



# Recherche d'image par le contenu

**Master1 – Analyse et traitement d'images - GMIN215**

Alexis JOLY - [alexis.joly@inria.fr](mailto:alexis.joly@inria.fr)

November 1<sup>st</sup> 2012

# Plan

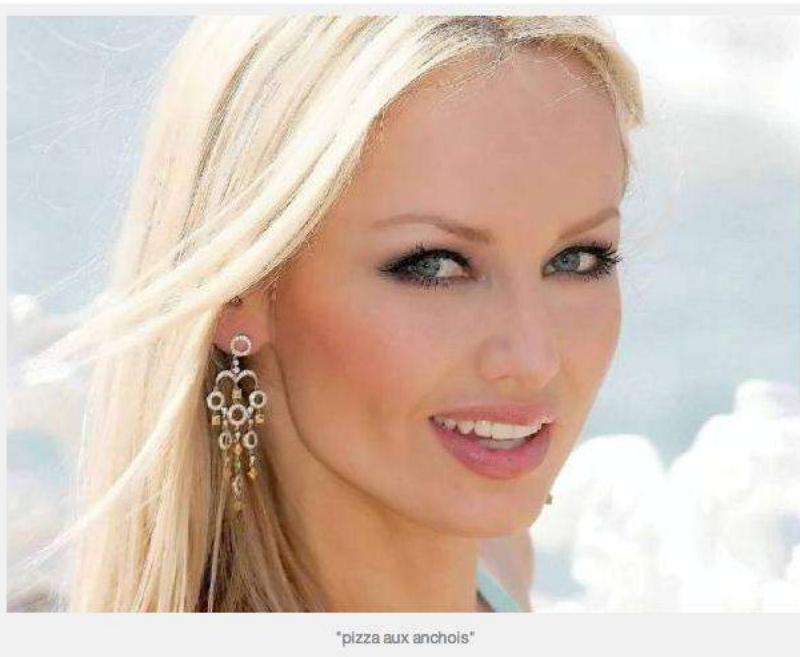
**Recherche d'information par le contenu I:** principes, applications, malédiction de la dimension, méthodes de partitionnement et de recherche

**Recherche d'information par le contenu II:** Présentation détaillée des arbres KD aléatoires (randomized kd-trees), Présentation détaillée de LSH et ses variantes, Présentation de deux bibliothèques (FLANN et VLFeat), KNN-graphs

# **Introduction, Principes et Applications de la recherche par le contenu**

# Introduction

## Recherche d'Informati uniquement le conten



### Pas de reconnaissance d'image pour Google.

L'unique objectif du test est de valider le fait que l'image ci-dessus se positionnera dans GG images pour la requête « pizza aux anchois ». Il n'y a pas de raison que ce ne soit pas le cas. Pour le moment, Google ne reconnaît pas les images, mais il prend en compte l'environnement textuel de celles-ci pour en déterminer le sujet. Et l'environnement ici, il est bien plus orienté **pizzéria** que mannequinat.

Maintenant, au train ou vont les évolutions techniques de la reconnaissance d'image, il y a fort à parier que les moteurs sauront un de ces jours différencier la jolie dame ci-dessus d'une vulgaire pizza.

### Que dire d'autre sur cette pizza ?

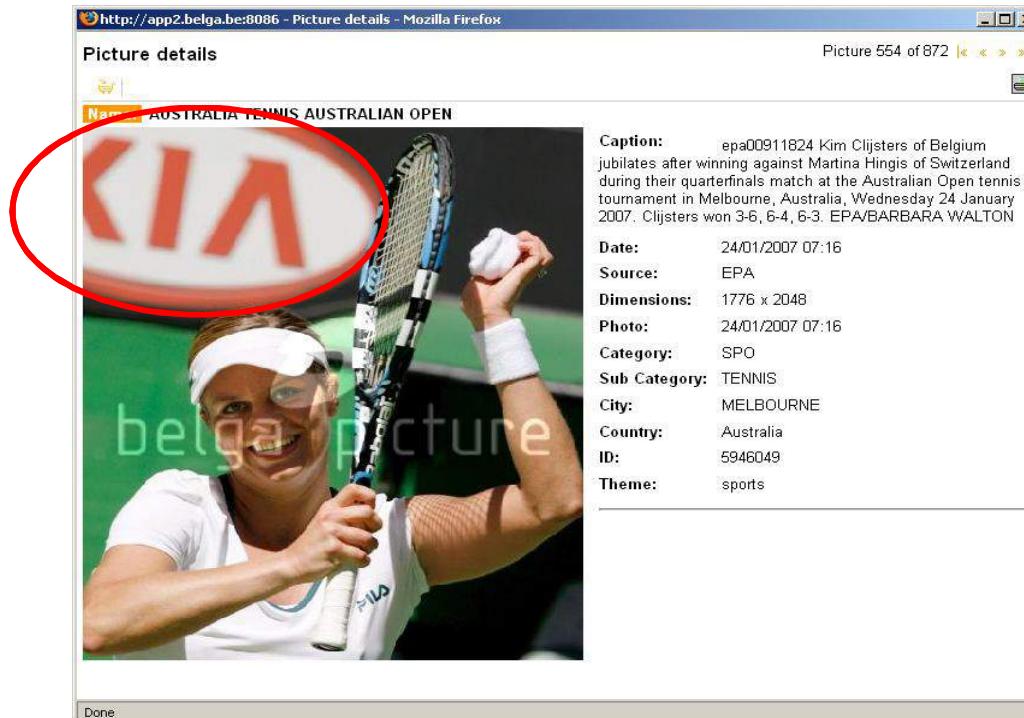
On peut dire comme pour notre test que cette recette de pizza ne fait pas l'unanimité. En effet, les anchois sont accueillis comme un met délicat par certains, et comme le pire pour les papilles par d'autres.

Pour les hurluberlus qui voudraient en savoir plus sur ce sujet, [passez voir ici](#), mais globalement, on s'en fiche un peu, tout ce qui nous intéresse, c'est de voir comment notre pizza va prendre chez le roi des moteurs.

### La recette de la pizza aux anchois

# Introduction

L'indexation et la recherche par le contenu visuel exploite l'information contenue dans les images



# Introduction

## Approches Content-to-text

- Reconnaissance d'objets, de visages, de textes (OCR)
- Speech-to-text, reconnaissance de locuteurs

Utilisation d'outils classiques d'indexation et de recherche de données

## Approches Content-based

- Utilisation de **descripteurs** (*signatures* ou *features*) = des vecteurs calculés à partir du contenu visuel ou audio caractérisant les couleurs, les formes, les fréquences, etc.
- Requête par l'exemple (Shazam, Google similarity search, Goggles)
- Query-by-window, Query-by-sketch, Query-by-singing, Query-by-tapping

# Introduction

## Approches Content-to-text

- Reconnaissance d'objets, de visages, de textes (OCR)

- Speed

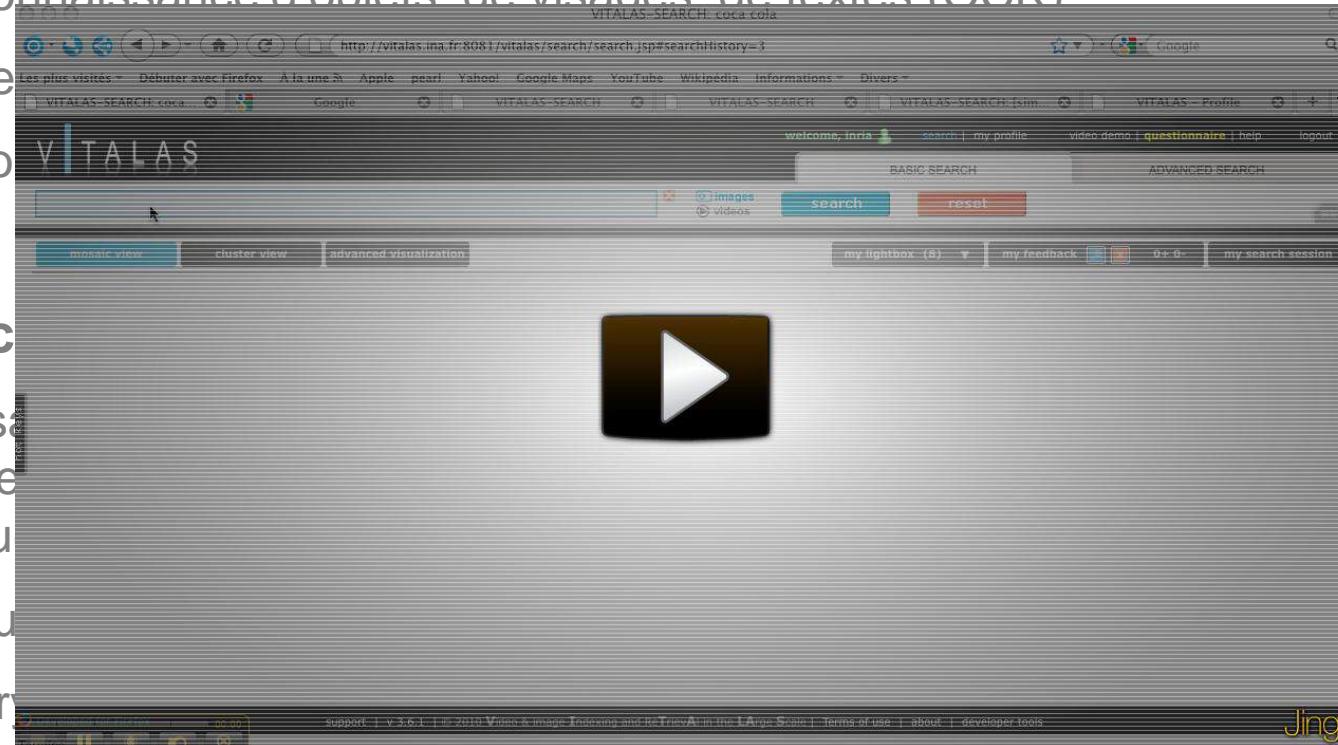
Utilisation

## Approches

- Utilisation de contexte et de fréquence

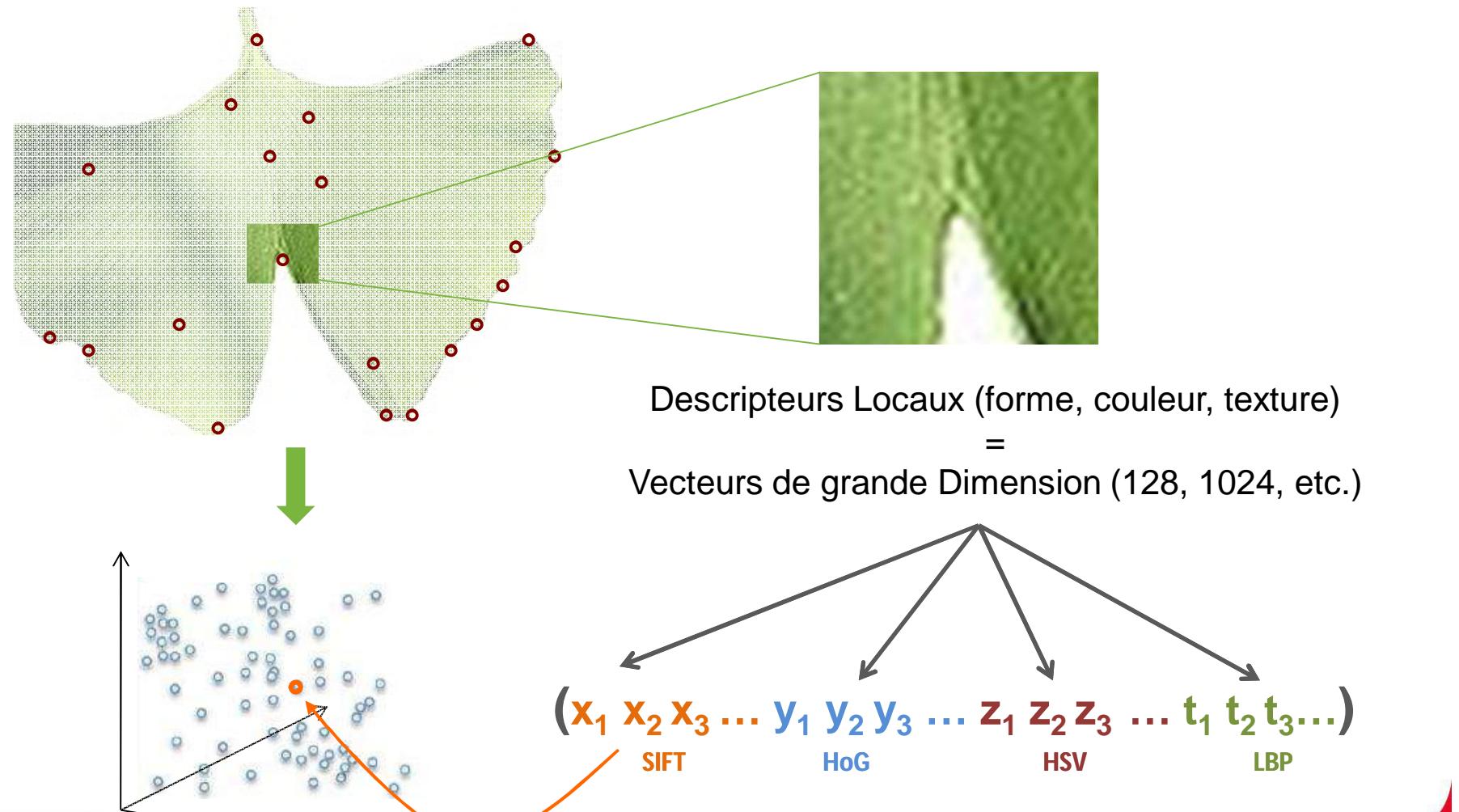
- Requête

- Query



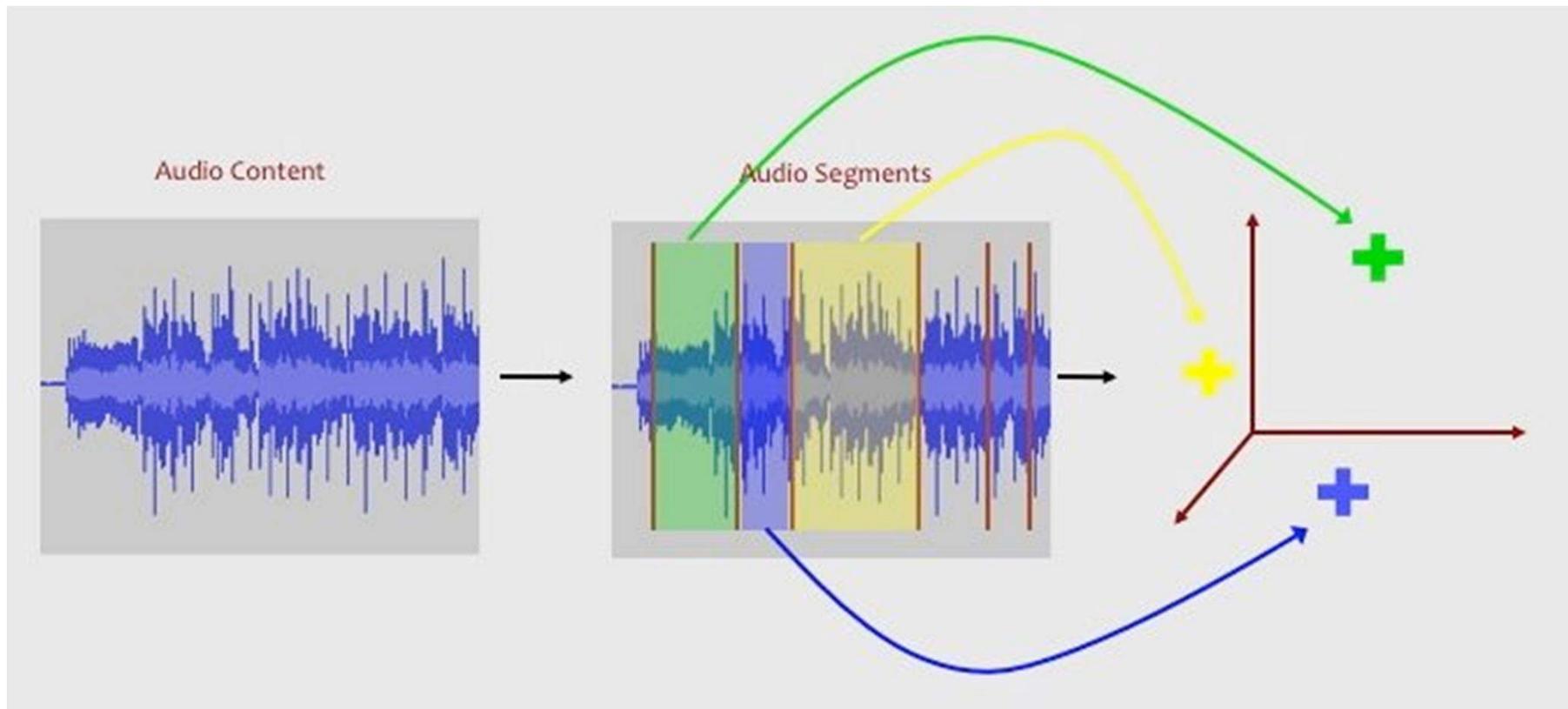
# Principe 1/4

## Des signatures du contenu visuel



## Principe 2/4

### ou du contenu audio



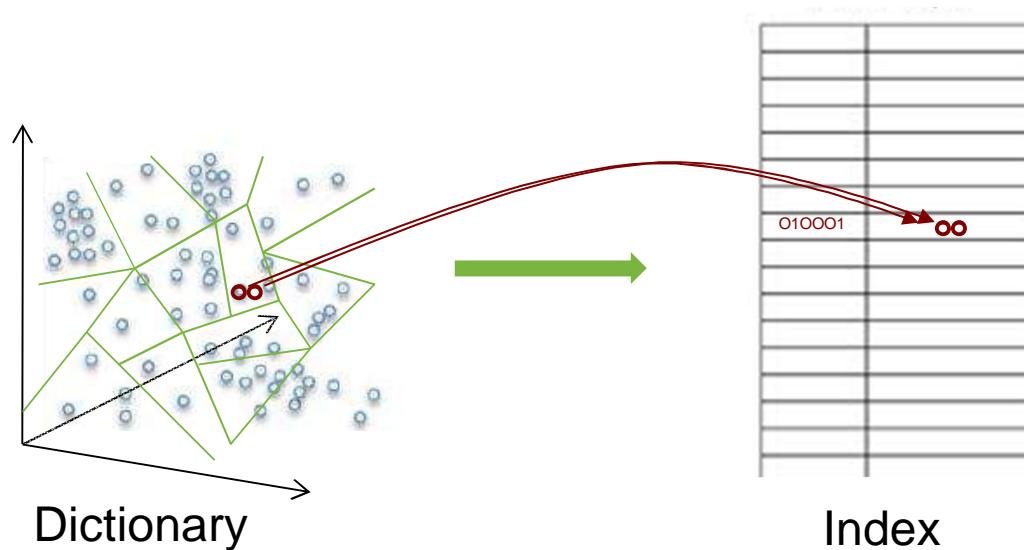
## Principe 3/4

### Des techniques d'indexation

Objectif: comparer efficacement une **signature requête** avec les **signatures de la base**

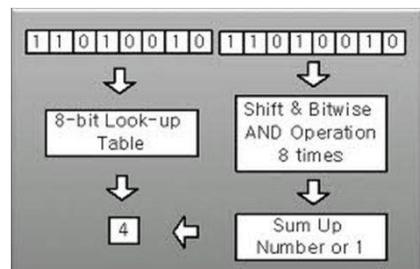
Problème: des **centaines de milliards** de comparaisons de vecteurs si on les traite exhaustivement + croissance exponentielle

Solution = structures d'indexation, **VQ**, hachage, compression, codage, encapsulation, regroupement, etc.



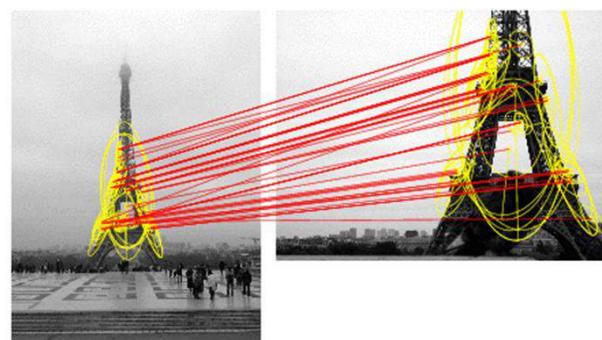
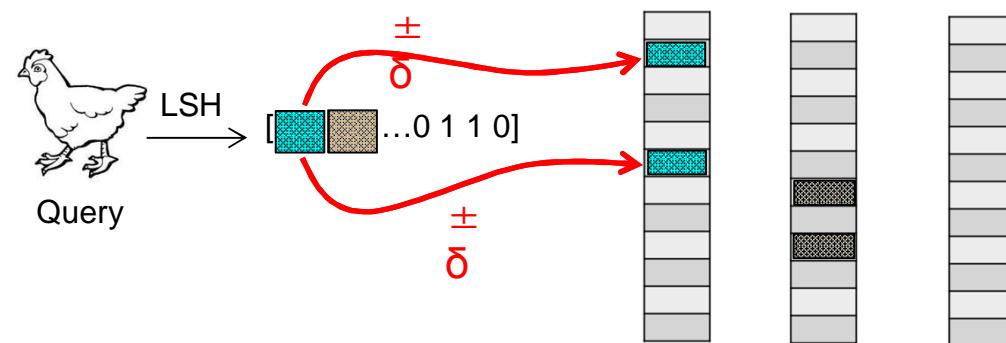
## Principe 4/4

### Des algorithmes de recherche par similarité efficaces et scalables

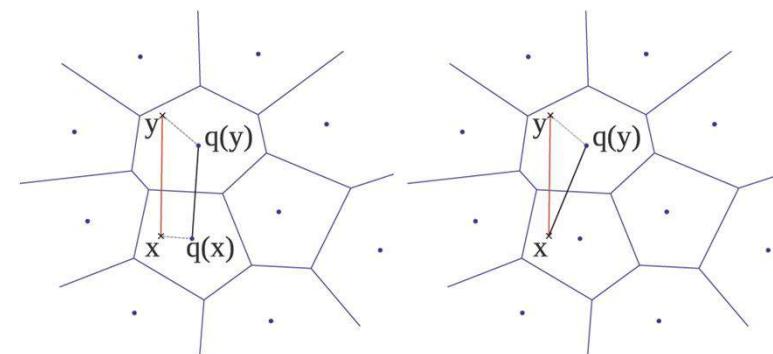


LUT-based Hamming distance

multi-probe queries in hash tables



Weak geometry ( $\sigma, \theta$ )



Asymmetric distance

# Les applications 1/5

## Détection de copies vidéo par le contenu

Monitoring de flux TV ou Web en temps-réel

15 M de vidéos indexées chez Youtube, 500 000 h de TV à l'INA

Forte robustesse aux attaques & transformations



# Les applications 2/5

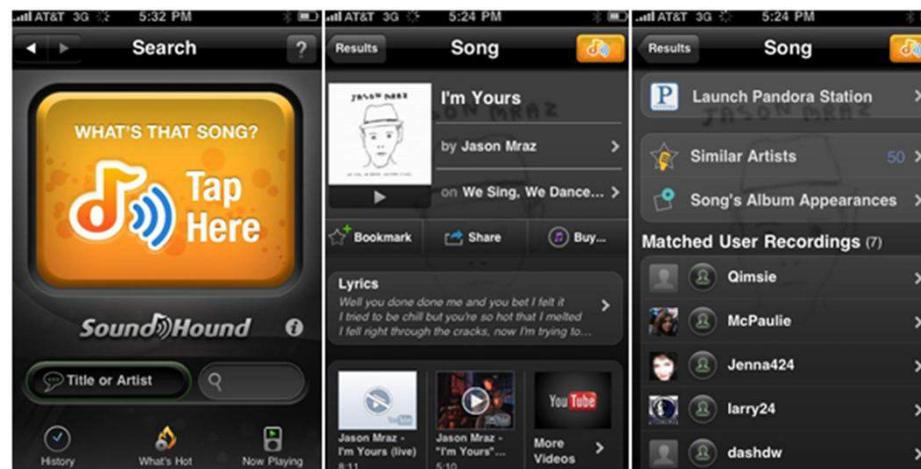
## Identification de chansons / musiques

Shazam > 10 Millions de titres

MusicID > 40 Millions titres

## Query-by-humming (plus compliqué)

SoundHound >10 Millions de titre



## Les applications 3/5

### Recherche d'images similaire (ou near-duplicates)



Google > 50 Milliards d'images

LTU > 100 Millions images

### Reverse image search

Tineye > 3 Milliards d'images

### Mobile search

Kooaba > 10M images

Goggles > ??



## Les applications 4/5

### L'identification des végétaux

Pl@ntNet > 3800 espèces

LeafSnap > 250 espèces



### L'identification des animaux

Encore à l'état de recherche

Reconnaissance de chants d'oiseaux

Vidéosurveillance de poissons dans les récifs

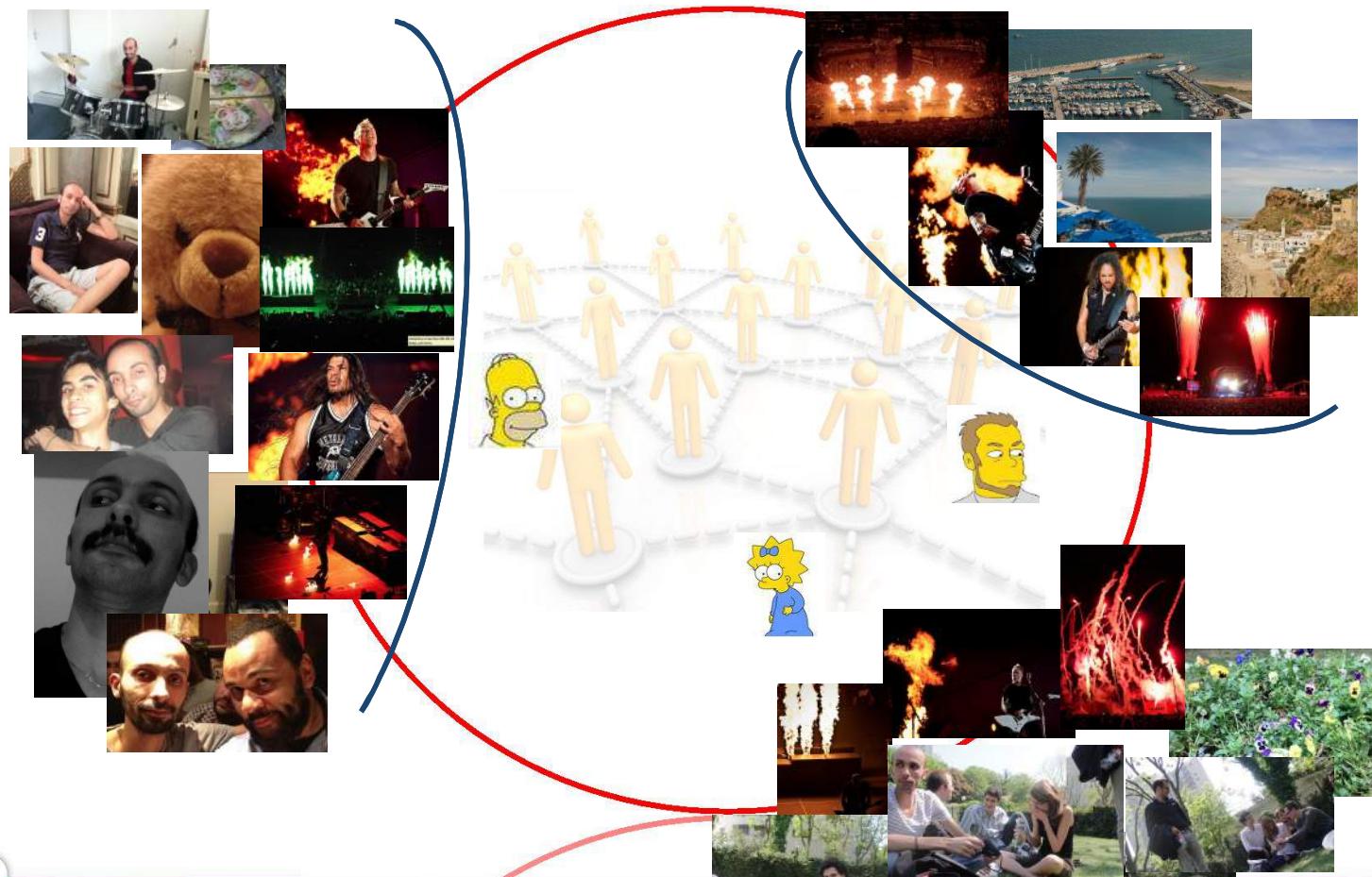
Suivi de baleines (images de la queue)



# Les applications 5/5

## Découverte d'évènements par contenu visuel

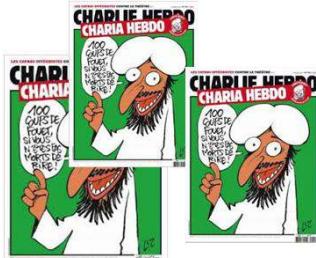
### 1. Dans les médias sociaux



# Les applications 5/5

## Découverte d'évènements par contenu visuel

1. Dans les médias sociaux
2. A travers différents médias (TV, web , AFP)



Arte TV

Lemonde



corto74.blogspot.com



0 to 2011-10-17 to 2011-10-24 to 2011-10-31 to 2011-11-07 to 2011-11-14 to  
17 2011-10-24 2011-11-07 2011-11-14 2011-11-21

2 to 2011-09-19 to 2011-09-26 to 2011-10-03 to 2011-10-10 to 2011-10-17 to 2011-10-24 to 2011-10-31 to  
19 .6 to 2011-10-03 to 2011-10-10 to 2011-10-17 to 2011-10-24 to 2011-10-31 to 2011-11-07  
03 2011-10-10 2011-10-17 2011-10-24 2011-10-31 2011-11-07

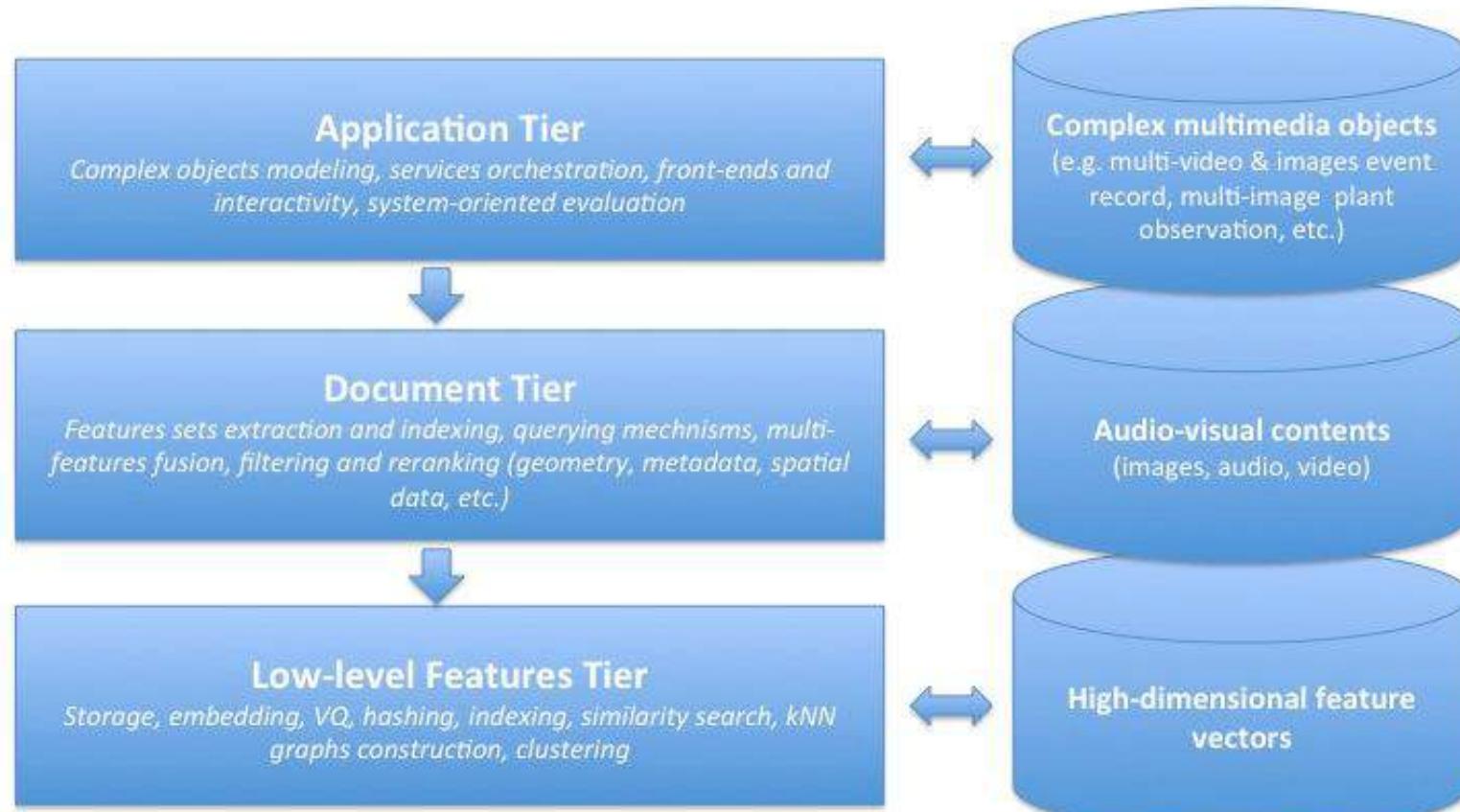
**2011-10-17 to 2011-10-24**



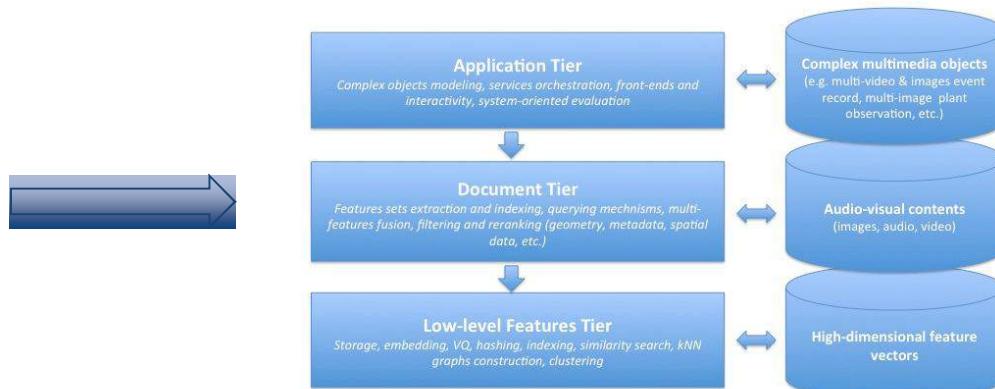
Le régime de Mouammar Khadafi aura duré 42 ans. Un régime inauguré à la fin des années 60 par un jeune officier qui deviendra un dictateur redouté dans son pays et au delà des frontières libyennes, notamment en raison de son soutien au terrorisme.

keywords : [ Libye ] [ Mouammar Khadafi ] [ OTAN ] [ FRA ]

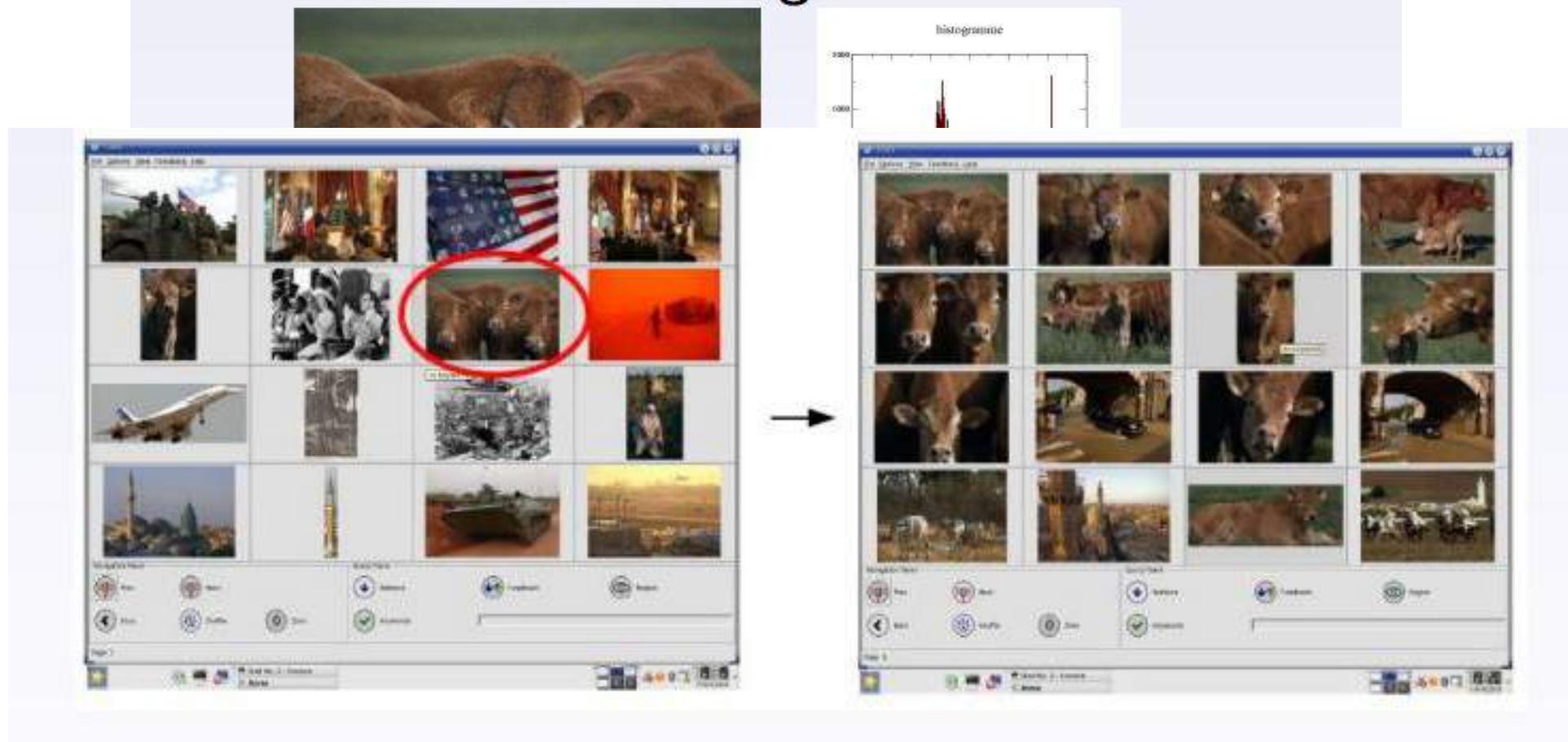
# Architecture



# Quelques descripteurs de contenu et mesures de similarité associées



- Cas d'école: Histogramme HSV



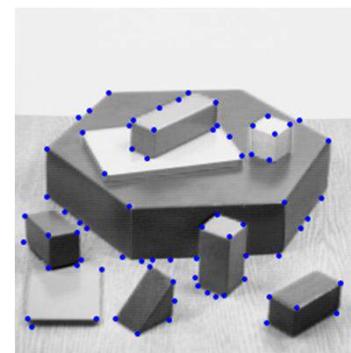
# La détection de points ou de régions d'intérêt

## Détection de points d'intérêt par différence de Gaussienne

Grille dense



Détecteur de Harris



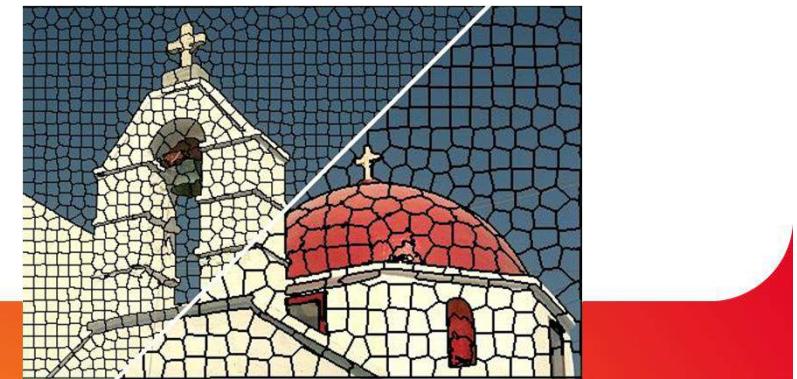
Maximally Stable Extremal Regions



Difference of Gaussians



Superpixels



## Les descri

## Calculés auto

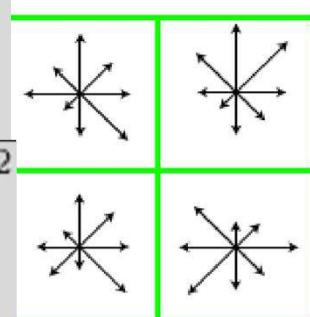
- SIFT (Lowe, 2004)

133.92 135.88 14.38 -2.732  
3 12 23 38 10 15 78 20 39 67 42 8 12 8 39 35 118 43 17 0  
0 1 12 109 9 2 6 0 0 21 46 22 14 18 51 19 5 9 41 52  
65 30 3 21 55 49 26 30 118 118 25 12 8 3 2 60 53 56 72 20  
7 10 16 7 88 23 13 15 12 11 11 71 45 7 4 49 82 38 38 91  
118 15 2 16 33 3 5 118 98 38 6 19 36 1 0 15 64 22 1 2  
6 11 18 61 31 3 0 6 15 23 118 118 13 0 0 35 38 18 40 96  
24 1 0 13 17 3 24 98  
132.36 99.75 11.45 -2.910  
94 32 7 2 13 7 5 23 121 94 13 5 0 0 4 59 13 30 71 32  
0 6 32 11 25 32 13 0 0 16 51 5 44 50 0 3 33 55 11 9  
121 121 12 9 6 3 0 18 55 60 48 44 44 9 0 2 106 117 13 2  
1 0 1 1 37 1 1 25 80 35 15 41 121 3 0 2 14 3 2 121  
51 11 0 20 93 6 0 20 109 57 3 4 5 0 0 28 21 2 0 5  
13 12 75 119 35 0 0 13 28 14 37 121 12 0 0 21 46 5 11 93  
29 0 0 3 14 4 11 99  
102.33 26.00 13.65 2.815

je L2

Métrique pour comparer deux vecteurs SIFT =  
Distance L2 = distance Euclidienne

$$d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$
$$= \sqrt{\sum_{i=1}^n (q_i - p_i)^2}.$$



- Autres. SIFT

Binary Pattern

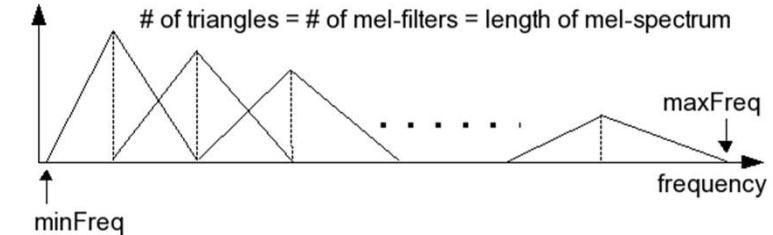
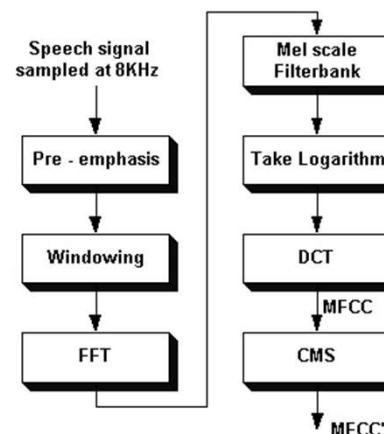
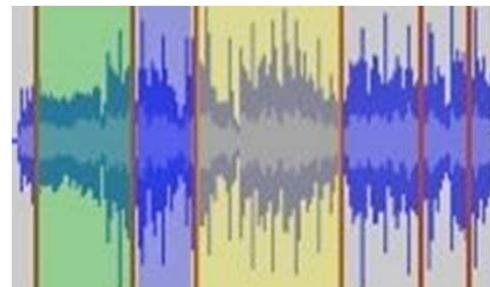
19 10 3 1 29 94 54 54  
88.61 134.04 13.25 2.721  
27 56 20 6 9 21 12 17 33 120 114 6 4 8 8 0 41 106 23 9  
19 42 3 0 45 54 13 0 14 26 1 3 45 12 6 22 24 22 29 35  
61 52 8 1 35 92 55 23 120 106 0 0 16 24 5 35 92 4 0 0  
33 40 9 30 81 74 22 31 4 2 5 8 52 77 34 9 39 37 15 32  
120 38 3 0 5 17 28 120 41 5 19 10 7 32 89 63 34 27 4 0  
0 6 33 17 40 60 29 0 0 8 23 22 120 113 79 5 0 0 2 16  
9 21 120 58 0 1 10 3

SIFT, Local

# Les descripteurs audio les plus utilisés

Calculés sur segments fixes ou variables

- MFCC (Lowe, 2004), dimension = 13 (valeurs float), L2



- Autres: Linear Predictor Coefficients, Line Spectral Frequency, Loudness coefficients, OBSI, etc.
- MFCC + many others available in Yaafe

# Représentation vectorielle de documents textes

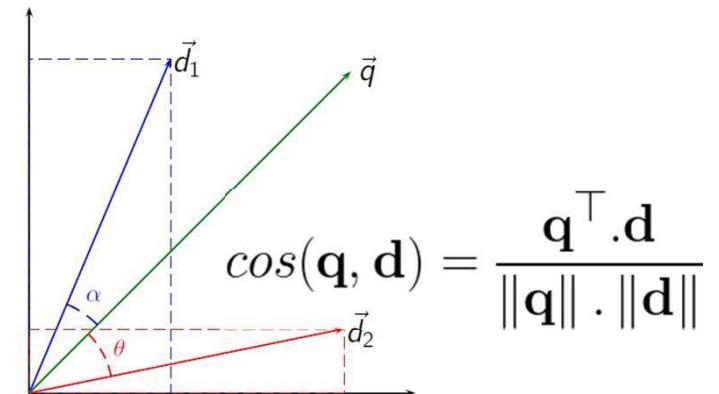
Documents et requêtes représentées par des vecteurs de fréquence d'apparition des mots

$$\begin{aligned} \mathbf{d}_j &= (w_{1,j}, w_{2,j}, \dots, w_{t,j}) \\ \mathbf{q} &= (w_{1,q}, w_{2,q}, \dots, w_{t,q}) \end{aligned}$$

Taille du dictionnaire (nombre de mots possibles) = dimension de l'espace de description (jusqu'à plusieurs centaines de milliers)

Mesure de similarité = **Produit scalaire** ou  
**Cosine measure**

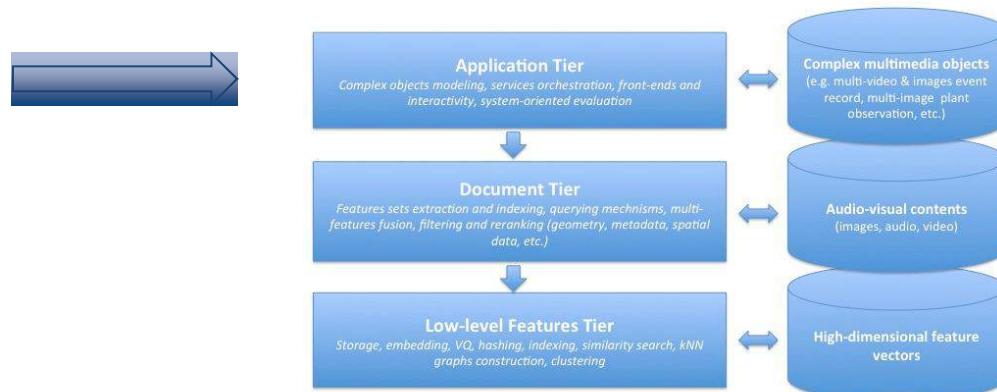
$$\mathbf{q}^\top \cdot \mathbf{d} = \sum_{i=1}^n q_i \cdot d_i = \|\mathbf{q}\| \cdot \|\mathbf{d}\| \cdot \cos(\mathbf{q}, \mathbf{d})$$



NB: La grande majorité des composantes étant nulles, on utilise en pratique une représentation parcimonieuse des vecteurs, e.g:

$\mathbf{q}: (\text{id\_mot1}, \text{fréquence\_mot1}) (\text{id\_mot2}, \text{fréquence\_mot2})$

# Un exemple de chaîne complète d'indexation et de recherche par le contenu



# Pl@ntNet: un exemple de système réel de recherche par le contenu visuel

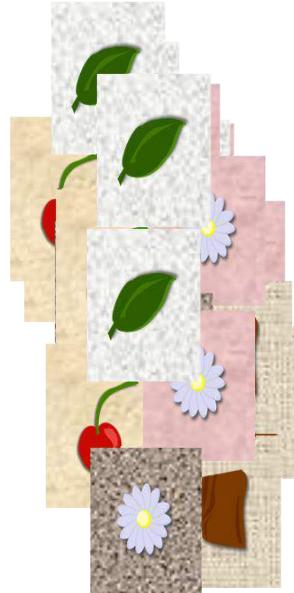
## L'application mobile Pl@ntNet

iPhone



# PI@ntNet: un exemple de système réel de recherche par le contenu visuel

## Indexation

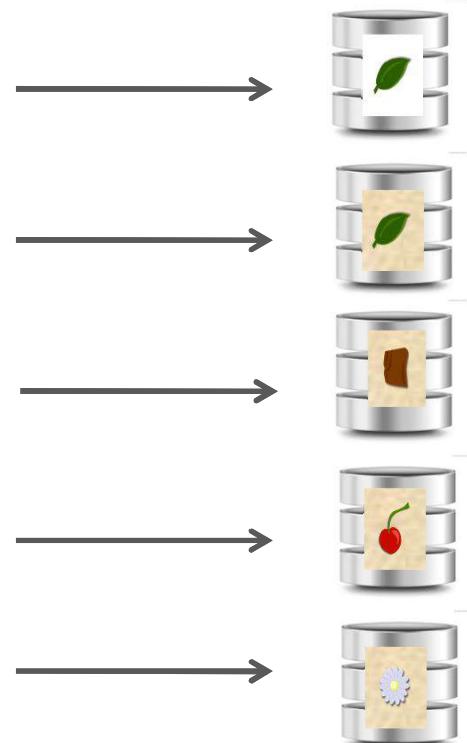


Database

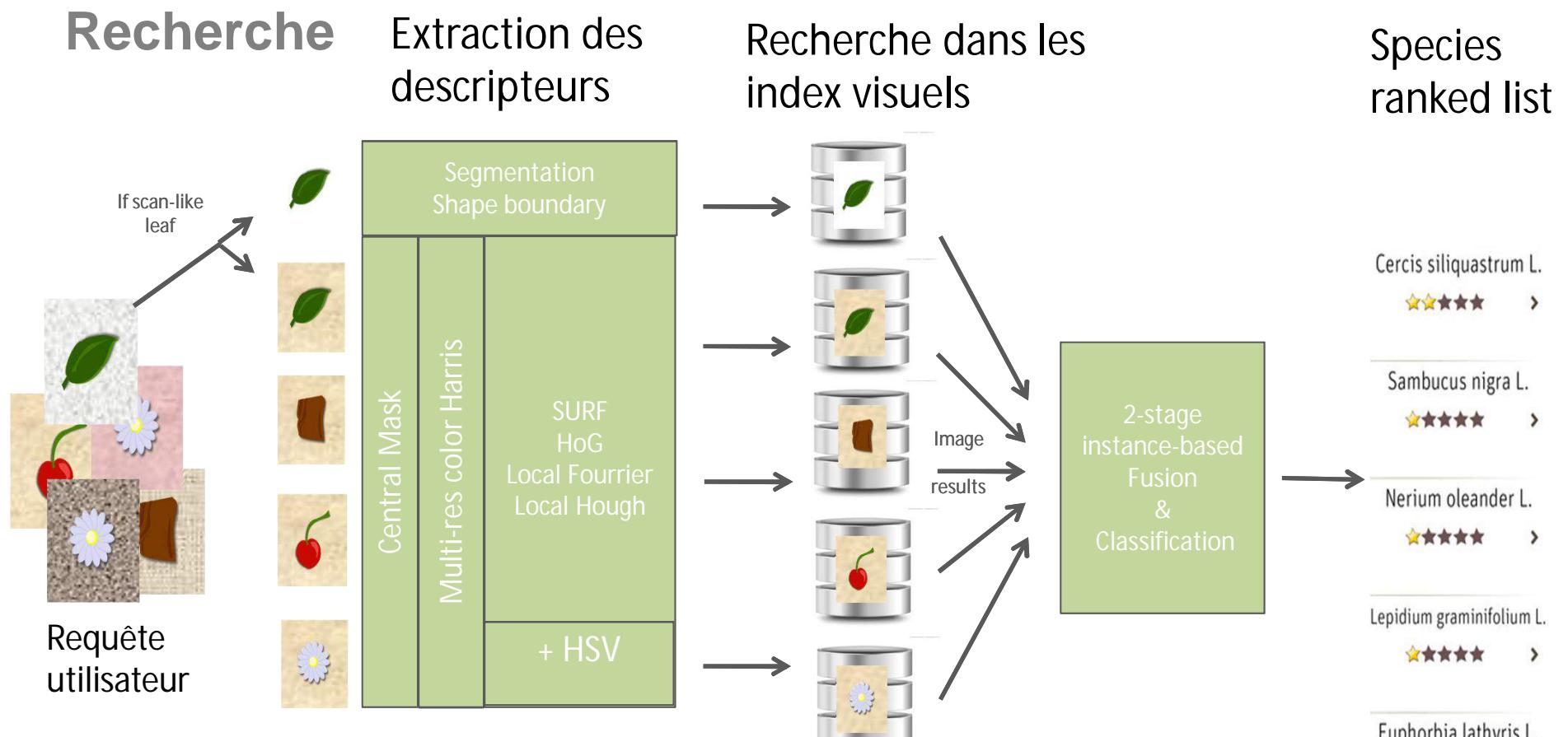
### Extraction des descripteurs



### Créations d'index visuels



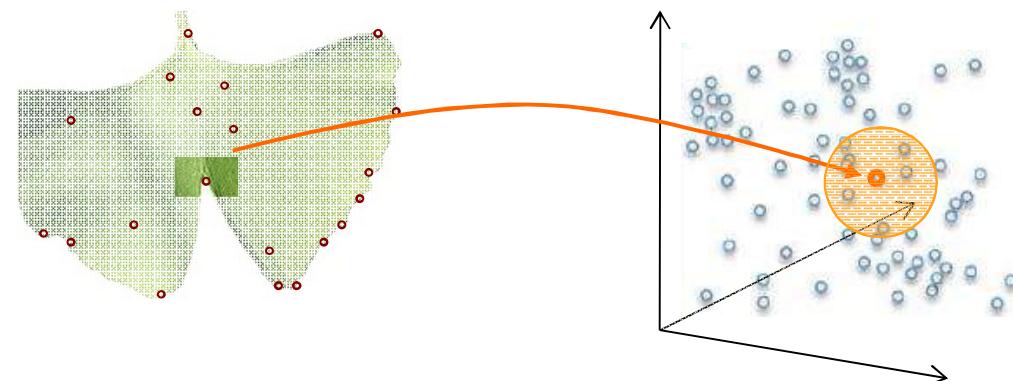
# PI@ntNet: un exemple de système réel de recherche par le contenu visuel



# PI@ntNet: un exemple de système réel de recherche par le contenu visuel

## Recherche d'une image dans un index visuel (appariement)

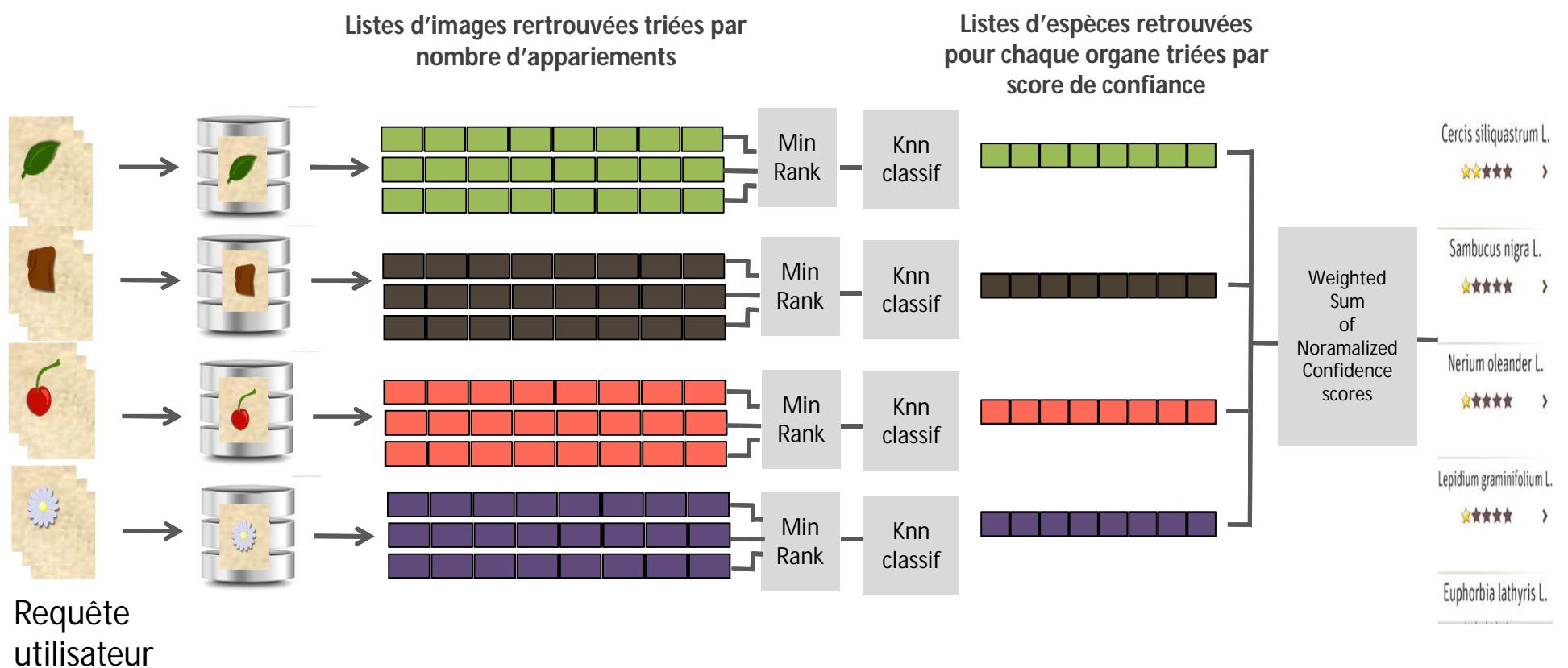
1. Pour chaque descripteur local, on recherche les descripteurs les plus similaires dans la base



2. On regroupe les descripteurs retrouvés appartenant à une même image de la base
3. On compte le nombre de descripteurs retrouvés par image (= le nombre d'appariements)
4. On trie les images retrouvées par nombre d'appariements décroissants

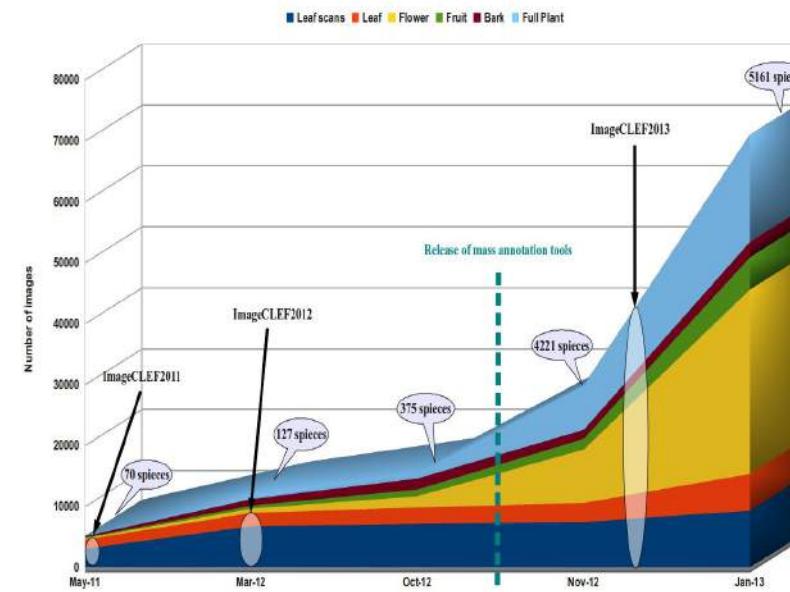
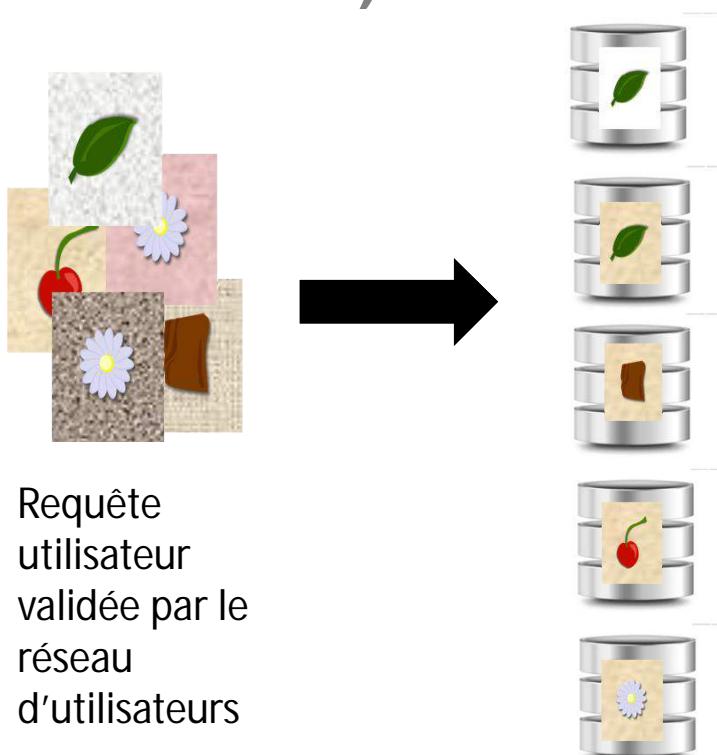
# PI@ntNet: un exemple de système réel de recherche par le contenu visuel

## Des listes d'images à la liste d'espèce



# PI@ntNet: un exemple de système réel de recherche par le contenu visuel

Enrichissement de la base: insertion des descripteurs dans les index (toutes les nuits, uniquement sur images validées)



# Méthode de recherche exhaustive et malédiction de la dimension



# Le problème

Recherche du (ou des k) plus proches voisins

Exemple

Espace de recherche: Espace  $\mathbb{U}$ , fonction similarité  $\sigma$



Desc. SIFT, dim=128, L2

Entrée: database  $S = \{p_1, \dots, p_n\} \subseteq \mathbb{U}$



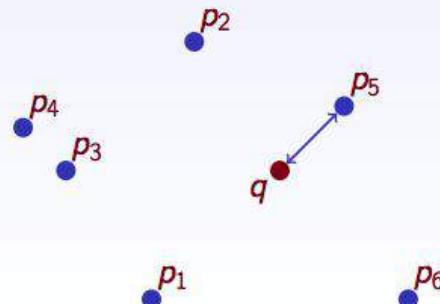
100K images x 1000 SIFT

Requête:  $q \in \mathbb{U}$

=

Tâche: trouver  $\operatorname{argmax}_{p_i} \sigma(p_i, q)$

100M de vecteurs SIFT



Problème fondamental abordé depuis l'époque de Voronoi (1908)

Il n'existe pas de solution exacte de complexité sous-linéaire en particulier pour les grandes dimensions

# La recherche exhaustive

1. Principe: parcours exhaustif de tous les vecteurs de la base et calcul de toutes les distances à la requête

```
for (int i=1; i<n ; i++) ComputeDistance(q,X[i]);
```

2. Recherche des top-k dans la liste des distances

- méthode naïve: on insère les éléments <i,dist[i]> dans un container (une liste de Map.Entry par exemple) et on utilise un sort

```
for (int i=1; i<n ; i++) {  
    list.add(<i, d(q,X[i])>); //insertion du n-ième élément  
}  
  
sort(list, comparator); //tri du container  
  
for (int j=1; j<k ; j++) knn.add(j,list.get(j)); //extraction des k premiers éléments
```

**Problème = sort est un algorithme de complexité O(n log n) qui peut être trop coûteux pour une recherche temps réel**

# La recherche exhaustive

1. Principe: parcours exhaustif de tous les vecteurs de la base et calcul de toutes les distances à la requête

```
for (int i=1; i<n ; i++) ComputeDistance(q,X[i]);
```

2. Recherche des top-k dans la liste des distances

- méthode **plus efficace**: on met à jour progressivement les top-k en conservant la position et la distance du k-nn courant

```
dist_max_knn=MAX_DIST;
for (int i=1; i<n ; i++) {
    if (ComputeDistance(q,X[i]) < dist_max_knn) { //si on a trouvé un meilleur voisin
        list.add(<i, d(q,X[i])>); //insertion dans la liste des knn
        [dist_max_knn,index_max_knn]=SearchFurthestElementInList(list); // cherche nouveau
max O(k)
        if (i>k) list.remove(index_max_knn); //retire nouveau max
    }
}
```

La complexité est  $O(n.k)$  dans le pire des cas mais en pratique beaucoup plus rapide car de moins en moins de meilleurs voisins

Problème = ne marche pas lorsque k est trop grand  $k \gg \log(n)$

# La recherche exhaustive

1. Parcours exhaustif de tous les vecteurs de la base et calcul de toutes les distances à la requête

```
for (int i=1; i<n ; i++) dist[i]=d(q,X[i]);
```

2. Recherche des top-k dans la liste des distances

- méthode pour k grand ( $>>\log N$ ): on utilise un Max heap ou une PriorityQueue

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

→ provides  $O(\log(n))$  time for the enqueueing and dequeuing methods: add & poll;



# La recherche exhaustive

1. Parcours exhaustif de tous les vecteurs de la base et calcul de toutes les distances à la requête

```
for (int i=1; i<n ; i++) dist[i]=d(q,X[i]);
```

2. Recherche des top-k dans la liste des distances

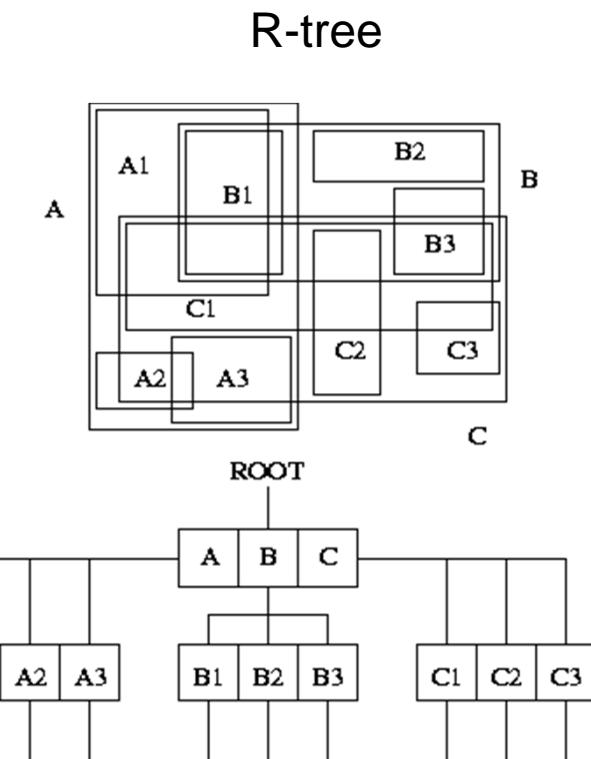
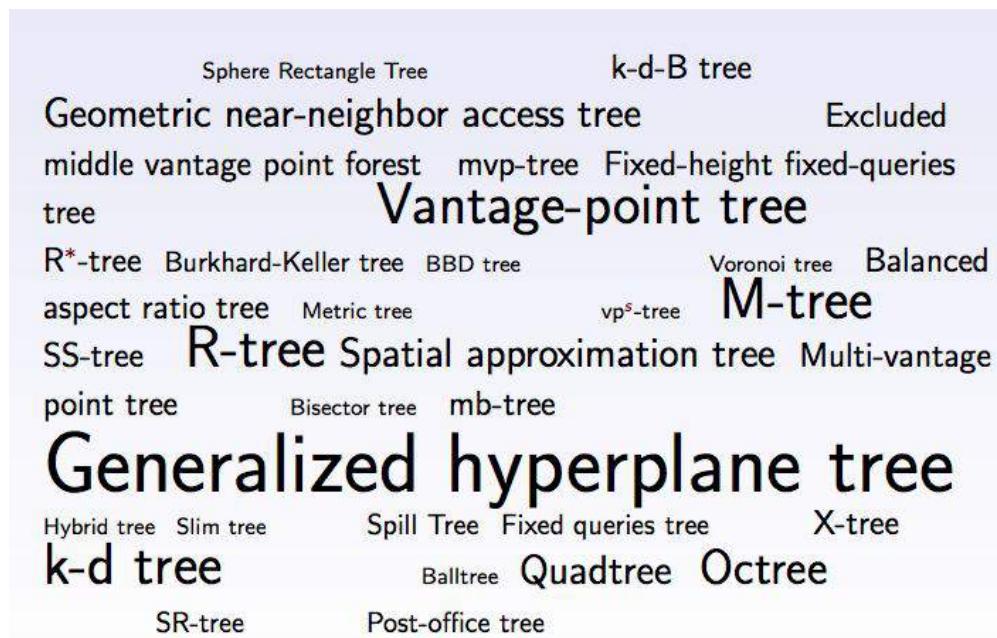
- méthode pour k grand ( $>>\log N$ ): on utilise un Max heap ou une PriorityQueue

```
PriorityQueue<DistElt> TopKqueue = new PriorityQueue<DistElt>();  
for (int i=1; i<n ; i++) {  
    TopKqueue.add(new DistElt(i, ComputeDistance(q,X[i]) ) );  
    if (i>k) TopKqueue.poll();  
}
```

Complexité  $O(n \log k)$



# Les structures d'indexation métriques et multi-dimensionnelles (1980-2000)

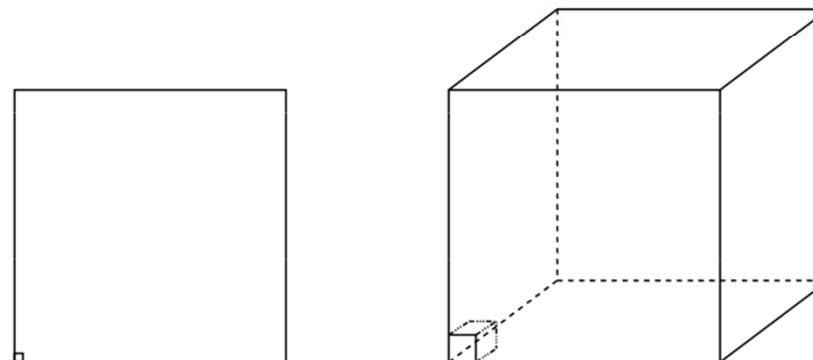


Weber a montré en 1998 qu'elles étaient moins efficaces que la recherche exhaustive pour  $\text{dim} > 16$  (environ)

# Curse of dimensionality (malédiction de la dimensionalité)

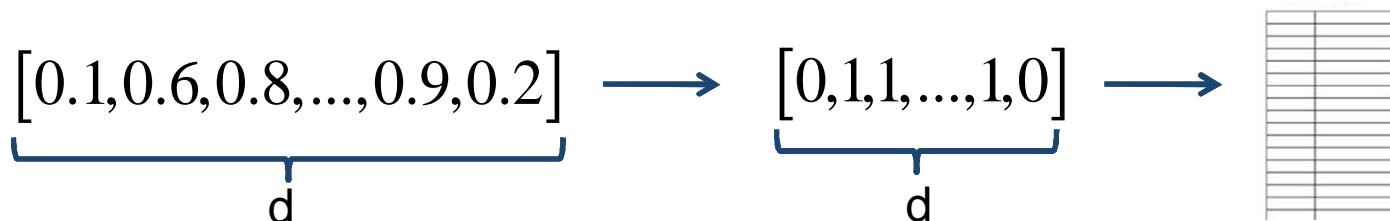
Lorsque la dimension de l'espace devient très grande il faut parcourir toutes les cellules de la partition pour être sûr de retrouver les k-nn exacts

- Assume 5000 points uniformly distributed in the unit hypercube and we want to apply 5-nn. Suppose our query point is at the origin.
  - In 1-dimension, we must go a distance of  $5/5000 = 0.001$  on the average to capture 5 nearest neighbors
  - In 2 dimensions, we must go  $\sqrt{0.001}$  to get a square that contains 0.001 of the volume.
  - In d dimensions, we must go  $(0.001)^{1/d}$



## Curse of dimensionality (malédiction de la dimensionalité)

La quantification binaire illustre bien l'aspect combinatoire du problème en grande dimension



Recherche à un rayon près  $r \approx \alpha \cdot d$  (en nb de bits dans l'espace d'arrivée)

$$\text{Nb de cases à visiter } n(d) = C_d^r = \frac{d!}{r!(d-r)!}$$

$$\frac{n(d+1)}{n(d)} > 1 \text{ et } \frac{n(d+1)}{n(d)} \approx \frac{1}{(1-\alpha)}$$

$\longrightarrow$  n(d): suite géométrique et donc une croissance exponentielle

Exemple:  $d=1000$ ,  $\alpha=0.05 \rightarrow n(d)=9.46e+84$

# **En pratique on utilise des méthodes approximatives et des heuristiques**

**Changements d'espace**

- techniques de réduction de la dimension

**Algorithmes de regroupement**

**Compression/quantification (avec perte)**

- Hachage

- Hamming embedding

**Algorithmes de recherche approximatifs**

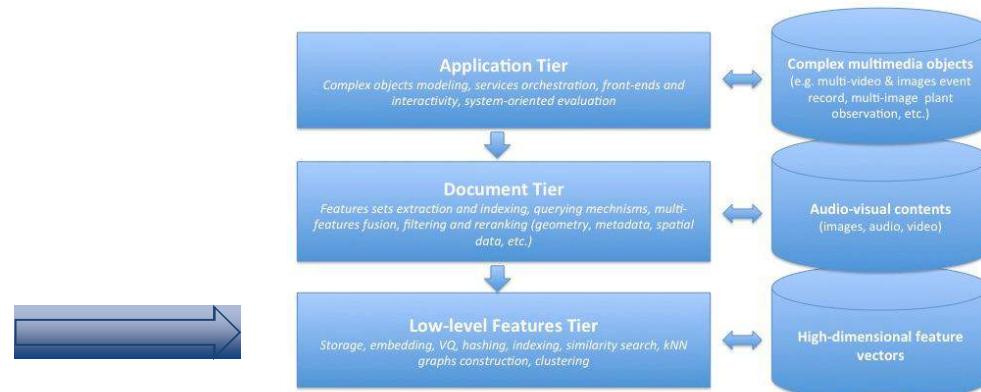
- à epsilon près

- statistiques

- heuristiques



# Les changements d'espace



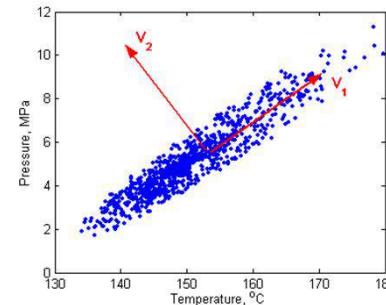
# Techniques de réduction de la dimension

## Analyse en composantes principales Projections aléatoires

} Les deux plus utilisées

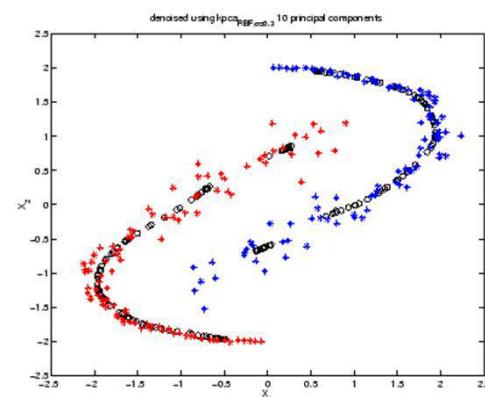
*Autres techniques linéaires:*

Analyse en composante indépendantes  
Multilinear subspace learning  
Latent semantic analysis



*Techniques non linéaires:*

Isomap  
Kernel PCA  
Autoencoder (réseaux de neurones)  
Manifold learning  
...



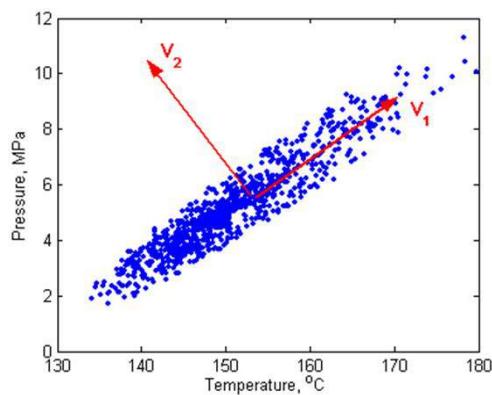
# Analyse en composantes principales

$$y = Ax$$

$y \in R^k$        $x \in R^d$

$A$  = matrice de projection  $d \times k$

Constituée des  $k$  vecteurs propres de  $\text{Cov}(X) = X^T X$   
ayant les valeurs propres les plus grandes



Intérêt:  
**Rotation** puis conservation des axes de variance maximale  
+  
Décorrélation (la nouvelle matrice de corrélation est diagonale)

# Analyse en composantes principales

$$y = Ax$$

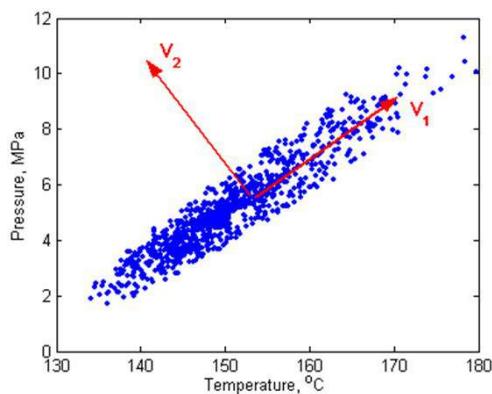
$y \in R^k$        $x \in R^d$

$A$  = matrice  $d \times k$

Méthode de calcul de A:

- 1) Centrage des données  $X \leftarrow X - \mu_X$

$\mu_X$ =centre de gravité des données



# Analyse en composantes principales

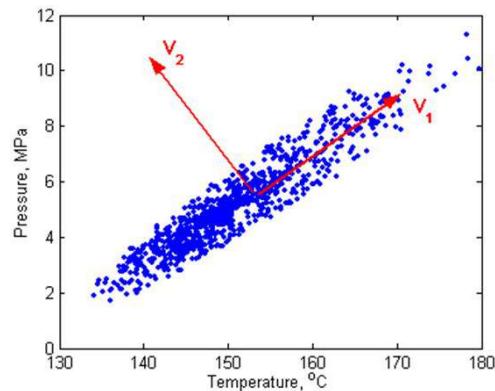
$$y = Ax$$

$y \in R^k$        $x \in R^d$

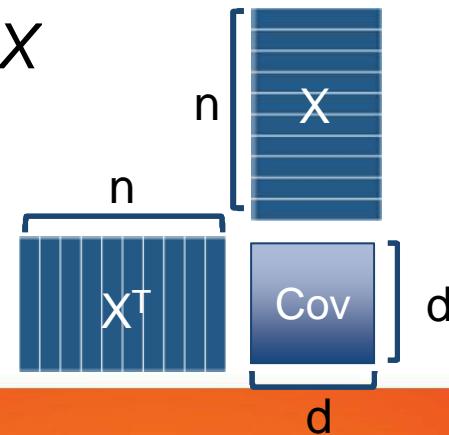
$A$  = matrice  $d \times k$

Méthode de calcul de A:

- 2) Calcul de la matrice de covariance sur un échantillon de  $n$  vecteurs  $\rightarrow O(n^2.d^2)$



$$\text{Cov} = X^T X$$



$$\text{Cov}(i, j) = \sigma_{i,j}^2$$

$$\text{Cov}(i, i) = \sigma_i^2$$

# Analyse en composantes principales

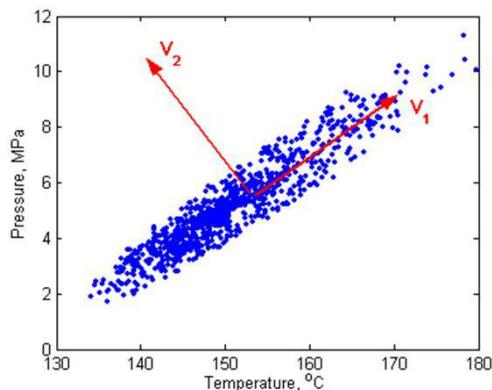
$$y = Ax$$

$y \in R^k$        $x \in R^d$

$A$  = matrice  $d \times k$

Méthode de calcul de A:

3) Diagonalisation de la matrice de covariance



$$\text{Cov} = P^T \Delta P$$

$$\left[ \begin{matrix} \text{Cov} \\ d \end{matrix} \right] = \left[ \begin{matrix} P^T \\ x \end{matrix} \right] \times \left[ \begin{matrix} \Delta \\ x \end{matrix} \right] \times \left[ \begin{matrix} P \\ x \end{matrix} \right]$$

# Analyse en composantes principales

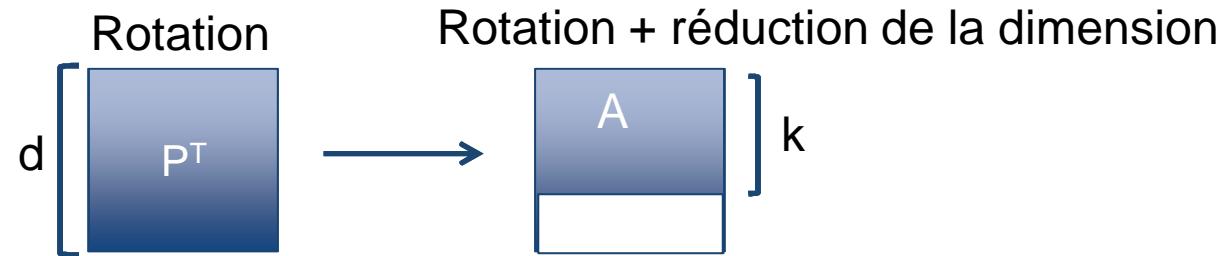
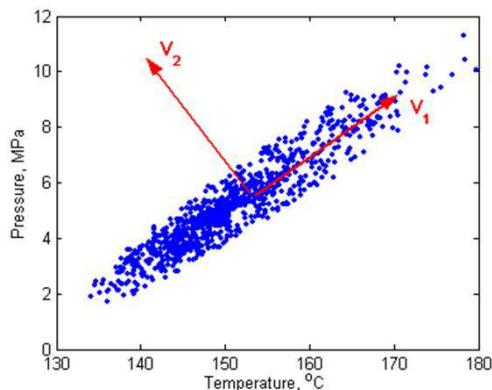
$$y = Ax$$

$y \in R^k$        $x \in R^d$

$A$  = matrice  $d \times k$

Méthode de calcul de A:

- 4) On ne conserve que les  $k$  vecteurs propres ayant les valeurs propres les plus élevées



# Projections aléatoires

$$y = Ax$$
$$\begin{aligned} y &\in \mathbb{R}^k \\ A &= \text{randn}(d, k) \\ &= \text{i.i.d sampled from Gaussian } N(0, 1) \end{aligned}$$

Un corollaire du lemme de Johnson-Lindenstrauss affirme qu'une telle transformation tend à préserver le produit scalaire

**Corollary 2.1.** Let  $u, v \in \mathbb{R}^d$  and that  $\|u\| \leq 1$  and  $\|v\| \leq 1$ . Let  $f = \frac{1}{\sqrt{k}}Ax$  where  $A$  is a  $k \times d$  matrix, where each entry is sampled i.i.d from a Gaussian  $N(0, 1)$  (or from  $U(-1, 1)$ ). Then,

$$\Pr(|u \cdot v - f(u) \cdot f(v)| \geq \epsilon) \leq 4e^{-(\epsilon^2 - \epsilon^3)k/4}$$

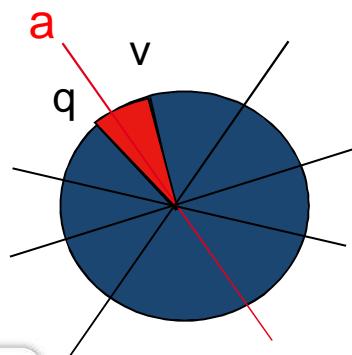
# Projections aléatoires

$$y = Ax$$

$y \in R^k$        $x \in R^d$

$A = \text{randn}(d, k)$   
= i.i.d sampled from Gaussian  $N(0, 1)$

Intuitivement, c'est le jeu de la roue. La probabilité que l'axe tiré aléatoirement sépare  $q$  et  $v$  est fonction de leur angle



$$\Pr[h_w(q) \neq h_w(v)]_w = \text{angle}(q, v) / \pi = \frac{1}{\pi} \cos^{-1}(q^T v)$$

# Comparaison

$$y = Ax$$

The diagram illustrates the decomposition of the vector  $y$  into two components:  $ACP$  (Auto-Contrainte Projetée) and  $\text{Projection aléatoire}$ . The vector  $y$  is shown as a horizontal line segment. An arrow labeled  $ACP$  points from the origin towards the left end of the segment. Another arrow labeled  $\text{Projection aléatoire}$  points from the origin towards the right end of the segment.

# Décorrélation = réduction de la dimension plus effective

# Premières composantes très bonnes

## Calcul plus coûteux de A et dépendant des données

C'est un apprentissage avec des problèmes possibles de généralisation à d'autres données

# **Universalité = aucune dépendance aux données**

**Scalabilité = Génération de A  
rapide même en très grande  
dimension**

# Pas de composante a priori plus utile que d'autres

# Composantes non décorrélées = réduction moins effective

# Combiner projections aléatoires et orthogonalisation

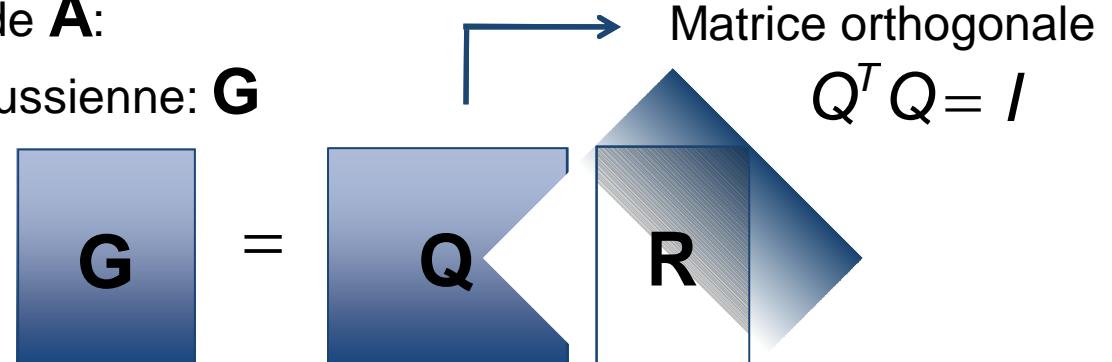
$$y = Ax$$

$y \in R^k$        $x \in R^d$

$A$  = matrice  $d \times k$  orthogonale générée aléatoirement

Procédure de construction de  $A$ :

1. Matrice  $k \times d$  aléatoire Gaussienne:  $\mathbf{G}$
2. Factorisation QR de  $G$ :



3. Conserver les  $k$  premières lignes de  $Q$

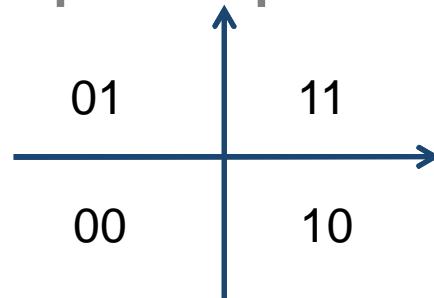
$$A = \begin{matrix} Q \\ \vdots \\ Q \end{matrix}$$

# **Les méthodes de partitionnement et de quantification**

# Quantification scalaire binaire

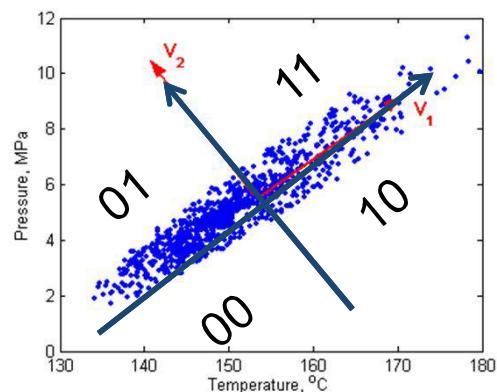
La plus simple = binarisation de chaque composante du vecteur

$$z_j = \text{sign}(x_j)$$



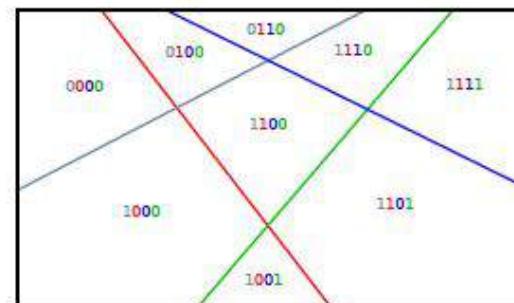
Utilisée en générale après centrage et normalisation des données ou changement d'espace

Ex1, après une ACP:



Ex2, après projections aléatoires

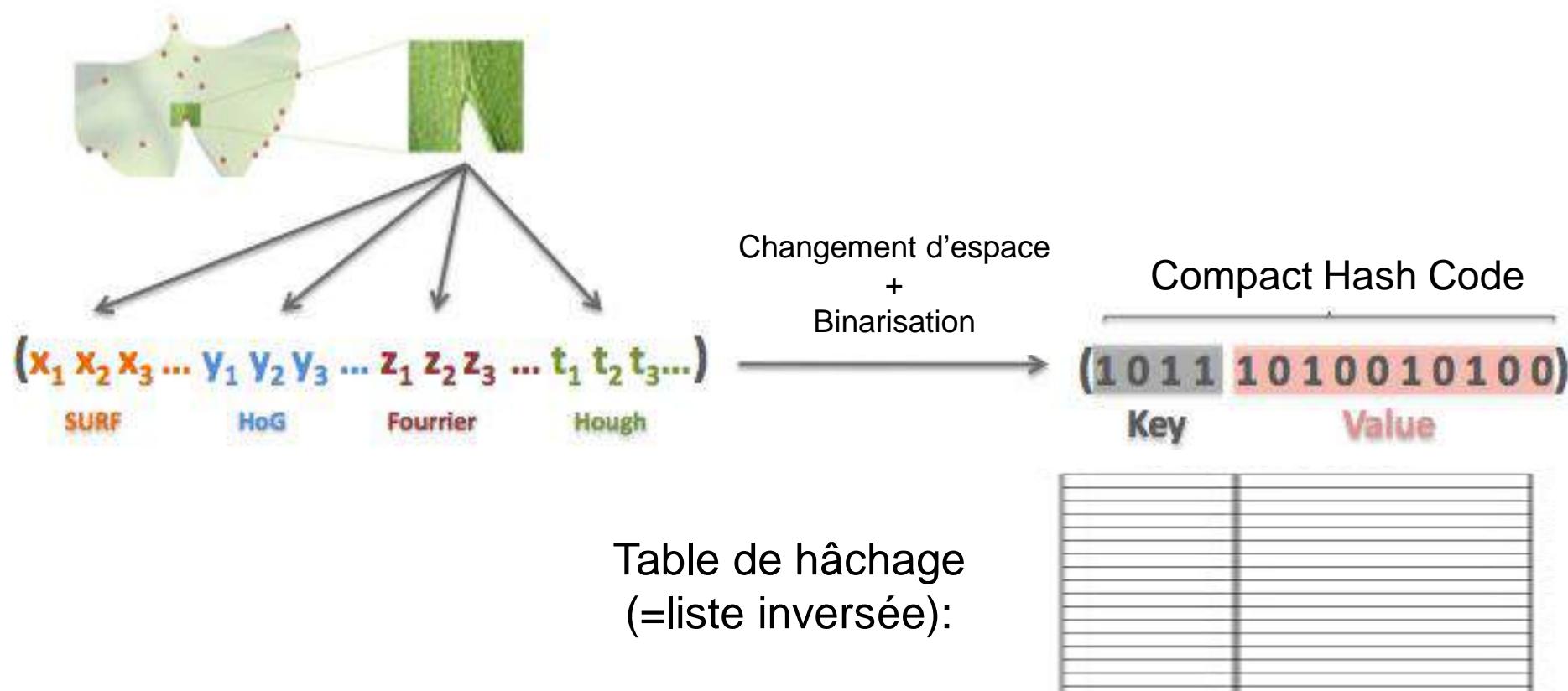
$$z = \text{sign}(Ax + b)$$



= LSH  
(Locality  
Sensitive  
Hashing)

# Quantification scalaire binaire

**Avantage: produit des hash codes binaires très compactes et très efficaces pour le stockage, l'adressage ou le calcul de distances**



# Quantification scalaire binaire

**Avantage:** produit des hash codes binaires très compactes et très efficaces pour le stockage, l'adressage ou le calcul de distances

**Hamming embedding:** on remplace les distances dans l'espace vectoriel d'origine par une distance de Hamming dans l'espace binarisé d'arrivée

$$d(q, v) \rightarrow d_H(z(q), z(v))$$

$$z(q) = 1 \text{ hash code} = [0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \dots 0 \ 0 \ 1 \ 0]$$

$$z(v) = 1 \text{ hash code} = [1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \dots 0 \ 1 \ 0 \ 0]$$



On verra que dans certains cas,  $d_H(z(q), z(v))$  converge vers  $d(q, v)$  lorsqu'on augmente le nombre de bits

La distance de Hamming peut-être de 10 à 100 fois plus rapide qu'une distance naïve dans l'espace d'origine

# Quantification scalaire binaire

Avantage: produit des hash codes binaires très compactes et très efficaces pour le stockage, l'adressage ou le calcul de distances

## 1. Nombreuses méthodes très efficaces de calcul de la distance de Hamming

- Utilisation de Look-Up Tables (LUT): typiquement par blocs de 8 ou 16 bits

$$d_H(\mathbf{z}(q), \mathbf{z}(v)) = \sum d_H(z_i^8(q), z_i^8(v))$$

Chaque bloc n'a que  $2^8$  valeurs possibles pour une requête q donnée. Ces valeurs peuvent être pré-calculées et stockées dans une table dont la clé est le hash code  $z_i^8(v)$

- Instructions assembleurs, e.g. SSE pop-count compte le nombre de bits à 1 de manière extrêmement efficace

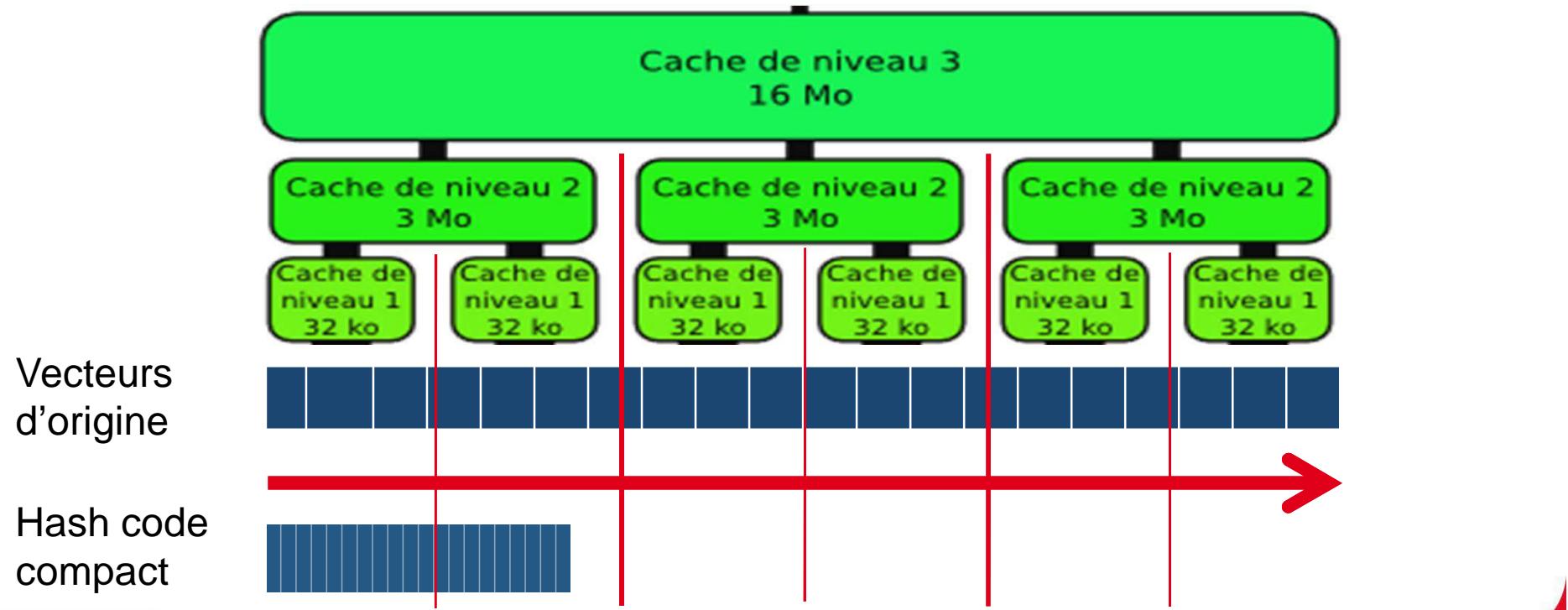
$$d_H(\mathbf{z}(q), \mathbf{z}(v)) = \text{POPCNT}(\text{NOR}(\mathbf{z}(q), \mathbf{z}(v)))$$



# Quantification scalaire binaire

Avantage: produit des hash codes binaires très compactes et très efficaces pour le stockage, l'adressage ou le calcul de distances

2. Données plus compactes = moins de sorties de cache



## La recherche exhaustive dans le cas binarisé

Il n'y a que **d** valeurs possibles pour la distance (d bits max diffèrent entre deux hash codes). La distance peut donc servir de clé dans une multiMap:

```
MultiMap match_map = new MultiHashMap();
for (i=1; i<n ; i++) match_map.put(ComputeDistance(q,X[i]), i); //rempli la MultiMap O(n)

dist=0; nb_nn=0;
While (nb_nn<k) {
    nb_nn+=match_map .get(m)); // accède aux éléments par distance croissante
    dist++;
}
```

Complexité linéaire **O(n+d)≈O(n)** meilleure que  
**O(n log n), O(n.k), O(n log k)**



# Grilles et lattices

**Avantage:** Les mêmes que quantification scalaire binaire mais avec une erreur de quantification qui peut être plus faible

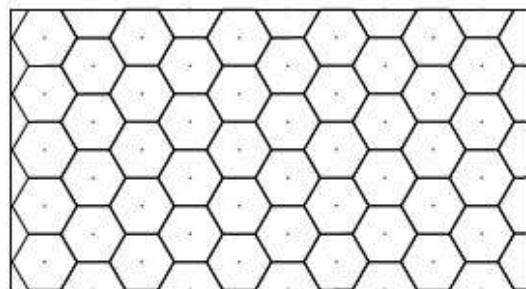
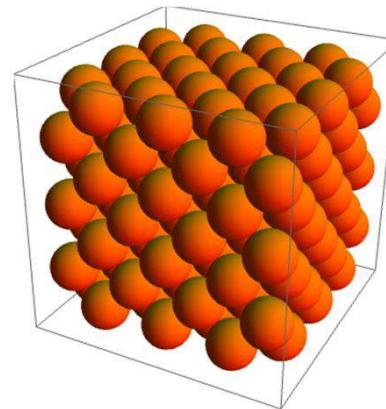
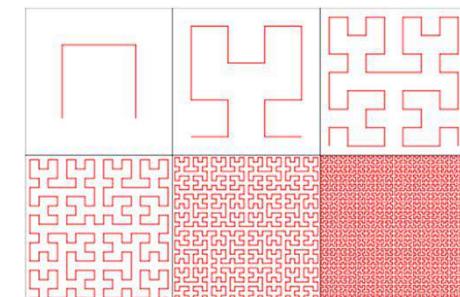


FIGURE 2.2 – Exemple d'un treillis hexagonal



Leech lattices

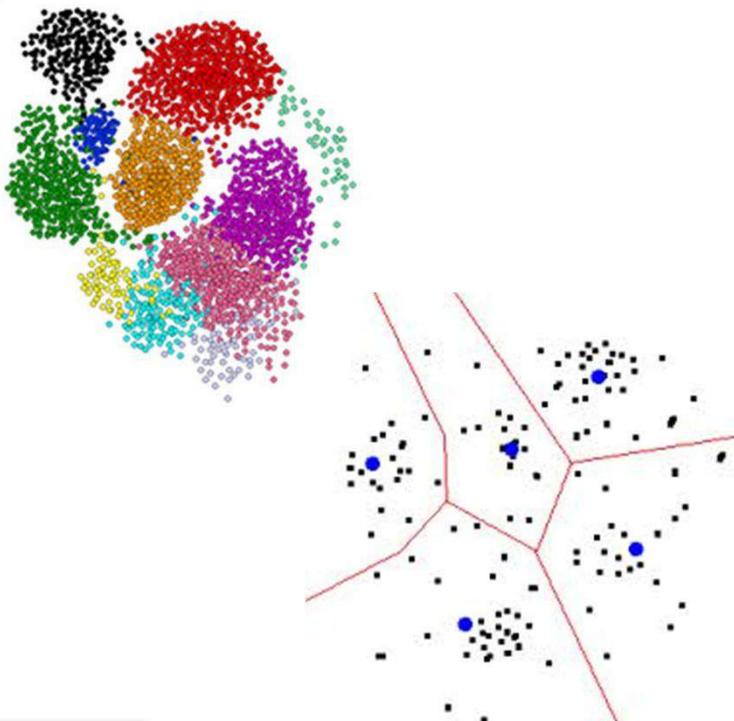


Space-filling curves

**Inconvénients:** Temps d'encodage peut être beaucoup plus long & la gain sur l'erreur de quantification dépend des données et peut être faible au final

# Les méthodes de partitionnement par regroupement de données (clustering)

Contrairement aux méthodes de quantification scalaires ou vectorielles vues précédemment, le partitionnement dépend des données



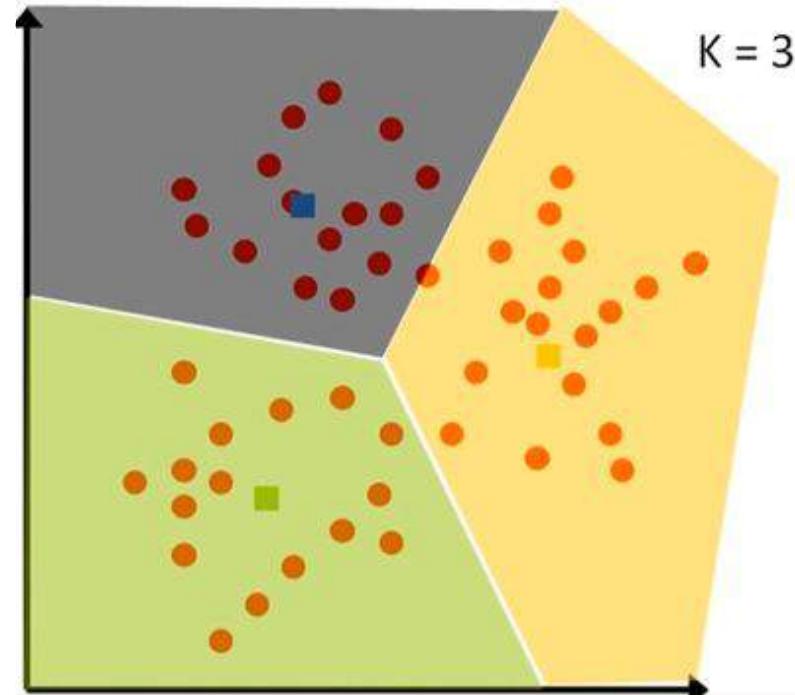
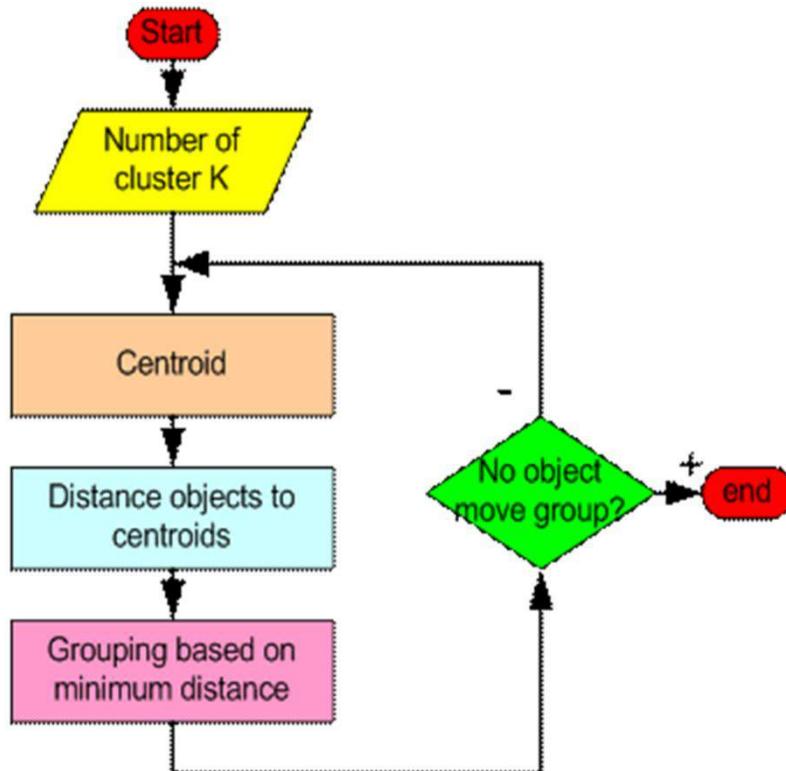
Des centaines de méthodes de clustering dans la littérature:

- Avec ou sans recouvrement
- Nombre de cluster fixes ou non
- Espace euclidien ou non
- Approches hiérarchiques ou non
- Approches probabilistes ou non
- Semi-supervisées (pair-wise constraints)
- Complexité

Un des problèmes les plus transversaux de tout l'informatique

# Les méthodes de partitionnement par regroupement de données (clustering)

Algo le plus utilisée en recherche d'information par le contenu = k-means



# Les méthodes de partitionnement par regroupement de données (clustering)

Algo le plus utilisée en recherche d'information par le contenu = k-means

## Intérêts:

- Adaptibilité aux données
- Minimisation de l'erreur de quantification L2
- Complexité d'apprentissage raisonnable  $O(k.N)$
- Contrôle de la taille de la partition

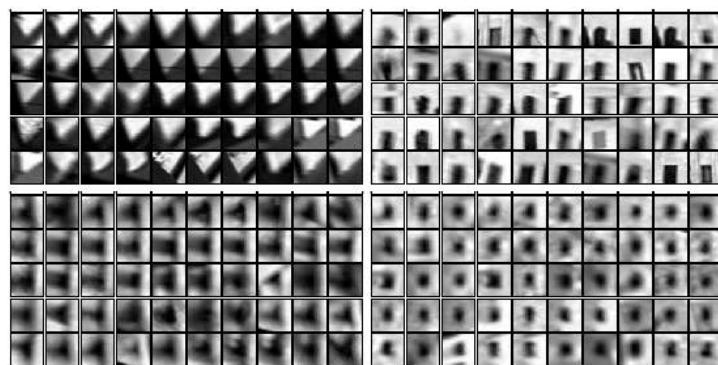
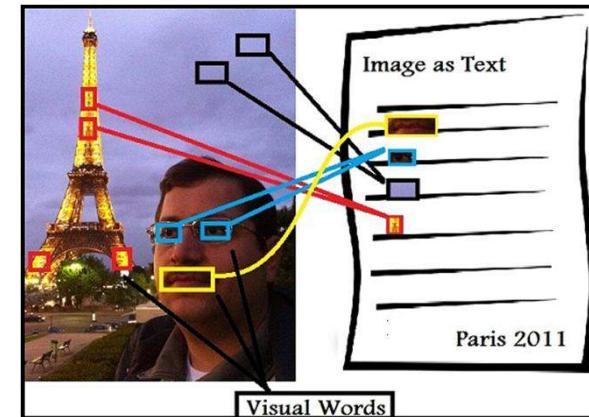
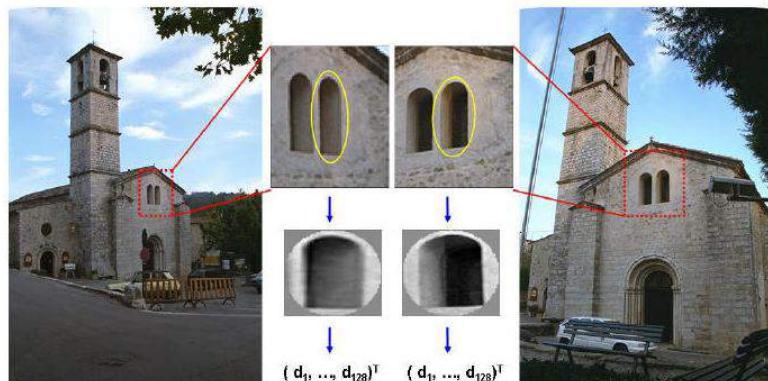
## Inconvénients:

- Temps d'encodage  $O(k)$  devient problématique lorsque  $k$  est grand
- Utile pour partitionner les données mais pas pour compresser finement (pas assez d'information !!)

# Les méthodes de partitionnement par regroupement de données (clustering)

K-means énormément utilisé pour partitionner les espaces de descripteurs visuels locaux

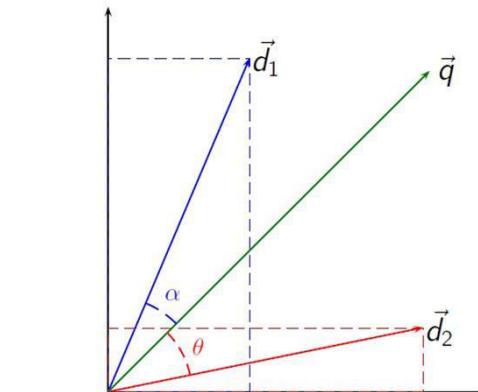
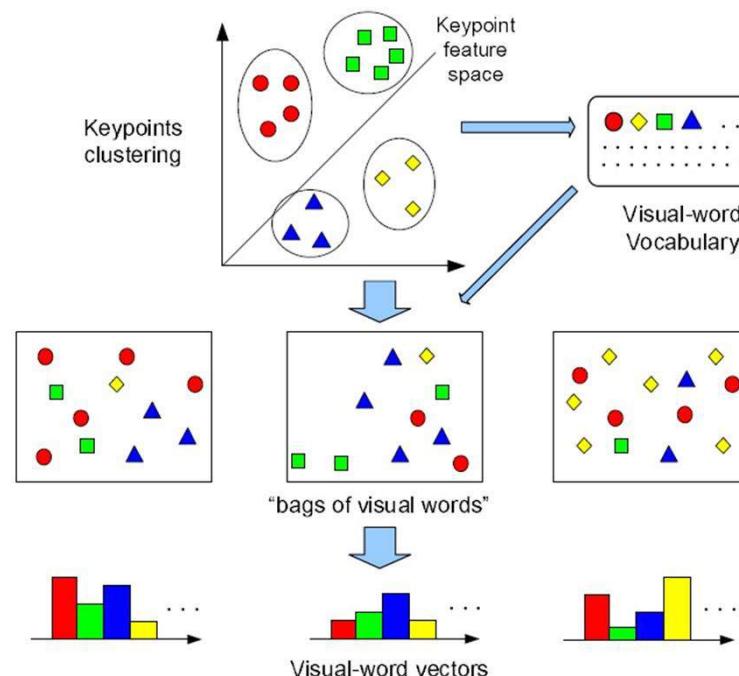
On parle de **mots visuels** pour qualifier les partitions produites



# Les méthodes de partitionnement par regroupement de données (clustering)

K-means énormément utilisé pour partitionner les espaces de descripteurs visuels locaux

Une image = un **sac de mots visuels** par analogie au texte

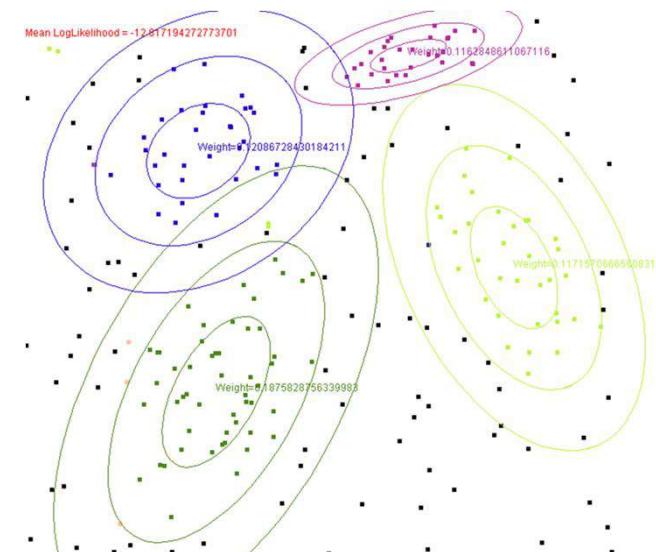
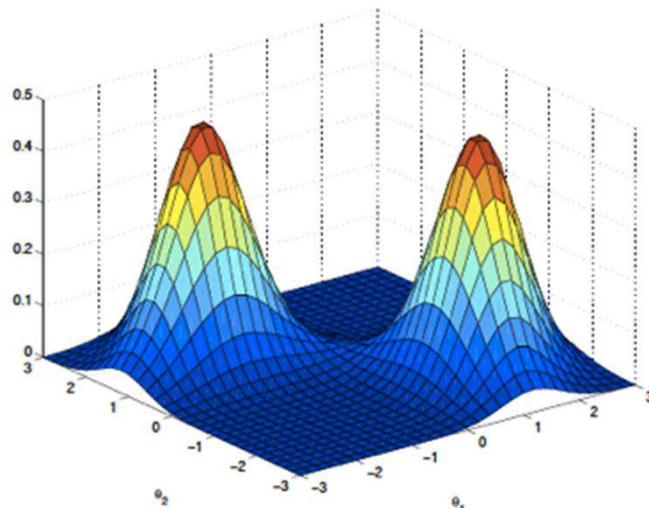


$$d_j = (w_{1,j}, w_{2,j}, \dots, w_{t,j})$$
$$q = (w_{1,q}, w_{2,q}, \dots, w_{t,q})$$

# Les méthodes de partitionnement par regroupement de données (clustering)

## Expectation Maximisation et mixtures de gaussiennes beaucoup utilisées en audio

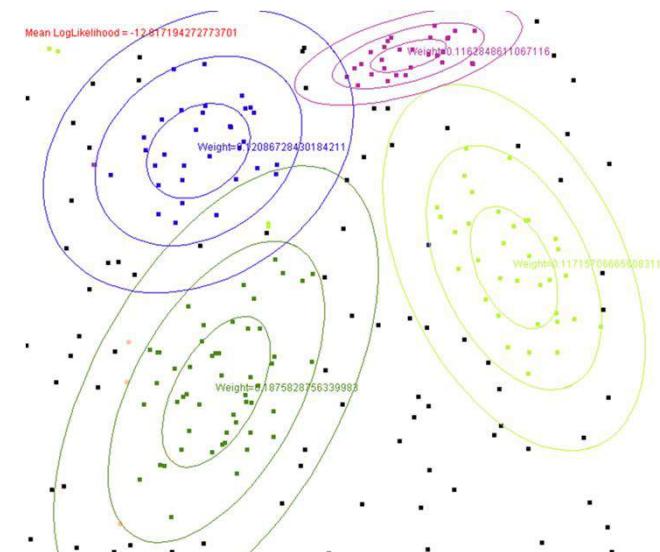
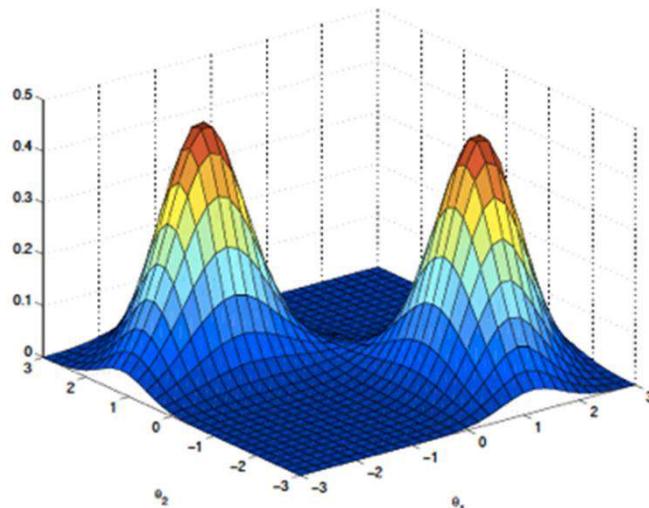
Même principe itératif que k-means mais on estime les paramètres d'une gaussienne multi-dimensionnelles au lieu de uniquement les coordonnées du centroïd



# Les méthodes de partitionnement par regroupement de données (clustering)

Expectation Maximisation et mixtures de gaussiennes beaucoup utilisées en audio

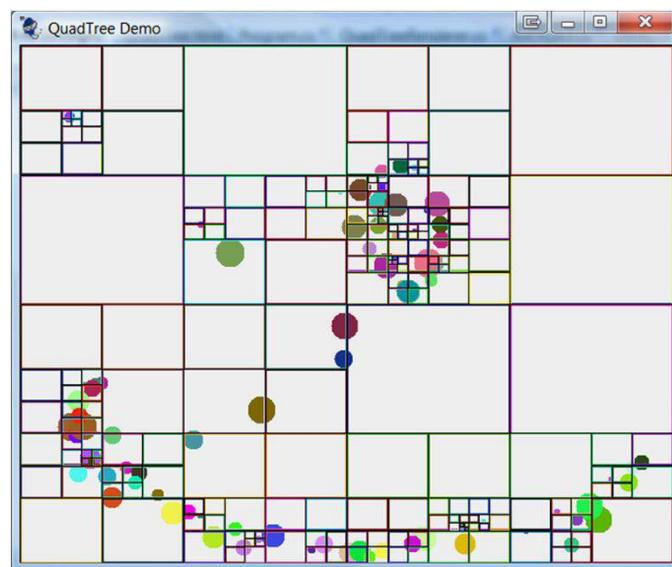
Modèle beaucoup plus fin des données (idéale pour reconnaissance de locuteurs, speech-to-text, etc.), mais plus couteux pour indexation, compression, etc.



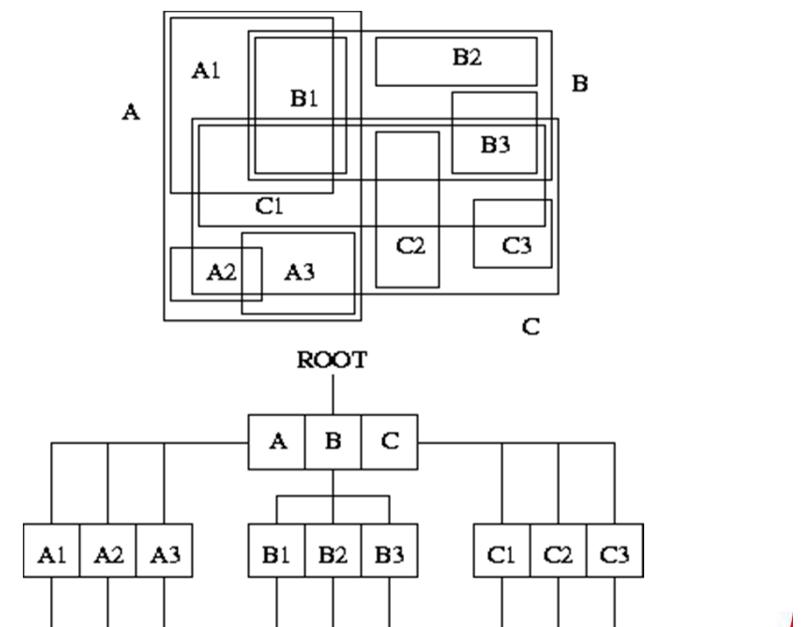
# Les méthodes de partitionnement hiérarchiques

De nombreuses méthodes à base d'arbres existent mais la plupart ne sont pas utilisés pour la recherche par le contenu à cause de la malédiction de la dimension

Quad-tree:  $2^d$  new buckets at each split !!



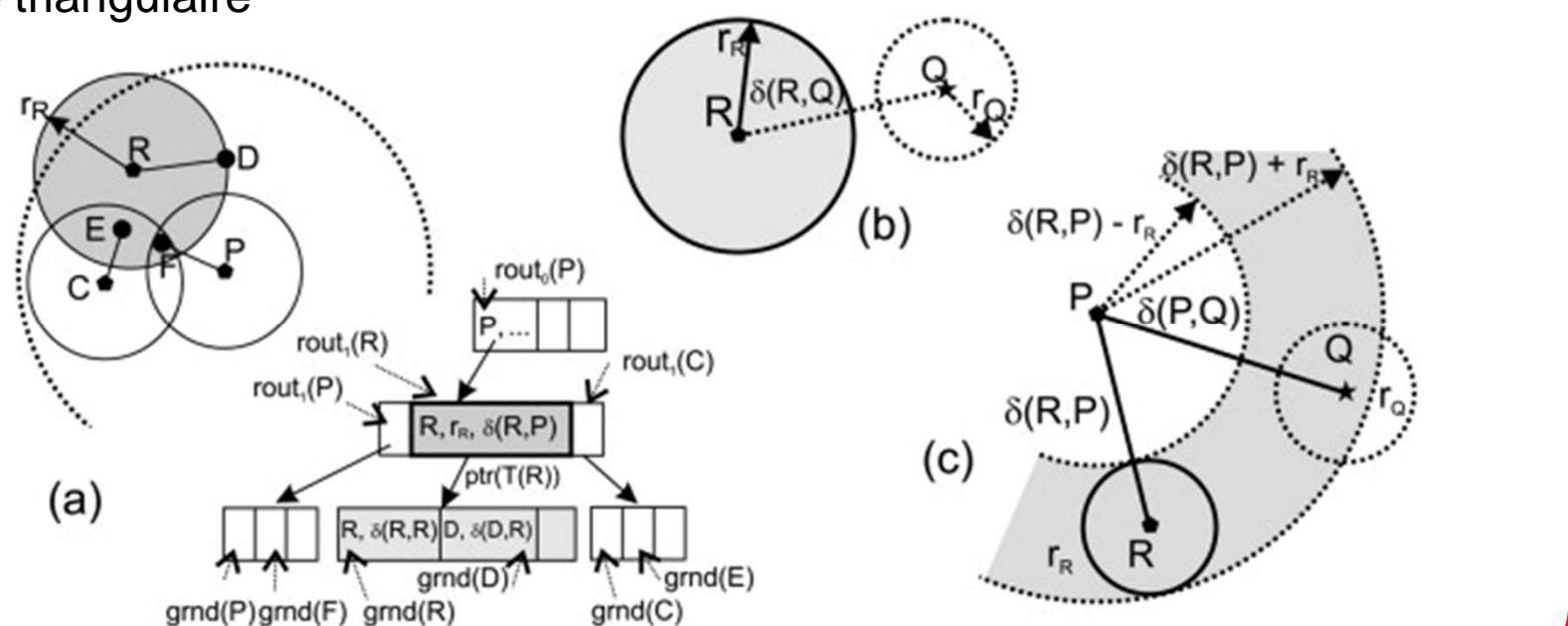
R-tree: Bounding box rectangulaires tendent à couvrir tout l'espace lorsque  $d$  est très grand



# Les méthodes de partitionnement hiérarchiques

Le M-tree (Metric Tree) est encore utilisé dans certaines applications de recherche d'information par le contenu

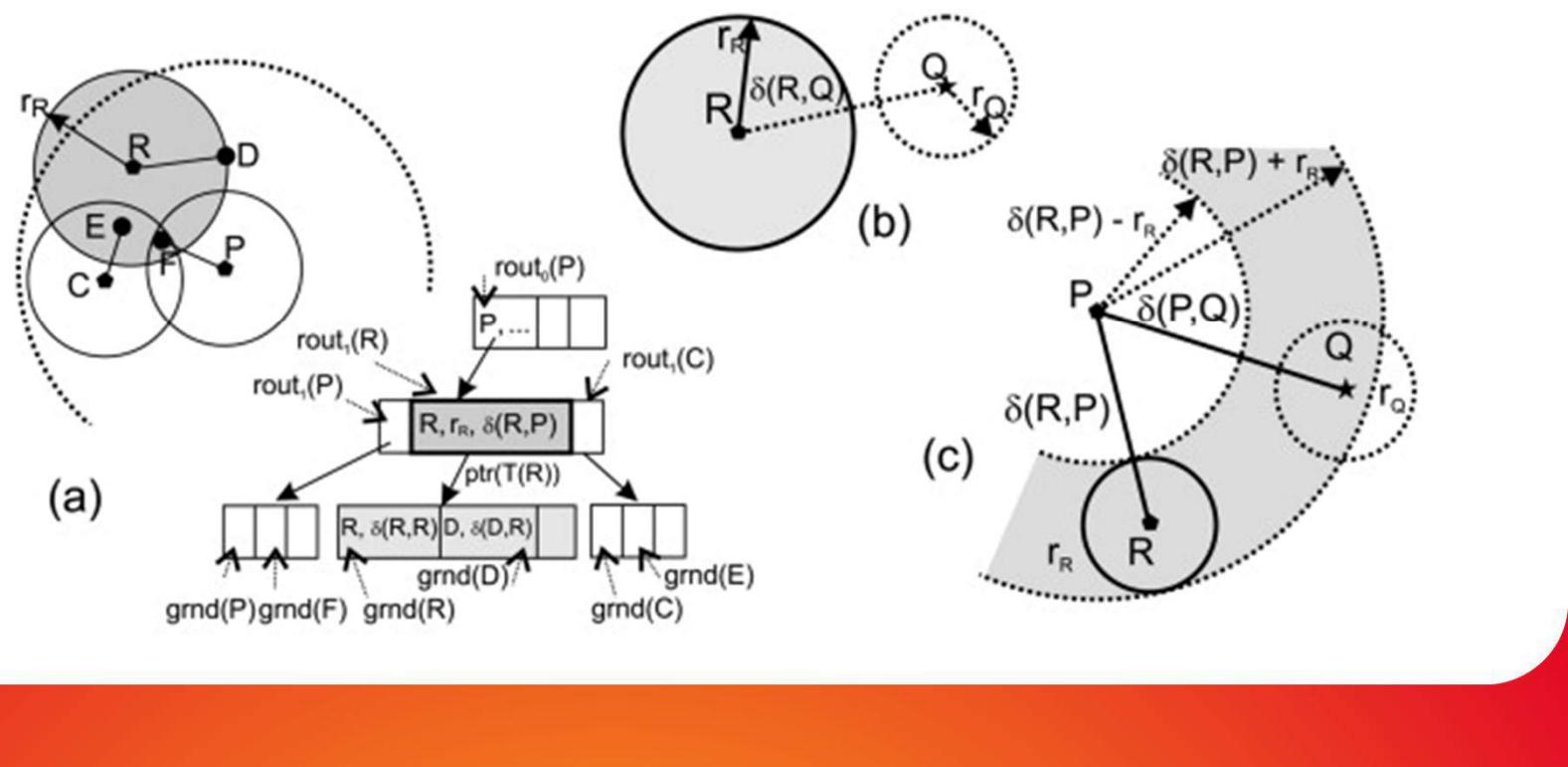
Intérêt: Le M-tree utilise uniquement des points pivots et des distances à ces points pour partitionner les données. On peut indiquer n'importe quels objets (même non vectoriel) associés à une métrique respectant l'inégalité triangulaire



# Les méthodes de partitionnement hiérarchiques

Le M-tree (Metric Tree) est encore utilisé dans certaines applications de recherche d'information par le contenu

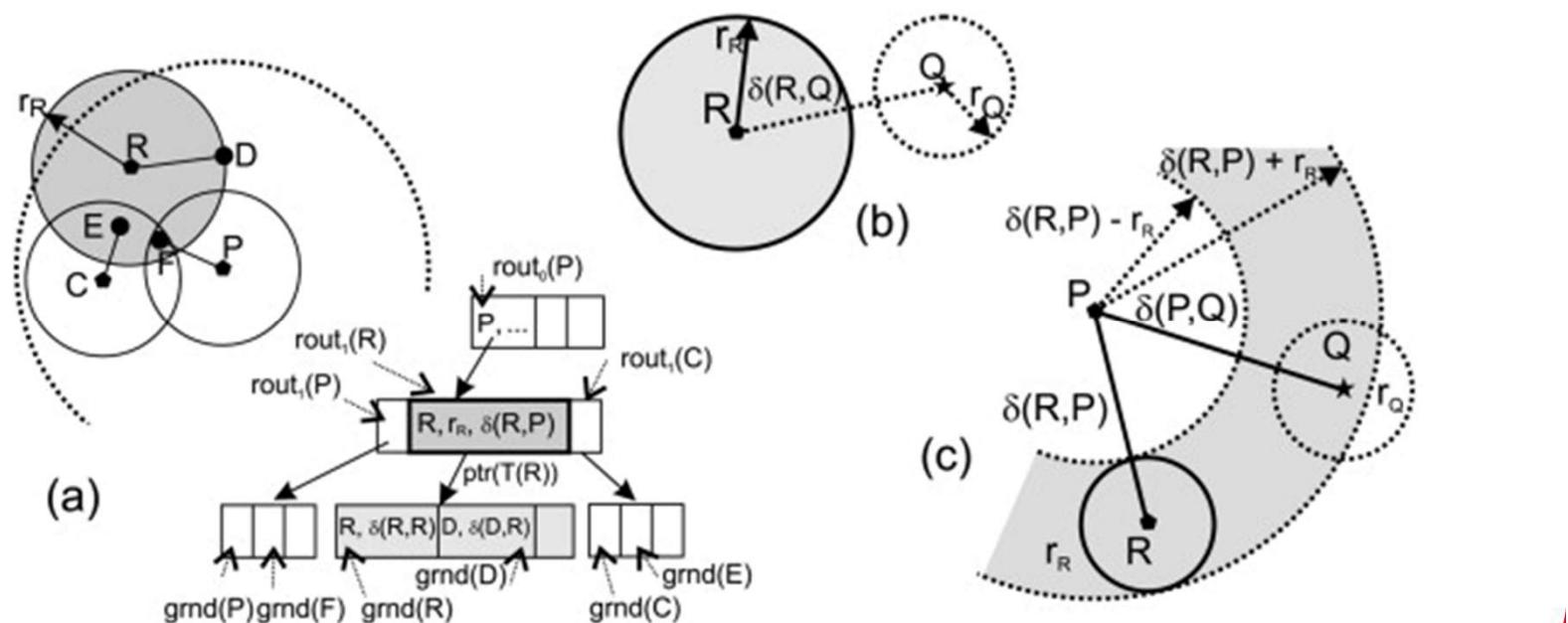
Intérêt: insertions ou suppressions dynamiques de complexité logarithmiques



# Les méthodes de partitionnement hiérarchiques

Le M-tree (Metric Tree) est encore utilisé dans certaines applications de recherche d'information par le contenu

Intérêt: Le M-tree souffre tout de même de la malédiction de la dimension et n'est rentable qu'avec des algorithmes de recherche approximatifs de type  $(1+\varepsilon)$ -approximatives (cf. chapitre algorithmes de recherche)

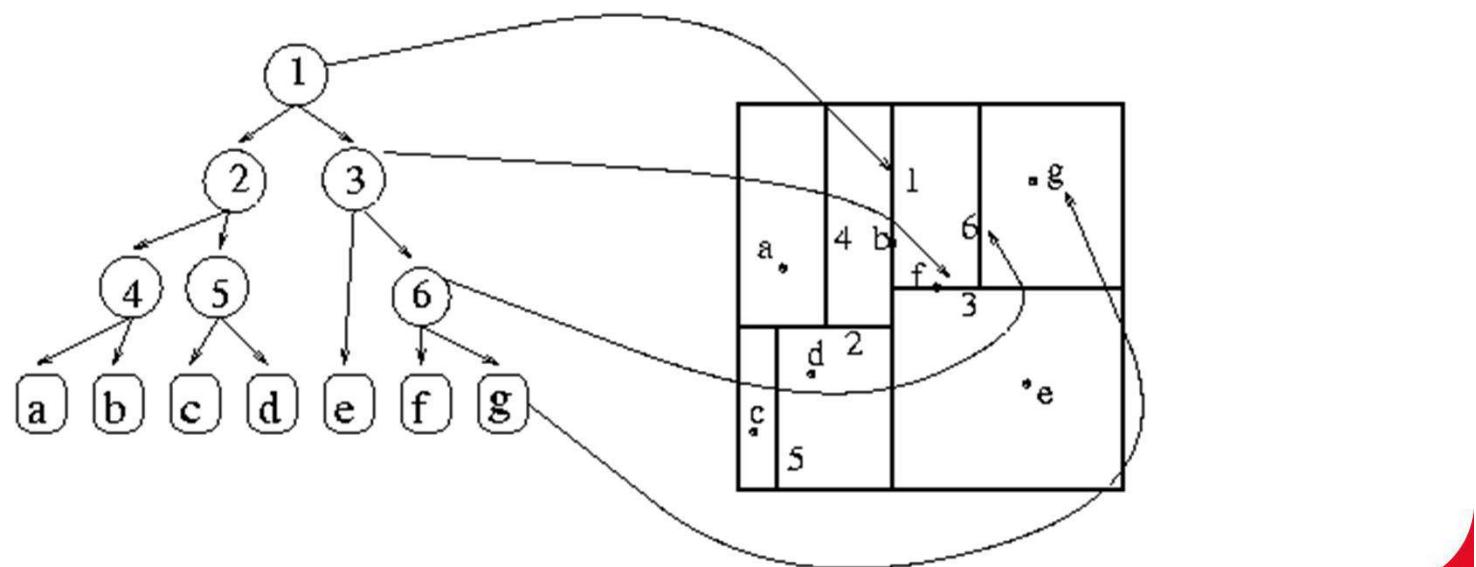


# Les méthodes de partitionnement hiérarchique

Le kd-tree est utilisé pour la recherche d'image par le contenu (e.g. dans la librairie FLANN)

## kd-tree classique:

A chaque niveau  $i$ , partitionne l'ensemble courant en deux parties égales par un hyperplan orthogonal à l'axe  $i \bmod d$

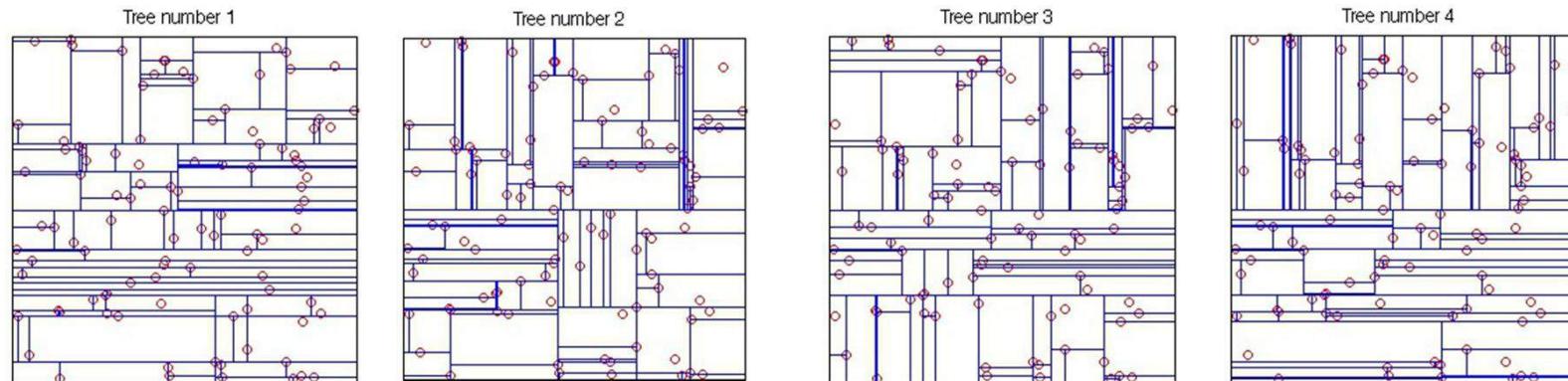


# Les méthodes de partitionnement hiérarchique

Le kd-tree est utilisé pour la recherche d'image par le contenu (e.g. dans la librairie FLANN)

## Randomized kd-trees:

En très grande dimension, on utilise plutôt une forêt d'arbres dans lesquels le choix de la dimension se fait aléatoirement parmi les  $d \leq D$  dimensions ayant la plus grande variance



# Les méthodes de partitionnement hiérarchique

## Hierarchical K-means

On applique un K-Means avec un très faible nombre de cluster ( $K = 10$  par exemple), puis on recommence un nouveau K-Means sur chaque sous-ensemble de descripteurs créés par les  $K$  clusters.

L'algorithme itère ensuite jusqu'à obtention d'un nombre de clusters total suffisant. Ce nombre étant égal à  $K^n$ , avec  $n$  le nombre d'itérations.

HKM permet de réduire la complexité de l'algorithme à  $O(N \log K)$  au lieu de  $O(NK)$ .

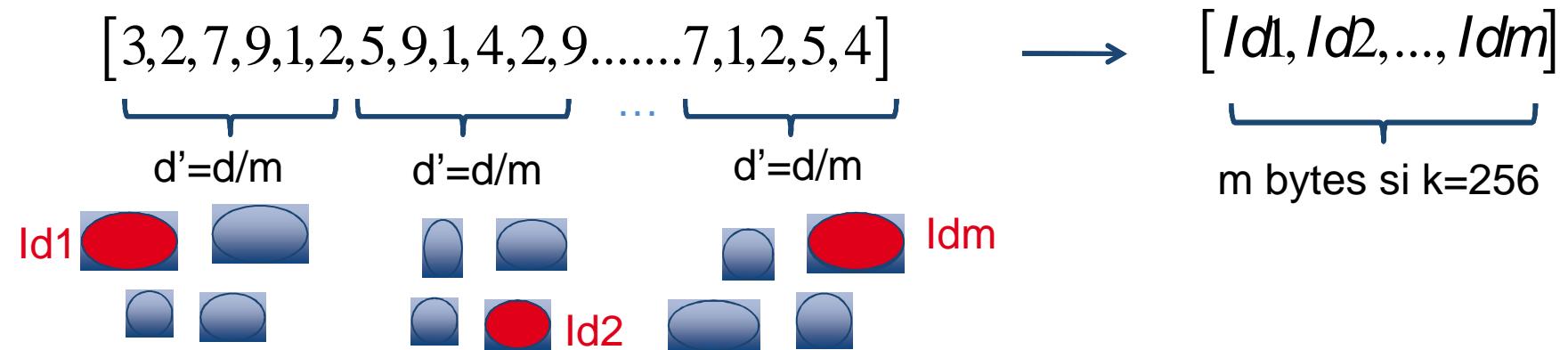
Bien que beaucoup plus efficace en termes de temps de calcul, HKM ne produit pas d'aussi bons résultats de clustering qu'un K-Means standard.



# Les méthodes de partitionnement dans des sous espaces

On découpe l'espace d'entrée en plusieurs sous espaces de dimension moindre puis on partitionne chaque sous espace

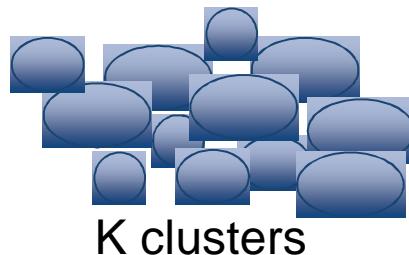
Exemple: Kmeans Product Quantizer



# Les méthodes de partitionnement dans des sous espaces

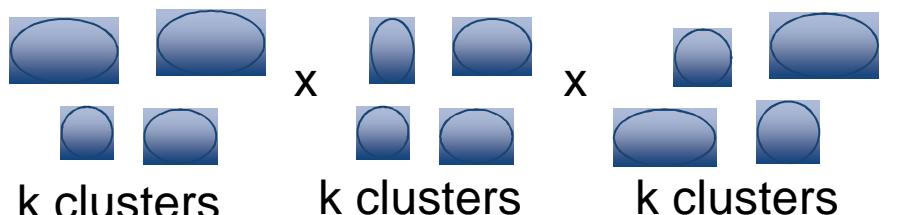
Exemple: Kmeans Product Quantizer

→ avantage vs. kmeans classique = on peut créer des partitions (vocabulaires) de très grande taille sans problème de temps d'encodage ou d'espace mémoire



=  **$K$  régions**

Construction  $O(Knd)$   
Encodage  $O(Kd)$   
Mémoire  $O(Kd)$



=  **$k^m$  régions**

Construction  $O(knd)$   
Encodage  $O(kd)$   
Mémoire  $O(kd)$   
 $k \ll K$

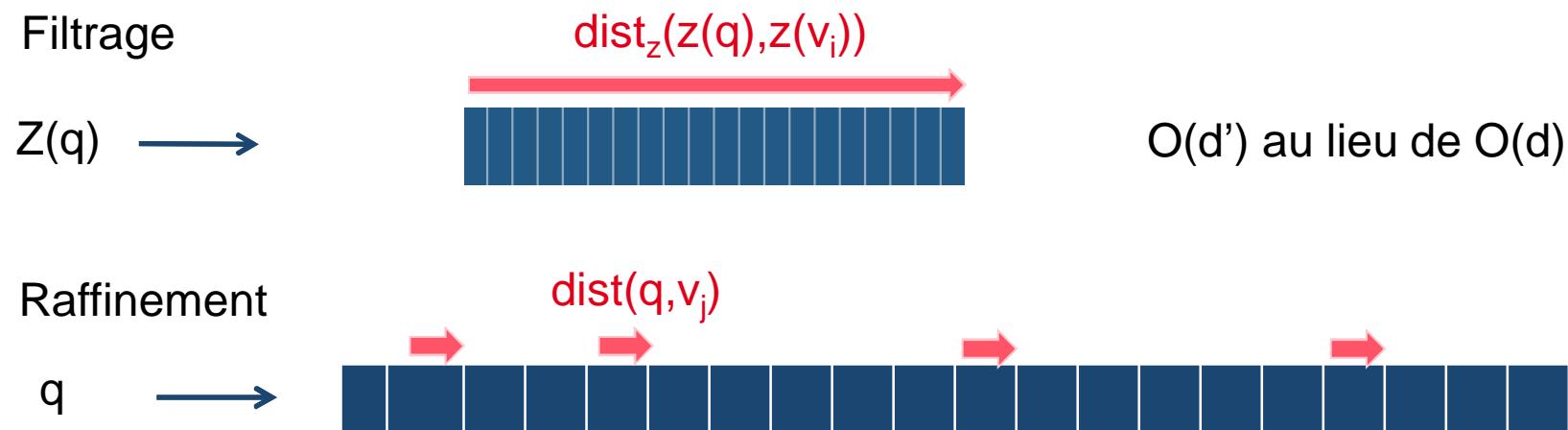
e.g.  $k:256, m:8, d:128 \Rightarrow 18.10^{18}$  régions

# Les structures d'index et algorithmes de recherche associées



# Tableaux de vecteurs quantifiés et recherche exhaustive à deux étages

On cherche un ensemble de voisins dans l'espace quantifié puis on raffine dans l'espace d'origine

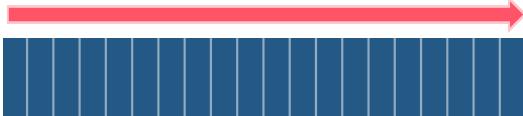


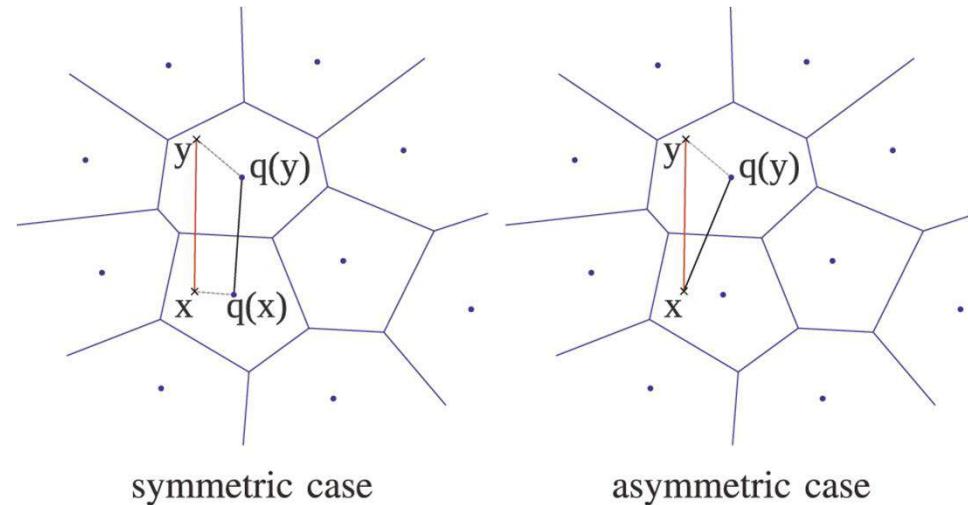
Le filtrage dans l'espace quantifié peut se faire de plusieurs façons:

- Les  $k'$  plus proches voisins avec  $k' \gg k$  voulu dans l'espace d'origine
- A un rayon près
  - méthode exact parfois possible (borne inf sur la bucket)
  - méthode probabiliste parfois possible (Ish: projections aléatoires)

# Les distances asymétriques

On peut ne pas quantifier la requête et utiliser une distance asymétrique avec les vecteurs quantifiés de l'index

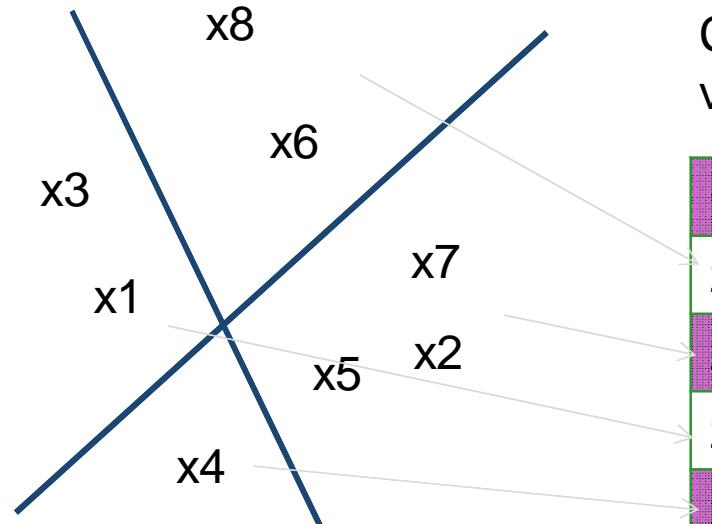
$$\text{dist}_{\text{asym}}(q, z(v_i))$$




Avantage: meilleure approximation de la distance dans l'espace d'origine sans augmentation de l'espace mémoire de l'index

Inconvénient: en général un peu plus lent que distance de Hamming

# Les tables et listes inversées



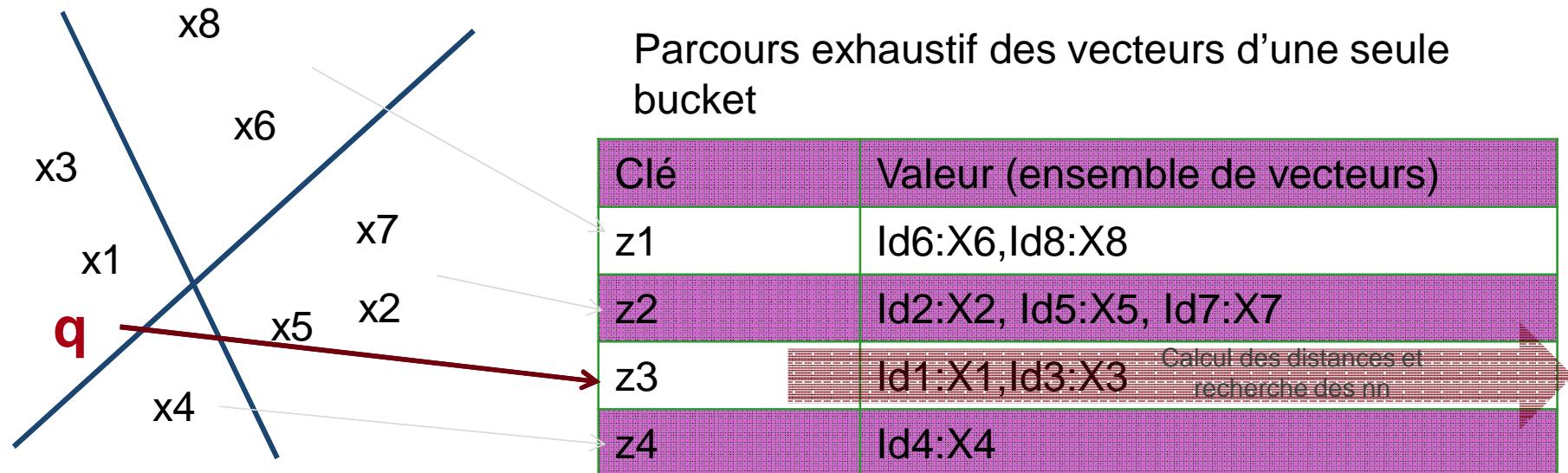
Clé=identifiant d'une région de la partition= 1 valeur dans l'espace quantifiée

Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6, Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1, Id3:X3
z4	Id4:X4

Zj peut être:

- Un identifiant de cluster (par exemple mot visuel)
- Un hash code binaire (stocké sous forme de string)
- Une position dans une grille
- Un identifiant de feuille dans un arbre
- ...

# Les tables et listes inversées: accès simple

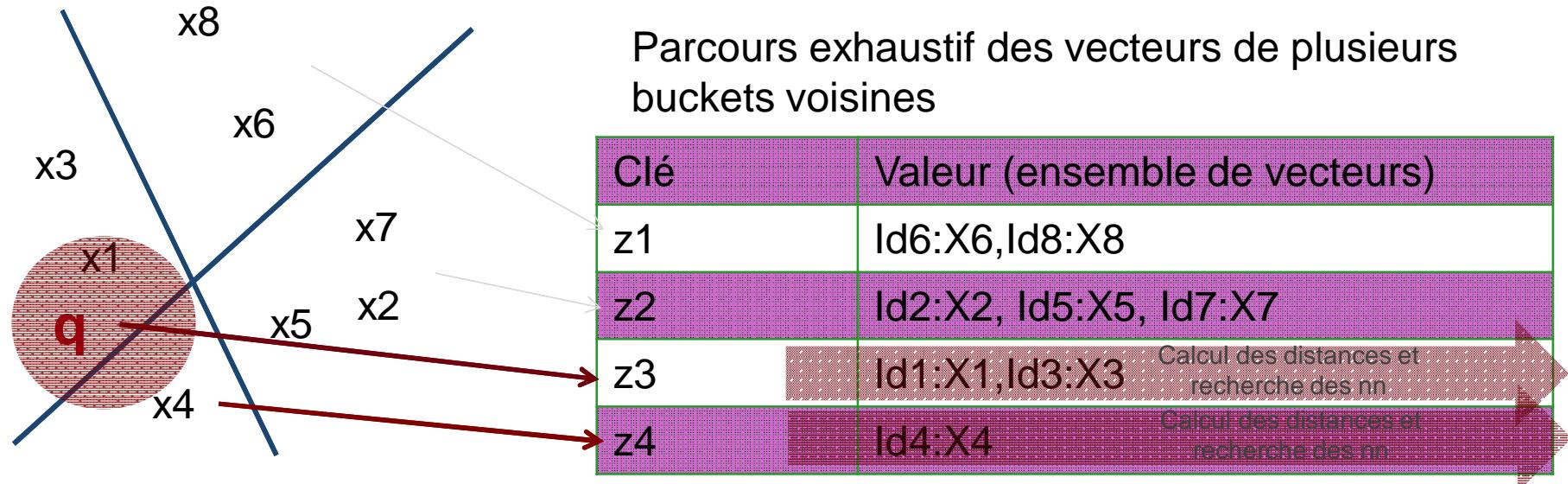


Avec ou sans calcul des distances pour affiner la recherche (knn ou range)

Très efficace, mais qualité faible en grande dimension quelle que soit la méthode de partitionnement:

- malédiction de la dimension
- beaucoup de plus proches voisins n'appartiennent pas à  $z(q)$
- Taille de la partition = compromis entre qualité et efficacité

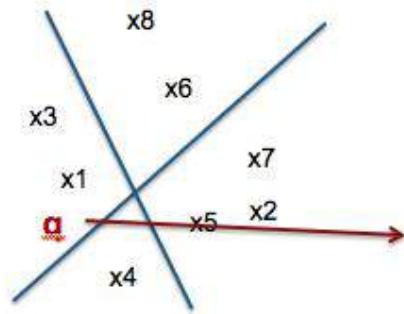
# Les tables et listes inversées: accès multiples



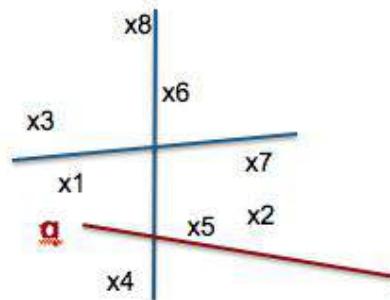
Amélioration du rappel en visitant les buckets voisins. La détermination des buckets voisins dépend du type de partition et peut se faire de plusieurs façons:

- Les  $k$  plus proches (typiquement pour kmeans, compromis qualité / temps de calcul ajustable en ligne)
- à un rayon près (pas toujours possible, danger: nombres de bucket peut être énorme)
- probabiliste (par apprentissage ou théorique dans le cas de projections aléatoires → étude détaillée de LSH)

# Les tables et listes inversées: tables multiples



Clé	Valeur (ensemble de vecteurs)
z1	Id6:X6, Id8:X8
z2	Id2:X2, Id5:X5, Id7:X7
z3	Id1:X1, Id3:X3
z4	Id4:X4



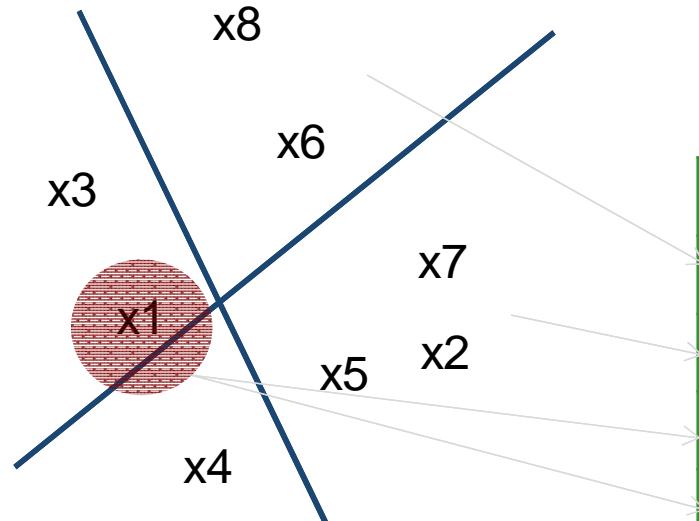
Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3, Id8:X8
z2	Id6:X6
z3	Id2:X2, Id5:X5, Id7:X7
z4	Id1:X1, Id4:X4

La manière de construire plusieurs tables dépend du type de partition:

- Plusieurs kmeans sur des données d'apprentissage différentes
- Plusieurs ensembles de projections aléatoires (cf. LSH)
- Analyse en composantes principales sur des sous-espaces

Peut être plus efficace que les accès multiples (moins d'accès au final) mais nécessite plus de mémoire pour stocker toutes les tables

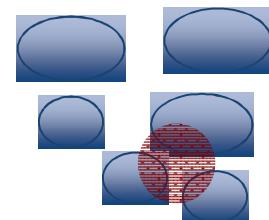
# Les tables et listes inversées: indexation dense



Indexation dans plusieurs buckets voisins,  
accès unique pendant le search

Clé	Valeur (ensemble de vecteurs)
z1	Id3:X3, Id6:X6, Id8:X8
z2	Id2:X2, Id5:X5, Id6:X6, Id7:X7
z3	Id1:X1, Id3:X3, Id4:X4
z4	Id1:X1, Id4:X4

- On troque du temps de calcul contre de la mémoire
- La mémoire requise peut devenir très grande pour certaines partitions
- La détermination des buckets voisins peut se faire de plusieurs manières (k plus proches, rayon, probabiliste)
- Particulièrement utilisée avec partition kmeans (

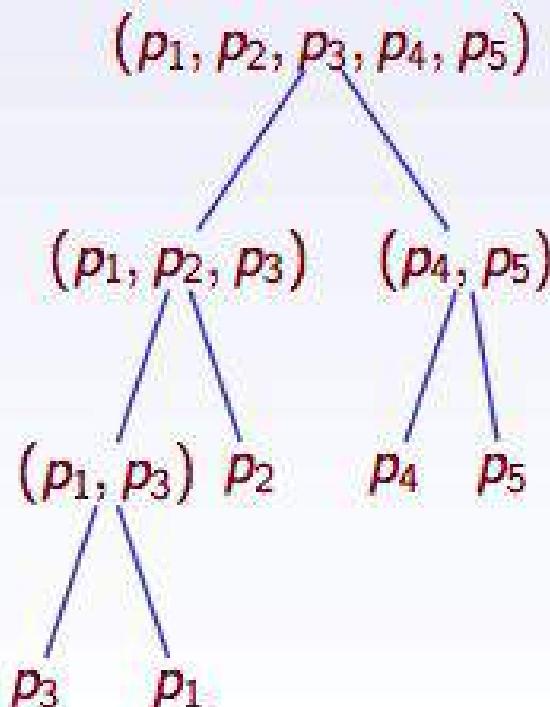


# Les arbres (pour méthodes de partitionnement hiérarchiques)

Base  $S = \{p_1, \dots, p_n\}$

représentée par un arbre:

- Chaque noeud est un sous-ensemble de  $S$
- La racine est  $S$  entier
- L'ensemble père est entièrement couvert par les ensembles fils
- Chaque noeud contient une "description" de son sous-arbre fournissant une **borne** inférieure pour  $d(q, \cdot)$  dans le sous-ensemble correspondant

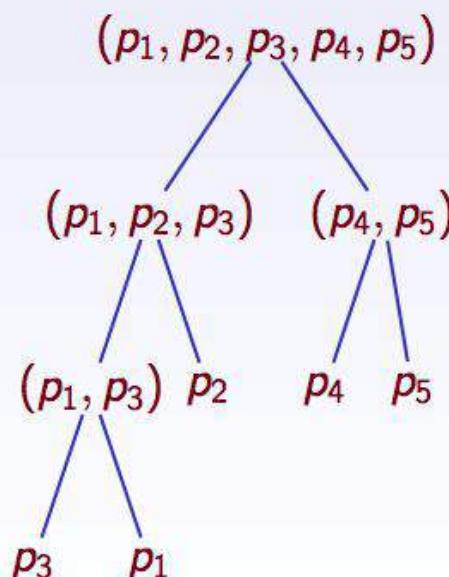


# Les arbres (pour méthodes de partitionnement hiérarchiques)

## Branch and Bound: Recherche à un rayon près

**Objectif:** Trouver tous les  $p_i$  tels que  $d(p_i, q) \leq r$ :

- ① Parcours en profondeur de l'arbre (algorithme récursif)
- ② A chaque noeud, calcul de la borne inférieure du sous-ensemble correspondant
- ③ Elimination des branches avec une borne inférieure supérieure à  $r$



# Les arbres (pour méthodes de partitionnement hiérarchiques)

## B&B: Recherche du plus proche voisin

**Objectif:** trouver  $\operatorname{argmin}_{p_i} d(p_i, q)$ :

- ① Choisir aléatoirement un  $p_i$ , initialiser  
 $p_{NN} := p_i, r_{NN} := d(p_i, q)$
- ② Commencer une recherche à  $r_{NN}$  près
- ③ Lorsque l'on rencontre un  $p'$  tel que  $d(p', q) < r_{NN}$ ,  
mise à jour  $p_{NN} := p', r_{NN} := d(p', q)$

# Les arbres (pour méthodes de partitionnement hiérarchiques)

## B&B: Best Bin First

**Objectif:** trouver  $\operatorname{argmin}_{p_i} d(p_i, q)$ :

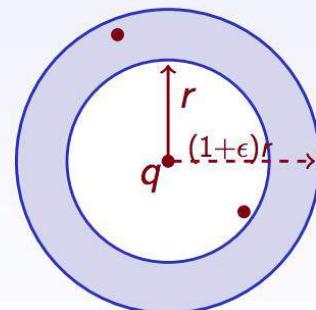
- ① Choisir aléatoirement un  $p_i$ , initialiser  $p_{NN} := p_i, r_{NN} := d(p_i, q)$
- ② Mettre le noeud racine dans une **queue d'inspection**
- ③ Répéter: choisir le noeud de la queue d'inspection ayant la borne inférieure la plus petite, calculer les bornes inf. des sous-ensembles fils
- ④ Insérer dans le queue les fils dont la borne inf. est inférieure à  $r_{NN}$ ; éliminer les branches des autres fils
- ⑤ Lorsque l'on rencontre un  $p'$  tel que  $d(p', q) < r_{NN}$ , mise à jour  $p_{NN} := p', r_{NN} := d(p', q)$

# Les arbres (pour méthodes de partitionnement hiérarchiques)

En dimension >15, seuls des algorithmes de recherche approximatifs permettent d'être plus rapide qu'une recherche exhaustive (malédiction de la dimension)

- early stopping
- requête à  $(1+\epsilon)$

**Requêtes à un rayon  $r$  près  $(1 + \epsilon)$ -approximatives:**  
s'il y a au moins un  $p \in S$  :  $d(q, p) \leq r$  retourne des  $p'$  :  $d(q, p') \leq (1 + \epsilon)r$



C'est une borne max sur l'erreur relative entre le(s) point(s) retrouvé(s) et le(s) knn exact

$$\frac{d(q, p') - d(q, p)}{d(q, p)} \leq \epsilon$$

# Les arbres (pour méthodes de partitionnement hiérarchiques)

## Quand utiliser des arbres ?

- Dimensionalité modérée (<128)
- Taille de base modérée (<1M de vecteur)
- Lorsque les données arrivent de manière incrémentale
- Lorsqu'il y a des besoins d'insertion/suppression dynamique

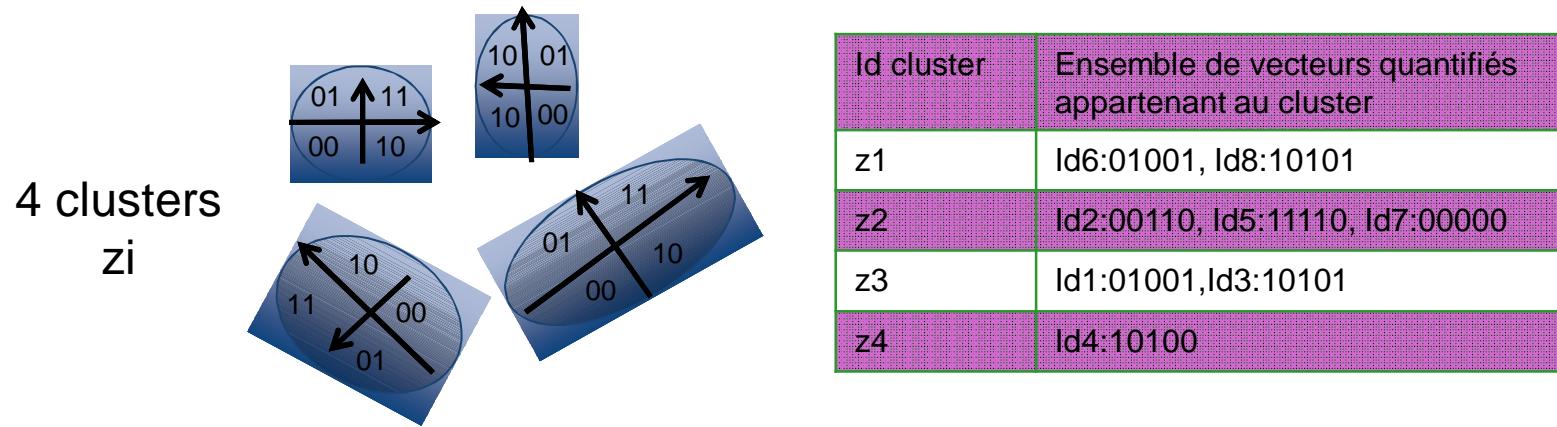
Les arbres ne sont en revanche pas adaptés aux très grandes tailles de base notamment lorsque l'accélération du temps de recherche reste le facteur primordial

- Surcoût mémoire pour stocker la structure
- Parcours d'arbre plus coûteux que des accès dans une table (sauts mémoires, sorties de cache, des « if », etc.)

# Les approches combinées à deux étages (coarse-to-fine)

On peut utiliser une quantification grossière sous forme de table ou d'arbre et des approches de quantifications vectorielles plus fines à l'intérieur des buckets

Exemple: k-means + ACP locale + Hamming Embedding



- La quantification binaire s'adapte à chaque cluster et est au final plus efficace qu'une ACP globale
- Au moment de la requête, on détermine d'abord le cluster puis on quantifie avec la bonne matrice de projection

# Les approches combinées

On peut imaginer beaucoup de combinaisons... nombreux travaux de recherche porte sur le sujet

- Randomized KD-tree pour filtrer + quantification fine avec projection aléatoire et ACP
- Kmeans normal pour construction table + indexation fine avec product quantizer
- ACP binarisée pour construction table + projections aléatoires pour quantification fine des vecteurs
- Possibilité de faire plus de 2 étages de quantification (coarse-to-fine)
- Construction de graphes de similarité approximatifs sur vecteurs quantifiés puis raffinement lors de la marche, etc.





# Recherche d'images par le contenu II

**Master1 – Traitement et analyse d'images- GMIN215**

Alexis JOLY - [alexis.joly@inria.fr](mailto:alexis.joly@inria.fr)

November 1<sup>st</sup> 2012

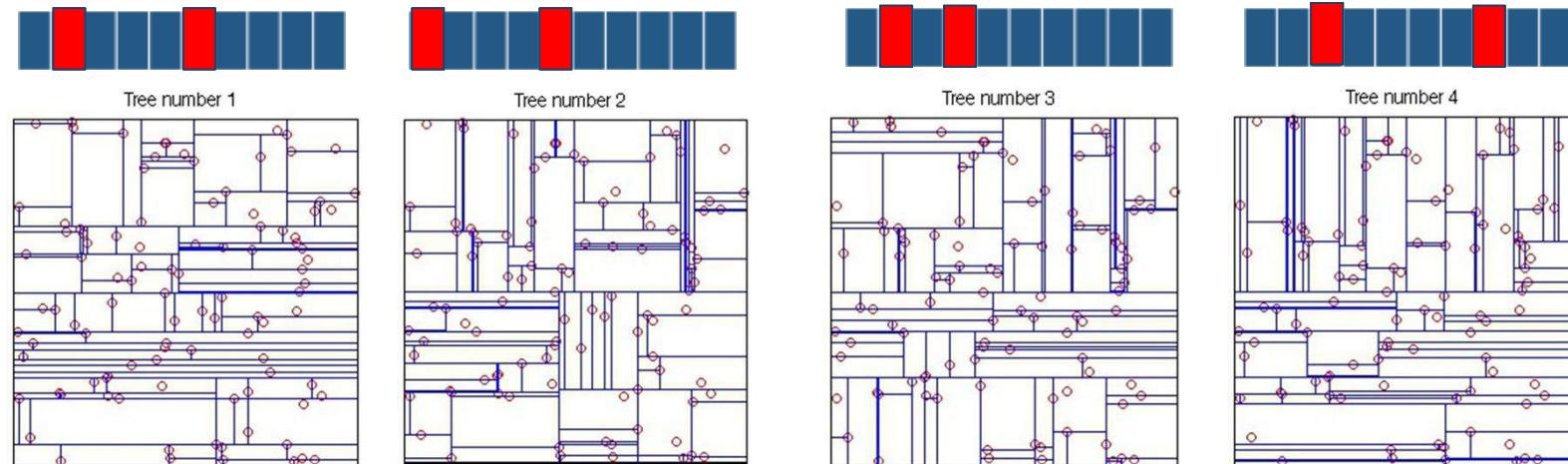
# Présentation du randomized kd-tree

# Motivation

Recherche exact dans les arbres (branch & band ou best bin first) rentable que si  $\text{dim} \ll 15$  à cause de la malédiction de la dimension

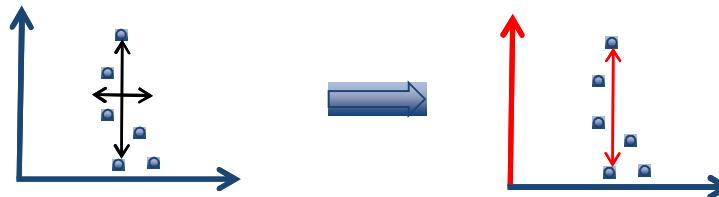
Au lieu de construire un seul arbre, on **construit L arbres de dimension réduite** en ne considérant qu'un **sous ensemble de composantes** dans chaque arbre

Les sous-espaces sont tirés aléatoirement **parmi les composantes** ou par **rotation aléatoire** de l'espace pour que les recherches dans chaque arbre soit **indépendantes** (complémentaires !!)

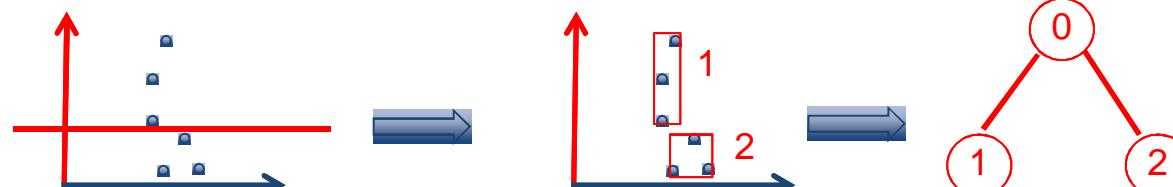


# Construction d'un kd-tree classique

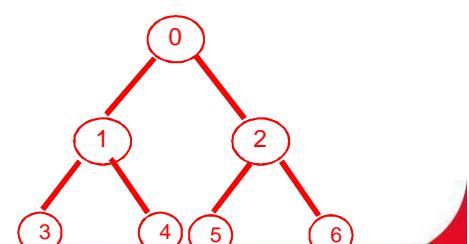
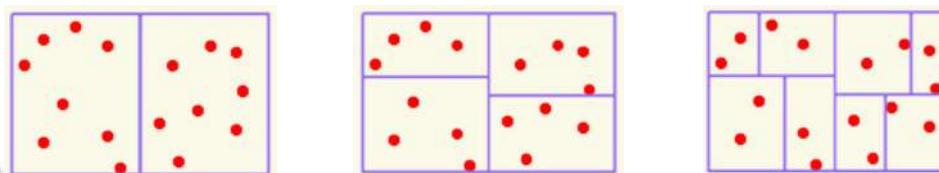
1. Parmi les  $d$  dimensions, on sélectionne celle pour laquelle la variance des points de la base est maximale



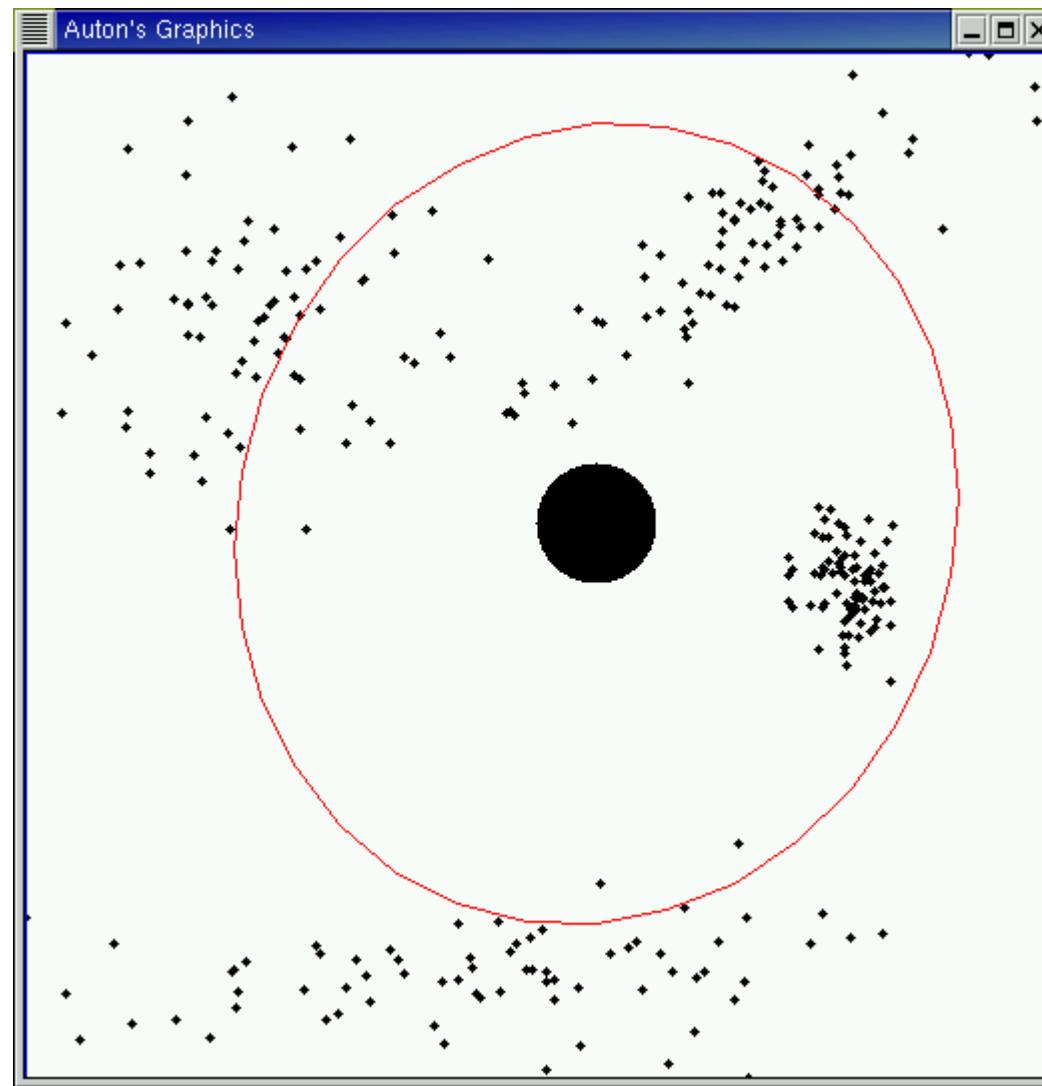
2. On coupe les données au niveau de la médiane et on calcule les paramètres de la forme englobante



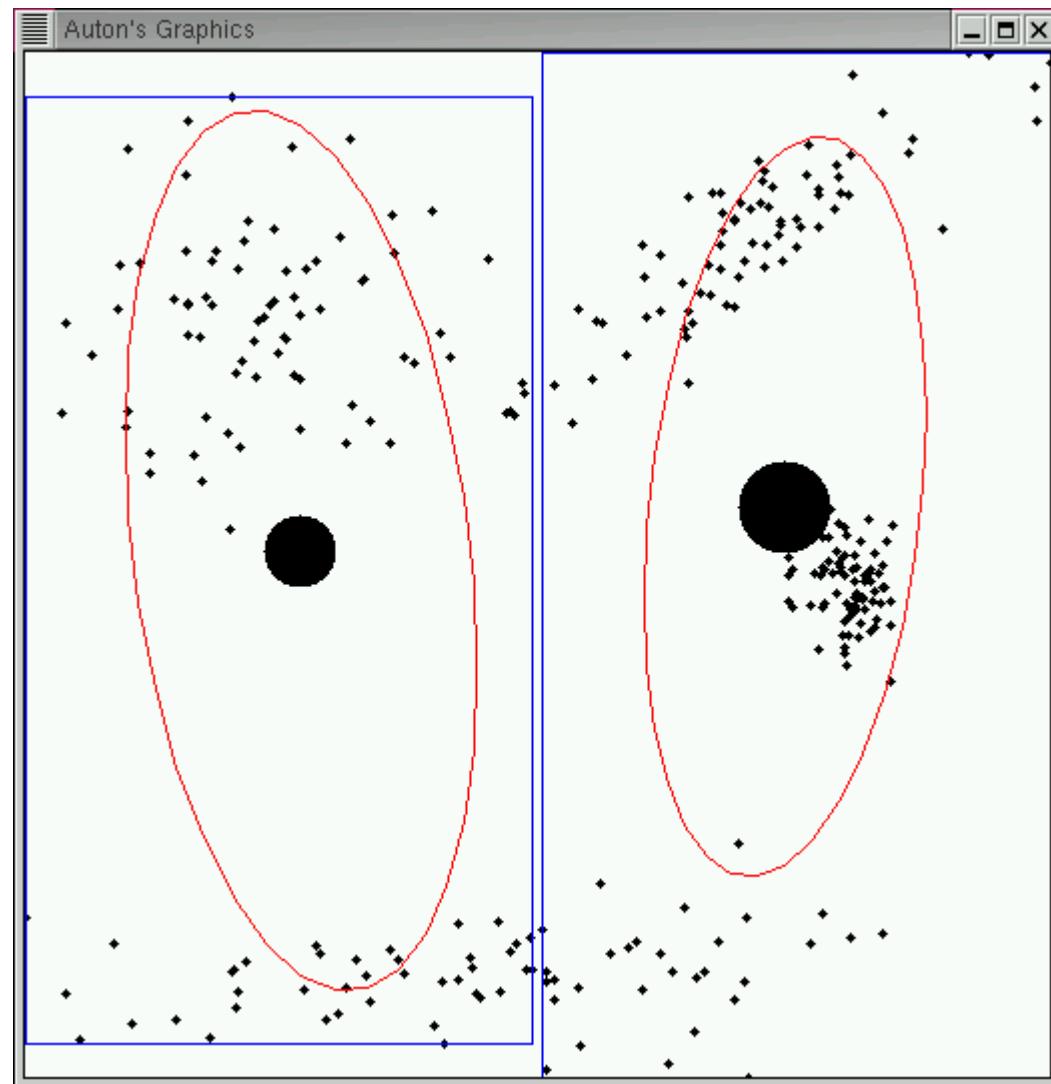
1. On réitère récursivement dans chaque partition fille.



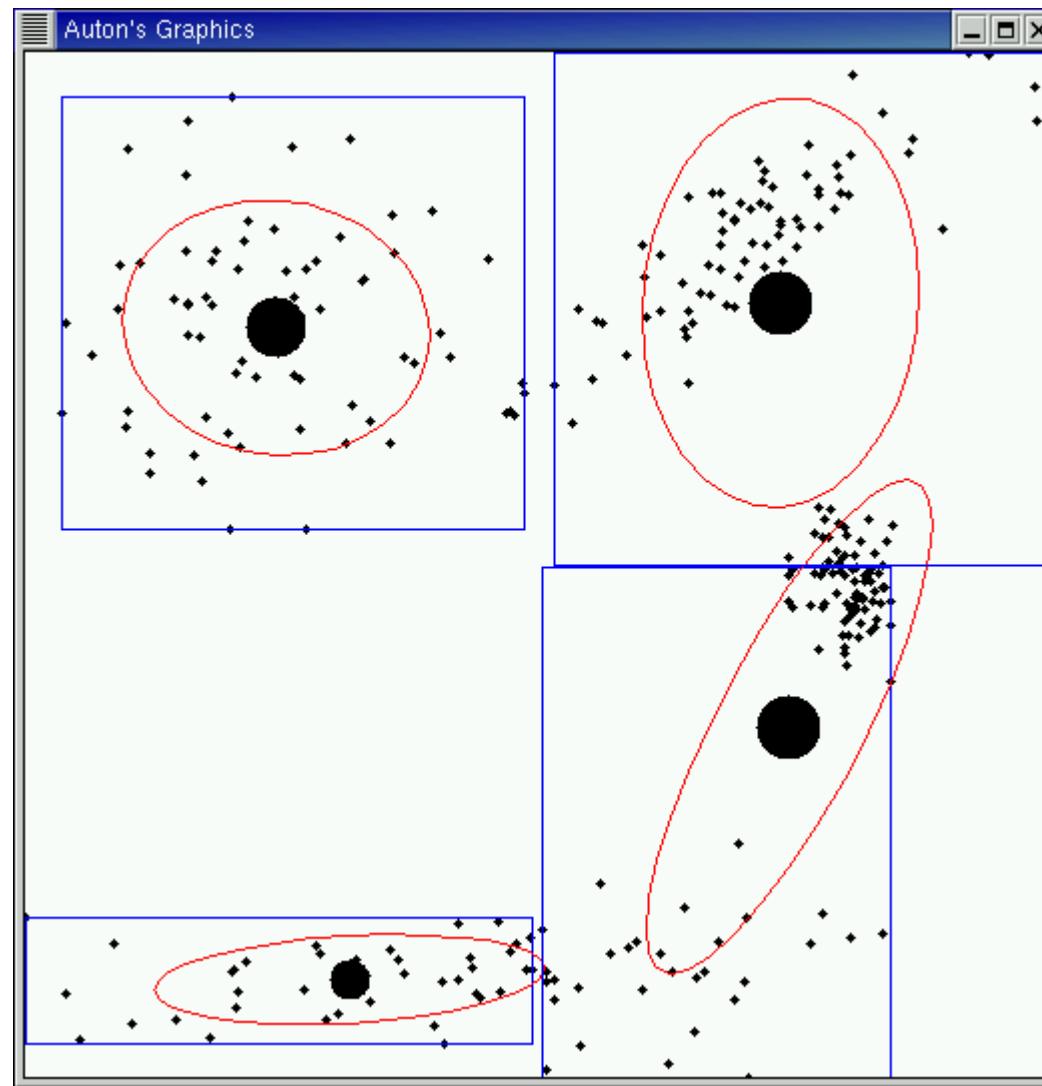
# Construction d'un kd-tree 2D



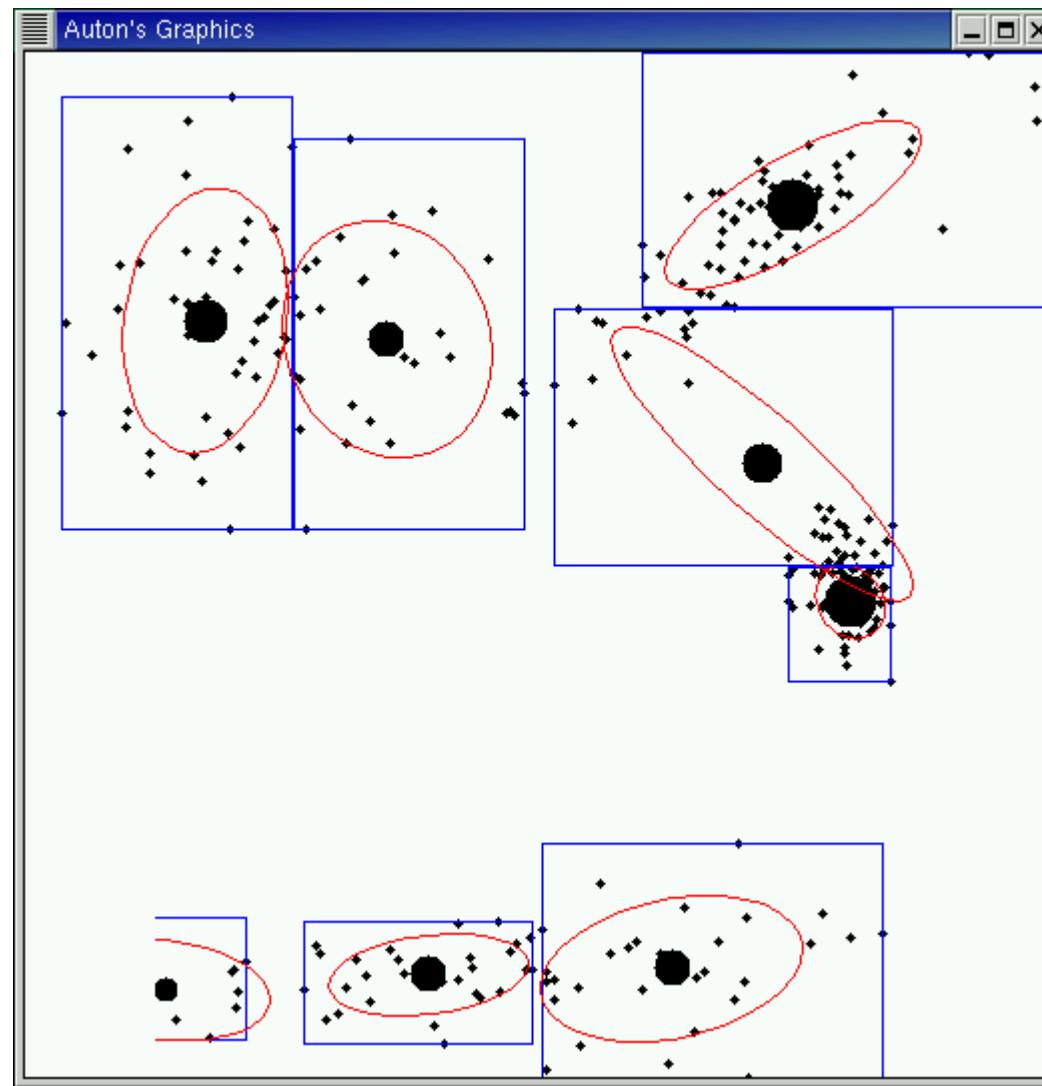
# Construction d'un kd-tree 2D



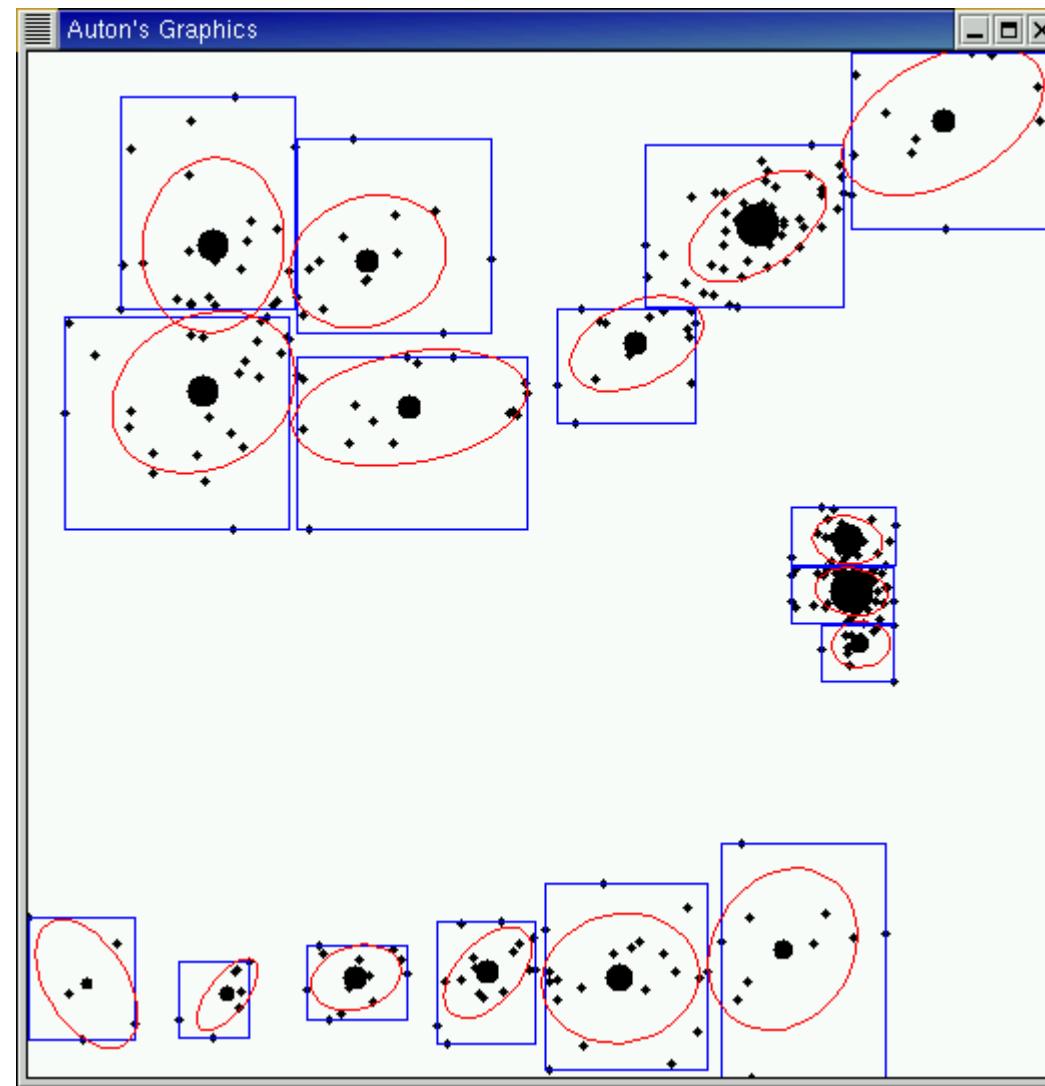
# Construction d'un kd-tree 2D



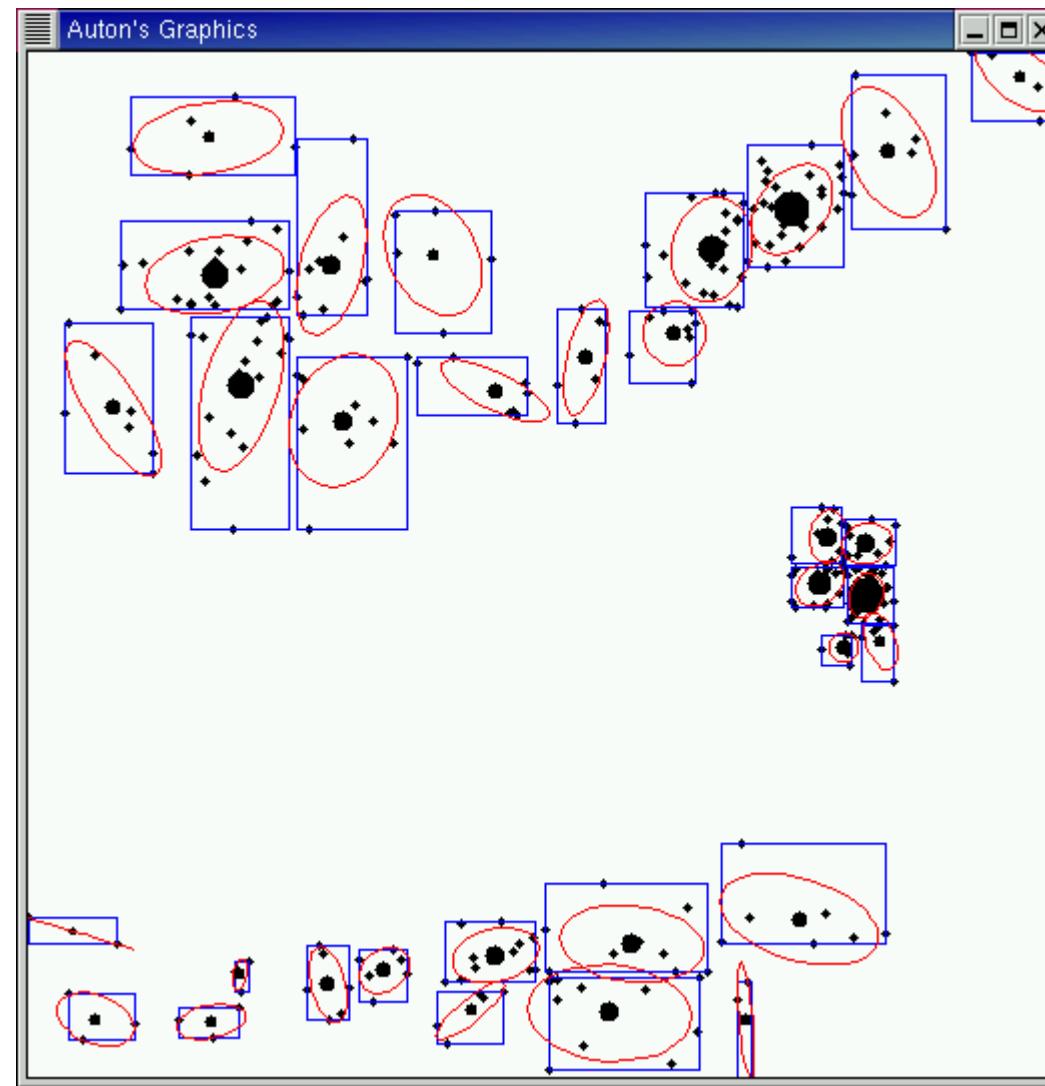
# Construction d'un kd-tree 2D



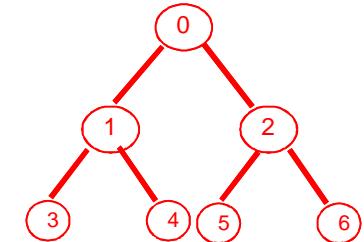
# Construction d'un kd-tree 2D



# Construction d'un kd-tree 2D



# Construction d'un kd-tree classique



L'arbre résultant est un arbre **binnaire balancé**

**Binaire** = chaque nœud a exactement deux fils

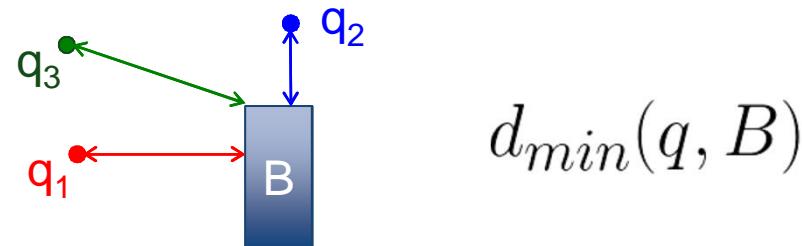
**Balancé** = Il y a toujours autant de points dans chacune des deux branches filles

La profondeur de l'arbre est  $\log_2(n)$  si on poursuit la procédure de construction jusqu'à ce que les feuilles ne contiennent plus qu'un point

Il peut être plus rentable de s'arrêter avant et de parcourir séquentiellement tous les points d'une feuille au moment de la recherche

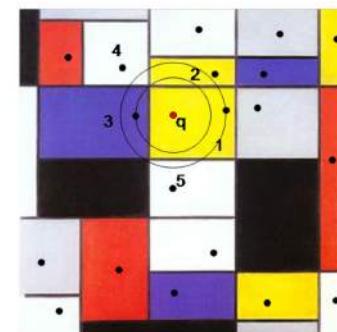
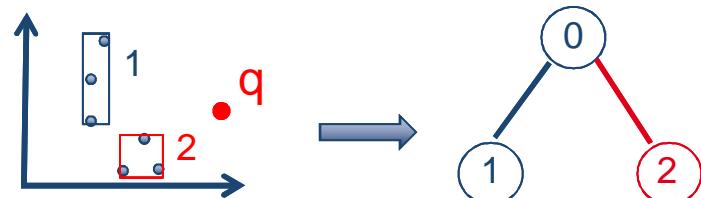
# Recherche dans un kd-tree classique

Borne min d'une région par rapport à la requête:



Algorithme de recherche = Branch and Bound

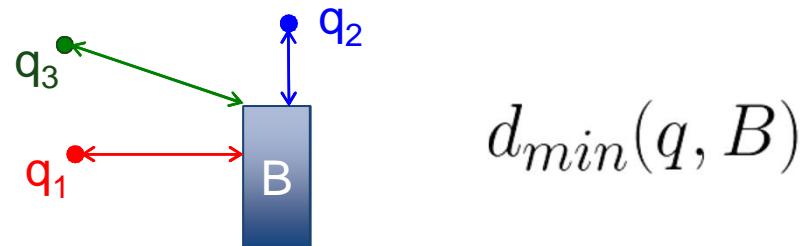
Parcours de l'arbre en profondeur, à chaque noeud, on choisit la région fille la plus proche de la requête:



Lorsque l'on rencontre un  $p'$  tel que  $d(p', q) > r_{NN}$  mise à jour du rayon de recherche :  $r_{NN} = d(p', q)$

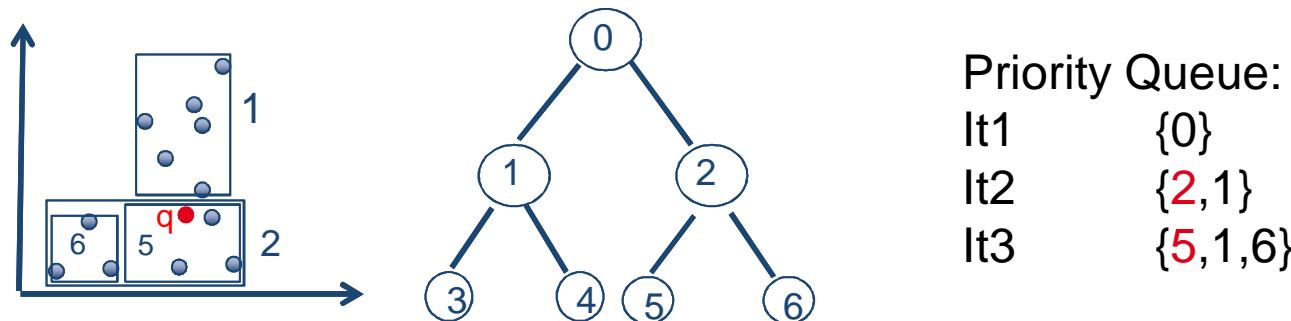
# Recherche dans un kd-tree classique

Borne min d'une région par rapport à la requête:



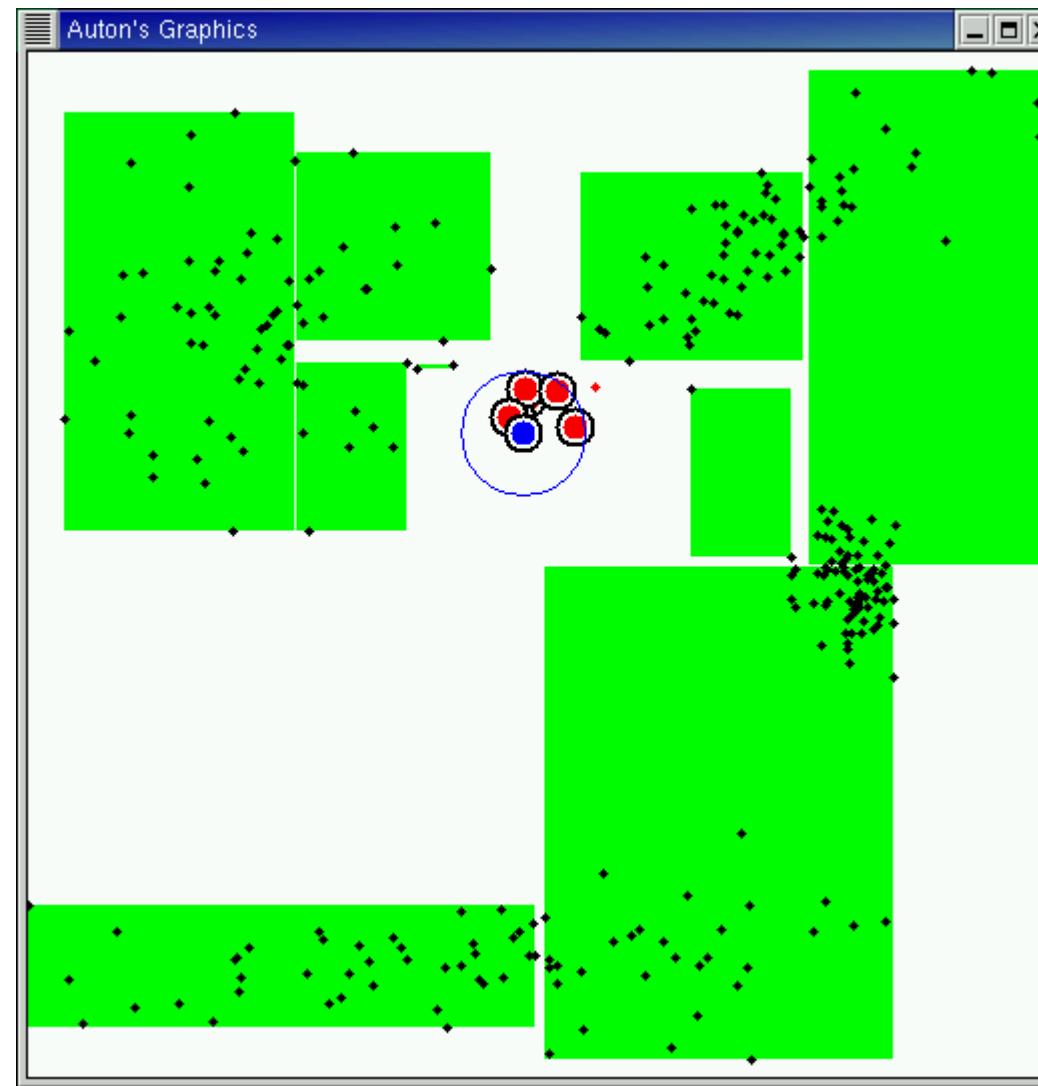
Algorithme de recherche = Best Bin First

On maintient les régions à parcourir dans une priority queue:

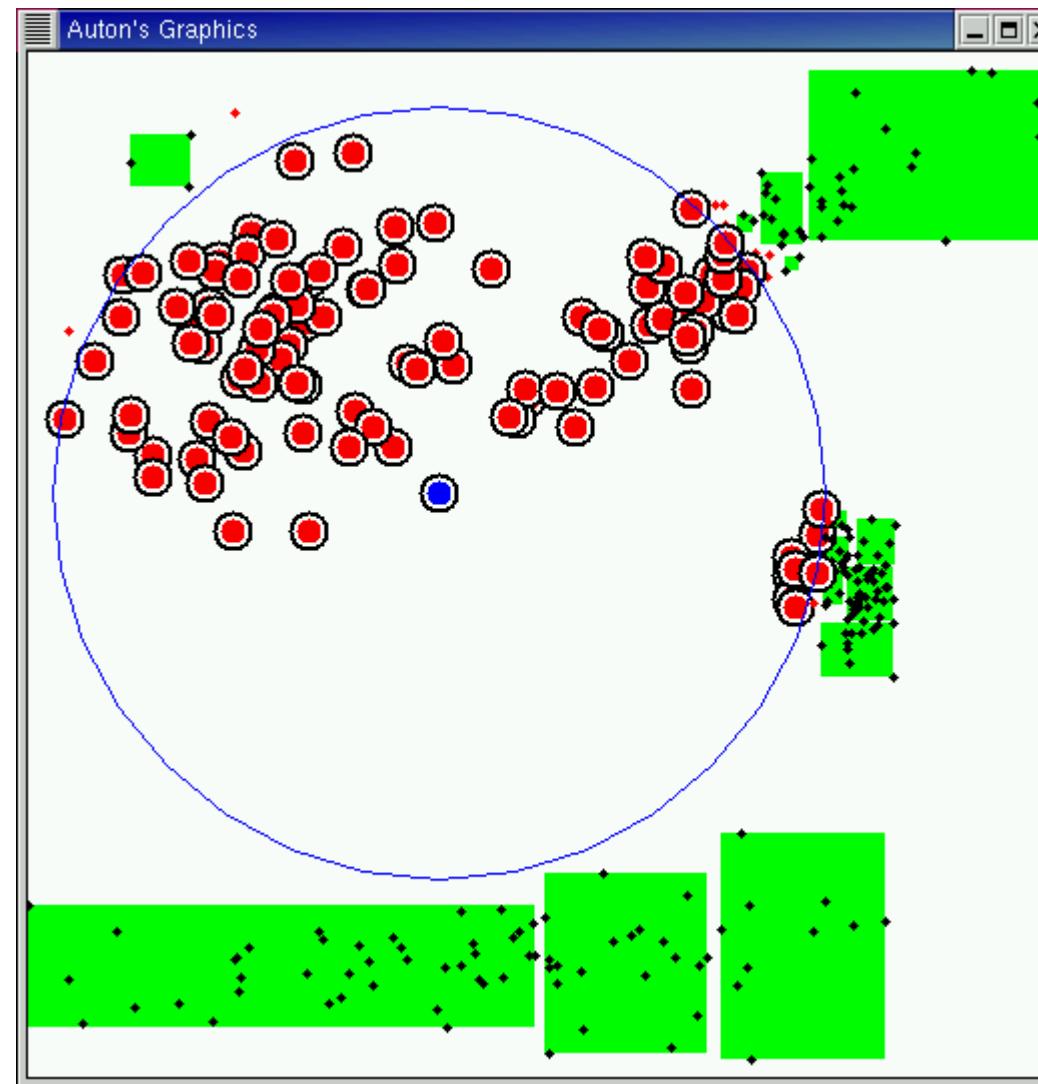


Lorsque l'on rencontre un  $p'$  tel que  $d(p',q) > r_{NN}$  mise à jour du rayon de recherche :  $r_{NN}=d(p',q)$

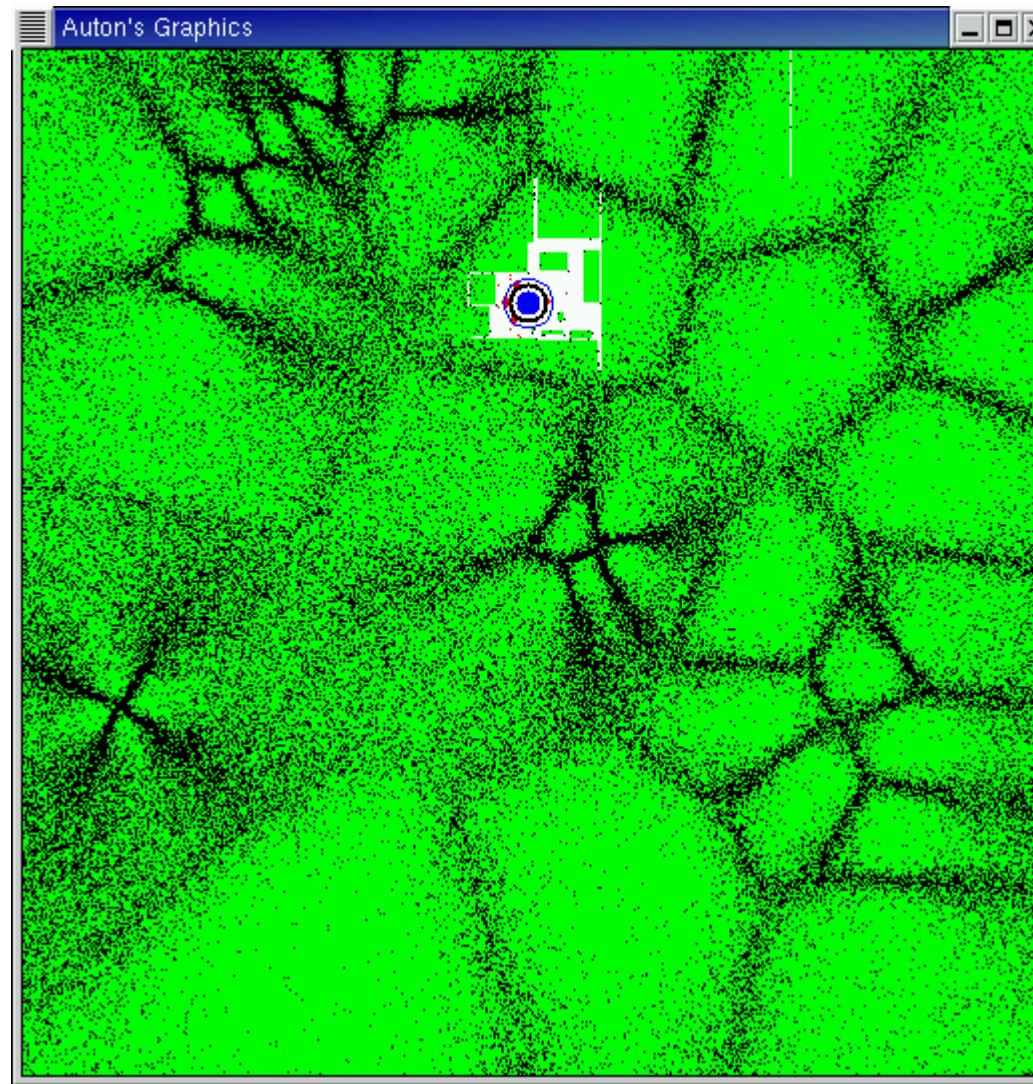
# Range Query: petit rayon



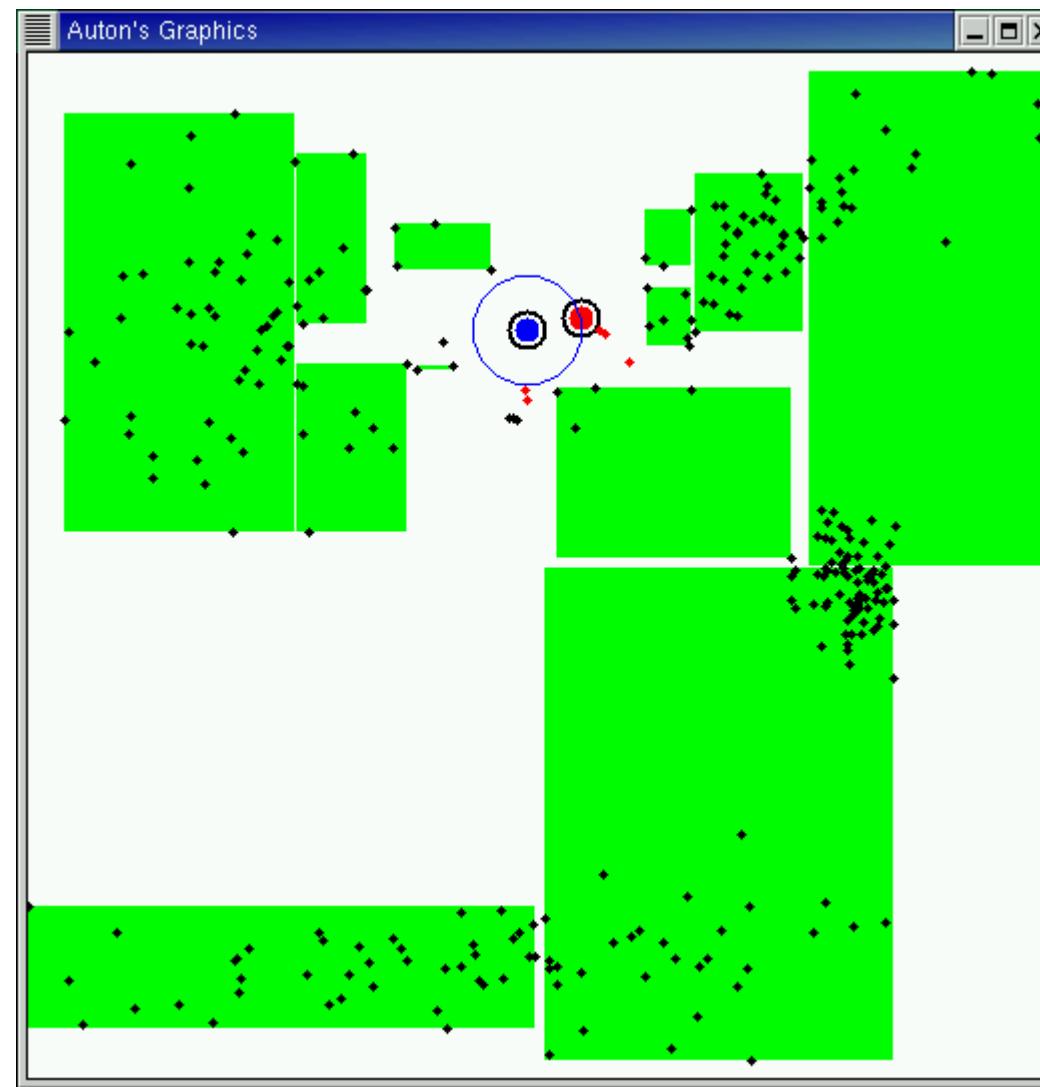
# Range Query: grand rayon



# Range Query: grande base

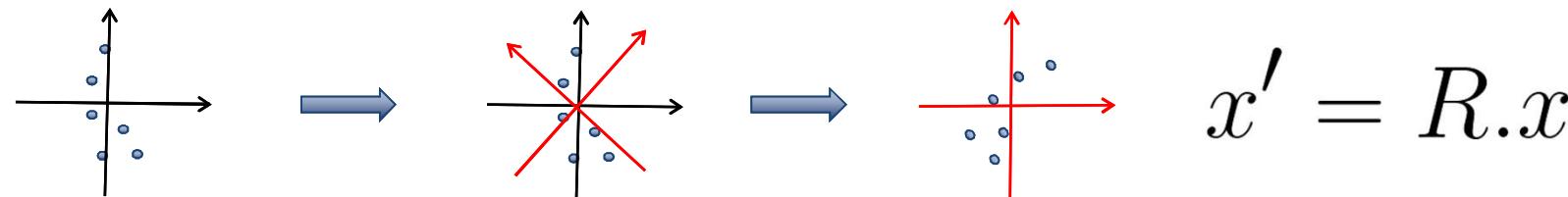


# Recherche du Plus Proche Voisin

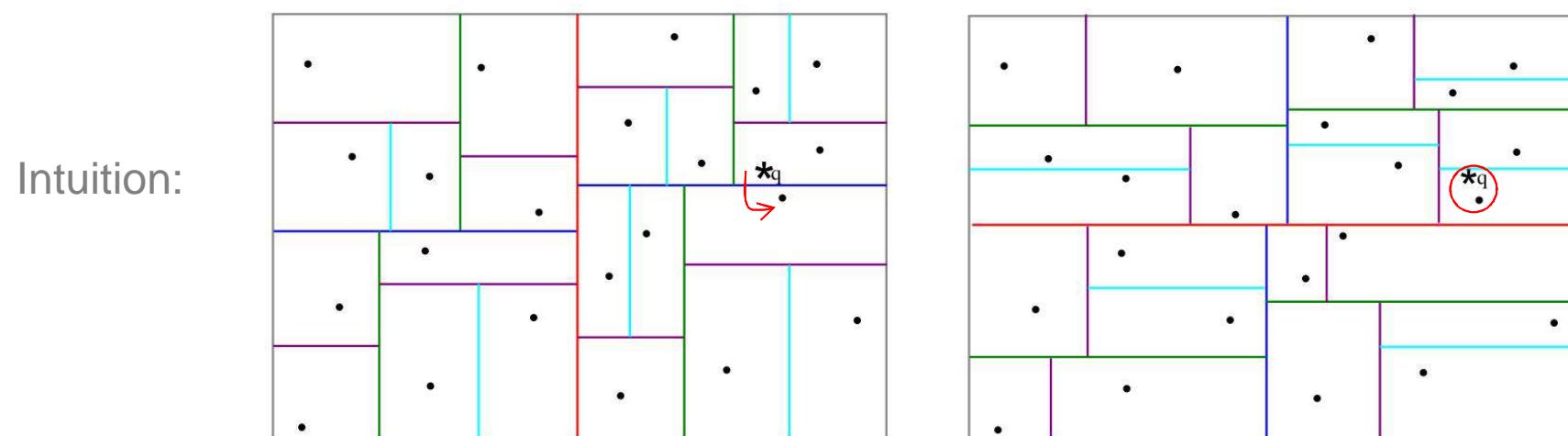


# Construction d'une forêt d'arbre “randomisés” par rotation aléatoire

1. Appliquer une rotation aléatoire sur les données



2. Construire un kd-tree classique sur l'espace résultant
3. Répéter L fois pour construire L arbres

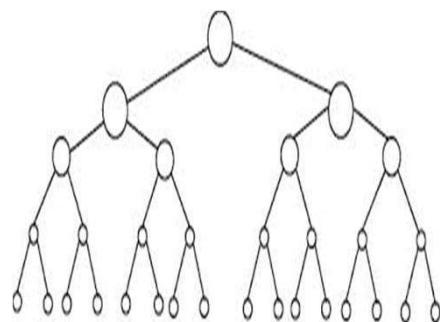


# Construction d'une forêt d'arbre “randomisés” par tirage de composantes aléatoires

1. Tirage aléatoire d'une **série de composantes** parmi les D composantes ayant la plus forte variance (exemple D=5)



1. Construire un kd-tree en utilisant la série d'axes déterminée aléatoirement



Split selon l'axe n° 2

Split selon l'axe n° 3

Split selon l'axe n° 2

Split selon l'axe n° 1

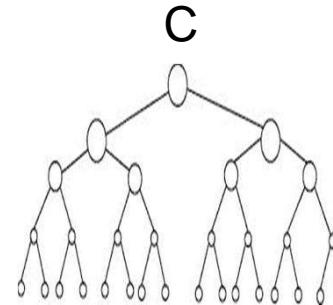
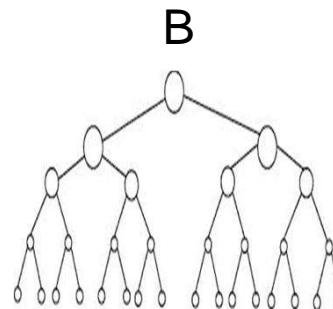
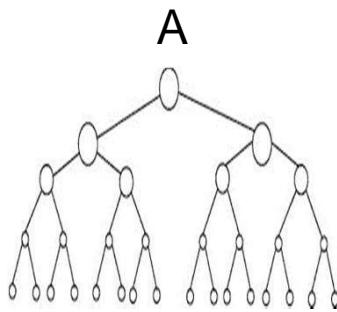
Split selon l'axe n° 4

1. Répéter L fois pour construire L arbres

**Avantages = construction et recherche plus simples & plus rapides car on travaille directement dans l'espace d'origine** (pas de projection matricielle avant split ou best bin selection)

# Recherche dans une forêt de kd-tree randomisés

On parcours tous les arbres en parallèle et on maintient une queue de priorité unique pour tous les arbres



Priority Queue (construite en fonction de  $d(q, B)$ ):

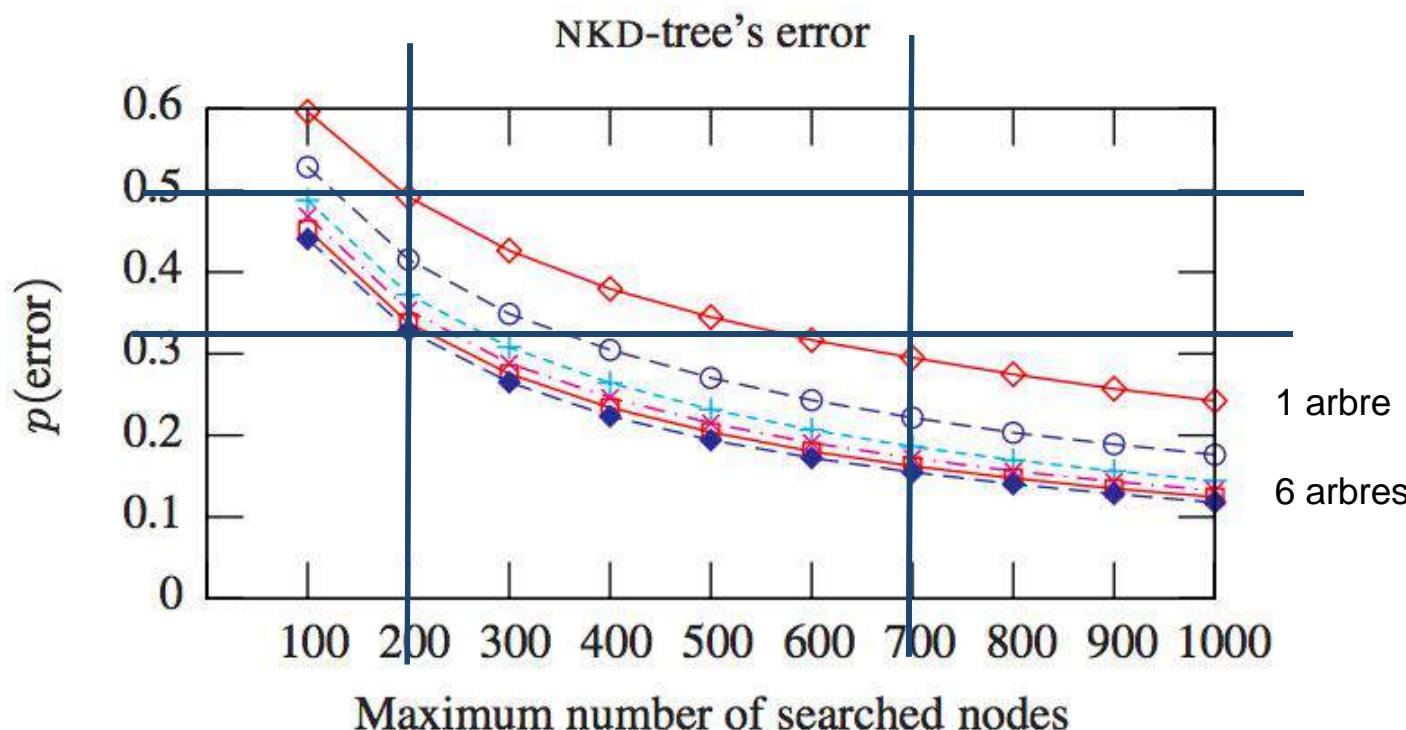
- It1 {**B0**, A0, C0}
- It2 {**B2**, A0, B1, C0}
- It3 {**A0**, B4, B1, C0, B3}
- It4 {**B4**, B1, A1, C0, A2, B3}

....

Condition d'arrêt = nombre de feuilles visitées pour lesquelles on mis à jour  
le  $r_{NN}$

# Impact du nombre d'arbres et du nombre d'itérations

Il est plus rentable d'augmenter le nombre d'arbres que d'augmenter le nombre de nœuds visités (nombre d'itération de la priority queue)



# Optimum sur le nombre d'arbres

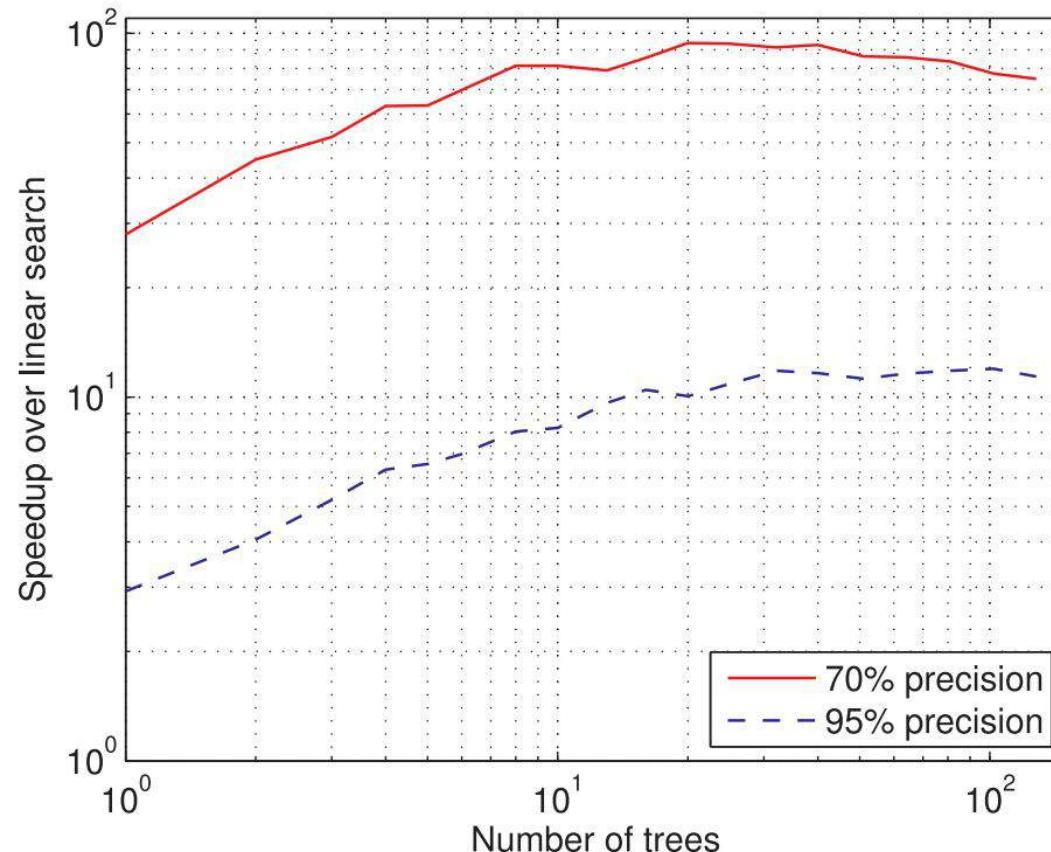


Fig. 1. Speedup obtained by using multiple randomized kd-trees (100K SIFT features data set).

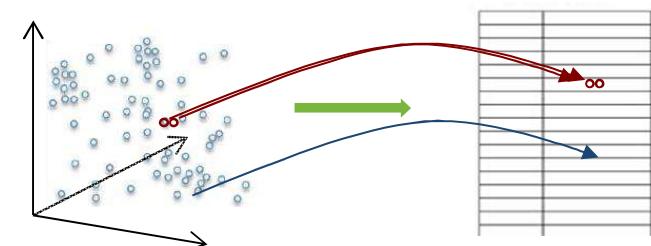
# Etude de LSH

# LSH: Intro

**LSH (Localité Sensitive Hashing), méthode de hachage introduite par Gionis, Indyk et Motwani en 1998 pour la recherche approximative dans les espaces de grande dimensions**

Sensible à la localité = la probabilité de collision des hash code est forte si les vecteurs sont proches et faible si les vecteurs sont éloignés

Formellement, selon Gionis et al. en 1998



$\mathcal{F}$  est une famille de fonctions  $h : \mathbb{R}^d \rightarrow S$  satisfaisant les conditions suivantes pour deux points quelconques  $p, q \in \mathbb{R}^d$  et une fonction  $h$  choisie aléatoirement parmi la famille  $\mathcal{F}$ :

- si  $d(p, q) \leq R$ , alors  $Pr_{h \in H}[h(p) = h(q)] \geq P_1$
- si  $d(p, q) \geq cR$ , alors  $Pr_{h \in H}[h(p) = h(q)] \leq P_2$

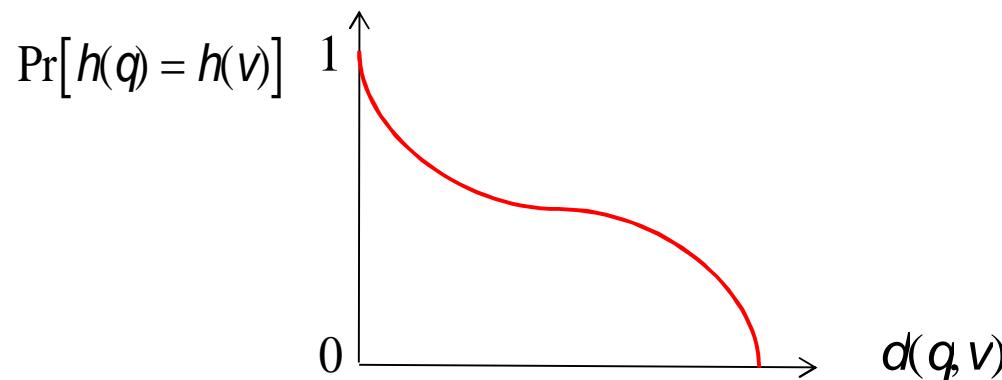
# LSH: Intro

**LSH (Localité Sensitive Hashing), méthode de hachage introduite par Gionis, Indyk et Motwani en 1998 pour la recherche approximative dans les espaces de grande dimensions**

Pour certaines familles LSH, la probabilité de collision peut même s'exprimer comme une fonction décroissante avec la distance

$$\Pr[h_\theta(q) = h_\theta(v)]_{p_\theta} = f(d(q, v))$$

- $\theta$  is i.i.d drawn from a known **data independent distribution**
- $f(d)$  is the sensitivity function (monotonically increasing from 0 to 1)



# LSH sensible au produit scalaire

Exemple: famille LSH sensible au produit scalaire

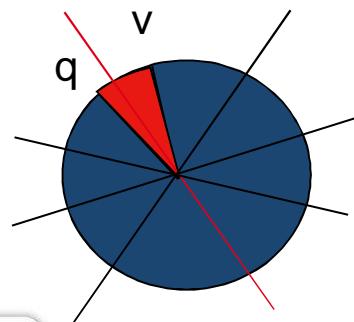
$$h(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x}) \quad p_w = N(0, I)$$

C'est le cas que l'on a déjà rencontré = projections aléatoires + quantification scalaire binaire

On peut montrer que la fonction de sensibilité vaut

Pour tous  $\mathbf{q}, \mathbf{v} \in \mathbb{R}^d$        $\Pr[h_w(\mathbf{q}) = h_w(\mathbf{v})]_w = 1 - \frac{1}{\pi} \cos^{-1}\left(\frac{\mathbf{q}^T \mathbf{v}}{\|\mathbf{q}\| \|\mathbf{v}\|}\right)$

Interprétation dans le cas normé:



$$\begin{aligned} \Pr[h_w(\mathbf{q}) \neq h_w(\mathbf{v})]_w &= \text{angle}(\mathbf{q}, \mathbf{v}) / \pi = \frac{1}{\pi} \cos^{-1}(\mathbf{q}^T \mathbf{v}) \\ &= \frac{1}{\pi} \cos^{-1}\left(1 - \frac{(d(\mathbf{q}, \mathbf{v}))^2}{2}\right) \end{aligned}$$

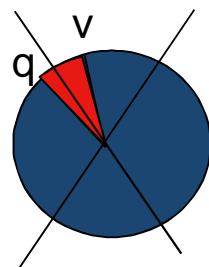
# LSH sensible au produit scalaire

Exemple: famille LSH sensible au produit scalaire

$$h(x) = \text{sgn}(w^T x) \quad p_w = N(0, I) \quad \Pr[h_w(q) = h_w(v)]_w = 1 - \frac{1}{\pi} \cos^{-1}\left(\frac{q^T v}{\|q\| \|v\|}\right)$$

On forme un hash code de D bits en utilisant D fonctions de hachage de la famille  $\mathcal{F}$

$$z(x) = [h_1(x), \dots, h_D(x)] = \text{sign}(Wx)$$



$$z(q) = 1 \text{ hash code} = [0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \dots 0 \ 0 \ 1 \ 0]$$



$$\approx \frac{D}{\pi} \cos^{-1}\left(\frac{q^T v}{\|q\| \|v\|}\right)$$

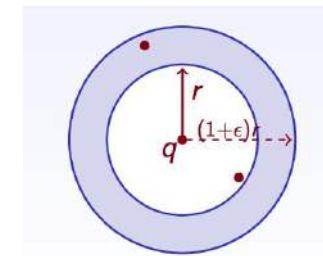
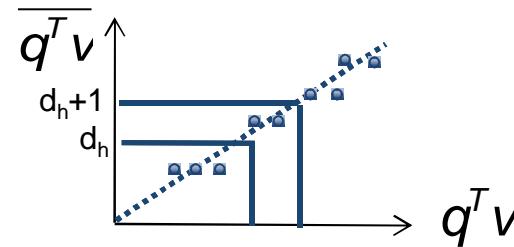
$$z(v) = 1 \text{ hash code} = [1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \dots 0 \ 1 \ 0 \ 0]$$

# LSH sensible au produit scalaire

La distance de Hamming entre deux hash code est un estimateur de la probabilité de collision et donc du produit scalaire entre  $q$  et  $v$

$$\overline{q^T v} = \|q\| \|v\| \cos\left(\frac{\pi}{D} d_h(\zeta(q), \zeta(v))\right)$$

Variance  $\sigma(k, q^T v)$



La variance  $\sigma(k, q^T v)$  de l'estimateur décroît avec le nombre de bits et l'estimateur converge vers le produit scalaire exact

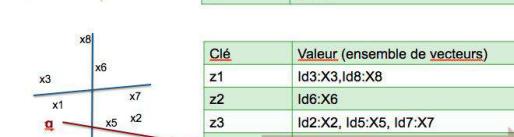
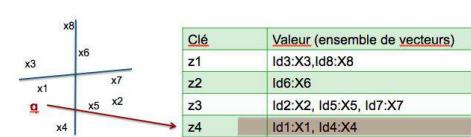
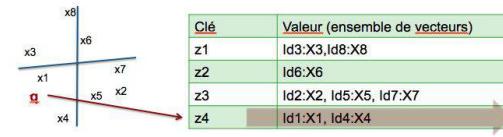
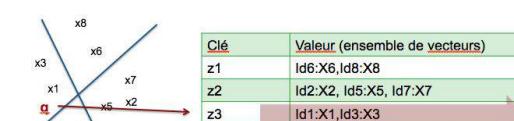
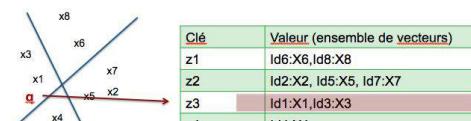
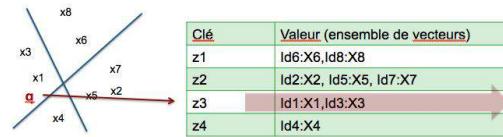
$$\lim_{k \rightarrow \infty} \overline{q^T v} = q^T v$$

Les  $k$  plus proches voisins dans l'espace de Hamming convergent vers les  $k$  plus proches voisins exactes

# LSH: algorithme de base pour l'indexation et la recherche

Tables multiples: création de L tables dont les clé sont générées par k fonctions de hachage (hash code de taille k pour chaque table). Taille index O(Ln)

Recherche = accès simple dans chaque table O(L)



# LSH: algorithme de base

Probabilité de collision dans une table de  $k$  bits:

$$\Pr[h_w(q) = h_w(v)]_w = f(d(q, v)) \xrightarrow{\text{Fonction de sensibilité}} \Pr[z(q) = z(v)]_w = (f(d(q, v)))^k$$

Probabilité de non collision dans une table:

$$\Pr[z(q) \neq z(v)]_w = 1 - (f(d(q, v)))^k$$

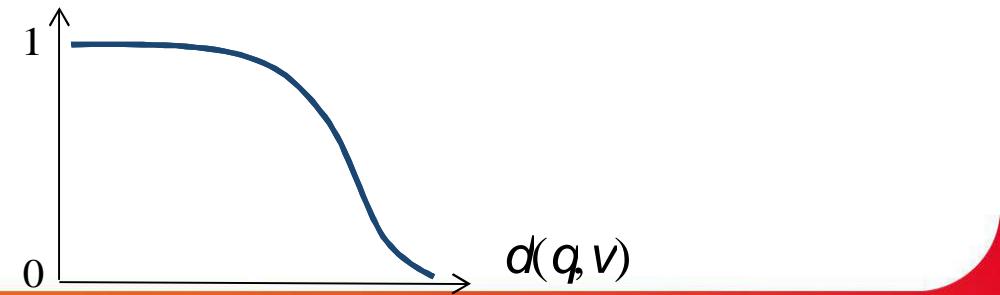
Très faible

Probabilité de non collision dans  $L$  tables:

$$(1 - (f(d(q, v)))^k)^L$$

Probabilité de collision dans au moins une table:

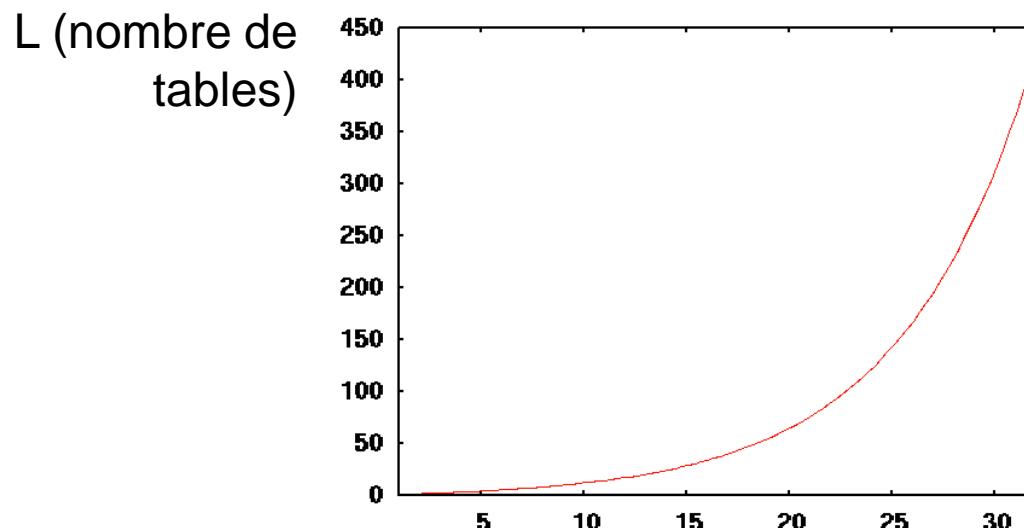
$$1 - (1 - (f(d(q, v)))^k)^L$$



## LSH: algorithme de base

Si on veux retrouver  $\alpha$  (e.g 95%) des  $v$  tels que  $d(q,v) < r$ , il faut que la probabilité de collision dans au moins une table soit supérieure à  $\alpha$

$$1 - \left(1 - (f(r))^k\right)^L > \alpha \quad \Rightarrow \quad L > \frac{\ln(1 - \alpha)}{\ln(1 - f(r)^k)}$$



A qualité  $\alpha$  constante et rayon  $r$  constant, le nombre de tables doit croître pour compenser l'augmentation de  $k$

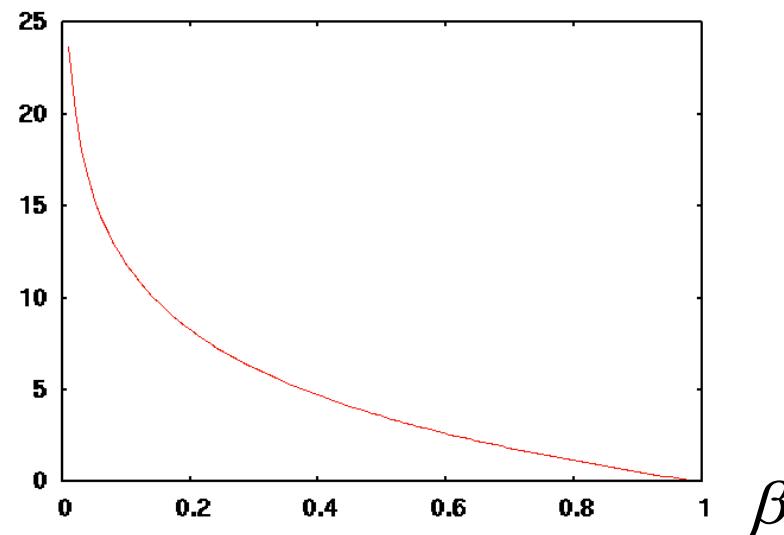
## LSH: algorithme de base

D'un autre côté, l'augmentation de  $k$  permet de réduire la probabilité de fausses collisions dans chaque table. Si on veux que la probabilité de collision soit faible lorsque  $d(q,v) > r + \varepsilon$

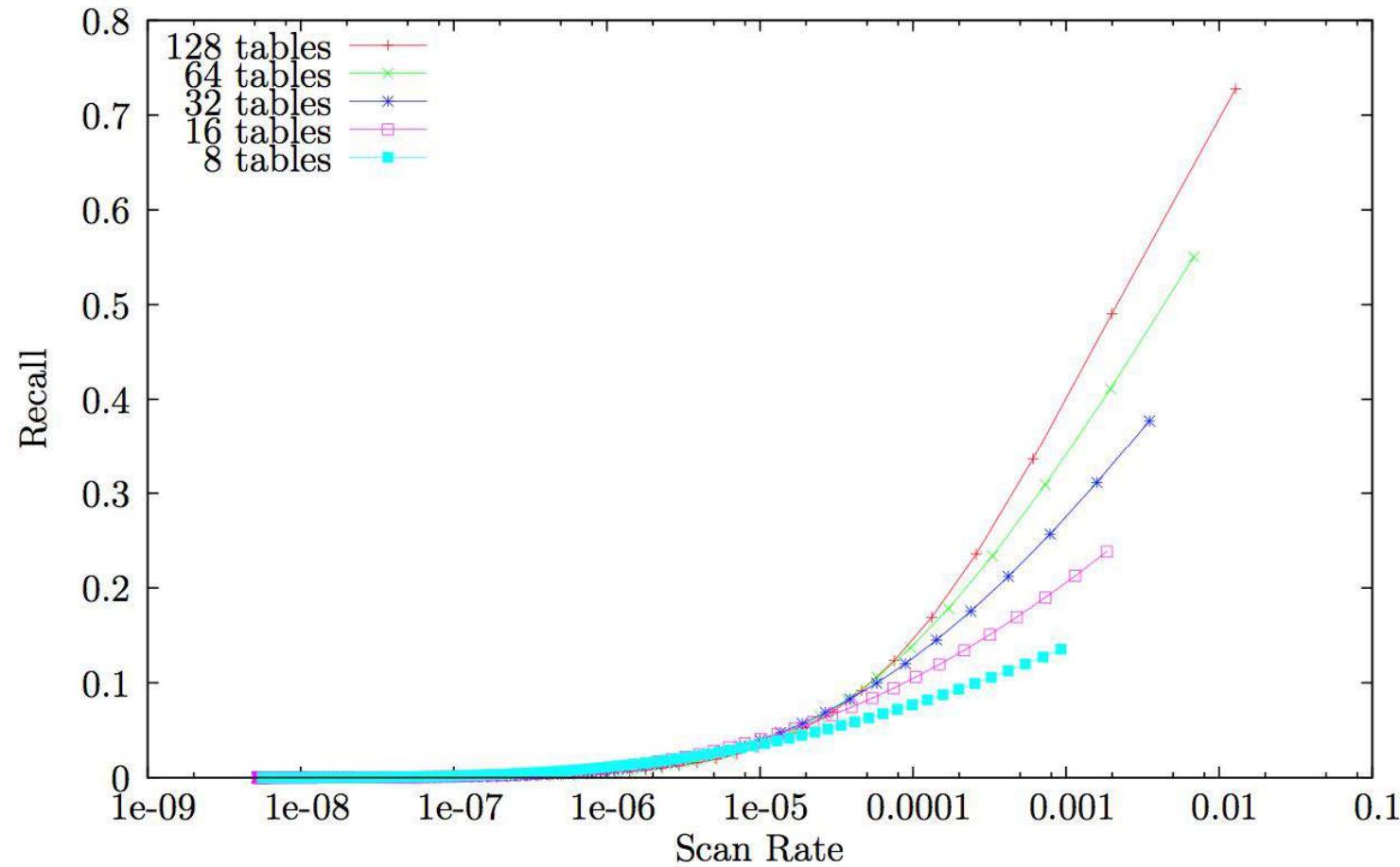
Proba de collision dans une table

$$n(f((1 + \varepsilon)r))^k < \beta \quad \Rightarrow \quad k > \frac{\ln(\beta)}{\ln(f((1 + \varepsilon)r))}$$

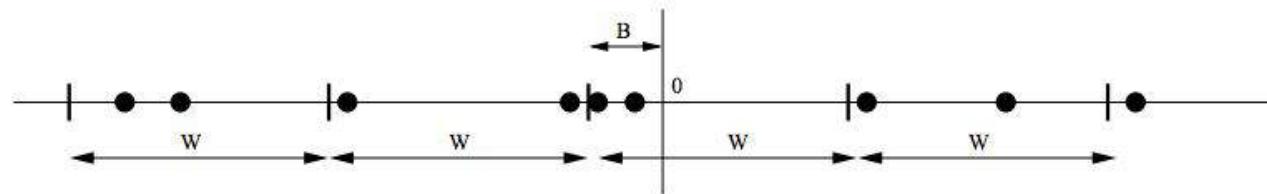
$k$  (nombre de bits par table)



# Impact du nombre de tables



# LSH sensible aux distances L<sub>p</sub>



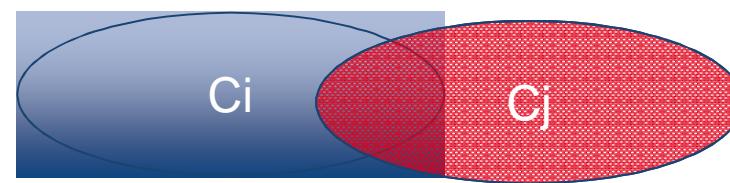
- Consider  $h_{\mathbf{a}, b} \in \mathcal{H}^w$ ,  $h_{\mathbf{a}, b}(\mathbf{v}) : \mathcal{R}^d \rightarrow \mathcal{N}$
- $\mathbf{a}$  is a  $d$  dimensional random vector whose each entry is drawn from a  $p$ -stable distr
- $b$  is a random real number chosen uniformly from  $[0, w]$  (random shift)
- $h_{\mathbf{a}, b}(\mathbf{v}) = \lfloor \frac{\mathbf{a} \cdot \mathbf{v} + b}{w} \rfloor$

# LSH sensible à la distance de Jaccard

MinHash: the min-wise independent permutations locality sensitive hashing scheme

Jaccard distance = mesure de similarité entre des ensembles d'objets

$$\text{sim}_J(C_i, C_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}$$



- View sets as columns of a matrix; one row for each element in the universe.  $a_{ij} = 1$  indicates presence of item i in set j
- Example

	$C_1$	$C_2$
0	1	
1	0	
1	1	
0	0	
1	1	
0	1	

$\text{sim}_J(C_1, C_2) = 2/5 = 0.4$

Hash function=

- Randomly permute rows
- Hash  $h(C_i)$  = index of first row with 1 in column  $C_i$
- Surprising Property

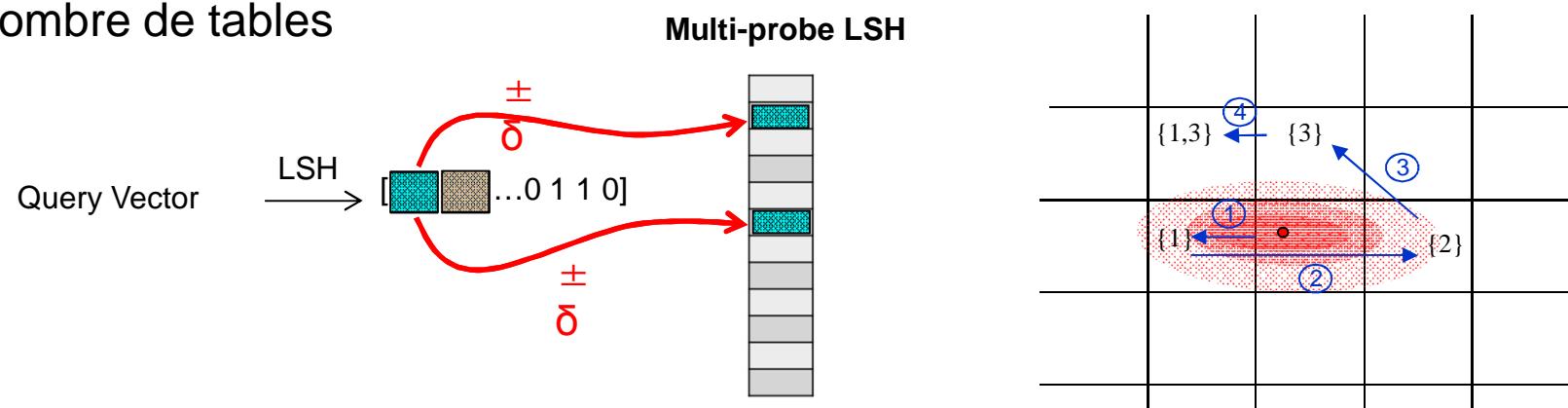
$$P[h(C_i) = h(C_j)] = \text{sim}_J(C_i, C_j)$$

# Limitations de LSH et variantes

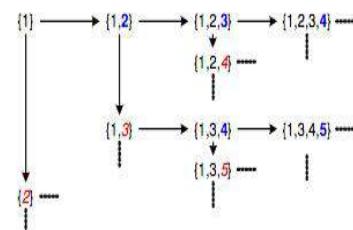
# Problème de LSH: espace mémoire et multi-probe LSH

LSH peut nécessiter un grand nombre de tables pour atteindre des qualités suffisantes ce qui rend la méthode problématique lorsque les données sont grandes

→ Utilisation d'accès multiples dans chaque table pour réduire le nombre de tables



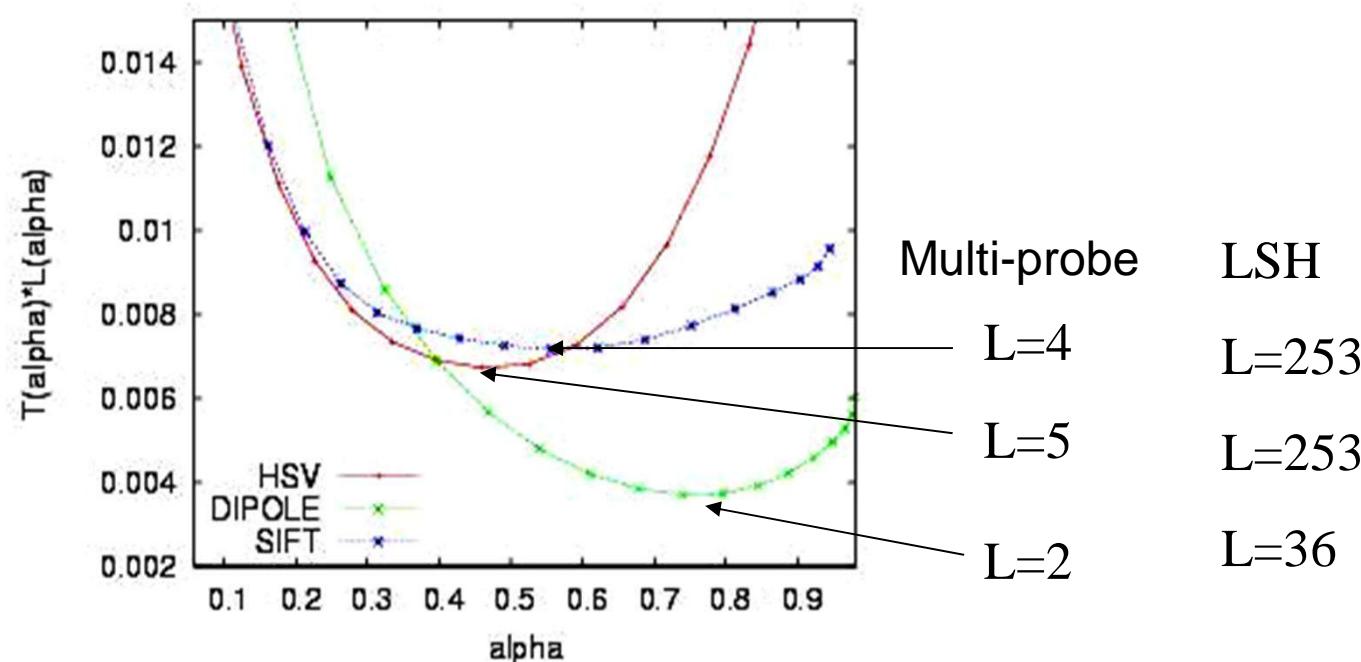
→ Algorithme glouton pour les bucket de la plus probable à la moins probable



## Nombre optimal de tables << LSH

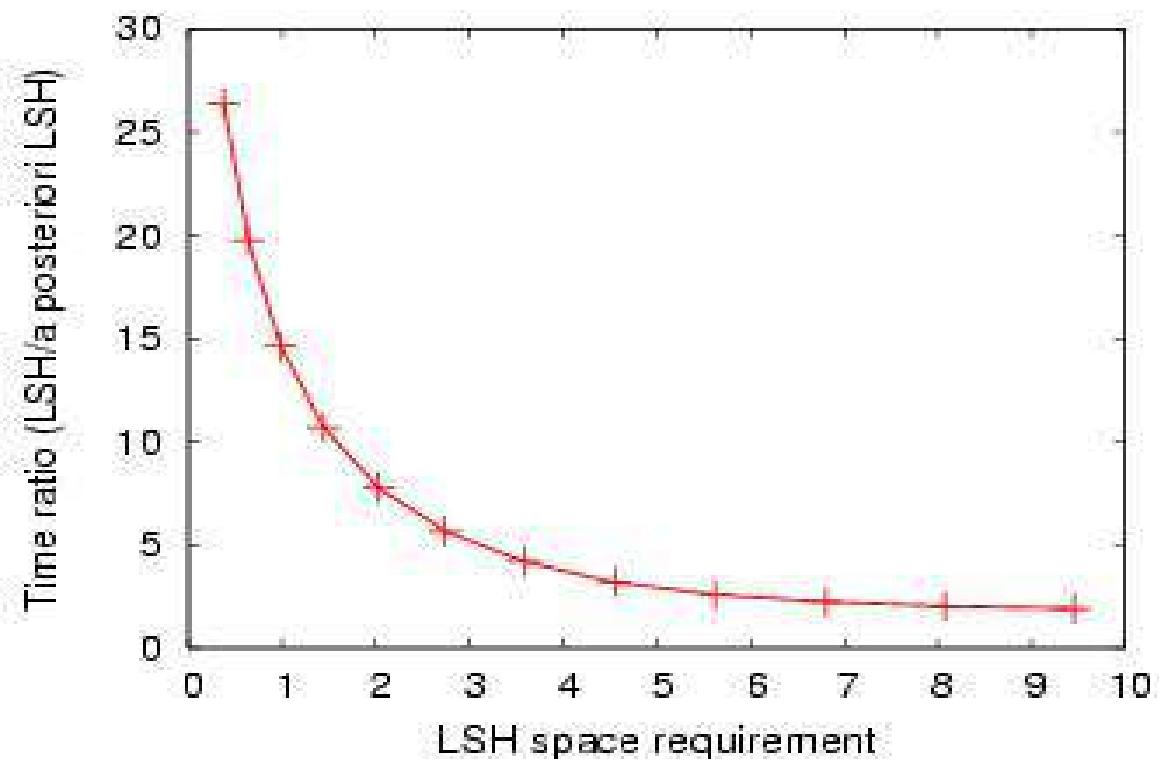
$$\alpha_T = 1 - (1 - \alpha)^L \longrightarrow L = \frac{\ln(1 - \alpha_T)}{\ln(1 - \alpha)}$$

$$\alpha \gg \alpha_{LSH}$$



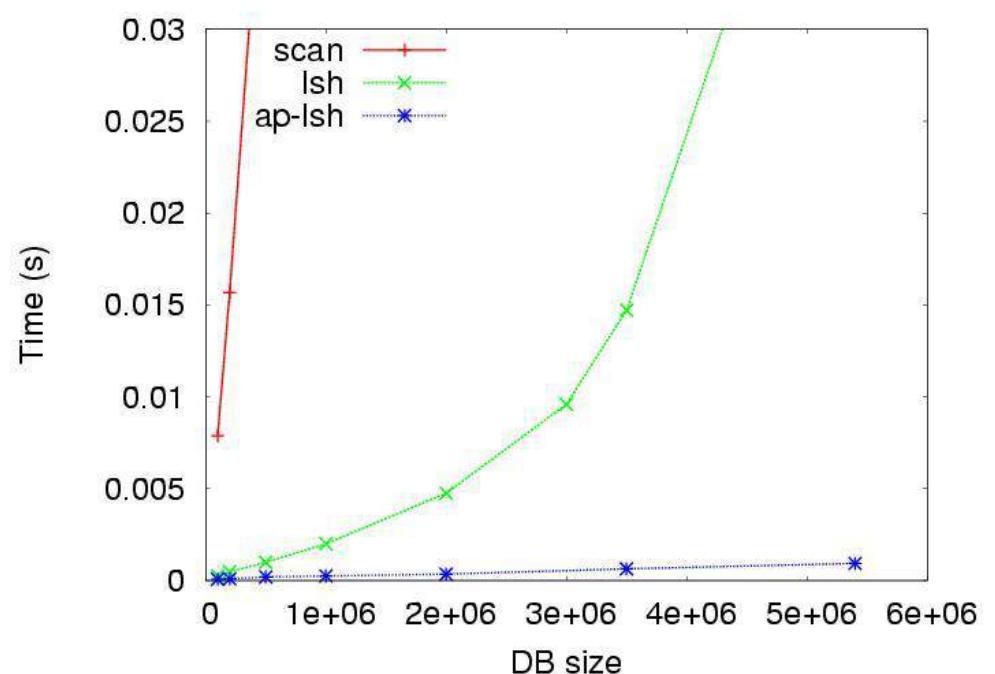
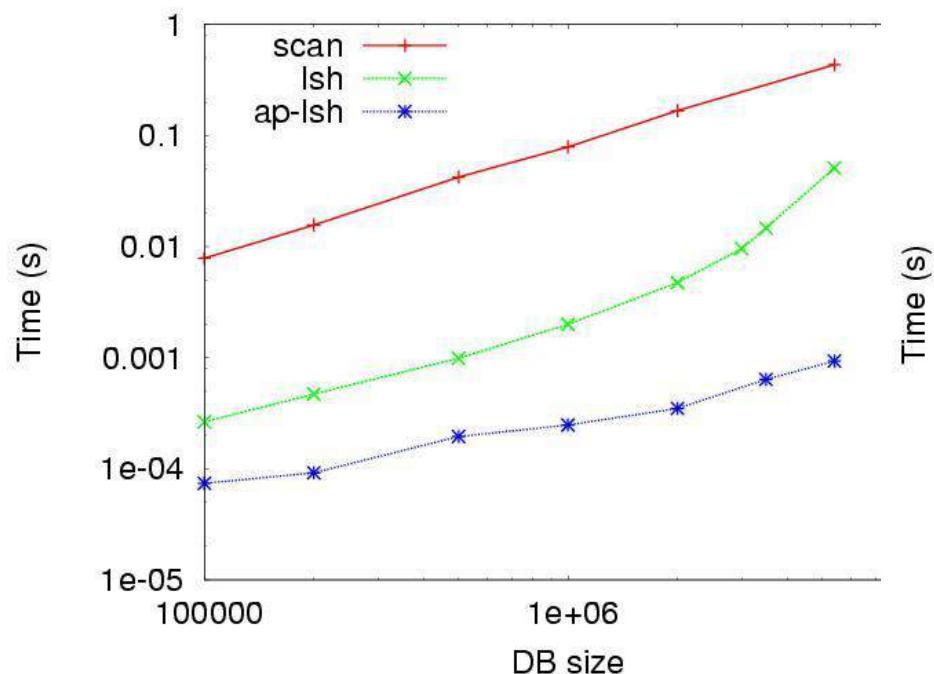
# Time/Space performances vs. LSH

- Time Gain vs. LSH space requirements



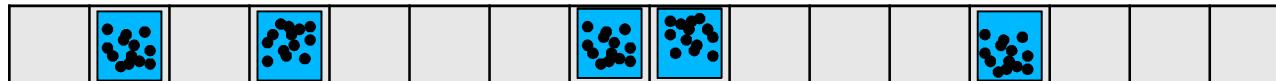
# Influence de la taille de la base

Limited memory (4 Gb)



## Autre problème de LSH: mauvais balancement

Sur certaines données réelles, LSH peut conduire à des partitions loin de l'uniformité. Certaines buckets peuvent concentrer presque tous les points:



- Augmentation du nombre de fausses collisions
- Augmentation du temps de raffinement
- Problématique dans les contexte distribués (load balancing)

Solution: utiliser des fonctions de hachage dépendant des données.

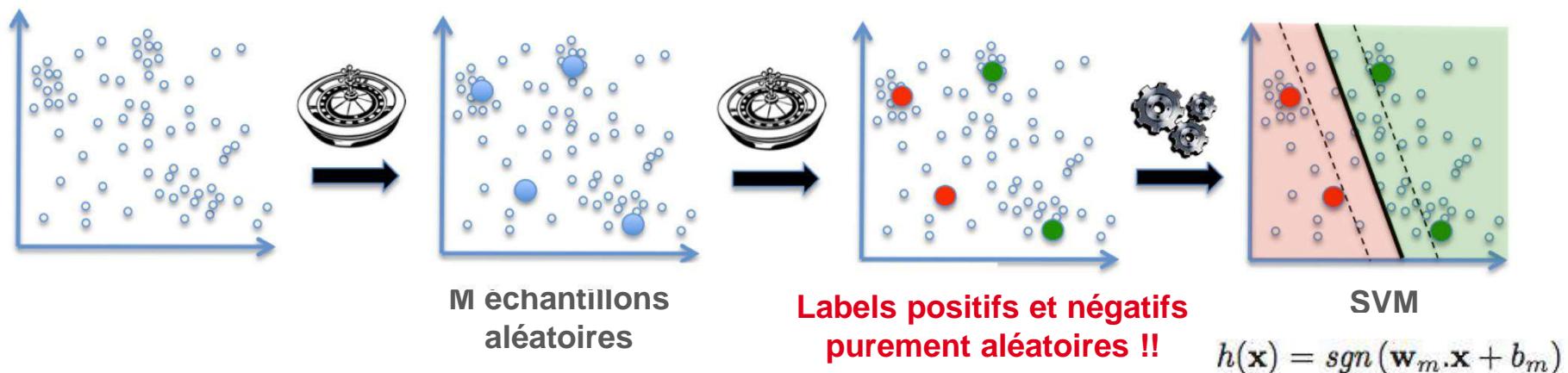
De nombreuses fonctions « data dependent » ont été proposées récemment dans la littérature (spectral-hashing, KLSH, RMMH, etc.)

# Random Maximum Margin Hashing

RMMH = une famille de fonctions de hachage basée sur le **partitionnement aléatoire des données**

Avantage = fonctions indépendantes + adaptabilité aux données

Basé sur **un concept théorique nouveau = l'apprentissage aléatoire de classifieurs** (1 par fonction de hachage = 1 par bit)



# Random Maximum Margin Hashing

## Pourquoi ça marche ?

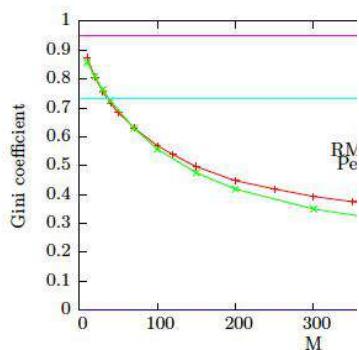
Cas extrême:  $M$  (nombre d'échantillons) =  $N$  (toute la base)

Uniformité

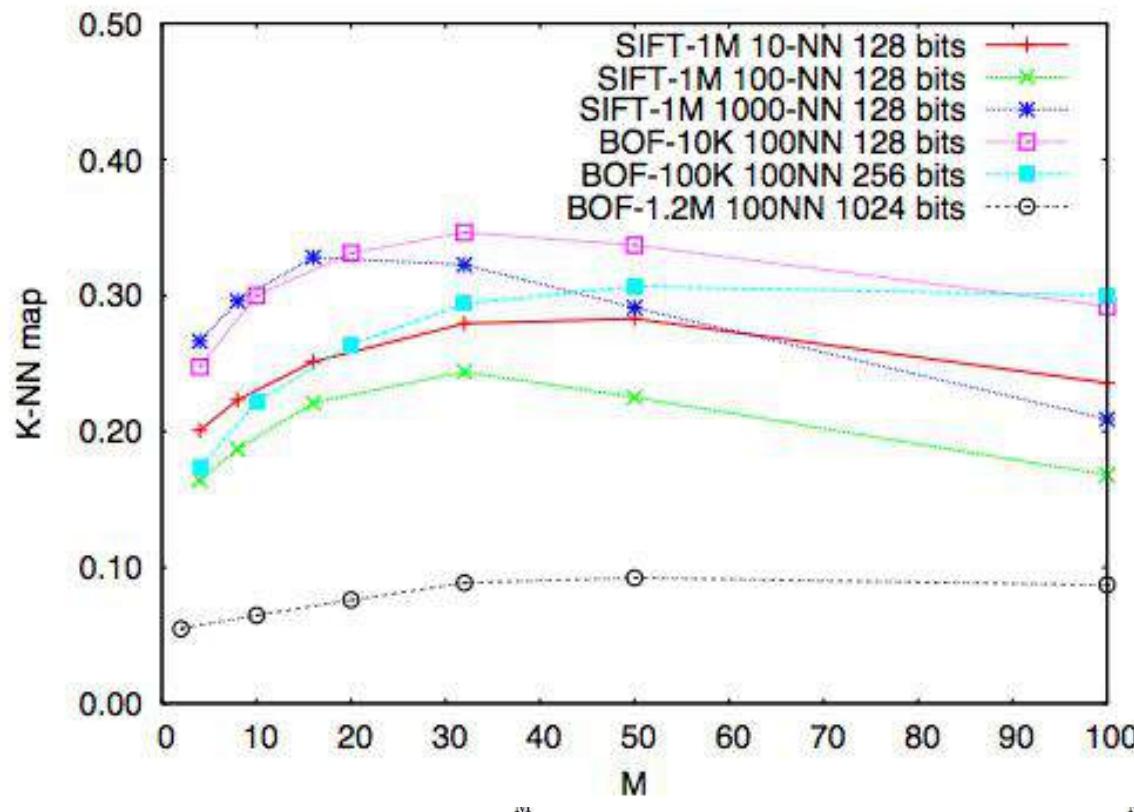
Overfitting

$M$  trop faible:

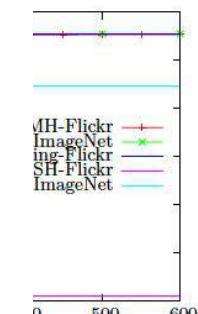
$M$  optimal = c



(a) Gini coefficient



(b) Avg. Max. bucket size



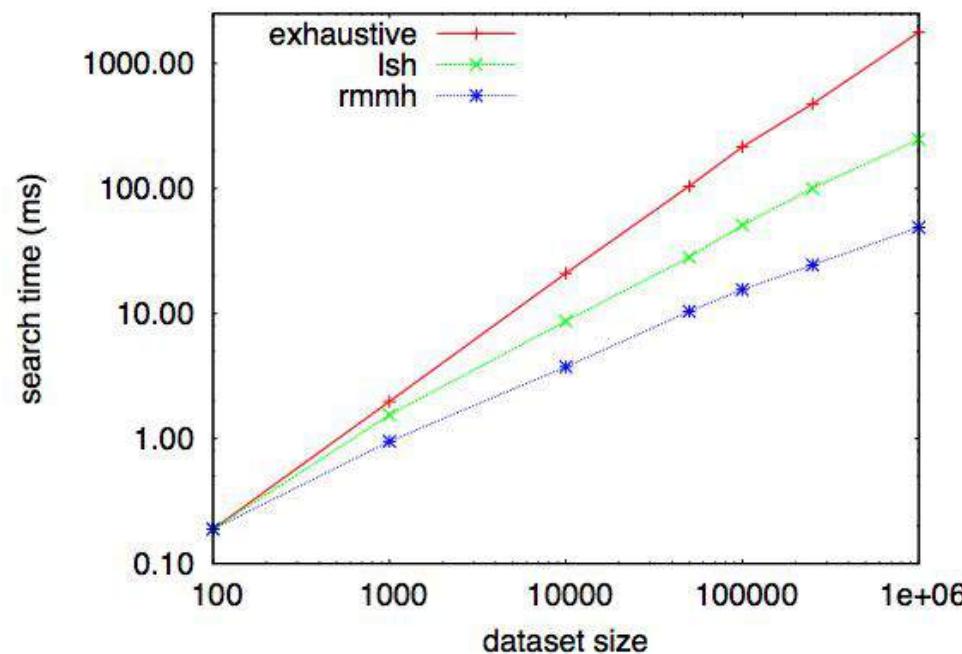
(c) Number of non empty buckets

# Random Maximum Margin Hashing

## Evaluation on ImageNet

1.2M images described by BovW features (dim 1000)

Supervised classification across 1000 semantic categories, 1000-NN



# Présentation de deux librairies: FLANN et VLfeat

## VL-Feat

VL-Feat est une librairie généraliste de vision par ordinateur avec un focus particulier sur la **recherche d'information par le contenu visuel**

VL-Feat contient un **grand nombre d'algorithmes** pour

1. l'extraction de descripteurs visuels (SIFT, Fisher vector, VLAD, MSER)
2. l'indexation et la recherche approximative de k-PP
3. L'apprentissage et la classification (SVM, etc.)

VL-Feat a été développé par un consortium d'universitaires internationaux et est sous licence open source BSD

VL-Feat est écrit en C++ mais intègre aussi des API pour C, MATLAB et Octave

URL: <http://www.vlfeat.org/>

# VL-Feat:algorithmes

Local feature frames	Covariant feature detectors
HOG features	SIFT detector and descriptor
Dense SIFT	LIOP local descriptor
MSER feature detector	Distance transform
Fisher Vector and VLAD	Gaussian Mixture Models
K-means clustering	Agglomerative Infromation Bottleneck
Quick shift superpixels	SLIC superpixels
Support Vector Machines (SVMs)	KD-trees and forests
Plotting AP and ROC curves	Miscellaneous utilities
Integer K-means	Hierarchical Integer k-means

## VL-Feat: kd-tree

VL-Feat intègre un module de recherche de k plus proches voisins dans des kd-tree classiques

Le search se fait avec un algorithme **best-bin-first**

Exemple de construction d'un arbre 2D avec l'API matlab:

```
x = rand(2, 100) ;  
  
kdtree = vl_kdtreebuild(x) ;
```

Exemple de recherches de PPV d'une requête Q (toujours en 2D):

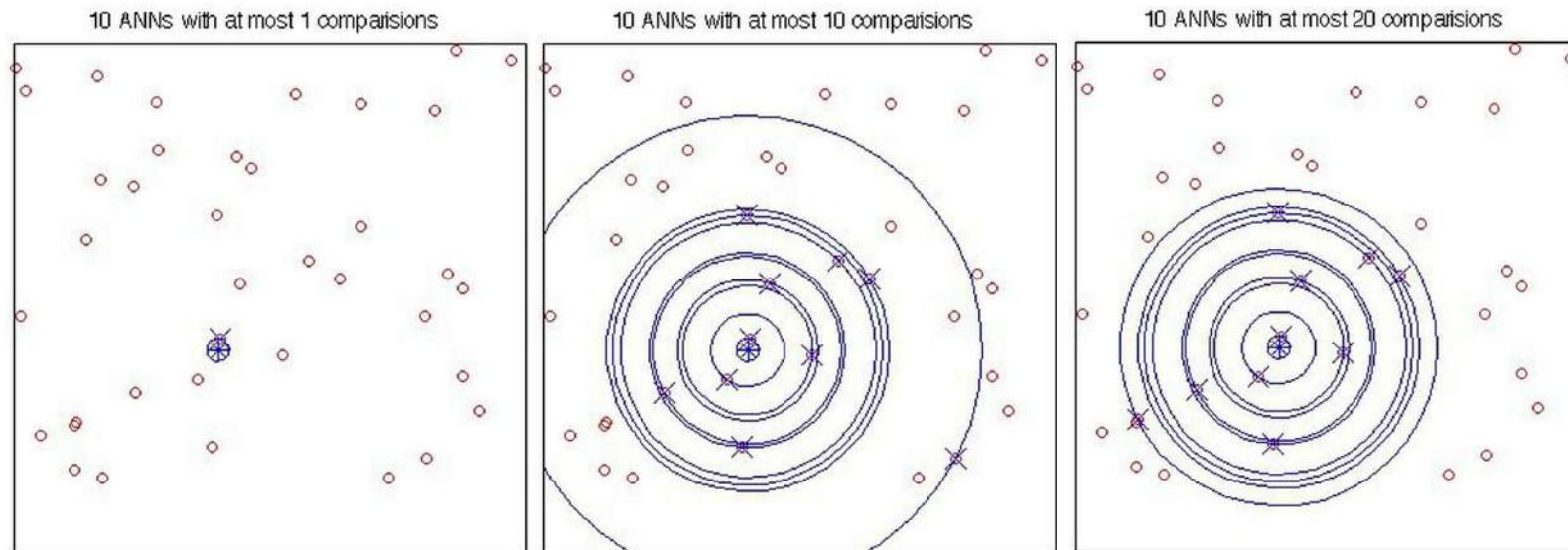
```
Q = rand(2, 1) ;  
[index, distance] = vl_kdtreequery(kdforest, x, Q) ;  
  
[index, distance] = vl_kdtreequery(kdtree, x, Q, 'NumNeighbors', 10) ;
```

« index » contient l'identifiant des plus proches voisins, « distance » les distances associées

# VL-Feat: randomized kd-tree

Exemple de recherche **approximative** de PPV d'une requête Q (toujours en 2D):

```
Q = rand(2, 1) ;  
[index, distance] = vl_kdtreequery(kdtree, X, Q, 'NumNeighbors', 10, 'MaxComparisons', 15) ;
```



# VL-Feat: randomized kd-trees forest

VL-Feat intègre un module de recherche de k plus proches voisins dans **une forêt de kd-tree aléatoires** (avec la méthode des axes aléatoires).

Exemple de construction et de recherche dans une forêt d'arbres 2D avec l'API matlab:

```
kdtree = vl_kdtreebuild(X, 'NumTrees', 4) ;
[index, distance] = vl_kdtreequery(kdtree, X, Q) ;
```

Quelques méthodes de l'API C++:

`VIKDForest * vl_kdforest_new (vl_type dataType, vl_size dimension, vl_size numTrees, VIVectorComparisonType normType)`  
Create new KDForest object. [More...](#)

`void vl_kdforest_build (VIKDForest *self, vl_size numData, void const *data)`  
Build KDTree from data. [More...](#)

`vl_size vl_kdforest_query (VIKDForest *self, VIKDForestNeighbor *neighbors, vl_size numNeighbors, void const *query)`  
Query the forest. [More...](#)

`void vl_kdforest_set_max_num_comparisons (VIKDForest *self, vl_size n)`  
Set the maximum number of comparisons for a search. [More...](#)



# VL-Feat: k-means

VL-Feat intègre également des algorithmes de clustering en particulier de nombreuses variantes de **kmeans** hierarchical k-means

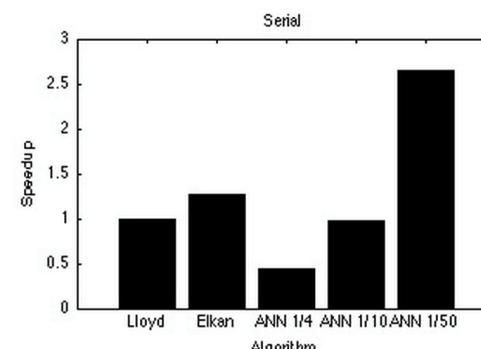
Caculer k-means avec matlab (avec l'algorithme de Loyd par défaut):

```
numData = 5000 ;
dimension = 2 ;
data = rand(dimension,numData) ;  
  
numClusters = 30 ;
[centers, assignments] = vl_kmeans(data, numClusters);
```

“centers” contient les centres de clusters (30 vecteurs), “assignments” les affectations de chaque point à chaque cluster

Versions optimisées: by setting the '**Algorithm**' parameter to '**Lloyd**', '**Elkan**' or '**ANN**'

→ La version ‘ANN’ utilise une forêt de kd-tree randomisés pour affecter les points au cluster les plus proches



# FLANN

# FLANN



FLANN est une librairie de **recherche approximative de k plus proches voisins** pour les espaces de grande dimensionnalité

FLANN contient un **ensemble d'algorithmes** pour la recherche de k-PP ainsi qu'un système permettant de **choisir automatiquement l'algorithme optimal et les paramètres optimaux pour un ensemble de données particulier**

FLANN a été développé par l'université de British Columbia et est sous licence open source BSD

FLANN est écrit en C++ mais intègre aussi des API pour C, MATLAB et Python

URL: <http://www.cs.ubc.ca/research/flann/>

# FLANN

Les 4 principaux algorithmes

1. Exhaustive Search
2. Multi-probe Locality Sensitive Hashing
3. Randomized kd-trees
4. Priority search k-means tree

# FLANN

Les 4 principaux algorithmes

1. Exhaustive Search
2. Multi-probe Locality Sensitive Hashing
3. Randomized kd-trees
4. **Priority search k-means tree**

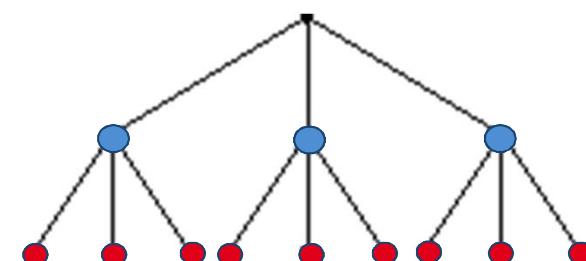
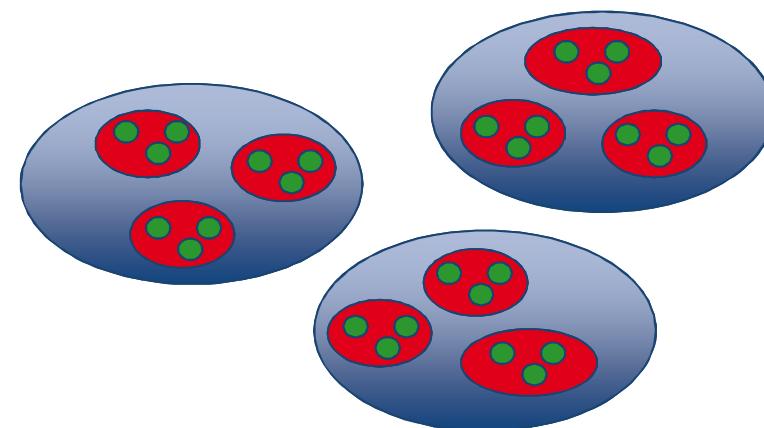
# Priority Search K-means tree

Exploite plus efficacement la structure naturelle des données en les regroupant par proximité selon toutes les composantes (contrairement au kd-tree qui partitionne selon une composante)

Construction d'un k-means tree

```
Function hk_means(X,k) {  
    if (size(X)<k) leaf node = X;  
    else [X1, ...,Xk]=K-means(X,k);  
    For m=1 à k {  
        create node Xm;  
        hk_means(Xm,k)  
    }  
}
```

→ Complexity  $O(n.k.\log(n))$



# Illustration de l'impact de k (branching factor)

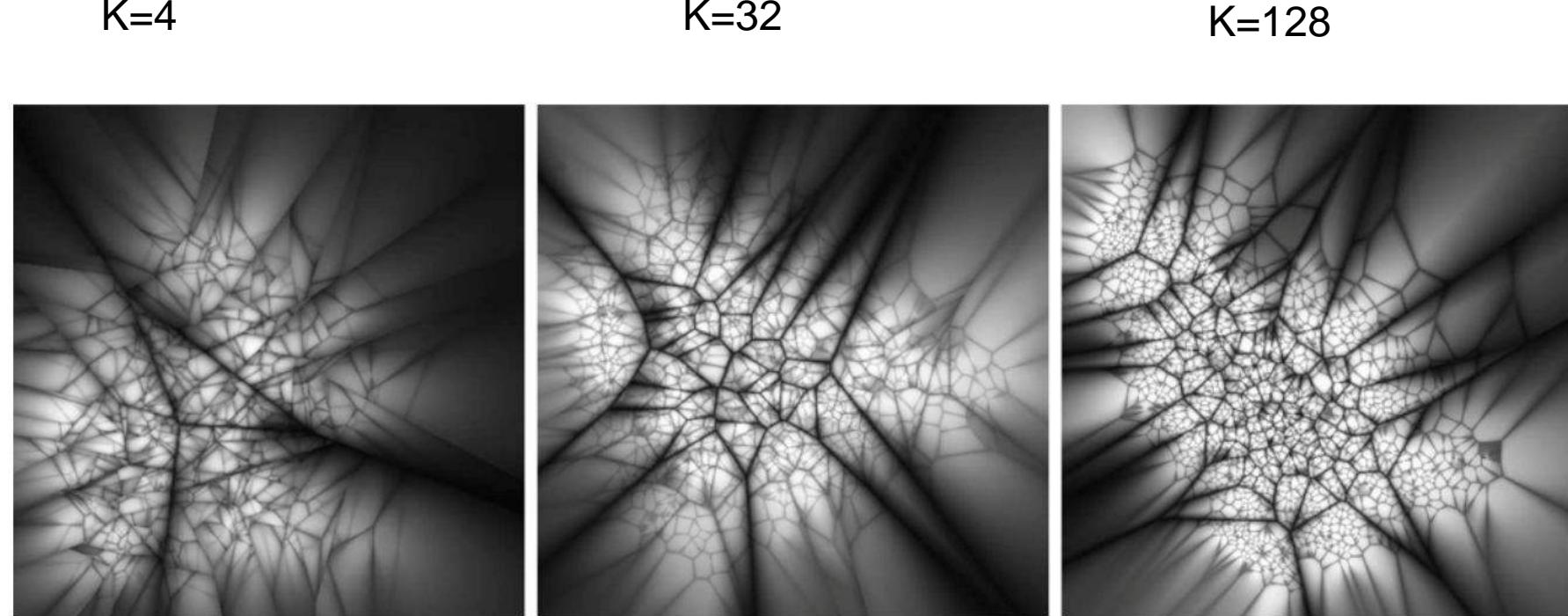
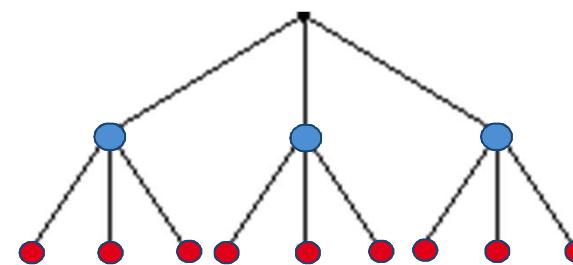
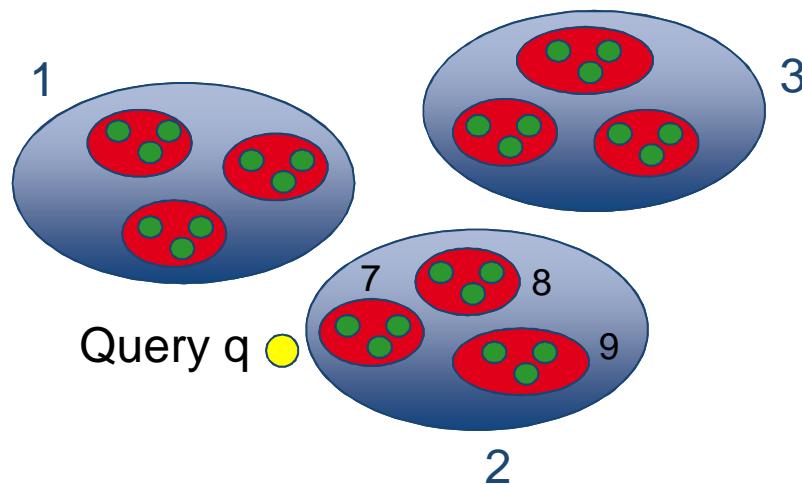


Fig. 3. Projections of priority search k-means trees constructed using different branching factors: 4, 32, 128. The projections are constructed using the same technique as in [26], gray values indicating the ratio between the distances to the nearest and the second-nearest cluster centre at each tree level, so that the darkest values (ratio  $\approx 1$ ) fall near the boundaries between k-means regions.

# Priority Search K-means tree

On maintient les régions à parcourir dans une priority queue:



Priority Queue:

It1	{0}
It2	{2,1,3}
It3	{7,1,8,9,3}

La distance de la requête à un noeud est calculée par la distance au centre de gravité du cluster

Lorsque l'on rencontre un  $p'$  tel que  $d(p',q) > r_{NN}$  mise à jour du rayon de recherche :  $r_{NN}=d(p',q)$

# FLANN: quick start

- C++

```
// file flann_example.cpp  
  
#include <flann/flann.hpp>
```

- Python

```
from pyflann import *  
from numpy import *  
from numpy.random import *  
  
dataset = rand(10000, 128)  
testset = rand(1000, 128)  
  
flann = FLANN()  
result,dists = flann.nn(dataset,testset,5,algorithm="kmeans",  
branching=32, iterations=7, checks=16);
```

```
indices = flann.knn_index(query, indices, dists, nn, flann::SearchAlgorithm::kmeans),  
flann::save_to_file(indices,"result.hdf5","result");
```

# FLANN: LSH parameters

```
struct LshIndexParams : public IndexParams  
{  
    LshIndexParams(unsigned int table_number = 12,//nombre de tables de hachage  
                  unsigned int key_size = 20, //nombre de bits par table  
                  unsigned int multi_probe_level = 2);//nombre de switch de bits pour le multi-probe  
};
```

# FLANN: Priority Search Kmeans parameters

```
struct KMeansIndexParams : public IndexParams
{
    KMeansIndexParams( int branching = 32, //nombre de clusters par noeud
                      int iterations = 11, //nombre d'itérations pour chaque appal de k-means
                      flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,//type
                      d'initialisation
                      float cb_index = 0.2 );
};
```

# FLANN: search parameters

```
struct SearchParams  
{  
  
    int checks;//nombre de feuilles max à visiter (“early stopping”)  
    float eps;//valeur de epsilon pour recherche à (1+epsilon) près  
    bool sorted;//tri des voisins ou non lorsqu'on fait une requête à un rayon près  
    int max_neighbors;//nombre max de voisins pour requête à un rayon près  
    tri_type use_heap;//utilisation ou non d'un max heap pour la recherche de knn  
    int cores;//nombre de coeurs affectés à la recherche  
};
```

# FLANN: Sélection automatique du meilleur algorithme

Les 4 principaux algorithmes: Exhaustive Search, Multi-probe Locality Sensitive Hashing, Randomized kd-trees, Priority search k-means tree

Problème:

Le choix de l'algorithme optimal dépend des données: dimensionnalité, taille et structure

Chaque algorithme a lui même des paramètres ayant une influence forte sur les performances

Espace de paramètres de grande dimensionnalité → impossible de tester toutes les combinaisons

# FLANN: Sélection automatique du meilleur algorithme

Formalisation du problème:

1. Un ensemble de paramètres à optimiser:

$$\theta = (\theta_1, \theta_2, \dots, \theta_m)$$

2. Une fonction de coût:

$$c(\theta) = \frac{s(\theta) + w_b b(\theta)}{\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))} + w_m m(\theta)$$

Temps de recherche      Temps de construction      Coût mémoire : taille index / taille données

3. Problème:  $\min_{\theta \in \Theta} c(\theta)$

# FLANN: Sélection automatique du meilleur algorithme

L'estimation de  $c(\theta)$  pour une valeur de  $\theta$  donnée se fait de manière empirique en construisant l'index et en recherchant un certain nombre d'échantillons (proportionnel au nombre de points dans la base).

Si  $m=5$  paramètres  $\theta = (\theta_1, \theta_2, \dots, \theta_5)$  et qu'on veux tester 20 valeurs par paramètre:

nb d'estimation empiriques =  $20^5 = 3,200,000$

Soit 3,200,000 construction d'index  $O(n \log n)$  et  $n$  fois 3,200,000 recherches  $O(\log n)$  ou  $O(n)$

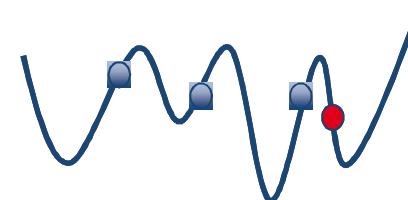
→ Impossible, des siècles de calcul

# FLANN: Sélection automatique du meilleur algorithme

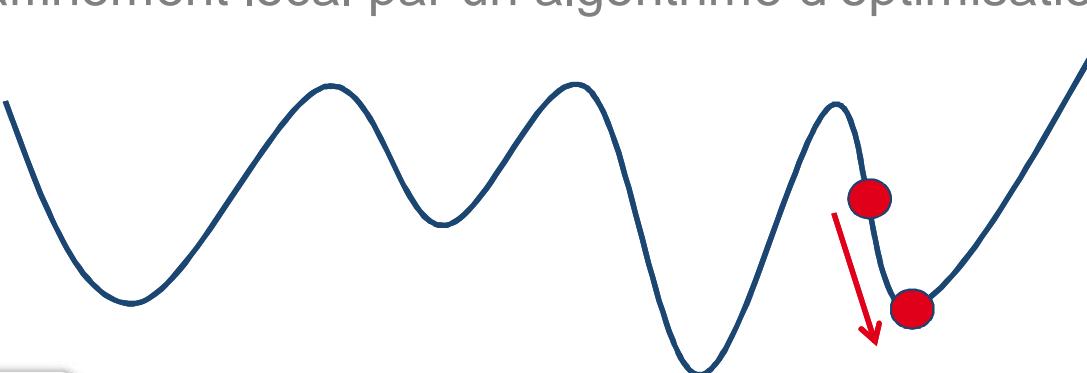
Solution FLANN = estimation globale grossière + raffinement local par

Estimation globale grossière, par exemple m=2 paramètres (dont le type d'algorithme et le param principal de chaque algo) + 4 valeurs par paramètres:

$$\text{nb d'estimation empiriques} = 4^2 = 16$$



Raffinement local par un algorithme d'optimisation (de type simplex)



# FLANN: Sélection automatique du meilleur algorithme

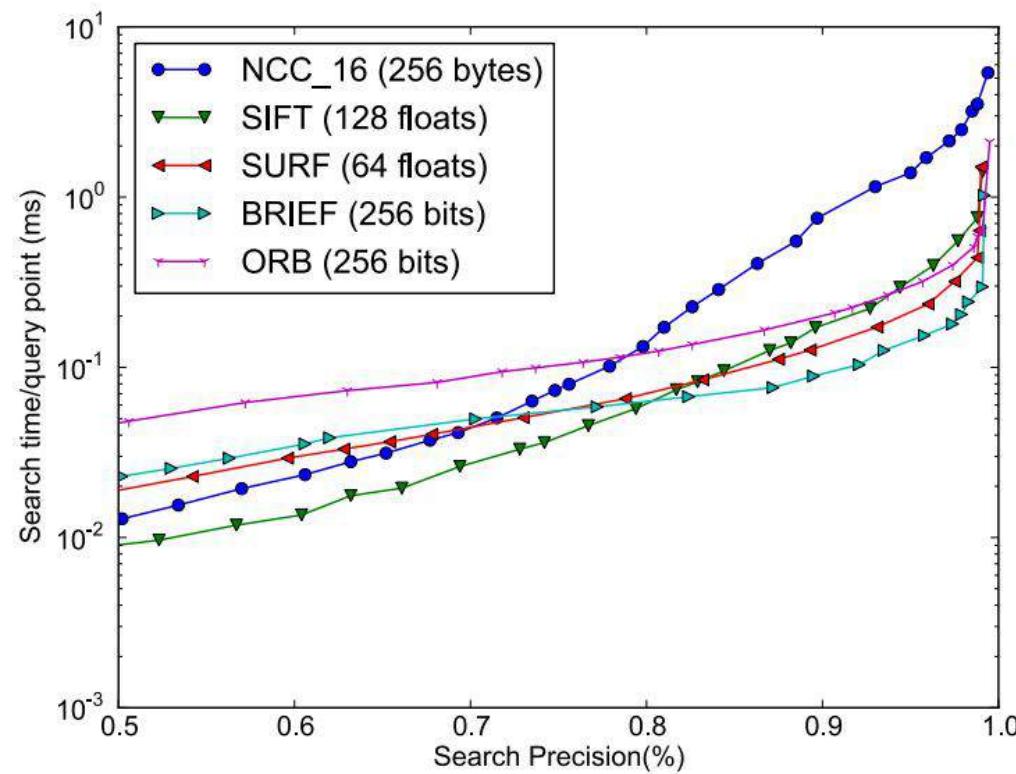
```
struct AutotunedIndexParams : public IndexParams
{
    AutotunedIndexParams( float target_precision = 0.9, //précision par rapport aux NN exacts

        float build_weight = 0.01, //pondération du temps de construction de l'index dans la fonction de coût

        float memory_weight = 0, //pondération de l'utilisation mémoire de l'index dans la fonction de coût

        float sample_fraction = 0.1 ); //fraction de la base à utiliser pour l'estimation de la fonction de coût (nombre de requêtes)
};
```

# FLANN: temps de recherche brutes en fonction de la qualité

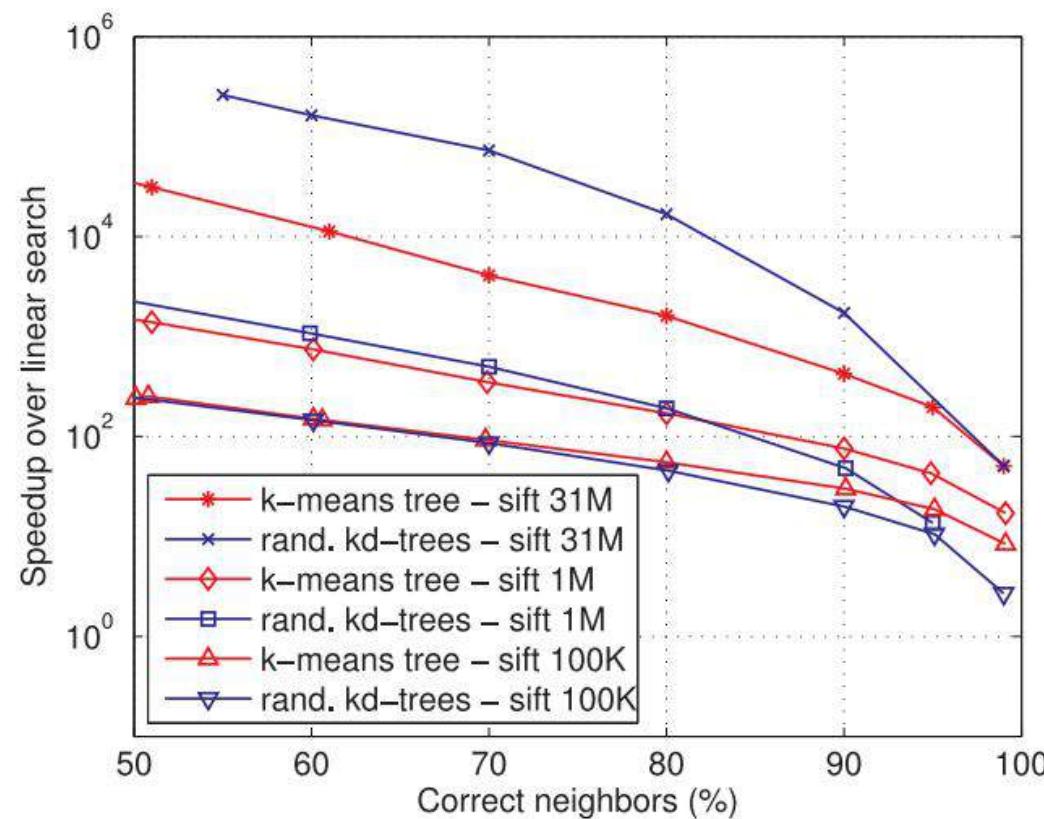


# FLANN: illustration sélection auto

Algorithmes choisis par la sélection automatique de FLANN

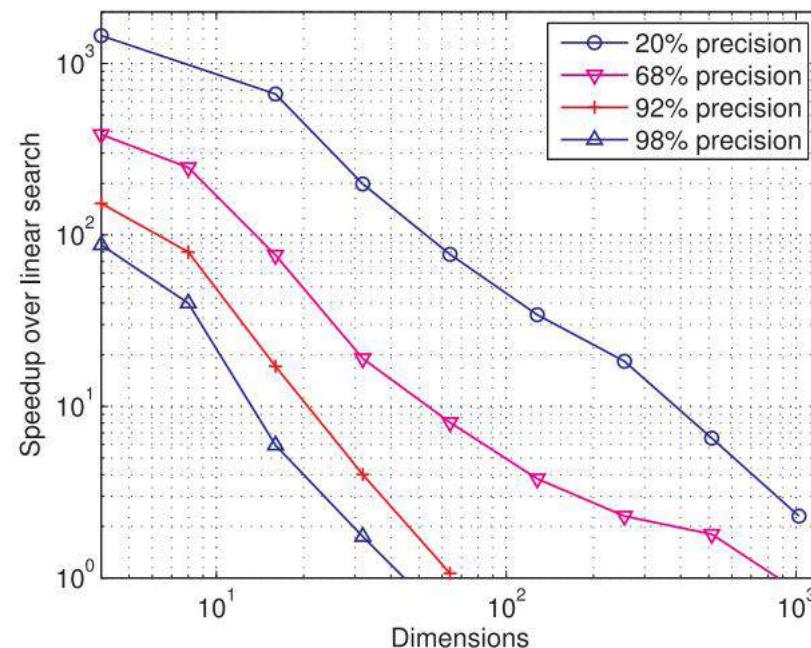
	Importance temps construction	Importance utilisation mémoire		Algorithm Configuration	Dist. Error	Search Speedup	Memory Used	Build Time
Pr. (%)	$w_b$	$w_m$						
60%	0	0		k-means, 16, 15	0.096	181.10	0.51	0.58
	0	1		k-means, 32, 10	0.058	180.9	0.37	0.56
	0.01	0		k-means, 16, 5	0.077	163.25	0.50	0.26
	0.01	1		kd-tree, 4	0.041	109.50	0.26	0.12
	1	0		kd-tree,1	0.044	56.87	0.07	0.03
	*	$\infty$		kd-tree,1	0.044	56.87	0.07	0.03
90%	0	0		k-means, 128, 10	0.008	31.67	0.18	1.82
	0	1		k-means, 128, 15	0.007	30.53	0.18	2.32
	0.01	0		k-means, 32, 5	0.011	29.47	0.36	0.35
	1	0		k-means, 16, 1	0.016	21.59	0.48	0.10
	1	1		kd-tree,1	0.005	5.05	0.07	0.03
	*	$\infty$		kd-tree,1	0.005	5.05	0.07	0.03

## FLANN: performances vs. taille de la base

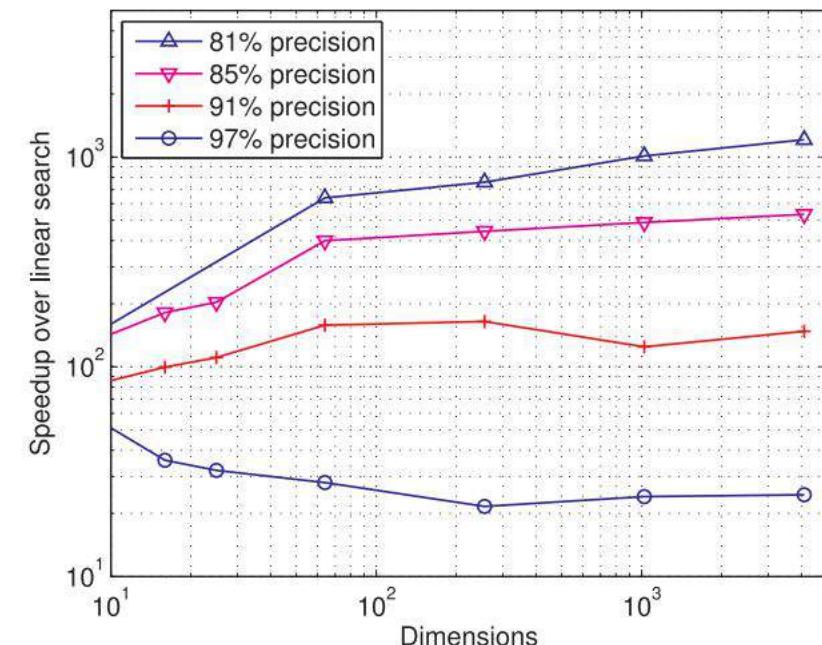


# FLANN: performances vs. dimensionnalité

Données synthétiques uniformes



Données réelles



# FLANN: distributed search

FLANN permet de traiter les requêtes en parallèle sur plusieurs machines  
En partitionnant aléatoirement et équitablement les données entre les machines

---

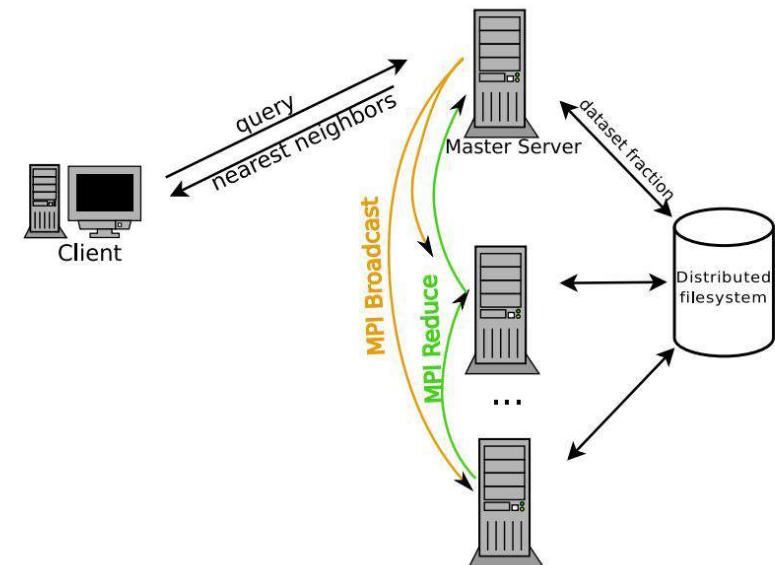
**Algorithm 5** Searching a distributed index on a computer cluster

---

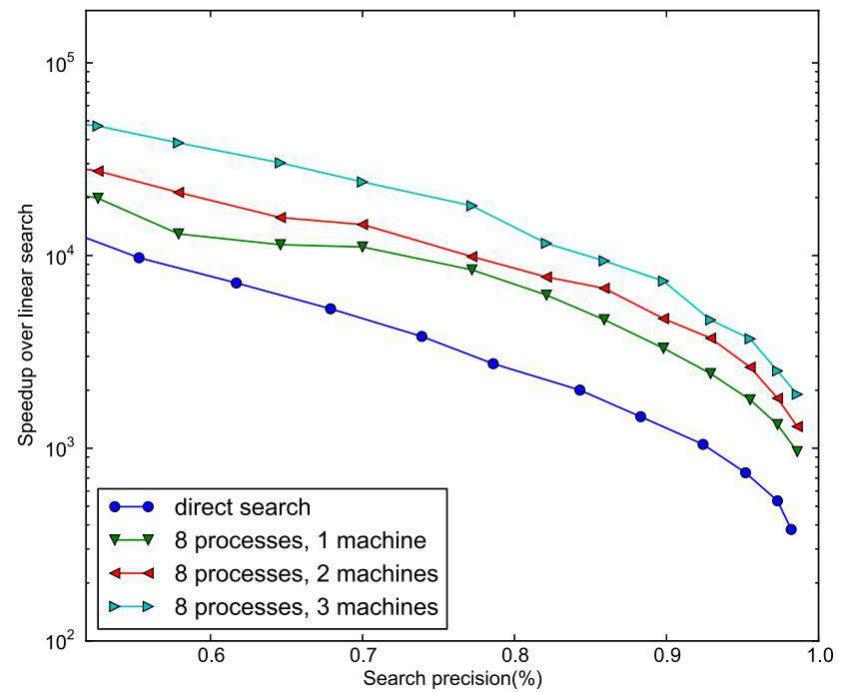
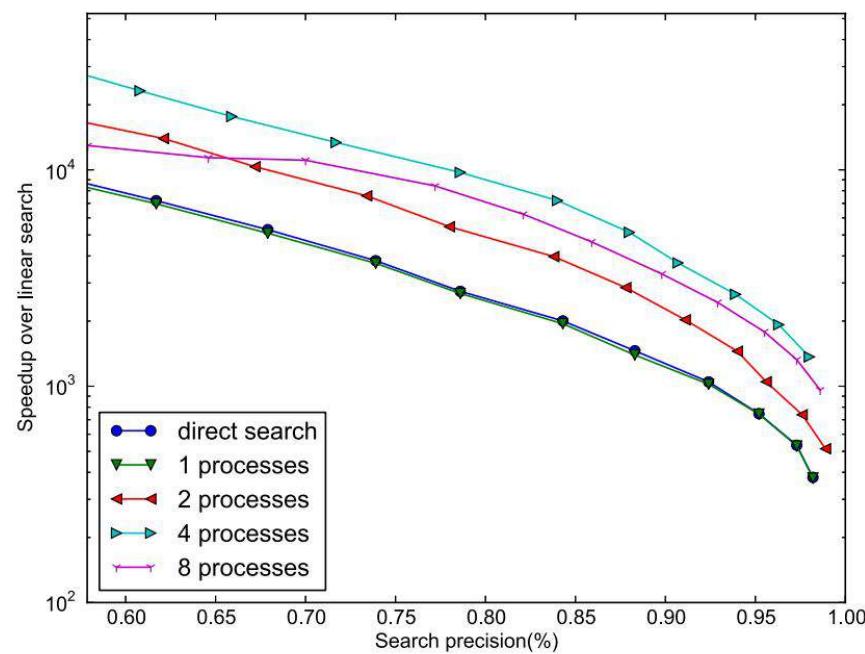
**Input:** query  $Q$ , search parameters  $P$

**procedure** SEARCHINDEX( $Q, P$ )

- 1: MPI\_broadcast( $Q, P$ ) // broadcast query and parameter to all processes
  - 2:  $NN_i \leftarrow$  run nearest neighbor search with query  $Q$  and parameters  $P$  on each process  $i$
  - 3:  $NN \leftarrow$  MPI\_reduce( $NN_i$ ) // merge results using a MPI reduce operation
  - 4: **return**  $NN$
- 



# FLANN: distributed search



# Pour aller plus loin avec FLANN

FLANN - Fast Library for Approximate Nearest  
Neighbors

User Manual

Marius Muja, mariusm@cs.ubc.ca  
David Lowe, lowe@cs.ubc.ca

January 24, 2013

[http://www.cs.ubc.ca/research/flann/uploads/FLANN/flann\\_manual-1.8.4.pdf](http://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_manual-1.8.4.pdf)

# KNN-graph approximation

# Graphes des plus proches voisins

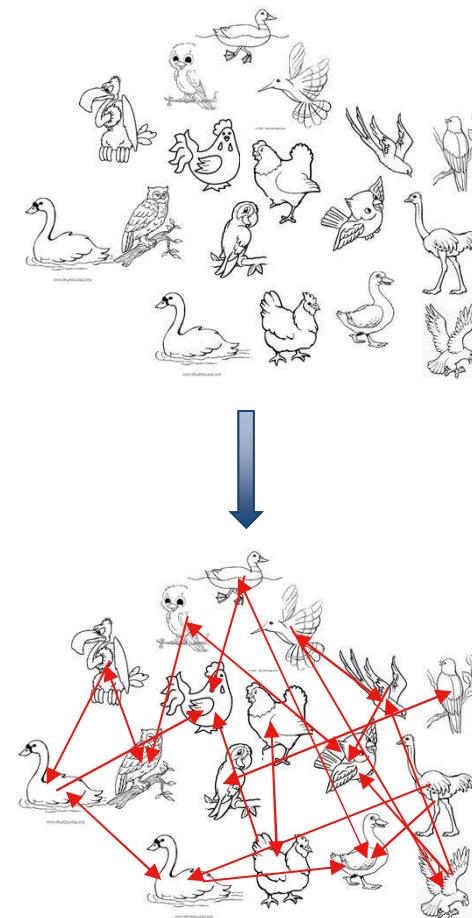
Structure clé ... pour diverses applications

Notamment:

- Suggestion, complétion dans les moteurs de recherche
- Filtrage collaboratif
- Analyse media sociaux

Impliquant:

- Des objets complexe:
  - Grandes dimensions
  - Mesures de similarités complexes
- Bases de données à échelle mondiale



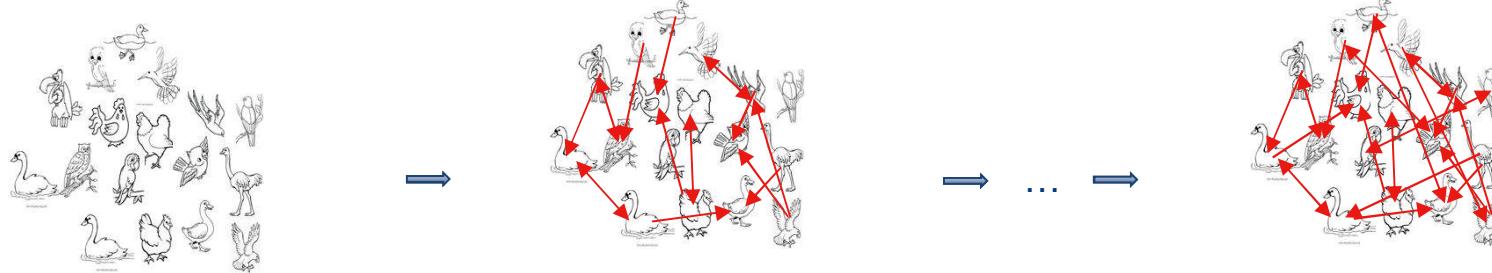
# État de l'art

Approches itératives (brute force, processor les nœuds de manière **itérative**):

- Exacte: recherche **exhaustive** des plus proches voisins  $O(n^2)$

Approches récursives:

- NN-Descent\* (méthode de référence complexité empirique  $O(n^{1.14})$ )

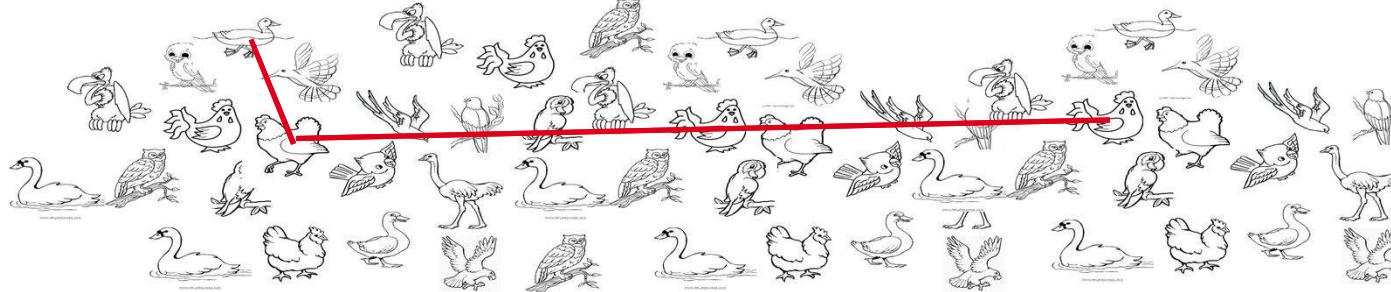


- Approche par jointure dans des tables de hachage sensible à la localité, performances similaires mais plus facilement distribuables

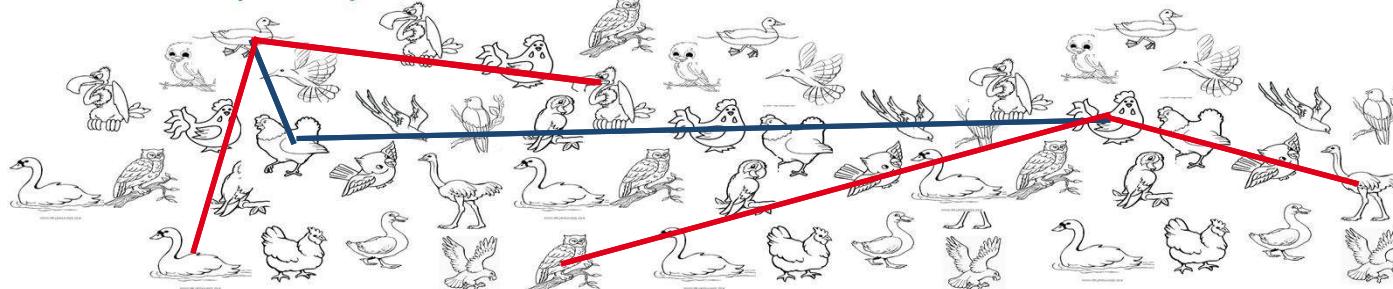
	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$	$T_7$
00			a,b,c,d	c	a,b,c,d	b,d	
01				a,b			
10		a,b,d		d		a,c	a,c
11	a,b,c,d	c					b,d

# NN descent: principe

1. Tirage aléatoire de k voisins pour chaque objet (ex: k=2)



2. Pour chaque objet, calcul de la distance aux voisins et aux voisins des voisins



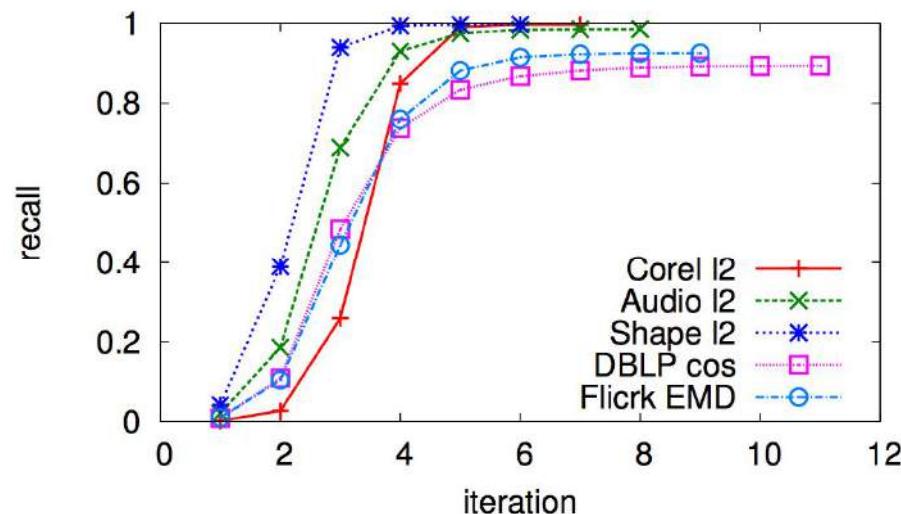
3. Mise à jour des k-pp voisins de chaque objet



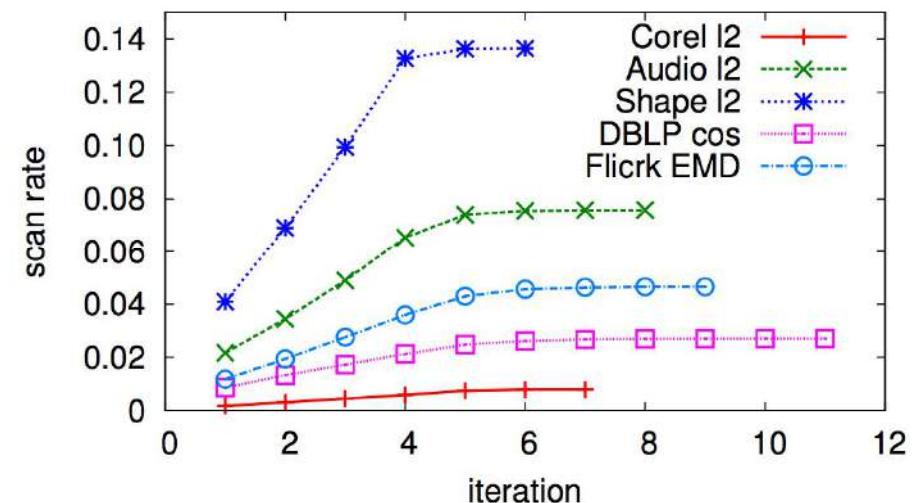
1. Répéter m fois

# NN descent: performances

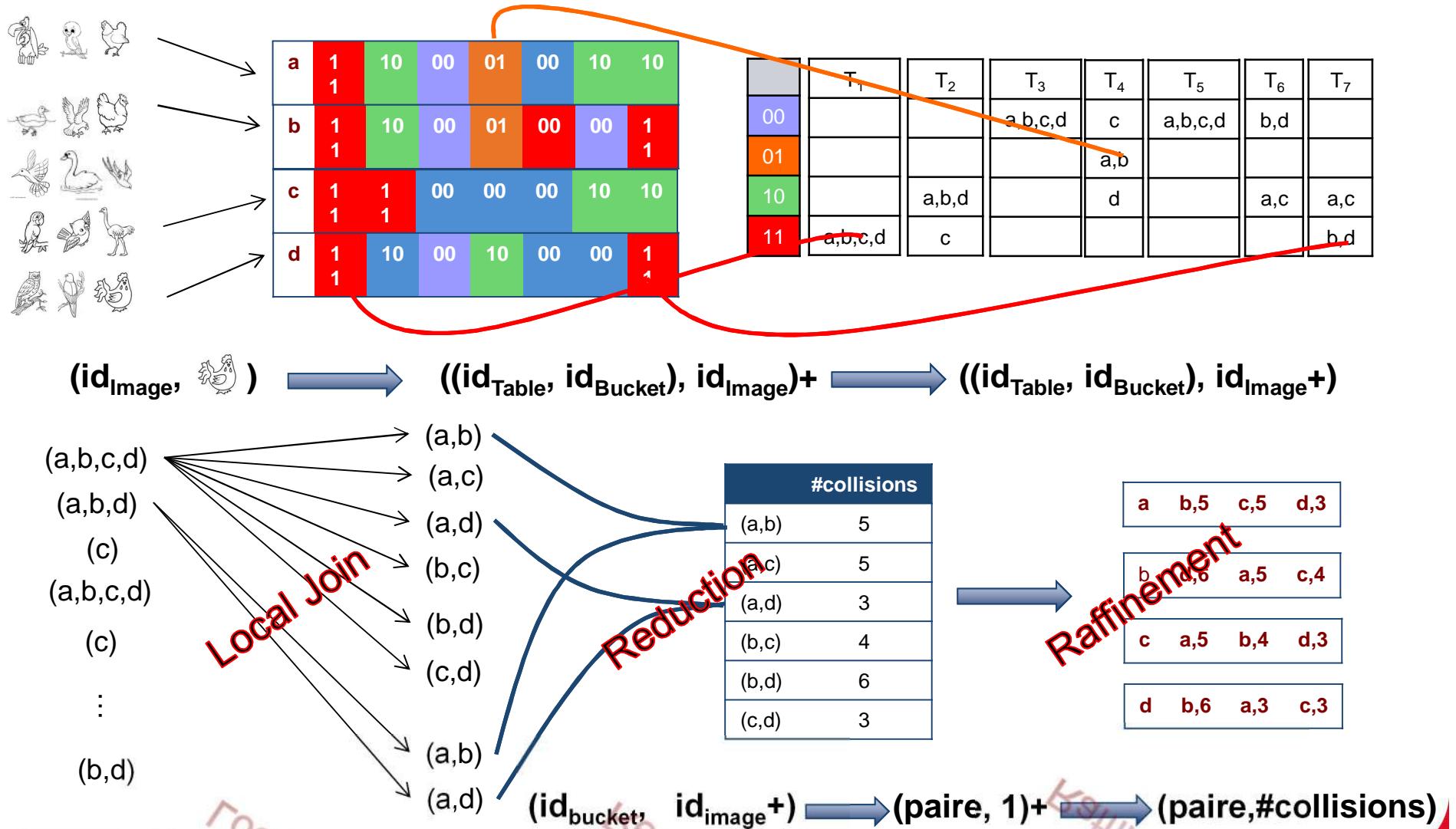
Pourcentage des k-NN exactes retrouvés en fonction du nombre d'itération



Fraction de la base parcourue en fonction du nombre d'itération



# Approche par jointure de tables de hachage sensible à la localité

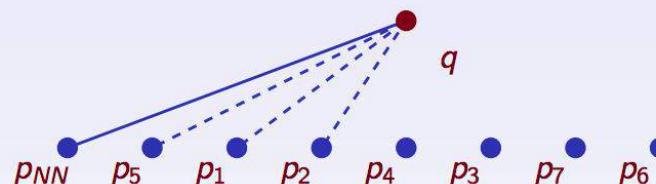


# Recherche de voisins dans les graphes de similarité (notamment knn-graph)

Exemple: Algorithme d'Orchard (très utilisé dans réseaux P2P)

## Preprocessing:

Pour tous les objets  $p_i \in S$  construire une liste  $L(p_i)$  de tous les autres objets triés par leur similarité à  $p_i$ :



## Orchard'91

## Traitement requête:

- Commencer avec un objet aléatoire  $p_{NN}$
- Inspecter les membres  $p' \in L(p_{NN})$  de gauche à droite
- Si  $d(p', q) < d(p_{NN}, q)$ , update  $p_{NN} := p'$
- Condition d'arrêt: On trouve un  $p'$  tel que  $d(p', q) \geq 2d(p_{NN}, q)$

