Introducton to Data Engineering 1

Paul Blondel

UTSEUS, Shanghai

March 30st, 2021

'You can have data without information, but you cannot have information without data.' Daniel Keys Moran

1. Introduction
2. Memory
   - Different memories for different usages
   - How it works?
3. Data
   - Data, types and properties
   - Units
   - Data types
   - Encoding characters
   - How to deal with images, audio and video
4. File formats
5. Databases
6. Data compression

Since the beginning of computers, data is a key component:

- Operating systems are stored as data in the hard-drive and loaded in the main memory
- Computer users' data are stored on hard-drive, optical discs, etc

Because, with computers, data can not only be read but also easily written and erased, one can...

- ...install/uninstall a software
- ...update the OS
- ...write a word document
- ...have a custom system and a personal experience

The software era

Compared to an old-school calculator: the flexibility to access and edit the data made a great difference and opened the era of software creation.

However nowadays, data is more than just your operating system, all your software and some word documents...

There have been a series of recent discoveries and breakthroughs making possible the **extraction, the storage, the pre-processing and the processing, the analysis and the building of statistical and Machine Learning models on bigger and bigger amounts of data**.

The era of data

Brand new applications are nowadays unlocked thanks to that: making the engineering of data a top priority in most companies worldwide.

First thing first!

Before manipulating petabytes of data, let's understand how to change information into data, and first, how to store data into... the memory!

Different types of memory:
- CPU registers
  - Stores data directly used by the CPU (instructions, memory addresses, program counter, accumulator, etc.)
- Cache Memory
  - Temporary stores copies of the data from frequently used and/or nearby main memory locations
- Main Memory (Random-access memory or RAM)
  - Stores running softwares and opened documents
- Mass Storage Memory
  - Stores installed softwares and all documents
  - Two main technologies on the market:
    - Hard-disk drive (HDD) (invented in 1954)
    - Solid-state drive (SSD) (invented in 1978)
- Optical discs
- GPU memory units

- CPU Registers
  - Y1, Y2, PSW, Z: registers used to store operation values, store computed value and remain
  - RI, CO: registers storing instructions
  - RAD: register storing memory address
  - RDO, general registers: registers storing data

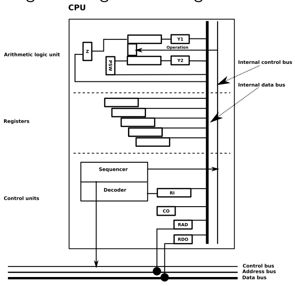


Figure: Simple CPU architecture

- Cache Memory
  - Located between the Computer Processing Unit (CPU) and the main memory
  - Help reducing the number of accesses to the main memory, reducing the overall memory access time
    - By temporary storing copies of the data from frequently used and/or nearby main memory locations

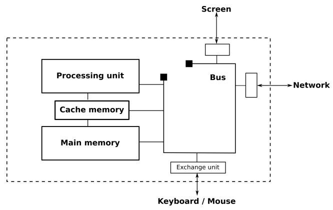


Figure: Von Neumann computer architecture

- RAM (Main Memory)
  - All the data is stored in an array of storage cells
  - 1 bit = 1 storage cell (DRAM: bit stored in a capacitor)
  - Volatile memory: need electric power for storage

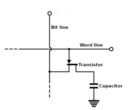


Figure: DRAM memory cell

- HDD (Mass Storage Memory)
  - All the data is stored in disks of ferromagnetic material
  - 1 bit = direction of magnetic field lines

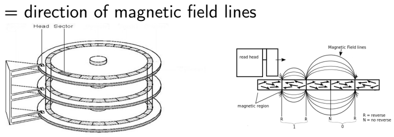


Figure: Left: disks of ferromagnetic material right: magnetic field lines

- SSD (Mass Storage Memory)
  - All the data is stored in an array of floating-gate transistors
  - No mechanical moving component = faster
  - Non-volatile memory: electrons trapped in the floating-gate (surrounded by oxide layers)

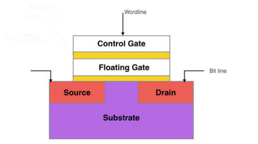


Figure: Floating gate transistor cell

- Optical Discs
  - Mechanical moving component, disc moving and laser head reading
  - Non-volatile memory: bits are "physically" created ("holes" or chemical reactions)

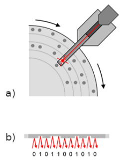


Figure: Reading optical discs

  - The laser arm and the disc move alike the HDD mechanism (a)
  - Bits are represented by pits etched on the disc itself (b)

Summary of the main properties of computer memories:

	CPU Registers	Cache	RAM	HDD / SSD	Optical Discs
Access speed	Ultra fast (1 or few CPU cycles)	Very fast (20000-200000MO/s)	Fast (2000-26600MO/s)	Slow (50-500MO/s)	Very Slow (0.150-3MO/s)
Storage capacity	Ultra low (32/64 bits)	Very low (512KB-8MB)	Low (4-32GB)	High (256GB - 1TB)	Moderate High (700MO-25GB)
Storage property	Volatile	Not Volatile	Volatile	Not Volatile	Not Volatile

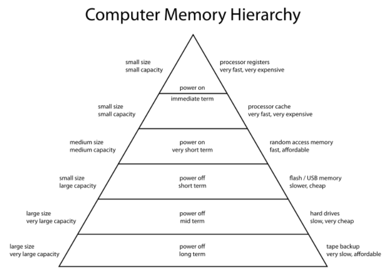


Figure: The Memory Hierarchy: a trade-off between storage capacity and speed

Notes

To optimize the speed of execution: top of the pyramid memories should be use as much as possible! (Require to design your program accordingly).

GPU memories: a complex assembly:

- Registers: very fast access, thread, few space
- Local Memory: slow access, thread, more space
- Shared Memory: fast access, block, more space
- Global Memory: slow access, all blocks, more space
- Constant Memory: fast access, all blocks, more space
- Texture Memory: fast access, all blocks, read-only

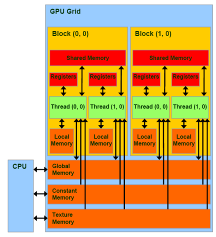


Figure: Memory units of Nvidia GPUs

GPU memories: A more complex trade-off

With GPU memory units: a trade-off storage capacity VS access speed VS accessibility. That's why coding optimal code for GPU is hard.

What is data?

Data is the result of a measurement

Examples: The diameter of the sun is **1.39 million kilometers**, the temperature of the room is **30°C**, and so on.

A data ...

- ... is **simple** or **complex** (composed of simple data)
- ... has a **type** (integer, character, and so on)
- ... has a set of authorized **instructions**

Examples: 'hello' is a complex data (composed of simple character data), 30°C is a simple integer data. 'hello' * 30 is not authorized, but 2 * 30 is.

The most common units are:

- 1 bit (0 or 1)
- 1 byte (or octet) = 8 bits
- 1 KB (or KO) = 1000 bytes[1]
- 1 MB (or MO) = 10^6 bytes = 1000 KB
- 1 GB (or GO) = 10^9 bytes = 10^6 KB = 1000 MB
- 1 TB (or TO) = 10^{12} bytes

Exemple: 'hello' is stored on 5 bytes (each character is stored on 1 byte, see later in this course why it is the case...).

[1] 1 KB is sometimes defined as equal to 1024 bytes, but this definition is no longer valid. In 1998, the International Electrotechnical Commission stated that 1 KB = 1000 bytes and that 1024 bytes = 1 KiB (kibibyte).

Data types in C/C++:

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
long int	8bytes	-2,147,483,648 to 2,147,483,647
float	4bytes	1.2E-38 to 3.4E+38
double	8bytes	2.3E-308 to 1.7E+308

```
#include <iostream>
int main(void) {
        std::cout << sizeof(char) << std::endl;
}
```


C/C++ Structure (aggregation of variables of different types)
Size of a structure = sum of variables sizes (1 byte: C++ and empty)
Example of a 8 bytes structure:

```
struct {
int a; //= 4 bytes
float b; //= 4 bytes
};
struct test; // 8 bytes
```

C Union (only one variable is used among many)
Size of an union = size of the biggest element of the union.
Example of a 8 bytes union:

```
union {
int a; //= 4 bytes
double b; //= 8 bytes
};
union test; // 8 bytes
```


C++ Class (aggregation of variables + methods)
Size of an instanced class = size of its variables
(+ N * (size of its variables) if N inheritances)
Example of a 16 bytes object:

```
class A{
int a; //= 4 bytes
float b; //= 4 bytes
A() {}
};
class B : A{
int ap; //= 4 bytes
float bp; // = 4 bytes
B(): A() {}
};
B btest; // 16 bytes object
```


Other language may have different data type sizes ...
Data types in Java:

Type	Typical Bit Width	Typical Range
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Sufficient for storing 15 decimal digits
boolean	1 bit	True or false values
char	2 bytes	Stores a single character/letter or ASCII values

```
public class example {
public static void main(String [] args) {
System.out.println(ObjectSizeCalculator.getObjectSize(3));//16
}}
```


Data types in Python (3.9.2 on Linux x86-64)

Type	Typical Bit Width
int	28 bytes
float	24 bytes
empty str	49 bytes
empty list	56 bytes
empty tuple	40 bytes
empty dict	232 bytes
set	216 bytes
frozenset	216 bytes
None	16 bytes

```
import sys
a = sys.getsizeof(12)
print(a)
b=sys.getsizeof('geeks')
print(b)
```

Storing integer values is relatively straightforward.

Examples:

- You all know that $8 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$. Thus, 8 is equal to 1000 in binary code (we just need 4 bits to store 8). We can use a "char" type to store this value in C++.
- Another example: $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. The binary form of 11 is just: 1011. Again we just need 4 bits, we can use a "char" type to store this value in C++
- A last example: $888 = 1101111000$ in binary code. In this case we need to store 10 bits. $10 bits > 1 bytes$ and $10 bits < 2 bytes$: we can use a short type to store this value in C++.

Rule

For unsigned values: you need to count the numbers of bits required to store the integer, and you know which type you can use. For signed values: beware! You also need to have a bit for the sign of the integer!

Storing floating numbers is little bit more complicated.

Floating point numbers usually follow **IEEE-754 representations** (mostly the **IEEE-754 Single Precision** and the **IEEE-754 Double Precision** representations).

- IEEE-754 Single Precision:

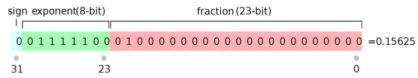


Figure: 8 bits for exponent part, 23 bits for fraction part and 1 bit for sign

- exponent part = real exponent + 127 (in binary)
  - 127 is a bias to handle negative exponents
- fraction part = fraction part (in binary)
- sign part = 0 (if positive) or 1 (if negative)
- $number\ (in\ binary) = (-1)^{sign} \times 1 \times fraction\ (in\ binary) \times 2^{real\ \ exponent}$

- Example of a real in IEEE-754 Single Precision binary representation:
  - $23.375 = (10111.011)_2 = (-1)^0 \times 1.0111011 \times 2^{-4}$
  - sign part $= 0_2$
  - fraction part $= (01110110000000000000000)_2$
  - exponential part $= (00000100)_2 + (10000000)_2 = (10000100)_2$
  - Final bit storage $= (01000010001110110000000000000000)_2$
- IEEE-754 Double Precision:

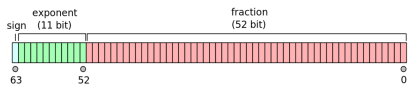


Figure: 11 bits for exponent part, 52 bits for fraction part and 1 bit for sign

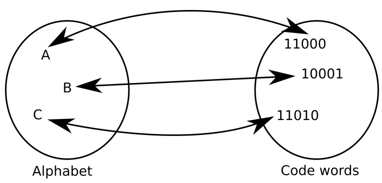
- exponent part = real exponent + 1023 (in binary)
  - 1023 is a bias to handle negative exponents
- fraction part = fraction part (in binary)
- sign part = 0 (if positive) or 1 (if negative)
- $number\ (in\ binary) = (-1)^{sign} \times 1 \times fraction\ (in\ binary) \times 2^{real\ \ exponent}$

Dealing with encoding characters:

- We know how integers and floating numbers are stored, but what about characters?
- Indeed, characters cannot be directly transformed into bits (such as integers or floating number).
- And all information need to be transformed into bits to be processed by a computer...
- We need an **encoding mechanism** to transform characters into bits.

What is encoding? Encoding the information consists in matching symbols to binary representations (code words) using an encoding map.

In the given example below the information is made of alphanumerical characters[2]:



[2] it is a bijection

How to get an encoding map of code words?

We can consider the character space as a system and each character as a state in this system. And we can use a fixed-state coding approach to get an encoding map:

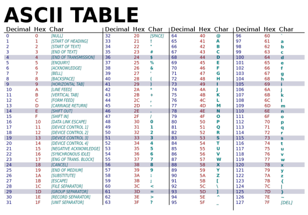
- Each state of the system is coded by a certain number of bits (called length of code) with:
  - 1 bit: we can code 2 states (0 or 1)
  - 2 bits: we can code 4 states (00, 01, 10, 11)
  - N bits: we can code 2^N states

The number of states required to code P states is n such as:
$2^{(n-1)} < P \leq 2^n$

Example: We have 26 letters in the alphabets, how many bits are necessary to code P values? We need 5 bits, because: $2^4 < 26 \leq 2^5$

Examples of character encoding systems:

- The **Baudot code** is encoded on 5 bits. With the Baudot code we can code up to 32 characters (and thus the 26 letters of the alphabet).
- The **ASCII table** (7 bits encoding). With the ASCII table uppercase letters, system and special characters can also be encoded (we can code 128 characters).



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

ASCII to unicodes:

With ASCII we can code only 128 characters...

With ISO-8859-X (coded on 8bits) other 128 characters can be used to code for other characters (because of the 8-th bit). Other characters specific to languages such as: latin, cyrillic, arabic, greek and hebrew can be encoded as well.

Example of ISO-8859-X unicode: ISO-8859-1.

With ISO-8859-X unicodes it's not possible to encode Chinese characters. The Guobiao character encoding permits that (for simplified characters: GB2312, GB13000, GB18030, GBK, GB_1988-80 and GB_198880, for traditional characters: BIG-5, BIG-FIVE, BIG5-HKSCS, BIG5, BIG5HKSCS, BIGFIVE, CN-BIG5, CN-GB and CN).

Examples:

- Let's imagine we have a book written in English with 150 pages and with an average of 1000 characters per page. What would be the approximate size of the specific content of this book? We can use ASCII encoding (because English)...
  - Approximate size = $150 \times 1000 \times 7$ bits = 131250B = 131.250KB
- The same book but written in French (same average number of characters and 1.2 times more pages. And we use ISO-8859-1 encoding.)
  - Approximate size = $180 \times 1000 \times 8$ bits = 180000B = 180KB
- The same book written in simplified Mandarin Chinese (this time with 80 pages and an average 2000 characters per page). We use GB 18030 encoding (4 bytes per character).
  - Approximate size = $80 \times 2000 \times 4$ bytes = 640000B = 640KB

How to deal with images, audio and videos?

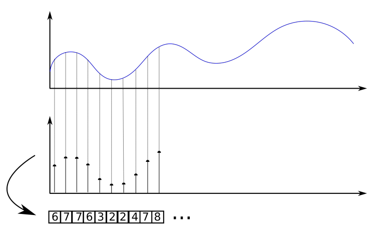
- Images, audio and videos are all continuous information.
- However, computers can only deal with discrete binary information.
- To store these information we have to **sample the continuous information into discrete values** and **transform these values into bits**.

How do we transform a continuous signal into bits?

The continuous signal has to be transformed into a string of bits to be stored in the memory by **sampling the signal**.

The sampling of a continuous signal can be made by extracting samples at regular times:

- The sampling interval is constant (= we have a fixed sampling frequency)
- The order of magnitude is kept
- Each sample is finally transformed into binary



Beware

Shannon Law: the **sampling frequency must be bigger or equal to twice the maximum frequency of the signal**. In order not to lose information.

Example 1:

The max frequency of the human voice is between 300Hz and 3400Hz.

Let's choose F_{max} =4000Hz

According to the Shannon law, $F_{sampling}$ must be $\geq 2 \times F_{max}$

So let's choose $F_{sampling}$ =8000Hz (8000 samples per seconds)

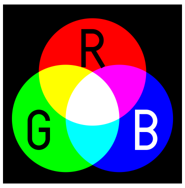
Besides, to get a correct audio reproduction each sample should be encoded on 12 bits.

So, in order to store in real time the human voice we should have a Bit Rate of $8000 \times 12 = 96000$ bits/second

Example 2:

As you know, the color can be decomposed into three different primary colors (red, green and blue).

The weighted sum of these three primary colors give the color (this is the additive synthesis).



Each image point in an image is represented by two values: the luminance (luminious intensity) and the chrominance (conveying color information).

These two values are bounded together by this relation:

$Y = 0.3 \times R + 0.59 \times V + 0.11 \times B$

Y: luminance R, G and B: chrominance components

For example, with the lowest quality of European Digital TV standard we have:

- 625 lines per image (576 lines are really used).
- 720 points per line
- 25 images/second

Thus we can store:

- 720 points per line for the luminance (encoded on 8bits)
- 360 points each for the blue and red chrominance (encoded on 8bits)

So we have 1440 points per line.

Number of bits to store for one image = $(1440 \times 8 \times 576) = 6635520$ bits

With a framerate of 25 images/second: The Bit Rate must be: $6635530 \times 25 = 166$Mbit/s

Actually this Bit Rate is difficult to realize, one solution is to compress the data to lower the required Bit Rate.

This is the goal of the "Motion Picture Expert Group" or MPEG: they define the compression algorithm for sound and video (see after in the course for more info about compression)...

On computers, data is mostly stored on files.

We have two categories of file formats: **binary** and **text** file formats:

- Binary file formats:
  - Can store multiple types of information in a single file (audio, text, video). Example: xlsx
  - Very application dependent (mostly, only Excel can read xlsx, you cannot open avi file with Excel, etc)
  - Here 1 byte = 1 custom data (huge byte sequence dependency)
- Text file formats:
  - Encoding dependent (ASCII, ISO-8859-1, GB18030, etc)
  - Can be opened on any system by any editor
  - Easy to read and understand
  - Less prone to get corrupted, more robust (1 byte error = may be just one error on an isolated and non important character)
  - 1 byte = 1 readable character

A non-exhaustive list of interesting file formats in Data Engineering:

- CSV, TSB: column and tab-based text file formats
- XLSX XLS: sheets+columns+media binary Excel file format
- JSON: key-value-based structured file format
- XML: tag-based structured file format
- AVRO: Hadoop binary dump file format
- PARQUET: column first binary file format
- JPG, PNG and so on: image binary file formats
- MPEG, OVG and so on: video binary file formats
- MP3 and so on: audio binary file formats
- Many other...

CSV (Comma Separated Value) - Text file

- A CSV is a file with the values comma-separated, this is used to store column and row data
- CSV files can be used with Microsoft Excel, Google Spreadsheets, etc
- It differs fron other spreadsheet format: you can only have one single sheet in a file. You cannot store formulas, images, etc.

CSV file exemple:

```
column_name_1, colum_name_2, column_name_3, ...
value1, value2, value3, ...
```

Code exemple (python/pandas):

```
import pandas as pd
data = pd.read_csv(sep=",", "example.csv")
print(data['column1'])
```

Code exemple (awk):

```
awk -F',' '{ print $1; }'
```


TSV (Tabular Separated Value) - Text file

- Values are tab-separated
- TSV is an alternative to the common comma-separated values (CSV) format
- CSV: difficulties if you use commas in values...

Exemple:

```
column_name_1 column_name_2 column_name_3 ...
value1 value2 value3 ...
```

Code (using Python/pandas):

```
import pandas as pd
data = pd.read_csv(sep="t", "example.csv")
print(data['column1'])
```

Code (using awk):

```
  awk -F'\t' '{ print $1; }'
```


XLSX, XLS (Excel file format) - Binary format

- Official spreadsheet format of Microsoft Excel
- Store column/raw data in sheets
- Can store more than one sheet, plus images, videos, etc.

Code (using pandas):

```
import pandas
data = pd.ExcelFile('example.xls')
df1 = pd.read\_excel(data, 'my sheet 1')
df2 = pd.read\_excel(data, 'my sheet 2')
```


JSON (JavaScript Object Notation) - Text format

- Data are stored and structured as attribute-value pairs
- Returned format of some APIs (Application Programming Interface)
- Open standard file format

Exemple:

```
{ "sport": {
            "ball": {
                "solo": ["tennis","pingpong"],
                "team": ["soccer", "rugby"]
              }
    }}
```

Code (using pandas):

```
import json
data = json.load(open("example.json")
print(data['sport']['ball'])
print(data['sport']['ball']['solo'])
```


XML (eXtensive Markup Language) - Text format

- Format used to structure data for storage and transport
- The structure is made of tags, options and clear text
- This is the returned type of some APIs (ex: WeChat API)

Exemple:

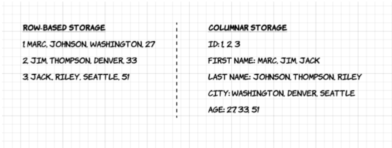
```
<?xml version="1.0" encoding="UTF-8"?>
<email>
  <from>Jani</from>
  <body>Message 1</body>
</email>
```

Code example (Python):

```
import xml.etree.ElementTree as ET
tree = ET.parse("example.xml")
root = tree.getroot()
item = root.find("email")
```

Parquet - Binary format

- Designed for large-scale analytical querying
- A flat columnar storage format (different than CSV/TSB that are row-based)
- Designed to bring efficiency
  - Row-based easier to conceptualize, but slower to use in some case
  - Column-based permits to skip non-relevant data quickly



Code example (Python):

```
import pyarrow.parquet as pq
pq.write_table(table, 'example.parquet')
```


Pickle (Serialized format) - Binary format

- Serialization refers to the process of converting object from the main memory to the disk
- Pickle is a well-known Python module permitting that
- Very useful to store objects, structures or Machine Learning models

Code exemple:

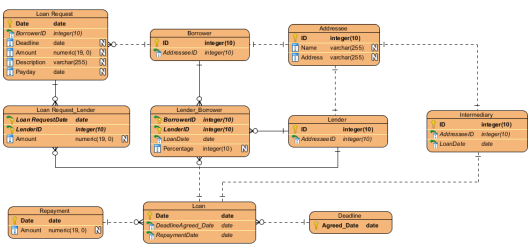
```
import pickle
class exampleClass:
     a = 35
     b = [1, 2, 3]
object = exampleClass()
outfile = open("example.bin",'wb')
pickled_object = pickle.dump(object, outfile)
infile = open("example.bin", 'rb')
loaded_pickled_object = pickle.load(infile)
```


Another popular way to save data is to store it in a **database**.

What is a database?

- Alternative at storing data in the file system
- An elegant way to **centralize all the data in one place**
- Force users and developers to access and store the data in **a unique and well defined way**
- Permit to **structure the data and manage it in a powerful way**
- **Different levels of permission** can be set
  - Administrator can read, write and delete everything
  - Users can only access their databases and in a specific way
- Databases usually **offer a set of very powerful functionality to extract and write data**
  - Accessible through a language (SQL), an API and/or the command line
  - Can be directly "plugged" to a software

Example of a relationship diagram of a relational database:



Actually, there are two main types of databases:

1. Relational Databases (using the SQL language)
   - MySQL: Relatively easy to use, suitable for most applications
   - PostgreSQL: Complex to use, but more powerful than MySQL
   - SqLite: Easy to use
   - Etc.
2. Non-relational Databases called NoSQL
   - MongoDB
   - ElasticSearch
   - Etc.

Comparisons between Relational and Non-Relational databases:

Property	Relational Databases	Non-Relational Databases
Data model	Relational	Non-relational
Structure	Table-based, with columns and rows	Document based, key-value pairs, graph, or wide-column
Schema	A predefined and strict schema in which every record (row)	A dynamic schema, records don't need to be of same nature
Query language	Structured Query Language (SQL)	Varies from database to database
Scalability	Vertical	Horizontal
Ability to add new properties	Need to alter the schema first (migration: risky!)	Possible without disturbing anything

Code example (Python and SQLite - a relational database):

```
import sqlite3
conn = sqlite3.connect('mydatabase.db')
cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
    print("NAME = {}".format(row[1]))
    print("SALARY = {}".format(row[2]))
  conn.close()
```

Code example (Python and MongoDB - a non-relational database):

```
from pymongo import MongoClient
client = MongoClient(host="localhost", port=27017)
db = client["mydb"]
collection = db.addresses
address = collection.find_one({"NAME": "Jon"})
collection.insert_one({"NAME": "Paul", "PHONE": "12334"}) #JSON format
client.close()
```


Data Compression, what is it and why we need it in Data Engineering?

- Data Compression: process of encoding information using fewer bits than the original representation
- We need it to:
  - Save transmission capacity
  - Save transmission time
  - Reduce storage occupancy
  - Reduce computation
- So it's all about saving resources and money...

Basically two main types of compression:

- Lossy compression
- Lossless compression

Examples of popular algorithms used for compressing data:

Lossless algorithms:

- Run-length
- Huffman (used with: zip, gzip, png, etc.)
- LZW (Lempel-Ziv-Welch) (used with: gif, tiff)
- LZMA (Lempel-Ziv-Markov chain algorithm) (used with 7zip)
- Etc.

Lossy algorithms:

- JPEG (images)
- MPEG (videos)
- MP3 (audio)

What is a lossless compression?

- It means there is **no loss of information when the data is compressed**
- **Original data can be exactly recovered from the compressed data** (exact size before and after)
- Usually used with **applications that cannot tolerate a loss of information**, for example:
  - word documents
  - computer programs
  - company's customer dataset
  - Etc.
- Principle:
  - **Redundant data is removed during compression**
  - **Redundant data is added back during decompression...**

What is a lossy compression?

- Lossy compression results in **lost data and quality from the original representation**
- However, because it removes data from original representation it **usually mean a greater compression**
- Remove just what have to be removed:
  - JPEG may reduce image's size by more than 80% without noticeable effect
  - MP3 may reduce down to one tenth the size of the original audio file
- Trade-off: **a greater compression will take up less space, but may alter the quality**

In Data Engineering we are mostly interested in used lossless compression. Let's compare[3] these lossless compression file formats then:

- **gzip**: uses Lempel-Ziv coding (LZ77) (tar czf pack.tar.gz rep/
- **bzip2**: uses the Burrows-Wheeler block sorting text compression algorithm and Huffman coding (tar cjf pack.tar.bz2 rep/)
- **zip**: ZIP MSDOS (zip -r pack.zip rep/)
- **rar**: proprietary archive file format (rar a pack.rar rep/)
- **lha**: based on Lempel-Ziv-Storer-Szymanski-Algorithm (LZSS) and Huffman coding (lha a pack.lha rep/)
- **lzma**: Lempel-Ziv-Markov chain algorithm (tar –lzma -cf pack.tar.lzma rep/)
- **lzop**: Similar to gzip but favors speed over compression ratio (tar –lzop -cf pack.tar.lzop rep/)

[3] Original study: https://binfalse.de/2011/04/04/comparison-of-compression/

Comparing the compression of binaries (total: 176.753.125 Bytes):

Method	Compressed Size	% of original	Time in s
gzip	161.999.804	91.65	10.18
bzip2	161.634.685	91.45	71.76
zip	179.273.428	101.43	13.51
rar	175.085.411	99.06	156.46
lha	180.357.628	102.04	35.82
lzma	157.031.052	**88.84**	129.22
lzop	165.533.609	93.65	**4.16**

Comparing plain text compression (total: 40.040.854 Bytes):

Method	Compressed Size	% of original	Time in s
gzip	11.363.931	28.38	1.88
bzip2	9.615.929	24.02	13.63
zip	12.986.153	32.43	1.6
rar	11.942.201	29.83	8.68
lha	13.067.746	32.64	8.86
lzma	8.562.968	**21.39**	30.21
lzop	15.384.624	38.42	**0.38**

Comparing the compression of a mix of data (plain+media+binary+pictures+random, total: 355.971.825 Bytes):

Method	Compressed Size	% of original	Time in s
gzip	294.793.255	82.81	20.43
bzip2	290.093.007	81.49	141.89
zip	313.670.439	88.12	23.78
rar	305.083.648	85.70	246.63
lha	315.669.631	88.68	64.81
lzma	283.475.568	**79.63**	258.05
lzop	307.644.076	86.42	**7.89**

For the next courses:

- Enough of the theory for now!
- Bring your laptop computer!
- You will need Linux on virtual machine: https://itsfoss.com/install-linux-in-virtualbox/
- Or installed on a USB stick: https://itsfoss.com/create-live-usb-of-ubuntu-in-windows/