken down into multiple "tasks"
  - "Tasks" are then performed in parallel

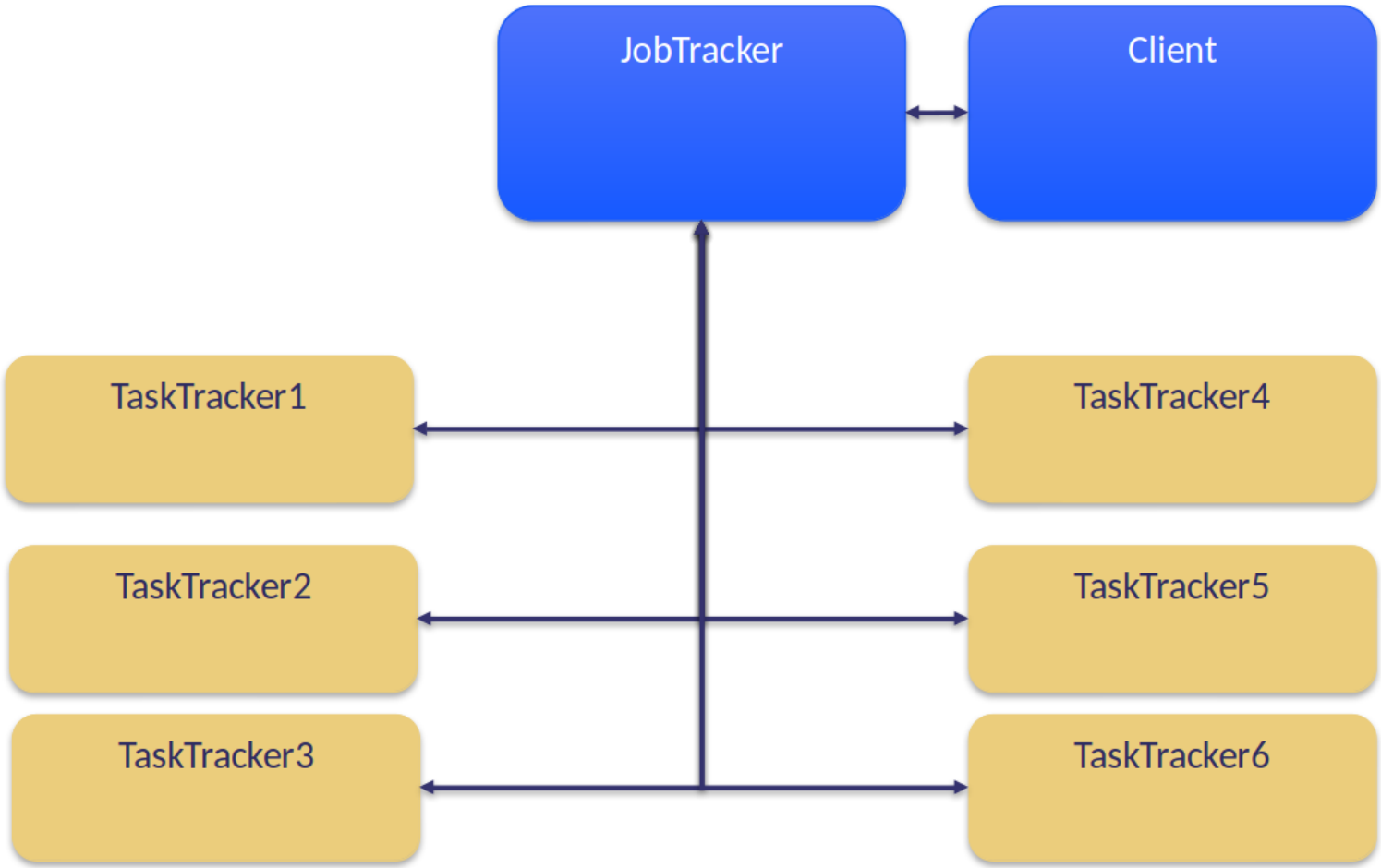
Orchestration of Hadoop MapReduce

- Hadoop clusters running **MapReduce v1** include the following daemons:
- **JobTracker**:
  - Runs on the single master server
  - The interface to clients wishing to run MapReduce jobs
  - Determines which computers do what and when
  - Will reassign (map and reduce) “tasks” to computers if tasks fail
    - Including if it stops hearing the heartbeat signal
- **TaskTrackers**:
  - Run on each worker server
  - Spawns (multiple) Java Virtual Machines (JVMs) to execute tasks
  - Maintains a heartbeat with JobTracker

Orchestration of Hadoop MapReduce

- In **MapReduce v2**:
  - **JobTracker** was replaced by Resource Manager and Application Master
  - **TaskTracker** was replaced by Resource Manager, Application Manager, and Application Master

JobTracker and TaskTracker



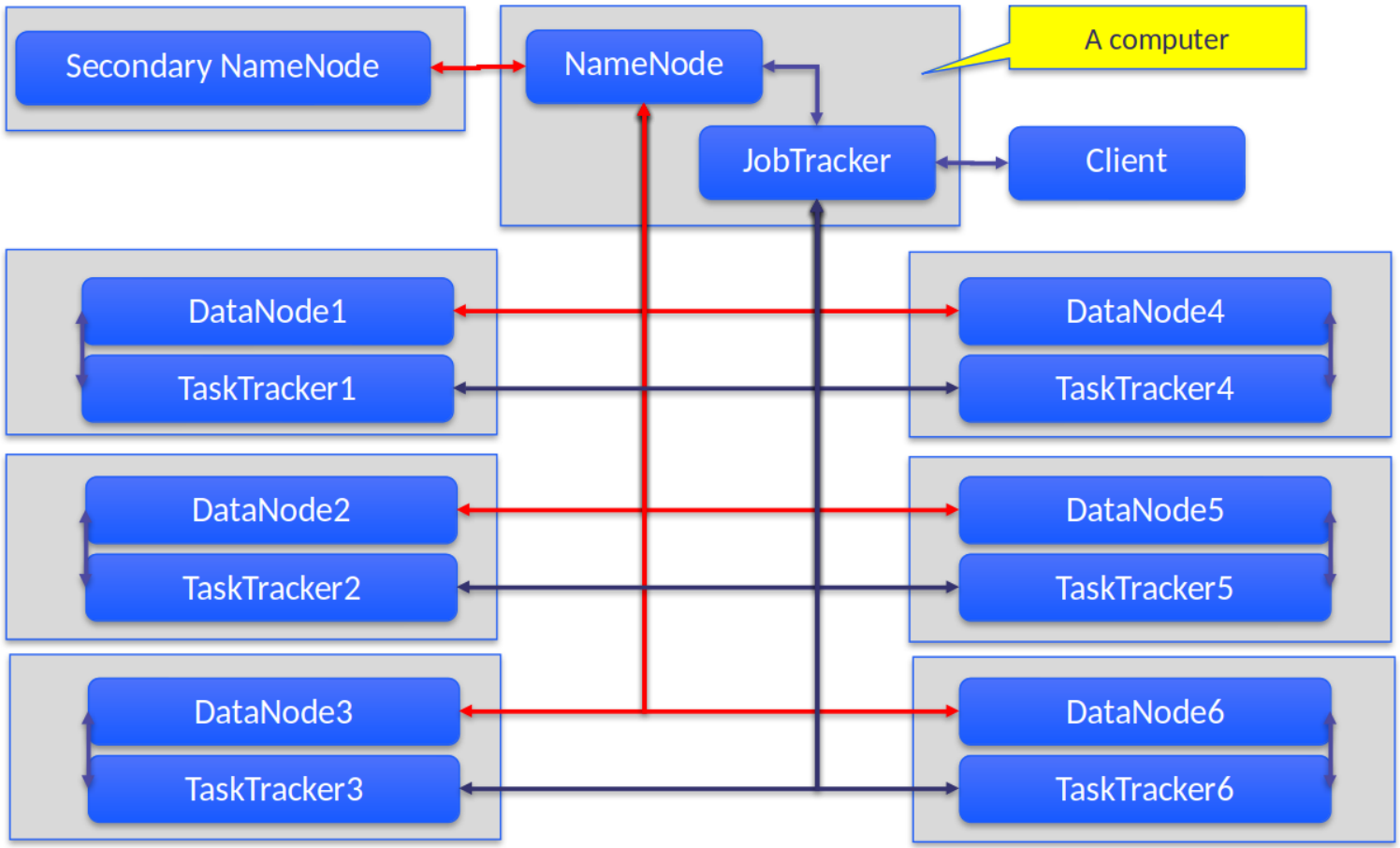
Execution Modes

- Hadoop can run in three execution modes:
  1. Fully distributed
  2. Pseudo-distributed
  3. Local

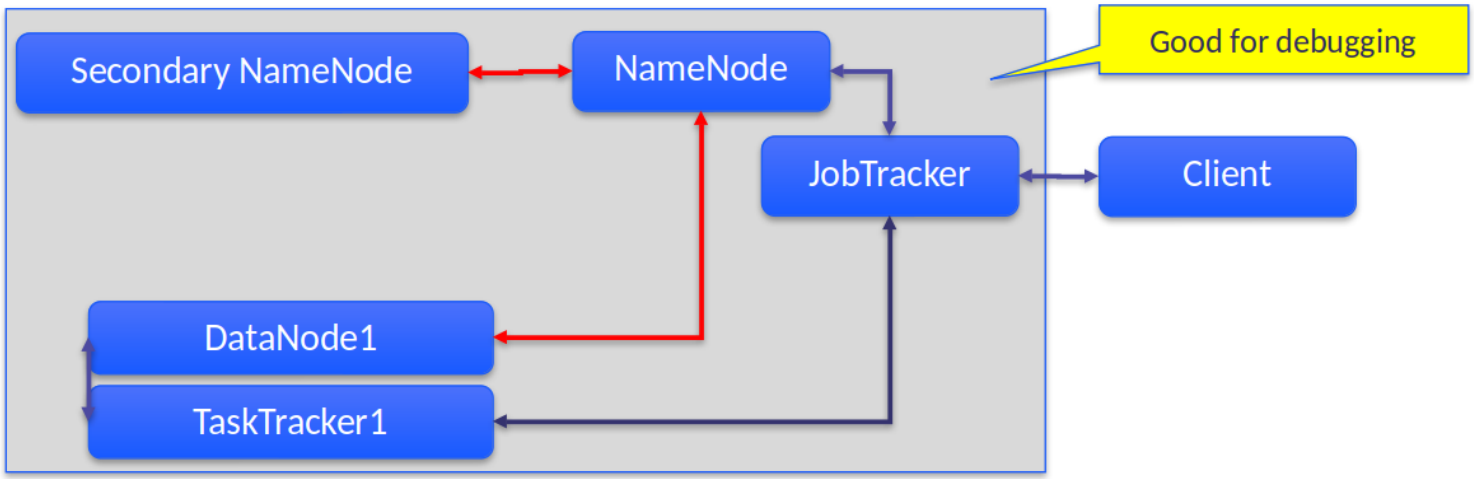
Fully Distributed Mode

- The following diagram represents a fully distributed mode in Hadoop.
  - The NameNode and Secondary NameNode handle metadata.
  - DataNodes store the actual data.
  - JobTracker and TaskTrackers manage the computational tasks.
- Key Components:
  - NameNode: Manages file system namespace and access.
  - DataNodes: Store data and report to NameNode.
  - JobTracker: Schedules and tracks jobs.
  - TaskTrackers: Execute tasks assigned by JobTracker.

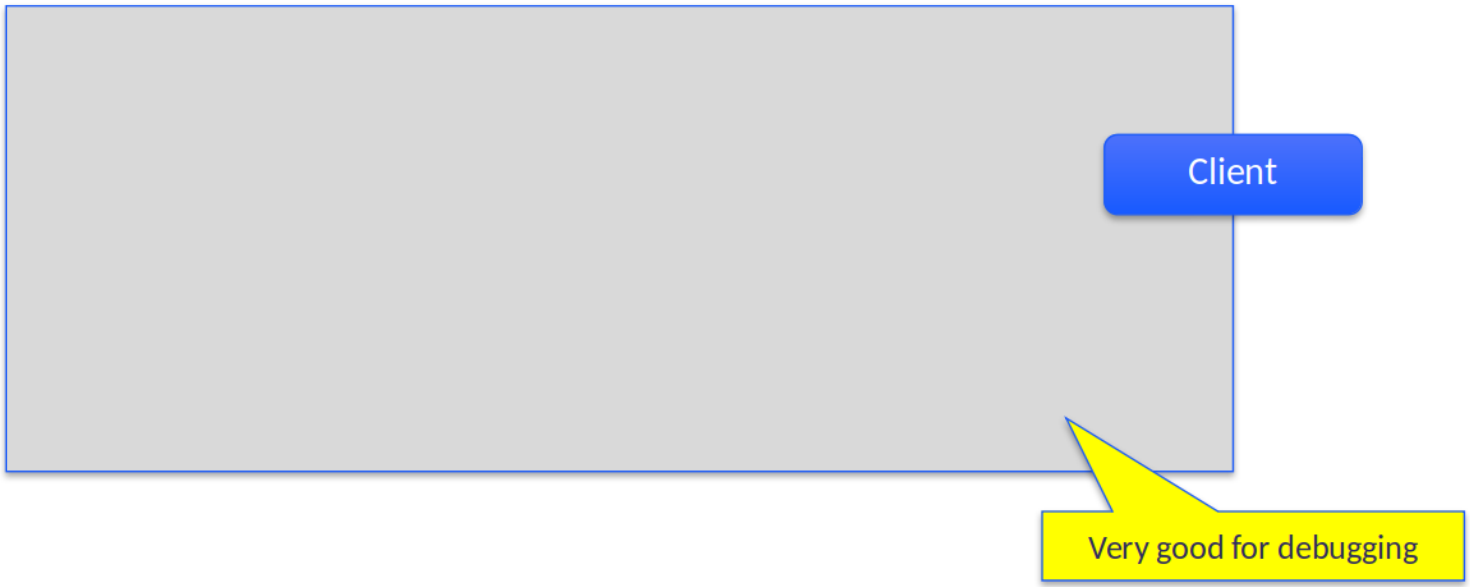
Fully Distributed Mode



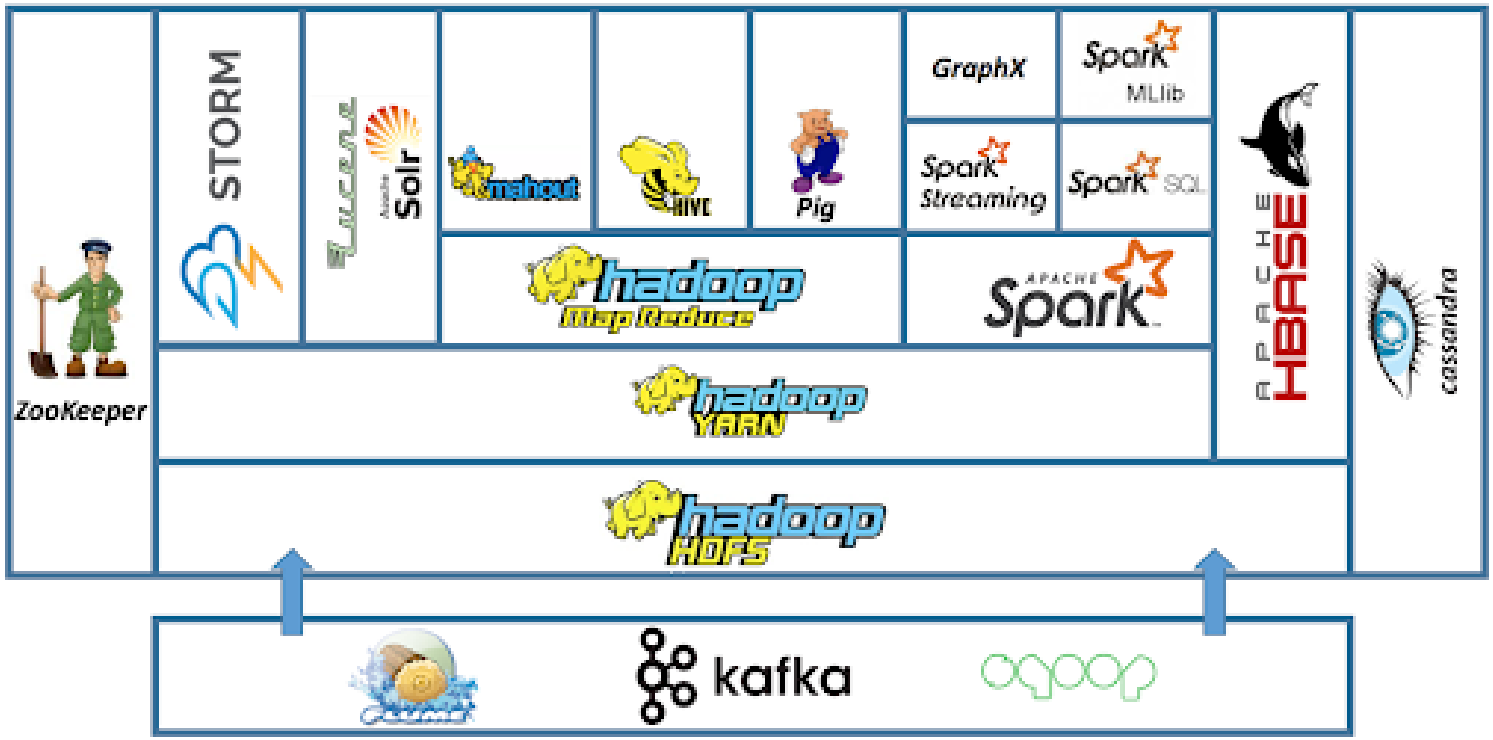
Psuedo-Distributed Mode



Local Mode



The Hadoop V2 Ecosystem



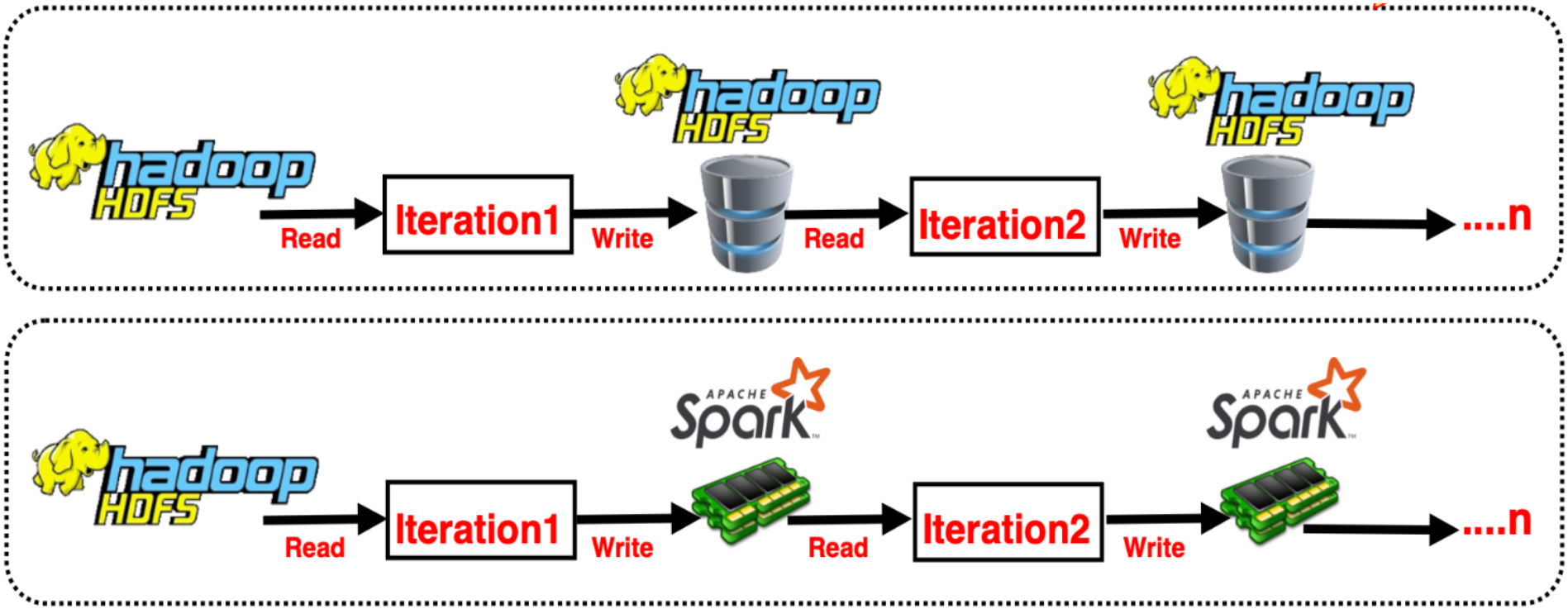
Spark

Spark

- Developed as an improvement of Hadoop MapReduce.
- Works with any Hadoop-supported storage system (HDFS, S3, Avro, etc.), but not a modified version of Hadoop.
- Generally faster than Hadoop MapReduce:
  - Spark can perform in-memory processing, while Hadoop MapReduce has to read from/write to a disk (100x faster)
- Supports a broader set of computations than Hadoop
- APIs in Java, Scala, Python, and R (often 2-10x less code vs Hadoop)

Hadoop Vs Spark

- Spark keeps data in-memory while Hadoop MapReduce writes to disk after each map and reduce operation.
- This makes Spark 10x-100x faster than Hadoop for MapReduce computations



Word Count in Spark and Hadoop

Spark also has a concise API, often resulting in far less code than Hadoop:

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Figure: Word Count in Hadoop

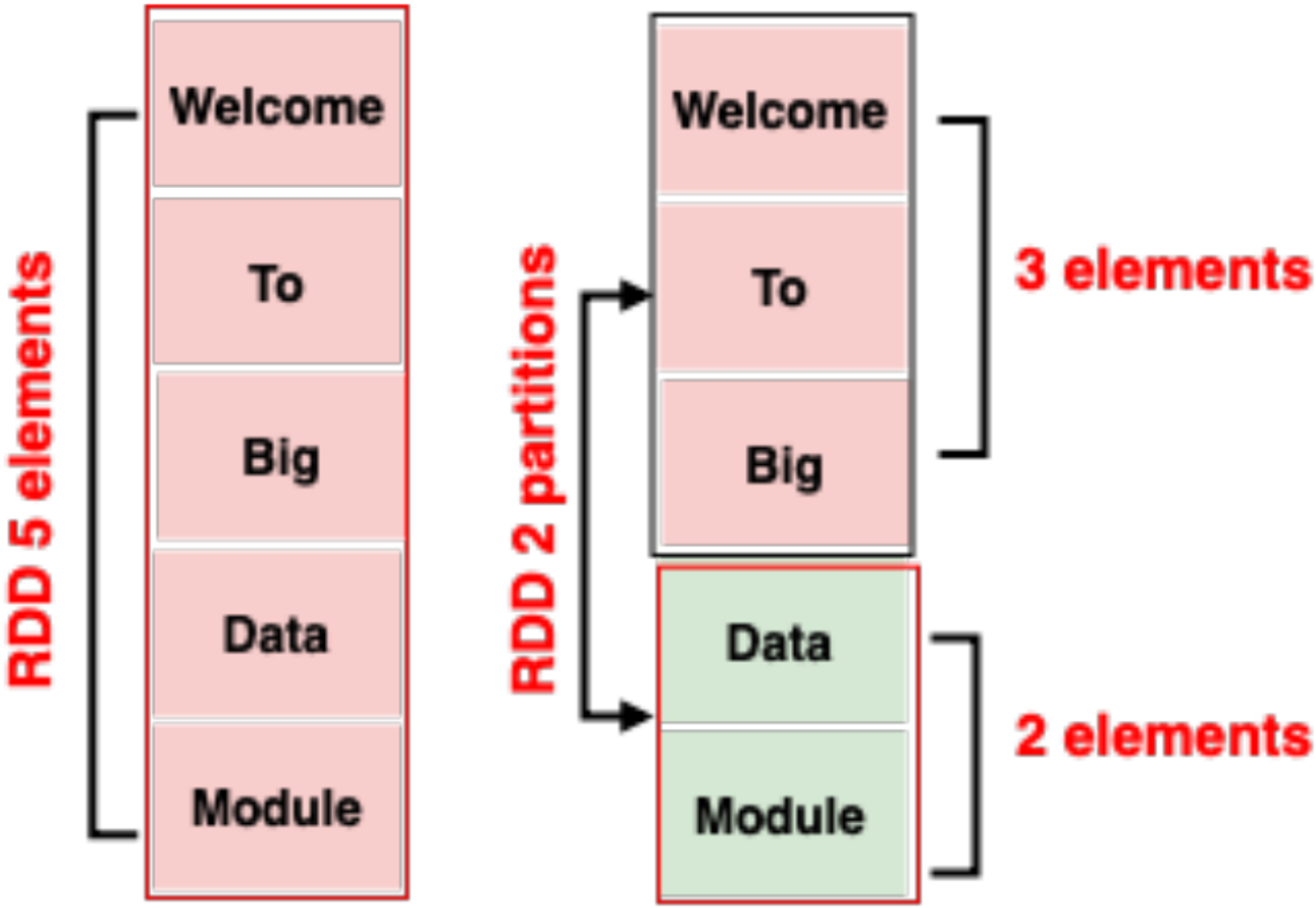
```
1  val f = sc.textFile(inputPath)
2  val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3  w.reduceByKey(_ + _).saveAsText(outputPath)
```

Figure: Word Count in Spark

Resilient Distributed Datasets

- Data is represented in spark as **Resilient Distributed Datasets** (RDDs)
- RDDs are immutable collections of items/objects
- As we saw last week with HDFS, RDDs are are split into partitions and can be stored on the worker nodes of the Spark cluster
- RDD operations can be performed on partitions in parallel
  - **Resilient** because each RDD partition is stored on multiple nodes of a Spark cluster
  - **Distributed** because different RDD partitions are spread across the cluster nodes, and can be processed in parallel

RDD Partitions



Immutability

- RDDs are **immutable** meaning that once they're created they can's be modified.
- This means that every RDD transformation creates a new RDD (just like lists in Haskell)

RDD1

```
input_file = sc.textFile("/usr/local/spark/input.txt")
```

RDD2

```
map = input_file.flatMap(lambda line: line.split("
")).map(lambda word: (word, 1))
```

RDD3

```
counts = map.reduceByKey(lambda a, b: a + b)
```

```
counts.saveAsTextFile("/path/to/output/")
```

RDD4

Creating RDDs

There are two ways to create an RDD:

1. Parallelising an existing collection in your driver program:

```python
# Parallelize in Python
wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
```

```scala
// Parallelize in Scala
val wordsRDD= sc.parallelize(List("fish", "cats", "dogs"))
```

```java
// Parallelize in Java
JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList("fish", "cats", "dogs"));
```

2. Reading from an external storage system or filesystem:

```
# Read a local txt file in Python
input_file = sc.textFile("/usr/local/spark/input.txt")
```

Installing and Using PySpark

Spark is very easy to run locally with Python using PySpark:

- Install PySpark using pip:

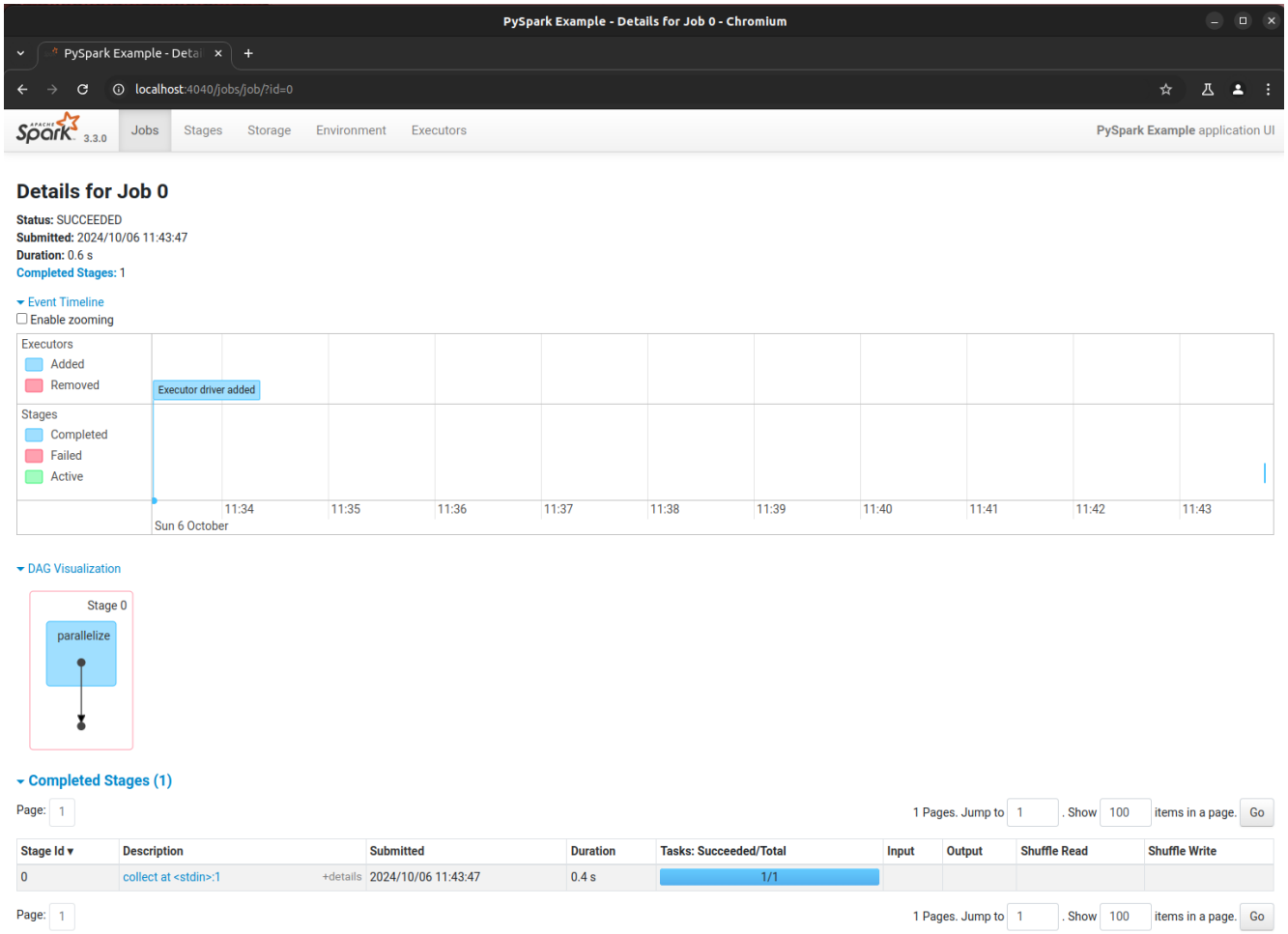
```
$ pip install pyspark
```

- Import PySpark, initialize a SparkContext and run some commands:

```
$ python
>>> from pyspark import SparkContext
>>> sc = SparkContext("local", "PySpark Example")
>>> data = [1, 2, 3, 4, 5]
>>> rdd = sc.parallelize(data)
>>> squared_rdd = rdd.map(lambda x: x * x)
>>> result = squared_rdd.collect()
>>> print(result) # Output: [1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
>>> sc.stop()
```

The Spark Application UI

Visiting **http://localhost:4040/**, we see:



The Spark Stack

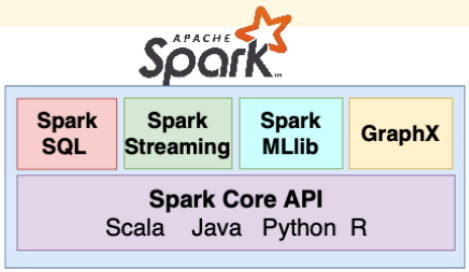
Spark Core, this is the heart of Spark, and is responsible for management functions such as task scheduling.

Spark SQL is a Spark module for structured data processing.

Spark Streaming is a framework for ingesting real-time streaming data from various sources.

Spark MLlib is a library of machine-learning algorithms (e.g., logistic regression, naïve Bayes classification, SVMs, decision trees, random forests, linear regression, and k-means clustering).

GraphX are data structures comprising vertices and the edges for graph and graph parallel computations.



Spark MLLib in Context

- A word of caution: Spark isn't always the best choice for a given machine learning application.
- Here are the alternatives:
  - **Spark MLLib:** Designed specifically to handle large-scale data processing across distributed computing environments. Ideal for big data applications.
  - **Python scikit-learn AKA sklearn:** A python library for machine learning. We'll use this in this course.
  - **R:** a language for statistics and machine learning.
  - **TensorFlow/PyTorch:** frameworks for designing and training deep neural networks.

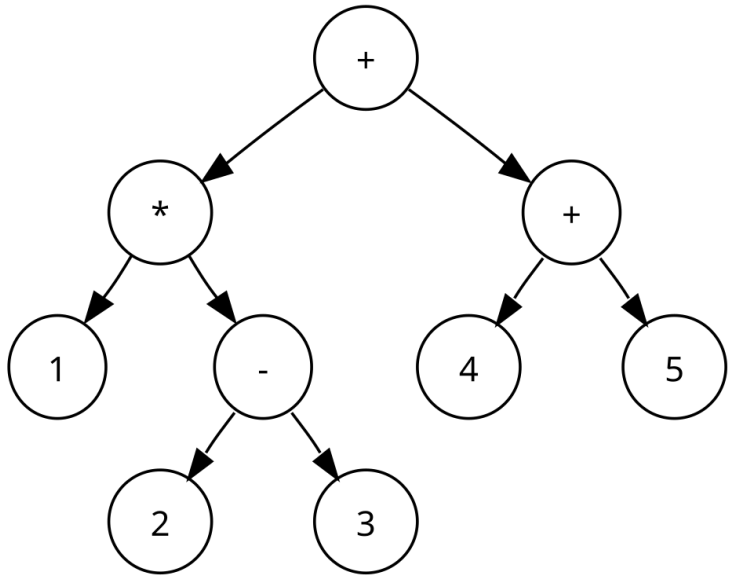
Comparison of Machine Learning Frameworks

Framework	Dataset Size	ML Techniques
Spark	Large-scale	Basic
sklearn	Small to Medium	Basic, Intermediate
R	Small to Medium	Basic, Intermediate, Advanced
TF/PyTorch	Small to Large-scale	Deep Learning

- R is the gold standard for stats and machine learning on small to medium sized datasets (with the exception that deep learning is mostly done in Python with TensorFlow or PyTorch)
- We don't use it in this course because there's a (small) learning curve due to the fact that it provides its own programming language
- However, the language is very easy to learn, so R is ideal for anything most stats and ML projects

Directed Acyclic Graphs and Data Flows

- Spark programming model is based on parallelizable operators, which can be composed together to form a **directed, acyclic graph (DAG)**
  - This is a directed graph with no directed cycles
  - E.g. here's a DAG for the trivial computation $1 * (2 - 3) + 4 + 5$:

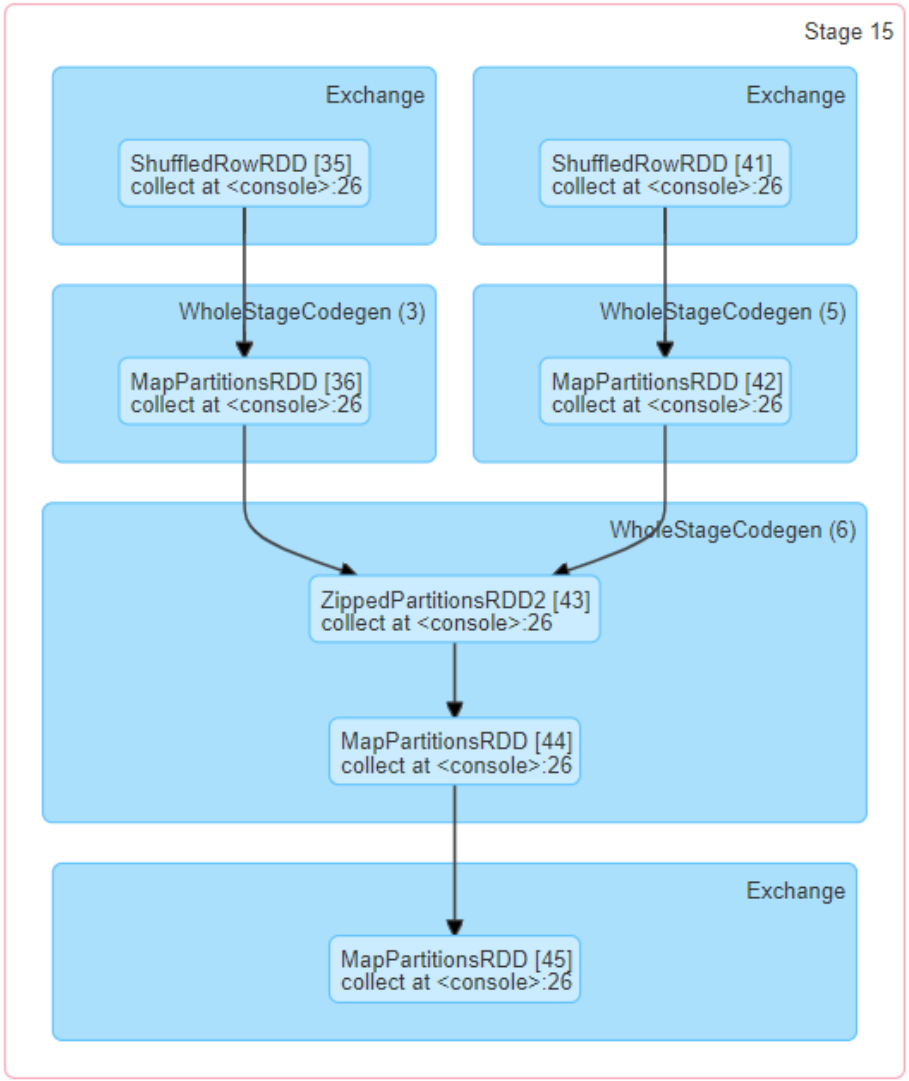


- A **data flow** is composed of any number of data sources, operators, and data sinks. This is done by connecting their inputs and outputs.

DAGs in Spark and Lazy Evaluation

- In Spark, DAGs are used to represent a sequence of computations performed on data, where
  - Each **node** in the graph is an RDD
  - Each **edge** in the graph is a transformation that creates a new RDD from an existing one
- The DAGs are evaluated **lazily** (just like functions in Haskell)
- This allows Spark to optimise the execution plan by reordering and combining transformations.
- This optimisation leads to efficient in-memory execution, which **explains why Spark is so much faster than Hadoop**

Directed Acyclic Graphs in Spark



Spark RDD Operations

Spark RDD support two types of operations: transformations and actions.

i. **Transformations** can be applied to an RDD to create a new RDD
   - These are the DAG edges
   - E.g. `x.map(f)` creates a new RDD by applying `f` to each element in x.

ii. **Actions** compute a result based on an RDD and either return a value or save it to an external storage system (e.g., HDFS).
   - E.g. `x.saveAsTextFile(path)` writes write one RDD element per line to `path`

Spark RDD Operations

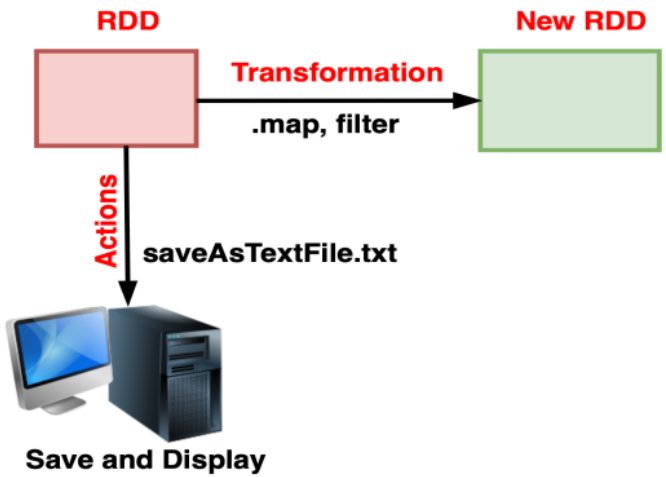
Transformation

```
input_file = sc.textFile("/usr/local/spark/input.txt") map
 = input_file.flatMap(lambda line: line.split("
")).map(lambda word: (word, 1))

counts = map.reduceByKey(lambda a, b: a + b)
```

```
counts.saveAsTextFile("/path/to/output/")
```

Action

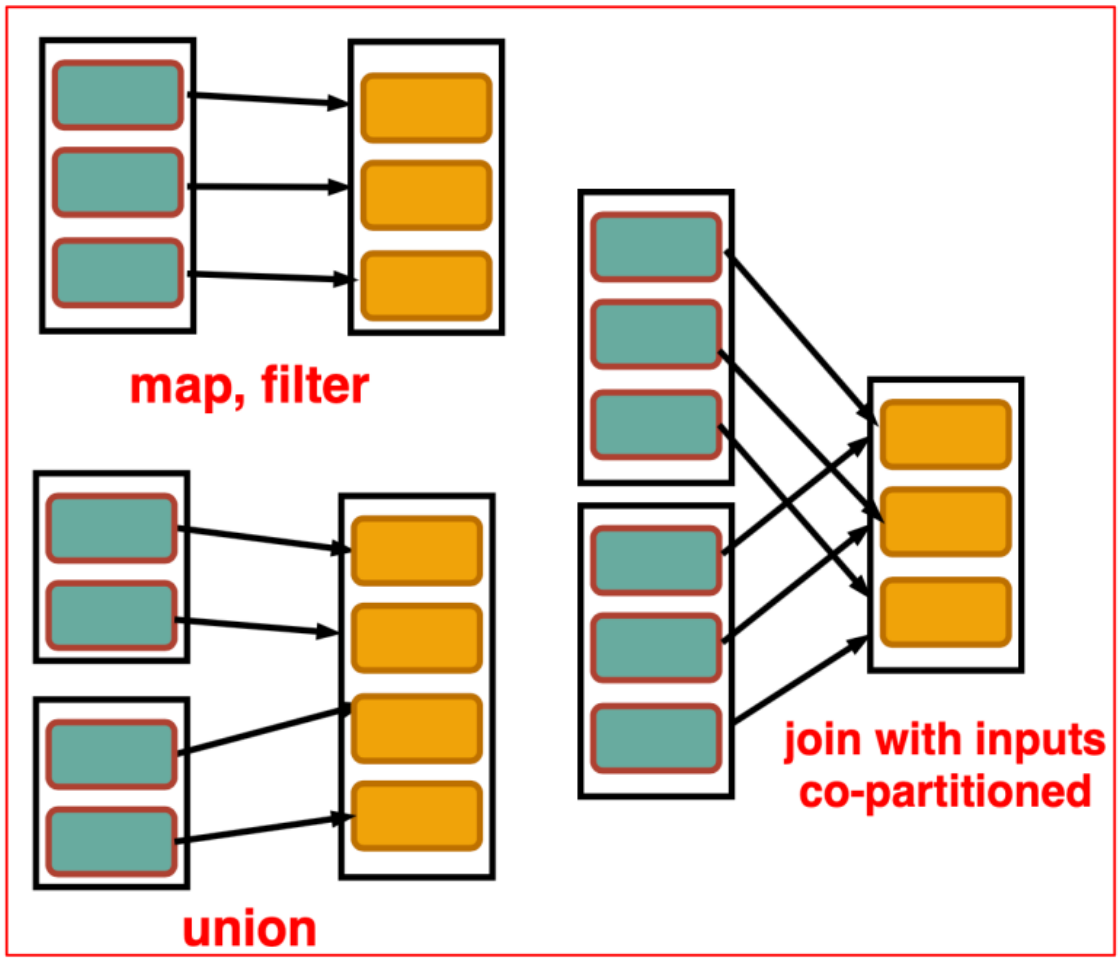


Narrow Vs Wide Transformation

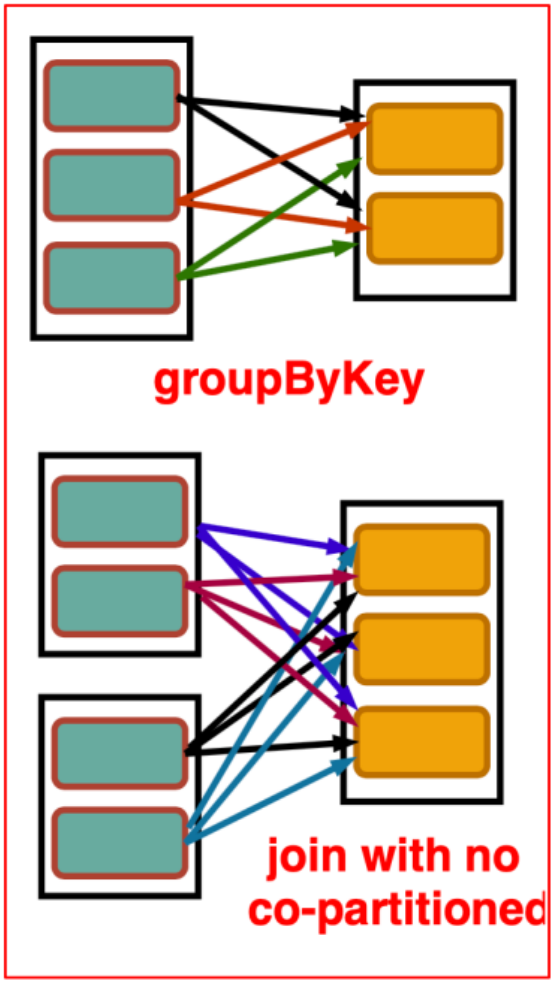
Transformations can be narrow or wide:

- **Narrow Transformation**: each partition of the parent RDD is used by at most one partition of the child RDD
- **Wide (AKA shuffle) transformations**: multiple child RDD partitions may depend on a single parent RDD partition

Narrow Transformations



Wide Transformations



RDD Operations in Spark

	General	Math / Statistical	Set Theory / Relational	Data Structure / I/O
Transformations	• map • filter • flatMap • mapPartitions • mapPartitionsWithIndex • groupBy • sortBy	• sample • randomSplit	• union • intersection • subtract • distinct • cartesian • zip	• keyBy • zipWithIndex • zipWithUniqueID • zipPartitions • coalesce • repartition • repartitionAndSortWithinPartitions • pipe
Actions	• reduce • collect • aggregate • fold • first • take • forEach • top • treeAggregate • treeReduce • forEachPartition • collectAsMap	• count • takeSample • max • min • sum • histogram • mean • variance • stdev • sampleVariance • countApprox • countApproxDistinct	• takeOrdered	• saveAsTextFile • saveAsSequenceFile • saveAsObjectFile • saveAsHadoopDataset • saveAsHadoopFile • saveAsNewAPIHadoopDataset • saveAsNewAPIHadoopFile

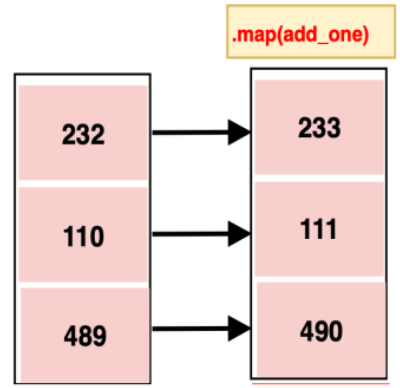
Highlighted operations are hard

map and flatMap

- **map:** takes as input a function which is applied to each element of the RDD and map each input item to another item

```
# transforming each element through a function

nums = sc.parallelize([232, 110, 489])
big_no = nums.map(lambda x: [n + 1])
```



If we want to map every record 1, .map(lambda x: [x,1]

(232, 1), (110,1), (489,1)

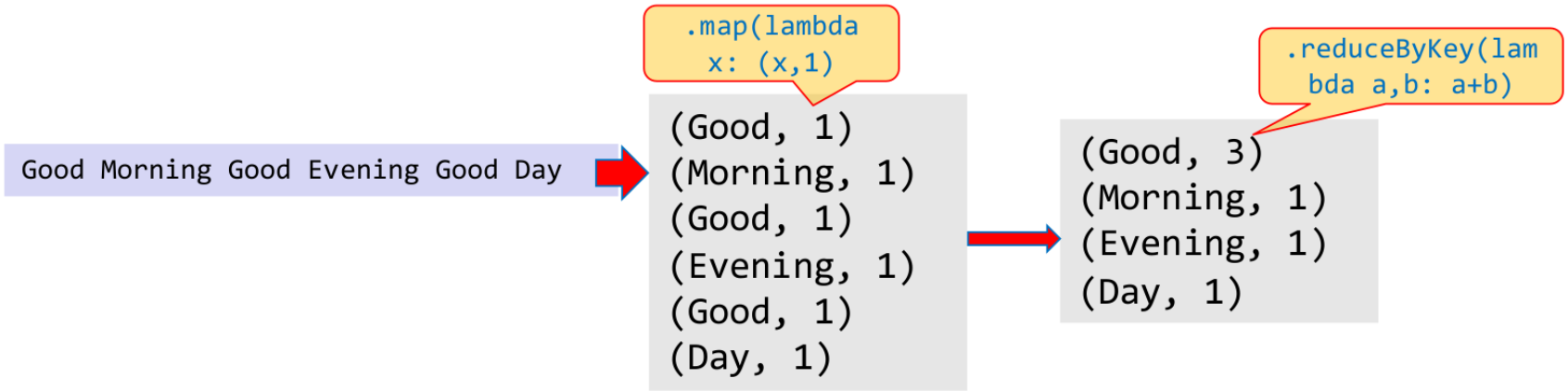
- **flatMap:** generates a new RDD by filtering the source dataset using the specified function

```
Good Morning Good Evening Good Day
Happy Birthday Happy New Year
```

```
lines.flatMap(lambda line: line.split()).collect()
['Good', 'Morning', 'Good', 'Evening', 'Good', 'Day', 'Happy',
'Birthday','Happy', 'New', 'Year']
```

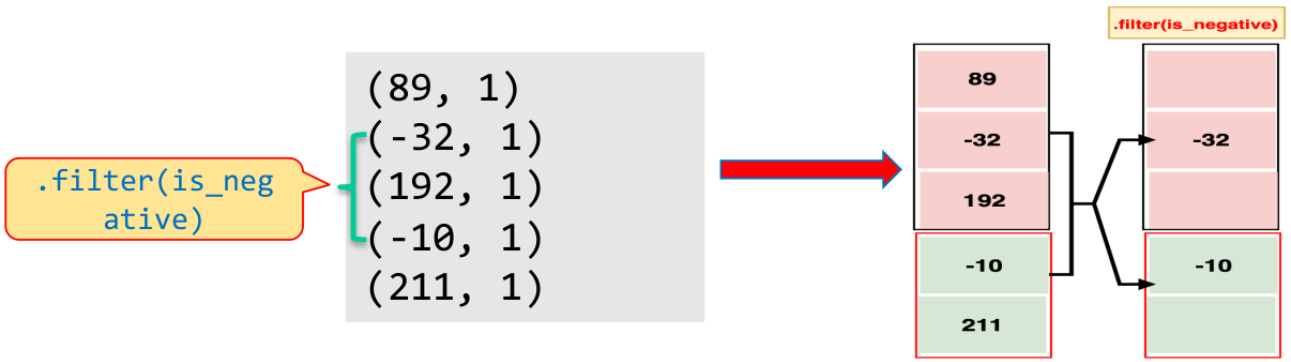
filter

filter generates a new RDD by filtering the source dataset using the specified function:



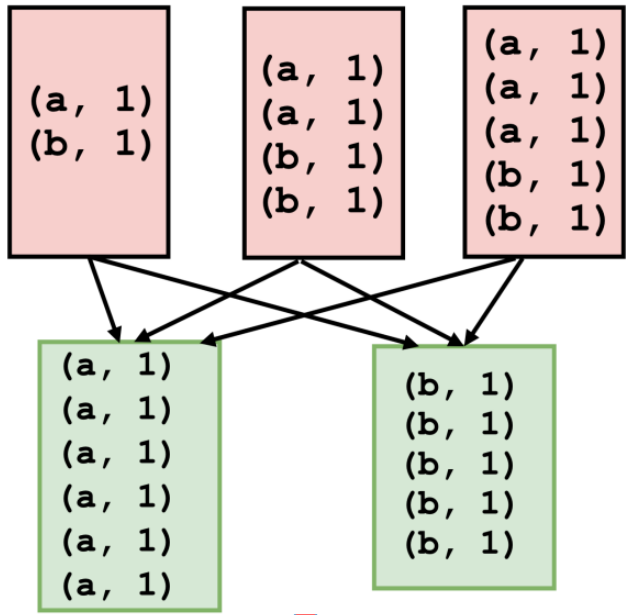

```
rdd6 = rdd5.filter (lambda x : 'a' in x[1])
```

="(1, 'Day')" that has a value 'a'.



groupByKey

groupByKey groups the data based on a key and returns an RDD of grouped data:



Note
For an aggregation (e.g., **sum or average**) over each key, using reduceByKey or aggregateByKey will provide much better performance.

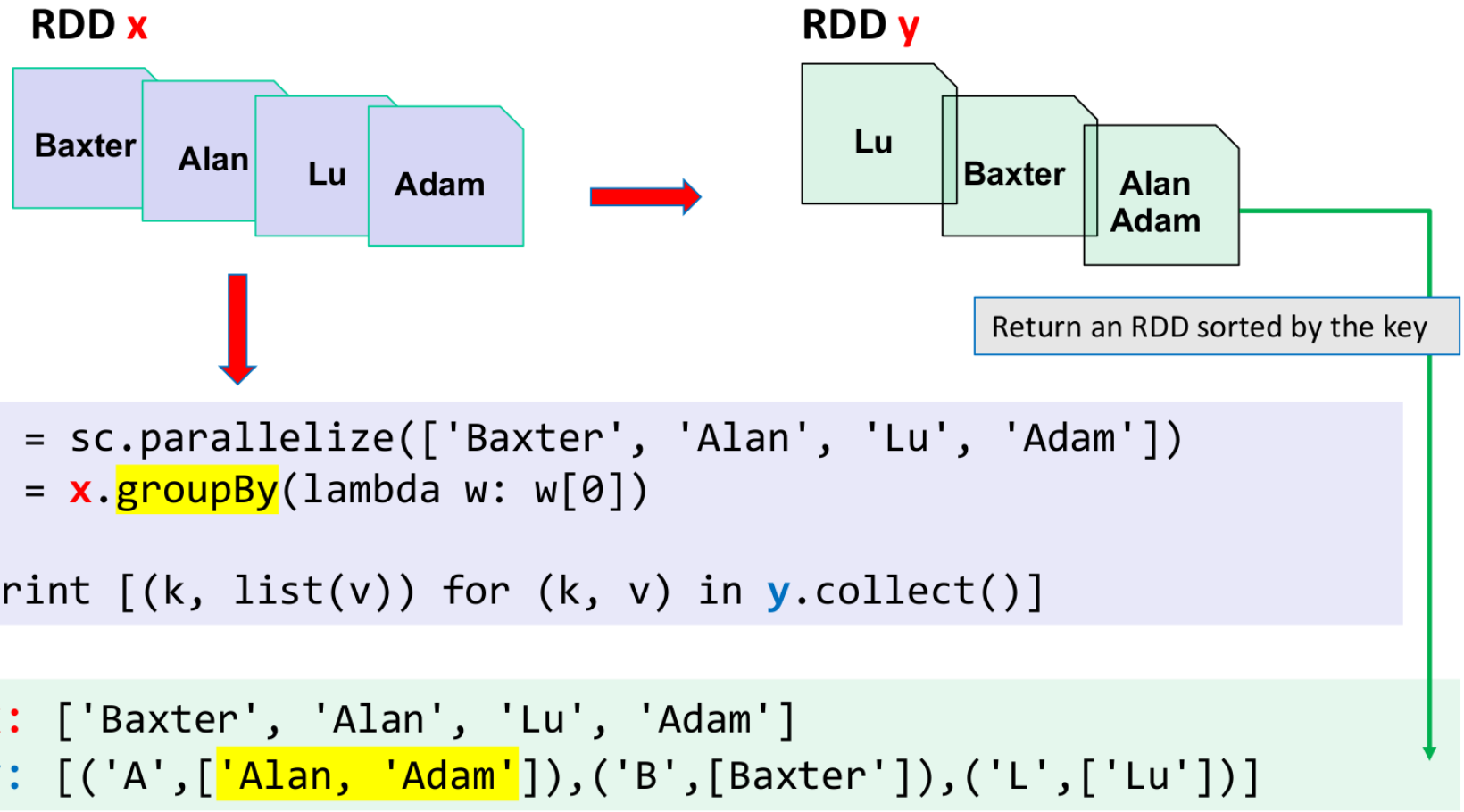
Group values with the same key

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b", 1), ("b", 1), ("a", 1), ("a", 1), ("a", 1), ("b", 1), ("b", 1)])
```

```
sorted(rdd.groupByKey().mapValues(list).collect()) [('a', [1, 1, 1, 1, 1, 1]), ('b', [1, 1, 1, 1, 1])]
```

groupBy

groupBy groups the data based on a user-provided function that is applied to each element in the RDD:



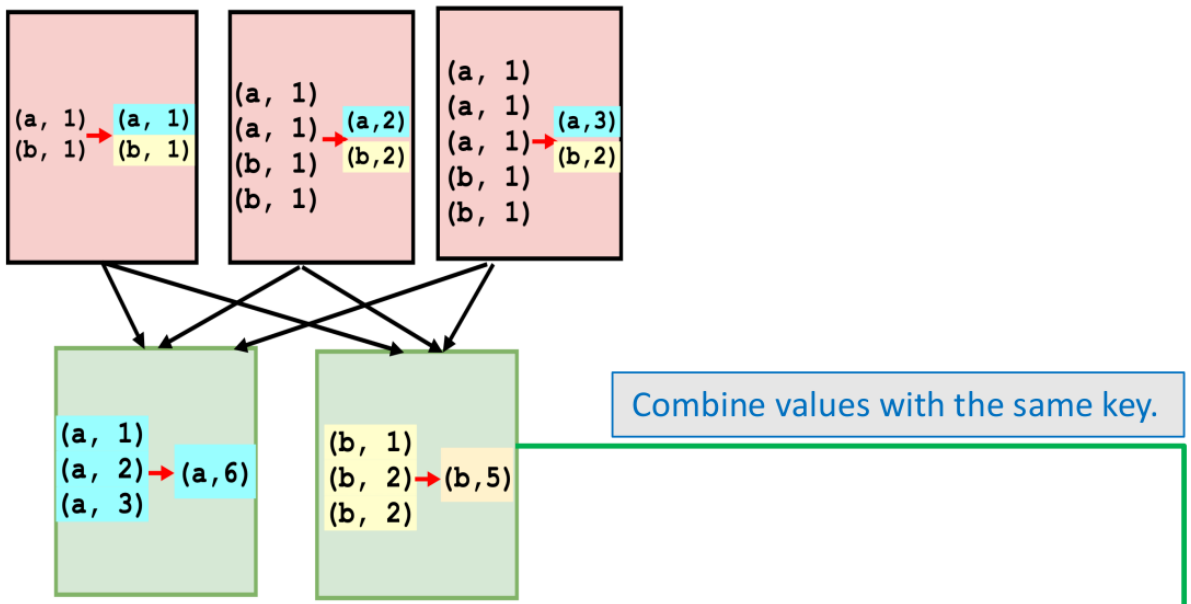

```
x = sc.parallelize(['Baxter', 'Alan', 'Lu', 'Adam'])
y = x.groupBy(lambda w: w[0])

print [(k, list(v)) for (k, v) in y.collect()]
```

```
x: ['Baxter', 'Alan', 'Lu', 'Adam']
y: [('A',['Alan, 'Adam']),('B',[Baxter']),('L',['Lu'])]
```

reduceByKey

reduceByKey aggregates elements of the RDD using a specified associative and commutative function:

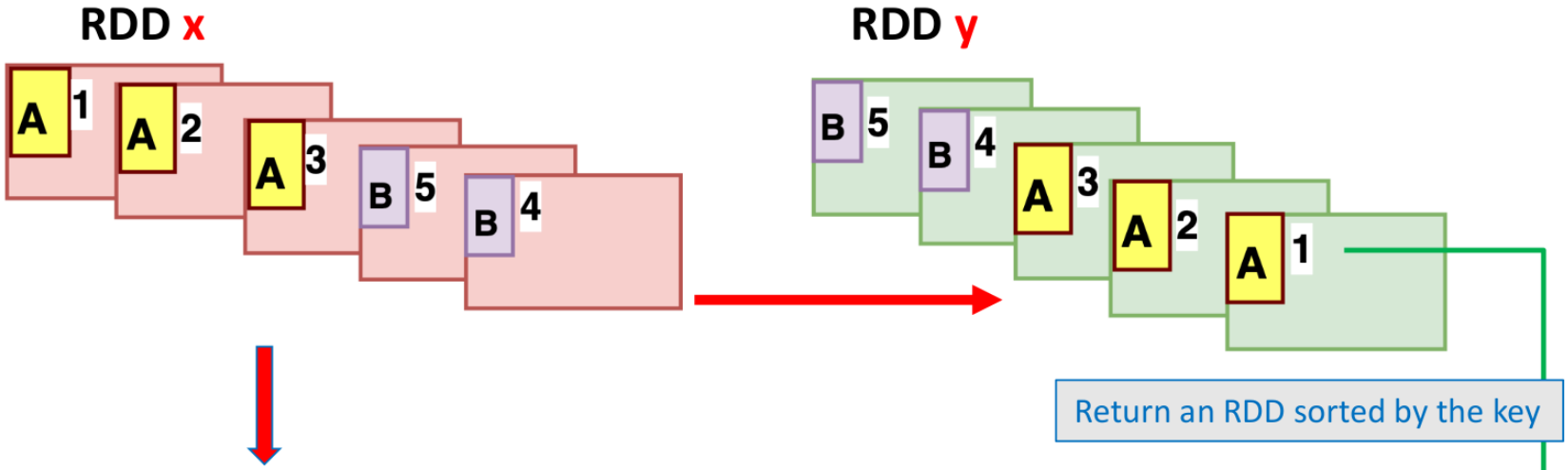



```
from operator import add
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b",
1), ("b", 1), ("a", 1), ("a", 1), ("a", 1), ("b", 1), ("b", 1)])
```

```
sorted(rdd.reduceByKey(add).collect()) [('a', 6), ('b', 5)]
```

sortByKey

sortByKey sorts elements by key:

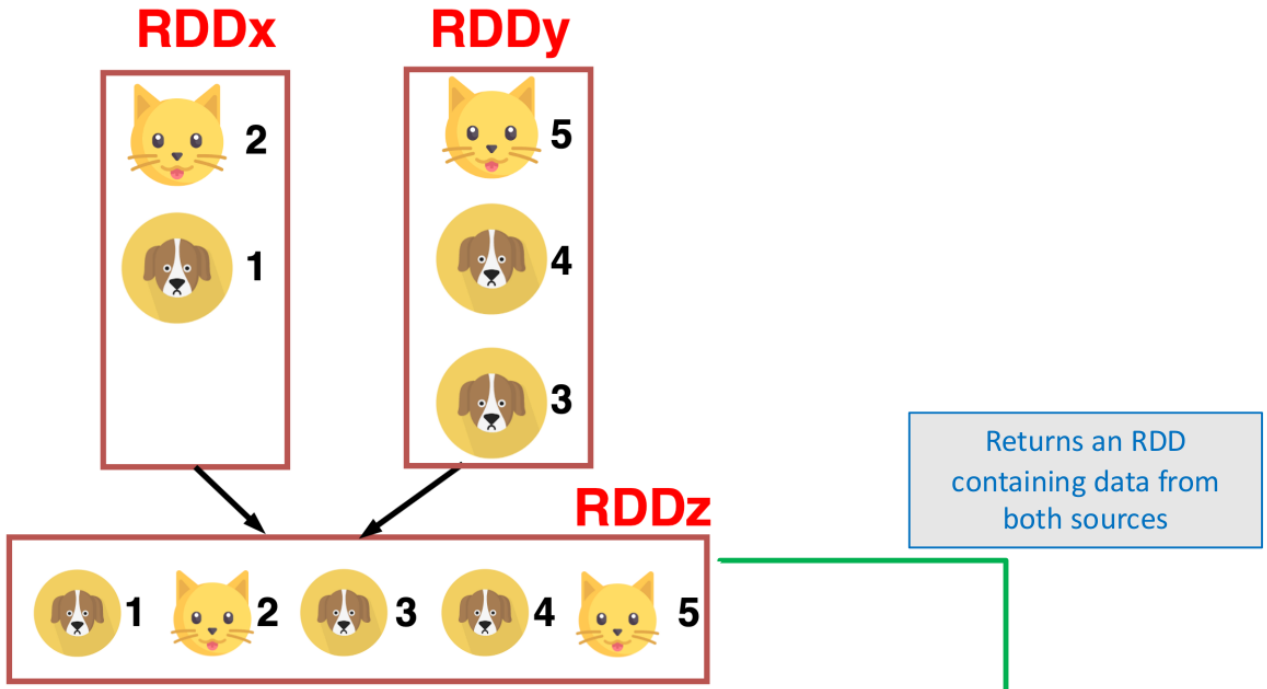



```
x = sc.parallelize([('B',4),('B',5),('A',3),('A',2),('A',1)])
y = x.sortByKey()
print(x.collect())
print(list((j[0], list(j[1])) for j in y.collect()))
```

```
x: [('B', 4),('B', 5),('A', 3),('A', 2),('A', 1)]
y: [('A', 1), ('A', 2), ('A',3), ('B', 4), ('B', 5)]
```

union

union returns the union of two RDDs:

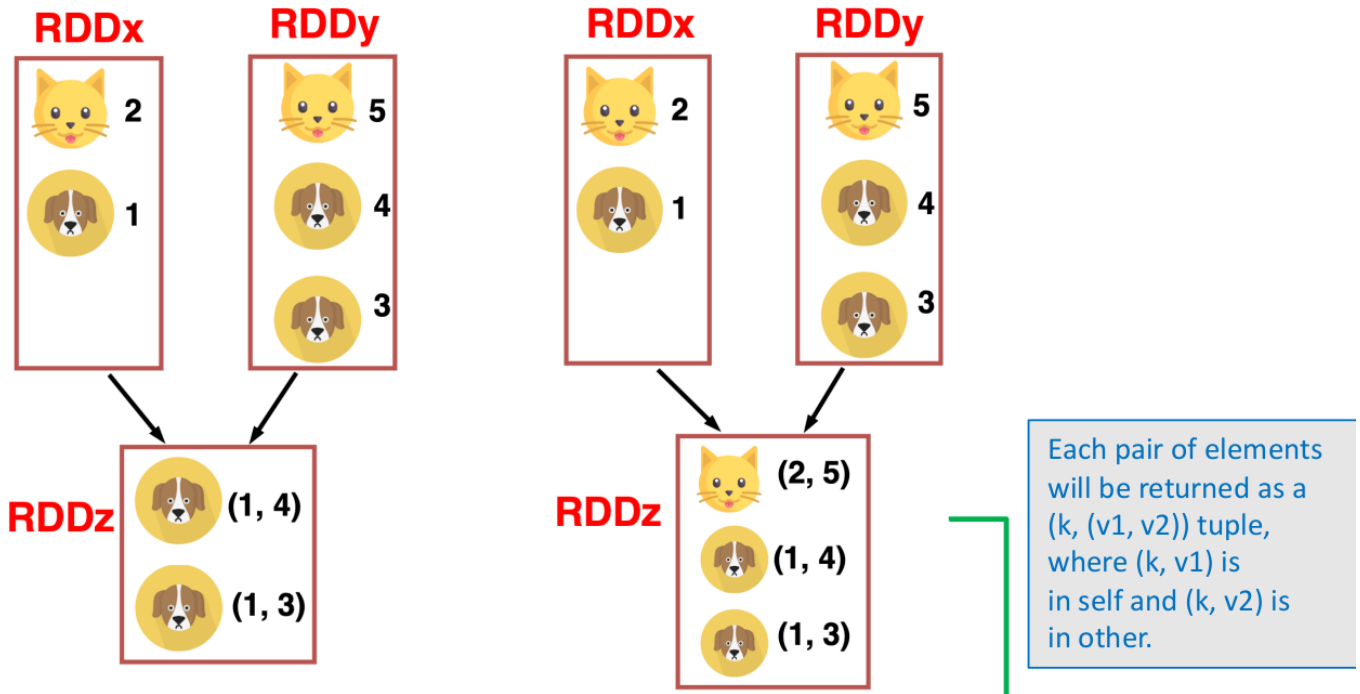



```
x = sc.parallelize([("dog",1), ("cat",2)])
y = sc.parallelize([("dog",3), ("dog",4), ("cat", 5)])
```

```
z =([("dog",1), ("cat",2), ("dog",3), ("dog",4), ("cat", 5)])
```

join

join returns the inner join of two RDDs:

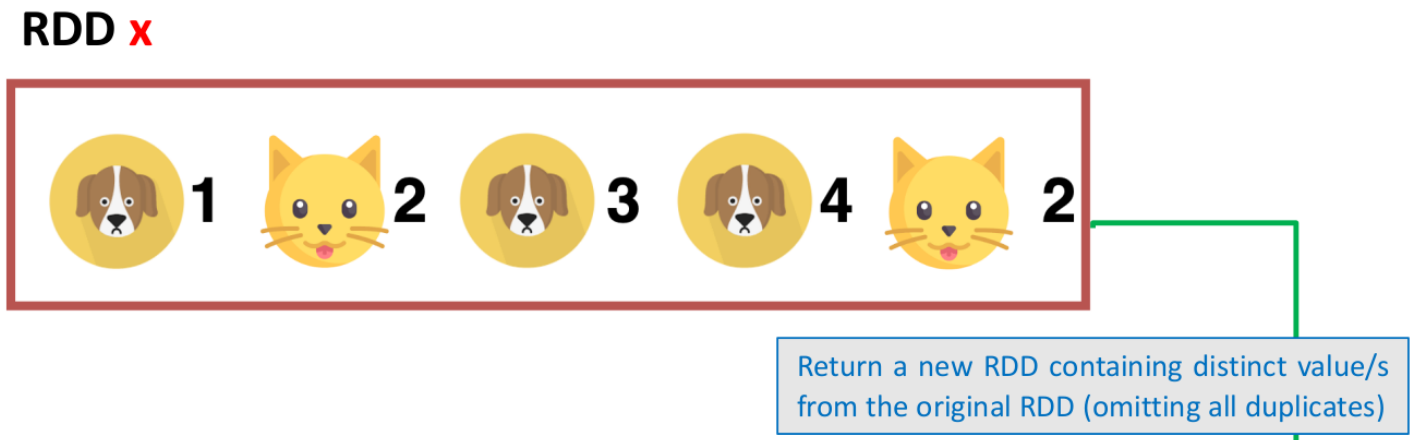



```
x = sc.parallelize([("dog", 1), ("cat", 2)])
y = sc.parallelize([("dog", 3), ("dog", 4), ("cat", 5)])
sorted(x.join(y ).collect())
```

```
z: [('dog', (1,3)), ('dog', (1,4), ('cat', (2, 5))]
```

distinct

distinct returns an RDD with distinct elements:




```
x = sc.parallelize([("dog", 1), ("cat", 2), ("dog", 3), ("dog", 4),
("cat", 2)])
y = x.distinct()
print(y.collect())
```

```
z = ([("dog", 1), ("cat", 2), ("dog", 3), ("dog", 4)])
```

Actions

- **reduce** AKA **aggregate** applies a binary operator to an initial state and all elements in an RDD, and reduces this to a single state. The final state is converted into the final result by applying a finish function.
  - Note that reduceByKey is a transformation because it returns an RDD of keys and transformed values.
  - However, reduce is an action because it returns a single value rather than an RDD
- **fold** is similar to reduce. It aggregates the elements of each partition, and then the results for all the partitions. In contrast to reduce, it uses a neutral “zero value”.
- **collect**: Returns all the elements of the dataset as a list (NB: avoid using with large datasets because it will dump the entire dataset into your local computer’s memory).

Actions

- **first** returns the first element in this RDD.
- **take(n)** returns the first n elements from the dataset.
- **forEach(func)** applies a function to all elements of the RDD, generally used for side effects (e.g., saving data to a database). Similar to map, except that is is an action, so it does not produce a new RDD.
- **top(n)** returns the top n elements of the RDD, based on the default or specified ordering.

Actions

- **count** return the number of elements in this RDD.
- **takeSample(withReplacement, num, [seed])** returns a fixed-size sampled subset of this RDD.
- **max** returns the maximum element in the RDD (must be comparable).
- **min** returns the minimum element in the RDD (must be comparable).
- **sum** computes the sum of elements in the RDD.
- **histogram(buckets)** computes a histogram of the elements in the RDD, dividing them into buckets.
- **mean** returns the mean of the dataset.
- **variance** returns the variance of the dataset.
- **stdev** returns the standard deviation of the dataset.

DataFrames

DataFrames are distributed collections of rows under named columns.

- Introduced to Spark in 2015
- Similar to Excel tables or Python DataFrames
- Build on top of RDDs
- Immutable in Nature: You can create a DataFrame, but you will not be able to change it. A DataFrame can be transformed just like an RDD.
- Lazy Evaluations: A task is not executed until an action is performed.
- Distributed: DataFrames, just like RDDs, are distributed in nature.

Creating a DataFrame in PySpark

```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession.builder \
...     .appName("PySpark Example") \
...     .getOrCreate()
>>> data = [
...     ("Alice", 25, "Data Scientist"),
...     ("Bob", 30, "Software Engineer"),
...     ("Catherine", 29, "Doctor")
... ]
>>> columns = ["Name", "Age", "Occupation"]
>>> df = spark.createDataFrame(data, columns)
>>> df.show()
+---------+---+-----------------+
|     Name|Age|       Occupation|
+---------+---+-----------------+
|    Alice| 25|   Data Scientist|
|      Bob| 30|Software Engineer|
|Catherine| 29|           Doctor|
+---------+---+-----------------+
```

How to Create DataFrames?

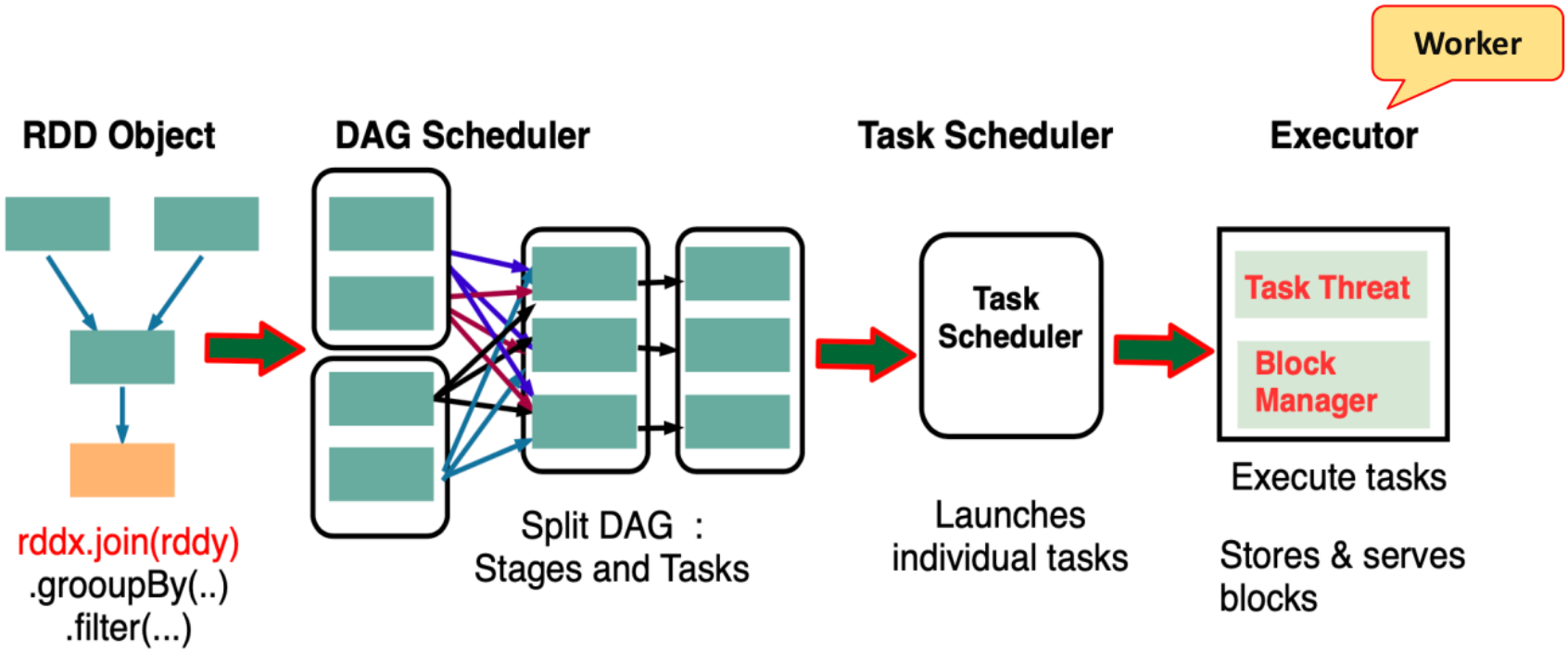
- **Using different data formats:** Data can be loaded from JSON, CSV, RDBMS, XML, or Parquet.
- **Loading from an existing RDD:** A DataFrame can be created from an already existing RDD.
- **Programmatically specifying a schema:** You can specify the schema programmatically to create a DataFrame.

Spark Components

- **Task:** Individual unit of work sent to one executor over a sequence of partitions.
- **Job:** Set of tasks executed as a result of an action.
- **Stage:** Set of tasks in a job that can be executed in parallel at the partition level.
- **RDD:** Parallel dataset with partitions.
- **DAG:** Logical graph of RDD operations.

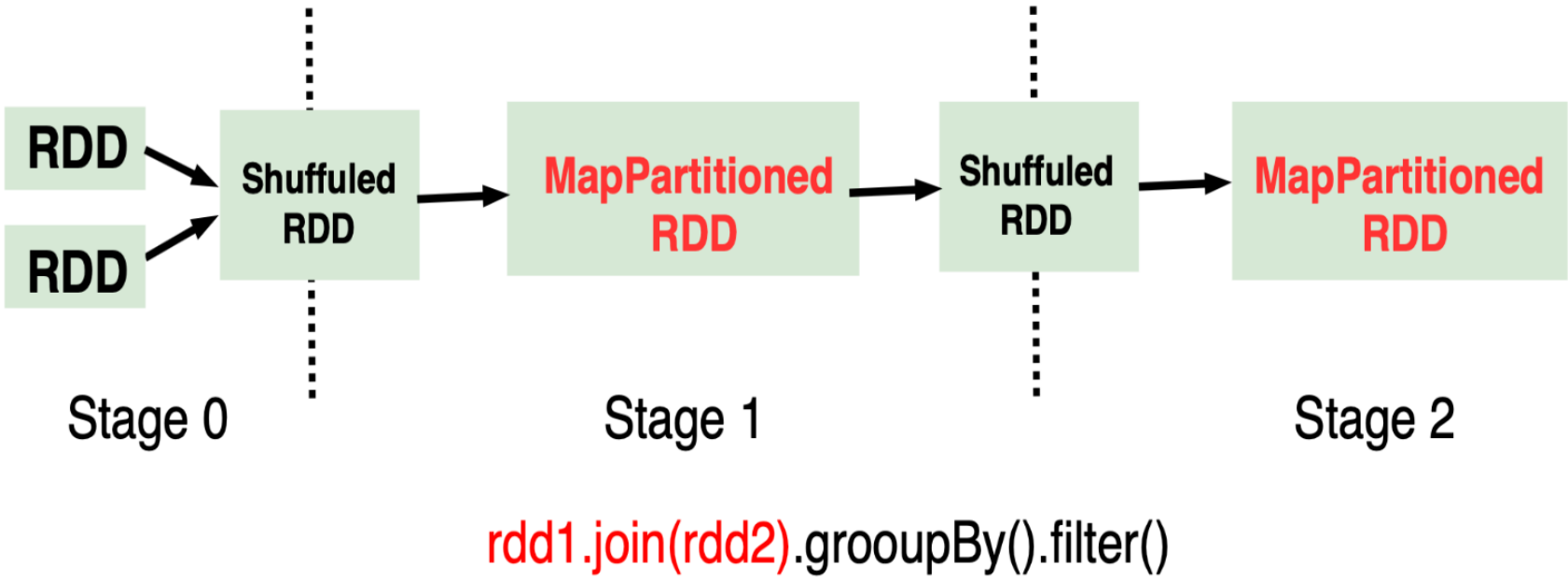
Job Scheduler

The job scheduler transforms a logical execution plan (i.e., RDD lineage of dependencies built using RDD transformations) to a physical execution plan (using stages).

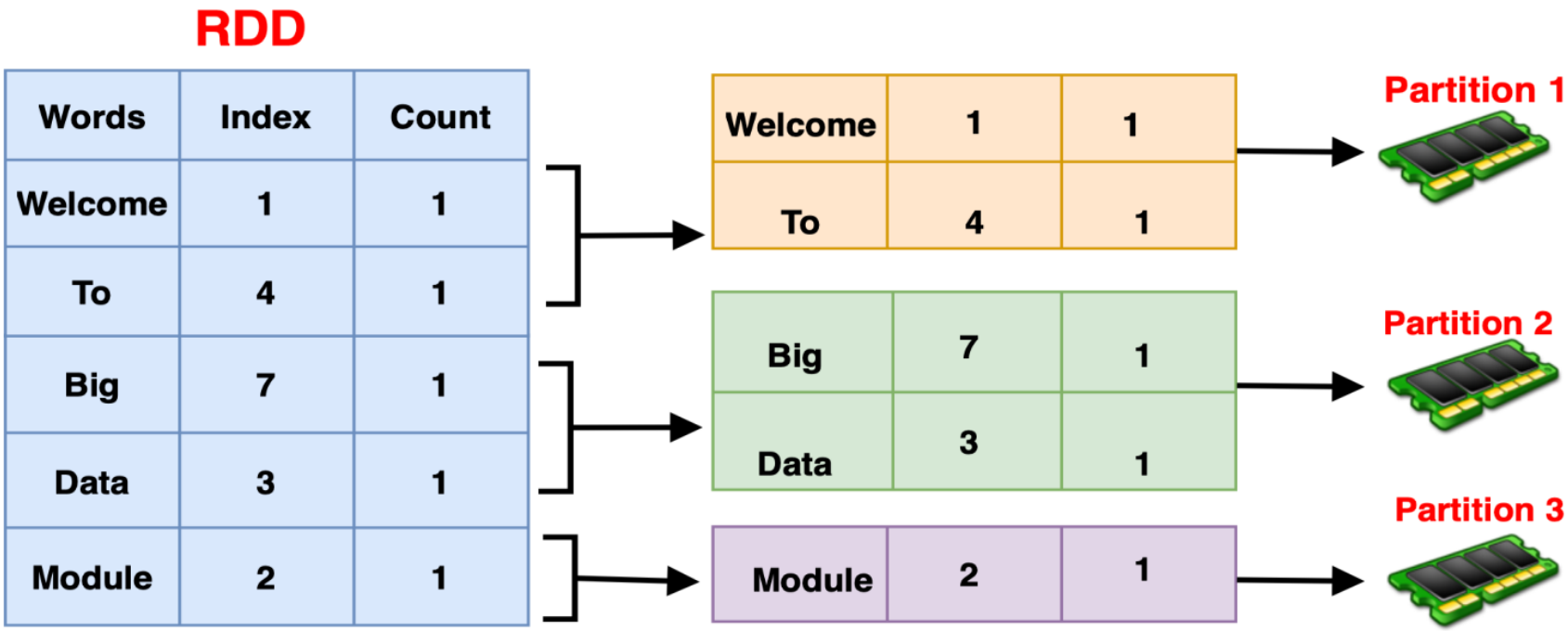


Abstraction 2: RDD Stages

- Each **Spark stage** will dynamically allocate executors.
- Stages contain many neighbouring (in the lineage graph) transformations and actions.



RDD


RDD
Words
Index
Count
Welcome
1
1
To
4
1
Big
7
1
Data
3
1
Module
2
1
Welcome
1
1
To
4
1
Big
7
1
Data
3
1
Module
2
1
Partition 1
Partition 2
Partition 3

Spark Architecture

Spark architecture is made up of a **driver** program and a set of **executor** programs.

Driver:

- Runs the `main()` function, located on a node in the cluster.
- Responsible for:
  - Maintaining information about the Spark applications.
  - Responding to a user's program or input, analyzing, distributing.
  - Scheduling work across the executors.

Executors:

- Perform all the data processing as assigned by the driver.
- Read from and write data to external sources.
- Store the computation results in-memory, cache, or on hard disk drives.
- Interact with the storage systems.

Spark: Standalone Cluster

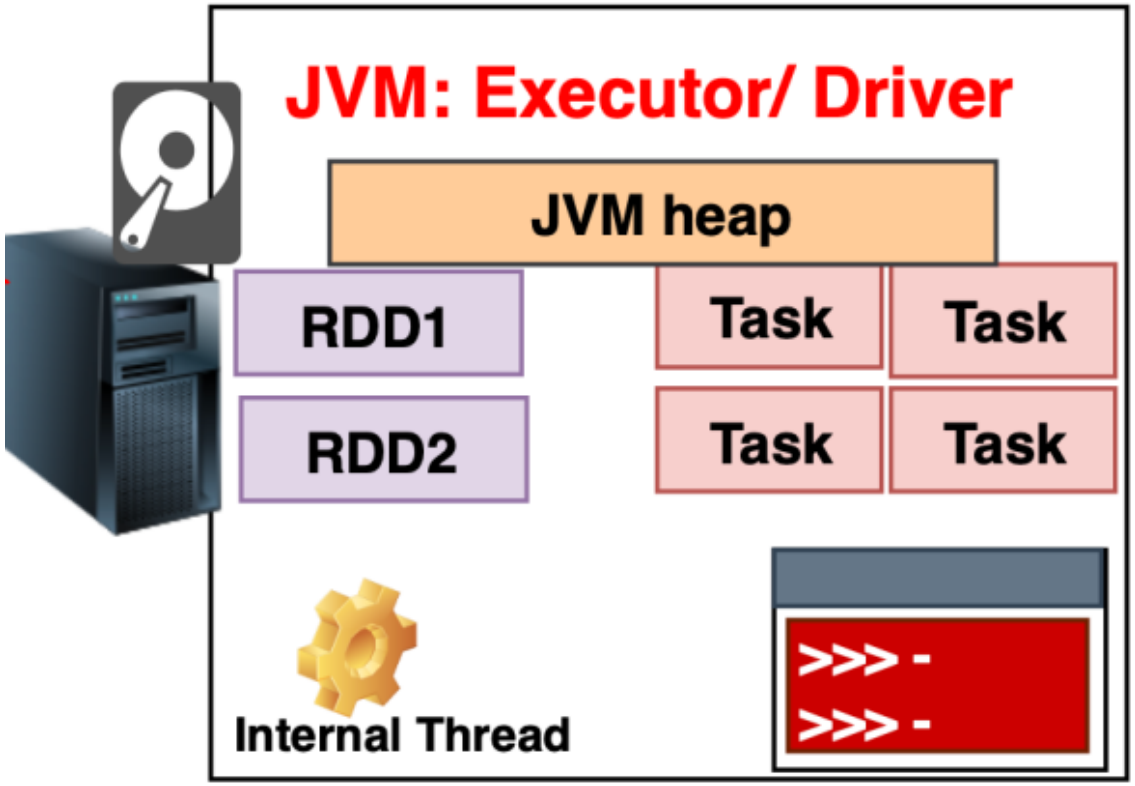
- Spark **local mode** and Spark **local cluster mode** are special cases of running Spark on a single machine.
  - **Local mode:** Runs Spark with one worker thread.
  - **Local[N]:** Runs Spark with N worker threads (like a thread pool).
  - **Local[*]:** Runs Spark with as many worker threads as there are logical cores on your machine.
- This mode is useful for running Spark applications on a local machine for testing or small workloads.

SparkContext Example in Local Mode

```
import pyspark
sc = pyspark.SparkContext('local[4]')
txt = sc.textFile('file:///usr/share/doc/python/copyright')
python_lines = txt.filter(lambda line: 'python' in line.lower())
print(python_lines.count())
```

- The SparkContext is created in local mode with 4 worker threads.
- The `textFile` method reads a file.
- The `filter` operation filters the lines containing the word "python."
- The `count` operation returns the number of matching lines.

Spark Standalone Cluster

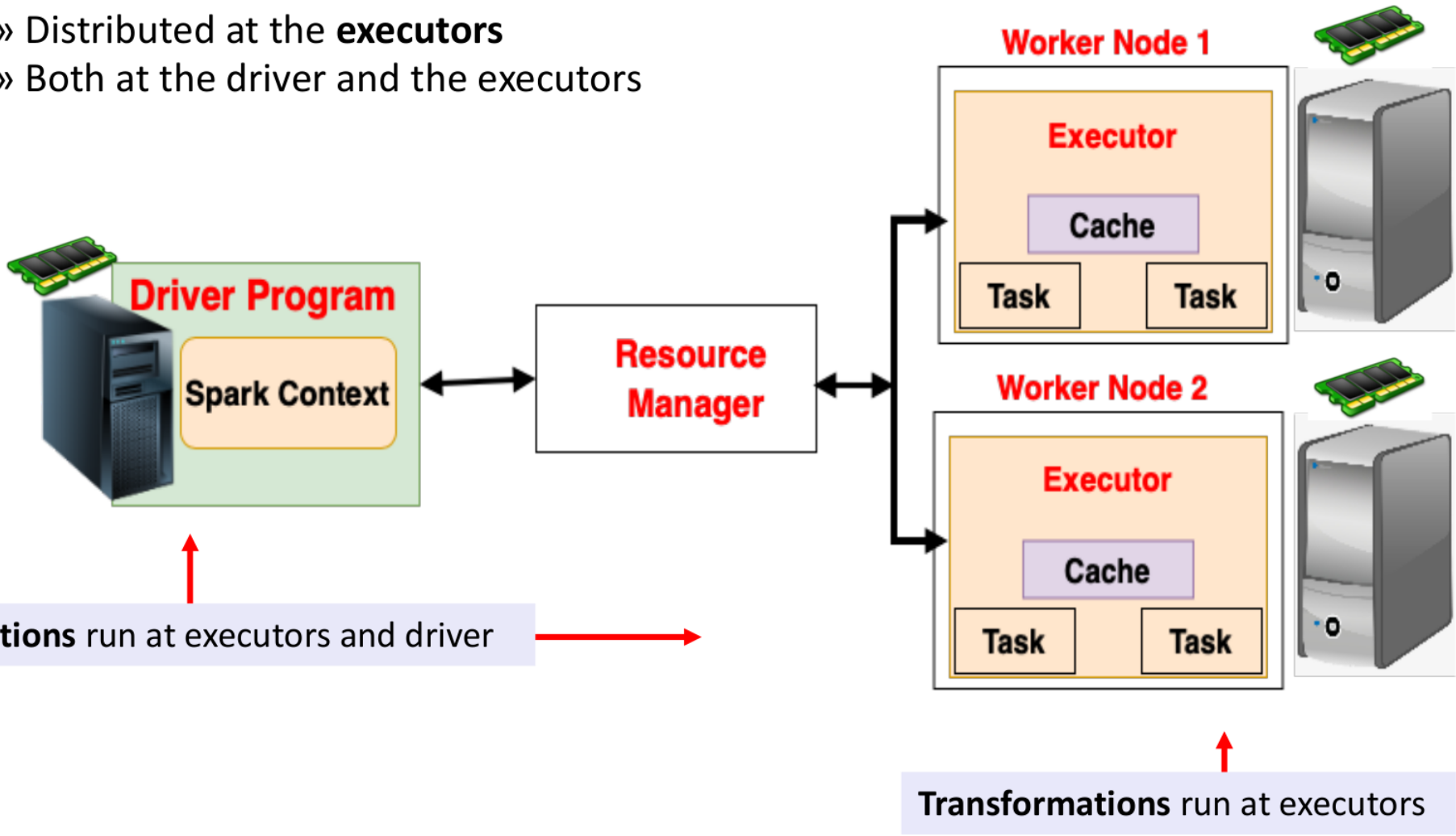


Cluster Mode

Where does code run?

» Distributed at the **executors**

» Both at the driver and the executors



References

- Feng, W. (2019). Learning Apache Spark with Python
- https://databricks.com/glossary/spark-api
- https://realpython.com/pyspark-intro/
- https://www.cs.princeton.edu/courses/archive/fall13/cos518/papers/spark.pdf