Compter efficacement

Introduction

Problème

Pour un ensemble de données potentiellement très grand, on veut compter le nombre d'éléments distincts dans cet ensemble

Questions

Quelles stratégies ?

- déterministe pour un comptage exact
- probabiliste pour un comptage approché

Qualité des solutions ?

- Quel temps faut-il ?
- Quel espace mémoire faut-il ?

Comment compter de manière efficace ?

Un facteur critique : l'espace mémoire

Des données de plus en plus grandes :

- http://www.internetlivestats.com/
- les séquences génomiques
  - Le génome humain : 3400 Mega paires de nucléotides
  - La fleur *Paris japonica* : 150 Giga paires de nucléotides

Taille de la mémoire RAM d'un ordinateur standard :

```
$ free -h
              total        utilisé      libre      partagé      cache   disponible
Mem:          7,4G         4,6G         464M         424M      2,3G      2,0G
Partition d'échange:          0B           0B           0B
```

La quantité de données augmente plus vite que les capacités mémoire des machines

Comment compter de manière efficace ?

Termes

Flux : une liste d'éléments

Cardinalité d'un flux : le nombre d'éléments distincts du flux

Exemple

le flux :

une liste de 11 entiers $(6, 7, 3, 7, 1, 3, 3, 7, 9, 2, 3)$

sa cardinalité :

$|\{6, 7, 3, 1, 9, 2\}| = 6$

Enjeu

Calculer la cardinalité d'un flux en économisant au mieux la ressource temps et, ici la plus critique, la ressource espace

Comment compter de manière efficace ?

Applications variées

- Combien de visiteurs uniques sur un site web
- Détection d'attaque sur les réseaux (nombre de connections distinctes anormalement élevé)
- Optimisation des requêtes dans les BD
- Comparaison de documents

Deux approches

1. Comptage exact
2. Comptage probabiliste. Sacrifier l'exactitude au profit d'un gain d'espace

Comptage exact

Première idée

1. on stocke en mémoire tous les éléments `el1, el2, ···` du flux
2. on les trie
3. on supprime les doublons

```
def compteNaif(flux):
    ens = [el for el in flux]
    ens.sort()
    res = list()
    pred = None
    for el in ens:
        if el != pred:
            res.append(el)
            pred = el
    return len(res)
```

Si `N` est le nombre d'éléments du flux, nécessite

- `N log(N)` comparaisons
- un espace linéaire en `N`

Comptage exact

Deuxième idée

Ne mémoriser que les éléments distincts

```
def compteNaifBis(flux):
    res = list()
    for el in flux:
        if el not in res:
            res.append(el)
    return len(res)
```

Si N est le nombre d'éléments du flux et n sa cardinalité, nécessite

- de l'ordre de $N \times n$ comparaisons
- un espace linéaire en n

Comptage exact

Troisième idée

Ne mémoriser que les éléments distincts en utilisant le hachage

Rappel sur les fonctions de hachage

Doc de la fonction `hash` de Python :

```
help(hash)

Help on built-in function hash in module builtins:

hash(obj, /)
    Return the hash value for the given object.

    Two objects that compare equal must also have the same hash value,
    but the reverse is not necessarily true.
```

Une fonction de hachage `h` prend en entrée un objet `el` et retourne un entier `h(el)`, appelé *empreinte* ou *haché* de l'objet.

`h` est déterministe : pour le même argument elle renvoie le même résultat (pour une exécution donnée)

`h` n'est a priori pas injective : deux objets distincts `x` et `y` peuvent avoir même empreinte (`h(x) = h(y)`) et entrer en *collision*

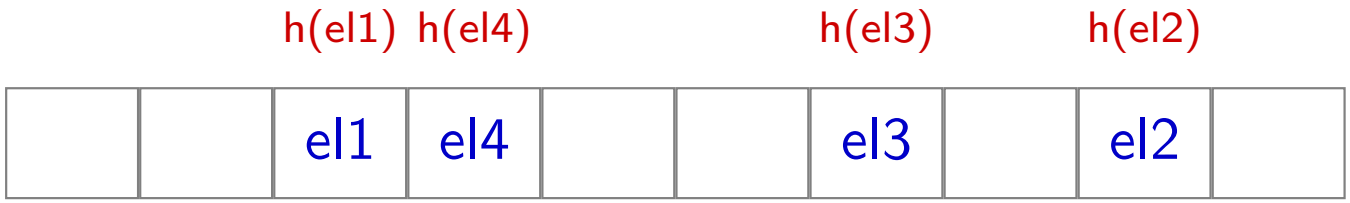
Comptage exact

Troisième idée : utiliser le hachage

La fonction de hachage `h` associe à chaque élément `el` une empreinte `h(el)`

Idée Stocker l'élément `el` dans une table à l'adresse `h(el)`

Flux : el1 , el2 , el3 , el2 , el4 , el1 , ...



Avantage, les différentes répétitions d'un élément se retrouvent à la même adresse.

La fonction n'étant pas forcément injective, il faut gérer les collisions. Stratégie simple, le *sondage linéaire* consiste à tester les places suivantes dans la table jusqu'à en trouver une libre.

Comptage exact

Troisième idée : utiliser le hachage

```
def compteAvecHachage(flux, m=40000):
    nb = 0
    table = [None]*m
    for el in flux:
        empreinte = hash(el) %m
        while table[empreinte] != None and
                                table[empreinte] != el:
            empreinte = (empreinte + 1) %m
        if table[empreinte] == None:
            table[empreinte] = el
            nb += 1
    return nb
```

Amélioration. Pour garantir que tous les éléments aient une place, au lieu d'initialiser une table de très grande taille, on part d'une table de taille modeste et on double sa taille quand elle devient à moitié pleine.

L'espace consommé est alors linéaire en la cardinalité du flux et le temps moyen linéaire en le nombre d'éléments du flux.

Comptage exact

Troisième idée : utiliser le hachage

Nota Bene

Dans les langages de haut niveau, on dispose de types prédéfinis – `set` en Python, `HashSet` en Java – qui ont comme structure de données sous-jacente une table de hachage.

```
def compteExact(flux):
    return len(set(flux))
```

Comptage exact

Ses limites

Que faire si le flux est trop grand et que ses valeurs distinctes dépassent les capacités mémoire de la machine ?

Dans ce cas, on ne peut pas faire de comptage exact. Seule une estimation est possible.

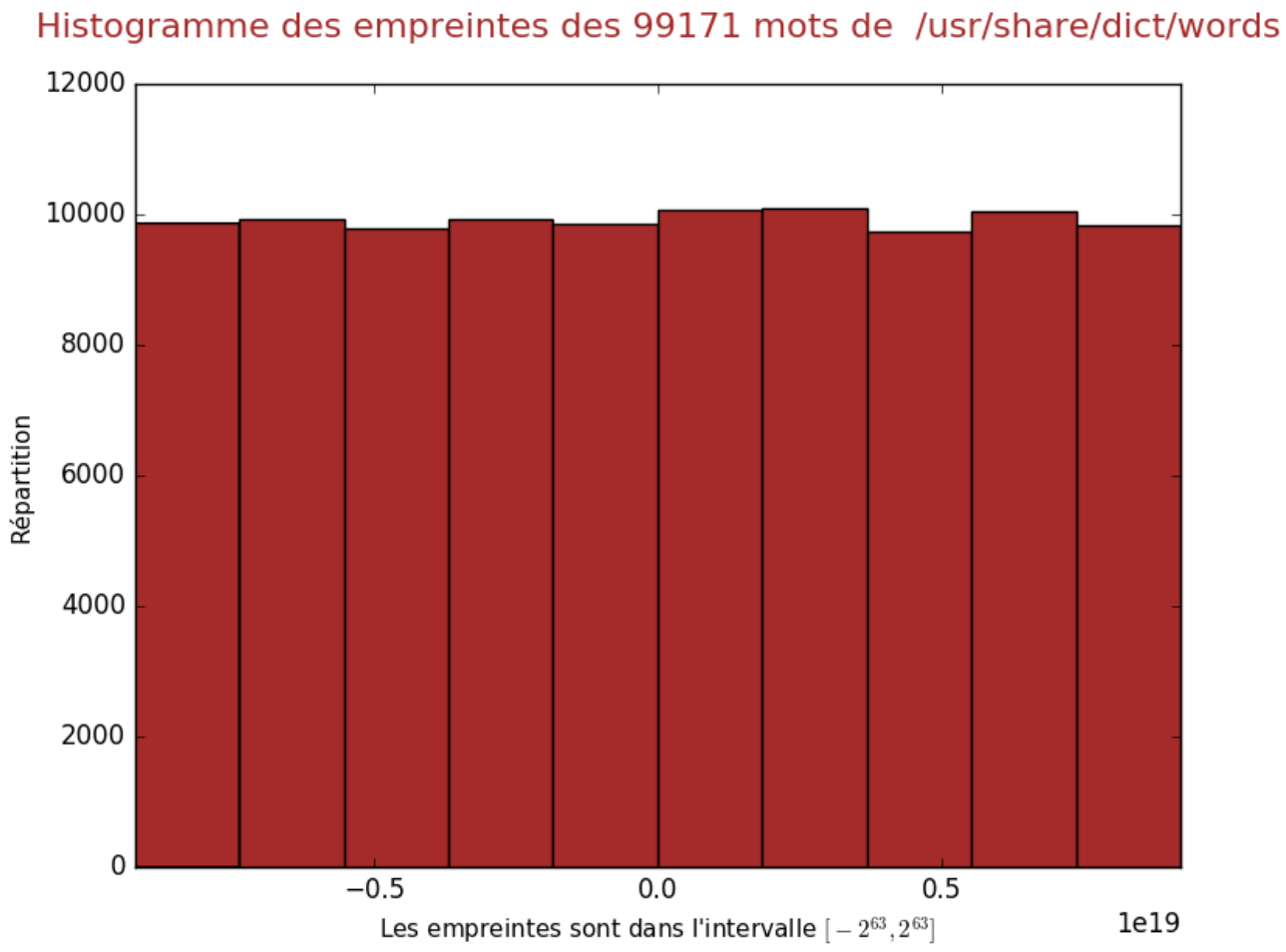
Problème - Comptage avec peu de mémoire

Calculer une estimation aussi précise que possible

- en effectuant qu'une seule passe sur les données
- en utilisant le moins de mémoire possible

Comptage avec peu de mémoire

Une propriété du hachage



Le hachage produit des empreintes uniformément distribuées

Comptage avec peu de mémoire

Idée

Exploiter le fait que les empreintes `h(el)` soient uniformément distribuées.

On considère l'écriture en binaire de ces empreintes et on observe les motifs (en particulier, les séquences de 0 consécutifs) qui y apparaissent. Ensuite, en se basant sur les propriétés de fréquence d'apparition de ces motifs, on estime la cardinalité du flux.

Comptage avec peu de mémoire

Du fait que les empreintes sont uniformément distribuées, on observe que les derniers bits des empreintes sont tels que :

- $\approx$ 50% terminent par 1
- $\approx$ 25% terminent par 10
- $\approx$ 12.5% terminent par 100
- $\approx$ 6.25% terminent par 1000

Si dans le flux, on observe des éléments dont les empreintes se terminent par 1, 10, 100, 1000 mais pas 10000 alors il y a de forte chance que l'ensemble est $\approx$ 16 éléments.

Avec 16 mots distincts dans le flux, environ 8 ont des empreintes qui terminent par 1, 4 ont des empreintes qui terminent par 10, 2 ont des empreintes qui terminent par 100, 1 a une empreinte qui termine par 1000 et 1 a une empreinte qui termine par 0000. Cette dernière peut terminer par 10000 ou 100000 ou 1000000 ou $\cdots$

Comptage avec peu de mémoire

flux	empreintes	empreintes écrites en binaire	position du premier 1 à droite
el1	h(el1)	1001010	1
el2	h(el2)	1101101	0
el3	h(el3)	1000000	6
el2	h(el2)	1101101	0
el4	h(el4)	0101100	2
⋮	⋮	⋮	≤ 4
eli	h(eli)	1010000	4
⋮	⋮	⋮	≤ 4
elj	h(elj)	1001000	3
⋮	⋮	⋮	≤ 4

Le premier trou dans les positions est 5

$\Rightarrow$ le flux a grosso modo $2^5 = 32$ éléments distincts.

Une analyse fine des fréquences d'apparition des positions du premier 1 définit un facteur de correction de 0.77351

$\Rightarrow$ la cardinalité du flux est estimée à $32/0.77351 \approx 41$

Comptage avec peu de mémoire

Algorithme élémentaire de comptage probabiliste

`Entrée` un flux

`Sortie` estimation de la cardinalité du flux

`hash` fonction de hachage qui, à un élément, associe un entier dans une fourchette de longueur 2^ℓ

ρ fonction qui a un entier `x` représenté en binaire compte le nombre de ses zéros consécutifs à droite

ϕ ≈ 0.77351 la constante magique

```
Initialiser un tableau BITMAP de ℓ zéros
Pour chaque élément el du flux faire
    index = ρ(hash(el))
    BITMAP[index] = 1
Calculer mini le plus petit i tel que BITMAP[i] = 0
Retourner 2^mini/ϕ
```

Espace utilisé : de l'ordre de ℓ bits

Comptage avec peu de mémoire

Algorithme élémentaire de comptage probabiliste

L'estimation est inévitablement grossière : l'écart entre deux estimations est grand

```
cteMagique = .77351
print([round(2**i/cteMagique) for i in range(4,20)])
[21, 41, 83, 165, 331, 662, 1324, 2648, 5295, 10591,
    21181, 42363,84725, 169451, 338902, 677804]
```

Par exemple, si le flux a 1000 éléments distincts, l'estimation aura une précision relative d'au mieux $\frac{1324-1000}{1000} = .324$
Souvent c'est bien pire

Comment obtenir plus de précision ?
Effectuer plusieurs estimations et faire la moyenne ne résout rien pour certaines cardinalités.

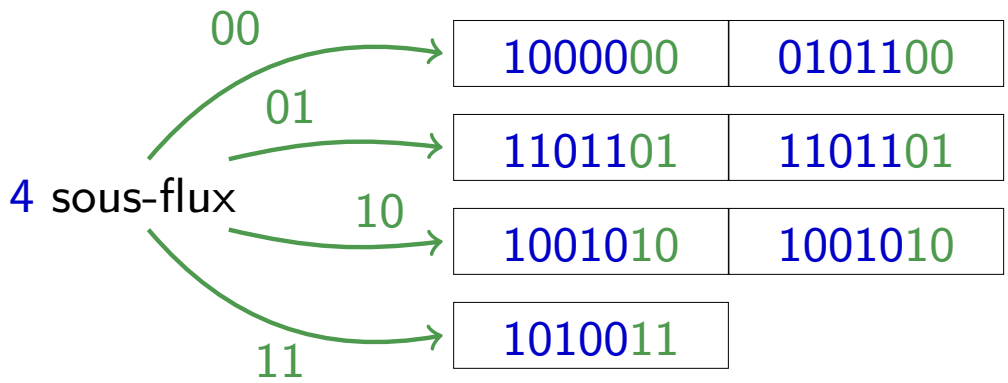
Comptage avec peu de mémoire

Comment obtenir plus de précision ?

Idée : Partager le flux initial en m flux indépendants.

On utilise les derniers $\log_2 m$ bits de chaque empreinte pour la rediriger vers l'un des m sous-flux.

Le flux : 1001010 , 1101101 , 1000000 , 1101101 , 0101100 , 1010011 , 1001010



Prendre comme m une puissance de 2

Comptage avec peu de mémoire

Comment obtenir plus de précision ?

Ensuite

- On applique l'algorithme élémentaire de comptage sur chacun des `m` sous-flux (en oubliant les derniers $\log_2 m$ bits utilisés pour le partage), on produit ainsi `m` tableaux `BITMAP`
- On calcule la signature de chaque tableau `BITMAP`, i.e., le plus petit index `i` tel que `BITMAP[i] = 0`
- On fait la moyenne `moy` des `m` signatures
- On retourne $2^{\texttt{moy}}/0.77351$

Comptage avec peu de mémoire

Algorithme de Flajolet & Martin

Entrée un flux et $m = 2^k$ le nombre de sous-flux

Sortie estimation de la cardinalité du flux

`hash` fonction de hachage a valeur dans une fourchette de longueur 2^ℓ

ρ compte le nombre de 0 consécutifs à droite d'un entier écrit en binaire

$\phi \approx 0.77351$ la constante magique

```
Initialiser une liste BITMAPS de m tables de ℓ zéros
Pour chaque élément el du flux faire
    empreinte = hash(el)
    numFlux = empreinte%m
    index = ρ(empreinte//m)
    BITMAPS[numFlux][index] = 1
cumul = 0
Pour numFlux de 0 à m - 1 faire
    Calculer mini le + petit i tq BITMAPS[numFlux][i] = 0
    cumul = cumul + mini
moy = cumul/m
Retourner m ×2^moy/φ
```

Comptage avec peu de mémoire

Algorithme de Flajolet & Martin

Espace mémoire utilisé

essentiellement $\mathtt{m}$ tables de longeur ℓ

Avec une valeur ℓ de 64, on peut alors compter jusqu'à $2^{64} \approx 10^{19}$ éléments

$\rightsquigarrow 64 \times \mathtt{m}$

Temps de calcul

linéaire en le nombre d'éléments du flux

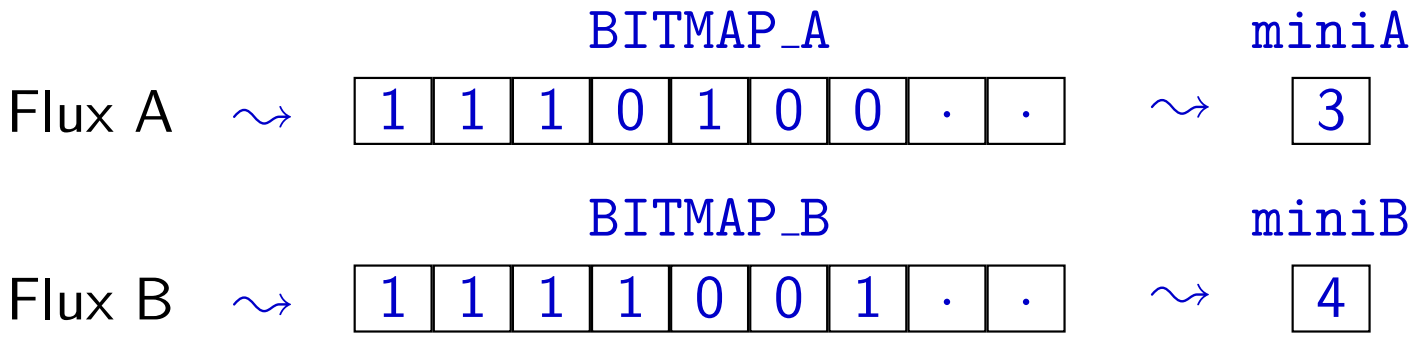
Précision relative de l'estimation

proche de $0.78/\sqrt{\mathtt{m}}$

Application à la comparaison de documents

Préliminaire – Estimation de la cardinalité de l'union de deux flux

On considère deux flux `A` et `B` et l'algo élémentaire de comptage sur chacun des deux

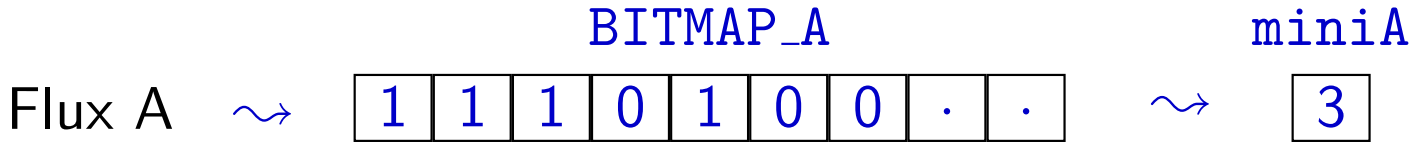


Comment obtenir à moindre coût une estimation de la cardinalité de $\mathtt{A} \cup \mathtt{B}$?

Application à la comparaison de documents

Préliminaire – Estimation de la cardinalité de l'union de deux flux

On considère deux flux `A` et `B` et l'algo élémentaire de comptage sur chacun des deux



Comment obtenir à moindre coût une estimation de la cardinalité de $A \cup B$?

`mini{A ∪ B}` est le plus petit `i` tel que

`BITMAP_A[i] = BITMAP_B[i] = 0`

Application à la comparaison de documents

Préliminaire – Estimation de la cardinalité de l'union de deux flux

Fait

Étant donnés deux flux, si l'on connaît leurs tableaux `BITMAP` calculés par l'algorithme élémentaire de comptage, ou mieux, leurs `m` tableaux `BITMAP` calculés par l'algorithme de Flajolet & Martin alors on peut de façon efficace et sans surcoût mémoire calculer également une estimation de la cardinalité de leur union.

Nota Bene

On peut appliquer la même stratégie pour calculer une estimation de l'union de `k` flux

Application à la comparaison de documents

Pour déterminer si deux documents sont proches, on peut évaluer le nombre de mots qu'ils ont en commun.

Définition (Indice de similarité)

L'*indice de similarité* de deux documents `A` et `B` est le rapport entre le nombre de mots distincts communs aux deux et le nombre de mots distincts de leur union :

$$\texttt{sim(A, B)} = \frac{|\texttt{A} \cap \texttt{B}|}{|\texttt{A} \cup \texttt{B}|}$$

Exemple

L'indice de similarité de `A` = "le soleil se lève côté Est" et `B` = "le soleil se couche côté Ouest" est `sim(A, B)` = `4/8` = `.5`

Application à la comparaison de documents

On souhaite calculer efficacement une estimation de cet indice.

Contrainte : ne réaliser qu'une seule passe sur chaque document.

Avec l'algorithme de Flajolet & Martin, on sait estimer les cardinalités de A et B, et comme on vient de le voir, il est alors simple d'estimer sa cardinalité sans surcoût mémoire.

Reste à noter qu'avec les estimations des cardinalités de A, B et $A \cup B$, on a tout ce qu'il faut pour calculer l'estimation de l'indice de similarité de A et B.
En effet, $|A \cap B| = |A| + |B| - |A \cup B|$, autrement dit :

$$\mathtt{sim}(A, B) = \frac{|A| + |B|}{|A \cup B|} - 1$$

Références

Philippe Flajolet.
Algorithmes probabilistes sur de grandes masses de données.
http://www.college-de-france.fr/site/gerard-berry/seminar-2008-01-25-11h30.htm

Philippe Flajolet, G. Nigel Martin.
Probabilistic Counting Algorithms for Data Base Applications.
Journal of Computer and System Sciences, 31(2) : 182-209 (1985).