Big Data e Big Problem

Normalmente si può trovare come definizione di big data la seguente:

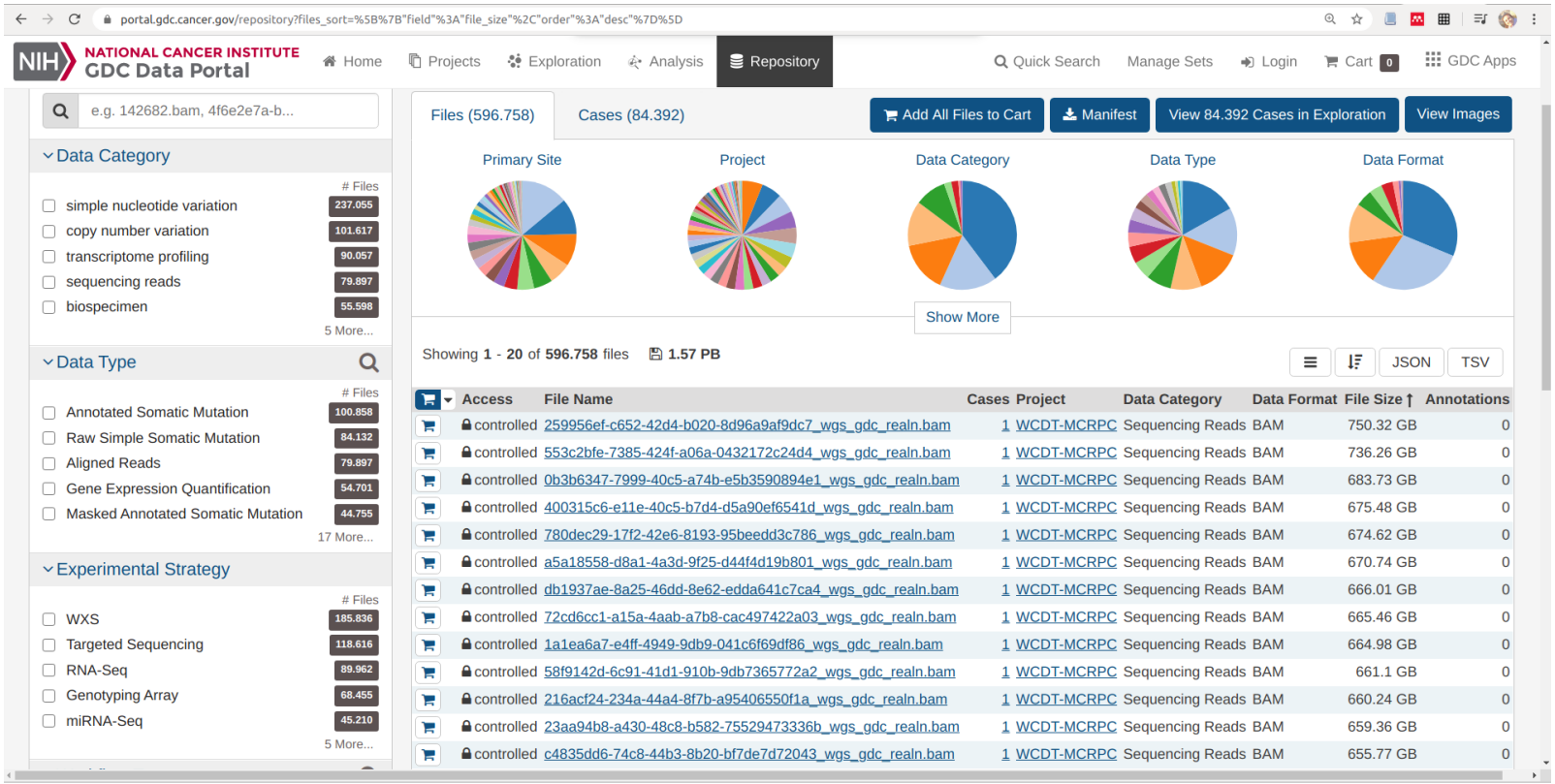
```
I dati sono troppo grandi per entrare in memoria/disco rigido
```

Esiste però un altro problema Big:

```
Anche su dati piccoli, il mio modello potrebbe richiedere più
risorse di quelle a mia disposizione
```

The Cancer Genome Atlas

https://www.cancer.gov/about-nci/organization/ccg/research/structural-genomics/tcga (https://www.cancer.gov/about-nci/organization/ccg/research/structural-genomics/tcga)

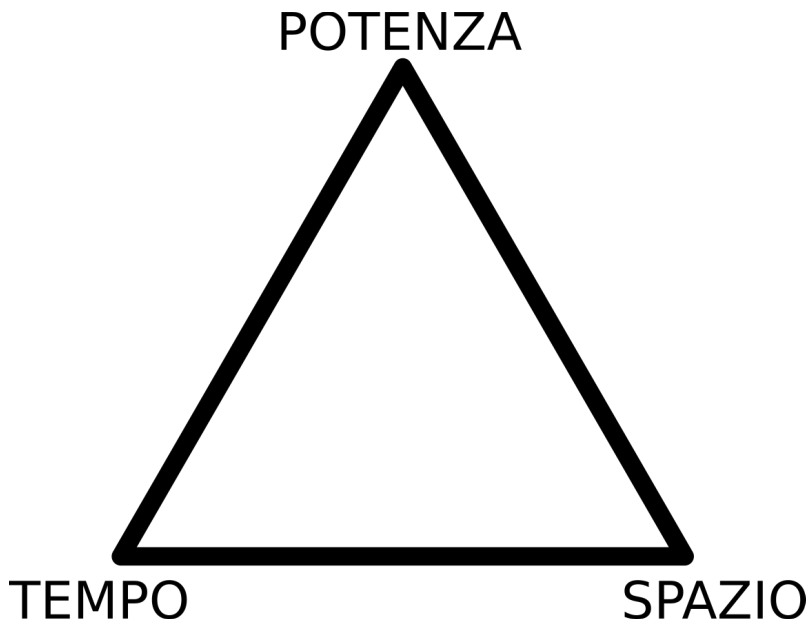


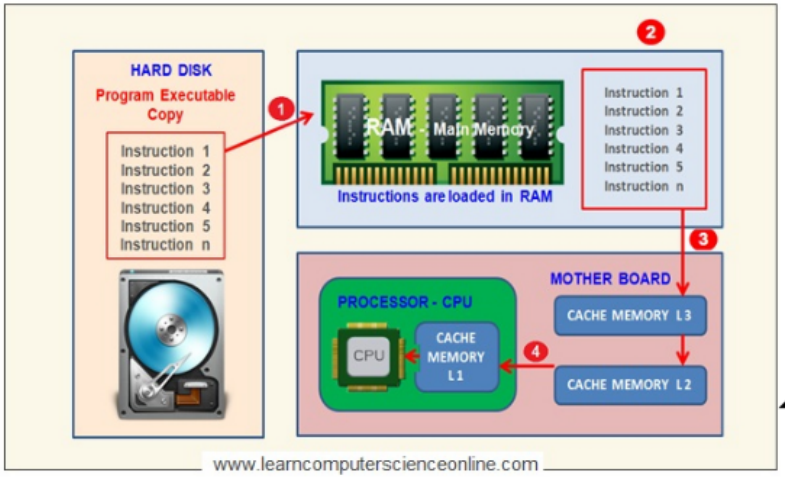
Nei big data dobbiamo fare delle scelte

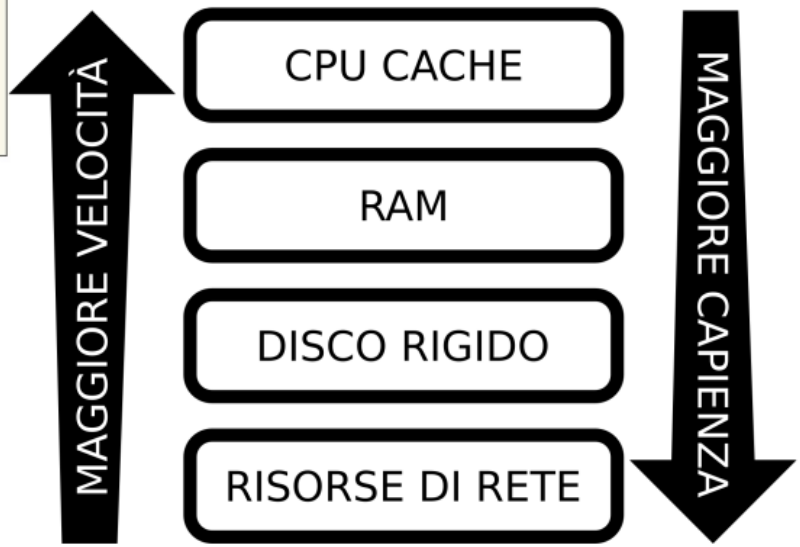
Semplificando, ci sono 3 variabili nel nostro sistema:

- il **tempo** che siamo disponibili ad aspettare
- lo **spazio** su RAM che siamo disponibili ad allocare
- l'ammontare di informazioni che vogliamo ottenere (statistical **power**)

Tanto più ottimizziamo uno di questi parametri, tanto più ci rimettiamo negli altri due.

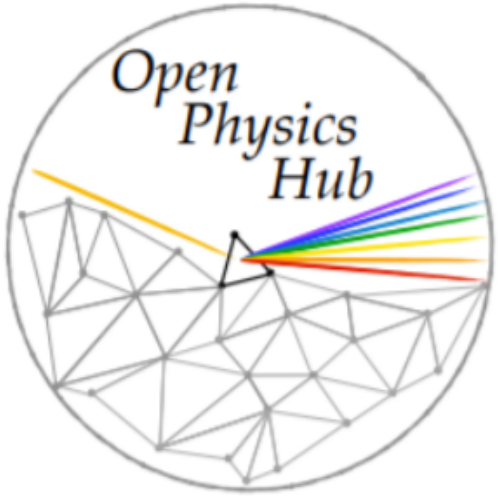

POTENZA
TEMPO
SPAZIO


HARD DISK
Program Executable Copy
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction n
1
2
RAM - Main Memory
Instructions are loaded in RAM
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction n
3
MOTHER BOARD
PROCESSOR - CPU
CPU
CACHE MEMORY L1
4
CACHE MEMORY L3
CACHE MEMORY L2
www.learncomputerscienceonline.com


MAGGIORE VELOCITÀ
CPU CACHE
RAM
DISCO RIGIDO
RISORSE DI RETE
MAGGIORE CAPIENZA

Questo schema non tiene conto anche di altri fattori:

- la leggibilità e trasferibilità del codice (potremmo scrivere in codice binario)
- il tempo che serve all'analista per produrre il codice



HIGH PERFORMANCE COMPUTING CLUSTER

"MATRIX" features

- 32 multi-core Intel XeonGold 5120 processors with dual thread
- 8 GB of memory per physical core (4 GB per virtual dual threaded core)
- 100 Gb/s InfiniBand connection
- 240 TB of disk space on SAS disks

"BLADE RUNNER" specifications:

Infiniband connected nodes:

- 2 nodes with 24 virtual cores and 64 GB RAM
- 2 nodes with 32 virtual cores and 64 GB RAM
- 1 node with 32 virtual cores and 128 GB RAM
- 2 nodes with 16 virtual cores and 24 GB RAM

Ethernet connected nodes:

- 2 nodes with 12 virtual cores and 16 GB RAM
- 2 nodes with 32 virtual cores and 64 GB RAM

https://site.unibo.it/openphysicshub/en

Leggere files (grandi)

Quando abbiamo a che fare con files molto grandi, per leggerli è utile usare la funzione `open` .

Questa funzione può essere usata per delimitare un blocco di codice all'interno del quale il file è aperto.

Al termine delle operazioni, questo verrà chiuso in modo automatico e sicuro, evitando che rimanga aperto e venga corrotto da altri processi.

In [1]:
```
%%file prova.txt
tonno
mandibola
rum
pinocchio
sigmoide
```

```
Overwriting prova.txt
```

In [2]:
```
with open('prova.txt') as file:

    for line in file:

        print(repr(line))

        print(len(line))
```

```
'tonno\n'
6
'mandibola\n'
10
'rum\n'
4
'pinocchio\n'
10
'sigmoide\n'
9
```

Sulla RAM non viene mai caricato tutto il file ma solo una riga alla volta.

Ci sono tre operazioni fondalmentali per l'analisi dati:

- iterazione lazy
- map (ripeti un'operazione su tutti gli elementi)
- filter (seleziona solo una parte degli elementi)

Altri tipi di operazioni che discuteremo saranno:

- reduce (comporre insieme gli elementi)
- functional programming

Iterazione Lazy

Che cosa intendiamo con iterazione **lazy**?

Le operazioni non vengono compiute finchè il risultato non è richiesto!

In Python questa cosa è gestita da degli oggetti chiamati **iteratori**.

Sono gli oggetti su cui faccio i cicli **for**.

$\rightarrow$ Un iteratore può essere percorso una volta sola!

Questo è controintuitivo: se provo a fare un ciclo for su di una lista, lo posso fare quante volte voglio

In [12]:

```
lista = [1, 2, 3]

print("--- prima iterazione ---")
for elemento in lista:
    print(elemento)

print("--- seconda iterazione ---")
for elemento in lista:
    print(elemento)
```

```
--- prima iterazione ---
1
2
3
--- seconda iterazione ---
1
2
3
```

Ma se provo a farlo su di un file, lo posso leggere una volta sola!

Se volessi rileggerlo, dovrei aprirlo di nuovo!

In [13]:
```
# creiamo il file NONTOCCARE
! echo "un testo inutile" > ./NONTOCCARE.TXT
```

In [14]:
```
import os # pip install os
```

In [15]:
```
os.getcwd() # in quale cartella mi trovo?
```

Out[15]:
```
'/home/claudia/Documenti/Didattica/AA2020_2021/2021_PLS/PLSBigDataNetworks'
```

In [16]:
```
os.listdir() # quali files e sotto-cartelle ci sono?
```

Out[16]:

```
['Lezione 1a - Introduzione a Python.ipynb',
 'Lezione 1b - Iteratori e Big Data.ipynb',
 'README.md']
```

In [17]:

```
directory = "./"
filename = "NONTOCCARE.TXT"

position = directory + filename

with open(position) as file:

    print("--- prima iterazione ---")
    for line in file:
        print(repr(line))

    print("--- seconda iterazione ---")
    for line in file:
        print(repr(line))
```

```
--- prima iterazione ---
'un testo inutile\n'
--- seconda iterazione ---
```

Python ce lo nasconde, ma in realtà ogni volta che iteriamo sulla lista lui crea un nuovo iteratore che scorre la lista e poi scompare.

Possiamo farlo esplicitamente con il comando **iter**

In [18]:
```
lista = [1, 2, 3]

iteratore_lista = iter(lista)

print("prima iterazione")
for elemento in iteratore_lista:
    print(elemento)

print("seconda iterazione")
for elemento in iteratore_lista:
    print(elemento)
```

```
prima iterazione
1
2
3
seconda iterazione
```

Non sempre è possibile evitare di caricare l'intero data set.

Supponiamo di voler calcolare tutte le combinazioni di elementi di una sequenza: non possiamo risolvere questo problema senza tenere in memoria l'intera sequenza!

Map

Un tipo di operazione molto frequente sulle sequenze è il cosiddetto **mapping**, ovvero applicare una funzione a tutti gli elementi di una lista, uno alla volta ed indipendentemente dagli altri.

Ad esempio, avendo una serie di numeri, potrei voler prendere il quadrato di ciascuno.

In [19]:
```
numeri = [0, 1, 2, 3, 4, 5, 6]

quadrati = []
for numero in numeri:
    quadrato = numero **2
    quadrati.append(quadrato)

print(quadrati)
```

```
[0, 1, 4, 9, 16, 25, 36]
```

Questo può essere espresso in modo più conciso con una *comprehension*, che è funzionalmente identica al ciclo visto prima, ma più sintetica

In [20]:
```
numeri = [0, 1, 2, 3, 4, 5, 6]

quadrati = [x**2 for x in numeri]

print(quadrati)
```

```
[0, 1, 4, 9, 16, 25, 36]
```

Il concetto di **map** è un'astrazione di questo procedimento.

Python fornisce una funzione, chiamata appunto **map**, che prende in input una funzione ed un oggetto iterabile e ritorna un iteratore i cui elementi sono il risultato dell'operazione.

In [21]:
```
def quadrato(n):
    return n**2

numeri = [0, 1, 2, 3, 4, 5, 6]
quadrati = map(quadrato, numeri)
print(quadrati)
```

```
<map object at 0x7fa9266c3780>
```

Ricordiamoci che il risultato delle operazioni sugli iteratori, quando possibile, è a sua volta un iteratore!

Siamo noi che dobbiamo esplicitamente **concretizzare** l'iterazione

In [22]:
```
list(quadrati)
```

Out[22]:
```
[0, 1, 4, 9, 16, 25, 36]
```

ricordiamoci che l'iterazione è compiuta una volta sola, quindi se vogliamo il risultato dobbiamo salvarcelo alla prima concretizzazione!

In [23]:
```
list(quadrati)
```

Out[23]:
```
[]
```

Filter

Le operazioni **filter** selezionano un sottoinsieme dei dati e risultano nella generazione di un secondo iteratore.

In [24]:
```
numeri = [-2, -1, 0, 1, 2]
positivi = []
for numero in numeri:
    if numero > 0:
        positivi.append(numero)

print(positivi)
```

```
[1, 2]
```

in modo simile all'operazione di **map**, anche l'operazione di **filter** ha un costrutto nel linguaggio tramite le *comprehension*

In [25]:
```
numeri = [-2, -1, 0, 1, 2]
positivi = [x for x in numeri if x > 0]
print(positivi)
```

```
[1, 2]
```

ed esattamente come prima, abbiamo una funzione che prende una funzione di filtro (che ci dice se l'elemento è accettabile o no) e la applica ad un operatore

In [26]:
```
def is_positive(n):

    return n > 0

positivi = filter(is_positive, numeri)

print(list(positivi))
```

```
[1, 2]
```

Reduce

questa operazione combina gli elementi di un iteratore in un elemento unico

In [27]:
```
numeri = [1, 2, 3, 4]

totale = 0

for numero in numeri:

    totale += numero

print(totale)
```

```
10
```

Come per casi precedenti, esiste una funzione preesistente per effettuare le operazioni di riduzione: la funzione **reduce**

In [28]:
```
from functools import reduce

def somma(a, b):

    return a + b

numeri = [1, 2, 3, 4]

totale = reduce(somma, numeri)

print(totale)
```

```
10
```

Questo tipo di operazioni è così comune che ci sono una serie di operazioni predefinite:

- **sum** per la somma
- **min** e **max** per il minimo e massimo

e così via

Una tipica riduzione, che useremo molto, è la stima delle frequenze.

In [29]:
```
from collections import Counter

numeri = [ 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 4 ]

Counter( numeri )
```

Out[29]:
```
Counter({1: 3, 2: 2, 3: 2, 4: 5})
```

Una proprietà importante delle riduzioni è che i risultati si possono combinare: dati i conteggi su due serie, posso sommare insieme i due conteggi ed ottenere i conteggi totali fra le due serie

Map - Reduce

il famoso metodo MAP-REDUCE è una combinazione di queste idee:

- prendo una sequenza, la divido in sottosequenze
- invio le sequenze a diversi computer
- compio una riduzione su ciascuna sottosequenza
- raccolgo le sottosequenze e le combino insieme
- tutto questo fatto in modo ricorsivo

In [ ]: