

Xebia



WE ARE A GROUP OF
PASSIONATE AGILE
SOFTWARE DEVELOPMENT EXPERTS!

HyperLogLog

Florent Le Gall @flornt
Développeur Full Stack

HyperAgendaAgenda

- L'algorithme démystifié
 - Le principe mathématique
 - Les différentes étapes
- Son implémentation guidée par les tests
 - Hands-on en Scala ou en Java

HyperLogLog

- Un algorithme pour compter les occurrences uniques
 - Combien d'adresses IPs uniques dans mes logs ?
 - Combien de mots dans toutes les oeuvres de Shakespeare ?

Compter c'est facile !

- 1, 2, 3, ...



Compter c'est facile !

- La méthode naïve :
 - `Set<String> counter = new HashSet<>()`
 - `counter.add(sheep1);`
 - `counter.add(sheep2);`
 -
 - `counter.add(sheep1337);`
 - `return counter.size()`

La méthode naïve

- Avantages
 - Le résultat est exact
- Inconvénients
 - Nécessite de stocker toutes les occurrences uniques
 - Difficile à répartir sans répliquer plein de données !

Comment stocker moins ?

- Ne pas stocker les mots, stocker uniquement les valeurs de hachage
- Assez peu économe, on a toujours besoin d'autant de valeurs que de mots

L'approche Bitmap

- Pour chaque mot : on calcule sa valeur de hachage
- On stocke la présence ou l'absence de la valeur de hachage dans un tableau de bits.
- À la position donnée par les N-premiers bits de la valeur de hachage
- Il suffit à la fin de compter le nombre de valeurs marquées

L'approche Bitmap

- Nécessite de bien dimensionner le tableau
 - Si trop petit : on va avoir beaucoup de collisions et des résultats complètement faux
- Il faut connaître à l'avance une approximation de ce qu'on cherche :(

L'approche probabiliste

Soit un ensemble de nombres aléatoires

0101XY...
1111XY...
0100XY...
0000XY...
1110XY...
0110XY...
0111XY...
1000XY...
1001XY...
0011XY...
1011XY...
0001XY...
1100XY...
1101XY...
0010XY...
1010XY...

0000XY

On a une probabilité de $1/16$ de trouver
un nombre qui commence par 0000

L'approche probabiliste

À l'inverse

0000XY

0101XY...
1111XY...
0100XY...
0000XY...
1110XY...
0110XY...
0111XY...
1000XY...
1001XY...
0011XY...
1011XY...
0001XY...
1100XY...
1101XY...
0010XY...
1010XY...

On rencontre un seul nombre
commençant par 4 zéros

La cardinalité de l'ensemble est
probablement de 16 : (2^4)

L'approche probabiliste

- Pour chaque mot
 - On calcule sa valeur de hachage
 - On compte le nombre de zéros au début de cette valeur
 - On conserve le nombre de zéros le plus élevé parmi toutes les valeurs rencontrées
- Une approximation grossière est :

$$2^{MaxZerosCount}$$

L'approche probabiliste

- Oui, mais c'est très grossier !
 - Il suffit d'une seule valeur de hachage très basse pour déterminer la cardinalité
- La précision est plutôt faible : 32 valeurs possibles pour 4 milliard de résultats
- Le résultat est exponentiel
- La marge d'erreur est élevée

Comment faire mieux ?

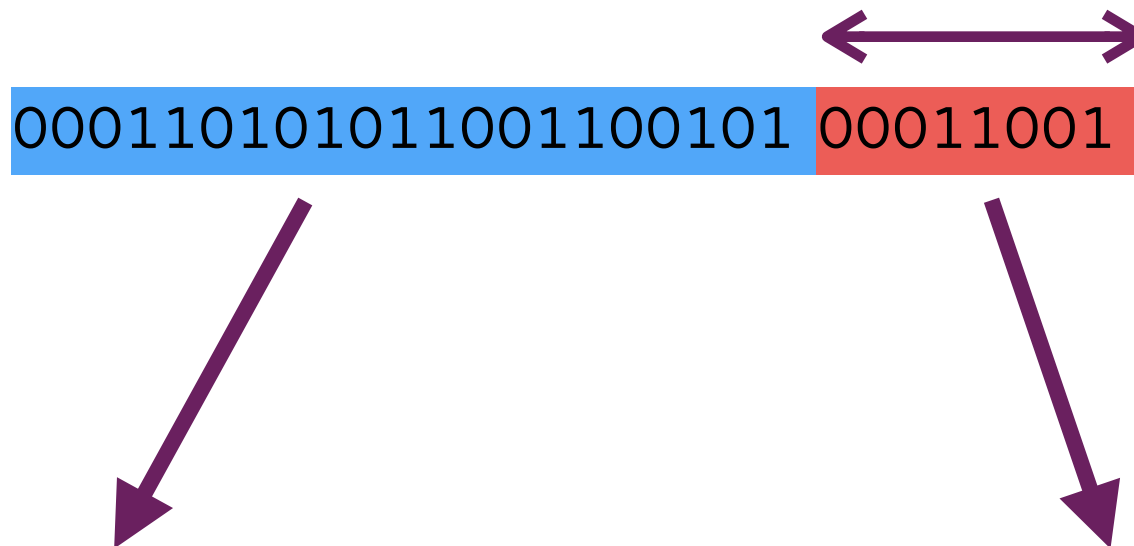
- Distribuer l'erreur sur plusieurs registres
- Pour chaque valeur de hachage :
 - Lui attribuer un registre
 - Compter le nombre de zéros au début de la valeur
 - Retenir le maximum du nombre de zéros par registre
- Faire la moyenne des registres !

*LogLog

Pour une valeur de hachage donnée

N bits pour l'index, soit 2^N registres

000110101011001100101 00011001



On détermine la position
du premier 1 (ici 4)

On obtient l'index à
utiliser (ici 25)

*LogLog

Pour la valeur

000010101011001100101 0101

On détermine son registre
registerIndex = 5

On détermine la position du 1er "1"

firstOneRank = 5

$\text{register}(5) = \max(\text{register}(5), \text{firstOneRank})$



0	5
1	4
2	5
3	6
4	3
5	4
6	5
7	6
8	3
9	6
10	5
11	4
12	3
13	5
14	6
15	5

*LogLog

Pour la valeur

000010101011001100101 0101

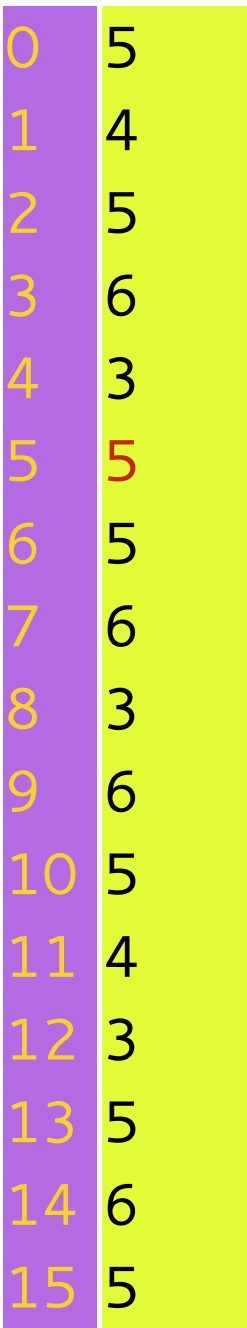
On détermine son registre
registerIndex = 5

On détermine la position du 1er "1"

firstOneRank = 5

$\text{register}(5) = \max(\text{register}(5), \text{firstOneRank})$

On met à jour le registre



0	5
1	4
2	5
3	6
4	3
5	5
6	5
7	6
8	3
9	6
10	5
11	4
12	3
13	5
14	6
15	5

LogLog

- LogLog utilise la moyenne arithmétique

$$\textit{count} = n * 2^{\frac{1}{n} \sum_{i=0}^n \textit{register}_i} * \textit{biasCorrection}$$

- n : le nombre de registres
- $\textit{register}_i$: la valeur du registre i
- $\textit{biasCorrection}$: la constante de correction

HyperLogLog

- HyperLogLog utilise la moyenne harmonique

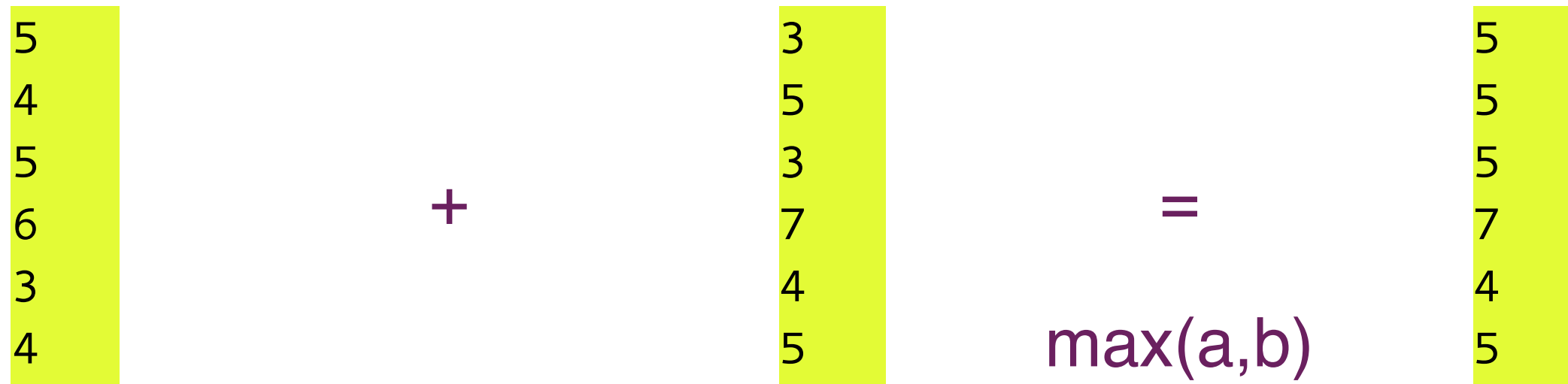
$$count = \frac{n^2 * biasCorrection}{\sum_{i=0}^n \frac{1}{2^{register_i}}}$$

- n : le nombre de registres
- $register_i$: la valeur du registre i
- $biasCorrection$: la constante de correction

Est-ce que ça marche bien ?

- LogLog :
 - Pour m registres : Précision: $1.30/\text{Sqrt}(m)$
 - 1024 registres (1ko) \rightarrow 4% d'erreur
- HyperLogLog :
 - Pour m registres : Précision: $1.04/\text{Sqrt}(m)$
 - 512 registres (1ko) \rightarrow 4% d'erreur

Fusionner deux compteurs



- La fusion (+) est une opération associative
- Le compteur vide est un élément neutre
- C'est donc un
- Monoïde !!

Fusionner deux compteurs

- Permet de répartir le décompte sur plusieurs environnements
- Chaque environnement remplit ses registres
- Les registres sont fusionnés plus tard
- Peu de données sont à transférer : juste un set de registres

Des algorithmes récents

- LogLog
 - Publié en 2003 par Marianne Durand & Philippe Flajolet (INRIA)
- HyperLogLog
 - Publié en 2007 par Philippe Flajolet, Eric Fusy, Olivier Gandouet et Frédéric Meunier (INRIA / LIRMM)

Pour approfondir

- Présentation à Devovx de DuyHai DOAN
 - http://cfp.devovx.fr/2015/talk/HLV-3716/Algorithmes_distribues_pour_le_Big_Data
- Les papiers sur l'algorithme :
 - LogLog : <http://algo.inria.fr/flajolet/Publications/DuFl03.pdf>
 - HyperLogLog : <http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>

Vous avez une heure !



- Scala or Java choose your weapon !
- <https://github.com/flegall/xke-hyperloglog>

Pour conclure

- C'est puissant, c'est très approprié si vos données sont réparties
- Même si c'est simple, ne les implémentez pas vous-même: il y'a pas mal d'implémentations déjà écrites
- <https://github.com/twitter/algebird>
(utilisé par Spark)

Questions





Merci!